Curious properties of latency distributions

Michał J. Gajda

2021-02-19

Problem

How to describe performance of distributed system?

- network:
  - latency
  - bandwidth
- computation:
  - cryptographic signature checking
  - data structure lookup

Answer

- **Capacity-insensitive** networking:

  we do not care about bandwidth

Answer

- **Capacity-insensitive** networking:

  we do not care about bandwidth

- Computation:

  we measure time-to-completion

Answer

- **Capacity-insensitive** networking:

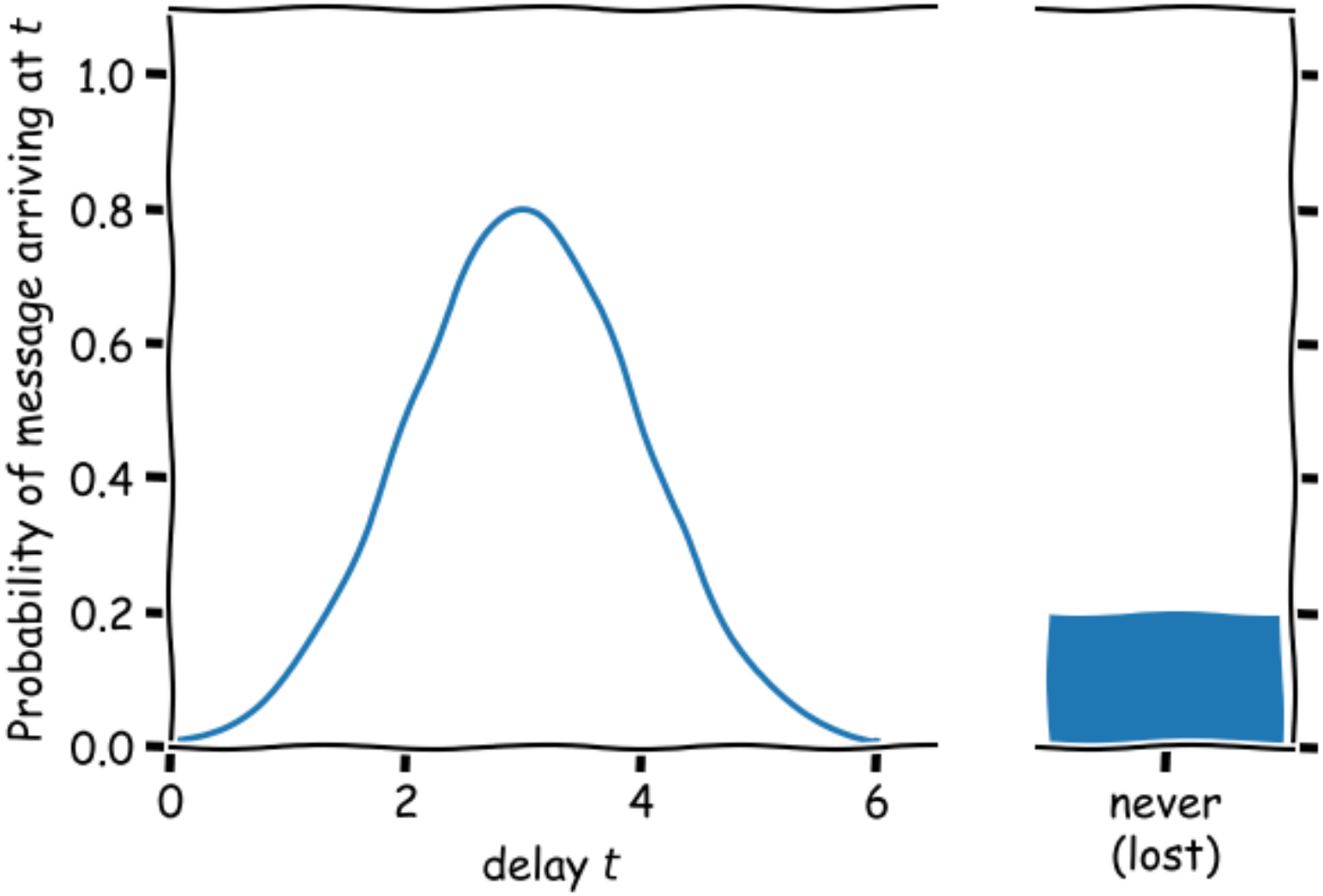
  we do not care about bandwidth
- Computation:

  we measure time-to-completion
- Everything is... **latency**

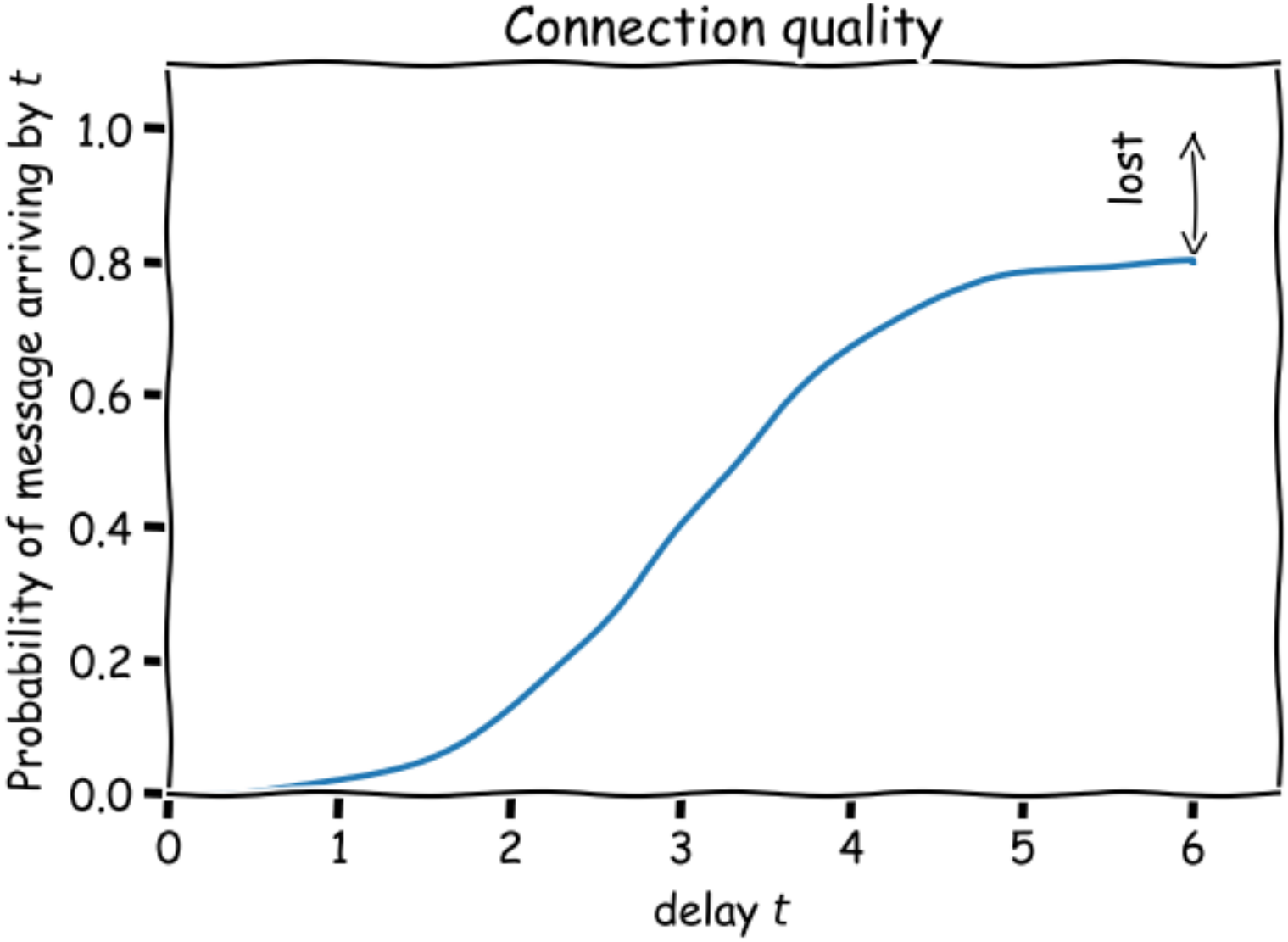
Plan

1. Why?
2. What is latency distribution?
3. Zeros and ones.
4. Sequential composition.
5. Parallel composition:
   - **first-to-finish** and **last-to-finish**
6. How to model a network?
7. Time service: **broadcast**
8. DynamoDB consensus: **n-out-of-k**

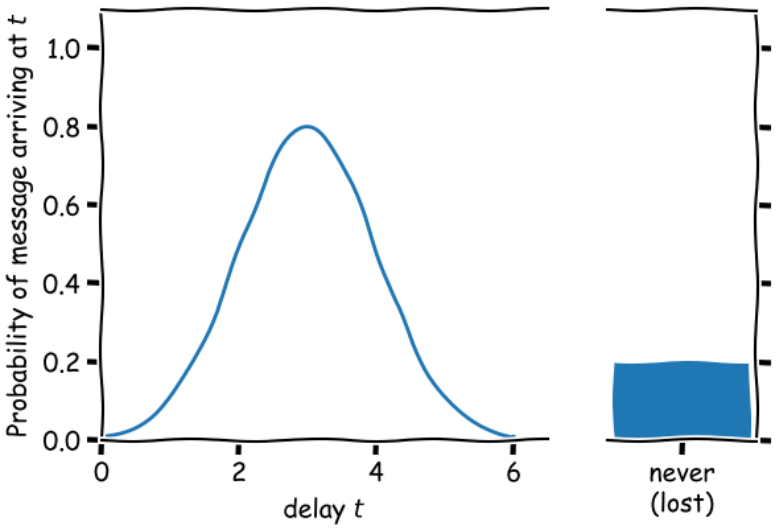
Generalized latency distribution: PDF



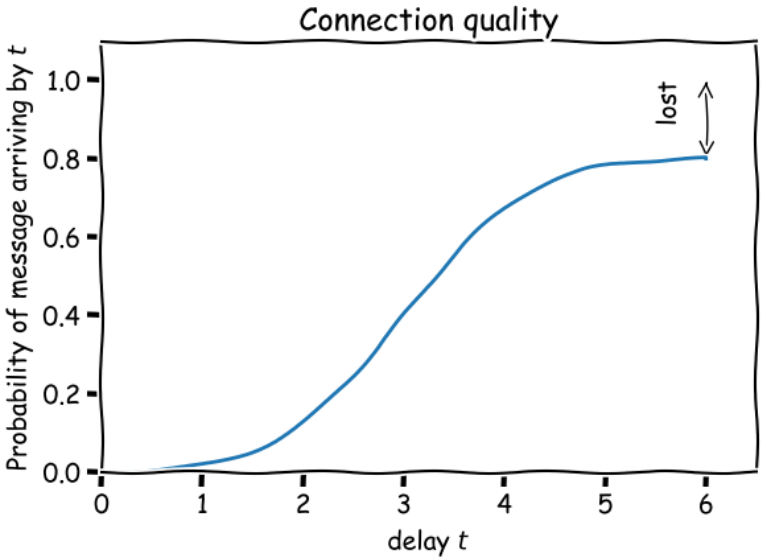
Ultimate arrival probability: CDF



Latency representations



is PDF



is CDF

Latency from PDF

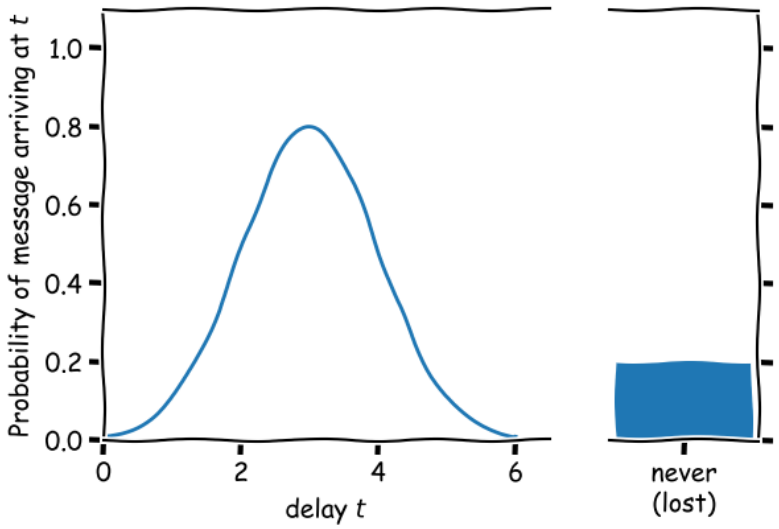


Figure 3: Latency distribution

```
newtype LatencyDistribution a =
  LatencyDistribution { pdf :: Series a }

newtype Series a = Series { unSeries :: [a] }
  deriving (Eq, Show, Read, Functor, Foldable,
            Applicative, Semigroup)
```

Finite difference and integration

Finite integration operator:

```
cumsum :: Num a => Series a -> Series a
cumsum = Series . tail . scanl (+) 0 . unSeries
```

Finite difference and integration

Finite integration operator:

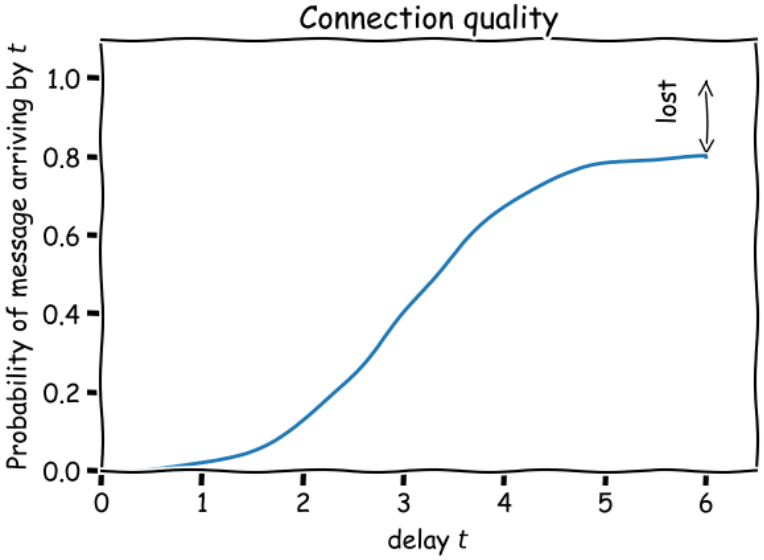
```
cumsum :: Num a => Series a -> Series a
cumsum = Series . tail . scanl (+) 0 . unSeries
```

Backwards finite difference operator:

```
diffEnc :: Num a => Series a -> Series a
diffEnc (Series []) = Series []
diffEnc (Series s ) = Series $
  zipWith (-) s (0:s)
```

```
cumsum  . diffEnc == id
diffEnc . cumsum  == id
```

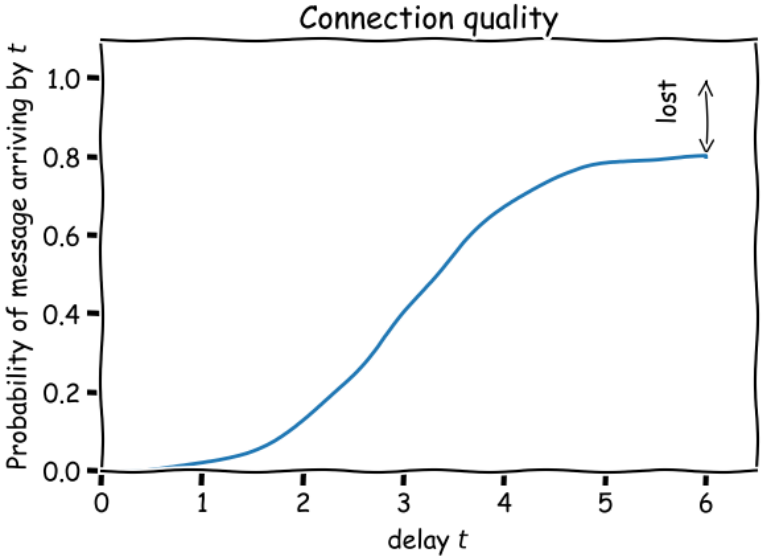
Latency from CDF



is CDF

```
cdf :: Num                      a
    => LatencyDistribution a
    -> Series                   a
cdf = cumsum . pdf
```

Latency from CDF



is CDF

```
cdf :: Num                      a
    => LatencyDistribution a
    -> Series                   a
cdf = cumsum . pdf

fromCDF :: Probability          a
        => Series               a
        -> LatencyDistribution a
fromCDF = fromPDF . diffEnc
```

Special: delay

- Delaying by t – composition with wait:

$$wait(t) = f(t) = \begin{cases} 0 & \text{for } t < t_d \\ 1.0 & \text{for } t = t_d \end{cases}$$

```
delayLD :: Probability a
        => Delay
        -> LatencyDistribution a
delayLD n = LatencyDistribution
        $ Series
        $ [0.0 | _ <- [(0::Delay)..n-1]] <> [1.0]
```

Special: attenuation

```
preserved :: Probability          a
          =>                      a
          -> LatencyDistribution a
preserved a = LatencyDistribution {
                pdf = Series [a]  }

allLostLD, noDelayLD :: Probability          a
                     => LatencyDistribution a
allLostLD = preserved 0.0
noDelayLD = preserved 1.0
```

Sequential composition

Sequential composition $\fatsemi$ or `afterLD`: given $\Delta Q_1(t)$ and $\Delta Q_2(t)$:

$$\Delta Q_1(t) \fatsemi \Delta Q_2(t)$$

.

- *associative*:

$$\Delta Q_1(t) \fatsemi [\Delta Q_2(t) \fatsemi \Delta Q_3(t)] = [\Delta Q_1(t) \fatsemi \Delta Q_2(t)] \fatsemi \Delta Q_3(t)$$

Sequential composition

Sequential composition $\fatsemi$ or `afterLD`: given $\Delta Q_1(t)$ and $\Delta Q_2(t)$:

$$\Delta Q_1(t) \fatsemi \Delta Q_2(t)$$

.

- *associative*:

$$\Delta Q_1(t) \fatsemi [\Delta Q_2(t) \fatsemi \Delta Q_3(t)] = [\Delta Q_1(t) \fatsemi \Delta Q_2(t)] \fatsemi \Delta Q_3(t)$$

- *commutative*

$$\Delta Q_1(t) \fatsemi \Delta Q_2(t) = \Delta Q_2(t) \fatsemi \Delta Q_1(t)$$

Sequential composition

Sequential composition $\fatsemi$ or `afterLD`: given $\Delta Q_1(t)$ and $\Delta Q_2(t)$:

$$\Delta Q_1(t) \fatsemi \Delta Q_2(t)$$

.

- *associative*:

$$\Delta Q_1(t) \fatsemi [\Delta Q_2(t) \fatsemi \Delta Q_3(t)] = [\Delta Q_1(t) \fatsemi \Delta Q_2(t)] \fatsemi \Delta Q_3(t)$$

- *commutative*

$$\Delta Q_1(t) \fatsemi \Delta Q_2(t) = \Delta Q_2(t) \fatsemi \Delta Q_1(t)$$

- *neutral element* is $1_{\mathcal{Q}}$ or `noDelay`, so:

$$\Delta Q(t) \fatsemi 1_{\mathcal{Q}} = 1_{\mathcal{Q}} \fatsemi \Delta Q(t) = \Delta Q(t)$$

Sequential composition

Sequential composition $\fatsemi$ or `afterLD`: given $\Delta Q_1(t)$ and $\Delta Q_2(t)$:

$$\Delta Q_1(t) \fatsemi \Delta Q_2(t)$$

.

- *associative*:

$$\Delta Q_1(t) \fatsemi [\Delta Q_2(t) \fatsemi \Delta Q_3(t)] = [\Delta Q_1(t) \fatsemi \Delta Q_2(t)] \fatsemi \Delta Q_3(t)$$

- *commutative*

$$\Delta Q_1(t) \fatsemi \Delta Q_2(t) = \Delta Q_2(t) \fatsemi \Delta Q_1(t)$$

- *neutral element* is $1_{\mathcal{Q}}$ or `noDelay`, so:

$$\Delta Q(t) \fatsemi 1_{\mathcal{Q}} = 1_{\mathcal{Q}} \fatsemi \Delta Q(t) = \Delta Q(t)$$

- *null element* is $0_{\mathcal{Q}}$ or `allLost`, so:

$$\Delta Q(t) \fatsemi 0_{\mathcal{Q}} = 0_{\mathcal{Q}} \fatsemi \Delta Q(t) = 0_{\mathcal{Q}}$$

Sequential composition in Haskell

```
afterLD :: Probability a
        => LatencyDistribution a
        -> LatencyDistribution a
        -> LatencyDistribution a
rd1 `afterLD` rd2 = fromPDF
                  $ pdf rd1 `convolve` pdf rd2
```

Convolution in Haskell

Convolution $\otimes$

```
Series              (f:fs)
  `convolve`
     gg@(Series (g:gs)) =
  Series
    (f*g :
      unSeries (f .* Series gs +
                (Series fs `convolve` gg)))
Series []     `convolve` _          = Series []
_             `convolve` Series [] = Series []
```

Thanks to Doug McIlroy "Power series, power serious"

Parallel alternative or first-to-finish

Alternative

$$\Delta Q_1(t) \vee \Delta Q_2(t)$$

:

- *associative*:

  $$\Delta Q_1(t) \vee [\Delta Q_2(t) \vee \Delta Q_3(t)] = [\Delta Q_1(t) \vee \Delta Q_2(t)] \vee \Delta Q_3(t)$$

Parallel alternative or first-to-finish

Alternative

$$\Delta Q_1(t) \vee \Delta Q_2(t)$$

:

- *associative*:

  $$\Delta Q_1(t) \vee [\Delta Q_2(t) \vee \Delta Q_3(t)] = [\Delta Q_1(t) \vee \Delta Q_2(t)] \vee \Delta Q_3(t)$$

- *commutative*

  $$\Delta Q_1(t) \vee \Delta Q_2(t) = \Delta Q_2(t) \vee \Delta Q_1(t)$$

Parallel alternative or first-to-finish

Alternative

$$\Delta Q_1(t) \vee \Delta Q_2(t)$$

:

- *associative*:

  $$\Delta Q_1(t) \vee [\Delta Q_2(t) \vee \Delta Q_3(t)] = [\Delta Q_1(t) \vee \Delta Q_2(t)] \vee \Delta Q_3(t)$$

- *commutative*

  $$\Delta Q_1(t) \vee \Delta Q_2(t) = \Delta Q_2(t) \vee \Delta Q_1(t)$$

- *neutral element* is $0_{\mathcal{Q}}$ or `allLost`, so:

  $$\Delta Q(t) \vee 0_{\mathcal{Q}} = 0_{\mathcal{Q}} \vee \Delta Q(t) = \Delta Q(t)$$

Parallel alternative or first-to-finish

Alternative

$$\Delta Q_1(t) \vee \Delta Q_2(t)$$

:

- *associative*:

  $$\Delta Q_1(t) \vee [\Delta Q_2(t) \vee \Delta Q_3(t)] = [\Delta Q_1(t) \vee \Delta Q_2(t)] \vee \Delta Q_3(t)$$

- *commutative*

  $$\Delta Q_1(t) \vee \Delta Q_2(t) = \Delta Q_2(t) \vee \Delta Q_1(t)$$

- *neutral element* is $0_{\mathcal{Q}}$ or `allLost`, so:

  $$\Delta Q(t) \vee 0_{\mathcal{Q}} = 0_{\mathcal{Q}} \vee \Delta Q(t) = \Delta Q(t)$$

- *null element* of `firstToFinish` is 1_Q:

  $$\Delta Q(t) \vee 1_{\mathcal{Q}} = 1_{\mathcal{Q}} \vee \Delta Q(t) = 1_{\mathcal{Q}}$$

Parallel alternative or first-to-finish

Alternative

$$\Delta Q_1(t) \vee \Delta Q_2(t)$$

:

- *associative*:

  $$\Delta Q_1(t) \vee [\Delta Q_2(t) \vee \Delta Q_3(t)] = [\Delta Q_1(t) \vee \Delta Q_2(t)] \vee \Delta Q_3(t)$$

- *commutative*

  $$\Delta Q_1(t) \vee \Delta Q_2(t) = \Delta Q_2(t) \vee \Delta Q_1(t)$$

- *neutral element* is $0_{\mathcal{Q}}$ or `allLost`, so:

  $$\Delta Q(t) \vee 0_{\mathcal{Q}} = 0_{\mathcal{Q}} \vee \Delta Q(t) = \Delta Q(t)$$

- *null element* of `firstToFinish` is $1_{\mathcal{Q}}$:

  $$\Delta Q(t) \vee 1_{\mathcal{Q}} = 1_{\mathcal{Q}} \vee \Delta Q(t) = 1_{\mathcal{Q}}$$

- *monotonically increasing*:

  $$\Delta Q_1(t) \vee \Delta Q_2(t) \geq \Delta Q_1(t)$$

Parallel conjunction or last-to-finish

$$\Delta Q_1(t) \wedge \Delta Q_2(t)$$

- *associative*:

  $$\Delta Q_1(t) \wedge [\Delta Q_2(t) \wedge \Delta Q_3(t)] = [\Delta Q_1(t) \wedge \Delta Q_2(t)] \wedge \Delta Q_3(t)$$

Parallel conjunction or last-to-finish

$$\Delta Q_1(t) \wedge \Delta Q_2(t)$$

- *associative*:

  $$\Delta Q_1(t) \wedge [\Delta Q_2(t) \wedge \Delta Q_3(t)] = [\Delta Q_1(t) \wedge \Delta Q_2(t)] \wedge \Delta Q_3(t)$$

- *commutative*

  $$\Delta Q_1(t) \wedge \Delta Q_2(t) = \Delta Q_2(t) \wedge \Delta Q_1(t)$$

Parallel conjunction or last-to-finish

$$\Delta Q_1(t) \wedge \Delta Q_2(t)$$

- *associative*:

  $$\Delta Q_1(t) \wedge [\Delta Q_2(t) \wedge \Delta Q_3(t)] = [\Delta Q_1(t) \wedge \Delta Q_2(t)] \wedge \Delta Q_3(t)$$

- *commutative*

  $$\Delta Q_1(t) \wedge \Delta Q_2(t) = \Delta Q_2(t) \wedge \Delta Q_1(t)$$

- *neutral element* is $0_{\mathcal{Q}}$ or `allLost`, so:

  $$\Delta Q(t) \wedge 0_{\mathcal{Q}} = 0_{\mathcal{Q}} \wedge \Delta Q(t) = \Delta Q(t)$$

Parallel conjunction or last-to-finish

$$\Delta Q_1(t) \wedge \Delta Q_2(t)$$

- *associative*:

  $$\Delta Q_1(t) \wedge [\Delta Q_2(t) \wedge \Delta Q_3(t)] = [\Delta Q_1(t) \wedge \Delta Q_2(t)] \wedge \Delta Q_3(t)$$

- *commutative*

  $$\Delta Q_1(t) \wedge \Delta Q_2(t) = \Delta Q_2(t) \wedge \Delta Q_1(t)$$

- *neutral element* is $0_{\mathcal{Q}}$ or `allLost`, so:

  $$\Delta Q(t) \wedge 0_{\mathcal{Q}} = 0_{\mathcal{Q}} \wedge \Delta Q(t) = \Delta Q(t)$$

- *null element* of `firstToFinish` is 1_Q:

  $$\Delta Q(t) \wedge 1_{\mathcal{Q}} = 1_{\mathcal{Q}} \wedge \Delta Q(t) = 1_{\mathcal{Q}}$$

Parallel conjunction or last-to-finish

$$\Delta Q_1(t) \wedge \Delta Q_2(t)$$

- *associative*:

  $$\Delta Q_1(t) \wedge [\Delta Q_2(t) \wedge \Delta Q_3(t)] = [\Delta Q_1(t) \wedge \Delta Q_2(t)] \wedge \Delta Q_3(t)$$

- *commutative*

  $$\Delta Q_1(t) \wedge \Delta Q_2(t) = \Delta Q_2(t) \wedge \Delta Q_1(t)$$

- *neutral element* is $0_{\mathcal{Q}}$ or `allLost`, so:

  $$\Delta Q(t) \wedge 0_{\mathcal{Q}} = 0_{\mathcal{Q}} \wedge \Delta Q(t) = \Delta Q(t)$$

- *null element* of `firstToFinish` is 1_Q:

  $$\Delta Q(t) \wedge 1_{\mathcal{Q}} = 1_{\mathcal{Q}} \wedge \Delta Q(t) = 1_{\mathcal{Q}}$$

- *monotonically increasing*:

  $$\Delta Q_1(t) \wedge \Delta Q_2(t) \geq \Delta Q_1(t)$$

Parallel conjunction or last-to-finish in Haskell

Easy to compute on CDFs:

$$\mathrm{P}_{max(a,b)}(x \leq t) = P_a(x \leq t) * P_b(x \leq t)$$

```
lastToFinishLD :: Probability a
               => LatencyDistribution a
               -> LatencyDistribution a
               -> LatencyDistribution a
rd1 `lastToFinishLD` rd2 = fromCDF
                         $ cdf rd1' .*. cdf rd2'
  where
    (rd1', rd2') = extendToSameLengthLD (rd1, rd2)
```

Algebra

```
class TimeToCompletion ttc where
  firstToFinish :: ttc -> ttc -> ttc
  lastToFinish  :: ttc -> ttc -> ttc
  after         :: ttc -> ttc -> ttc
  delay         :: Delay -> ttc
  allLost       :: ttc
  noDelay       :: ttc
  noDelay        = delay 0

infixr 7 `after`
infixr 5 `firstToFinish`
infixr 5 `lastToFinish`
```

Approximation: latest arrival

```
newtype SometimeOrNever =
        SometimeOrNever
          { unSometimeOrNever :: Maybe Delay }
  deriving (Eq)

pattern Never      = SometimeOrNever Nothing
pattern Sometime t = SometimeOrNever (Just t)
```

Approximation: latest arrival

```
newtype SometimeOrNever =
        SometimeOrNever
          { unSometimeOrNever :: Maybe Delay }
  deriving (Eq)

pattern Never       = SometimeOrNever Nothing
pattern Sometime t = SometimeOrNever (Just t)

newtype Latest =
        Latest { unLatest :: SometimeOrNever }
  deriving (Eq, Ord, Show)

instance TimeToCompletion Latest where
  firstToFinish = min
  lastToFinish  = max
  after         = liftM2 (+)
  delay         = Sometime
  allLost       = Never
```

How to model a network?

Upper triangular matrix of $A_{i,j}$

Broadcasting over network in N steps:

$$\begin{aligned} R_0(A) &= 1 \\ R_n(A) &= R_{n-1}(A) * A \end{aligned}$$

Matrix multiplication $|*|$ with $(+, *)$ replaced by $(\vee, ;)$.

Time service: **broadcast**

We broadcast current timestamp to N nodes, without correction:

Limit of the series $R_n(A) = A^n$ is called A^*.

```
optimalConnections  ::
    (Probability                          a
    ,KnownNat n
    ,Real                                 a
    ,Metric                               a)
  => SMatrix  n (LatencyDistribution a)
  -> SMatrix  n (LatencyDistribution a)
optimalConnections a =
  converges (fromIntegral $ size a)
                          (|*|a) a
```

DynamoDB database consensus

- There are n nodes.

DynamoDB database consensus

- There are n nodes.
- k nodes are needed for write consensus.

DynamoDB database consensus

- There are n nodes.
- k nodes are needed for write consensus.
- $n - k + 1$ nodes are needed for read consensus.

DynamoDB consensus: **n-out-of-k**

$$F_{\binom{a_n}{k}}(t) = a_n \wedge F_{\binom{a_{n-1}}{k-1}}(t) \oplus (\overline{a_n} \wedge F_{\binom{a_{n-1}}{k}}(t))$$

where:

- sum of mutually exclusive event latencies $\oplus$
- $\otimes$ is convolution
- $[\overline{a_n}, a_n]$ is two element series having complement of a_n as a first term, and a_n as a second term.

DynamoDB consensus in Haskell

```
kOutOfN :: (TimeToCompletion a
           ,ExclusiveSum     a
           ,Complement       a
           ) => Series       a
             -> Series       a
kOutOfN (Series []    ) =
  error "kOutOfN of empty series"
kOutOfN (Series [x]   ) = [complement x, x]
kOutOfN (Series (x:xs)) = [x] `convolution`
                          Series xs
  where
    convolution = convolve_ exAdd lastToFinish
                  `on` kOutOfN
```

Conclusion

- Modeling distributed system with a simple approximation.

Conclusion

- Modeling distributed system with a simple approximation.
- Easy to implement

Conclusion

- Modeling distributed system with a simple approximation.
- Easy to implement
- Deep insights by computing latency distributions.

Conclusion

- Modeling distributed system with a simple approximation.
- Easy to implement
- Deep insights by computing latency distributions.
- Deep maths is easier to grok as a program

Conclusion

- Modeling distributed system with a simple approximation.
- Easy to implement
- Deep insights by computing latency distributions.
- Deep maths is easier to grok as a program
- QuickCheck can quickly test math laws.