A formal definition and statistical model checking of the constellation blockchain

March 1, 2018

Abstract

Introduction

Fundamental Data Structures, Types and Functions

T <: Numeric

Numeric type delineating tier within multi-tier hierarchy

BlockData[T] <: U

Consider a type *BlockData* which serves as the parent type (under universal type U) of all data going on chain. All transactions, validator requests, meme data etc. All BlockData is equivalent via covariance and thus all BlockData can be compressed into a Block.

Block[U] <: U

Contains compressed form of Block Data. This is the result of consensus. Each unit of *BlockData* must reference the previous round of consensus, this is satisfied by containing the *Block* hash. It is worth noting that $Block[U] \equiv Block[BlockData[T]]$ through covariance. Block[U] is a Fix Point Type [1]

Transaction <: BlockData[0]

Transaction is a subtype of BlockData that can only exist on the bottom tier. It is the building block of our currency.

[1] https://jto.github.io/articles/typelevel-fix/

LeftHand <: RightHand <: Transaction

Note: LeftHand and RightHand are symmetric subtypes of Transaction. For a Transaction to be valid it must be 'signed' by the counterparty. This is satisfied by sending a RightHand transaction that references a LeftHand transaction.

Definition: Consensus Cluster := $collection[node]$

A set of validators, undergoing cryptographic consensus to produce a *Block*.

Definition: Validator

A node that has 'woken up' by starting a *consensus* process. It now gossips transactions to it's neighbors and is waiting to be selected as a Delegate.

Definition: Delegate

A node who has been selected to perform consensus. Delegates are implicitly chosen, locally on each node via the Generating Function (see below). Once a new *Block* is received, it is passed to the Generator Function, which tells this node if it is a delegate.

Generating Function $g : Block \rightarrow$collection[node, reputation score]

This function is used to determine the next set of delegates from a given set of validators (consensus cluster) by selectively sampling a subset of nodes, based on reputation score and a probability distribution. See GURU for examples selective sampling via probability distribution.

Consensus (function) $f : collection[node] \rightarrow Block$

A function that maps a consensus cluster to a Block. It follows that this is isomorphic to a catamorphism, with the collection being a cluster's mempool and the result the reduce being a Block.

General Architecture

Node Architecture

A node consists of two processes, and a remote procedure call (rpc) interface.

```
/**
  * This needs to be refactored to spin up one or two processes, one for
      protocol and one (or zero) for consensus
  * the main method to this singleton needs to execute a hylomorphic
      method that takes types representing these actors
```

```
  * ex:
  * def hylo[F[_] : Functor, A, B](f: F[B] => B)(g: A => F[A]): A => B =
      a => f(g(a) map hylo(f)(g))
  */
object BlockChainApp extends App with RPCInterface {
  implicit val system: ActorSystem = ActorSystem("BlockChain")
  implicit val materializer: ActorMaterializer = ActorMaterializer()
  implicit val executionContext: ExecutionContextExecutor =
      system.dispatcher

  val id = args.headOption.getOrElse("ID")
  val chains = new DAG()
  val protocol = system.actorOf(new Protocol(chains), "constellation")
  val consensus = system.actorOf(new Consensus, "constellation")
  Http().bindAndHandle(routes, config.getString("http.interface"),
      config.getInt("http.port"))
}
```

The *protocol* process handles transaction signing and essentially all functionality to send and receive payments.

```
  case class AccountData(var balance: Long,
                         unsignedTxs: mutable.HashMap[String, Tx] =
                             mutable.HashMap[String, Tx](),
                         metaData: List[BlockData] = Nil //TODO: we'll
                             need to figure out some sort of buffering
                        )

  case class GetLatestBlock()

  case class GetChain()

  case class FullChain(blockChain: Seq[CheckpointBlock])

  case class Balance(balance: Long)

trait ProtocolInterface {
  this: PeerToPeer with Receiver =>

  val buffer: ListBuffer[BlockData] = new ListBuffer[BlockData]()
  val chainCache = mutable.HashMap[String, AccountData]()
  val signatureBuffer = new ListBuffer[BlockData]()

  val chain = new DAG
  val publicKey: String

  receiver {
    case transaction: BlockData =>
      logger.info(s"received transaction from ${sender()}")
```

```
      buffer += transaction
      broadcast(transaction)
      sender() ! transaction

    case GetLatestBlock =>
      logger.info(s"received GetLatestBlock request from ${sender()}")
      sender() ! chain.globalChain.headOption

    case GetChain =>
      logger.info(s"received GetChain request from ${sender()}")
      logger.info(s"chain is ${chain.globalChain}")
      sender() ! FullChain(chain.globalChain)

    case GetId =>
      logger.info(s"received GetId request ${sender()}")
      sender() ! Id(publicKey)

    /*
    This should eventually query up the tiers. Would be part of a rest
        service for quick tx signing (ensuring both sigs in one block)
     */
    case GetBalance(account) =>
      val balance = chainCache.get(account).map(_.balance).getOrElse(0L)
      logger.info(s"received GetBalance request, balance is: $balance")
      sender() ! Balance(balance)

    /*
      This should be coupled with a state transition and within a
          separate Consensus process.
     */
    case checkpointBlock: CheckpointBlock =>
      logger.info(s"received CheckpointBlock request ${sender()}")
      if (validCheckpointBlock(checkpointBlock))
        addBlock(checkpointBlock)
        sender() ! checkpointBlock

    case fullChain: FullChain =>
      val newBlocks = fullChain.blockChain.diff(chain.globalChain)
      logger.info(s"received fullChain from ${sender()} \n diff is
          $newBlocks")
      if (newBlocks.nonEmpty && newBlocks.forall(validCheckpointBlock)){
        newBlocks.foreach(addBlock)
        sender() ! FullChain(chain.globalChain)
      }
  }
}
```

The *consensus* process implements the responsibilities of a validator node. Nodes can be 'sleepy', turning on the *consensus* process at will.

```
/**
  * Interface for Consensus. This needs to be its own actor and have a
      type corresponding to the metamorphism that
  * implements proof of meme. The actual rep/dht will be dispatched via
      this actor.
  *
  * We need to implement this to follow the signature below
  *
  *
  * def metaSimple[F[_] : Functor, A, B](f: F[B] => B)(g: A => F[A]): A
      => B =
  *   cata(g) andThen ana(f)
  *
  * and also figure out a way to define the algebras in terms of a
      bialgebra given that
      http://comonad.com/reader/2009/recursion-schemes/
  * and we also need a vector space for proof of meme
      https://en.wikipedia.org/wiki/Bialgebra
  *
  * for ref
      http://www.cs.ox.ac.uk/jeremy.gibbons/publications/metamorphisms-scp.pdf
  */
object Consensus {
  case class PerformConsensus()

  /**
    * Just a stub for now, res will be checkpoint block which will be
        used by updateCache
    * @param actorRefs
    * @return
    */
  def performConsensus(actorRefs: Seq[ActorRef]): CheckpointBlock =
      CheckpointBlock("hashPointer", 0L, "signature",
      mutable.HashMap[ActorRef, Option[BlockData]](), 0L)
}

sealed trait State

case object IsDelegate extends State
case object IsValidator extends State

class Consensus(chain: DAG, mostRecentBlock: BlockData =
    Consensus.performConsensus(Seq.empty[ActorRef])) extends FSM[State,
    BlockData] {
  startWith(IsValidator, mostRecentBlock)

  onTransition {
    case IsValidator -> IsDelegate =>
```

```
    /*
    Initialize consensus process
     */
  }
  onTransition {
    case IsDelegate -> IsValidator =>
    /*
    Update local block and gossip result
     */
  }
  initialize()
}
```

The *chain* class is a local blockchain made up of all transactions/interactions with the chain that this node has made. Each link in the chain is a sub type of BlockData.

The rpc interface $RPCInterface$ connects the node to the outside world through a set of endpoints for sending/signing transactions and 'waking up' the node by starting a *consensus* process.

Underlying protocol

Consensus

Consensus is the process of forming a $Block$, it can be thought of as a function $f : collection[node] \rightarrow Block$. Consensus clusters are formed in a tiered hierarchy. The top most tier forms the global state of the chain, which is made up of $Block$s from preceding tiers. Each tier, from top down until the second to last, has responsibility for routing transactions and validating the blocks from consensus clusters.

What is the lifecycle of a transaction

When a transaction is sent, it is sent from a node to a higher tier which 'routes' the transaction to the consensus cluster(s) that host its 'shard', or the shard of the blockchain that host's it's public key's history. Each transaction has a left and right half. The initiator of the transaction sends the $LeftHalf$ to the network. Its $LeftHalf$ is then referenced by the $RightHalf$ which is sent by the counterparty.

Why do we double sign?

It preserves the notion of ordering. Scenario: I have 5 dollars, I send to two people. I send 5 to Wyatt and then 5 to Preston. I will need to wait until the top tier finishes consensus because both transactions are effectively 'racing'

each other. Double signing allows us to preserve ordering (with high probability) without waiting for the total network to update state. Without this, consumer facing point of sale systems (think grocery store) are not possible; we would need to wait for a global state to reach consensus. Double siginig gives users the illusion of instant transaction confirmation.

How is consensus performed

Consensus $f : collection[node] \rightarrow Block$ is the act of creating notarized data in the form of $Blocks$. In our case, we are using the HoneyBadgerBFT, which prevents against sybil attacks using encryption [2]. Nodes are rewarded based upon successful completion of consensus, the number of transactions they provide, and metadata about their performance. This is all calculated post facto by proof of meme and rewards are given within a set interval of blocks.

How are delegates selected

Delegates are selected locally, by passing the previous block into our Generating Function. This happens within the Consensus FSM.

How is this secure/fit in with our incentive model

Double spends across asynchronous consensus is prevented by double signing transactions. Consensus clusters are rewarded for processing transactions and sybil attacks are mitigated via encryption in honeybadgerBFT. We also are able theoretically improve typical byzantine fault tolerance over 40% thanks to GURU and our reputation model. Ddos attacks can be mitigated via throttling of accounts with low reputation scores. I propose an incentive for routing where each node that routes a transaction signs the tx, and when a tx is notarized each account that routed the tx is given a reputation increase.

[2] ch. 4.3 https://eprint.iacr.org/2016/199.pdf

Scaling Tiers

Hylochain: Recursive Parachains

Routing and Distributed Data Storage

DAG Architecture

Definition using chain complexes

Scaling formula

Dynamic Partitioning for Recursive Parachains

Proof of meme

Proof of meme and consensus

Proof of meme and routing