Temporal Innovation on the Blockchain

ChronoLogic

October 2, 2018
v0.1

Abstract

Running a service like a TimeNode can be profitable for the operators, however, given the many variables that influence its payout, it's not trivial to estimate the expected profits. The proposed model and simulation code has been created to address that problem.

1 Introduction

Scheduling is a feature that is native to the modern financial world. We schedule our routine bank transfers and bill payments; whenever we know ahead of time when a payment should go through we set it up to pay automatically instead of waiting for the exact time and date. Financial transactions are increasingly moving to the self-sovereign and trustless world of cryptocurrencies. As more volume is moved on networks like Ethereum daily, there is more expectation of the foundational tooling that fiat world provides. Users expect scheduling as a feature from the Ethereum ecosystem because they have learned to rely on it in their everyday lives. In fact, scheduling calls on Ethereum is more powerful than scheduling movements of money in the fiat world– users can now schedule smart contracts that contain much more complex logic. We also identify the tooling gap that exists for developers working on the Ethereum stack, used to cron jobs on Unix. Contrary to intuition, scheduling calls on a decentralized peer-to-peer blockchain is not a trivial problem to solve.

There are many reasons why a person would want to schedule smart contract calls. The most facile of use cases involves the simple scheduling of value transfers – the movement of ether or an ERC20 token from one address to another. With the future indoctrination of concepts such as DAOs, the world will start to see smart contracts employed in the use of organizational governance and other high stakes uses. Users may want to schedule a vote to a DAO for instance that will determine the zoning of their neighborhood that they live in. Additionally, developers may invent new use cases in which scheduling is used under the hood, without the user even realizing the complexity behind the application that they are interacting with. Blockchains are the new software platform. People who set up and maintain software environments use cron to schedule jobs (commands or

shell scripts) to run periodically at fixed times, dates, or intervals. There needs to be a cron for the world of Ethereum.

1.1 Centralized Scheduling

On Ethereum there does not exist any native way to have a smart contract be called at a specified time in the future, or to make this call recurring. The obvious need for this feature has led developers to create their own solutions. In the (now deprecated) Parity UI there was a built-in scheduling feature that would locally hold the transaction in memory before sending it at the desired time. Although the solution worked fine, there were some major drawbacks including 1) the necessity of the user running their own local node and using the specific Parity client software, and 2) the single point of failure experienced in the instance that the local node disconnected from the peer-to-peer network. These issues can actually be composed into one issue: centralization.

1.2 Decentralized Scheduling

The advantages of a decentralized protocol over a centralized one is that it does not have a single point of failure. The scheduling aspect of the Ethereum Alarm Clock [1] is done entirely through smart contracts running on Ethereum on the backend and allows for permissionless implementations of user-friendly frontends [2]. The execution aspect of the protocol is handled by a network of off-chain clients known as TimeNodes. TimeNodes are incentivized to operate by the user-set bounty payment, which can be thought of as a small reward provided by the user for the TimeNode to expense its own computation costs for monitoring the state of Ethereum. From a user perspective, the existence of TimeNodes never has to be known and they can pay the small premium of the bounty without any understanding of the execution flow of the Ethereum Alarm Clock protocol entirely.

2 Ethereum Alarm Clock 1.0.0

In November 2017, nearly one year after the DDoS attacks that metaphorically threw a ratchet in the Ethereum Alarm Clock's gears, developers from the ChronoLogic team began to work on and reboot the Ethereum Alarm Clock protocol. After about nine months of work including updating the core smart

[1] The Ethereum Alarm Clock was created by Piper Merriam, now Lead of the Python team at Ethereum, on August 26, 2015. Throughout its development history it was deployed to the mainnet and maintained almost solely by a single developer. The adoption of the protocol did not take off and after the 2016 DDoS attacks on the Ethereum network the smart contracts started to fall out of date as Piper began pursuing new projects

[2] Among two of the already implemented front-ends on the current iteration of the Ethereum Alarm Clock are the ChronoLogic Chronos Dapp (app.chronologic.network) and a native integration with the popular MyCrypto wallet (mycrypto.com)

contracts, rewriting the test suite in JavaScript for the Truffle framework, building developer libraries and constructing a new TimeNode client in TypeScript, and performing numerous internal audits and an external audit with a notable member of the community, the ChronoLogic team deployed the 1.0.0 stable release of the Ethereum Alarm Clock smart contracts on August 24, 2018. The following information will discuss the architecture, analyze the cryptoeconomic incentives, and enumerate on some of the limitations found in the stable release.

2.1 Architecutre

The Ethereum Alarm Clock architecture is best described in two symbiotic and differential parts– the smart contracts which are deployed to the Ethereum main chain and operate trustlessly and the TimeNode network, which are off-chain execution agents in charge of handling the logic of executions.

2.1.1 Smart Contracts

The smart contracts of the Ethereum Alarm Clock can be thought of as consisting mainly of three parts: the RequestFactory, the Schedulers (further divided into the TimestampScheduler and BlockScheduler), and the TransactionRequestCore. Within these three main sections of the architecture are numerous libraries that contain various pieces of the core logic and functionality. There are nine library contracts, two top-level scheduler contracts, on contract for Request Factory and one for TransactionRequestCore, adding in Zeppelin's SafeMath and Ownable brings the total to 15 smart contracts which compose the entire Ethereum Alarm Clock architecture.

The logic for the scheduled transactions exist on the TransactionRequestCore contract which acts as the central library for the numerous instances of the delegatecall clone contracts that get deployed via the RequestFactory. Below is shown the code example of the primary actions of the TransactionRequestCore, which allows TimeNodes to execute, cancel or claim a scheduled transaction.

```
function execute() public returns (bool);
function cancel() public returns (bool);
function claim() public payable returns (bool);
```

The Schedulers (BlockScheduler.sol and TimestampScheduler.sol) are the top level APIs which will get exposed to the user facing applications. These contracts provide a simpler interface taking only the necessary parameters for a scheduled transaction and setting the rest of the options to sane defaults. The Scheduler contracts are also the objects which hold the data concerning the taking of fees. The purpose of separating the logic for taking fees into the Scheduler contracts is to allow third party developers and new integrations to customize their fee structure (and fee destination) in their own preferred way.

Allowing the option for third party integration to customize their fee structures creates an incentive for them to make the integration into their own product. Each scheduler contracts works off the same Request Factory backend so it benefits from the same network of TimeNodes.

The RequestFactory contract is the core of the Ethereum Alarm Clock protocol and deploys the new delegatecall clone contracts. It is possible to call the Request Factory directly but most of the time the Request Factory will be called by an "internal transaction" by one of the Scheduler contracts. The RequestFactory has two functions which perform different kinds of validation of the parameters which are passed in. The lower level 'createRequest()' will attempt the raw creation of a Transaction Request without any validation. By not performing validation the function call saves gas but may lead to the possibility that the parameters contained an error and will not be executed properly. For most use cases some validation of the input parameters is desired and the 'createValidatedRequest()' will be used instead. The Request Factory will also be the factory that TimeNodes watch in order to keep up to date with scheduled transactions since it emits the 'RequestCreated' event which alerts TimeNodes of user scheduled transactions.

2.1.2 TimeNode

TimeNodes are the off-chain execution agents which represent individual nodes of the decentralized peer-to-peer execution network behind the Ethereum Alarm Clock.

The first and primary TimeNode action is the execute function while the second action is claim - which well go into detail in the next section. Execute is called anytime a TimeNode tries to execute a transaction that has been scheduled to be executed.

By executing a transaction, a TimeNode receives a reward (bounty). This helps incentivize TimeNodes to execute incoming scheduled transactions.

The execute action is run by triggering a function within the smart contract that holds the scheduled transaction. This trigger requires the TimeNode to spend a small amount of gas to trigger this action.

A problem arises when multiple TimeNodes send the execute trigger at the same time, i.e. within the same block. Only one of them will trigger the execution, while the other TimeNode's transaction will get reverted and cost a small amount of gas.

Should a TimeNode notice that it often collides with other TimeNodes during the execution time of scheduled transactions, it can choose to claim transactions.

2.2 Claiming Mechanism

Claiming is an advanced opt-in feature of the Ethereum Alarm Clock protocol which helps TimeNodes lower the risk of transaction collisions.

When users schedule a transaction, this scheduled transaction appears on the blockchain and TimeNodes keep track of them. Once it is scheduled, a

TimeNode can attempt to claim the scheduled transaction by depositing a small amount of ETH.

Claiming transactions removes the risk of colliding with other TimeNodes during the execution time, but brings forward another set of problems:

1. Transaction collisions - The same transaction collisions that used to happen on the execute function will now be a problem with the claim function. There is a chance of multiple TimeNodes trying to claim the same transaction within the same block, which will again result in a transaction collision and only one of those will be able to claim it.

2. Deposit loss - When a TimeNode claims a transaction it deposits a small amount of ETH as a guarantee that it will be online to execute it when the time is right. Should a TimeNode (due to unforeseen circumstances) go offline at the exact time at which the scheduled transaction was due to be executed, it will lose its deposit.

2.2.1 Remote Providers

The effect of sending the claim or execute actions for a scheduled transaction can bring more risks, depending on whether a TimeNode is connected to a local or a remote node.

Connecting the TimeNode to a remote provider can introduce delays due to network conditions and can facilitate failed claims/executions. Running a local RPC provider node would be the recommended way of running the TimeNode to eliminate the risk of slow responses for the RPC requests made by the TimeNode.

2.2.2 Mitigation Efforts

Efforts have been made to mitigate the risks mentioned above by introducing the hasPending check to the timenode-core library. hasPending allows the TimeNodes to check the transaction pool (txpool) of the providers it is connected for any incoming execute or claim actions on a certain scheduled transactions.

This allows the TimeNodes to avoid transaction collisions if they use a node that has a txpool such as Parity or Geth. Keep in mind that RPC providers without a txpool (e.g. Infura over HTTP) will not work with this feature.

2.2.3 Recommendation

In order to avoid the mentioned risks of running a TimeNode, we recommend the following setup:

- Connect the TimeNode to a local Parity/Geth node
- Turn on claiming transactions
- Reliable internet connection - minimum offline time

These risks are currently unquantifiable and it is still not clear how the network will behave in real world conditions.

2.3 Cryptoeconomics

The Ethereum Alarm Clock protocol incorporates cryptoeconomic incentives to reward the decentralized network of TimeNodes to continue operation. The incentive consists of the extra amount of ether sent to the TimeNode following an execution. The amount of the bounty is variable and depends on when the transaction was claimed as well as how much extra gas was paid for the execution of the transaction. As a general rule, TimeNodes will not execute transactions which will return a net loss to their ETH balance. The formula for the bounty calculation is provided below.

$$B_{total} = P \times B_{set} - (G_{actual} - G_{set})$$

where P is payment modifier

Given that scheduled transactions are expected to be executed at the exact time and the network of n competing nodes exists, we expect to face the "swarming" problem which can be described as: uncoordinated attempts of execution by n nodes at the same time. This problem may result in unnecessary costs for TimeNode making the operations potentially not profitable as only 1-of-n is going to earn the TimeBounty for the execution, other TimeNodes trying to execute at the same block will pay the failing transaction cost. By introducing payment modifier P TimeNode operators are able to pick theirs profitability point, this effectively allows to mitigate the "swarming" problem as we expect TimeNodes to try to claim on different blocks/moment of time. However, under perfect competition, the profitability metric of the TimeNode trends toward perfect zero leading TimeNodes to often send claim transaction within the same block.

2.4 Limitations

2.4.1 Gas price

The Ethereum network uses the concept of gas as a unit of measuring the computational work performed over the network. Every transaction broadcasted to the network sets the gas limit and the gas price. Gas limit describes the maximum amount of gas allowed to be consumed by the transaction, gas price describes the amount of ether (ETH) you are willing to pay for each unit of gas.

While setting the gas limit is in most cases very straightforward and automatic, setting the correct gas price is not a trivial task. Gas price determines the time between broadcasting and block inclusion, depends on current network capacity. We observed many spikes in the gas price due to popular ICOs, CryptoKitties or last FCoin on-chain voting. Given the nature of scheduled transactions, protocol needs to handle the execution using predicted gas price. The most important characteristics are:

- execution prioritization
- TimeNode withholding protection

There are three possible solution for this problem, let analyze each of them separately.

1. Fixed gas price - Allows setting the gas price that has to be used by TimeNode for execution, declared gas price is reimbursed by the scheduler

   Pros:

   - simple
   - protects from TimeNode withholding

   Cons:

   - fails upon gas price spikes
   - requires users to guess the price in the future

2. Range gas price - Allows setting minimum and maximum gas price by the scheduler, the maximum gas price is reimbursed, any gas price used by the TimeNode between min and max (spot price) will allow the split of the remaining budget between the scheduler and the TimeNode.

   *For e.g, the scheduler sets the range from 20gwei to 100gwei and thus locks enough funds to cover 100gwei * gas limit. TimeNode is incentivized to pick a good price between 20 and 100, let's say 50 as remaining 50 is split, making TimeNode earn extra 25.*

   Pros:

   - protects from TimeNode withholding
   - incentivize TimeNodes to pick correct price within bounds

   Cons:

   - given the front-running, the equilibrium is max
   - requires a decently high max in order to cover the spikes

3. Minimum gas price - Allows setting the minimum gas price by the scheduler, the minimum gas price is reimbursed, higher gas prices covered by TimeNode (effectively reducing the reward for execution)

   Pros:

   - protects from TimeNode withholding
   - allows the TimeNode to decide where equilibrium is

   Cons:

   - covers spikes up the minimum reimbursement + reward

In order to pick the right solution we need understand the Ethereum Alarm Clock execution process. There are 3 different time windows used for:

- pre-execution claiming process: where TimeNodes are bidding on reservation for execution
- reserved execution: time period for TimeNode that claimed transaction
- execution: free for all execution, when TimeNode that claimed missed the execution or no claiming happened

Minimum gas price is the best option for both reserved execution and execution as it allows TimeNodes to control and pick the gas price that best fits their own profit model.

2.4.2 Account abstraction

For cases where the address of the sender has to be known upfront and given the Ethereum Alarm Clock architecture where each scheduled transaction is represented by separate smart contract and have unique Ethereum address there is a need to use a proxy wallet. Proxy wallet address will be seen as msg.sender in destination contract/account. An example procedure off scheduling transaction using proxy wallet could be described as follows:

1. create schedule request such as the destination is set to proxy wallet
2. send scheduling request using proxy wallet so it becomes the owner of the request
3. whitelist the scheduled request in the proxy wallet so it can be relayed

The need for such a workaround will be necessary when native account abstraction [3] will be available on Ethereum network.

2.4.3 Gas cost

Using smart contract storage for keeping data with conditions and parameters for scheduled transaction come with a cost. Currently each scheduled transaction costs approx. 500 000 gas. Even though the total cost in USD is below $1[4] the "premium" for scheduling is over 20x comparing to regular ETH transfers.

[3] https://github.com/ethereum/EIPs/issues/859
[4] Given current 8.2Gwei gas price