Ethereum Alarm Clock Documentation

Release 1.0.0

Piper Merriam

November 08, 2015

Contents

The Ethereum Alarm Clock is a service that allows scheduling of contract function calls at a specified block number in the future. These scheduled calls are then executed by other nodes on the ethereum network who are reimbursed for their gas costs plus a small payment for the transaction.

Contents:

CHAPTER 1

Overview

The Ethereum Alarm service is a contract on the ethereum network that facilitates scheduling of function calls for a specified block in the future. It is designed to require little or no trust between any of the users of the service, as well as providing no special access to the creator of the contract.

1.1 Scheduling Function Calls

When a contract, or individual wants to schedule a function call with the Alarm service it will perform the following steps.

1. Schedule the function call with the service.
2. Register any call data that will be required to make the function call (optional for functions that have no arguments).

1.1.1 Call Scheduling

Function calls can be scheduled for any block at least 40 blocks *(~10 minutes)* in the future. Scheduling is done by providing the Alarm service with the following information:

1. Contract address the call should be executed on.
2. ABI signature of the function that should be called.
3. Target block number that the call should be executed on.

Optionally, these additional pieces of information can be supplied.

- Suggested gas amount that should be provided to the function. **default: 0 to indicate no suggestion**
- Number of blocks after the target block during which it still ok to execute the call. (between 64 - 255 blocks) **default: 255**
- Payment amount in wei that will be paid to the executor of the call. **default: 1 ether**
- Fee amount in wei that will be paid to the creator of the Alarm service. **default: 100 finey**

The scheduling transaction must also include enough ether to pay for the gas costs of the call as well as the payment and fee values.

Once scheduled, the call waits to be picked up and executed at the desired block.

1.1.2 Registering Call Data

The Alarm service is not aware of the function ABI for the calls it executes. Instead, it uses the function ABI signature and raw call data to execute the function call.

To do this, any data that needs to be used in the call must be registered prior to scheduling the call. Functions which do not have any arguments can skip this step.

1.2 Execution of scheduled calls

Scheduled function calls can ultimately be executed by anyone who wishes to initiate the transaction. This will likely be an automated process that monitors for upcoming scheduled calls and executes them at the appropriate block.

1.2.1 Usage Fees

In addition to the gas costs, schedulers are also encouraged to pay the call executor as well as the creator of the service for their effort. This value can be specified by the scheduler, meaning that you may choose to offer any amount (including zero).

The scheduling function uses the following defaults if specific values are not provided.

- Payment to the executor: **1 ether**
- Payment to the service creator: **100 finney**

1.3 Guarantees

1.3.1 Will the call happen?

There are no guarantees that your function will be called. This is not a shortcoming of the service, but rather a fundamental limitation of the ethereum network. Nobody is capable of forcing a transaction to be included in a specific block.

The design of this service is meant to provide the proper motivation for calls to be executed, but it is entirely possible that certain calls will be missed due to unforseen circumstances. Providing a higher Payment amount is a potential way to get your scheduled call handled at a higher priority as it will be more profitable to execute.

1.3.2 Will I get paid for executing a call?

If you are diligent about how you go about executing scheduled calls then executing scheduled calls is guaranteed to be profitable. See the section on executing calls for more information.

CHAPTER 2

Scheduling

Call scheduling is the core of the Ethereum Alarm Service. Calls can be scheduled on any block at least 40 blocks *(~10 minutes)* in the future.

When a call is scheduled, the service deploys a new contract that represents the scheduled function call. This contract is referred to as the **call contract**. holds all of the metadata associated with the call as well as the funds that will be used to pay for the call.

2.1 Lifecycle of a Call Contract

- **creation** - The call contract is created as part of call scheduling.
- **data-registered** - The data for the call is registered with the contract. This step can be skipped for function calls that take no arguments.
- **locked** - The contract is locked, preventing cancellation starting 10 blocks before the call's target block through the last block in the call window.
- **execution** - The executing transaction is sent, which triggers the call contract to execute the function call.
- **payment** - payments are sent to the executor and the creator of the alarm service.
- **suicide** - The contract suicides itself, sending the remainder of it's funds to the call scheduler.

2.2 Scheduling the Call

Function calls are scheduled with the `scheduleCall` function on the Alarm service. This creates a new **call contract** that represents the function call.

- **Solidity Function Signature:** `function scheduleCall(address contractAddress, bytes4 abiSignature, uint targetBlock, uint suggestedGas, uint8 gracePeriod, uint basePayment, uint baseFee) public returns (address)`
- **ABI Signature:** `TODO`

The `scheduleCall` function takes the following parameters:

Required Arguments

- **address contractAddress:** The contract address that the function should be called on.
- **bytes4 abiSignature:** The 4 byte ABI function signature for the call.

- **uint targetBlock:** The block number the call should be executed on.

Optional Arguments

- **uint suggestedGas:** A suggestion to the call executor as to how much gas should be provided to execute the call. (default: 0)
- **uint8 gracePeriod:** The number of blocks after `targetBlock` that it is ok to still execute this call. Cannot be less than 64. (default: 255)
- **uint basePayment:** The base amount in wei that should be paid to the executor of the call. (default: 1 ether)
- **uint baseFee:** The base amount in wei that should be paid to the creator of the alarm service. (default: 100 finney)

The optional arguments are implemented through the following alternate invocation signatures. The default value for each of the optional arguments will be used if any of the following signatures are used.

- **Solidity Function Signature:** `function scheduleCall(address contractAddress, bytes4 abiSignature, uint targetBlock) public returns (address)`
- **ABI Signature:** `TODO`
- **Solidity Function Signature:** `function scheduleCall(address contractAddress, bytes4 abiSignature, uint targetBlock, uint suggestedGas) public returns (address)`
- **ABI Signature:** `TODO`
- **Solidity Function Signature:** `function scheduleCall(address contractAddress, bytes4 abiSignature, uint targetBlock, uint suggestedGas, uint8 gracePeriod) public returns (address)`
- **ABI Signature:** `TODO`
- **Solidity Function Signature:** `function scheduleCall(address contractAddress, bytes4 abiSignature, uint targetBlock, uint suggestedGas, uint8 gracePeriod, uint basePayment) public returns (address)`
- **ABI Signature:** `TODO`

If the `scheduleCall` function is being used from within a contract, the address of the newly created call contract is returned. If instead, the function is being called directly in a transaction, the address of the call contract can be extracted from the transaction logs under the `CallScheduled` event.

2.2.1 Contract scheduling its own call

Contracts can take care of their own call scheduling.

```
contract Lottery {
    address alarm; // set by some other mechanism.

    function beginLottery() public {
        ... // Do whatever setup needs to take place.

        // Now we schedule the picking of the winner.

        bytes4 sig = bytes4(sha3("pickWinner()"));
        // approximately 24 hours from now
        uint targetBlock = block.number + 5760;
        // 0xTODO is the ABI signature computed from `bytes4(sha3("scheduleCall(...)"))`.
        alarm.call(TODO, address(this), sig, targetBlock)
```

```
    }

    function pickWinner() public {
        ...
    }
}
```

In this example `Lottery` contract, every time the `beginLottery` function is called, a call to the `pickWinner` function is scheduled for approximately 24 hours later (5760 blocks).

2.2.2 Scheduling a call for a contract

Alternatively, calls can be scheduled to be executed on other contracts

Lets look at an example where we want to schedule a funds transfer for a wallet contract of some sort.

Note: This example assuming that you have the Alarm contract ABI loaded into a web3 contract object.

```
// Now schedule the call
> signature = ... // the 4-byte ABI function signature for the wallet function that transfers funds.
> targetBlock = eth.getBlock('latest') + 100  // 100 blocks in the future.
> alarm.scheduleCall.sendTransaction(walletAddress, signature, targetBlock, {from: eth.coinbase})
```

2.3 Registering Call Data

If a function call requires arguments then it is up to the scheduler to register the call data prior to call execution.

The call contract allows for call data registration via two mechanisms. The primary mechanism is through the fallback function on the contract. This will set the call data as the full call data of the transaction.

```
// Register some call data
> web3.eth.sendTransaction({to: scheduler.address, data: "0x...."})
```

Or, from within your contract.

```
contract Lottery {
    address alarm; // set by some other mechanism.

    function beginLottery() public {
        uint lotteryId = ...;

        // Now we schedule the picking of the winner.
        bytes4 sig = bytes4(sha3("pickWinner(uint256)"));
        // 0x1991313 is the ABI signature computed from `bytes4(sha3("scheduleCall(address,bytes4,ui
        alarm.call(0x1991313, address(this), sig, 100)

        // Register the call data
        alarm.call(lotteryId);
    }

    function pickWinner(uint lotteryId) public {
        ...
    }
}
```

If however, your call data either has a `bytes4` value as it's first argument, or, the first 4 bytes of the call data have a collision with one of the existing function signatures on the call contract, you can use the `registerData` function instead.

- **Solidity Function Signature:** `registerData()`
- **ABI Signature:** `0xb0f07e44`

In solidity, this would look something like the following.

Upon receiving this call, the Alarm service strips off the first four bytes from `msg.data` to remove the ABI function signature and then stores the full call data.

Once data has been registered, it cannot be modified. Attempts to do so will result in an exception.

2.3.1 ABI Encoding and address.call

The `call()` function on an address in solidity does not do any ABI encoding, so in cases where a scheduled call must pass something like a `bytes` variable, you will need to handle the ABI encoding yourself.

2.4 Cancelling a call

A scheduled call can be cancelled by its scheduler up to 10 blocks before it's target block. To cancel a scheduled call use the `cancel` function.

- **Solidity Function Signature:** `cancel(address callAddress)`
- **ABI Signature:** `0xea8a1af0`

This will result in the call contract suiciding, and sending any remaining funds to the scheduler's address.

2.5 Looking up a Call

You can lookup whether a particular address is a known scheduled call with the `isKnownCall` function.

- **Solidity Function Signature:** `isKnownCall(address callAddress) returns (bool)`
- **ABI Signature:** `0x523ccfa8`

Returns a boolean as to whether this address represents a known scheduled call.

CHAPTER 3

Call Contract API

3.1 Properties of a Call Contract

A call contract for a scheduled call has the following publicly accessible values.

- **address contractAddress:** the address of the contract the function should be called on.
- **address schedulerAddress:** the address who scheduled the call.
- **uint targetBlock:** the block that the function should be called on.
- **uint8 gracePeriod:** the number of blocks after the `targetBlock` during which it is stll ok to execute the call.
- **uint anchorGasPrice:** the gas price that was used when the call was scheduled.
- **uint suggestedGas:** a suggestion to the call executor as to how much gas the called function is expected to need.
- **uint basePayment:** the amount in wei that will be paid to the address that executes the function call.
- **uint baseFee:** the amount in wei that will be paid the creator of the Alarm service.
- **bytes4 abiSignature:** the 4 byte ABI function signature of the function on the `contractAddress` for this call.
- **bytes callData:** the data that will be passed to the called function.

3.1.1 Contract Address

address contractAddress

The address of the contract that the scheduled function call should be executed on. Retrieved with the `contractAddress` function.

- **Solidity Function Signature:** `contractAddress() returns (address)`
- **ABI Signature:** `0xf6b4dfb4`

3.1.2 Scheduler Address

address schedulerAddress

The address that the scheduled function call. Retrieved with the `schedulerAddress` function.

- **Solidity Function Signature:** `schedulerAddress() returns (address)`
- **ABI Signature:** `0xae45850b`

3.1.3 Target Block

uint targetBlock

The block number that this call should be executed on. Retrieved with the `targetBlock` function.

- **Solidity Function Signature:** `targetBlock() returns (uint)`
- **ABI Signature:** `0xa16697a`

3.1.4 Grace Period

uint8 gracePeriod

The number of blocks after the **targetBlock** that it is still ok to execute this call. Retrieved with the `gracePeriod` function.

- **Solidity Function Signature:** `gracePeriod() returns (uint8)`
- **ABI Signature:** `0xa06db7dc`

3.1.5 Anchor Gas Price

uint anchorGasPrice

The value of `tx.gasprice` that was used to schedule this function call. Retrieved with the `anchorGasPrice` function.

- **Solidity Function Signature:** `anchorGasPrice() returns (uint)`
- **ABI Signature:** `0x37f4c00e`

3.1.6 Suggested Gas

uint suggestedGas

A suggestion for the amount of gas that a caller should expect the called function to require. Retrieved with the `suggestedGas` function.

- **Solidity Function Signature:** `suggestedGas() returns (uint)`
- **ABI Signature:** `0x6560a307`

3.1.7 Base Payment

uint basePayment

The base amount, in wei that the call executor's payment will be calculated from. Retrieved with the `basePayment` function.

- **Solidity Function Signature:** `basePayment() returns (uint)`
- **ABI Signature:** `0xc6502da8`

3.1.8 Base Fee

uint baseFee

The base amount, in wei that the fee to the creator of the alarm service will be calculate from. Retrieved with the `baseFee` function.

- **Solidity Function Signature:** `baseFee() returns (uint)`
- **ABI Signature:** `0x6ef25c3a`

3.1.9 ABI Signature

bytes4 abiSignature

The ABI function signature that should be used to execute this function call. Retrieved with the `abiSignature` function.

- **Solidity Function Signature:** `abiSignature() returns (uint)`
- **ABI Signature:** `0xca94692d`

3.1.10 Call Data

bytes callData

The full call data that will be used for this function call. Retrieved with the `callData` function.

- **Solidity Function Signature:** `callData() returns (bytes)`
- **ABI Signature:** `0x4e417a98`

3.2 Functions of a Call Contract

3.2.1 Cancel

Cancels the scheduled call, suiciding the call contract and sending any funds to the scheduler's address. This function cannot be called from 10 blocks prior to the **target block** for the call through the end of the grace period.

- **Solidity Function Signature:** `cancel() public onlyscheduler`
- **ABI Signature:** `0xea8a1af0`

3.2.2 Is Alive

Always returns `true`. Useful to check if the contract has been suicided.

CHAPTER 4

Caller Pool

The Alarm service maintains a pool of bonded callers who are responsible for executing scheduled calls. By joining the caller pool, an account is committing to executing scheduled calls in a reliable and consistent manner. Any caller who reliably executes the calls which are allocated to them will make a consistent profit from doing so, while callers who don't get removed from the pool and forfeit some or all of their bond.

4.1 Caller Bonding

In order to execute scheduled calls, callers put up a small amount of ether up front. This bond, is held for the duration that a caller remains in the caller pool.

4.1.1 Minimum Bond

The bond amount is set as the maximum allowed transaction cost for a given block. This value can be retrieved with the `getMinimumBond` function.

- **Solidity Function Signature:** `getMinimumBond() returns (uint)`
- **ABI Signature:** `0x23306ed6`

This value can change from block to block depending on the gas price and gas limit.

4.1.2 Depositing your bond

Use the `depositBond` function on the Caller Pool to deposit ether towards your bond.

- **Solidity Function Signature:** `depositBond()`
- **ABI Signature:** `0x741b3c39`

4.1.3 Checking bond balance

Use the `getBondBalance` function to check the bond balance of an account.

- **Solidity Function Signature:** `getBondBalance(address) returns (uint)`
- **ABI Signature:** `0x33613cbe`

Or to just check the balance of the sender of the transaction.

- **Solidity Function Signature:** `getBondBalance(address) returns (uint)`

- **ABI Signature:** `0x3cbfed74`

4.1.4 Withdrawing your bond

Use the `withdrawBond` function on the Caller Pool to withdraw the bond ether.

- **Solidity Function Signature:** `withdrawBond()`
- **ABI Signature:** `0xc3daab96`

If you are currently in a call pool, either active or queued, you will not be able to withdraw your account balance below the minimum bond amount.

4.2 Bond Forfeiture

In the event that a caller fails to execute a scheduled call during their allocated call window, a portion of their bond is forfeited and they are removed from the caller pool. The amount forfeited is equal to the current minimum bond amount.

There are no restrictions on re-entering the caller pool as long as a caller is willing to put up a new bond.

4.3 About Pools

The caller pool maintains a lists of caller addresses. Whenever a change is made to the pool, either addition of a new member or removal of an existing member, a new generation is queued to take place of the current generation.

The new generation will be set to begin 160 blocks in the future. During the first 80 blocks of this window, other members may leave or join the generation. The new generation will be frozen for the 80 blocks leading up to it's starting block. Each new generation overlaps the previous generation by 256 blocks.

For example, if we are currently on block 1000, and a member exits the pool.

- The new generation will become active at block 1160
- The new generation will not allow any membership changes after block 1080
- The current generation will be set to end at block 1416 (1160 + 256)

Each new generation carries over all of the previous members upon creation.

Once the queued generation becomes active, members are once again allowed to enter and exit the pool.

Each time a new generation is created, the ordering of its members is shuffled.

Note: It is worth pointing out that from the block during which you exit the pool, you must still execute the calls that are allocated to you until the current generation has ended. Failing to do so will cause bond forfeiture.

4.4 Entering the Pool

An address can enter the caller pool if the following conditions are met.

- The caller has deposited the minimum bond amount into their bond account.
- The caller is not already in the active pool, or the next queued pool.

- The next queued pool does not go active within the next 80 blocks.

To enter the pool, call the `enterPool` function on the Caller Pool.

- **Solidity Function Signature:** `enterPool()`
- **ABI Signature:** `0x50a3bd39`

If the appropriate conditions are met, you will be added to the next caller pool. This will create a new generation if one has not already been created. Otherwise you will be added to the next queued generation.

You can use the `canEnterPool` function to check whether a given address is currently allowed to enter the pool.

- **Solidity Function Signature:** `canEnterPool(address callerAddress) returns (bool)`
- **ABI Signature:** `0x8dd5e298`

Or to to check for the address sending the transaction.

- **Solidity Function Signature:** `canEnterPool() returns (bool)`
- **ABI Signature:** `0xc630f92b`

4.5 Exiting the Pool

An address can exit the caller pool if the following conditions are met.

- The caller is in the current active generation.
- The caller has not already exited or been removed from the queued pool (if it exists)
- The next queued pool does not go active within the next 80 blocks.

To exit the pool, use the `exitPool` function on the Caller Pool.

- **Solidity Function Signature:** `exitPool()`
- **ABI Signature:** `0x50a3bd39`

If all conditions are met, a new caller pool will be queued if one has not already been created and your address will be removed from it.

You can use the `canExitPool` function to check whether a given address is currently allowed to exit the pool.

- **Solidity Function Signature:** `canExitPool(address callerAddress) returns (bool)`
- **ABI Signature:** `0xb010d94a`

Alernatively, you can check for the address sending the transaction.

- **Solidity Function Signature:** `canExitPool(address callerAddress) returns (bool)`
- **ABI Signature:** `0x5a5383ac`

CHAPTER 5

Call Execution

Call execution is the process through which scheduled calls are executed at their desired block number. After a call has been scheduled, it can be executed by account which chooses to initiate the transaction. In exchange for executing the scheduled call, they are paid a small fee of approximately 1% of the gas cost used for executing the transaction.

5.1 Executing a call

Use the `execute` function to execute a scheduled call.

- **Solidity Function Signature:** `execute(address callAddress)`
- **ABI Signature:** `0xfcf36918`

When this function is called, the following things happen.

1. A few checks are done to be sure that all of the necessary pre-conditions pass. If any fail, the function exits early without executing the scheduled call:
    - the call has not already been suicided
    - the current block number is within the range this call is allowed to be executed.
    - the caller is allowed to execute the function (see caller pool)
2. The call is executed
3. The gas cost and fees are computed and paid.
4. The call contract suicides, sending any remaining funds to the scheduling address.

5.1.1 Payment

Each scheduled call sets its own payment value. This can be looked up with the `basePayment` accessor function.

The final payment value for executing the scheduled call is the `basePayment` multiplied by a scalar value based on the difference between the gas price of the executing transaction and the gas price that was used to schedule the transaction. The formula for this scalar is such that the lower the gas price of the executing transaction, the higher the payment.

5.1.2 Setting transaction gas and gas price

Each call contract has a `suggestedGas` property that can be used as a suggestion for how much gas the function call needs. In the case where this is set to zero it means the scheduler has not provided a suggestion.

This suggested gas value should be used in conjuction with the `basePayment` and `baseFee` amounts with respect to the ether balance of the call contract. The provided gas for the transaction should not exceed `(balance - 2 * (basePayment + baseFee)) / gasPrice` if you want to guarantee that you will be fully reimbursed for gas expenditures.

5.1.3 Getting your payment

Payment for executing a call is sent to you as part of the executing transaction, as well as being logged by the `CallExecuted` event.

5.2 Determining what scheduled calls are next

You can query the Alarm service for the call key of the next scheduled call on or after a specified block number using the `getNextCall` function

- **Solidity Function Signature:** `getNextCall(uint blockNumber) returns (address)`
- **ABI Signature:** `0x9f927be7`

Since there may be multiple calls on the same block, it is best to also check if the call has any *siblings* using the `getNextCallSibling` function. This function takes a call contract address and returns the address that is scheduled to come next.

When checking for additional calls in this manner, you should check the target block of each subsequent call to be sure it is within a range that you care about.

- **Solidity Function Signature:** `getNextCallSibling(address callAddress) returns (address)`
- **ABI Signature:** `0x48107843`

Note: 40 blocks into the future is a good range to monitor since new calls must always be scheduled at least 40 blocks in the future. You should also monitor these functions up to 10 blocks before their target block to be sure they are not cancelled.

5.3 Designated Callers

If the Caller Pool has any bonded callers in the current active pool, then only designated callers will be allowed to execute a scheduled call. The exception to this restriction is the last few blocks within the call's grace period which the call enters *free-for-all* mode during which anyone may execute it.

If there are no bonded callers in the Caller Pool then the Alarm service will operate in *free-for-all* mode for all calls meaning anyone may execute any call at any block during the call window.

5.3.1 How callers designated

Each call has a window during which it is allowed to be executed. This window begins at the specified `targetBlock` and extends through `targetBlock + gracePeriod`. This window is inclusive of it's bounding blocks.

For each 16 block section of the call window, the caller pool associated with the `targetBlock` is selected. The members of the pool can be though of as a circular queue, meaning that when you iterate through them, when you reach the last member, you start back over at the first member. For each call, a random starting position is selected in the member queue and the 16 block sections of the call window are assigned in order to the membes of the call pool beginning at this randomly chosen index..

The last two 16 block sections (17-32 blocks depending on the gracePeriod) are not allocated, but are considered *free-for-all* allowing anyone to call.

Use the `getDesignatedCaller` function to determine which caller from the caller pool has been designated for the block.

- **Solidity Function Signature:** `getDesignatedCaller(address callAddress, uint256 blockNumber) returns (bool, address)`
- **ABI Signature:** `0x5a8dd79f`
- **callAddress:** specifies the address of the call contract.
- **blockNumber:** the block number (during the call window) in question.

This returns a boolean and an address. The boolean designates whether this scheduled call was designated (if there are no registered caller pool members then all calls operate in free-for-all mode). The address is the designated caller. If the returned address is `0x0` then this call can be executed by anyone on the provided block number.

5.3.2 Missing the call window

Anytime a caller fails to execute a scheduled call during the 4 block window reserved for them, the next caller has the opportunity to claim a portion of their bond merely by executing the call during their window. When this happens, the previous caller who missed their call window has the current minimum bond amount deducted from their bond balance and transferred to the caller who executed the call. The caller who missed their call is also removed from the pool. This removal takes 416 blocks to take place as it occurs within the same mechanism as if they removed themselves from the pool.

5.3.3 Free For All

When a call enters the last two 16-block chunks of its call window it enters free-for-all mode. During these blocks anyone, even unbonded callers, can execute the call. The sender of the executing transaction will be rewarded the bond bonus from all callers who missed their call window.

5.4 Safeguards

There are a limited set of safeguards that Alarm protects those executing calls from.

- Enforces the ability to pay for the maximum possible transaction cost up front.
- Ensures that the call cannot cause the executing transaction to fail due to running out of gas (like an infinite loop).
- Ensures that the funds to be used for payment are locked during the call execution.

5.5 Tips for executing scheduled calls

The following tips may be useful if you wish to execute calls.

5.5.1 Only look in the next 40 blocks

Since calls cannot be scheduled less than 40 blocks in the future, you can count on the call ordering remaining static for the next 40 blocks.

5.5.2 No cancellation in next 8 blocks

Since calls cannot be cancelled less than 8 blocks in the future, you don't need to check cancellation status during the 8 blocks prior to its target block.

5.5.3 Check that it was not already called

If you are executing a call after the target block but before the grace period has run out, it is good to check that it has not already been called.

5.5.4 Check that the scheduler can pay

It is good to check that the scheduler has sufficient funds to pay for the call's potential gas cost plus fees.

CHAPTER 6

Call pricing and fees

The Alarm service operates under a **scheduler pays** model, which means that the scheduler of a call is responsible for paying for the full gas cost and fees associated with executing the call.

These funds must be presented upfront at the time of scheduling and are held by the call contract until execution.

6.1 Call Payment and Fees

When a call is scheduled, the scheduler can either provide values for the payment and fee, or leave them off in favor of using the default values.

The account which executes the scheduled call is reimbursed 100% of the gas cost + payment for their service as well as sending the fee to the creator of the service.

6.1.1 The GasPriceScalar multiplier

Both the payment and the fee are multiplied by the **GasPriceScalar**.

GasPriceScalar is a multiplier that ranges from 0 - 2 which is based on the difference between the gas priced used for call execution and the gas price used during call scheduling. This number incentivises the call executor to use as low a gas price as possible.

This multiplier is computed with the following formula.

- *IF* `gasPrice > anchorGasPrice`

```
anchorGasPrice / gasPrice
```

- *IF* `gasPrice <= anchorGasPrice`

```
anchorGasPrice / (2 * anchorGasPrice - gasPrice)
```

Where:

- **anchorGasPrice** is the `tx.gasprice` used when the call was scheduled.
- **gasPrice** is the `tx.gasprice` used to execute the call.

At the time of call execution, the `anchorGasPrice` has already been set, so the only value that is variable is the `gasPrice` which is set by the account executing the transaction. Since the scheduler is the one who ends up paying for the actual gas cost, this multiplier is designed to incentivize the caller using the lowest gas price that can be expected to be reliably picked up and promptly executed by miners.

Here are the values this formula produces for a `baseGasPrice` of 20 and a `gasPrice` ranging from 10 - 40;

gasPrice	multiplier
15	1.20
16	1.17
17	1.13
18	1.09
19	1.05
20	1.00
21	0.95
22	0.91
23	0.87
24	0.83
25	0.80
26	0.77
27	0.74
28	0.71
29	0.69
30	0.67
31	0.65
32	0.63
33	0.61
34	0.59
35	0.57
36	0.56
37	0.54
38	0.53
39	0.51
40	0.50

You can see from this table that as the `gasPrice` for the executing transaction increases, the total payout for executing the call decreases. This provides a strong incentive for the entity executing the transaction to use a reasonably low value.

Alternatively, if the `gasPrice` is set too low (potentially attempting to maximize payout) and the call is not picked up by miners in a reasonable amount of time, then the entity executing the call will not get paid at all. This provides a strong incentive to provide a value high enough to ensure the transaction will be executed.

6.2 Overhead

The gas overhead that you can expect to pay for your function is about 100,000 gas.

CHAPTER 7

Contract ABI

Beyond the simplest use cases, the use of `address.call` to interact with the Alarm service is limiting. Beyond the readability issues, it is not possible to get the return values from function calls when using `call()`.

By using an abstract solidity contract which defines all of the function signatures, you can easily call any of the Alarm service's functions, letting the compiler handle computation of the function ABI signatures.

7.1 Abstract Solidity Contracts

The following abstract contracts can be used alongside your contract code to interact with the Alarm service.

7.1.1 Abstract Scheduler Contract Source Code

The following abstract solidity contract can be used to interact with the scheduling contract from a solidity contract.

```
contract SchedulerAPI {
    /*
     *  Call Scheduling API
     */
    function getMinimumGracePeriod() constant returns (uint);
    function getDefaultPayment() constant returns (uint);
    function getDefaultFee() constant returns (uint);

    function isKnownCall(address callAddress) constant returns (bool);

    function scheduleCall(address contractAddress, bytes4 abiSignature, uint targetBlock) public retu
    function scheduleCall(address contractAddress, bytes4 abiSignature, uint targetBlock, uint sugges
    function scheduleCall(address contractAddress, bytes4 abiSignature, uint targetBlock, uint sugges
    function scheduleCall(address contractAddress, bytes4 abiSignature, uint targetBlock, uint sugges
    function scheduleCall(address contractAddress, bytes4 abiSignature, uint targetBlock, uint sugges

    /*
     *  Call Execution API
     */
    function execute(address callAddress) public;

    /*
     *  Caller Pool bonding
     */
    function getMinimumBond() constant returns (uint);
    function depositBond() public;
```

```
    function withdrawBond(uint value) public;
    function getBondBalance() constant returns (uint);
    function getBondBalance(address callerAddress) constant returns (uint);

    /*
     *  Caller Pool Membership
     */
    function getGenerationForCall(bytes32 callKey) constant returns (uint);
    function getGenerationSize(uint generationId) constant returns (uint);
    function getGenerationStartAt(uint generationId) constant returns (uint);
    function getGenerationEndAt(uint generationId) constant returns (uint);
    function getCurrentGenerationId() constant returns (uint);
    function getNextGenerationId() constant returns (uint);
    function isInPool() constant returns (bool);
    function isInPool(address callerAddress) constant returns (bool);
    function isInGeneration(uint generationId) constant returns (bool);
    function isInGeneration(address callerAddress, uint generationId) constant returns (bool);

    /*
     *  Caller Pool Metadata
     */
    function getPoolFreezePeriod() constant returns (uint);
    function getPoolOverlapSize() constant returns (uint);
    function getPoolRotationDelay() constant returns (uint);

    /*
     *  Caller Pool Entering and Exiting
     */
    function canEnterPool() constant returns (bool);
    function canEnterPool(address callerAddress) constant returns (bool);
    function canExitPool() constant returns (bool);
    function canExitPool(address callerAddress) constant returns (bool);
    function enterPool() public;
    function exitPool() public;

    /*
     *  Next Call API
     */
    function getCallWindowSize() constant returns (uint);
    function getGenerationIdForCall(address callAddress) constant returns (uint);
    function getDesignatedCaller(address callAddress, uint blockNumber) constant returns (bool, addre
    function getNextCall(uint blockNumber) constant returns (bytes32);
    function getNextCallSibling(address callAddress) constant returns (bytes32);
}
```

7.1.2 Abstract Call Contract Source Code

The following abstract solidity contract can be used to interact with a call contract from a solidity contract.

```
contract CallContractAPI {
    uint public targetBlock;
    uint8 public gracePeriod;

    address public owner;
    address public schedulerAddress;

    uint public basePayment;
```

```
    uint public baseFee;

    function contractAddress() constant returns (address);
    function abiSignature() constant returns (bytes4);
    function callData() constant returns (bytes);
    function anchorGasPrice() constant returns (uint);
    function suggestedGas() constant returns (uint);

    function isAlive() constant public;

    // cancel and registerData are only callable by the scheduler of the
    call contract.
    function cancel() public onlyscheduler;
    function registerData() public onlyscheduler;
}
```

7.1.3 Only use what you need

The contracts above have stub functions for every API exposed by Alarm and CallerPool. It is safe to remove any functions or events from the abstract contracts that you do not intend to use.

CHAPTER 8

Caller Pool API

The Caller Pool contract exposes the following api functions.

8.1 Bond Management

The following functions are available for managing the ether deposited as a bond with the Caller Pool.

8.1.1 Get Minimum Bond

Use the `getMinimumBond` function to retrieve the current minimum bond value required to be able to enter the caller pool.

- **Solidity Function Signature:** `getMinimumBond() returns (uint)`
- **ABI Signature:** `0x23306ed6`

8.1.2 Check Bond Balance

Use the `getBondBalance` function to check the bond balance for the provided address.

- **Solidity Function Signature:** `getBondBalance(address callerAddress) returns (uint)`
- **ABI Signature:** `0x33613cbe`

Or to check the balance of the sender of the transaction.

- **Solidity Function Signature:** `getBondBalance() returns (uint)`
- **ABI Signature:** `0x3cbfed74`

8.1.3 Deposit Bond

Use the `depositBond` function to deposit you bond with the caller pool.

- **Solidity Function Signature:** `depositBond()`
- **ABI Signature:** `0x741b3c39`

8.1.4 Withdraw Bond

Use the `withdrawBond` function to withdraw funds from your bond.

- **Solidity Function Signature:** `withdrawBond()`
- **ABI Signature:** `0xc3daab96`

When in either an active or queued caller pool, you cannot withdraw your account below the minimum bond value.

8.2 Call Scheduling and Execution

The following function is available for callers.

8.2.1 Get Designated Caller

Use the `getDesignatedCaller` function to retrieve which caller address, if any, is designated as the caller for a given block and scheduled call.

- **Solidity Function Signature:** `getDesignatedCaller(bytes32 callKey, uint256 blockNumber)`
- **ABI Signature:** `0x3c941423`
- **callKey:** specifies the scheduled call.
- **blockNumber:** the block number (during the call window) in question.

This returns the address of the caller who is designated for this block, or `0x0` if this call can be executed by anyone on the specified block.

8.3 Pool Information

The following functions are available to query information about call pools.

8.3.1 Pool Generations

Use the `getCurrentGenerationId` function to lookup the id of the pool generation that is currently active. (returns 0 if no generations exist)

- **Solidity Function Signature:** `getCurrentGenerationId() returns (uint)`
- **ABI Signature:** `0xb0171fa4`

Use the `getNextGenerationId` function to lookup the generation that is queued to become active. Returns `0x0` if there is no next generation queued.

- **Solidity Function Signature:** `getNextGenerationId() returns (uint)`
- **ABI Signature:** `0xa502aae8`

Use the `getGenerationStartAt` function to lookup the block on which a given generation will become active.

- **Solidity Function Signature:** `getGenerationStartAt(uint generationId) returns (uint)`
- **ABI Signature:** `0xf8b11853`

Use the `getGenerationEndAt` function to lookup the block on which a given generation will end and become inactive. Returns `0` if the generation is still open ended.

- **Solidity Function Signature:** `getGenerationEndAt(uint generationId) returns (uint)`
- **ABI Signature:** `0x306b031d`

Use the `getGenerationSize` function to query the number of members in a given generation.

- **Solidity Function Signature:** `getGenerationSize(uint generationId) returns (uint)`
- **ABI Signature:** `0xb3559460`

8.3.2 Get Pool Key for Block

Use the `getGenerationIdForCall` function to return the `generationId` that should be used for the given call key. This can be helpful to determine whether your call execution script should pay attention to specific calls if you are in the process of entering or exiting the pool.

- **Solidity Function Signature:** `getGenerationIdForCall(bytes32 callKey) returns (uint)`
- **ABI Signature:** `0xdb681e54`

8.4 Pool Membership

The following functions can be used to query about an address's pool membership.

8.4.1 Is In Pool

Use the `isInPool` function to query whether an address is in either the currently active generation or the queued generation.

- **Solidity Function Signature:** `isInPool(address callerAddress) returns (bool)`
- **ABI Signature:** `0x8baced64`

Or to check whether the current calling address is in the pool.

- **Solidity Function Signature:** `isInPool() returns (bool)`
- **ABI Signature:** `0x1ae460e5`

8.4.2 Is In Generation

Use the `isInGeneration` function to query whether an address is in a specific generation.

- **Solidity Function Signature:** `isInGeneration(address callerAddress, uint256 generationId) returns (bool)`
- **ABI Signature:** `0x7772a380`

Or to query whether the current calling address is in the pool.

- **Solidity Function Signature:** `isInGeneration(uint256 generationId) returns (bool)`
- **ABI Signature:** `0xa6c01cfd`

8.5 Entering and Exiting Pools

The following functions can be used for actions related to entering and exiting the call pool.

8.5.1 Can Enter Pool

Use the `canEnterPool` function to query whether a given address is allowed to enter the caller pool.

- **Solidity Function Signature:** `canEnterPool(address callerAddress) returns (bool)`
- **ABI Signature:** `0x8dd5e298`

Or to query whether the current calling address is allowed.

- **Solidity Function Signature:** `canEnterPool() returns (bool)`
- **ABI Signature:** `0xc630f92b`

8.5.2 Can Exit Pool

Use the `canExitPool` function to query whether or not you are allowed to exit the caller pool.

- **Solidity Function Signature:** `canExitPool(address callerAddress) returns (bool)`
- **ABI Signature:** `0xb010d94a`

Or to query whether the current calling address is allowed.

- **Solidity Function Signature:** `canExitPool(address callerAddress) returns (bool)`
- **ABI Signature:** `0x5a5383ac`

8.5.3 Enter Pool

Use the `enterPool` function to enter the caller pool.

- **Solidity Function Signature:** `enterPool() returns (bool)`
- **ABI Signature:** `0x50a3bd39`

8.5.4 Exit Pool

Use the `exitPool` function to exit the caller pool.

- **Solidity Function Signature:** `exitPool() returns (bool)`
- **ABI Signature:** `0x29917954`

CHAPTER 9

Events

The following events are used to log notable events within the Alarm service.

9.1 Scheduler Events

The Scheduler contract logs the following events.

9.1.1 Call Scheduled

- **Solidity Event Signature:** `CallScheduled(address callAddress)`
- **ABI Signature:** `0x2b05d346`

Logged when a new scheduled call is created.

9.1.2 Call Rejected

- **Solidity Event Signature:** `CallRejected(address indexed schedulerAddress, bytes32 reason)`
- **ABI Signature:** `0x513485fc`

Logged when an attempt to schedule a function call fails.

9.1.3 Awarded Missed Block Bonus

- **Solidity Event Signature:** `AwardedMissedBlockBonus(address indexed fromCaller, address indexed toCaller, uint indexed generationId, address callAddress, uint blockNumber, uint bonusAmount)`
- **ABI Signature:** `0x1effaa2`

Executed anytime a pool member's bond is awarded to another address due to them missing a scheduled call that was designated as theirs to execute.

9.2 Call Contract Events

Each CallContract logs the following events.

9.2.1 Call Executed

- **Solidity Event Signature:** `CallExecuted(address indexed executor, uint gasCost, uint payment, u` ;
- **ABI Signature:** `0x4538b7ec`

Executed when the call is executed.

9.2.2 Call Aborted

- **Solidity Event Signature:** `_CallAborted(address executor, bytes32 reason)`
- **ABI Signature:** `0xe92bb686`

Executed when an attempt is made to execute a scheduled call is rejected. The `reason` value in this log entry contains a short string representation of why the call was rejected. (Note that this event name starts with an underscore)

9.3 Caller Pool Events

The following events are logged related to the caller pool.

9.3.1 Added To Generation

- **Solidity Event Signature:** `_AddedToGeneration(address indexed callerAddress, uint indexed pool)`
- **ABI Signature:** `0x4327115b`

Executed anytime a new address is added to the caller pool.

9.3.2 Removed From Generation

- **Solidity Event Signature:** `_RemovedFromGeneration(address indexed callerAddress, uint indexed pool)`
- **ABI Signature:** `0xd6940c8c`

Executed anytime an address is removed from the caller pool.

CHAPTER 10

Terminology

Definitions for various terms that are used regularly to describe parts of the Alarm service.

10.1 General

Ethereum Alarm Clock, Alarm, Alarm Service Generic terms for the service as a whole.

Scheduler, Scheduler Contract The solidity contract responsible for scheduling a function call.

Call Contract The solidity contract that is deployed for each scheduled call. This contract handles execution of the call, registration of call data, gas reimbursment, and payment and fee dispursment.

Scheduled Call A contract function call that has been registered with the Alarm service to be executed at a specified time in the future (currently denoted by a block number).

Caller Pool A group of ethereum addresses that have registered to be execute scheduled function calls.

10.2 Calls and Call Scheduling

Scheduler The account which scheduled the function call.

Executor The account which initiates the transaction which executes a scheduled function call.

Target Block The first block number that a scheduled call can be called.

Grace Period The number of blocks after the **target block** that a scheduled call can be be called.

Call Window Used to refer to either the full window of blocks during which a scheduled call can be executed, or a portion of this window that has been designated to a specific caller.

Payment The ethereum amount that is paid to the executor of the scheduled call.

Fee The ethereum amount that is paid to the creator of the Alarm service.

Anchor Gas Price The gas price that was used when scheduling the scheduled call.

Gas Price Scalar A number ranging from 0 - 2 that is derived from the difference between the gas price of the executing transaction and the **anchor gas grice**. This number equals 1 when the two numbers are equal. It approaches 2 as the executing gas grice drops below the anchor gas price. It approaches zero as the executing gas price rises above the anchor gas price. This multiplier is applied to the payment and fee values, intending to motivate the executor to use a reasonable gas price.

CHAPTER 11

Changelog

11.1 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)

11.2 0.4.0

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

11.3 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.

11.4 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.

11.5 0.1.0

- Initial release.