Formally Specifying Blockchain Protocols

IOHK

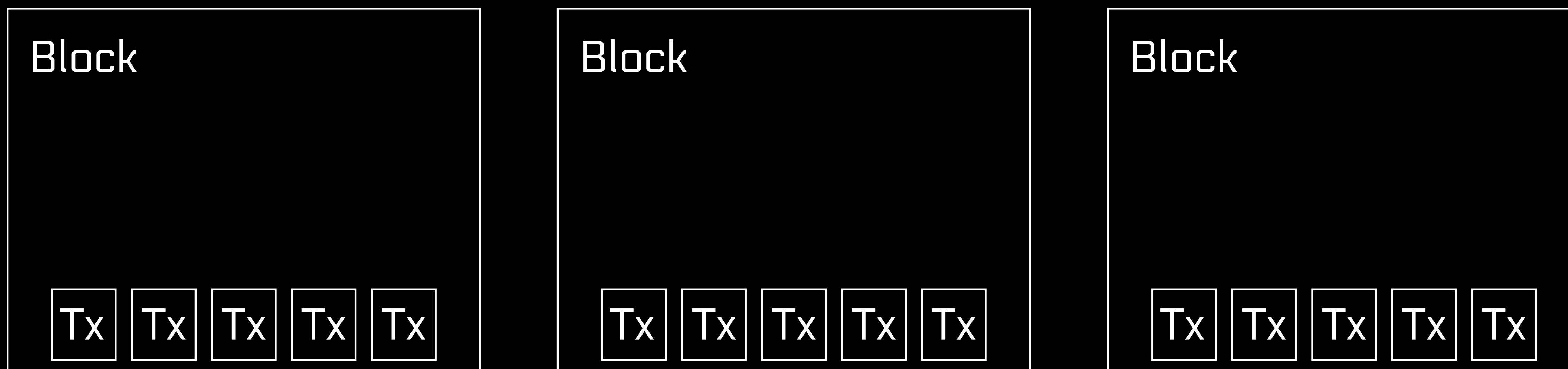
- company building blockchain applications
- research focused
- invested in functional programming
- built Cardano network, Ada cryptocurrency

Blockchain Protocols

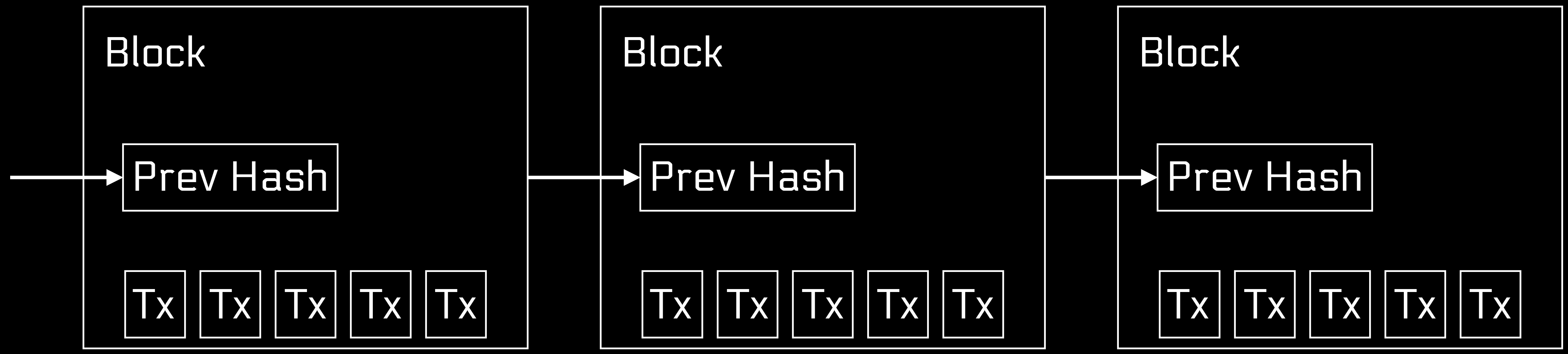
Permission-less Decentralised Ledger

- decentralisation
  no trusted authority
- permission-less
  anyone can join
- persistence
  established entries can not be deleted
- liveness
  entries submitted to the system will be included

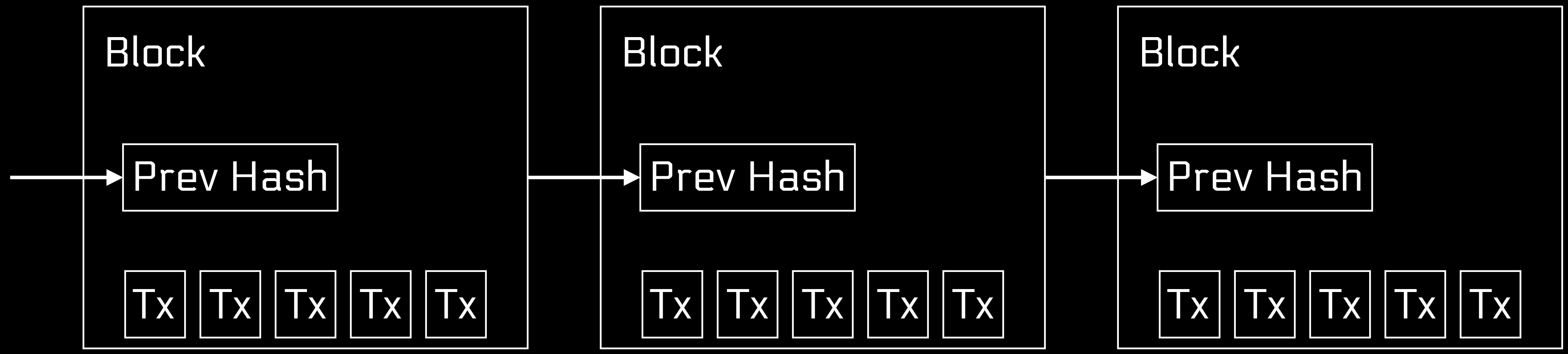
Bitcoin Blockchain



Bitcoin Blockchain

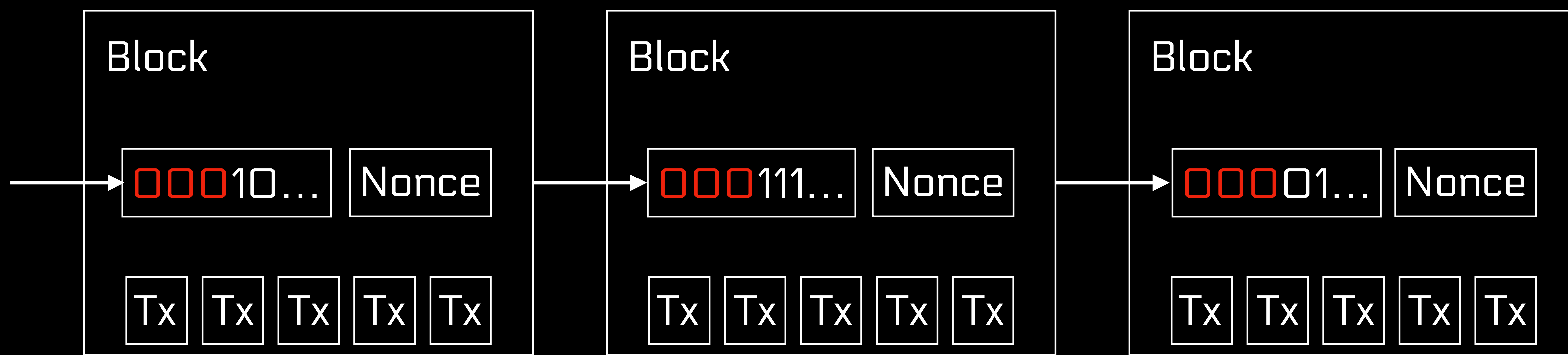


Bitcoin Blockchain



- split ledger into “blocks”
- everyone takes turns, assume honest majority
- permission-less: Sybil attack

Bitcoin Blockchain



- split ledger into “blocks”
- everyone takes turns, assume honest majority
- permission-less: Sybil attack

Proof of Work

Proof of Work

- randomised leader election, one CPU, one vote
  reward the winner

- longest chain wins: hard to revert old blocks
  unless you have >50% of CPUs

problems:

- huge energy consumption

- mining pools lead to centralisation

Proof of Stake

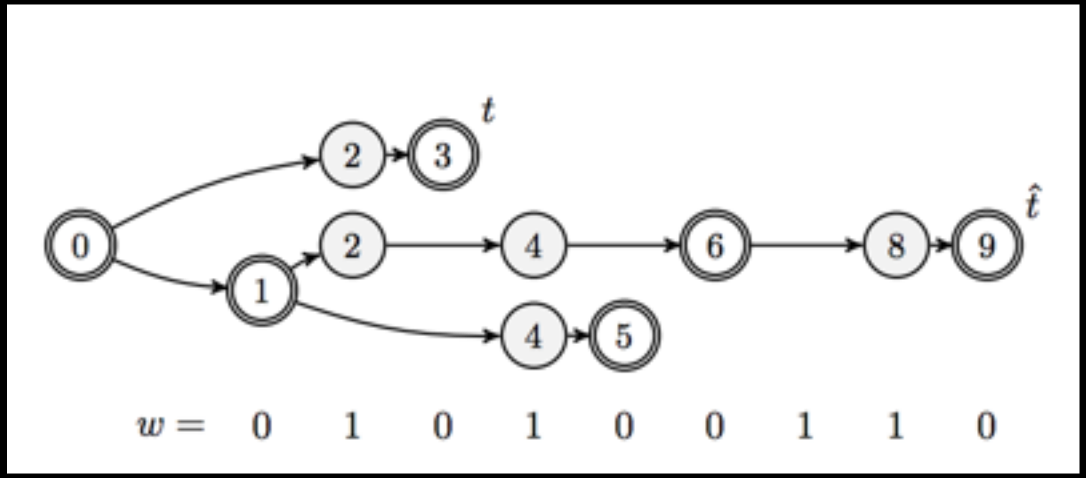
Different leader selection: weighted by stake

- each time slot, randomly pick one coin owner produces a block
- needs randomness, naive approaches vulnerable to grinding attack

Ouroboros

First Provably Secure Proof of Stake Protocol

- split time into slots, elect leader for each slot based on stake
- stakeholders are responsible for agreeing on randomness for next epoch
- proven secure against adversary with less than 50% stake



Adversary	BTC	OB Covert	OB General
0.10	50	3	5
0.15	80	5	8
0.20	110	7	12
0.25	150	11	18
0.30	240	18	31
0.35	410	34	60
0.40	890	78	148
0.45	3400	317	663

- running in production in Cardano

Ouroboros Praos

- extension of Ouroboros to semi-synchronous setting
- deal gracefully with message delay
- as delay increases, adversary grows stronger
- currently implementing for future versions of Cardano

From Paper to Implementation

Functionality $\mathcal{F}_{\text{KES}}$

$\mathcal{F}_{\text{KES}}$ is parameterized by the total number of signature updates T, interacting with a signer U_S and stakeholders U_i, as follows:

- **Key Generation.** Upon receiving a message (KeyGen, *sid*, U_S) from a stakeholder U_S, send (KeyGen, *sid*, U_S) to the adversary. Upon receiving (VerificationKey, *sid*, U_S, v) from the adversary, send (VerificationKey, *sid*, v) to U_S, record the triple (*sid*, U_S, v) and set counter $k_{ctr} = 1$.
- **Sign and Update.** Upon receiving a message (USign, *sid*, U_S, m, j) from U_S, verify that (*sid*, U_S, v) is recorded for some *sid* and that $k_{ctr} \le j \le T$. If not, then ignore the request. Else, set $k_{ctr} = j + 1$ and send (Sign, *sid*, U_S, m, j) to the adversary. Upon receiving (Signature, *sid*, U_S, m, j, σ) from the adversary, verify that no entry $(m, j, \sigma, v, 0)$ is recorded. If it is, then output an error message to U_S and halt. Else, send (Signature, *sid*, m, j, σ) to U_S, and record the entry $(m, j, \sigma, v, 1)$.
- **Signature Verification.** Upon receiving a message (Verify, *sid*, m, j, σ, v') from some stakeholder U_i do:
  1. If $v' = v$ and the entry $(m, j, \sigma, v, 1)$ is recorded, then set $f = 1$. (This condition guarantees completeness: If the verification key v' is the registered one and σ is a legitimately generated signature for m, then the verification succeeds.)
  2. Else, if $v' = v$, the signer is not corrupted, and no entry $(m, j, \sigma', v, 1)$ for any σ' is recorded, then set $f = 0$ and record the entry $(m, j, \sigma, v, 0)$. (This condition guarantees unforgeability: If v' is the registered one, the signer is not corrupted, and never signed m, then the verification fails.)
  3. Else, if there is an entry (m, j, σ, v', f') recorded, then let $f = f'$. (This condition guarantees consistency: All verification requests with identical parameters will result in the same answer.)
  4. Else, if $j < k_{ctr}$, let $f = 0$ and record the entry $(m, j, \sigma, v, 0)$. Otherwise, if $j = k_{ctr}$, hand (Verify, *sid*, m, j, σ, v') to the adversary. Upon re... (Verified, *sid*, m, j, ϕ) from the adversary let $f = \phi$ and record the entry (m, j, σ, v', ϕ). (This condition guarantees that the adversary is only able to forge signatures under keys b... t... corru...ed p...tie... for ...me ...riods corresponding to the current or future slots.)

  Output (Verified, *sid*, m, j, f) to U_i.

Fig. 1: Functionality $\mathcal{F}_{\text{KES}}$.

Theorem 4. *Let $f \in (0, 1]$, $\Delta \ge 1$, and ... be such that $\alpha(1 - ...)^{\Delta} = (1$ ... a string drawn from $\{0, 1, \perp\}^R$ according to $\mathcal{D}^f_\alpha$. Then we have* $\Pr[\mathrm{div}$...

Proof. Observe that $\mathrm{div}_0(\cdot)$ is monotone in the sense that if $\check{y}$ is a pre... this follows because any fork $\check{F} \vdash_0 \check{y}$ can be "extended" to a fork ... of $\check{F}$. Additionally, we note that $\mathrm{div}_0(\cdot)$ has a straightforward "Lips... then $\mathrm{div}_0(y) \le \mathrm{div}_0(\check{y}) + s$; this follows because any fork $F \vdash_0 y$ can ... by retaining only vertices labeled by $\check{y}$—this can trim no more than ...

In light of Lemma 1 we conclude that

$$\mathrm{div}_\Delta(w) \le \mathrm{div}_0(\rho_\Delta(w)) \le \mathrm{div}_0(\rho_\Delta(w)^{\lceil \Delta}) + \Delta \le \mathrm{div}_0 \ldots$$

where the last inequality follows because the random variable $\rho_\Delta($... length no more than R. As the random variables z_i are binomial wi... conclusion of Theorem 4 now follows directly from the assumption t... Theorem 3.

```
...Block
...lockWorkMode ctx m
...nqueueMsg m → MainBlockHeader → m (Map NodeId (m ∅))
announceBlock enqueue header = do
    logDebug $ sformat ("Announcing header to others: "%shortHashF)
        (headerHash header)
    enqueue (MsgAnnounceBlockHeader OriginSender) (λaddr _ → announceBlockDo addr)
  where
    announceBlockDo
        :: BlockWorkMode ctx m
        ⇒ NodeId → NonEmpty (Conversation m ∅)
    announceBlockDo nodeId = pure $ Conversation $ λcA → do
        SecurityParams{..} ← view (lensOf @SecurityParams)
        let throwOnIgnored nId =
                whenJust (nodeIdToAddress nId) $ λaddr →
                    whenM (shouldIgnoreAddress addr) $
                        throwM AttackNoBlocksTriggered
        when (AttackNoBlocks `elem` spAttackTypes) (throwOnIgnored nodeId)
        logDebug $
            sformat
                ("Announcing block "%shortHashF%" to "%build)
                (headerHash header)
                nodeId
        send cA $ MsgHeaders (one (Right header))
        handleHeadersCommunication cA
```

Paper and Implementation

publication

- high level of abstraction
- written in plain English and mathematical formulae
- has proofs of security

code

- deals with all the details
- written in Haskell
- proofs?

Small Steps – no Big Leap

- translate the algorithm to a formal language
  executable specification
  same level of abstraction

- small steps of incremental refinement
  small enough to verify/prove
  explicit design decisions

- simulate & test every refinement

Process Calculi

model distributed systems by processes and channels

- parallel and sequential composition
- sending and receiving data
- observational equivalence, bisimilarity
  equational reasoning

$$P \sim Q : P \to^{\alpha} P', Q \to^{\alpha} Q', P' \sim Q'$$

CCS, CSP, ACP, π-calculus

Psi Calculus

$\mathbf{0}$	Nil
$\overline{M}N\,.\,P$	Output
$\underline{M}(\lambda\tilde{x})N\,.\,P$	Input
$\mathbf{case}\ \varphi_1 : P_1\ [\!]\ \cdots\ [\!]\ \varphi_n : P_n$	Case
$(\nu a)P$	Restriction
$P \mid Q$	Parallel
$!P$	Replication
$(\!\mid \Psi \mid\!)$	Assertion

T	the (data) terms, ranged over by M, N
C	the conditions, ranged over by φ
A	the assertions, ranged over by Ψ

Parametric Family of Process Calculi

- specify types of terms, conditions, assertions
- well-established theory and tooling
  Psi Calculi Workbench

EDSL in Haskell

implement Psi calculus as EDSL in Haskell
write Ouroboros Praos in this language

starting point for

- simulations
- export to proof assistant (Isabelle, Coq)
- refine, add networking, … → production code

Psi in Haskell

```haskell
data Psi where
  Done    :: Psi                            -- completed process
  New     :: (Channel a → Psi) → Psi        -- create new unicast channel
  Inp     :: Channel a → (a → Psi) → Psi    -- unicast input
  Out     :: Channel a → a → Psi → Psi      -- unicast output
  Log     :: String → Psi bs → Psi bs       -- logging

-- interpreters
simulatePsi :: [Psi] → [String] -- simulate, print logs
exportPsi   :: [Psi] → [String] -- export to, say, Isabelle
runPsi      :: [Psi] → IO ∅    -- run concurrent processes
```

Psi in Haskell

```haskell
data Psi where
  Done    :: Psi                           -- completed process
  New     :: (Channel a → Psi) → Psi       -- create new unicast channel
  Inp     :: Channel a → (a → Psi) → Psi   -- unicast input
  Out     :: Channel a → a → Psi → Psi     -- unicast output
  Log     :: String → Psi bs → Psi bs      -- logging

-- interpreters
simulatePsi :: [Psi] → [String] -- simulate, print logs
exportPsi   :: [Psi] → [String] -- export to, say, Isabelle
runPsi      :: [Psi] → IO ∅     -- run concurrent processes
```

add broadcast channels, sub-processes

```haskell
data Psi :: [Type] → Type where
  Done    :: Psi bs                                      -- completed process
  New     :: Summarize a ⇒ (Unicast a → Psi bs) → Psi bs  -- create new unicast channel
  UInp    :: Unicast a → (a → Psi bs) → Psi bs            -- unicast input
  UOut    :: Unicast a → a → Psi bs → Psi bs              -- unicast output
  BInp    :: Broadcast bs a → (a → Psi bs) → Psi bs       -- broadcast input
  BOut    :: Broadcast bs a → a → Psi bs → Psi bs         -- broadcast output
  Fork    :: ProcId → Psi ^[] → Psi bs → Psi bs           -- fork new process
  Log     :: String → Psi bs → Psi bs                     -- logging
```

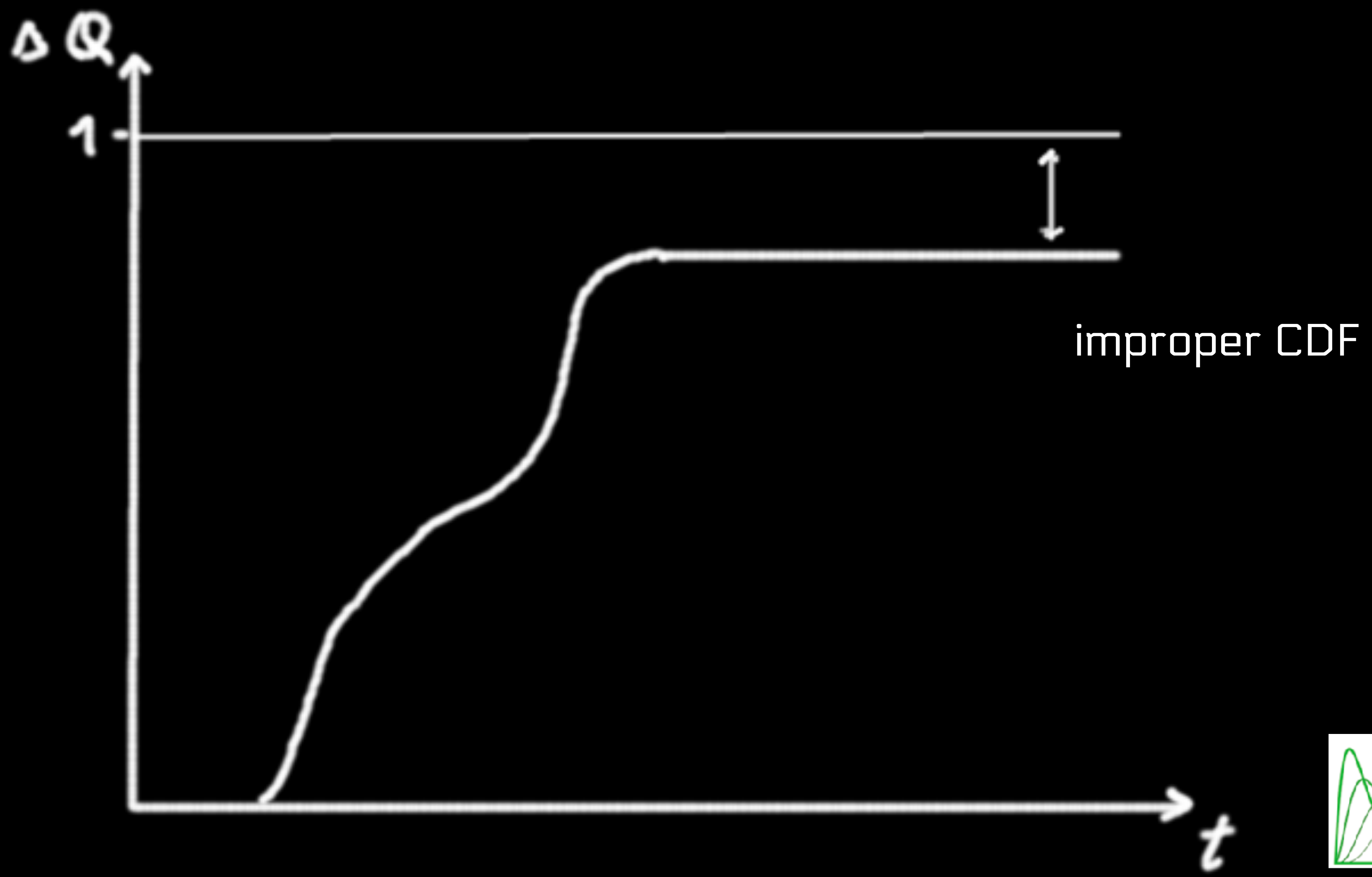
Modelling Performance

And Failure

Timeliness in Blockchains

- how long does it take for transactions to be recorded?
- how long does it take to join the network?
- can blocks propagate through the whole network in a single slot?
- what are the resource requirements for a node?

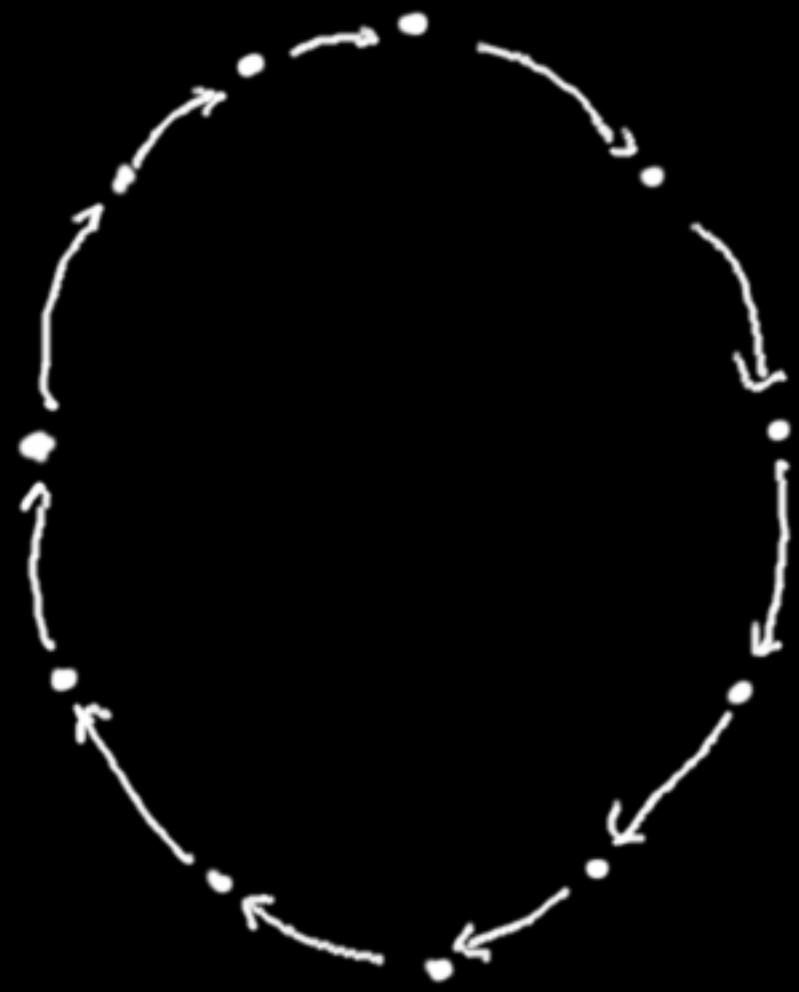
Impairment of Quality: ΔQ



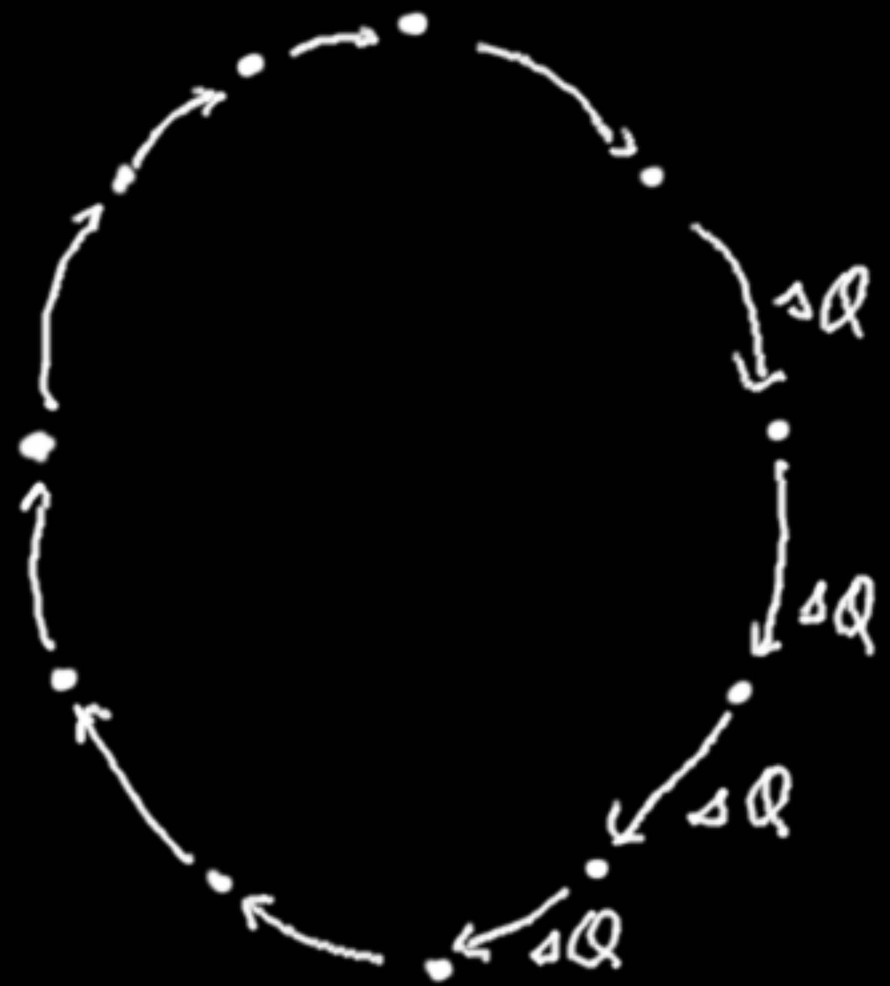
ΔQ in Haskell

```haskell
newtype DeltaQ = DeltaQ (StdGen → (Maybe Seconds, StdGen))

-- event happens exactly after s seconds
dirac :: Maybe Seconds → DeltaQ
dirac (Just s)
    | s < 0 = error "seconds must not be negative"
dirac s = DeltaQ $ λg → (s, g)

-- total reliability, total unreliability
miracle, never :: DeltaQ
miracle  = dirac $ Just 0
never    = dirac Nothing

-- uniform distribution
between :: (Seconds, Seconds) → DeltaQ

-- sequential composition
instance Monoid DeltaQ where
    mempty                        = miracle
    DeltaQ a `mappend` DeltaQ b   = DeltaQ $ λg →
        let (mda, g')   = a g
            (mdb, g'')  = b g'
            md          = (+) <$> mda <*> mdb
        in  (md, g'')

-- external choice
mix :: (DeltaQ, Rational) → (DeltaQ, Rational) → DeltaQ
```

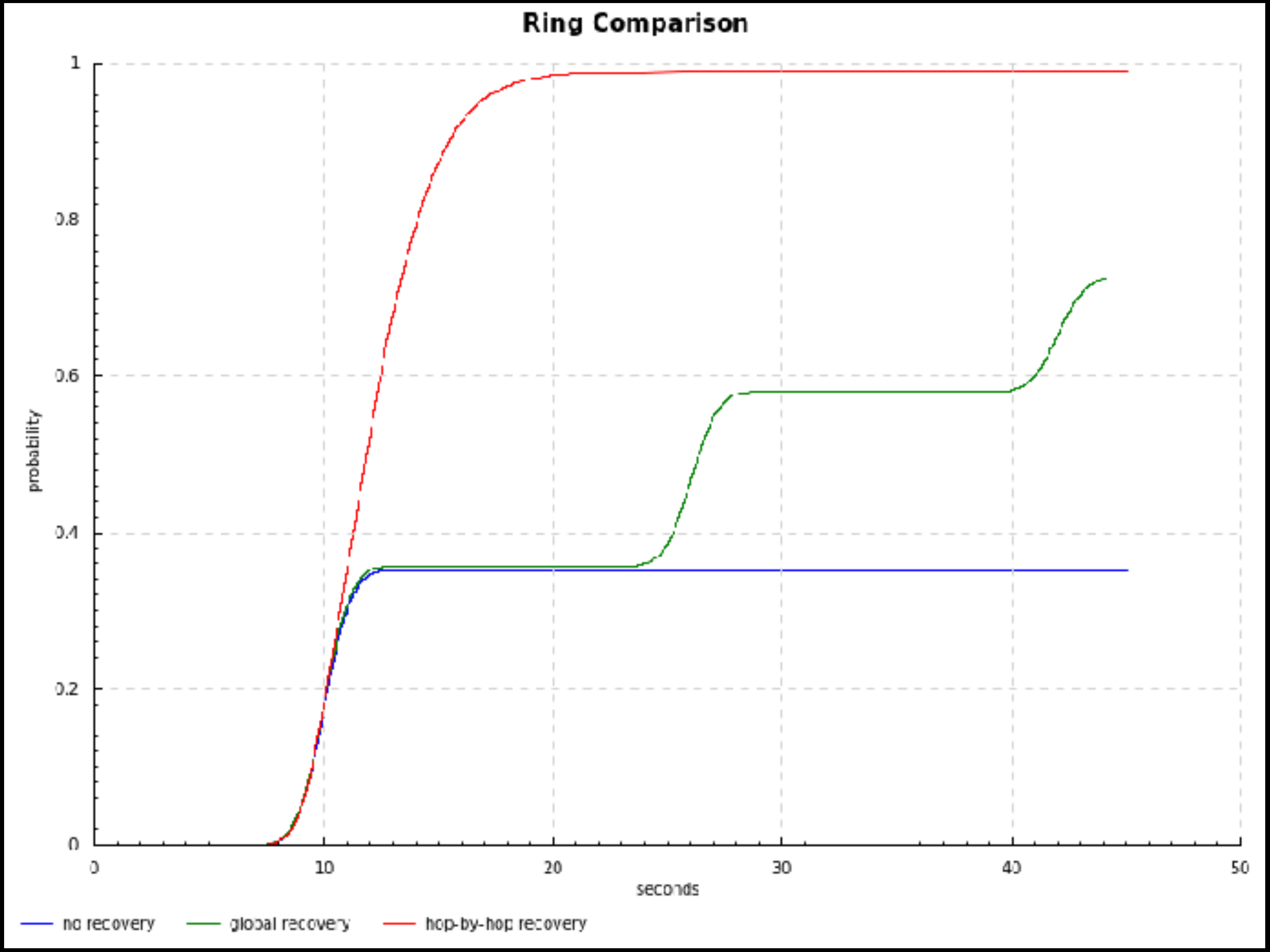
Example: Ring



Example: Ring



Example: Ring

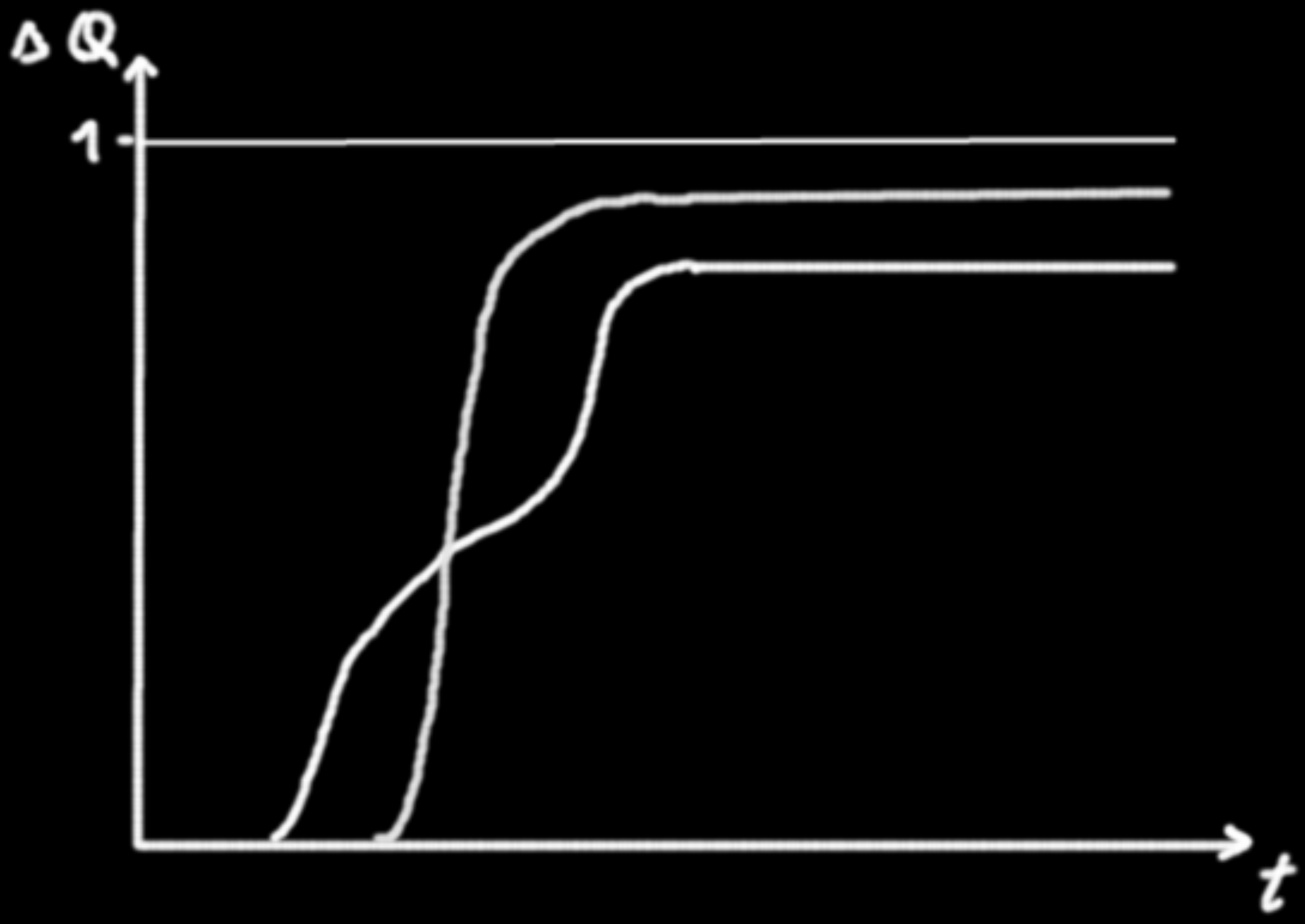


Annotate Psi

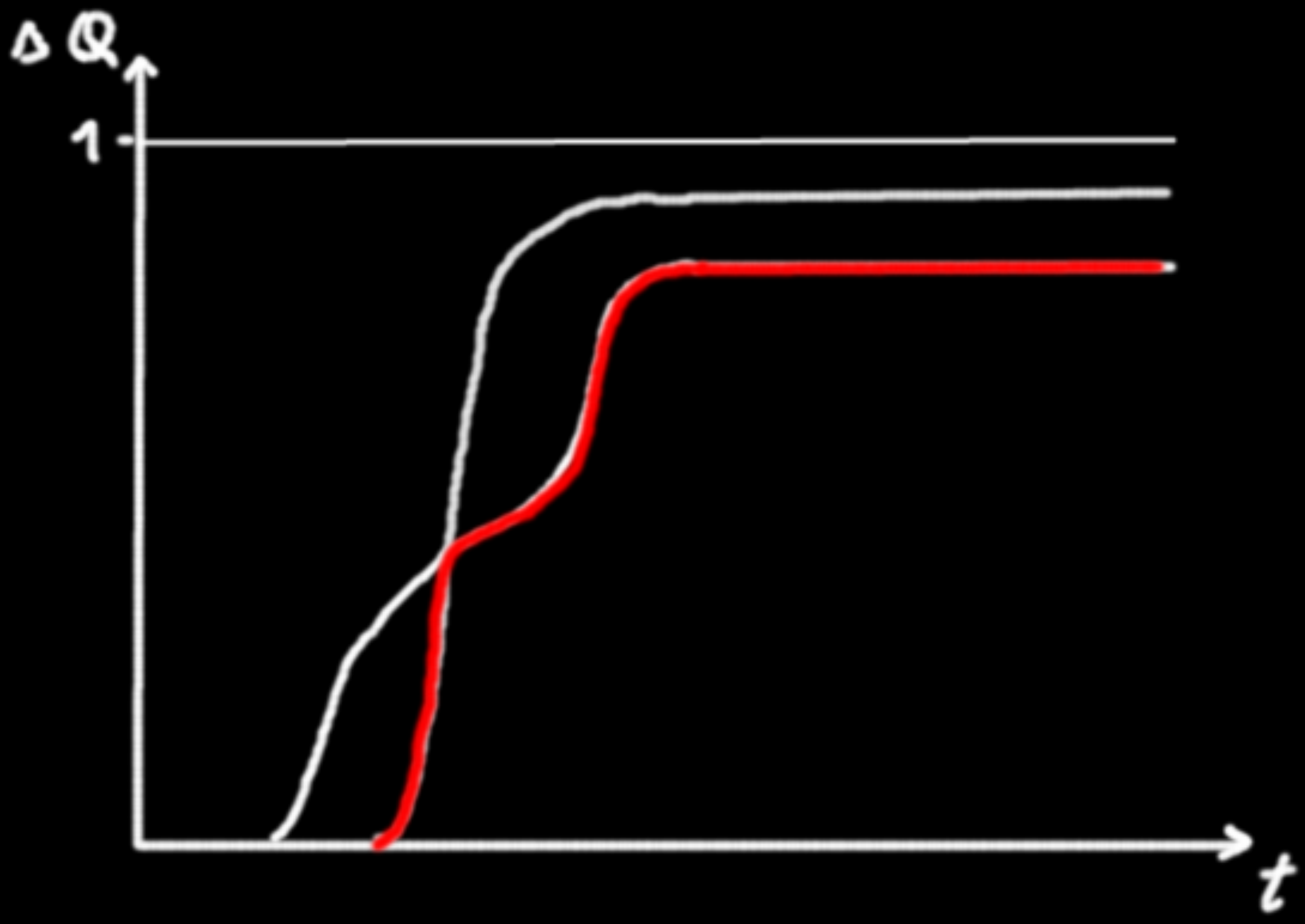
```
data Psi :: [Type] → Type where
  Done    :: Psi bs                                                        -- completed process
  New     :: Summarize a ⇒ (Unicast a → Psi bs) → Psi bs                   -- create new unicast channel
  UInp    :: Unicast a → Seconds → ((Maybe a, Seconds) → Psi bs) → Psi bs  -- unicast input
  UOut    :: Unicast a → DeltaQ → a → Psi bs → Psi bs                      -- unicast output
  BInp    :: Broadcast bs a → Seconds → ((Maybe a ,Seconds) → Psi bs) → Psi bs -- broadcast input
  BOut    :: Broadcast bs a → DeltaQ → a → Psi bs → Psi bs                 -- broadcast output
  Fork    :: ProcId → Psi ^□ → Psi bs → Psi bs                             -- fork new process
  Delay   :: Seconds → Psi bs → Psi bs                                     -- delay
  Observe :: Typeable a ⇒ a → Psi bs → Psi bs                              -- observing
  Log     :: String → Psi bs → Psi bs                                      -- logging
```

- simulations will take ΔQ annotations into account
- can be ignored when exporting to proof assistant

Composing ΔQ

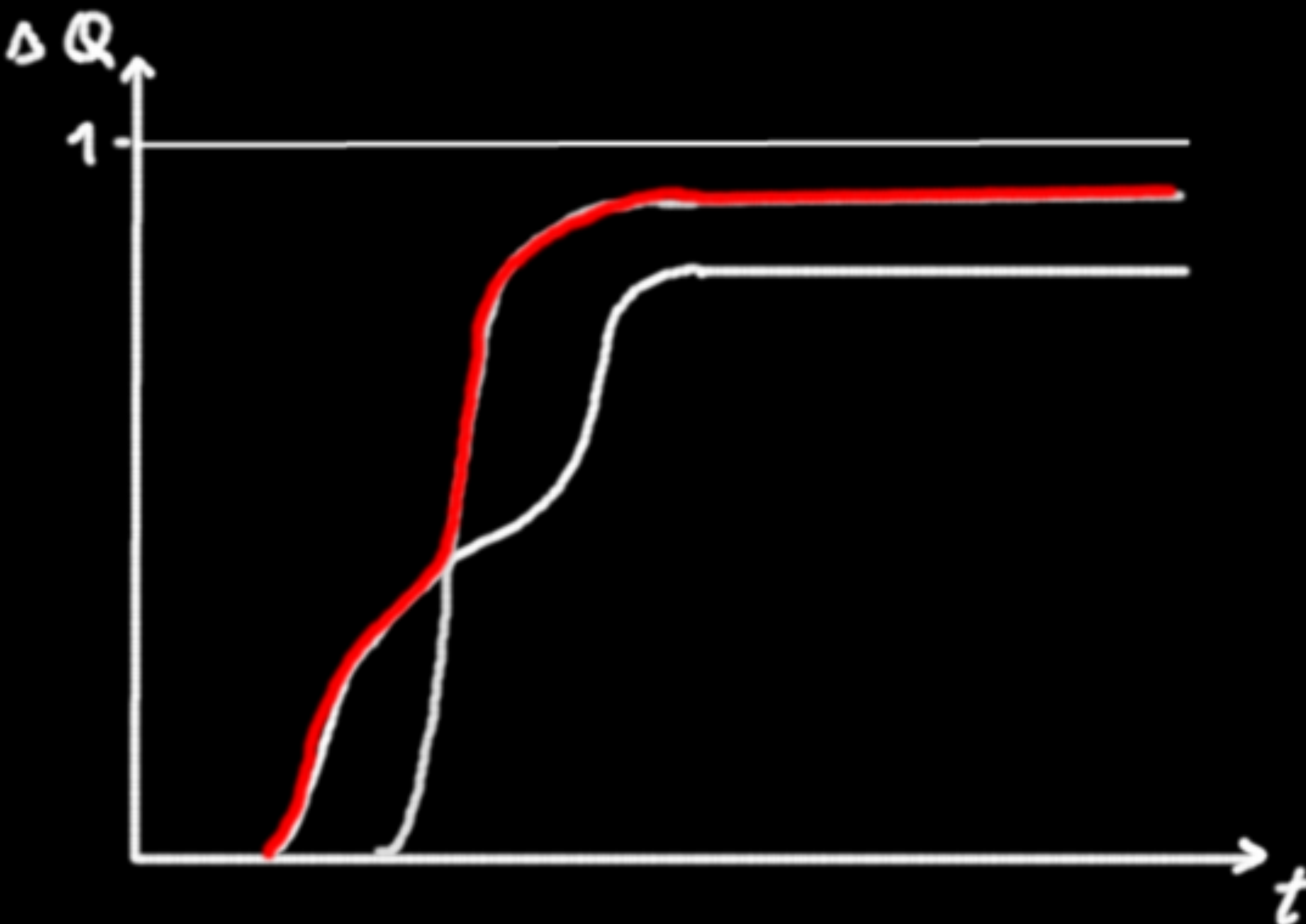


Composing ΔQ



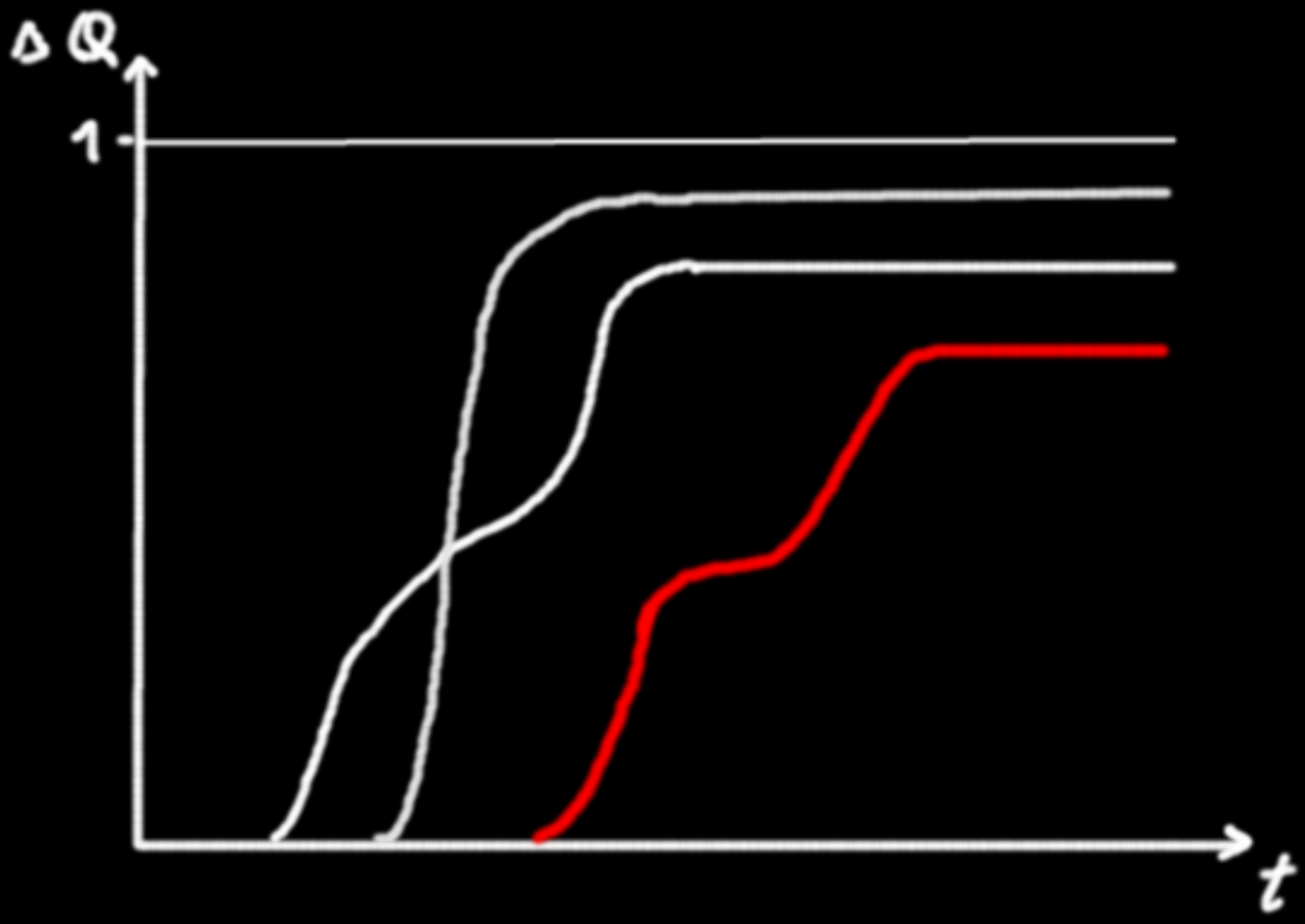
Last to Finish: min

Composing ΔQ



First to Finish: max

Composing ΔQ



Sequential Composition: Convolution

Symbolic ΔQ

```haskell
type Prob = Double

data DeltaQ =
      Exact Int                    -- Exactly $n$
    | Var String                   -- Variable
    | DeltaQ :> DeltaQ             -- Sequential composition
    | Ftf [DeltaQ]                 -- First-to-finish
    | Ltf [DeltaQ]                 -- Last-to-finish
    | PChoice [(Prob, DeltaQ)]     -- Probabilistic choice
    | DepFtf [(DeltaQ, DeltaQ)]    -- Dependent first-to-finish
```

- assign ΔQ terms to atomic operations, channels
- algebraic rules to manipulate ΔQ expressions
- complementary to simulations: see *why* performance is as it is

High-Assurance Blockchain Implementations

- cryptocurrencies carry large value
- blockchains proposed for other critical infrastructure (land deeds) needs to be fit for purpose

need high assurance

- peer reviewed, provably secure protocols
- high-assurance software development methodology
  take small steps from protocol to production code
- design for performance
- open repository: https://github.com/input-output-hk/ouroboros-spec