Ethereum Computation Market Documentation

Release 0.1.0

Piper Merriam

July 27, 2016

Contents

Contents:

CHAPTER 1

What is it?

This is a service on the Ethereum blockchain that facilitates execution of computations that would be too costly to perform within the Ethereum Virtual Machine (EVM).

Running code on the EVM has a cost which makes it costly to perform large computations. Each execution of code within the EVM must be paid for using an abstraction referred to as **gas**. Complex computations could get expensive very quickly as they would require a large quantity of gas to be executed.

The Computation Marketplace allows for someone to pay someone to execute an algorithm outside of the network and report the result back to them. Each algorithm will have an on-chain implementation which can be used to verify whether the submitted result is correct. In the event of a dispute over what the correct result is, the on chain version of the computation is executed to determine the correct answer.

CHAPTER 2

Overview

The **Computation Market** facilitates the execution of expensive computations off-chain in a manner that is both trustless and verifiable.

2.1 How it works

The marketplace can only fulfill computation requests for algorithms that have been implemented within the EVM. A user who wishes to have one of these algorithms computed for them would submit the desired input for that algorithm as a `bytes` value along with a payment for whoever fulfills the request.

Answering to the request involves both submitting the result of the computation as well as a deposit. This deposit is determined by the cost of executing the computation on chain and thus will vary for each algorithm. The deposit must be sufficient to pay the full gas costs for on-chain execution.

After the answer is received the request has a wait period during which someone may challenge the answer. The challenge must also submit what they believe to be the correct computation result along with an equal deposit.

If no challenge is received within a certain number of blocks, the the answer can be finalized at which point the submitter may reclaim their deposited funds along with the payment for their computation.

In the event of a challenge, the computation is carried out on-chain. The result of the on-chain computation is used to check the submitted answers. If either answer is correct, that submitter is paid the payment for their computation and returned their full deposit. The submitter which submitted the wrong answer may claim the remainder of their deposit that is left after having the gas costs for the on-chain computation deducted.

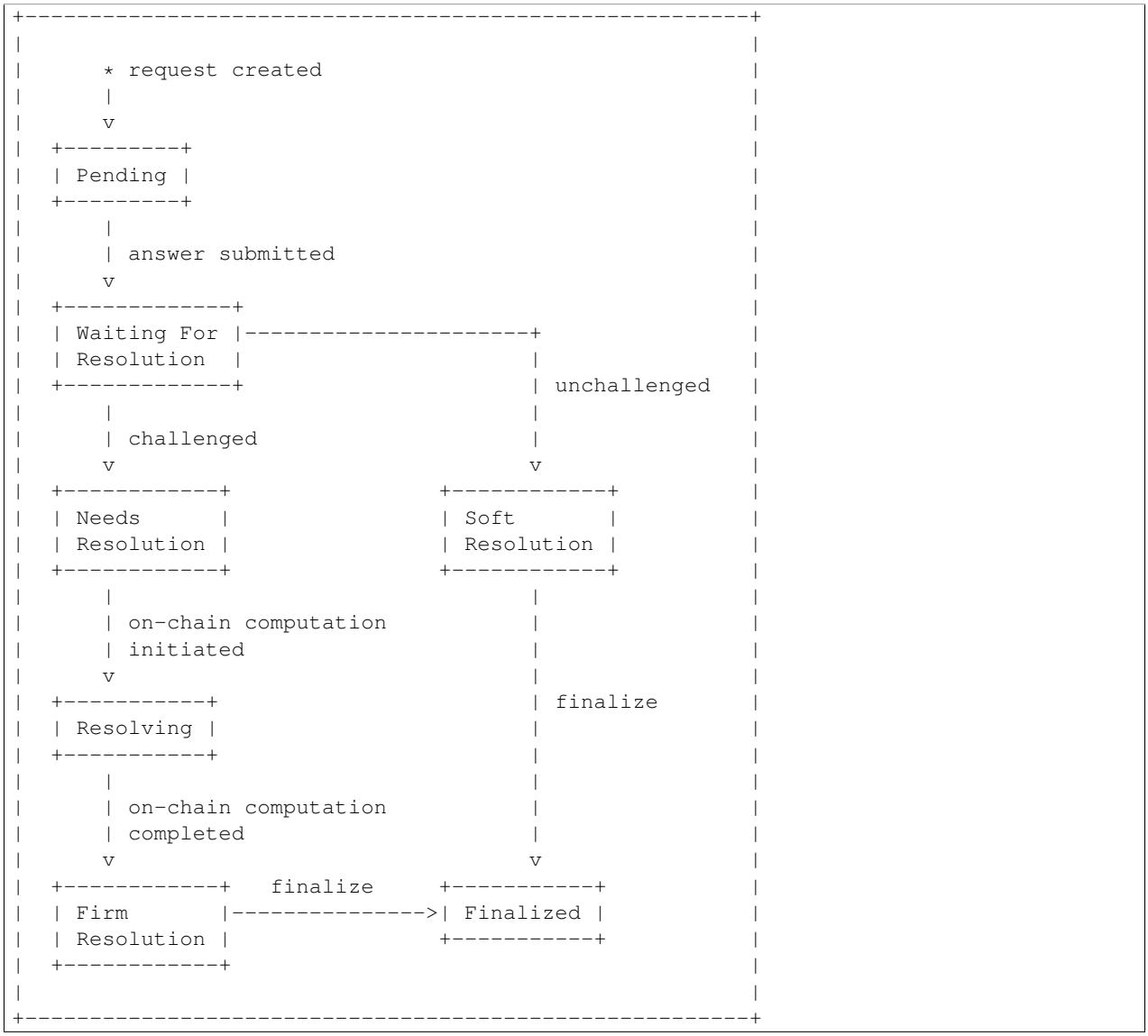
Note: In the event that neither the original answer submitter and the challenger submitted the correct answer, the gas costs for on-chain computation are split evenly between them and the payment value is sent back to the user who requested the computation.

2.2 Markets within the Marketplace

Each individual market within the overall marketplace is an implementation of a single algorithm.

2.3 Computation Lifecycle

The flow chart can be used to visualize the lifecycle of a computation request:



2.3.1 Pending

When a request is received it begins in the **pending** status

2.3.2 Waiting For Resolution

When an answer is submitted towards a computation request it moves the request into the **waiting for resolution** status. From here it can take one of two paths.

1. If the answer is challenged, it will transition into the **needs resolution** status.
2. If the requester manually accepts the answer *or* the predetermined wait time passes, the answer can be transitioned into the **soft resolution** status.

2.3.3 Needs Resolution

When challenged, an answer is set to the **needs** resolution status. On chain verification of the computation is now required.

2.3.4 Resolving

Once the execution contract has been deployed which will perform the on-chain computation the the request is updated to the **resolving** status. It will remain in this status until the computation has been completed.

2.3.5 Firm Resolution

Once computation has been completed, the request is set to the **firm-resolution** status.

2.3.6 Soft Resolution

If no challenge is made for a predetermined wait time after answer submition then the request can be transitioned to **soft-resolution** status.

2.3.7 Finalized

Once and answer is either *soft* or *hard* resolved it can be finalized. This sends the payment for computation to the appropriate party and unlocks the deposits of the answer submitter and challenger.

CHAPTER 3

Markets

The Marketplace is composed of many individual *markets* which each offer computation of a specific algorithm.

3.1 Components

Each algorithm is comprised of three distinct contracts.

- Broker
- Factory
- Execution Contract

3.1.1 Broker

The *Broker* contract coordinates the requests and fulfillments of computations.

3.1.2 Factory

The *Factory* contract handles deployment of the *Execution* contracts in the event that an answer is disputed and must be computed on-chain. The Factory packages the *Execution* contract as well as providing meta-data such as compiler version which would be needed to recompile and verify the bytecode.

3.1.3 Execution

The *Execution* contract is the actual implementation of the algorithm. It can carry out one full on-chain execution of the computation.

3.2 Performing Computations

It is not safe to blindly participate as either the requester or fulfiller in a market. The computation marketplace can provide a trustless wrapper around each algorithm, but it cannot guarantee the correctness or fairness of the underlying algorithm. Since algorithms are implemented as smart contracts they are capable of anything that a smart contract can do.

CHAPTER 4

Computation Requests

A *Computation Request* is an offer to pay someone to compute a given marketplace computation off chain in exchange for a payment.

4.1 Creating a Request

```
function requestExecution(bytes args, uint softResolutionBlocks) public
returns (uint)
```

The `requestExecution` function can be used to create a request for computation. This function takes two arguments as well as an ether value which specifies the payment amount being offered for this computation.

- `bytes args` is the *serialized* inputs to the function.
- `uint softResolutionBlocks` is the number of blocks after an initial answer has been submitted before the answer should be assumed correct if no challenge has been received. Once this limit has passed the request can be finalized.

Any *ether* sent along with this function is set as the payment amount that will be sent in exchange for fulfilling the request.

This function returns an unsigned integer which is the *id* of the created request. This *id* is necessary for all future actions on the request.

4.2 Request Details

The details of a request can be queried with the following three functions:

```
function getRequest(uint id) constant returns (bytes32 argsHash,
                                                bytes32 resultHash,
                                                address requester,
                                                address executable,
                                                uint creationBlock,
                                                Status status,
                                                uint payment,
                                                uint softResolutionBlocks,
                                                uint gasReimbursements,
                                                uint requiredDeposit);
function getRequestArgs(uint id) constant returns (bytes result);
function getRequestResult(uint id) constant returns (bytes result);
```

The primary function `getRequest` returns the following values.

- `bytes32 argsHash`: The *sha3* of the `args` input value for this request.
- `bytes32 resultHash`: The *sha3* of the `result` of this computation. This will return `0x0` if the request has not been finalized.
- `address requester`: The address that requested the computation.
- `address executable`: The address of the executable contract that was deployed to settle a challenged answer. This will be `0x0` if no challenge has been made.
- `uint creationBlock`:: The block number on which this request was created.
- `uint status`:: Unsigned integer related to the `Status` enum. See Request Status for more details.
- `uint payment`:: The amount in wei that this request will pay in exchange for fullfillment of the computation.
- `softResolutionBlocks`:: The number of blocks after answer submission before the answer deposit may be reclaimed if no challenge has been submitted.
- `gasReimbursements`:: Amount in wei that has been paid out so far in gas reimbursments during dispute resolution.
- `requiredDeposit`:: Amount in wei that must be provided when submitting an answer to this request as well as challenging that submitted answer.

4.3 Request Status

Computation requests use an **Enum** to track their status. The possible values are.

- **0**: Pending: Created but has no submitted answers.
- **1**: WaitingForResolution: Has exactly one answer. This answer has not been verified or accepted, nor has it been submitted long enough to allow the submitter to reclaim their deposit.
- **2**: NeedsResolution: Has a challeng answer. Neither of the answers have and resolution has not begun.
- **3**: Resolving: Resolution has been initiated and is in-progress.
- **4**: SoftResolution: This request has a single answer which has not been challenged and was submitted long enough ago that the submitter has been allowed to reclaim their deposit.
- **5**: FirmResolution: This request has an answer which was verified via the on-chain implementation of the computation.
- **6**: Finalized: This request has a result. Deposits may now be returned and payment issued.
- **7**: Cancelled: The request has been cancelled.

The current status of a request can be gotten by looking at the unsigned integer value at index **7** returned from `getRequest`.

4.4 Cancelling

- `function cancelRequest(uint id) public`

A computation request can be cancelled as long as no answers have been submitted. Cancellation is done with the `cancelRequest` function which takes the *id* of the request to be cancelled as its sole argument.

Only the requester of the computation may cancel a request.

4.5 Submitting an Answer

- `function answerRequest(uint id, bytes result) public`

Submition of an answer to a computation is done with the `answerRequest` function. It takes the *id* of the request being answered as well as the `bytes` serialized answer to the computation.

4.5.1 Answer Deposit

Answer submission requires a deposit in ether. This deposit is held until the request has reached either a soft or hard resolution. The required deposit amount can be gotten from the unsigned integer value at index 9 of the return value of `getRequest`.

4.5.2 Retrieve Answer

The data related to the submitted answer can be retrieved with the following functions:

```
function getInitialAnswer(uint id) constant returns (bytes32 resultHash,
                                                      address submitter,
                                                      uint creationBlock,
                                                      bool isVerified,
                                                      uint depositAmount);
function getInitialAnswerResult(uint id) constant returns (bytes);
```

- `bytes32 resultHash` - The *sha3* of the submitted result. This will be `0x0` if no answer has been submitted.
- `address submitter` - The address that submitted this answer.
- `creationBlock` - The block number this answer was submitted on.
- `isVerified` - Whether this answer was verified via on-chain computation.
- `depositAmount` - The amount in wei that was is currently being held as a deposit for this answer. This amount will not reflect any gas reimbursment charges that may be incurred due to an answer challenge situation.

4.6 Soft Resolution

- `function softResolveAnswer(uint id) public`

Soft resolution occurs when an answer is accepted without being challenged. If the address which requested the computation chooses, they can call the `softResolveAnswer` anytime after submission to accept the answer and transition it into the **SoftResolution** status.

Otherwise, after the number of blocks specified by **softResolutionBlocks** have passed since the submission of the answer, anyone may call this function.

4.7 Finalization

- `function finalize(uint id) public returns (bytes32)`

Once a request is in either the **SoftResolution** or **FirmResolution** status it can be finalized via the `finalize` function. This function sets the final result of the computation, pays the correct parties for their computation, and returns the *sha3* of the result as a return value.

4.8 Reclaiming Deposits

- `function reclaimDeposit(uint id) public`

Once a request has been finalized, the deposits of the answer submitter and challenger can be reclaimed. If the submitted answer was found to be incorrect during on-chain computation the deposit will have had the gas costs of that computation deductd from it.

CHAPTER 5

Challenging Answers

The core mechanism that allows the computation marketplace to operate is the ability to perform the computation on-chain in the event of a dispute. In the event that a participant in the computation side of the marketplace sees an answer submitted to a request which does not match their own computations, they may *challenge* the answer.

This initiates a 3-step process which will execute the computation within the EVM to verify which submitted answer is correct. The gas costs for this computation are paid for from the deposit of the incorrect submitter.

5.1 Step 1: Challenge

- `function challengeAnswer(uint id, bytes result) public`

Anytime after an answer has been submitted and the request is in the **WaitingForResolution** the answer may be challenged with the `challengeAnswer`. The arguments for challenging and answer are the same as the `answerRequest` function.

5.1.1 Challenge Deposit

Challenging an answer requires the same deposit amount as the initial answer submission. This minimum value is returned at the 9th index of the return value of `getRequest`.

5.1.2 Retrieve Challenge Answer

The data related to the challenge answer can be retrieved with the following functions:

```
function getChallengeAnswer(uint id) constant returns (bytes32 resultHash,
                                                        address submitter,
                                                        uint creationBlock,
                                                        bool isVerified,
                                                        uint depositAmount);
function getChallengeAnswerResult(uint id) constant returns (bytes);
```

These two functions follow the same API and return values as the `getInitialAnswer` and `getInitialAnswerResult` functions.

5.2 Step 2: Initialize Dispute

- `function initializeDispute(uint id) public returns (address)`‘

Once an answer has been challenged, it needs to have the dispute resolution initialized. This is done by calling the `initializeDispute`. This function may be called by anyone once a challenge has been submitted. The broker contract will use it's *factory* to deploy a new *executable* contract initialized with the inputs for this request.

The gas costs for calling this function are fully reimbursed during execution.

5.3 Step 3: Computation & Resolution

- `function executeExecutable(uint id, uint nTimes) public returns (uint i, bool isFinished)`

Once the dispute has been initialized, the `executeExecutable` function must be called until the the computation has been completed. Once computation completes the function will be set to the **FirmResolution** status.

The gas costs for calling this function are fully reimbursed during execution.

5.4 Finalization

Once a request is in the **FirmResolution** status it can be finalized the same as if it were soft resolved. In the **FirmResolution** case, the result of the on-chain computation is set as the final result of the requested computation.

- If one of the submitted answers was correct, they may then reclaim their full deposit and are sent the payment value in wei.
- If both of the submitted answers were wrong, the submitters split the gas costs evenly. In this case, the payment value is returned to the address that requested the computation.
- Either submitter who's answer was incorrect can reclaim the remainder of their deposit that was not used for on-chain gas costs.

CHAPTER 6

On Chain Computation

The Computation Marketplace is intended for algorithms which are sufficiently complex that performing them on-chain will be costly. For many algorithms, this also means that computation must be split across multiple steps to fit within the **gas limit** of the EVM.

6.1 Stateless Computations

An executable is **stateless** if the implementation does not require any additional data beyond the output from the previous step.

A fibonacci computation that was stateless would be implemented such that each step of computation returned the two most recently computed fibonacci numbers. This would allow each step to operate purely on the return value of the previous step.

This design allows for computation of the 3rd fibonacci number to be executed as follows.

- *fib_3rd = fib.step(3, fib.step(2, fib.step(1, "3")))*

Each of these calls could be made using *.call()* allowing all computation to be done using the on-chain implementation without actually sending any transactions.

Executable contracts that are **stateless** are superior to **stateful** implementations in cases where the **stateless** implementation does not introduce unacceptable complexity. Stateless implementations allow for participants in the computation market to compute the requested computation using the actual on-chain implementation.

The fibonacci example is best done as stateless since the implementation overhead of returning the latest two fibonacci numbers is small.

6.2 Statefull Computations

An executable is **stateful** if the implementation requires additional information beyond the return value of the previous step.

A fibonacci implementation that was stateful might store each computed number in contract storage, only returning the latest computed number. On each step, this function would need to lookup the computed number from two steps ago in order to compute the next number. This reliance on local state is what makes the contract **stateful**. It also disallows using *.call()* to compute the final result since each execution of the *step* function must be done within a transaction in order to update the local state of the contract.

An algorithm like scrypt could theoretically be done as a stateless contract, but each step would have to return a very large lookup table due to the the nature of the scrypt algorithm. This large input and return value would reduce the

number of actual computations that could be executed in a single step which would cause a significant increase in the total number of steps necessary to complete the computation.

6.3 Authoring an Algorithm

The simplest way to author a new algorithm is to use the abstract solidity contracts provided by the service.

6.3.1 StatelessExecutable and StatefulExecutable

- `contracts/Execution/Execution.sol::StatelessExecutable`
- `contracts/Execution/Execution.sol::StatelessExecutable`

These abstract contracts can be used to implement **stateless** or **stateful** algorithms. They only require implementating the `step` function from the *Execution Contract** api.

6.3.2 FactoryBase

- `contracts/Factory.sol::FactoryBase`

This abstract contract can be used to implement the *Factory* API for an *Execution Contract*. It requires you implement either a `_build` function which returns the address of the newly deployed *Execution Contract*. This function implements a default `build` function which logs an event with the address of the newly deployed contract which can be overridden if this behavior isn't wanted.

6.4 Example Stateless Fibonacci Contracts

The following example code implements a **stateless** *Execution Contract* and *Factory* for computing fibonacci numbers.

```
import {StatelessExecutable, ExecutableBase} from "contracts/Execution.sol";
import {FactoryBase} from "contracts/Factory.sol";


contract Fibonacci is StatelessExecutable {
    function Fibonacci(bytes args) StatelessExecutable(args) {
    }

    function step(uint currentStep, bytes _state) public returns (bytes result, bool) {
        /*
         *  Uses a 64-byte return value to serialize the previous two
         *  computed fibonacci numbers.
         */
        uint i;
        bytes memory fib_n;

        if (currentStep == 1 || currentStep == 2) {
            // special case the first two fibonacci numbers
            fib_n = toBytes(1);
        }
        else {
            // otherwise extract the previous two fibonacci numbers from
            // the previous return value to compute the next fibonacci number.
```

```
            var n_1 = _state.extractUint(0, 31);
            var n_2 = _state.extractUint(32, 63);
            fib_n = toBytes(n_1 + n_2);
        }

        if (currentStep > toUInt(input)) {
            // If we have computed the desired fibonacci number just return
            // it as a serialized bytes value.
            result = fib_n;
        }
        else if (currentStep == 1) {
            // Special case the first step to initialize the 64-byte return
            // value.
            result = new bytes(64);
            for (i = 0; i < fib_n.length; i++) {
                result[32 + i] = fib_n[i];
            }
        }
        else {
            // Write the latest two computed numbers to the 64-byte return
            // value.
            result = new bytes(64);
            for (i = 0; i < 32; i++) {
                result[i] = _state[i + 32];
                if (i < fib_n.length) {
                    result[32 + i] = fib_n[i];
                }
            }
        }

        return (result, (currentStep > toUInt(input)));
    }

    /*
     *  Functions used to serialize and deserialize unsigned integers to
     *  and from bytes arrays.
     */
    function toUInt(bytes v) constant returns (uint result) {
        // Helper function which converts a bytes value to an unsigned integer.
        for (uint i = 0; i < v.length; i++) {
            result += uint(v[i]) * 2 ** (8 * i);
        }
        return result;
    }

    function extractUint(bytes v, uint startIdx, uint endIdx) constant returns (uint result) {
        // Helper function which extracts an unsigned integer from a slice
        // of a bytes array.
        if (startIdx >= endIdx || endIdx >= v.length) throw;
        for (uint i = startIdx; i < endIdx; i++) {
            result += uint(v[i]) * 2 ** (8 * (i - startIdx));
        }
        return result;
    }
}


contract FibonacciFactory is FactoryBase {
```

```
    function FibonacciFactory() FactoryBase("ipfs://test", "solc 9000", "--fake") {

    function _build(bytes args) internal returns (address) {
        var fibonacci = new Fibonacci(args);
        return address(fibonacci);
    }
}
```

6.5 API Requirements

A developer authoring an algorithm for the the computatoin market **must** conform to the following API requirements.

All algorithms must be implented as a contract with the following API.

It **Must** take a `bytes` value for any arguments receives as input.

It **Must** implement the following functions.

6.5.1 isStateless()

- `isStateless() constant returns (bool)`

Returns whether this computation relies on intermediate state. If each step of execution only relies on the output of the previous step then this should return `true`. Otherwise it should return `false`.

6.5.2 isFinished()

- `function isFinished() constant returns (bool)`

Return `true` if the computation has finished, or `false` if computation is in progress.

6.5.3 getOutputHash()

- `function getOutputHash() constant returns (bytes32)`

Return the `sha3()` of the final return value of the computation. Return `0x0` if the function has not completed computation.

6.5.4 requestOutput(bytes4 sig)

- `function requestOutput(bytes4 sig) public returns (bool)`

This function should send the `bytes` return value of the computation to the `msg.sender` by calling the function indicated by the provided 4-byte signature as follows.

```
function requestOutput(bytes4 sig) public returns (bool) {
    if (isFinal) {
        return msg.sender.call(sig, output.length, output);
    }
    return false;
}
```

This allows circumvention of the inability of contracts to receive `bytes` values from the return values of external function calls.

6.5.5 step(uint currentStep, bytes _state)

- `function step(uint currentStep, bytes _state) public returns (bytes result, bool)`

This function should perform one unit of computation that is sufficiently small to fit within the EVM **gas limit**.

It will be provided the following arguments.

- **uint currentStep** - The 1-indexed step number for the current computation. This increases by one each time the contract executes a unit of computation.
- **bytes _state** - The return value from the previous step, or the `bytes` input value that the contract was initialized with if this is the first step.

This function must return a tuple of `(bytes, bool)` where the `bool` value indicates whether computation has completed.

6.5.6 execute()

- `function execute() public`

A call to this function **must** advance the computation by a single step. If the computation has already completed, this function **should** throw an exception.

6.5.7 executeN(uint nTimes)

- `function executeN(uint nTimes) public returns (uint iTimes)`

A call to this function **should** advance the computation up to but not exceeding **nTimes**. This function **must** interpret `nTimes == 0` as instruction to run as many steps as possible prior to returning. This function **must** return the number of steps that were completed. This function **should** throw an exeception if computation has already completed when called. This function **must** handle the case where execution of **nTimes** steps would exceed the **gas limit** and still execute successfuly in these cases.

Factory

This contract is used to deploy instances of the *Execution Contract**.

It **must** implement the following API.

6.5.8 sourceURI()

- `function sourceURI() returns (string)`

Returns the URI where the source code for this contract can be found.

6.5.9 compilerVersion()

- `function compilerVersion() returns (string)`

Returns information about the compiler and version which should be used to recompile the bytecode of this contract.

6.5.10 compilerFlags()

- `function compilerFlags() returns (string)`

Returns any configuration arguments that should be used to recompile the bytecode of this contract.

6.5.11 build(bytes args)

- `function build(bytes args) public returns (address addr)`

A call to this function should deploy a new instance of the *Execution Contract* initialized with the provided `bytes args` value. This function **must** return the address of a contract which conforms to the *Execution Contract* API and which can be used to compute the result of the computation given the provided `bytes args` input value.

6.5.12 isStateless()

- `isStateless() constant returns (bool)`

Returns whether this factory's *Execution Contract* is **stateless** or **stateful**. If each step of execution only relies on the output of the previous step then this should return `true`. Otherwise it should return `false`.

6.5.13 totalGas()

- `function totalGas(bytes args) constant returns(int)`

Returns the total gas estimate in wei that would be required to compute the computation for the given input.

CHAPTER 7

API

7.1 Broker

7.1.1 Solidity API

```
TODO
```

7.1.2 ABI

```
TODO
```

7.2 Factory

7.2.1 Solidity API

```
TODO
```

7.2.2 ABI

```
TODO
```

7.3 Executable

7.3.1 Solidity API

```
TODO
```

7.3.2 ABI

```
TODO
```

CHAPTER 8

Events

The computation market contracts emit the following events.

8.1 Broker Contract Events

8.1.1 Created

- `event Created(uint id, bytes32 argsHash)`

Logged when a new computation request is created.

uint id The id of the request

bytes32 argsHash The sha3 of the computation arguments.

8.1.2 Cancelled

- `event Cancelled(uint id)`

Logged when a computation request is cancelled.

uint id The id of the request

8.1.3 AnswerSubmitted

- `event AnswerSubmitted(uint id, bytes32 resultHash, bool isChallenge)`

Logged when either the initial answer or an answer challenge is submitted.

uint id The id of the request

bytes32 resultHash The sha3 of the submitted answer

bool isChallenge Whether the submitted answer was the initial answer or an answer challenge.

8.1.4 Execution

- `event Execution(uint id, uint nTimes, bool isFinished)`

Logged when the on chain execution contract has advanced at least one step in execution.

uint id The id of the request

uint nTimes The number of execution steps that were completed successfully.

bool isFinished Boolean indicating if on-chain computation has completed.

8.1.5 GasReimbursement

- `event GasReimbursement(uint id, address to, uint value)`

Logged when a gas reimbursement is sent.

uint id The id of the request

address to The address that the reimbursement was sent to.

uint value The amount in wei that was sent.

8.1.6 Payment

- `event Payment(uint id, address to, uint value)`

Logged when the payment for a computation is sent.

uint id The id of the request

address to The address that was paid.

uint value The amount in wei that was sent.

8.1.7 DepositReturned

- `event DepositReturned(uint id, address to, uint value)`

Logged when a deposit is returned to either the initial answer submitter or the challenger.

uint id The id of the request

address to The address that was returned.

uint value The amount in wei that was sent.

8.2 Factory Contract

A *Factory* contract *should* emit the following events.

Note: Each algorithm author is in full control of how they construct their Factory contract. These events are not part of the required API so it may be left out for some markets.

8.2.1 Constructed

- `event Constructed(address addr, bytes32 argsHash)`

Logged when a new *Execution Contract* is deployed.

address addr The address of the newly created contract

bytes32 argsHash The sha3 of the constructor arguments that were passed to the the contract.

8.3 Execution Contract

The *Execution Contract* API does not define any event.

CHAPTER 9

Changelog

TODO

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

A

B

U