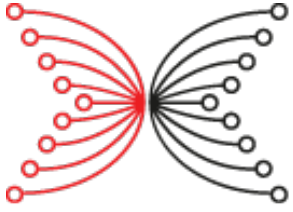
Building a Distributed System with Real-time Constraints

Using concurrent Functional Programming tools

Armando Santos

September 12, 2023 —



INPUT | OUTPUT

Well-Typed

The Haskell Consultants

Introduction

Networking Team

Armando Santos
Well-Typed

Marcin Szamotulski
Input Output Global

Duncan Coutts
Well-Typed

Neil Davies
PNSol

Peter Thompson
PNSol

INPUT | OUTPUT

Well-Typed

What we are responsible for at the networking team

Cardano Node

Ouroboros algorithm paper	Refining step	Implementation	Testing
Formal Specification	Real time constraints	Architecture & Design	Properties
Threat models	Concurrency	Protocols	Simulation
Non-realistic assumptions	Operation & Performance	Scale	Reliability
Performance objectives		Exceptions & Corner cases	CI

Presentation overview

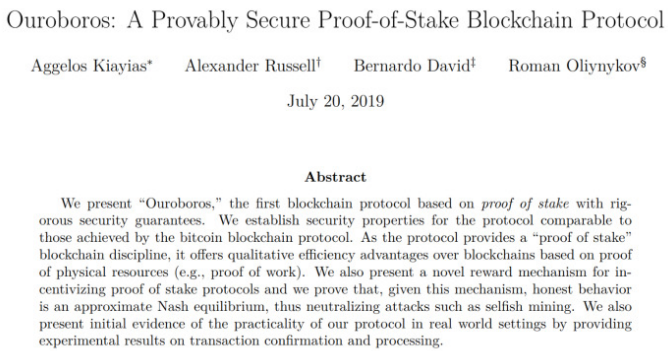
The Challenge: Real-time Distributed System

Cardano's Vision of a Distributed System

- It's a distributed system, but
  - Latency
  - Scalability
  - Fault Tolerance
  - Honest stake assumption
  - Real-time constraints
  - Adversarial behavior

Constraints, Assumptions, and Implementation Challenges in Cardano

Faithfulness to Research vs. Real-World Implementation



Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol

Aggelos Kiayias* Alexander Russell† Bernardo David‡ Roman Oliynykov§

July 20, 2019

Abstract

We present "Ouroboros," the first blockchain protocol based on *proof of stake* with rigorous security guarantees. We establish security properties for the protocol comparable to those achieved by the bitcoin blockchain protocol. As the protocol provides a "proof of stake" blockchain discipline, it offers qualitative efficiency advantages over blockchains based on proof of physical resources (e.g., proof of work). We also present a novel reward mechanism for incentivizing proof of stake protocols and we prove that, given this mechanism, honest behavior is an approximate Nash equilibrium, thus neutralizing attacks such as selfish mining. We also present initial evidence of the practicality of our protocol in real world settings by providing experimental results on transaction confirmation and processing.

Figure: Ouroboros paper

```
{- HLINT ignore "Fuse concatMap/map" -}
{- HLINT ignore "Redundant <$>" -}
{- HLINT ignore "Use fewer imports" -}

runNode
  :: PartialNodeConfiguration
  -> IO ()
runNode cmdPc = do
    installSigTermHandler

    Crypto.cryptoInit

    configYamlPc <- parseNodeConfigurationFP . getLast $ pncConfigFile cmdPc

    nc <- case makeNodeConfiguration $ defaultPartialNodeConfiguration <> configYamlPc <> cmdPc of
            Left err -> error $ "Error in creating the NodeConfiguration: " <> err
            Right nc' -> return nc'

    putStrLn $ "Node configuration: " <> show nc

    case shelleyVRFFile $ ncProtocolFiles nc of
      Just vrfFp -> do vrf <- runExceptT $ checkVRFFilePermissions (File vrfFp)
                       case vrf of
                         Left err ->
                           Text.putStrLn (renderVRFPrivateKeyFilePermissionError err) >> exitFailure
                         Right () ->
                           pure ()
      Nothing -> pure ()

    eitherSomeProtocol <- runExceptT $ mkConsensusProtocol
                                         (ncProtocolConfig nc)
                                         -- TODO: Convert ncProtocolFiles to Maybe as relay nodes
                                         -- don't need these.
                                         (Just $ ncProtocolFiles nc)

    p :: SomeConsensusProtocol <-
      case eitherSomeProtocol of
        Left err -> putStrLn (Api.displayError err) >> exitFailure
        Right p  -> pure p

    let networkMagic :: Api.NetworkMagic =
          case p of
            SomeConsensusProtocol _ runP ->
              let ProtocolInfo { pInfoConfig } = fst $ Api.protocolInfo @IO runP
              in getNetworkMagic $ Consensus.configBlock pInfoConfig

    case p of
      SomeConsensusProtocol blockType runP ->
        handleNodeWithTracers cmdPc nc p networkMagic blockType runP

-- | Workaround to ensure that the main thread throws an async exception on
-- receiving a SIGTERM signal.
installSigTermHandler :: IO ()
installSigTermHandler = do
#ifdef UNIX
  -- Similar implementation to the RTS's handling of SIGINT (see GHC's
  -- https://gitlab.haskell.org/ghc/ghc/-/blob/master/libraries/base/GHC/TopHandler.hs).
  runThreadIdWk <- mkWeakThreadId =<< myThreadId
  _ <- Signals.installHandler
    Signals.sigTERM
```

Figure: Code base

INPUT | OUTPUT

Well-Typed

Need to fill the gap

- Translate formal specification into code
- Highly concurrent logic
- High assurance needs (i.e. as verifiable and robust as possible)
- Clear code structures, predictable behavior

Strongly Statically Typed Purely Functional Programming

Haskell

Figure: Source

But:

- Memory consumption
- Debugging can be tricky

In a nutshell

Figure: Source

Tools & Techniques

Software Transactional Memory (STM)

- Compositional and modular concurrency
- Isolated from IO side effects

Abstraction & Libraries

- First to finish vs Last to finish STM semantics
- Typed-protocols
  - Type safe
  - Session types
  - Deadlock free!

```haskell
-- | Last-to-finish synchronisation.  It is the multiplicative semigroup of
-- the [near-semiring](https://www.wikiwand.com/en/Near-semiring) for which addition is
-- given by 'FirstToFinish'.
--
-- This is similar to 'Ap' (see 'LastToFinishM') in the sense that it will wait
-- for all monadic effects, but unlike 'Ap' it will not combine all results
-- using a monoid instance, but rather it will return the last one.  Also unlike
-- 'Ap' it does not have a monoidal unit.
--
-- > -- | Read all 'TMVar's and return the one that was filled last.
-- > --
-- > readAllTMVars :: Monoid a => NonEmpty (TMVar a) -> STM a
-- > readAllTMVars = runLastToFinish
-- >                 . foldMap1 (LastToFinish . readTMVar)
-- >                   -- from 'semigroupoids' package or use 'foldr1' and 'fmap'
-- >                   -- from 'base'
newtype LastToFinish m a = LastToFinish { runLastToFinish :: m a }
  deriving newtype Functor
  deriving         ( Generic
                   , Generic1
                   , Applicative
                   , Alternative
                   , Monad
                   , MonadPlus
                   , Traversable
                   )
  deriving Foldable via (Ap m)

instance MonadPlus m => Semigroup (LastToFinish m a) where
    (<>) :: MonadPlus m => LastToFinish m a -> LastToFinish m a -> LastToFinish m a
    LastToFinish left <> LastToFinish right = LastToFinish $ do
      a <- Left  <$> left
        <|> Right <$> right
      case a of
        Left  {} -> right
        Right {} -> left
```

3.13.2 State machine

Agency	
Client has Agency	StIdle
Server has Agency	StBusy

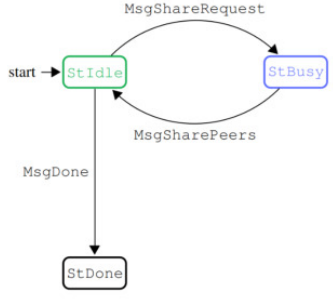


Figure 3.10: State machine of the peer sharing protocol.

INPUT | OUTPUT

Well-Typed

Testing approach

- Property based testing vs unit tests
  - Input random generation;
  - Shrinking;
  - Reproducibility;
  - Coverage checks.
- Simulation
  - Early detection of critical races;
  - Simulation of rare edge cases;
  - Mocking and error injection;
  - Simulate time passing;
  - Looking for different schedules.

Highlights from the Networking Team

Highlights

Good:

- Found several tricky edge case bugs
- Network is running for years non-stop
- Simulate years of total time per week!

But:

- We are far from perfect, we still fix and find small bugs every month related to networking

We try our best:

Our CI test suite runs on average between 1 and 5 hours of simulated time per test per input per PR per OS. So:

- Assuming around 100 tests in our test suite and
- each test generates 100 random inputs;
- Assuming 3 PRs per week and testing on Windows, OSX and Linux;
- Results on around **225 000 hours** of simulated time per week!

Takeaways

Takeaways

- Complex systems spans performance characteristics we can not control;
- Functional Programming, namely Haskell and its concurrency tools helps us manage complexity;
- We do our best in searching through all state space efficiently;
- Well-founded confidence;
- Progress is achievable.

Thank you!