Rutile White Paper

by Franklin Waller

Executive Summary

Blockchain is great for generating trust and transparency around your application. But it suffers a lot of issues for the user. Using the blockchain get confusing very quickly. By needing to educate users about what gas prices are and convincing them to pay fees for your application.
Even when your user understands these terms it can still be very error prone. Users can lose their money by incorrectly setting fees. Developers can mess up their smart contract resulting in users paying insane amounts for using your application.
Even with current blockchain technologies most platforms sacrifice security and decentralisation in order to solve some of these problems. Resulting in projects struggling to update because their delegates refuse to update (DPoS).
This all results in frustration for both the users and businesses.

Rutile want to tackle this problem by not relaying on a blockchain, but rather on a DAG. This allows users to execute the application code themselves, which translates to no transaction fees. Everyone can participate in the network so security and decentralisation is not sacrificed.

This all greatly reduces the cost for businesses to use Rutile as a Dapp platform. All with having the benefits of using a decentralised system, such as a higher uptime.

About Rutile

Rutile is an open source decentralised application platform that is based on the Directed Acyclic Graph instead of a blockchain. This allows for a high uptime and no fees for the client. With Rutile businesses don't have to maintain a server farm themselves. This greatly reduces the complexity of creating a high scalable network that users can relay on.

Apps on Rutile are written in Web Assembly which allows any developer with any language to write for Rutile. Developers don't have to know how the Rutile network works in order to start creating. They simply use their favourite language and start writing their Dapp. This greatly reduces the learning curve and allows for faster development.

Since the scripts are deployed to the Rutile network any node can access all scripts and execute them. This allows users not be depended by one node in order to execute the scripts. Even when nodes crash the network won't be effected. This way Rutile can create a high uptime network that is not depended on one company.

Innovations

Rutile is not build on blockchain but rather on a *Directed Acyclic Graph* (Or DAG for short). Which is used by IOTA to reach consensus (Also known in IOTA as Tangle). Not only allows this for a faster transaction rate, but it also allows for fee less transactions. While other projects require clients to pay fees to use their Dapp, Rutile is free for users to use.

One of Rutile's features is that it allows executions to be done with any language by using WASM. This allows thousands of developers to execute functions on the rutile network while not needing to know how decentralisation works.

Rutile is aiming to be an all in one solution to act like a backend. This includes execution, storage and hosting. This way users can build a full blown website with all the needs of creating a website or app without needing a centralised server solution.

Rutile for Decentralised Applications

Storage

Rutile's storage will be handled by the InterPlanetary File System. IPFS allows files to be stored decentralised. Each Rutile node will have an IPFS node installed so that it can handle storage of scripts and other files. Files are stored on multiple Rutile nodes so even when a node shuts-down files will still be available. The files itself are not stored on the DAG but rather a reference to the IPFS hash is stored on the DAG. Nodes will get payed by the client for the storage space they are providing to them.

A client can either choose to store their files encrypted or unencrypted. This is because some Dapps require files to be publicly available.

Execution

Rutile allows small programs to be executed. We refer to them as "functions". Functions are small pieces of code that can accept input and returns an output. After the execution has been completed the process will be terminated. Functions are written in Web Assembly, this way developers can pick any language they want to build applications for Rutile.

The Rutile encapsulates a lot of its API in order so that the developers don't need to know how Rutile itself works, but still be able to write decentralised apps.

Functions have a maximum time to execute, this is configured to 5 minutes as default. If the execution has not been completed, the function will be terminated. Clients can however choose to give a function longer the time to execute.

Functions are executed in a virtual machine which only has access to certain functions that are secure for the node. This way a node and a client are not harmed by the functions executed on it. Rutile does expose an API to retrieve files, get state and update the state. If the function throws an error while executing, the transaction will become invalid and will not be able to be added to the network.

A client can execute the functions itself and send the result back to the network which will be validated by other nodes so we know the client didn't cheat and updated the state incorrectly. In this case of a false state update the transaction will be considered invalid and discarded.

When a client is not powerful enough it can choose to let execution to be done by a full node. In this case the client does pay a gas fee to that node. Gas is a fixed unit for each operation in a function. Each op call in WASM is assigned a gas value. Every time a op call is executed, the amount of gas is added to the total gas used. The client pays the total gas and what was left gets returned to the user.

The gas price is determined by the client in tiles. So for example a function costs 10 gas to execute. The gas price is 10.000 tiles. 10*10.000 = 100.000 tiles. The 100.000 tiles will then be rewarded to the node that executed the function. (More on Tiles in the section **RUT Token**)

Hosting

Hosting is done by utilising the storage layer of Rutile and exposing it. This can be done by a Rutile client. But since we do not want to limit websites to using a Rutile client to access websites we expose an HTTP layer. Developers can deploy a zip package with the contents of their UI (HTML, CSS and JavaScript). A user can then ask any Rutile node to return the contents of the zip package. The format http://ip:port/host/[HASH_ID] is used. Where HASH_ID is the id of the transaction that includes the HTML, CSS and JavaScript to run the Dapp. Hosting is payed by the function creator in bytes.

State

Not only the amount of tokens is kept in the state. In Rutile you can keep a state of your application as-well. For example what the last high score for a user was. The state is not meant to store huge amount of data, but rather small pieces of data that can be distributed across the network. It can however be combined with the storage layer to store lots of data.
The state can be updated by sending a transaction to the network that contains instructions (In form of a function) to update the state.

Node types

Full node

The full node is for users that want to help the network by executing functions and hosting files. The full node will be written on the Node.js platform. This allows computers that run Windows, Mac OS and Linux to run a full node and participate in the network.

Full nodes are connected using Peer to Peer by using the WebRTC protocol. WebRTC allows for a quick communication between computers and browsers. This way browsers can connect to nodes via the same protocol Rutile uses and don't need a different protocol to talk to each other.

Clients (Or lite nodes) can use full nodes to store files on them and submit transactions to the DAG.

Lite node

Lite nodes are way to access the Rutile network and submit transactions to the network. These transactions can send tokens to an address and update the current state of the network. They can either connect via WebRTC or via HTTP to the Rutile network. The DAG will not be stored on lite nodes and will have to relay on full nodes to retrieve transactions that needs to be validated.

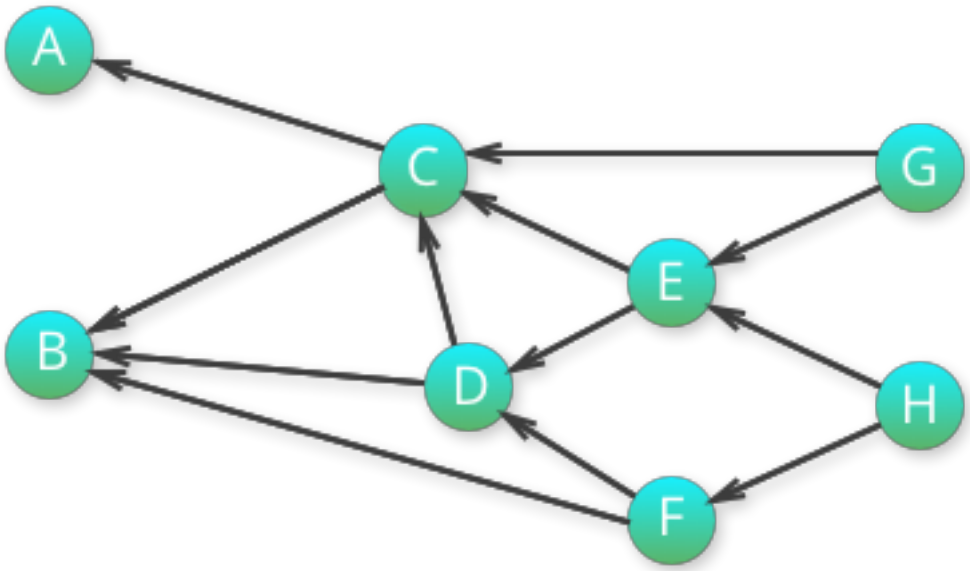
The lite node will be available as JavaScript SDK, which will sign transaction and give them to the network. The SDK can be used to create Dapp's and wallets.

Consensus

In order to reach consensus in the network, Rutile uses the DAG. Different to blockchain, DAG allows multiple transactions to be added in parallel. The more transactions added to the network the more it will scale.

In addition to using DAG over blockchain the transactions are fee-less and scalable. This is done by letting the client validate 2 other transactions and do the Proof of Work instead of letting the network do the work. In current blockchain applications the client have to pay in order to use the Dapp.

We will not go in full detail on how the consensus work since many papers cover this. But the basics are that each transaction (green bubble) gets reference by two other transactions. This creates a graph of transactions.



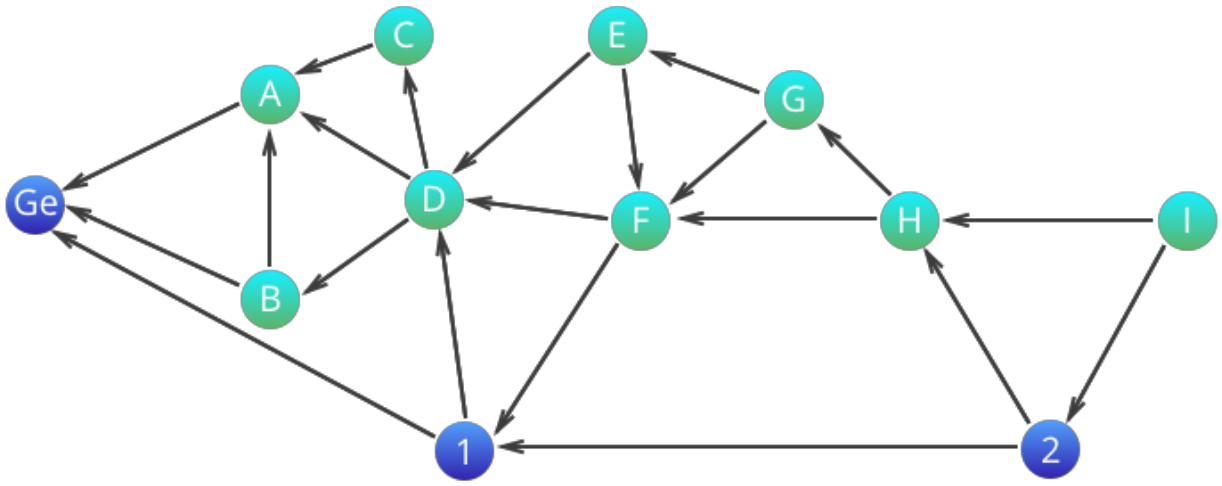
In order for a transaction to be confirmed the new transactions have to point indirectly to a different transaction. In the above's case A, B, C, D, E are "confirmed". This is because both G and H are referencing these transactions indirectly. Transaction F is not indirectly or directly pointed by G. This is why the confirmation percentage for transaction F is 50% (Because only 1 of the 2 new transactions is pointing to it). The confirmation percentage is used as a reference for the chances it's going to be chosen in the next milestone.

Milestones

Rutile is different than a typical DAG in the sense that there is main chain built into it. The purpose of the main chain is to allow state updated to be put in an ascending order. This way every transaction get put in order.

The main chain is created using milestones instead of blocks. Milestones are a special type of transaction that only full nodes can create. Each milestone points back to the previous milestone and to 1 transaction in the DAG. This results in a chain of milestones and a ripple effect inside the DAG of state updates.
You can consider milestones as a “new genesis” since newer transactions should point to the latest milestone.

Take a look at the figure:



In this figure milestones are in effect. Each milestone has an index of the previous index + 1 starting at the genesis milestone. This way nodes know which order to execute the milestones. When a node hits a milestone it should follow the same path the milestone creator took.

The path goes as follows: Take the previous milestone and work backwards to the next milestone. This is to make sure transactions are executed in order. Let's take the figure as an example starting at the Genesis block. Transactions start flowing in and validate the Genesis and another transaction (In the case of transaction A there is no other transactions to be validated.)

Milestone index 1 is created by a node and attached to transaction C. Milestone creator should work backwards from Genesis. But as you can see in this case there is both A and B that could be followed backwards to the Milestone index 1. In this case the network should pick the transaction that has the most cumulative weight to it. Cumulative weight is a measurement that means how "important" a transaction is by looking at the amount of transactions are pointed at it.

It is however possible that 2 transactions have the same cumulative weight. In this case the PoW hash will be converted to a number. Then the lowest number will be selected by the milestone. Transactions that did not get chosen will be selected after the first milestone walk has been completed. This will loop until all transactions that where directly or indirectly pointed by the milestone has been processed.

After the processing the milestone will include its path it took inside the milestone transaction. This way even though cumulative weight may change, the order does not. The milestone only references the transaction id's, not the transaction contents.

Milestone creators

Milestone creators are full nodes that help creating the main chain. These nodes are also picked in a decentralised manner using Proof of Stake. A node puts some of their stake in the network to get a chance of being selected by the network. The stake determines how high your chances are.

Once selected by the network the node creates a milestone and validates the transactions. After completing its task it will broadcast its milestone to the network. After this the next node who creates a milestone validates the previous milestone and rewarding the node with newly minted tokens.

If a node acts maliciously and creates a milestone that is not considered to be valid other nodes can discard the milestone and their stake will get burned. This encourages nodes to behave according to the protocol.

It’s also possible that a node doesn’t respond to it is task to create a milestone. In this case a node is not punished in the sense of burning stake. But the network will pick a different node to create the new milestone. This way the node is only punished in the chance of earning rewards.

RUT Token

RUT tokens are Rutile's native currency. They are used to pay for resources such as storing files. RUT tokens are split in multiple Tiles. Tiles is the smallest unit of a RUT token. 1 RUT token is equal to 10^{18} Tiles.

In the early stages of development the token will be on the Ethereum network. (as a ERC-20 token). Once the Rutile network is considered stable users can transfer their tokens over to the Rutile network.

Token creation

At the ERC-20 token creation there will be a fixed amount stetted at: $150{,}000{,}000$ RUT tokens. On Ethereum these tokens are non mint-able.

Once on the Rutile network, tokens can be minted by milestone creators. This would be roughly 2 RUT tokens per minute. This translates to around 1 million inflation per yer (0.6% inflation)

Token distribution

Tokens will be distributed through an airdrop. 70% of all tokens ($105{,}000{,}000$) will be distributed across its participants. 22% is used for company operations (such as hiring new developers). And the last 8% is given to the early contributors of the Rutile team.

Sources

- WebAssembly (https://webassembly.org/)
- IOTA (https://www.iota.org/)
- IPFS (https://ipfs.io/)
- OByte (https://obyte.org/)
- WebRTC (https://webrtc.org/)
- Ethereum (https://ethereum.org/)