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 cannot 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 will lead to recentralization as those with the most money get to choose the state of the network. Ethereum pioneered the use of smart contracts, but their implementation does not allow the transactional throughput to use as financial infrastructure for most 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 graph) 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 mobile 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 enable mobile 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 scalability can be realized with Constellation 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.

MeshChain

Our consensus architecture, MeshChain 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 in MeshChain). 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 an approach similar to side chains, yet contained within the underlying protocol. According to statistical analysis of of ExtendedTrustChain in [1], the probability of an adversary taking control of consensus approaches 1/2 as its control of consensus participants approaches 1/3. We can use this bound to pick parameters to tune fault tolerance. As throughput is linear with respect to the number of participants, yet a linear increase in fault tolerance costs a polynomial increase in communication cost. Thus we must determine consensus parameters that ensure maximum throughput with bounded fault tolerance and transaction confirmation time. How are we to scale a network with such bounds? Our approach is to compose our network's blockchain as a hierarchy of sub chains from sub nets. Each sub net forms a locality sensitive block hash and sends it to a parent net, which treats locality sensitive block hashes as ordinary transactions. The choice of neighbors in a sub net need only depend on minimization of latency, as each node keeps track of it's own transaction history.

Consider the architecture above of one individual consensus network as a solar system. Multiple of these will exist independently, and their composite will be contained in a checkpoint block of consensus blocks from each solar system. To continue this metaphor, our validator nodes act as a star clusters, running a meta consensus on checkpoint blocks made up of consensus blocks from solar systems. As the network expands, star cluster blocks would then get hashed into galaxy chains, and the set of galaxy chains makes up our blockchain universe. In each level of scale, the same self similar architecture of gossiped message passing is employed, allowing for the same nodes to participate at each level with only a change in resources provided (implicitly larger checkpoint block sizes would require more memory for scar clusters/galaxies.)

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 our threshold for transaction throughput and cryptographic security (the number of facilitators/the number of participants) as outlined above, it only allows new nodes when others leave; new nodes must wait to form a new solar system or join an existing one when a space opens up. Thus new solar systems, or consensus networks will be dynamically allocated as members leave and join the network. The goal of this self similar structure and our staking model outlined below is to enable dynamic autoscaling based on network resource and latency, allowing for consistent throughput through incentives for resource allocation (either with transactions or reputation.) It is trivial to show that this architecture exhibits the power law intrinsic of scale-free networks which are notorious for their application to fault tolerance in complex systems such as distributed computing.

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

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

Staking model: Proof of Meme

Our architecture requires a reputation system for selecting consensus participants. We present Proof of Meme; a node's historical participation in the network is used to determine its probability of being chosen to participate in consensus. Our use of reputation such as proof of meme in conjunction with each node acting as individual account allows our system the benefits of a permissioned system, at least with regard to fault tolerance.

A meme in our sense is a feature vector corresponding to each node's account, in the simplest case an array of float values used as input to a deterministic 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. This is meant to maximize throughput by relaxing the number of facilitators yet maintaining the same level of fault tolerance and confirmation times. Reputable memes will be given preference but new memes will also have an opportunity to participate. Logs of performance will be notarized on chain, 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 based on new data of their performance).

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 acts faulty or provably 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, memes 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

Highly available, elastic distributed systems thrive on a serverless architecture. In the case of a distributed operating system this implies a network of distributed microserveices. Thus, in constellation, 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 node with a corresponding 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 external programs through an RPC interface if JVM sandboxing is infeasible. If these microservices are built with concrete service level agreements or better yet, type signature, their composite logic can be chained and composed into distributed applications intuitively. This is where 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.