Ethereum Alarm Clock Documentation

Release 1.0.0

Piper Merriam

Apr 12, 2018

Contents

The Ethereum Alarm Clock (EAC for short) is a collection of smart contracts on Ethereum that aims to allow for the scheduling of transactions to be executed at a later time. It is a "ĐApp" in the truest sense of the word since it has the qualities of being trustless, censorship resistant and decentralized. No matter who you are or where you are located, there is no way for anyone to stop you from scheduling a transaction or running an execution client. There is no priviledged access given to any party, not even to the developers =)

On a high level it works by some users scheduling transactions and providing the execution details and payments up front, while other users run execution clients that scan the blockchain looking for upcoming transactions. When the execution clients find upcoming transactions they keep track of them and compete to claim and execute them – whoever gets the rights to the execution gets paid the exeuction reward.

The code for this service is open source under the MIT license and can be viewed on the github repository. Each release of the alarm service includes details on verifying the contract source code.

For a more complete explanation of what this service does check out the *Introduction*.

If you are a smart contract developer and would like to see an example for how you can use the EAC smart contracts from your own code see *Quickstart*.

If you are looking to acquire a deeper understanding of the architecture then it is recommnended you skim the documentation in full. It is recommnended to also view the source code.

Contents:

CHAPTER 1

Introduction

1.1 What problem does this solve?

The simplest way to explain the utility of the EAC is to explain the problem it solves.

We will begin with a refresher about the two types of accounts on Ethereum and the differences between them. There exists:

1. User accounts (controlled by the holder of the private key)
2. Contracts *(which are not controlled by a private key)*

User accounts are the accounts that humans control and operate. The controller of a user account is always the person who holds the private key. In contrast, contract accounts are not controlled by a private key but are instead deployed code which execute in a determined way when they are called.

All code execution in the Ethreum Virtual Machine (the EVM) must be triggered by a private key based account. This is done by sending a transaction, which may do something simple like transfering ether, or it may do something more complex like calling a function on a contract account.

Whenever a user account initiates a contract account, the execution of the contract is immediate. Therefore all calls to contract accounts are included in the same block as the initial call.

The Ethereum protocol does not provide any way to create a transaction to be executed at a later time. So, if a developer is creating an application that needs to fire off transactions that must happen at a future date or if a user would like to perform an action at a specific time without being present, there is no inherent way to do this on Ethereum.

The EAC service aims to solve these issues while also creating a decentralized incentive based protocol that ensures pretty good guarantees that someone will execute all scheduled transactions.

1.2 How transactions are executed

When a user schedules a new transaction, they are deploying a new smart contract that holds all of the information necessary for the execution of the transaction. A good analogy to compare this smart contract to is an order on an exchange. When this contract "order" is called during the specified execution window, the contract will send the transaction as set by the user. It will also pay the account that triggered the execution and if a fee was specified in the data, a transaction to the fee recipient.

These contracts are of the type called `TransactionRequest` and are written to provide strong guarantees of correctness to both parties.

The creator of the `TransactionRequest` contract can know that their transaction will only be sent during the window they specified and that the transaction parameters will be sent exactly as specified.

Similarly, the account that executes the `TransactionRequest` contract can know that no matter what occurs during the execution of the transaction (including if the transaction fails) that they will receive full gas reimbursement as well as their payment for execution.

1.3 Execution guarantees

You may have noted at this point that this service relies on external parties to initiate the execution of these transactions. This means that it is possible that your transaction will not be executed at all. Indeed, if no one is running an execution client then your transaction will not be executed and will expire. However, incentives have been baked into the system to encourage the running of execution clients and the hope is that many parties will compete for the rights to execute transactions.

In an ideal situation, there is a sufficient volume of scheduled transactions that operating a server to execute these transactions is a profitable endeavor.

1.4 How scheduling transactions works

A transaction is scheduled by providing some or all of the following information.

- Details about the transaction itself such as which address the transaction should be sent to, or how much ether should be sent with the transaction.
- Details about when the transaction can be executed. This includes things like the window of time or blocks during which this transaction can be executed.
- Ether to pay for the transaction gas costs as well as the payment that will be paid to the account that triggers the transaction.

Scheduling is done by calling a `Scheduler` contract which handles creation of the individual `TransactionRequest` contract.

CHAPTER 2

Quickstart

2.1 Introduction

This guide is inteded for smart contract developers that may want to use the EAC services from within their own applications. Since all the functionality of the Alarm Clock is built into the Ethereum chain via smart contracts, it can be accessed from other contracts. This makes it useful as a foundational tool to design and implement more complex utilities that depend on future transactions. For this guide we will be using the Solidity language. If you are unfamiliar with Solidity we recommend you familiarize yourself with its documentation first.

2.2 Scheduling your first transaction

The first step is to establish how we will interact with the EAC service's `Scheduler` contract. We can use the Scheduler Interface to accomplish this. The Scheduler interface contract contains some logic that is shared between both the Block Scheduler and the Timestamp Scheduler. The function that we are interested in is the *schedule()* function. See the signature of this function below:

```
function schedule(address   _toAddress,
                  bytes     _callData,
                  uint[7]   _uintArgs)
    doReset
    public payable returns (address);
```

`SchedulerInterface.sol` is an abstract contract that exposes the API for the Schedulers including the `schedule()` function that we will use in the contract we write.

function `schedule` which will return the address of the newly created *TransactionRequest* contract. We will import this contract into the contract we write so to ensure that our contract will be compliant with the official API of the EAC.

Now lets write a simple contract that can use the scheduling service.

```
pragma solidity 0.4.19;

import 'contracts/Interface/SchedulerInterface.sol';

/// Example of using the Scheduler from a smart contract to delay a payment.
contract DelayedPayment {

    SchedulerInterface public scheduler;

    uint lockedUntil;
    address recipient;

    function DelayedPayment(
        address _scheduler,
        uint    _numBlocks,
        address _recipient
    ) {
        scheduler = SchedulerInterface(_scheduler);
        lockedUntil = block.number + _numBlocks;
        recipient = _recipient;

        scheduler.schedule.value(2 ether)(
            recipient,              // toAddress
            "",                     // callData
            [
                2000000,            // The amount of gas to be sent with the␣
↪transaction.
                0,                  // The amount of wei to be sent.
                255,                // The size of the execution window.
                lockedUntil,        // The start of the execution window.
                30000000000 wei,    // The gasprice for the transaction (aka 30 gwei)
                12345 wei,          // The fee included in the transaction.
                224455 wei,         // The bounty that awards the executor of the␣
↪transaction.
                20000 wei           // The required amount of wei the claimer must␣
↪send as deposit.
            ]
        );
    }

    function () {
        if (this.balance > 0) {
            payout();
        } else {
            revert();
        }
    }

    function payout()
        public returns (bool)
    {
        require(getNow() >= lockedUntil);
        recipient.transfer(this.balance);
        return true;
    }

    function getNow()
        internal view returns (uint)
    {
        return block.number;
    }

}
```

The contract above is designed to lock away whatever ether it is given for `numBlocks` blocks. In its constructor, it makes a call to the `schedule` method on the `scheduler` contract. We would pass in the address of the scheduler we would want to interact with as the first parameter of the constructor. For instance, if we wanted to use the Block Scheduler that is deployed on the Ropsten test net we would use address `0x96363aea9265913afb8a833ba6185a62f089bcef`.

The function only takes 3 parameters at deployment, with the 7 parameters being into the schedule function being set by the developer of the smart contract. The schedule function takes 7 arguments, each of which we will go over in order.

- `address toAddress`: The `address` which the transaction will be sent to.
- `bytes callData`: The `bytes` that will be used as the data for the transaction.
- `uint callGas`: The amount of gas that will be sent with the transaction.
- `uint callValue`: The amount of ether (in wei) that will be sent with the transaction.
- `uint8 windowSize`: The number of blocks after `windowSize` during which the transaction will still be executable.
- `uint windowStart`: The first block number that the transaction will be executable.
- `uint gasPrice`: The gas price which must be sent by the executing party to execute the transaction.
- `uint fee`: The fee amount included in the transaction.
- `uint payment`: The payment included in the transaction.

Let's look at the other function on this contract. For those unfamiliar with solidity, the function without a name is known as the fallback function. The fallback function is what triggers if no method is found on the contract that matches the sent data, or if data is not included. Usually, sending a simple value transfer function will trigger the fallback function. In this case, we explicitly pass an empty string as the `callData` variable so that the scheduled transaction will trigger this function when it is executed.

When the fallback function is executed, if the contract has a balance (which it does since we sent it 2 ether on deployment) it will route the call into the `payout()` function. The `payout()` function will check the current block number and check if it is not below the `lockedUntil` time or else it reverts the transaction. After it checks that the current block number is greater than or equal to the lockedUntil variable, the function will transfer the entrie balance of the contract to the specified recipient.

As can be seen, this will make it so that a payment is scheduled for a future date but won't actually be sent until that date. This example uses a simple payment, but the EAC will work with arbitrary logic. As the logic for a scheduled transaction increases, be sure to increase the required `callGas` in accordance. Right now, the gas limit for a transaction is somewhere in the ballpark of 8,000,000 so there's plenty of wiggle room for experimentation.

CHAPTER 3

Architecture

3.1 Overview

The Ethereum Alarm Clock infrastructure consists of the following contracts:

- `TransactionRequest`: Represents a single scheduled transaction.
- `RequestFactory`: Low level API for creating `TransactionRequest` contracts.
- `RequestTracker`: Tracks the scheduled transactions.
- `BlockScheduler`: High level API for creating `TransactionRequest` contracts configured to be executed at a specified block number.
- `TimestampScheduler`: High level API for creating `TransactionRequest` contracts configured to be executed at a certain time, as specified by a timestamp.

Note: Actual functionality of most of the contracts is housed separately in various libraries.

`class RequestTracker`

3.2 RequestTracker

The `RequestTracker` is a database contract which tracks upcoming transaction requests. It exposes an API suitable for someone wishing to execute transaction requests to be able to query which requests are scheduled next as well as other common needs.

The `RequestTracker` database indexes requests based on the address that submits them. Therefore, the `RequestTracker` is un-permissioned and allows any address to report scheduled transactions and to have them stored in their own personal index. The address which submits the transaction request is referred to as the *scheduler address*.

The flexibility of the RequestTracker storage enables those executing transaction requests to choose which *scheduler addresses* they wish to watch for upcoming transactions.

`class RequestFactory`

3.3 RequestFactory

The `RequestFactory` contract is designed to be a low-level interface for developers who need fine-grained control over all of the various parameters that the `TransactionRequest` can be configured with.

Parameter validation is available, but not mandatory.

It provides an API for creating new `TransactionRequest` contracts.

`class BlockScheduler`

`class TimestampScheduler`

3.4 BlockScheduler and TimestampScheduler

The `BlockScheduler` and `TimestampScheduler` contracts are a higher-level interface that most developers should want to use in order to schedule a transaction for a future block or timestamp.

Both contracts present an identical API for creating new `TransactionRequest` contracts. Different from `RequestFactory`, request parameters are always validated.

`BlockScheduler` treats all of the scheduling parameters as meaning block numbers, while `TimestampScheduler` treats them as meaning timestamps and seconds.

CHAPTER 4

Transaction Request

class TransactionRequest

Each `TransactionRequest` contract represents one transaction that has been scheduled for future execution. This contract is not intended to be used directly as the `RequestFactory` contract can be used to create new `TransactionRequest` contracts with full control over all of the parameters.

4.1 Interface

```
pragma solidity 0.4.19;

contract TransactionRequestInterface {

    // Primary actions
    function execute() public returns (bool);
    function cancel() public returns (bool);
    function claim() public payable returns (bool);

    // Proxy function
    function proxy(address recipient, bytes callData)
        public payable returns (bool);

    // Data accessors
    function requestData() public view returns (address[6],
                                                bool[3],
                                                uint[15],
                                                uint8[1]);

    function callData() public view returns (bytes);

    // Pull mechanisms for payments.
    function refundClaimDeposit() public returns (bool);
    function sendFee() public returns (bool);
    function sendBounty() public returns (bool);
    function sendOwnerEther() public returns (bool);
}
```

4.2 Events

`TransactionRequest.`**`Cancelled`**`(`*uint rewardPayment*, *uint measuredGasConsumption*`)`

When a request is cancelled, the `Cancelled` event will be logged. The `rewardPayment` is the amount that was paid to the party that cancelled the request. This will always be 0 when the owner of the request cancels the request.

`TransactionRequest.`**`Claimed`**`()`

When a request is claimed this event is logged.

`TransactionRequest.Aborted(uint8 reason);`

When an attempt is made to execute a request but one of the pre-execution checks fails, this event is logged. The `reason` is an error code which maps to the following errors.

- `0 => WasCancelled`
- `1 => AlreadyCalled`
- `2 => BeforeCallWindow`
- `3 => AfterCallWindow`

- `4 => ReservedForClaimer`
- `5 => InsufficientGas`
- `6 => MismatchGasPrice`

`TransactionRequest.`**`Executed`**`(`*uint payment*, *uint donation*, *uint measuredGasConsumption*`)`

When a request is successfully executed this event is logged. The `payment` is the total payment amount that was awarded for execution. The `donation` is the amount that was awarded to the `donationBenefactor`. The `measuredGasConsumption` is the amount of gas that was reimbursed which should always be slightly greater than the actual gas consumption.

4.3 Data Model

The data for the transaction request is split into 5 main sections.

- **Transaction Data**: Information specific to the execution of the transaction.
- **Payment Data**: Information related to the payment and donation associated with this request.
- **Claim Data**: Information about the claim status for this request.
- **Schedule Data**: Information about when this request should be executed.
- **Meta Data**: Information about the result of the request as well as which address owns this request and which address created this request.

4.3.1 Retrieving Data

The data for a request can be retrieved using two methods.

`TransactionRequest.`**`requestData`**`()`

This function returns the serialized request data (excluding the `callData`) in a compact format spread across four arrays. The data is returned alphabetical, first by type, and then by section, then by field.

The return value of this function is four arrays.

- `address[6] addressValues`
- `bool[3] boolValues`
- `uint256[15] uintValues`
- `uint8[1] uint8Values`

These arrays then map to the following data fields on the request.

- **Addresses (`address`)**
  - `addressValues[0] => claimData.claimedBy`
  - `addressValues[1] => meta.createdBy`
  - `addressValues[2] => meta.owner`
  - `addressValues[3] => paymentData.donationBenefactor`
  - `addressValues[4] => paymentData.paymentBenefactor`
  - `addressValues[5] => txnData.toAddress`
- **Booleans (`bool`)**

    - `boolValues[0] => meta.isCancelled`
    - `boolValues[1] => meta.wasCalled`
    - `boolValues[2] => meta.wasSuccessful`
- **Unsigned 256 bit Integers (`uint` aka `uint256`)**
    - `uintValues[0] => claimData.claimDeposit`
    - `uintValues[1] => paymentData.anchorGasPrice`
    - `uintValues[2] => paymentData.donation`
    - `uintValues[3] => paymentData.donationOwed`
    - `uintValues[4] => paymentData.payment`
    - `uintValues[5] => paymentData.paymentOwed`
    - `uintValues[6] => schedule.claimWindowSize`
    - `uintValues[7] => schedule.freezePeriod`
    - `uintValues[8] => schedule.reservedWindowSize`
    - `uintValues[9] => schedule.temporalUnit`
    - `uintValues[10] => schedule.windowStart`
    - `uintValues[11] => schedule.windowSize`
    - `uintValues[12] => txnData.callGas`
    - `uintValues[13] => txnData.callValue`
    - `uintValues[14] => txnData.gasPrice`
- **Unsigned 8 bit Integers (`uint8`)**
    - `uint8Values[0] => claimData.paymentModifier`

`TransactionRequest.`**`callData`**`()`

Returns the `bytes` value of the `callData` from the request's transaction data.

4.3.2 Transaction Data

This portion of the request data deals specifically with the transaction that has been requested to be sent at a future block or time. It has the following fields.

`address toAddress`
The address that the transaction will be sent to.

`bytes callData`
The bytes that will be sent as the `data` section of the transaction.

`uint callValue`
The amount of ether, in wei, that will be sent with the transaction.

`uint callGas`
The amount of gas that will be sent with the transaction.

`uint gasPrice`
The gas price required to send when executing the transaction.

4.3.3 Payment Data

Information surrounding the payment and donation for this request.

`uint anchorGasPrice`
The gas price that was used during creation of this request. This is used to incentivise the use of an adequately low gas price during execution.

See gas-multiplier for more information on how this is used.

`uint payment`
The amount of ether in wei that will be paid to the account that executes this transaction at the scheduled time.

`address paymentBenefactor`
The address that the payment will be sent to. This is set during execution.

`uint paymentOwed`
The amount of ether in wei that is owed to the `paymentBenefactor`. In most situations this will be zero at the end of execution, however, in the event that sending the payment fails the payment amount will be stored here and retrievable via the `sendPayment()` function.

`uint donation`
The amount of ether, in wei, that will be sent to the *donationBenefactor* upon execution.

`address donationBenefactor`
The address that the donation will be sent to.

`uint donationOwed`
The amount of ether in wei that is owed to the `donationBenefactor`. In most situations this will be zero at the end of execution, however, in the event that sending the donation fails the donation amount will be stored here and retrievable via the `sendDonation()` function.

4.3.4 Claim Data

Information surrounding the claiming of this request. See *Claiming* for more information.

`address claimedBy`
The address that has claimed this request. If unclaimed this value will be set to the zero address `0x0000000000000000000000000000000000000000`

`uint claimDeposit`
The amount of ether, in wei, that has been put down as a deposit towards claiming. This amount is included in the payment that is sent during request execution.

`uint8 paymentModifier`
A number constrained between 0 and 100 (inclusive) which will be applied to the payment for this request. This value is determined based on the time or block that the request is claimed.

4.3.5 Schedule Data

Information related to the window of time during which this request is scheduled to be executed.

`uint temporalUnit`
Determines if this request is scheduled based on block numbers or timestamps.

- Set to `1` for block based scheduling.
- Set to `2` for timestamp based scheduling.

All other values are interpreted as being blocks or timestamps depending on what this value is set as.

`uint windowStart`
The block number or timestamp on which this request may first be executed.

`uint windowSize`
The number of blocks or seconds after the `windowStart` during which the request may still be executed. This period of time is referred to as the *execution window*. This period is inclusive of it's endpoints meaning that the request may be executed on the block or timestamp `windowStart + windowSize`.

`uint freezePeriod`
The number of blocks or seconds prior to the `windowStart` during which no activity may occur.

`uint reservedWindowSize`
The number of blocks or seconds during the first portion of the the *execution window* during which the request may only be executed by the address that address that claimed the call. If the call is not claimed, then this window of time is treated no differently.

`uint claimWindowSize`
The number of blocks prior to the `freezePeriod` during which the call may be claimed.

4.3.6 Meta Data

Information about ownership, creation, and the result of the transaction request.

`address owner`
The address that scheduled this transaction request.

`address createdBy`
The address that created this transaction request. This value is set by the *`RequestFactory`* meaning that if the request is *known* by the request factory then this value can be trusted to be the address that created the contract. When using either the *`BlockScheduler`* or *`TimestampScheduler`* this address will be set to the respective scheduler contract..

`bool isCancelled`
Whether or not this request has been cancelled.

`bool wasCalled`
Whether or not this request was executed.

`bool wasSuccessful`
Whether or not the execution of this request returned `true` or `false`. In most cases this can be an indicator that an execption was thrown if set to `false` but there are also certain cases due to quirks in the EVM where this value may be `true` even though the call technically failed.

4.4 Actions

The *`TransactionRequest`* contract has three primary actions that can be performed and a fourth action, *proxy*, which will be called in certain circumstances.

- Cancellation: Cancels the request.
- Claiming: Reserves exclusive execution rights during a portion of the execution window.
- Execution: Sends the requested transaction.

4.4.1 Cancellation

`TransactionRequest.`**`cancel`**`()`

Cancellation can occur if either of the two are true.

- The current block or time is before the freeze period and the request has not been claimed.
- The current block or time is after the execution window and the request was not executed.

When cancelling prior to the execution window, only the `owner` of the call may trigger cancellation.

When cancelling after the execution window, anyone may trigger cancellation. To ensure that funds are not forever left to rot in these contracts, there is an incentive layer for this function to be called by others whenever a request fails to be executed. When cancellation is executed by someone other than the `owner` of the contract, `1%` of what would have been paid to someone for execution is paid to the account that triggers cancellation.

4.4.2 Claiming

`TransactionRequest.`**`claim`**`()`

Claiming may occur during the `claimWindowSize` number of blocks or seconds prior to the freeze period. For example, if a request was configured as follows:

- `windowStart`: block #500
- `freezePeriod`: 10 blocks
- `claimWindowSize`: 100 blocks

In this case, the call would first be claimable at block 390. The last block in which it could be claimed would be block 489.

See the *Claiming* section of the documentation for details about the claiming process.

4.4.3 Execution

`TransactionRequest.`**`execute`**`()`

Execution may happen beginning at the block or timestamp denoted by the `windowStart` value all the way through and including the block or timestamp denoted by `windowStart + windowSize`.

See the *Execution* section of the documentation for details about the execution process.

4.4.4 Proxy

`TransactionRequest.`**`proxy`**`(`*address _to*, *bytes _data*`)`

Proxy can only be called by the user who scheduled the TransactionRequest and only after the execution window has passed. It exposes two fields, *_to* which will be the address of that the call will send to, and *_data* which is the encoded data to be sent with the transaction. The purpose of this function is to call another contract to do things like for example, transfer tokens. In the case a user schedules a call to buy from an ICO with the EAC, they will need to proxy call the token contract after the execution in order to move the tokens they bought out of the TransactionRequest contract.

4.5 Retrieval of Ether

All payments are automatically returned as part of normal request execution and cancellation. Since it is possible for these payments to fail, there are backup methods that can be called individually to retrieve these different payment or deposit values.

All of these functions may be called by anyone.

4.5.1 Returning the Claim Deposit

`TransactionRequest.`**`refundClaimDeposit`**`()`

This method will return the claim deposit if either of the following conditions are met.

- The request was cancelled.
- The execution window has passed.

4.5.2 Retrieving the Payment

`TransactionRequest.`**`sendPayment`**`()`

This function will send the `paymentOwed` value to the `paymentBenefactor`. This is only callable after the execution window has passed.

4.5.3 Retrieving the Donation

`TransactionRequest.`**`sendDonation`**`()`

This function will send the `donationOwed` value to the `donationBenefactor`. This is only callable after the execution window has passed.

4.5.4 Return any extra Ether

This function will send any exta ether in the contract that is not owed as a donation or payment and that is not part of the claim deposit back to the `owner` of the request. This is only callable if one of the following conditions is met.

- The request was cancelled.
- The execution window has passed.

CHAPTER 5

Claiming

`class TransactionRequest`

5.1 The Problem

The claiming mechanism solves a very important problem contained within the incentive scheme of the EAC. It's best to provide an example first then go into the specifics of the solution later.

Consider a situation where there are two people, Alice and Bob, competing to execute the same request. The request will issue a payment of 100 wei to whomever executes it.

Suppose that Alice and Bob both send their execution transactions at approximately the same time, but out of luck, Alice's transaction is included before Bob's.

Alice will receive the 100 wei payment, while Bob will receive no payment as well as having paid the gas costs for his execution transaction that was rejected. Suppose that the gas cost Bob has now incurred is 25 wei.

In this situation we could assume that Alice and Bob have a roughly 50% chance of successfully executing any given transaction request, but since 50% of their attempts end up costing them money, their overall profits are being reduced by each failed attempt.

In this model, their expected payout is 75 wei for every two transaction requests they try to execute.

Now suppose that we add more competition via three additional people attempting to execute each transaction. Now Bob and Alice will only end up executing an average of 1 out of every 5 transaction requests, with the other 4 costing them 25 wei each. Now the result is that nobody is making a profit because the cost of the failed transactions cancel out any profit they are making.

5.2 The Solution

The claiming process is the current solution to this issue.

Prior to the execution window there is a section of time referred to as the claim window during which the request may be claimed by a single party for execution. An essiential part of claiming is that the claimer must put down a claim deposit in order to attain the rights to execute the request.

When a request has been claimed, the claimer is granted exclusive rights to execute the request during a window of blocks at the beginning of the execution window.

Whomever ends up executing the request receives the claim deposit as part of their payment. This means that if the claimer fulfills their commitment to execute the request their deposit is returned to them intact. Otherwise, if someone else executes the request then they will receive the deposit as an additional reward.

5.3 Claim Deposit

In order to claim a request you must put down a deposit. This deposit amount is specified by the scheduler of the transaction. The account claiming the transaction request must send at least the `claimDeposit` amount when they attempt to claim an execution.

The `claimDeposit` is returned to the claiming account when they execute the transaction request or when the call is cancelled. However, if the account that claims the call later fails to execute then they will lose their claim deposit to whoever executes instead.

5.4 How claiming effects payment

A claimed request does not pay the same as an unclaimed request. The earlier the request is claimed, the less it will pay, and conversely, the later the request is claimed, the more it pays.

This is a linear transition from getting paid 0% of the total payment if the request is claimed at the earliest possible time up to 100% of the total payment at the very end of the claim window. This multiplier is referred to as the *payment modifier*. Refer to the code block pasted below to see how the smart contract calculates the multiplier. This examples is taken from lines 71 - 79 of *RequestScheduleLib.sol*.

```
function computePaymentModifier(ExecutionWindow storage self)
    internal view returns (uint8)
{
    uint paymentModifier = (getNow(self).sub(firstClaimBlock(self)))
                            .mul(100).div(self.claimWindowSize);
    assert(paymentModifier <= 100);

    return uint8(paymentModifier);
}
```

It is important to note that the *payment modifier* does not apply to gas reimbursements which are always paid in full. No matter when a call is claimed, or how it is executed, it will **always** provide a full gas reimbursement.

Note: In the past, this was not always the case since the EAC used a slightly different scheme to calculate an anchor gas price. In version 0.9.0 the anchor gas price was removed in favor of forcing the scheduler of the transaction to explicitly specify an **exact** gas price. So the gas to execute a transaction is always reimbursed exactly to the executor of the transaction.

For clarification of the payment modifier let's consider an example. Assume that a transaction request has a `payment` set to 2000 wei, a `claimWindowSize` of 255 blocks, a `freezePeriod` of 10 blocks, and a `windowStart` set at block 500. The first claimable block is calculated by subtracting the `claimWindowSize` and the `freezePeriod` from the `windowStart` like so:

```
first_claim_block = 500 - 255 - 10 = 235
```

In this case, the request would have a payment of 0 at block 235.

```
(235 - 235) * 100 // 255 = 0
```

At block 245 it would pay 60 wei or 3% of the total payment.

```
(245 - 235) * 100 // 255 = 3
```

At block 489 it would pay 1980 wei or 99% of the total payment.

```
(489 - 235) * 100 // 255 = 99
```

5.5 Gas Costs

The gas costs for claim transactions are *not* reimbursed. They are considered the cost of doing business and should be taken into consideration when claiming a request. If the request is claimed sufficiently early in the claim window it is possible that the `payment` will not fully offset the transaction costs of claiming the request. EAC clients should take precaution that they do not claim transaction requests without estimating whether they will be profitable first.

CHAPTER 6

Execution

`class TransactionRequest`

Warning: Anyone wishing to write their own execution client should be sure they fully understand all of the intricacies related to the execution of transaction requests. The guarantees in place for those executing requests are only in place if the executing client is written appropriately. Reading this documentation is a good start.

6.1 Important Windows of Blocks/Time

6.1.1 Freeze Window

Each request may specify a `freezePeriod`. This defines a number of blocks or seconds prior to the `windowStart` during which no actions may be performed against the request. This is primarily in place to provide some level of guarantee to those executing the request. For anyone executing requests, once the request enters the `freezePeriod` they can know that it will not be cancelled and that they can send the executing transaction without fear of it being cancelled at the last moment before the execution window starts.

6.1.2 The Execution Window

The **execution window** is the range of blocks or timestamps during which the request may be executed. This window is defined as the range of blocks or timestamps from `windowStart` till `windowStart + windowSize`.

For example, if a request was scheduled with a `windowStart` of block 2100 and a `windowSize` of 255 blocks, the request would be allowed to be executed on any block such that `windowStart <= block.number <= windowStart + windowSize`.

As another example, if a request was scheduled with a `windowStart` of block 2100 and a `windowSize` of 0 blocks, the request would only be allowed to be executed at block 2100.

Very short `windowSize` configurations likely lower the chances of your request being executed at the desired time since it is not possible to force a transaction to be included in a specific block. The party executing your request may either fail to get the transaction included in the correct block *or* they may choose to not try for fear that their transaction will not be mined in the correct block, thereby not receiving their reimbursment for their gas costs.

Similarly, very short ranges of time for timestamp based calls may even make it impossible to execute the call. For example, if you were to specify a `windowStart` at 1480000010 and a `windowSize` of 5 seconds then the request would only be executable on blocks whose `block.timestamp` satisfied the conditions `1480000010 <= block.timestamp <= 1480000015`. Given that it is entirely possible that no blocks are mined within this small range of timestamps there would never be a valid block for your request to be executed.

Note: It is worth pointing out that actual size of the execution window will always be `windowSize + 1` since the bounds are inclusive.

6.1.3 Reserved Execution Window

Each request may specify a `claimWindowSize` which defines a number of blocks or seconds at the beginning of the execution window during which the request may only be executed by the address which has claimed the request. Once this window has passed the request may be executed by anyone.

Note: If the request has not been claimed this window is treated no differently than the remainder of the execution window.

For example, if a request specifies a `windowStart` of block 2100, a `windowSize` of 100 blocks, and a `reservedWindowSize` of 25 blocks then in the case that the request was claimed then the request would only be executable by the claimer for blocks satisfying the condition `2100 <= block.number < 2125`.

Note: It is worth pointing out that unlike the *execution window* the *reserved execution window* is not inclusive of it's righthand bound.

If the `reservedWindowSize` is set to 0, then there will be no window of blocks during which the execution rights are exclusive to the claimer. Similarly, if the `reservedWindowSize` is set to be equal to the full size of the *execution window* or `windowSize + 1` then there will be not window after the *reserved execution window* during which execution can be triggered by anyone.

The *`RequestFactory`* will allow a `reservedWindowSize` of any value from 0 up to `windowSize` + 1, however, it is highly recommended that you pick a number around 16 blocks or 270 seconds, leaving at least the same amount of time unreserved during the second portion of the *execution window*. This ensures that there is sufficient motivation for your call to be claimed because the person claiming the call knows that they will have ample opportunity to execute it when the *execution window* comes around. Conversely, leaving at least as much time unreserved ensures that in the event that your request is claimed but the claimer fails to execute the request that someone else has plenty of of time to fulfill the execution before the *execution window* ends.

6.2 The Execution Lifecycle

When the **:method:'TransactionRequest.execute()'** function is called the contract goes through three main sections of logic which are referred to as a whole as the *execution lifecycle*.

1. Validation: Handles all of the checks that must be done to ensure that all of the conditions are correct for the requested transaction to be executed.
2. Execution: The actual sending of the requested transaction.
3. Accounting: Computing and sending of all payments to the necessary parties.

6.2.1 Part 1: Validation

During the validation phase all of the following validation checks must pass.

Check #1: Not already called

Requires the `wasCalled` attribute of the transaction request to be `false`.

Check #2: Not Cancelled

Requires the `isCancelled` attribute of the transaction request to be `false`.

Check #3: Not before execution window

Requires `block.number` or `block.timestamp` to be greater than or equal to the `windowStart` attribute.

Check #4: Not after execution window

Requires `block.number` or `block.timestamp` to be less than or equal to `windowStart + windowSize`.

Check #5 and #6: Within the execution window and authorized

- **If the request is claimed**
    - **If the current time is within the *reserved execution window***
        - Requires that `msg.sender` to be the `claimedBy` address
    - **Otherwise during the remainder of the *execution window***
        - Always passes.
- **If the request is not claimed.**
    - Always passes if the current time is within the *execution window*

Check #8: Sufficient Call Gas

Requires that the current value of `msg.gas` be greater than the *minimum call gas*. See minimum-call-gas for details on how to compute this value as it includes both the `callGas` amount as well as some extra for the overhead involved in execution.

6.2.2 Part 2: Execution

The execution phase is very minimalistic. It marks the request as having been called and then dispatches the requested transaction, storing the success or failure on the `wasSuccessful` attribute.

6.2.3 Part 3: Accounting

The accounting phase accounts for all of the payments and reimbursements that need to be sent.

The *fee* payment is the mechanism through which developers can earn a return on their development efforts on the Alarm service. When a person schedules a transaction they may choose to enter a `fee` amount which will get sent to the developer. This value is multiplied by the *gas multiplier* (see gas-multiplier) and sent to the `feeRecipient` address.

Next the payment for the actual execution is computed. The formula for this is as follows:

```
totalPayment = payment * gasMultiplier + gasUsed * tx.gasprice +
claimDeposit
```

The three components of the `totalPayment` are as follows.

- `payment * gasMultiplier`: The actual payment for execution.
- `gasUsed * tx.gasprice`: The reimbursement for the gas costs of execution. This is not going to exactly match the actual gas costs, but it will always err on the side of overpaying slightly for gas consumption.
- `claimDeposit`: If the request is not claimed this will be 0. Otherwise, the `claimDeposit` is always given to the executor of the request.

After these payments have been calculated and sent, the `Executed` event is logged, and any remaining ether that is not allocated to be paid to any party is sent back to the address that scheduled the request.

6.3 Sending the Execution Transaction

In addition to the pre-execution validation checks, the following things should be taken into considuration when sending the executing transaction for a request.

6.3.1 Gas Reimbursement

If the `gasPrice` of the network has increased significantly since the request was scheduled it is possible that it no longer has sufficient ether to pay for gas costs. The following formula can be used to compute the maximum amount of gas that a request is capable of paying:

```
(request.balance - 2 * (payment + fee)) / tx.gasprice
```

If you provide a gas value above this amount for the executing transaction then you are not guaranteed to be fully reimbursed for gas costs.

6.3.2 Minimum ExecutionGas

When sending the execution transaction, you should use the following rules to determine the minimum gas to be sent with the transaction:

- Start with a baseline of the `callGas` attribute.
- Add `180000` gas to account for execution overhead.
- If you are proxying the execution through another contract such that during execution `msg.sender != tx.origin` then you need to provide an additional `700 * requiredStackDepth` gas for the stack depth checking.

For example, if you are sending the execution transaction directly from a private key based address, and the request specified a `callGas` value of 120000 gas then you would need to provide `120000 + 180000 => 300000` gas.

If you were executing the same request, except the execution transaction was being proxied through a contract, and the request specified a `requiredStackDepth` of 10 then you would need to provide `120000 + 180000 + 700 * 10 => 307000` gas.

CHAPTER 7

Request Factory

`class RequestFactory`

7.1 Introduction

The `RequestFactory` contract is the lowest level API for creating transaction requests. It handles:

- Validation and Deployment of `TransactionRequest` contracts
- Tracking of all addresses that it has deployed.

This contract is designed to allow tuning of all transaction parameters and is probably the wrong API to integrate with if your goal is to simply schedule transactions for later execution. The *Request Factory* API is likely the right solution for these use cases.

7.2 Interface

```
pragma solidity 0.4.19;

contract RequestFactoryInterface {

    event RequestCreated(address request, address indexed owner);

    function createRequest(address[3] addressArgs,
                           uint[12] uintArgs,
                           bytes callData)
        public payable returns (address);

    function createValidatedRequest(address[3] addressArgs,
                                    uint[12] uintArgs,
                                    bytes callData)
        public payable returns (address);

    function validateRequestParams(address[3] addressArgs,
                                   uint[12] uintArgs,
                                   bytes callData,
                                   uint endowment)
        public view returns (bool[6]);

    function isKnownRequest(address _address)
        public view returns (bool);
}
```

7.3 Events

`RequestFactory.`**`RequestCreated`**(*address request*)

The `RequestCreated` event will be logged for each newly created *`TransactionRequest`*.

`RequestFactory.`**`ValidationError`**(*uint8 error*)

The `ValidationError` event will be logged when an attempt is made to create a new *`TransactionRequest`* which fails due to validation errors. The `error` represents an error code that maps to the following errors.

- `0 => InsufficientEndowment`
- `1 => ReservedWindowBiggerThanExecutionWindow`
- `2 => InvalidTemporalUnit`
- `3 => ExecutionWindowTooSoon`
- `4 => CallGasTooHigh`
- `5 => EmptyToAddress`

7.4 Function Arguments

Because of the call stack limitations imposed by the EVM, all of the following functions on the `RequestFactory` contract take their arguments in the form of the following form.

- `address[3] _addressArgs`
- `uint256[11] _uintArgs`
- `bytes _callData`

The arrays map to to the following `TransactionRequest` attributes.

- **Addresses (`address`)**
  - `_addressArgs[0] => meta.owner`
  - `_addressArgs[1] => paymentData.feeRecipient`
  - `_addressArgs[2] => txnData.toAddress`
- **Unsigned Integers (`uint` aka `uint256`)**
  - `_uintArgs[0] => paymentData.fee`
  - `_uintArgs[1] => paymentData.payment`
  - `_uintArgs[2] => schedule.claimWindowSize`
  - `_uintArgs[3] => schedule.freezePeriod`
  - `_uintArgs[4] => schedule.reservedWindowSize`
  - `_uintArgs[5] => schedule.temporalUnit`
  - `_uintArgs[6] => schedule.windowStart`
  - `_uintArgs[7] => schedule.windowSize`
  - `_uintArgs[8] => txnData.callGas`
  - `_uintArgs[9] => txnData.callValue`
  - `_uintArgs[10] => txnData.gasPrice`

7.5 Validation

`RequestFactory.`**`validateRequestParams`**(*address[3] _addressArgs, uint[11] _uintArgs, bytes _callData, uint _endowment) public returns (bool[6] result*)

The `validateRequestParams` function can be used to validate the parameters to both `createRequest` and `createValidatedRequest`. The additional parameter `endowment` should be the amount in wei that will be sent during contract creation.

This function returns an array of `bool` values. A `true` means that the validation check succeeded. A `false` means that the check failed. The `result` array's values map to the following validation checks.

7.5.1 Check #1: Insufficient Endowment

- `result[0]`

Checks that the provided `endowment` is sufficient to pay for the fee and payment as well as gas reimbursment.

The required minimum endowment can be computed as the sum of the following:

- `callValue` to provide the ether that will be sent with the transaction.
- `2 * payment` to pay for maximum possible payment
- `2 * fee` to pay for maximum possible fee
- `callGas * txnData.gasPrice` to pay for `callGas`.
- `180000 * txnData.gasPrice` to pay for the gas overhead involved in transaction execution.

7.5.2 Check #2: Invalid Reserved Window

- `result[1]`

Checks that the `reservedWindowSize` is less than or equal to `windowSize + 1`.

7.5.3 Check #3: Invalid Temporal Unit

- `result[2]`

Checks that the `temporalUnit` is either `1` to specify block based scheduling, or `2` to specify timestamp based scheduling.

7.5.4 Check #4: Execution Window Too Soon

- `result[3]`

Checks that the current `now` value is not greater than `windowStart - freezePeriod`.

- When using block based scheduling, `block.number` is used for the `now` value.
- When using timestamp based scheduling, `block.timestamp` is used.

7.5.5 Check #5: Call Gas too high

- `result[5]`

Check that the specified `callGas` value is not greater than the current `gasLimit - 140000` where `140000` is the gas overhead of request execution.

7.5.6 Check #6: Empty To Address

- `result[6]`

Checks that the `toAddress` is not the null address `0x0000000000000000000000000000000000000000`.

7.6 Creation of Transaction Requests

`RequestFactory.`**`createRequest`**(*address[3] _addressArgs*, *uint[11] _uintArgs*, *bytes _callData) public payable returns* (*address*)

This function deploys a new *`TransactionRequest`* contract. This function does not perform any validation and merely directly deploys the new contract.

Upon successful creation the `RequestCreated` event will be logged.

`RequestFactory.`**`createValidatedRequest`**(*address[3] _addressArgs*, *uint[11] _uintArgs*, *bytes _callData) public payable returns* (*address*)

This function first performs validation of the provided arguments and then deploys the new *`TransactionRequest`* contract when validation succeeds.

When validation fails, a `ValidationError` event will be logged for each validation error that occured.

7.7 Tracking API

`RequestFactory.`**`isKnownRequest`**(*address _address) returns* (*bool*)

This method will return `true` if the address is a *`TransactionRequest`* that was created from this contract.

CHAPTER 8

Request Tracker

class RequestTracker

8.1 Introduction

The *RequestTracker* contract is a simple database contract that exposes an API suitable for querying for scheduled transaction requests. This database is *permissionless* in so much as it partitions transaction requests by the address that reported them. This means that *anyone* can deploy a new request scheduler that conforms to whatever specific rules they may need for their use case and configure it to report any requests it schedules with this tracker contract.

Assuming that such a scheduler was written to still use the *RequestFactory* contract for creation of transaction requests, the standard execution client will pickup and execute any requests that this scheduler creates.

8.2 Interface

```
pragma solidity 0.4.19;

contract RequestTrackerInterface {

    function getWindowStart(address factory, address request)
        public view returns (uint);
    function getPreviousRequest(address factory, address request)
        public view returns (address);

    function getNextRequest(address factory, address request)
        public view returns (address);

    function addRequest(address request, uint startWindow)
        public returns (bool);

    function removeRequest(address request)
        public returns (bool);

    function isKnownRequest(address factory, address request)
        public view returns (bool);

    function query(address factory, bytes2 operator, uint value)
        public view returns (address);

}
```

8.3 Database Structure

All functions exposed by the `RequestTracker` take an `address` as the first argument. This is the address that reported the request into the tracker. This address is referred to as the *scheduling address* which merely means that it is the address that reported this request into the tracker. Each *scheduling address* effectively receives it's own database.

All requests are tracked and ordered by their `windowStart` value. The tracker does not distinguish between block based scheduling and timestamp based scheduling.

It is possible for a single `TransactionRequest` contract to be listed under multiple scheduling addresses since any address may report a request into the database.

8.4 Chain of Trust

Since this database is permissionless, if you plan to consume data from it, you should validate the following things.

- Check with the `RequestFactory` that the `request` address is known using the **:method:'RequestFactory.isKnownRequest()'** function.
- Check that the `windowStart` attribute of the `TransactionRequest` contract matches the registered `windowStart` value from the `RequestTracker`.

Any request created by the `RequestFactory` contract regardless of how it was created should be safe to execute using the provided execution clients.

8.5 API

`RequestTracker.`**`isKnownRequest`**(*address scheduler, address request) constant returns (bool*)

Returns `true` or `false` depending on whether this address has been registered under this scheduler address.

`RequestTracker.`**`getWindowStart`**(*address scheduler, address request) constant returns (uint*)

Returns the registered `windowStart` value for the request. A return value of 0 indicates that this address is not known.

`RequestTracker.`**`getPreviousRequest`**(*address scheduler*, *address request) constant returns (address*)

Returns the address of the request who's `windowStart` comes directly before this one.

`RequestTracker.`**`getNextRequest`**(*address scheduler*, *address request) constant returns (address*)

Returns the address of the request who's `windowStart` comes directly after this one.

`RequestTracker.`**`addRequest`**(*address request*, *uint startWindow) constant returns (bool*)

Add an address into the tracker. The `msg.sender` address will be used as the *scheduler address* to determine which database to use.

`RequestTracker.`**`removeRequest`**(*address request) constant returns (bool*)

Remove an address from the tracker. The `msg.sender` address will be used as the *scheduler address* to determine which database to use.

`RequestTracker.`**`query`**(*address scheduler*, *bytes2 operator*, *uint value) constant returns (address*)

Query the database for the given scheduler. Returns the address of the 1st record which evaluates to `true` for the given query.

Allowed values for the `operator` parameter are:

- `'>'`: For strictly greater than.
- `'>='`: For greater than or equal to.
- `'<'`: For strictly less than.
- `'<='`: For less than or equal to.
- `'=='`: For less than or equal to.

The `value` parameter is what the `windowSize` for each record will be compared to.

If the return address is the null address `0x0000000000000000000000000000000000000000` then no records matched.

CHAPTER 9

Request Factory

`class Scheduler`

9.1 Introduction

The `Scheduler` contract is the high level API for scheduling transaction requests. It exposes a very minimal subset of the full parameters that can be specified for a `TransactionRequest` in order to provide a simplified scheduling API with fewer foot-guns.

The Alarm service exposes two schedulers.

- `BlockScheduler` for block based scheduling.
- `TimestampScheduler` for timestamp based scheduling.

Both of these contracts present an identical API. The only difference is which `temporalUnit` that each created `TransactionRequest` contract is configured with.

9.2 Interface

```
pragma solidity 0.4.19;

import "contracts/Library/RequestScheduleLib.sol";
import "contracts/Library/SchedulerLib.sol";

/**
 * @title SchedulerInterface
 * @dev The base contract that the higher contracts: BaseScheduler, BlockScheduler and TimestampSch
 */
contract SchedulerInterface {
    using SchedulerLib for SchedulerLib.FutureTransaction;

    // The RequestFactory address which produces requests for this scheduler.
    address public factoryAddress;

    // The TemporalUnit of this scheduler.
    RequestScheduleLib.TemporalUnit public temporalUnit;

    /*
     * Local storage variable used to house the data for transaction
     * scheduling.
     */
    SchedulerLib.FutureTransaction public futureTransaction;

    /*
     * When applied to a function, causes the local futureTransaction to
     * get reset to it's defaults on each function call.
     */
    modifier doReset {
        if (temporalUnit == RequestScheduleLib.TemporalUnit.Blocks) {
            futureTransaction.resetAsBlock();
        } else if (temporalUnit == RequestScheduleLib.TemporalUnit.Timestamp) {
            futureTransaction.resetAsTimestamp();
        } else {
            revert();
        }
        _;
    }

    function schedule(address   _toAddress,
                      bytes     _callData,
                      uint[8]   _uintArgs)
        doReset
        public payable returns (address);

}
```

9.3 Defaults

The following defaults are used when creating a new *TransactionRequest* contract via either *Scheduler* contract.

- `feeRecipient`: `0xecc9c5fff8937578141592e7E62C2D2E364311b8` which is the address of the developer contribution wallet, which is used to fund the project.
- `payment`: `1000000 * tx.gasprice` set at the time of scheduling.

- `fee`: `10000 * tx.gasprice` or 1/100th of the default payment.
- `reservedWindowSize`: 16 blocks or 5 minutes.
- `freezePeriod`: 10 blocks or 3 minutes
- `claimWindowSize`: 255 blocks or 60 minutes.

9.4 API

There is just one `schedule` method on each *`Scheduler`* contract with different call signatures. (Prior versions of the EAC had 2 API methods, we reduced this down to only the full API to force specification of all parameters.)

`Scheduler.`**`schedule`**(*address _toAddress*, *bytes _callData*, *uint[7] _uintArgs) public payable returns (address newRequest*)

The `_toAddress` is the recipient that the transaction will be sent to when it is executed. The recipient can be any valid Ethereum address including both user accounts and contracts. `_callData` is the encoded bytecode that will be sent with the transaction. Simple value transfers can set this variable to an empty string, but more complex calls will need to encode the method of the inteded call and pass it in this variable.

The `_uintArgs` map to the following variables:

- `_uintArgs[0]`: The `callGas` to be sent with the executing transaction.
- `_uintArgs[1]`: The `value` in wei to be sent with the transaction.
- `_uintArgs[2]`: The `windowSize`, or size of the exeuction window.
- `_uintArgs[3]`: The `windowStart`, or the block / timestamp of when the execution window begins.
- `_uintArgs[4]`: The `gasPrice` that must be sent with the executing transaction.
- `_uintArgs[5]`: The `fee` value attached to the transaction.
- `_uintArgs[6]`: The `payment` value attached to the transaction.

The method returns the `address` of the newly created *`TransactionRequest`*.

9.5 Endowments

When scheduling a transaction, you must provide sufficient ether to cover all of the execution costs with some buffer to account for possible changes in the network gas price. See *Check #1: Insufficient Endowment* for more information on how to compute the endowment.

CHAPTER 10

Changelog

10.1 0.9.1

- Replaced term *donation* and *donationBenefactor* with *fee* and *feeRecipient* (done)
- Replaced term *payment* to *bounty*. (done)
- Removed the hardcoding of FEE_RECIPIENT to be passed in on creation of schedulers. (done)
- Added an indexed parameter to *RequestCreated()* event in RequestFactory.sol (done)
- Peg contracts to compiler version 0.4.19 (done)
- Change *paymentData.hasBenefactor()* to *paymentData.hasFeeRecipient()* (done)
- Tidied up and cleaned the test suite. (in progress)

10.2 0.9.0-beta

- Update contracts to solidity 0.4.18.
- Digger.sol removed due to **'EIP 150'_** making it obsolete.
- All stack depth checking also obsolete due to EIP150 removed.
- SafeSendLib.sol removed due to Solidity keywords *transfer* and *send* making it obsolete.
- Simplified scheduling API to singular *schedule()* function.
- Added the *proxy()* function to *TransactionRequest* contract.
- Integrate Truffle framework.
- Rewrote entire test suite to use Truffle.
- Revamped the documentation.

10.3 0.8.0 (unreleased)

- Full rewrite of all contracts.
- Support for both time and block based scheduling.
- New permissionless call tracker now used to track scheduled calls.
- Donation address can now be configured.
- Request execution window size is now configurable.
- Reserved claim window size is now configurable.
- Freeze period is now configurable.
- Claim window size is now configurable.
- All payments now support pull mechanism for retrieving payments.

10.4 0.7.0

- Scheduled calls can now specify a required gas amount. This takes place of the `suggestedGas` api from 0.6.0
- Scheduled calls can now send value along with the transaction.
- Calls now protect against stack depth attacks. This is configurable via the `requiredStackDepth` option.
- Calls can now be scheduled as soon as 10 blocks in the future.
- Experimental implementation of market-based value for the `defaultPayment`
- `scheduleCall` now has 31 different call signatures.

10.5 0.6.0

- Each scheduled call now exists as it's own contract, referred to as a call contract.
- Removal of the Caller Pool
- Introduction of the claim api for call.
- Call Portability. Scheduled calls can now be trustlessly imported into future versions of the service.

10.6 0.5.0

- Each scheduled call now exists as it's own contract, referred to as a call contract.
- The authorization API has been removed. It is now possible for the contract being called to look up `msg.sender` on the scheduling contract and find out who scheduled the call.
- The account management API has been removed. Each call contract now manages it's own gas money, the remainder of which is given back to the scheduler after the call is executed.
- All of the information that used to be stored about the call execution is now placed in event logs (gasUsed, wasSuccessful, wasCalled, etc)

10.7 0.4.0

- Convert Alarm service to use library contracts for all functionality.
- CallerPool contract API is now integrated into the Alarm API

10.8 0.3.0

- Convert Alarm service to use Grove for tracking scheduled call ordering.
- Enable logging most notable Alarm service events.
- Two additional convenience functions for invoking `scheduleCall` with **gracePeriod** and **nonce** as optional parameters.

10.9 0.2.0

- Fix for Issue 42. Make the free-for-all bond bonus restrict itself to the correct set of callers.
- Re-enable the right tree rotation in favor of removing three `getLastX` function. This is related to the pi-million gas limit which is restricting the code size of the contract.

10.10 0.1.0

- Initial release.

Index