Constellation

A blockchain microservice operating system

July 14 2017

Abstract

Distributed applications need a distributed operating system. The current scaling issues with Bitcoin/Counterparty and Ethereum are issues with transparency in a distributed system. The underlying consensus mechanisms and smart contract systems can not scale to what we expect for consumer applications. Reformulation of cryptographically secure consensus into modern serverless architecture will allow mainstream applications to use blockchain technology. Presented is a blockchain microservice operating system that addresses these issues.

Introduction

Despite the goal of decentralization, cryptocurrencies are operated by increasingly centralized networks. As with the rise of post Bitcoin networks, the main goal of processing financial transactions has been expanded to provide trustworthy contractual agreements. While implementations of these smart contract ecosystems exist they are siloed from existing software, making use of blockchain technology costly and time intensive. Native blockchain technology requires a protocol that functions as an operating system.

Bitcoin was created to solve the problem of distributed consensus for financial transactions, but relied upon an intensive process known as proof of work that pools vertical computational power from a select few individuals with domain specific knowledge, into coins for that same select few. Antiquated throughput requirements (aka block size) combined with a time intensive algorithm has caused transaction times and cost to skyrocket. Ethereum uses proof of work but is moving to proof of stake which literally just allows those with the most money to choose the state of the network. Ethereum pioneered the use of smart contracts, but their implementation is a barrier to use as financial infrastructure for digital services. Serverless architectures are key to providing the same consumer grade products we currently use off chain.

The cryptoeconomy has been looking to new implementations of consensus to solve the scalability issue. How can we process more transactions and perform more services with smart contracts in less time with less cost? We propose an approach that applies techniques from data intensive software engineering to solve these problems, namely horizontal scalability and MapReduce. Horizontal scalability is the application of concurrent programming; it means that the more computers that join the network, the more work can be done. MapReduce is process of breaking computation into simple operations that can can be fed into an asynchronous DAG (directed acyclic graphs) of computation, increasing the efficiency of an already concurrent program. The Constellation protocol implements a horizontally scalable blockchain architecture known as Extended Trust Chain with a peer to peer layer known as a gossip protocol that can be deployed on a personal device. Constellation approaches smart contracts with a microservice architecture allowing for highly available services to be chained and composed into distributed applications with just an understanding of each microservice's SLA's (service level agreement) and/or type signature.

Gossip protocols allow for large networks to communicate total network state. Each member of the network keeps track of neighbors and when it receives new messages, it propagates that message to all of its neighbors. This has some interesting mathematical properties, but in our case, it allows for a large pool of connected devices to share the state of their resources.

Horizontal scalability inherently means splitting one program into multiple asynchronous operations. One approach to blockchain consensus that allows this is a TrustChain, namely ExtendedTrustChain. In ExtendedTrustChain, there are multiple node types, each with their own responsibility that asynchronously govern different aspects of the protocol. Specifically, there are basic nodes that send transactions and host individual chains (each individual's chain forms DAG's with others when a transaction is made and is hashed into linear blocks). There are node processes that perform consensus on checkpoints, collections of transactions that will be hashed and become the next block in the chain. Finally, there are validator processes, who host the current chain state and seed chain state/history to the node. Each node is associated with an account and a meme (for reputation.) This architecture will allow for personal devices to run full nodes and perform fast transactions.

The formulation of distributed applications as MapReduce operations on smart contracts has been proposed by the Plasma upgrade to Ethereum. However with its application to the EVM, it still suffers the same pitfalls of synchronous architecture. The Solidity programming language was built to address the risks of non-determinism in smart contract development. The need for deterministic instruction stems from the need to calculate the cost of running a Smart Contract and prevent against DDOS attacks. However, if compiled byte code could be notarized on chain and verified in a JVM, then deterministic compilation becomes a looser requirement. Reduction in gas consumption, circumvention of gas limits and smart contracts with the complexity of microservices are now a possibility.

It is logical to design Dapps as compositions of successful building blocks. On chain interoperability allows for blockchain companies to provide services and use others. A truly decentralized ecosystem cannot exist with monolithic smart contracts. A successful ecosystem tends towards diversity. Successful ecological actors siphon energy from more antiquated species and their traits die out.

The use of the JVM as a conduit for distributed applications is an industry standard in the Big Data community. Consumer applications are deployed as a collection of microservices on autoscaling groups which rely on easy provisioning, like the JVM provides. Thousand node clusters can be provisioned and autoscaled on AWS virtual instances and run distributed applications on the JVM. The same can be of smart contract microservices.

This type of distributed architecture inherently requires the modularity that functional programming provides. Smart contract microservices designed as concrete types provide a structure for less technical application developers to create new functionality; by combining smart contracts of applicable type into composite applications. Constellation's smart contract interface will be designed with covariant and contravariant typing in mind. Smart contracts implementing modular interfaces can be composed with the same ease even across different blockchains, which we will support pending solutions to cross chain liquidity issues.

Considering our mention of functional programming as a design principle of our smart contract microservices, it is worth noting that our consensus protocol could be described by the category theoretic definition of a hylomorphism.

Consensus Architecture

Our consensus architecture builds upon ExtendedTrustChain. In Extended trust chain, each node operates as an account, transactions are signed by the initiator and counterparty then broadcasted to the network via gossip. A set of facilitators, who perform consensus on the transactions within a checkpoint, are chosen randomly. The consensus result is broadcasted again and added as a normal transaction to the next block. [1]

Our approach scales this with self similar structure. Consider the architecture above of one individual consensus network as a solar system. Multiple

[1] Kelong Cong, et al. https://repository.tudelft.nl/islandora/object/uuid:86b2d4d8-642e-4d0f-8fc7-d7a2e331e0e9?collection=education

of these can exist independently, and their composite could be contained in a checkpoint block of checkpoint blocks. To adapt the approach of [1] our validator nodes will run this meta checkpoint of checkpoint blocks, which we can consider as a star cluster. Star clusters then get mined by Galaxies, and the galaxies make up our blockchain universe.

Nodes can be ”sleepy” i.e. join and leave the network. As they join the network, their resources are allocated to a new solar system. Once the solar system reaches a threshold for transaction throughput and cryptographic security (the number of facilitators/the number of participants), it only allows new nodes when others leave. Thus new solar systems, or consensus networks will be dynamically allocated as members leave and join the network.

As these processes are horizontally scalable, it is possible run a basic node or facilitator process on a personal or mobile device. This could offset the cost of transaction fees by allowing participants to earn those fees by participating in consensus on these devices. It may also be possible remove transaction fees and incorporate an incentive structure into our consensus model, which we describe below.

Consensus Model: Proof of Meme

Our architecture requires a reputation system for selection consensus participants. We present Proof of Meme; a node's previous participation in the network is used to determine its probability of being chosen to participate in consensus. A meme in our sense is a feature vector, in the simplest case an array of float values used as input to a machine learning algorithm. Each feature is representative of a node's utility to the network (has it been notable faulty during consensus, how large are its checkpoint block sizes etc). A meme can be transformed into a numeric value by passing it into a function, which is how we will quantify a meme's reputation. In order to promote new nodes to improve their meme, a probability distribution of meme scores is drawn over brackets of fixed size; the distribution outlines the probability that a meme of a score in a particular bracket gets selected. Reputable memes will be given preference but new memes will also have an opportunity to participate. Logs of performance will be checkpointed so memes that perform faulty or adversarially will have their reputation reduced and well performing memes will have theirs increase. Our idea for consensus participant selection in the inductive case is as follows:

1) Consensus is performed.

2) The hash block is fed to a deterministic algorithm running that outputs updated meme scores for each neighbor (one hop) from this consensus node.

3) This constant is used to shuffle our distribution (move memes between buckets).

4) The previous block hash (result of last consensus) is used to sort the contents of each bucket and the top N from each bucket are selected for this round. The meme of a consensus participant who provable acts faulty or malicious (as verifiable within logs) will be docked in the next consensus participant election.

It is possible that transaction fees could be phased out as meme reputation begins to hold more value. Smart contract services could look to reputable memes to host services which could be more profitable than earning transaction fees for performing consensus. Instead of increasing prices when latency is high, meme's with low reputation scores could have their transactions processed with lower priority. A meme in this case could provision resources, thus increasing throughput for themselves and solving bandwidth issues for the network.

Smart contracts as microservices

Smart contracts themselves are microservices running on a sandboxed JVM. They have the ability to send transactions, sign as a counterparty and perform consensus. The goal is for the microservices themselves to operate as a meme, providing services for agreed upon amounts, like a POS system. They can serve the same role as smart contracts in Ethereum or Counterparty, but so much more as they can provide more complex logic with the existing codebase in the JVM ecosystem and can talk to more robust programs through an RPC interface if JVM sandboxing gets in the way. If these microservices are build with concrete service level agreements or better yet, type signature, their composite logic can be chained and composed into distributed applications intuitively. This is MapReduce operators come into play. A smart contract microservice can be designed to send and receive data models that improve computational complexity (given our asynchronous architecture) and can be repurposed for new applications.