BlueBeast - White Paper (alpha)

**Abstract**

*BlueBeast* is a software that enables a submitter (a person or an organization) to pay a small fee in order to let processors perform calculations on the behalf of the submitter. The *BlueBeast* software aims to be a lightweight client that consist of two parts, a local processor & a smart contract hierarchy.

**Protocol**

The local processor runs through *Docker* within a *Vagrant* virtualized environment and the installation of *BlueBeast* is therefore not only more secure, due to the virtualization, but also agnostic to the operative system upon installation as long as *Virtualbox* & *Vagrant* is installed as prerequisites.

An *Ethereum* wallet is also either imported or created locally with the *BlueBeast* software in order to sign transactions on the *Ethereum* blockchain. *BlueBeast* will at first be introduced on the *Ethereum* platform and have its smart contract written in *Solidity*. However, the aim is to keep the decentralized platform easily exchangeable so that the contracts could be switched out for other languages, like *EOS* if they were to be proven better.

Even though the language is *Solidity*, the contracts will be deployed on *GoChain* due to a consensus model that makes it much faster than *Ethereum* while still retaining its backwards compatibility for the smart contracts.

*“The BlueBeast protocol is not a blockchain, it’s a local application that communicates with decentralized logic.”*

*BlueBeast* wants grant public blockchains as well as private organizations the ability to reach consensus on decentrally computed results and reward processors for doing the computation work in order to reach the results. One issue that arises is therefore that the processors, in order to analyze a ledger, need full access to that particular ledger. While some processors may run the *BlueBeast* on top of an existing node, not everyone would. Not only would every node need a copy of the ledger they want to run on, but a consensus regarding the input and verify where the nodes would get their data from would also have to be established. This would create a very specific approach where each protocol would need their own custom integration instead of a generic pipeline that fits all.

The issue then becomes that anyone could create a job and claim to be submitting work for a chain when in fact they are not. By including a method to tag jobs with what chain the job is assigned to we can thereby circumvent untrustworthy sources claiming to provide data for a specific protocol. By creating one or multiple authorized protocol address, that the protocol gives permission to submit jobs in their name, we can ensure that all jobs submitted with a certain tagged protocol comes from a trustworthy source. Even though we won’t be able to accurately see where the data the submitter submits come from we can point to a by the protocol authorized address, as a sole submitter of that blockchains jobs. All addresses authorized within each protocol would also be able to authorize more addresses, enabling each blockchain using *BlueBeast* to have their own custom made access to providing jobs for processing.

Adding blockchains/protocols to the smart contracts themselves are made by the *BlueBeast* foundation, so if you want to get your protocol listed feel free to reach out to us. Getting listed costs X *Ether* and all money will be used to continue to fuel the development of the *BlueBeast*.

**UNIT & Reputation**

The main audience for the *BlueBeast* will be other blockchains that easily want to receive calculated rewards to encourage certain behaviors on their platform but there’s also a possibility that private corporations or anonymous actors wants to utilize the space. The *BlueBeast* will enable different protocols to offer payment for jobs in their own native currency but there’s no way for us to enforce that the payouts actually happens. The *BlueBeast* therefore feature its own integrated currency known as *UNIT*. *UNIT* is an *ERC20* tokens that can be transferred between different Ethereum addresses, and by submitters staking *UNITs* we can ensure that the ones processing results will receive their share if their results align with the consensus.

The UNIT approach will be appealing for protocols & corporations that don’t run their own currency but still want to reward the processors for doing work. Blockchains with established currencies can also incentivize processors by offering part-payment in *UNIT* & part-payment in their own currency and thereby securing payment partially for the processors.

*UNITs* will be purchasable by sending *Ether* to contract address that handles *UNITs*. Just as *EOS* started of with *ERC20* Tokens, the vision for *BlueBeast* is also to migrate to its own transaction platform at a later point.

To each address we will also tie a reputation variable that increases as an processors submit result that align with consensus or decrease if no result is returned, or the result doesn’t align with consensus. If the consensus returns an error message, no reputation change will happen for the processors but instead affect the job submitter negatively (more about this further down). Results that don’t receive error will grant the submitter positive reputation, however processors can also manually vote positive or negative on a submitter address manually. This may be fitting if a protocol promise rewards in its own currency but then later don’t pay it out. This however requires the processor to supply the id of the job transaction & appear as Authorized in the its list of processors authorized to work. So no processor can down/upvote a submitter without having performed a job for them.

**Usage**

At first, a user enters our website and heads to post a job. Once there a form will appear featuring multiple fields.

* Internal Max Stake (How much UNITs the user is willing to spend on the job, we can ensure that UNIT payouts always happens)
* External Max Stake (How much of the protocols own currency they are willing to spend on the job, we cannot ensure that this get paid out, but we can provide proof that the processor addresses deserve them)
* Minimum Nodes (Minimum nr of nodes the submitter want consensus of)
* Maximum Nodes (Maximum nr of nodes the submitter want consensus of)
* IPFS Hash (A hash that contains an IPFS-address leading to the Data the submitter wants to get computed as well as the Logic that want to use to compute it)
* Bidding Block (A blocknr where all bidding for the job will end).
* Expiration Block (A blocknr to which all stake will be refunded unless the job has been processed and result has been returned).
* Protocol (A drop-down menu that is filled with all the protocols that the address is authorized to submit for. General addresses will only have “Unknown” as a protocol, but a Neo authorized address will have “Neo” & “Unknown”.

This way we enable protocol to promise payout in their own currency while at the same time incentivizing other users to process data through unknown transactions. By letting each protocol having authorized addresses that are manually added, we can promise that each authorized address is tied to a reliable entity in the given protocol.

All jobs are also verifiable by looking into each transaction hash of the job & seeing what data & logic that is stored on the *IPFS* hash submitted by that transaction.

Once all fields have been filled the transaction is signed by the submitter & the job is sent to the blockchain. Processors can then either by manually enter the website and bid on jobs (from a list that reads from the smart contracts) or by running the local processor automatically bid in accordance with a bid amount set in their settings. In these settings the user is also able to specify want kinds of protocols they want the processor to run.

The processors will then sign transactions and bid on jobs that their settings allow them to bid on. This cost a small amount of gas which also incentivize the processors to not use shame bids. Once the jobs *Bidding Block* appears the job will decline any further bidding and assign X amount of processors where X is more than minimum nodes & where the sum of all Xs bids are less than max stake the job. If no such agreement can be found, all remaining stake will be sent back to the submitter and the job cancelled. If not, all processors that has bid on a job will run “IsAuthorized()” with their own Address as well as the job as parameters.

Only the addresses of processors that won the hidden auction will be added to the list of Authorized and any node that isn’t authorized will be returned false, and stop monitoring the job.

A node that returns true however, will be sent a *IPFS* hash which the processor will download. The files stored on *IPFS* will have an entrypoint (where the code starts to run) & an endpoint (where the result is put). This way all file uploads to *IPFS* follows a similar structure and should be computable generically. There is however a possibility that different computation will require different dependencies and installations for prerequisites will have to be added to *Vagrant*. As a start we will run with *Python 3.7* and add more as we go.

Once a job has been computed its result will be signed with the processors private key and sent as a transaction to the job that gave them the *IPFS* hash. The job itself will keep track of how many nodes that return & then either when all nodes have returned something or when *Expiration Block* happens compute the result. This is done by looping through the list of results and checking as to whether the most common is result appears more than 51% of the times. If that happens, all nodes that computed the most common result will be granted a reward equal to their stake for the job. A node that computes a result that’s not within consensus will not receive any reward. All excessive stake will be returned to the submitter of the job together with the job result, addresses of all nodes that were assigned the job, addresses of all nodes that returned the job & addresses of all nodes that returned but were not within consensus. The job is than archived and closed.

Each address also has a tied score for reputation, where returning results that align with consensus increases score, returning result that don’t align slightly decrease it & getting assigned a job but not returning at all decreases slightly more.

Two possible issues that may arise however is:

* What if the computational job is so intensive that a result can’t be returned before the E*xpiration Block*?
* What if the computation returns an error?

The solution to those questions goes hand in hand. There’s of course a possibility that some jobs might not be computed within a reasonable amount of time, but punishing the processors for calculating overly intensive jobs is not a good solution. That is why the processors will also read the assigned job´s *Expiration Block* & X blocks before that (X being customizable in settings) abort the computation, generate an error message and publish that as a result.

So what then happens if an error is published as result? Well, there is a chance that nodes will form a consensus on error. Maybe the user posted a intensive job that only 20% of nodes had the ability to compute within the timeframe given. That way 80% of the jobs (providing everyone returns) would submit the same error message: *“Computation too intensive for the timeframe provided”.* When the smart contract then checks for consensus the error message would be the most common result.

But what about the processors that actually completed the job on time? Should they be punished? No, they should not. When a consensus is reached on the default error message all processors get rewarded but no reputation changes happens, except for the submitter of the job that receives negative reputation (they get positive reputation for successful results). That way all processors are rewarded and the submitter is punished by not only losing stake, but paying for reaching consensus on an error message. It’s therefore important that the submitter either:

* Have a high stake to attract the best processors.
* Have a high expiration for a computationally intensive job.
* Have a decent understanding about how intensive the algorithm they want to run is.

**So what is it that makes BlueBeast special?**

* Majority of computation happens locally and therefore do not clog the network.
* We are agnostic to data & can add any input stream or chain.
* We enable different protocols to reward users in their own native currency.
* We enable anonymous actors to receive consensus on results.
* We are cheap, not only because we run on *GoChain* but because we don’t use Gas Costs for storage, but *IPFS*.
* We are fast, because most data from the blockchain is read, not written, computations happens locally & we run on *Gochain*.
* We are trusted but trustless, since all protocol addresses has been authorized by the foundations themselves.
* We are flexible, new protocols can be added & the smart contracts are upgradeable by separating logic & storage. We can adapt to different smart contract platforms by following the same architecture.
* Our use case stretches outside of blockchain, any organization or person may use us to perform reach consensus on computations.
* We are secure, all operations the processors perform are virtualized and blackboxed. All executions are preconfigured and happens automatically.
* We are lightweight & easy to use. Just one download needed & we don’t need to store local ledger copies.