Ethereum Alarm Clock Documentation

Release 1.0.0

Piper Merriam

Dec 12, 2017

Contents

The Ethereum Alarm Clock is a service that allows scheduling transactions to be executed at a later time on the ethereum blockchain. This is accomplished by specifying all of the details for the transaction you wish to send, as well as providing up-front payment for gas costs, allowing your transaction to be executed for you at a later time.

The service is completely trustless, meaning that the entire service operates as smart contracts on the Ethereum blockchain, with no priviledged access given to any party.

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 start scheduling transactions now then check out the *Quickstart*.

If you are looking to build a lower level integration with the service then our *./TODO* is a good place to start.

Contents:

CHAPTER 1

Introduction

1.1 What problem does this solve

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

First, you need to understand the difference between private key based accounts and contract accounts. There are two types of accounts on the Ethereum blockchain.

1. Accounts that have a private key.
2. Contracts *(which do not have a private key)*

Private key accounts are the accounts that humans operate, where as contract accounts are deployed pieces of code capable of executing some computer program. Contract accounts cannot however trigger their own code execution.

All code execution in the Ethreum Virtual Machine, or 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.

The second part of the problem is that when you send a transaction it is executed as soon as it is included in a block. The Ethereum protocol does not provide any way to create a transaction to be executed at a later time.

This leads us to the problem that the Alarm service solves. With the functionality provided by this service, transactions can be securely scheduled to be executed at a later time.

1.2 How transactions are executed

When a transaction is scheduled a new smart contract is created that holds all of the information needed to execute the transaction. It may be useful to think of this as an order on an exchange. When called during the specified execution window, this contract will send the transaction as specified and then pay the account that triggered the execution.

These contracts are referred to as *TransactionRequest* contracts 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 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.

In an ideal situation, there is a sufficient volume of scheduled transactions that operating a server to execute these transactions is a profitable endeavor. The reality is that I operate between 3-5 execution servers dedicated filling this role until there is sufficient volume that I am confident I can turn those servers off or until it is no longer feasible for me to continue paying their costs.

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 Scheduling your first transaction

The first step is to establish how we will interact with the Alarm service's *Scheduler* contract. Lets create an abstract contract to accomplish this.

```
contract SchedulerInterface {
    //
    // params:
    // - uintArgs[0] callGas
    // - uintArgs[1] callValue
    // - uintArgs[2] windowStart
    // - uint8 windowSize
    // - bytes callData
    // - address toAddress
    //
    function scheduleTransaction(address toAddress,
                                 bytes callData,
                                 uint8 windowSize,
                                 uint[3] uintArgs) public returns (address);
}
```

This abstract contract exposes the function `scheduleTransaction` which will return the address of the newly created *TransactionRequest* contract.

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

```
contract DelayedPayment {
    SchedulerInterface constant scheduler = SchedulerInterface(0xTODO);
```

```
    uint lockedUntil;
    address recipient;

    function DelayedPayment(address _recipient, uint numBlocks) {
        // set the time that the funds are locked up
        lockedUntil = block.number + numBlocks;
        recipient = _recipient;

        uint[3] memory uintArgs = [
            200000,      // the amount of gas that will be sent with the txn.
            0,           // the amount of ether (in wei) that will be sent with the␣
↪txn
            lockedUntil, // the first block number on which the transaction can be␣
↪executed.
        ];
        scheduler.scheduleTransaction.value(2 ether)(
            address(this),  // The address that the transaction will be sent to.
            "",             // The call data that will be sent with the transaction.
            255,            // The number of blocks this will be executable.
            uintArgs,       // The tree args defined above
        )
    }

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

    function payout() public returns (bool) {
        if (now < lockedUntil) return false;

        return recipient.call.value(this.balance)();
    }
}
```

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 `scheduleTransaction` method on the `scheduler` contract. The function takes a total of 6 parameters, 3 of which are passed in as an array. Lets briefly go over what each of these parameters are.

`scheduleTransaction(address toAddress, bytes callData, uint8 windowSize, [uint callGas, uint callValue, uint windowStart])`

- `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.
- `uint windowStart`: The first block number that the transaction will be executable.
- `uint8 windowSize`: The number of blocks after `windowSize` during which the transaction will still be executable.

TODO: more

CHAPTER 3

Architecture

3.1 Overview

The Alarm service is made 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.

This database tracks requests based on the address that submits them. This allows the *RequestTracker* to be un-permissioned allowing 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*.

This also enables those executing transaction requests to choose which *scheduler addresses* they wish to execute transactions for.

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 0.4.1;


contract TransactionRequestInterface {
    /*
     *  Primary actions
     */
    function execute() public returns (bool);
    function cancel() public returns (bool);
    function claim() public returns (bool);

    /*
     *  Data accessors
     */
    function requestData() constant returns (address[6],
                                             bool[3],
                                             uint[15],
                                             uint8[1]);
    function callData() constant returns (bytes);

    /*
     *  Pull mechanisms for payments.
     */
    function refundClaimDeposit() public returns (bool);
    function sendDonation() public returns (bool);
    function sendPayment() 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 => StackTooDeep`
- `6 => InsufficientGas`

`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.requiredStackDepth`
- **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 requiredStackDepth`
The number of stack frames required by this 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.

- 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.

4.5 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

To understand the claiming mechanism it is important to understand the problem it solves.

Consider a situation where there are two people Alice and Bob competing to execute the same request that 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 are 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 nobody is making a profit because the cost of the failed transactions now cancels 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. Part of claiming includes putting down a deposit.

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 equal to twice the `payment` amount associated with this request.

The deposit is returned during execution, or when the call is cancelled.

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*.

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. The only case where this may end up not being true is in cases where the gas price has changed drastically since the time the request was scheduled and the contract's endowment is now sufficiently low that it is not longer funded with sufficient ether to cover these costs.

For example, if the request has a `payment` of 2000 wei, a `claimWindowSize` of 255 blocks, a `freezePeriod` of 10 blocks, and a `windowStart` set at block 500. In this case, the request would have a payment of 0 at block 235. At block 235 it would provide a payment of 20 wei. At block 245 it would pay 220 wei or 11% of the total payment. At block 489 it would pay 2000 wei or 100% of the total payment.

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.

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.

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 and thus 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 included in the correct block and thus they will not recieve a 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 #7: Stack Depth Check

In order to understand this check you need to understand the problem it solves. One of the more subtle attacks that can be executed against a requested transaction is to force it to fail by ensuring that it will encounter the EVM stack limit. Without this check the executor of a transaction request could force *any* request to fail by arbitrarily increasing the stack depth prior to execution such that when the transaction is sent it encounters the maximum stack depth and fails. From the perspective of the *`TransactionRequest`* contract this sort of failure is indistinguishable from any other exception.

In order to prevent this, prior to execution, the *`TransactionRequest`* contract will ensure that the stack can be extended by a number of stack frames equal to `requiredStackDepth`. This check passes if the stack can be extended by this amount.

This check will be skipped if `msg.sender == tx.origin` since in this case it is not possible for the stack to have been arbitrarily extended prior to execution.

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 *donation* payment is the mechanism through which developers can earn a return on their development efforts on the Alarm service. For the *official* scheduler deployed as part of the alarm service this defaults to 1% of the default payment. This value is multiplied by the *gas multiplier* (see *Gas Multiplier*) and sent to the `donationBenefactor` 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 Gas Multiplier

To understand the *gas multiplier* you must understand the problem it solves.

Transactions requests always provide a 100% reimbursment of gas costs. This is implemented by requiring the scheduler to provide sufficient funds up-front to cover the future gas costs of their transaction. Ideally we want the sender of the transaction that executes the request to be motivated to use a `gasPrice` that is as low as possible while still allowing the transaction to be included in a block in a timely manner.

A naive approach would be to specify a *maximum* gas price that the scheduler is willing to pay. This might be possible for requests that will be processed a short time in the future, but for transactions that are scheduled sufficiently far in the future it isn't feasible to set a gas price that is going to reliably reflect the current normal gas prices at that time.

In order to mitigate this issue, we instead provide a financial incentive to the party executing the request to provide as low a gas cost as possible while still getting their transaction included in a timely manner.

Those executing the request are already sufficiently motivated to provide a gas price that is high enough to get the transaction mined in a reasonable time since if the price they specify is too low it is likely that someone else will execute the request before them, or that their transaction will not be included before the *execution window* closes.

So, to provide incentive to keep the gas cost reasonably low, the *gas multiplier* concept was introduced. Simply put, the multiplier produces a number between 0 and 2 which is applid to the `payment` that will be sent for fulfilling the request.

At the time of scheduling, the `gasPrice` of the scheduling transaction is stored. We refer to this as the `anchorGasPrice` as we can assume with some reliability that this value is a *reasonable* gas cost that the scheduler is willing to pay.

At the time of execution, the following will occur based on the `gasPrice` used for the executing transaction:

- If `gasPrice` is equal to the `anchorGasPrice` then the *gas multiplier* will be 1, meaning that the `payment` will be issued as is.
- When the `gasPrice` is greater than the `anchorGasPrice`, the *gas multiplier* will approach 0 meaning that the payment will steadily get smaller for higher gas prices.
- When the `gasPrice` is less than the `anchorGasPrice`, the *gas multiplier* will approach 2 meaning that the payment will steadily get larger for lower gas prices.

The formula used is the following.

- If the execution `gasPrice` is greater than `anchorGasPrice`:

  ```
  gasMultiplier = anchorGasPrice / tx.gasprice
  ```

- Else (if the execution `gasPrice` is less than or equal to the `anchorGasPrice`:

  `gasMultiplier = 2 - (anchorGasPrice / (2 * anchorGasPrice - tx.gasprice))`

For example, if at the time of scheduling the gas price was 100 wei and the executing transaction uses a `gasPrice` of 200 wei, then the gas multiplier would be `100 / 200 => 0.5`.

Alternatively, if the transaction used a `gasPrice` of 75 wei then the gas multiplier would be `2 - (100 / (2 * 100 - 75)) => 1.2`.

6.4 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.

6.4.1 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 + donation)) / 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.

6.4.2 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 0.4.1;


contract RequestFactoryInterface {
    event RequestCreated(address request);

    function createRequest(address[3] addressArgs,
                           uint[11] uintArgs,
                           bytes callData) returns (address);
    function validateRequestParams(address[3] addressArgs,
                                   uint[11] uintArgs,
                                   bytes callData,
                                   uint endowment) returns (bool[7]);
    function createValidatedRequest(address[3] addressArgs,
                                    uint[11] uintArgs,
                                    bytes callData) returns (address);
    function isKnownRequest(address _address) 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`
- `0 => ReservedWindowBiggerThanExecutionWindow`
- `0 => InvalidTemporalUnit`
- `0 => ExecutionWindowTooSoon`
- `0 => InvalidRequiredStackDepth`
- `0 => CallGasTooHigh`
- `0 => 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.donationBenefactor`
  - `addressArgs[2] => txnData.toAddress`
- **Unsigned Integers (`uint` aka `uint256`)**
  - `uintArgs[0] => paymentData.donation`
  - `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.requiredStackDepth`

7.5 Validation

`RequestFactory.`**`validateRequestParams`**(*address[3] addressArgs*, *uint[11] uintArgs*, *bytes callData*, *uint endowment) returns (bool[7] 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 donation 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 * donation` to pay for maximum possible donation
- `2 * callGas * tx.gasprice` to pay for `callGas` with up to a 2x increase in the network gas price.
- `2 * 700 * requiredStackDepth * tx.gasprice` to pay gas for the stack depth checking with up to a 2x increase in network gas costs.
- `2 * 180000 * tx.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: Invalid Stack Depth Check

- `result[4]`

Checks that the `requiredStackDepth` is less than or equal to 1000.

7.5.6 Check #6: 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.

7.5.7 Check #7: 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) 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) 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 partitiions 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 0.4.1;


contract RequestTrackerInterface {
    function getWindowStart(address factory, address request) constant returns (uint);
    function getPreviousRequest(address factory, address request) constant returns␣
↪(address);
```

```
    function getNextRequest(address factory, address request) constant returns␣
↪(address);
    function addRequest(address request, uint startWindow) constant returns (bool);
    function removeRequest(address request) constant returns (bool);
    function isKnownRequest(address factory, address request) constant returns (bool);
    function query(address factory, bytes2 operator, uint value) constant 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 client.

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 0.4.1;


import {RequestScheduleLib} from "contracts/RequestScheduleLib.sol";
import {SchedulerLib} from "contracts/SchedulerLib.sol";


contract SchedulerInterface {
    using SchedulerLib for SchedulerLib.FutureTransaction;

    address public factoryAddress;

    RequestScheduleLib.TemporalUnit public temporalUnit;

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

    /*
     * When applied to a function, causes the local futureTransaction to
     * get reset to it's defaults on each function call.
     *
     * TODO: Compare to actual enum values when solidity compiler error is fixed.
     * https://github.com/ethereum/solidity/issues/1116
     */
    modifier doReset {
        if (uint(temporalUnit) == 1) {
            futureTransaction.resetAsBlock();
        } else if (uint(temporalUnit) == 2) {
            futureTransaction.resetAsTimestamp();
        } else {
            throw;
        }
        _
    }

    /*
     *  Full scheduling API exposing all fields.
     *
     *  uintArgs[0] callGas
     *  uintArgs[1] callValue
     *  uintArgs[2] windowSize
     *  uintArgs[3] windowStart
     *  bytes callData;
     *  address toAddress;
     */
    function scheduleTransaction(address toAddress,
                                 bytes callData,
                                 uint[4] uintArgs) doReset public returns (address);

    /*
     *  Full scheduling API exposing all fields.
     *
     *  uintArgs[0] callGas
     *  uintArgs[1] callValue
     *  uintArgs[2] donation
```

```
     *  uintArgs[3] payment
     *  uintArgs[4] requiredStackDepth
     *  uintArgs[5] windowSize
     *  uintArgs[6] windowStart
     *  bytes callData;
     *  address toAddress;
     */
    function scheduleTransaction(address toAddress,
                                 bytes callData,
                                 uint[7] uintArgs) doReset public returns (address);
}
```

9.3 Defaults

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

- `donationBenefactor`: `0xd3cda913deb6f67967b99d67acdfa1712c293601` which is the address of Piper Merriam, the creator of this project.
- `payment`: `1000000 * tx.gasprice` set at the time of scheduling.
- `donation`: `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.
- `requiredStackDepth`: 10 stack frames.

9.4 API

There are two `scheduleTransaction` methods on each `Scheduler` contract with different call signatures.

`Scheduler.`**`scheduleTransaction`**(*address toAddress*, *bytes callData*, *uint[4] uintArgs) returns (address*)

This method allows for configuration of the most common parameters needed for transaction scheduling. Due to EVM restrictions, all of the unsigned integer arguments are passed in as an array. The array values are mapped to the `TransactionRequest` attributes as follows.

- `uintArgs[0] => callGas`
- `uintArgs[1] => callValue`
- `uintArgs[2] => windowSize`
- `uintArgs[3] => windowStart`

`Scheduler.`**`scheduleTransaction`**(*address toAddress*, *bytes callData*, *uint[4] uintArgs) returns (address*)

This method presents three extra fields allowing more fine controll for transaction scheduling. Due to EVM restrictions, all of the unsigned integer arguments are passed in as an array. The array values are mapped to the `TransactionRequest` attributes as follows.

- `uintArgs[0] => callGas`

- `uintArgs[1] => callValue`
- `uintArgs[2] => donation`
- `uintArgs[3] => payment`
- `uintArgs[4] => requiredStackDepth`
- `uintArgs[5] => windowSize`
- `uintArgs[6] => windowStart`

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

CLI Interface

The alarm service ships with a command line interface that can be used to interact with the service in various ways.

10.1 Requirements

The `ethereum-alarm-clock-client` python package requires some system dependencies to install. Please see the pyethereum documentation for more information on how to install these.

This package is only tested against Python 3.5. It may work on other versions but they are explicitely not supported.

This package is only tested on unix based platforms (OSX and Linux). It may work on other platforms but they are explicitely not supported.

10.2 Installation

The `ethereum-alarm-clock-client` package can be installed using `pip` like this.

```
$ pip install ethereum-alarm-clock-client
```

Or directly from source like this.

```
$ python setup.py install
```

If you are planning on modifying the code or developing a new feature you should instead install like this.

```
$ python setup.py develop
```

10.3 The `eth_alarm` executable

Once you've installed the package you should have the `eth_alarm` executable available on your command line.

```
$ eth_alarm
Usage: eth_alarm [OPTIONS] COMMAND [ARGS]...

Options:
  -t, --tracker-address TEXT      The address of the RequestTracker contract
                                  that should be used.
  -f, --factory-address TEXT      The address of the RequestFactory contract
                                  that should be used.
  --payment-lib-address TEXT      The address of the PaymentLib contract that
                                  should be used.
  -r, --request-lib-address TEXT  The address of the RequestLib contract that
                                  should be used.
  -l, --log-level INTEGER         Integer logging level - 10:DEBUG 20:INFO
                                  30:WARNING 40:ERROR
  -p, --provider TEXT             Web3.py provider type to use to connect to
                                  the chain.  Supported values are 'rpc',
                                  'ipc', or any dot-separated python path to a
                                  web3 provider class
  --ipc-path TEXT                 Path to the IPC socket that the IPCProvider
                                  will connect to.
  --rpc-host TEXT                 Hostname or IP address of the RPC server
  --rpc-port INTEGER              The port to use when connecting to the RPC
                                  server
  -a, --compiled-assets-path PATH
                                  Path to JSON file which contains the
                                  compiled contract assets
  --back-scan-seconds INTEGER     Number of seconds to scan into the past for
                                  timestamp based calls
  --forward-scan-seconds INTEGER  Number of seconds to scan into the future
                                  for timestamp based calls
  --back-scan-blocks INTEGER      Number of blocks to scan into the past for
```

```
                                    block based calls
  --forward-scan-blocks INTEGER   Number of blocks to scan into the future for
                                    block based calls
  --help                            Show this message and exit.

Commands:
  client:monitor  Scan the blockchain for events from the alarm...
  client:run
  repl            Drop into a debugger shell with most of what...
  request:create  Schedule a transaction to be executed at a...
```

10.3.1 Rollbar Integration

Monitoring these sorts of things can be difficult. I am a big fan of the rollbar service which provides what I feel is a very solid monitoring and log management solution.

To enable rollbar logging with the `eth_alarm` client you'll need to do the following.

1. Install the python rollbar package. * `$ pip install rollbar`
2. Run `eth_alarm` with the following environment variables set. * `ROLLBAR_SECRET` set to the *server side* token that rollbar provides. * `ROLLBAR_ENVIRONMENT` set to a string such as *'production'* or *'ec2-instance-abcdefg''*.

10.4 Running a server

The scheduler runs nicely on the *small* AWS EC2 instance size. The following steps should get an EC2 instance provisioned with the scheduler running.

10.4.1 1. Setup an EC2 Instance

- Setup an EC2 instance running Ubuntu. The smallest instance size works fine.
- Add an extra volume to store your blockchain data. 20GB should be sufficient for a short while (after April 2017) if storing the entire history, block-for-block, is not required. Otherwise, a much larger size should be used.
- Optionally mark this volume to persist past termination of the instance so that you can reuse your blockchain data.
- Make sure that the security policy leaves *30303* open to connections from the outside world.

10.4.2 2. Provision the Server

- `sudo apt-get update --fix-missing`
- `sudo apt-get install -y supervisor`
- `sudo apt-get install -y python3-dev python build-essential libreadline-gplv2-dev libncursesw5-dev libssl-dev libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev python-virtualenv libffi-dev autoconf`

10.4.3 3. Mount the extra volume

The following comes from the AWS Documentation and will only work verbatim if your additional volume is `/dev/xvdb`.

- `sudo mkfs -t ext4 /dev/xvdb`
- `sudo mkdir -p /data`
- `sudo mount /dev/xvdb /data`
- `sudo mkdir -p /data/ethereum`
- `sudo chown ubuntu /data/ethereum`

Modify */etc/fstab* to look like the following. This ensures the extra volume will persist through restarts.

```
#/etc/fstab
LABEL=cloudimg-rootfs   /        ext4   defaults,discard        0 0
/dev/xvdb       /data   ext4    defaults,nofail        0        2
```

Run `sudo mount -a` If you don't get any errors then you haven't borked your `etc/fstab`

10.4.4 4. Install Geth or Parity

Install the go-ethereum client.

- `sudo apt-get install -y software-properties-common`
- `sudo add-apt-repository -y ppa:ethereum/ethereum`
- `sudo apt-get update`
- `sudo apt-get install -y ethereum`

or install the parity client.

- `bash <(curl https://get.parity.io -Lk)`

10.4.5 5. Install the Alarm Client

Install the Alarm client.

- `mkdir -p ~/alarm-0.8.0`
- `cd ~/alarm-0.8.0`
- `virtualenv -p /usr/bin/python3.5 env && source env/bin/activate`
- `pip install setuptools --upgrade`
- `pip install ethereum-alarm-clock-client==8.0.0b1`

10.4.6 6. Configure Supervisord

Supervisord will be used to manage both `geth` and `eth_alarm`.

If you are using Go-Ethereum, put the following in `/etc/supervisord/conf.d/geth.conf`

```
[program:geth]
command=geth --datadir /data/ethereum --unlock 0 --password /home/ubuntu/scheduler_
↪password --fast
user=ubuntu
stdout_logfile=/var/log/supervisor/geth-stdout.log
stderr_logfile=/var/log/supervisor/geth-stderr.log
autorestart=true
autostart=false
```

and the following in `/etc/supervisord/conf.d/scheduler-v8.conf`

```
[program:scheduler-v8]
user=ubuntu
command=/home/ubuntu/alarm-0.8.0/env/bin/eth_alarm --ipc-path /data/ethereum/geth.ipc␣
↪client:run
directory=/home/ubuntu/alarm-0.8.0/
environment=PATH="/home/ubuntu/alarm-0.8.0/env/bin"
stdout_logfile=/var/log/supervisor/scheduler-v8-stdout.log
stderr_logfile=/var/log/supervisor/scheduler-v8-stderr.log
autorestart=true
autostart=false
```

If you are using Parity, put the following in `/etc/supervisord/conf.d/parity.conf`

```
[program:parity]
command=parity --db-path /data/ethereum --unlock <your-account-address> --password /
↪home/ubuntu/scheduler_password
user=ubuntu
stdout_logfile=/var/log/supervisor/parity-stdout.log
stderr_logfile=/var/log/supervisor/parity-stderr.log
autorestart=true
autostart=false
```

and the following in `/etc/supervisord/conf.d/scheduler-v8.conf`

```
[program:scheduler-v8]
user=ubuntu
command=/home/ubuntu/alarm-0.8.0/env/bin/eth_alarm --ipc-path /home/ubuntu/.parity/
↪jsonrpc.ipc client:run
directory=/home/ubuntu/alarm-0.8.0/
environment=PATH="/home/ubuntu/alarm-0.8.0/env/bin"
stdout_logfile=/var/log/supervisor/scheduler-v8-stdout.log
stderr_logfile=/var/log/supervisor/scheduler-v8-stderr.log
autorestart=true
autostart=false
```

10.4.7 7. Generate an account

For Go-Ethereum

- `$ geth --datadir /data/ethereum account new`

For parity

- `$ parity account new`

Place the password for that account in `/home/ubuntu/scheduler_password`.

You will also need to send this account a few ether. A few times the maximum transaction cost should be sufficient as this account should always trend upwards as it executes requests and receives payment for them.

Don't forget to back up the key file! Go-Ethereum should have put it in

- `/data/ethereum/keystore/`

and Parity in

- `/home/ubuntu/.local/share/io.parity.ethereum/keys/`

10.4.8 8. Turn it on

Reload supervisord so that it finds the two new config files.

- `sudo supervisord reload`

You'll want to wait for Go-Ethereum or Parity to fully sync with the network before you start the `scheduler-v8` process.

10.4.9 9. Monitoring

You can monitor these processes with `tail`

- `tail -f /var/log/supervisor/geth*.log`
- `tail -f /var/log/supervisor/parity*.log`
- `tail -f /var/log/supervisor/scheduler-v8*.log`

10.4.10 10. System Clock

You might want to add the following line to your crontab. This keeps your system clock up to date. I've had issues with my servers *drifting*.

```
0 0 * * * /usr/sbin/ntpdate ntp.ubuntu.com
```

CHAPTER 11

Changelog

11.1 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.

11.2 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.

11.3 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.

11.4 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)

11.5 0.4.0

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

11.6 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.

11.7 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.

11.8 0.1.0

- Initial release.

Index