Curious properties of latency distributions

MICHAŁ J. GAJDA

Network latency distributions, their algebra, and use examples.

1 INTRODUCTION

Here we explain the network simulation context which gave context to this paper and give key references.

Capacity insensitive networking Most of the network connections on the internet are called mice [2] for a good reason: they have bandwidth-latency product of less than 12k, and thus are largely insensitive to the network’s capacity. The deciding factor in performance of these flows is thus latency, hence we ignore capacity limitation for the remainder of this paper.

Packet loss modelling using improper CDFs In order to accurately simulate capacity-insensitive network miniprotocols we formally define network latency distribution as improper CDF (cumulative distribution function) of arrived messages over time. We call it improper CDF, because it does not end at 100%: some messages can be lost.

Time-limited model For practical purposes we ignore answers that are delivered after certain deadline: that is network connection timeout.

Starting with description of its apparent properties, we identify their mathematical definitions, and ultimately arrive at algebra of $\Delta Q$ [3] with basic operations that correspond to abstract interpretations of network miniprotocols[8].

This allows us to use objects from single algebraic body to describe behaviour of entire protocols as improper CDFs.

1.1 Related work

Then we discuss expansion of the concept to get most sensitive metrics of protocol and network robustness[4]. However instead of heuristic measure like effective graph resistance [5], we use logically justified measure derived from the actual behaviour of the network.

This is similar to network calculus[6] but uses simpler methods and uses more logical description with improper latency distribution functions, instead of max-plus and min-plus algebras.¹ Basic operations $\land$ and $\lor$ are similar to last-to-finish and first-to-finish synchronizations [3].

This approach allows us to use abstract interpretation[8] of computer program to get its latency distribution, or a single execution to approximate latency distribution assuming the same loss profile of packets.

2 PRELIMINARIES

2.0.1 Nulls and units of multiplication. We will be interested in null and unit of a single multiplication for each modulus we will consider:

```
class Unit a where
  unitE :: a

class Null a where
```

¹We describe how it generalizes these max-plus and min-plus algebras later.

Author’s address: Michał J. Gajda, mjgajda@migamake.com.
nullE :: a
instance Unit Int where
  unitE = 1
instance Null Int where
  nullE = 0

2.1 Discrete delay
Discrete delays are defined as:

newtype Delay = Delay { unDelay :: Int }

For ease of implementation, we express each function as a series of values for discrete delays. First value is for no delay. We define \( \text{start} \in \mathcal{T} \) as smallest Delay (no delay).

\[
\text{start} :: \text{Delay}
\]
\[
\text{start} = \text{Delay} 0
\]

2.2 Power series representing distributions
We follow [7] exposition of power series, but use finite series and shortcut evaluation:

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

To have a precise discrete approximation of distributions, we encode them as a series. Finite series can be represented by a generating function \( F_t \):

\[
F(t) = f_0 \cdot t^0 + f_1 \cdot t^1 + f_2 \cdot t^2 + \ldots + f_n \cdot t^n
\]

Which is represented by the Haskell data structure:

\[
f \_ t = \text{Series} [a_0, a_1, \ldots, a_n]
\]

2.2.1 Differential encoding and cumulative sum. For any probability distribution, we need a notion of integration, that converts probability distribution function (PDF) into cumulative distribution function (CDF).

Cumulative sums computes sums of 1..n-th term of the series:

\[
\text{cumsum} :: \text{Num a} \Rightarrow \text{Series a} \rightarrow \text{Series a}
\]
\[
\text{cumsum} = \text{Series} \cdot \text{tail} \cdot \text{scanl} (+) 0 \cdot \text{unSeries}
\]

After defining our integration operator, it is not time for its inverse. Differential encoding is lossless correspondent of discrete differences,\(^2\) but with first coefficient being copied. (Just like there was a zero before each series, so that we always preserve information about the first term.) This is \emph{backward antidifference}\(^3\) as defined by [11].

That makes it an inverse of cumsum. It is \emph{backward finite difference operator}, as defined by [10].

\[
\text{diffEnc} :: \text{Num a} \Rightarrow \text{Series a} \rightarrow \text{Series a}
\]
\[
\text{diffEnc} (\text{Series []}) = \text{Series []}
\]
\[
\text{diffEnc} (\text{Series s}) = \text{Series} \circ \text{head s : zipWith} (-) (\text{tail s}) s
\]

\(^2\)Usually called \emph{finite difference operators}.

\(^3\)Antidifference is an inverse of finite difference operator. Backwards difference subtracts immediate \emph{predecessor} from a successor term in the series.
So that \texttt{diffEnc} of CDF will get PDF, and \texttt{cumsum} of PDF will get CDF.

Since we are only interested in answers delivered before certain deadline, we will sometimes cut series at a given index:

\[
\text{cut} :: \text{Delay} \rightarrow \text{Series a} \rightarrow \text{Series a}
\]

\[
\text{cut} (\text{Delay } t) (\text{Series s}) = \text{Series (take } t \text{ s)}
\]

\textbf{instance} \text{IsList (Series a)} \textbf{where}

\textit{type} Item (Series a) = a
\textit{fromList} = Series
tolist (Series s) = s

Series enjoy few different multiplication operators.

Simplest is scalar multiplication:

\textbf{infixl 7 \texttt{\* -- same precedence as \*}}
\texttt{(\*): Num a => a \rightarrow Series a \rightarrow Series a}
\texttt{c \texttt{\* (Series (f:fs))} = Series (c\texttt{\* f : unSeries (c \texttt{\* Series fs))}}
\texttt{\_ \texttt{\* (Series [])} = Series []}

\[
F(t) = f_0 \ast t^0 + f_1 \ast t^1 + f_2 \ast t^2 + \ldots + f_n \ast t^n
\]

Second multiplication is convolution, which most commonly used on distributions:

\[
F(t) \otimes G(t) = \Sigma_{t=0}^{\infty} x^t \ast f(t) \ast g(t - \tau)
\]

Wikipedia’s definition:

\[
(f \otimes g)(t) \triangleq \int_{-\infty}^{\infty} f(\tau)g(t - \tau) \, d\tau.
\]

Distribution is from 0 to +\infty: Wikipedia’s definition:

1. First we fix the boundaries of integration:

\[
(f \otimes g)(t) \triangleq \int_{0}^{\infty} f(\tau)g(t - \tau) \, d\tau.
\]

(Assuming \(f(t) = g(t) = 0\) when \(t < 0\).)

2. Now we change to discrete form:

\[
(f \otimes g)(t) \triangleq \Sigma_{\tau=0}^{\infty} f(\tau)g(t - \tau)
\]

3. Now we notice that we approximate functions up to term \(n\):

\[
(f \otimes g)(t) \triangleq \Sigma_{\tau=0}^{n} f(\tau)g(t - \tau)
\]

Resulting in convolution:

\[
F(t) \otimes G(t) = \Sigma_{t=0}^{\infty} x^t \ast f(\tau) \ast g(t - \tau)
\]

\textbf{infixl 7 `convolve` -- like multiplication}
\texttt{convolve :: Num a}
\texttt{=> Series a}
\texttt{=> Series a}
\texttt{=> Series a}
\texttt{Series (f:fs) `convolve`
\[ gg@,(Series (g; gs)) = \]
\[ Series \]
\[ (f \cdot g : \]
\[ unSeries (f \cdot Series gs + \]
\[ (Series fs \cdot convolve \cdot gg)) \]
\[ Series [] \cdot convolve \cdot Series [] = Series [] \]
\[ _ \cdot convolve \cdot Series [] = Series [] \]

Elementwise multiplication, assuming missing terms are zero.

\[ (\cdot \cdot) :: Num a \]
\[ => Series a \]
\[ -> Series a \]
\[ -> Series a \]
\[ Series a \cdot \cdot Series b = Series (zipWith (\cdot\cdot) a b) \]

Since we use finite series, we need to extend their length when operation is done on series of different length.

Note that for emphasis, we also allow convolution with arbitrary addition and multiplication:

\[ convolve_ :: (a -> a -> a) \]
\[ -> (a -> a -> a) \]
\[ -> Series a -> Series a -> Series a \]
\[ convolve_ (+) (+) (Series (f:fs)) gg@ (Series (g; gs)) = \]
\[ Series \]
\[ (f \cdot g : \]
\[ zipWithExpanding \]
\[ (+) \]
\[ (f \cdot gs) \]
\[ (unSeries (convolve_ (+) (+)) \]
\[ (Series fs) gg)) \]

where
\[ a \cdot bs = (a \cdot) <$> bs \]
\[ convolve_ _ _ (Series []) _ = Series [] \]
\[ convolve_ _ _ (Series []) = Series [] \]

### 2.3 Expanding two series to the same length

We need a variant of `zipWith` that assumes that shorter list is expanded with unit of the operation given as argument:

\[ zipWithExpanding :: (a -> a -> a) \]
\[ -> [a] -> [a] -> [a] \]
\[ zipWithExpanding f = go \]

where
\[ go [] ys = ys -- unit \ 'f' y = y \]
\[ go x:xs [] = x -- x \ 'f' unit = x \]
\[ go (x:xs) (y:ys) = (x \ 'f' y):go xs ys \]

Here we use extension by a given element `e`, which is 0 for normal series, or 1 for complement series.

Extend both series to the same length with placeholder zeros. Needed for safe use of complement-based operations.

\[ extendToSameLength e (Series a, Series b) = \]
\[ (Series resultA, Series resultB) \]

Proc. ACM Program. Lang., Vol. 1, No. POPL, Article 1. Publication date: January 2021.
where

\[(\text{resultA}, \text{resultB}) = \text{go a b}\]
\[\text{go \ [\] \ [\] } = (\ [\], \ [\])\]
\[\text{go (b:bs) (c:cs)} = (\ b:\text{bs}', \ c:\text{cs}')\]
\[\text{where}\]
\[\sim(\text{bs}', \text{cs}') = \text{go bs cs}\]
\[\text{go (b:bs)} \ [\] = (\ b:\text{bs}, \ e : \text{cs}')\]
\[\text{where}\]
\[\sim(\text{bs}', \text{cs}') = \text{go bs \ [\]}\]
\[\text{go \ [\]} (c:cs) = (\ e : \text{bs}', \ c : \text{cs})\]
\[\text{where}\]
\[\sim(\text{bs}', \_ ) = \text{go \ [\]} \text{cs}\]

In a rare case (CDFs) we might also prolong by the length of the last entry:
We will sometimes want to extend both series to the same length with placeholder of last element.

\[\text{extendToSameLength} '\ (\text{Series a}, \text{Series b}) =\]
\[\ (\text{Series resultA}, \text{Series resultB})\]
\[\text{where}\]
\[\ (\text{resultA}, \text{resultB}) = \text{go a b}\]
\[\text{go \ [\] \ [\] } = (\ [\], \ [\])\]
\[\text{go (b:bs)} \ [\] = (\ b:\text{bs}', \ e : \text{cs}')\]
\[\text{where}\]
\[\sim(\text{bs}', \text{cs}') = \text{go bs \ [\]}\]
\[\text{go \ [\]} (c:cs) = (\ e : \text{bs}', \ c : \text{cs})\]
\[\text{where}\]
\[\sim(\text{bs}', \_ ) = \text{go \ [\]} \text{cs}\]

2.4 Series modulus

We can present an instance of number class for Series:

\[\text{instance Num a => Num (Series a) where}\]
\[\text{Series a + Series b} = \text{Series}\]
\[\text{zipWithExpanding (+) a b}\]
\[\text{(*)} \quad = \text{convolve}\]
\[\text{abs} \quad = \text{fmap abs}\]
\[\text{signum} \quad = \text{fmap signum}\]
\[\text{fromInteger} = \text{seriesFromInteger}\]
\[\text{negate} = \text{fmap negate}\]

Note that we do not know yet how to define \text{fromInteger} function. Certainly we would like to define \text{null} and \text{unit} (neutral element) of convolution, but it is not clear what to do about the others:

\[\text{seriesFromInteger 0} = \text{nullE}\]
\[\text{seriesFromInteger 1} = \text{unitE}\]
\[\text{seriesFromInteger other} =\]
\[\text{error} \$ \text{"Do not use fromInteger\_\_\_\_"}\]
\[\quad <> \text{show other} <> \text{"\_\_\_\_\_\_Series!\"}\]
Given a unit and null elements, we can give unit and null element of a Series.\(^4\)

\[\text{instance } \text{Unit } a \Rightarrow \text{Unit (Series } a) \text{ where} \]
\[\text{unitE } = [\text{unitE}]\]

\[\text{instance } \text{Null } a \Rightarrow \text{Null (Series } a) \text{ where} \]
\[\text{nullE } = [\text{nullE}]\]

We may be using Series of floating point values that are inherently approximate.
In this case, we should not ever use equality, but rather a similarity metric that we can generate from the similar metric on values:

\[\text{instance } \text{Real } a \Rightarrow \text{Metric (Series } a) \text{ where} \]
\[a \text{'distance'} b = \sqrt{\text{realToFrac} \sum \text{fmap square} a-b} \]

\[\text{similarityThreshold } = 0.001\]

\[\text{square } x = x \times x\]

Note that generous similarity threshold of 0.001 is due to limited number of simulations we can quickly run when checking distributions in the unit tests (10k by default).

For a Series of objects having complement, there is also well established definition:

\[\text{instance } \text{Complement } a \Rightarrow \text{Complement (Series } a) \text{ where} \]
\[\text{complement } = \text{fmap complement}\]

**Square matrices of declared size**

This is a simple description of square matrices with fixed size.\(^5\) First we need natural indices that are no larger than \(n\):

\[\text{newtype } \text{UpTo } (n :: \text{Nat}) = \text{UpTo } \{ \_\text{unUpTo} :: \text{Natural} \} \]
\[\text{deriving } (\text{Eq, Ord, Num, Typeable})\]

The only purpose of safe indices is to enumerate them:

\[\text{allUpTo } :: \text{KnownNat } n \Rightarrow [\text{UpTo } n]\]

Armed with safe indices, we can define square matrices:

\[\text{newtype } \text{SMatrix } (n :: \text{Nat}) a = \]
\[\text{SMatrix } \{ \text{unSMatrix } :: \text{EM.Mat} \text{rix } a \} \]
\[\text{deriving } (\text{Show, Eq, Functor, Applicative, Foldable, Traversable, Typeable, Generic})\]

\[\text{size } :: \text{KnownNat } n \Rightarrow \text{SMatrix } n a \Rightarrow \text{Int} \]
\[\text{size } (s :: \text{SMatrix } n a) = \text{intVal } (\text{Proxy } :: \text{Proxy } n)\]

We also need to identity and null matrices (for multiplication):

\(^4\)Note that we in this context we are mainly interested in null and unit of multiplication.
\(^5\)Note that we considered using matrix–static, but it does not have typesafe indexing.
instance (KnownNat n, Null a) => Null (SMatrix n a) where
nullE = sMatrix Proxy (_ => nullE)

instance (KnownNat n, Null a, Unit a) => Unit (SMatrix n a) where
  unitE = sMatrix Proxy elt
  where
    elt (i, j) | i == j = unitE
    elt (i, j) = nullE

Definition of parametrized matrix multiplication is standard, so we can test it over other objects with defined multiplication and addition-like operators.

sMatMult :: KnownNat n => (a -> a -> a) -- ^ addition
           -> (a -> a -> a) -- ^ multiplication
           -> SMatrix n a
           -> SMatrix n a
           -> SMatrix n a
sMatMult add mul a1 (a2 :: SMatrix n a) =
  sMatrix (Proxy :: Proxy n) gen
  where
    gen :: KnownNat n => (UpTo n, UpTo n) -> a
    gen (i, j) = foldr1 add
                  [ (a1 ! (i, k)) `mul` (a2 ! (k, j))
                    | k <- allUpTo 'i' ]

Note that to measure convergence of the process, we need a notion of distance between two matrices.

Matrix addition for testing:

( | + | ) :: (Num a , KnownNat n ) => SMatrix n a
         -> SMatrix n a
         -> SMatrix n a
         a |+| b = (+) <$> a <*> b

One might also want to iterate over rows or columns in the matrix:

rows, columns :: KnownNat n => SMatrix n a -> [[a]]
rows sm = [[sm ! (i, j) | j <- allUpTo ] | i <- allUpTo]
columns sm = [[sm ! (i, j) | i <- allUpTo ] | j <- allUpTo]
3 LATENCY DISTRIBUTIONS

3.1 Introducing improper CDF

To define our key object, let’s imagine a single network connection. In this document, we ignore capacity-related issues. So $\Delta Q(t)$ is improper cumulative distribution function of event arriving at some point of time:

For the sake of practicality, we also mark a deadline as the last possible moment when we still care about messages. (Afterwards, we drop them, just like TCP timeout works.)

For example, when only 0.99 of messages arrive at all within desired time $t$, and we silently drop those that arrive later.

For each distribution, we will define deadline formally as $d(t) = \maxarg_t(\Delta Q(t))$ or such $t$ for which our improper CDF reaches maximum value. We also define ultimate arrival probability formally as $a_u(\Delta Q) = \max(\Delta Q)$. Our improper CDFs are assumed to be always defined within finite subrange of delays, starting at 0. Ultimate arrival probability allows us to compare attenuation between two links.

In the following we define domain of arrival probabilities as $\mathcal{A} \in [0, 1.0]$, which is probability.

We also define domain of time or delays as $\mathcal{T}$. We also call a domain of $\Delta Q$ functions as $Q = (\mathcal{T} \rightarrow \mathcal{A})$. 
Below is Haskell specification of this datatype:

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

The representation above holds PDF (probability density function). Its cumulative distribution function can be trivially computed with running sum:

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

Since it is common use complement of CDF, we can have accessor for this one too:

```haskell
complementCDF :: Probability a => LatencyDistribution a -> Series a
complementCDF = complement . cumsum . pdf
```

### 3.2 Canonical form

Sometimes we need to convert possibly improper `LatencyDistribution` into its canonical representation. Here we define `canonicalize` `LatencyDistribution` when (i) it is a valid improper probability distribution so sum does not go over 1.0; (ii) it does not contain trailing zeros after the first element (which are redundant). This definition assumes we have a finite series, and assures that any distribution has a unique representation.

```haskell
canonicalizeLD :: Probability a => LatencyDistribution a
    LatencyDistribution a
    LatencyDistribution a
   -Series
   - assureAtLeastOneElement
   - dropTrailingZeros
   - cutWhenSumOverOne 0.0
   - unSeries
   - pdf
```

```haskell
where
    cutWhenSumOverOne aSum [] = []
    cutWhenSumOverOne aSum (x:xs)
        | aSum+x > 1.0 = [1.0 - aSum]
    cutWhenSumOverOne aSum (x:xs) =
        x : cutWhenSumOverOne (aSum+x) xs
    assureAtLeastOneElement [] = [0.0]
    assureAtLeastOneElement other = other
    dropTrailingZeros = reverse
        . dropWhile (== 0.0) . reverse
```

### 3.3 Construction from PDF and CDF

We use `canonicalizeLD` to make sure that every distribution is kept in canonical form (explained below), we might also want to make constructors that create `LatencyDistribution` from a series that represents PDF or CDF:
fromPDF :: Probability a
    => Series a
    => LatencyDistribution a
fromPDF = canonicalizeLD . LatencyDistribution

To create LatencyDistribution from CDF we need \( \text{diffEnc} \) (\text{differential encoding} or \text{backward finite difference operator} from \text{Series} module):

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

Similar we can create \text{LatencyDistribution} from complement of CDF:

fromComplementOfCDF :: Probability a
    => Series a
    => LatencyDistribution a
fromComplementOfCDF = fromCDF . complement

3.4 Intuitive properties of latency distributions

1. We can define few \textit{linear} operators on \( \Delta Q \) (for exact definition, see next section):
   A. Stretching in time – ignored in here.
   B. Delaying by \( t \) – composition with wait:
      \[
      \text{wait}(t) = f(t) = \begin{cases} 
      0 & \text{for } t < t_d \\
      1.0 & \text{for } t = t_d 
      \end{cases}
      \]
   C. Scaling arrival probability – in other words, attenuation.

2. We distinguish special distribution that represents \textit{null delay} or \textit{neutral element} of sequential composition (\( \# \) or afterLD), where we pass every message with no delay:
   \[
   \text{preserved}(1) = \text{wait}(0) = 1_Q 
   \]

3. We can say that one \( \Delta Q \) no worse than the other, when it is improper CDF values never less than the other after making it fit a common domain:
   \( \Delta Q_1 \geq q \Delta Q_2 \equiv \forall t. X(\Delta Q_1)(t) \geq X(\Delta Q_2)(t) \)

   Here assuming \( X(\Delta Q) \) defined:
   \[
   X(\Delta Q)(t) = \begin{cases} 
   \Delta Q(t) & \text{for } t \leq d(\Delta Q) \\
   \Delta Q(d(\Delta Q)) & \text{otherwise}
   \end{cases}
   \]

3.5 Basic operations on \( \Delta Q \)

To model connections of multiple nodes, and network protocols we need two basic operations: sequential and parallel composition.

Interestingly both of these operations form an associative-commutative monoids (with unit that changes nothing) with null element (zero that nullifies the other), however their null and neutral elements swap places.

1. Sequential composition \( \# \) or afterLD\(^6\) given \( \Delta Q_1(t) \) and \( \Delta Q_2(t) \) of two different connections, we should be able to compute the latency function for routing the message through pair of

\(^6\)Sometimes named \( ; \).
these connections in sequence:

\[ \Delta Q_1(t) \triangleright \Delta Q_2(t) \]

- **associative:**
  \[ \Delta Q_1(t) \triangleright [\Delta Q_2(t) \triangleright \Delta Q_3(t)] = [\Delta Q_1(t) \triangleright \Delta Q_2(t)] \triangleright \Delta Q_3(t) \]

- **commutative**
  \[ \Delta Q_1(t) \triangleright \Delta Q_2(t) = \Delta Q_2(t) \triangleright \Delta Q_1(t) \]

- **neutral element** is \( 1_Q \) or noDelay, so:
  \[ \Delta Q(t) \triangleright 1_Q = 1_Q \triangleright \Delta Q(t) = \Delta Q(t) \]

- **null element** is \( 0_Q \) or allLost, so:
  \[ \Delta Q(t) \triangleright 0_Q = 0_Q \triangleright \Delta Q(t) = 0_Q \]

---

Here is the Haskell code for naive definition of these two operations: We can also introduce alternative of two completion distributions. It corresponds to a an event that is earliest possible conclusion of one of two alternative events.

That can be easily expressed with improper cumulative distribution functions:

\[ P_{\min(a,b)}(x \leq t) = 1 - (1 - P_a(x \leq t)) \ast (1 - P_b(x \leq t)) \]

That is, event \( \min(a, b) \) occurred when \( t < a \) or \( t < b \), when:

- it **did not occur** (top complement: 1 - ...), that:
  - either \( a \) did **not** occur \( 1 - P_a(x \leq t) \),
  - and \( b \) did **not** occur \( 1 - P_b(x \leq t) \):
\[ firstToF\text{\textunderscore}FinishLD \iff Probability a \Rightarrow \text{LatencyDistribution a} \Rightarrow \text{LatencyDistribution a} \Rightarrow \text{LatencyDistribution a} \]

rd1 `firstToF\text{\textunderscore}FinishLD` rd2 =
fromComplementOfCDF $  
  complementCDF rd1 `*`. 
  complementCDF rd2 `*`. 

\[ \text{where} \]

\[(rd1 `*`, rd2 `*`) = \text{extendToSameLengthLD} (rd1 , rd2) \]

Notes:

1. Since we model this with finite discrete series, we first need to extend them to the same length.
2. Using the fact that \text{cumsum} is discrete correspondent of integration, and \text{diffEnc} is its direct inverse (\text{backward finite difference}), we can try to differentiate this to get PDF directly:

\[ P_a(x \leq t) = \Sigma_{x=0}^t P_a(x) \]

\[ \nabla (\Sigma_{x=0}^t P_a(x)dx) = P_a(t) \]

In continuous domain, one can also differentiate both sides of the equation above to check if we can save computations by computing PDF directly.

Unfortunately, that means that instead of 2x cumulative sum operations, 1x elementwise multiplication, and 1x differential encoding operation, we still need to perform the same 2x cumulative sums, and 3x pointwise additions and 3x pointwise multiplications, and two complements.

So we get an equation that is less obviously correct, and more computationally expensive.

Code would look like:

rd1 `firstToF\text{\textunderscore}FinishLD` rd2 = canonici\text{\textunderscore}zizeLD $  
LatencyDistribution {  
pdf = rd1 `*`. complementCDF rd2 `*`.  
  + rd2 `*`. complementCDF rd1 `*`.  
  + rd1 `*`. rd2 `*`. }  

\[ \text{where} \]

\[(rd1 `*`, rd2 `*`) = \text{extendToSameLengthLD} (rd1 , rd2) \]
complement :: Series Probability  
  -> Series Probability  
complement = fmap (1.0 -)

\text{Note that complement above will be correct only if both lists are of the same length.}

In order to use this approach in here, we need to prove that \text{cumsum} and \text{diffEnc} correspond to integration, and differentiation operators for discrete time domain.

Now let’s define neutral elements of both operations above:

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

allLostLD , noDelayLD :: Probability a  
  => LatencyDistribution a
Curious properties of latency distributions

\[ \text{allLostLD} = \text{preserved } 0.0 \]
\[ \text{noDelayLD} = \text{preserved } 1.0 \]

Here:
- `allLost` indicates that no message arrives ever through this connection
- `noDelay` indicates that all messages always arrive without delay

3. Conjunction of two different actions simultaneously completed in parallel, and waits until they both are:

\[ P_{\max(a,b)}(x \leq t) = P_a(x \leq t) \ast P_b(x \leq t) \]

`lastToFinishLD` :: `Probability a` 
=> `LatencyDistribution a` 
-> `LatencyDistribution a`

\[
\text{rd1 \ `lastToFinishLD` \ rd2} = \text{fromCDF}$
\text{cdf rd1'} \ast \text{cdf rd2'}
\]

where
\[
(\text{rd1'}, \text{rd2'}) = \text{extendToSameLengthLD (rd1, rd2)}
\]

(Attempt to differentiate these by parts also leads to more complex equation: \(\text{rd1} \ast \text{cumsum rd2} + \text{rd2} \ast \text{cumsum rd1}\)).

Now we can make an abstract interpretation of protocol code to derive corresponding improper CDF of message arrival.

It is also:
- commutative
- associative
- with neutral element of noDelay

4. Failover \( A < t > B \) when action is attempted for a fixed period of time \( t \), and if it does not complete in this time, the other action is attempted:

\[
\text{failover deadline rdTry rdCatch} = \text{fromPDF}$
\text{initial <> fmap (remainder \ast)} (\text{pdf rdCatch})
\]

where
\[
\text{initial} = \text{cut deadline} \$ \text{pdf rdTry}
\text{remainder} = 1 - \text{sum initial}
\]

Algebraic properties of this operator are clear:
- Certain failure converts deadline into delay:
  \[ \text{fail } t > A = \text{wait}_t; A \]
- Failover to certain failure only cuts the latency curve:
  \[ A < t > \text{fail} = \text{cut}_t A \]
- Certain success ignores deadline:
  \[ 1_Q < t > A = 1_Q \text{ when } t > 0 \]
- Failover with no time left means discarding initial operation:
  \[ A < 0 > B = B \]
- When deadline is not shorter than maximum latency of left argument, it is similar to alternative, with extra delay before second argument:
  \[ A < t > B = A \lor \text{wait}(t)B \text{ when } t > d(A) \]
5. Retransmission without rejection of previous messages: $A < t >> B$, when we have a little different algebraic properties with *uncut* left argument:

$$
A < t > B = A \lor \text{wait}_t; B
$$
$$
A < 0 > B = A \lor B
$$
$$
A < t > \text{fail} = A
$$
$$
\text{fail} < t > A = A
$$

6. Some approaches [3] propose using operator $A \leftrightarrow_p B$ for probabilistic choice between scenarios $A$ with probability $p$, and $B$ with probability $1 - p$. In this work we assume the only way to get non-determinism is due to latency, for example if protocols use unique agency property like those used in Cardano network layer [??].

### 3.6 Abstracting over implementation

We can encapsulate basic operations with `TimeToCompletion` class, describing interface that will be used both for latency distributions and their approximations:

```haskell
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`

instance Probability a => TimeToCompletion (LatencyDistribution a) where
    firstToFinish = firstToFinishLD
    lastToFinish = lastToFinishLD
    after = afterLD
    delay = delayLD
    allLost = allLostLD
    noDelay = noDelayLD
```

#### 3.6.1 General treatment of completion distribution over time

Whether might aim for minimum delay distribution of message over a given connection $\Delta Q(t)$, minimum time of propagation of the message over entire network ($\Delta R(t)$, reachability), we still have a distribution of completion distribution over time with standard operations.

We will need a standard library for treating these to speed up our computations.

We can also define a mathematical ring of (probability, delay) pairs. Note that `LatencyDistribution` is a modulus over ring $\mathbb{R}$ with `after` as multiplication, and whicheverIsFaster as addition. Then `noDelay` is neutral element of multiplication (unit or one), and `allLost` is neutral element of addition.\(^7\) Note that both of these binary operators give also rise to two almost-scalar multiplication operators:

\(^7\)This field definition will be used for multiplication of connection matrices.
Curious properties of latency distributions

scaleProbability :: Probability a
=> a
-> LatencyDistribution a
-> LatencyDistribution a
scaleProbability a = after $ preserved a

scaleDelay :: Probability a
=> Delay
-> LatencyDistribution a
-> LatencyDistribution a
scaleDelay t = after $ delayLD t

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

To compare distributions represented by series of approximate values we need approximate
equality:

instance (Metric a, Num a, Real a) => Metric (LatencyDistribution a) where
LatencyDistribution l
`distance`
LatencyDistribution m =
realToFrac $ sum $ fmap (^2) l - m
similarityThreshold = 1/1000

Choosing 0.001 as similarity threshold (should depend on number of samples)

instance Unit a
=> Unit (LatencyDistribution a) where
unitE = LatencyDistribution (Series [unitE])

instance Null a
=> Null (LatencyDistribution a) where
nullE = LatencyDistribution (Series [nullE])

3.7 Bounds on distributions

Note that we can define bounds on LatencyDistribution that behave like functors over basic operations
from TimeToCompletion class.

• Upper bound on distribution is the Latest possible time:

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

8Here liftBinOp is for lifting an operator to a newtype.
newtype SometimeOrNever =  
  SometimeOrNever  
  { unSometimeOrNever :: Maybe Delay }  
  deriving (Eq)  

instance Show SometimeOrNever where  
  showsPrec _ Never = (*"Never"++)  
  showsPrec prec (Sometime t) =  
    showParen (prec>app_prec)$  
    (*"Sometime"++) . showsPrec (app_prec+1) t  
  where  
    app_prec = 10  

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

instance Ord SometimeOrNever where  
  Never `compare` Never = EQ  
  Never `compare` Sometime _ = GT  
  Sometime _ `compare` Never = LT  
  Sometime t `compare` Sometime u = t `compare` u  

latest :: Probability a  
  => LatencyDistribution a  
  -> Latest  
latest ((<1.0) . sum . pdf -> True) = Latest Never  
latest (Last . unSeries . pdf -> 0.0) =  
  error (*"Canonical_LatencyDistribution"*  
    <> "should always end with non-zero value")  
latest x =  
  Latest . Sometime . Delay . (-1+)  
  . length . unSeries . pdf $ x  

onLatest = liftBinOp unLatest Latest  

instance TimeToCompletion Latest where  
  firstToFinish = onLatest min  
  lastToFinish = onLatest max  
  after =  
    liftBinOp (unSometimeOrNever . unLatest)  
    (Latest . SometimeOrNever)  
    (liftM2 (+))  
  delay = Latest . Sometime  
  allLost = Latest . Never  

  • Lower bound on distribution is the Earliest possible time:

newtype Earliest =  
  Earliest  
  { unEarliest :: SometimeOrNever }  
  deriving (Eq, Ord, Show)  
earliest :: Probability a  
  => LatencyDistribution a
Curious properties of latency distributions

\[ \rightarrow \text{Earliest} \]
\[ \text{earliest} [0.0] = \text{Earliest Never} \]
\[ \text{earliest} [_] = \text{Earliest $\text{Sometime 0}$} \]
\[ \text{earliest (last . unSeries . pdf -> 0.0) = error ("Canonical \_ LatencyDistribution\_" \» "should\_always\_end\_with\_non\_zero\_value\")} \]
\[ \text{earliest other = Earliest . Sometime} \]
\[ \quad . \text{Delay . max 0 . length} \]
\[ \quad . \text{takeWhile (0==) . unSeries} \]
\[ \quad . \text{pdf $\text{other}} \]
\[ \text{onEarliest} = \text{liftBinOp unEarliest Earliest} \]

\text{instance TimeToCompletion Earliest where}
\[ \text{firstToFinish} = \text{onEarliest min} \]
\[ \text{lastToFinish} = \text{onEarliest max} \]
\[ \text{after} = \]
\[ \text{liftBinOp (unSometimeOrNever . unEarliest)} \]
\[ \quad (\text{Earliest . SometimeOrNever)} \]
\[ \quad (\text{liftM2 (+))} \]
\[ \text{delay} = \text{Earliest . Sometime} \]
\[ \text{allLost} = \text{Earliest Never} \]

These estimates have the property that we can easily compute the same operations on estimates, without really computing the full \text{LatencyDistribution}. 

4 FAILURE MODELS

4.1 Transient failures

Up to now, we used probability distributions and their discretizations in order to model \textit{transient failures}:

- congestion leading to dropped packages
- transmission errors leading to dropped messages

4.2 Persistent failures

In order to deal with \textit{persistent failures}, we would need a notion of failure that is not independent: when we see persistent failure, it is likely that we will see it again during retransmissions.

Since persistent failures are likely to continue, we can easily model them as a change of network topology: some connection latencies will be reset to zero, when persistent failure between two nodes occurs.

We can easily accommodate it as a new layer over our model of the network: we will assign probabilities to each link about how often the persistent failure occurs, and compute with these in mind.

5 REPRESENTING NETWORKS

Adjacency matrix is classic representation of network graph, where \(i\)-th row corresponds to outbound edges of node \(i\), and \(j\)-th column corresponds to inbound edges of node \(j\). So \(A_{i,j}\) is 1 when edge is connected, and 0 if edge is not connected.

It is common to store only upper triangular part of the matrix, since:

- it is symmetric for undirected graphs
• it should have 1 on the diagonal, since every node is connected to itself.

We use this trick to avoid double counting routes with different directions.

So network connectivity matrix is:

• having units on the diagonal
• having connectivity information between nodes $i$, and $j$, for $j > i$ in element $a_{i,j}$.

Generalizing this to $\Delta Q$-matrices we might be interesting in:

• whether $A^n$ correctly mimicks shortest path between nodes (Earliest)
• whether $A^n$ correctly keeps paths shorter than $n$
• for a strongly connected graph there should exist $n \leq \text{dim}(A)$, such that $A^n$ is having non-null elements on an upper triangular section.

More rigorous formulation is:

$$R_0(A) = 1$$
$$R_n(A) = R_{n-1}(A) \ast A$$

Where:

• 1 or $I_d$ denotes a unit adjacency matrix, that is matrix where every node is connected with itself but none else. And these connections have no delay at all.
• $A$ is connection matrix as defined above, and distribution for a transmission from a single packet from $i$-th to $j$-th node. For pre-established TCP this matrix should be symmetric.

Our key metric would be diffusion or reachability time of the network $\Delta R(t)$, which is conditioned by quality of connection curves $\Delta Q(t)$ and the structure network graph.

Later in this section, we discuss on how $\Delta R(t)$ encompasses other plausible performance conditions on the network.

This gives us interesting example of using matrix method over a modulus, and not a group.

5.0.1 Reachability of network broadcast or $\Delta R(t)$.
Reachability curve $\Delta R(t)$ or diffusion time is plotted as distribution of event where all nodes are reached by broadcast from committee node, against time. We want to sum the curve for all possible core nodes, by picking a random core node.

Area under the curve would tell us the overall quality of the network. When curve reaches 100% rate, then we have a strongly connected network, which should be eventually always the case after necessary reconfigurations.

Note that when running experiments on multiple networks, we will need to indicate when we show average statistics for multiple networks, and when we show a statistic for a single network.

5.0.2 Description of network connectivity graph in terms of $\Delta Q$.
Traditional way of describing graphs is by adjacency matrix, where 0 means there is no edge, and 1 means that there is active edge.

We may generalize it to unreliable network connections described above, by using $\Delta Q$ instead of binary.

So for each value diagonal, the network connection matrix $A$ will be $	ext{noDelay}$, and $A_{ij}$ will represent the connection quality for messages sent from node $i$ to node $j$.

That allows us to generalize typical graph algorithms executed to algorithms executed on network matrices:

1. If series $R_n(A)$ converges to matrix of non-zero ($\text{non-allLost}$) values in all cells in a finite number of steps, we consider graph to be strongly connected [9]. Matrix multiplication follows uses $(; , \lor)$ instead of $(\ast, +)$. So sequential composition afterLD in place of multiplication, and alternative selection firstToFinish in place of addition.
When it exists, limit of the series \( R_n(A) = A^n \) is called \( A^* \).

In case of non-zero delays, outside diagonal, we may also consider convergence for delays up to \( t \). NOTE: We need to add estimate of convergence for cutoff time \( t \) and number of iterations \( n \), provided that least delay is in some relation to \( n \times t \).

Also note that this series converges to \( \Delta Q \) on a single shortest path between each two nodes. That means that we may call this matrix \( R_{\text{min}}(t) \), or optimal diffusion matrix.

```haskell
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
```

Of course, this requires a reasonable approximate metric -- and definition of convergence:

```haskell
converges :: Metric r
  => Int -- ^ max number of steps
  -> (r -> r)
  -> r
  -> r
converges 0 step r =
  error "Solution did not converge"
converges aLimit step r | r ~~ r' =
  then r
  else converges (pred aLimit) step r'
  where
    r' = step r
```

We will use this to define path with shortest \( \Delta Q \). It corresponds to the situation where all nodes broadcast value from any starting point \( i \) for the duration of \( n \) retransmissions.\(^9\)

2. Considering two nodes we may consider delay introduced by retransmissions in naive miniprotocol:
  - we have two nodes sender and receiver
  - sender sends message once per period equal maximum network latency deadline
  - the message is resent if receiver fails to send back confirmation of receipt …

Assuming latency of the connection \( l \), and timeout \( t > d(l) \), we get simple solution:

\[
\mu X.l;l;X
\]

3. We need to consider further examples of how our metrics react to issues detected by typical graph algorithms.

We define a matrix multiplication that uses \_firstToFinish\_ in place of addition and \_after\_ in place of multiplication.

```haskell
(|*|) :: (Probability a, KnownNat n)
=> SMatrix n (LatencyDistribution a)
-> SMatrix n (LatencyDistribution a)
```

\(^9\)That we do not reduce loss over remainder yet?
\[ \text{SMatrix} n \ (\text{LatencyDistribution} \ a) \]
\[ (+ \cdot) = \text{sMatMult firstToFinish after} \]

Note that to measure convergence of the process, we need a notion of distance between two matrices.

Here, we use Frobenius distance between matrices, parametrized by the notion of distance between any two matrix elements.

\[
\text{instance} \ (\text{Metric a} , \text{KnownNat n}) \Rightarrow \text{Metric (SMatrix n a) where}
\]
\[
a \ `\text{distance}` \ b = \sqrt{\sum \text{square} ((a ! (i, k)) \ `\text{distance}` (b ! (i, k)))}
\]
\[
\text{where}
\]
\[
\text{square x} = x \times x
\]
\[
\text{similarityThreshold} = \text{similarityThreshold} @a
\]

6 HISTOGRAMMING

To provide histograms of average number of nodes reached by the broadcast, we need to define additional operations:

- sum of mutually exclusive events
- K-out-of-N synchronization

6.1 Sum of mutually exclusive events

First we need to define a precise sum of two events that are mutually exclusive \(\oplus\). That is different from firstToFinish which assumes that they are mutually independent.

\[
\text{infixl 5 `exSum` -- like addition}
\]
\[
\text{class} \ \text{ExclusiveSum a where}
\]
\[
\text{exAdd :: a -> a -> a}
\]
\[
exSum :: \text{ExclusiveSum a} \Rightarrow [a] \rightarrow a
\]
\[
exSum = \text{foldr1 exAdd}
\]
\[
\text{instance} \ \text{ExclusiveSum ApproximateProbability where}
\]
\[
exAdd = (+)
\]
\[
\text{instance} \ \text{ExclusiveSum IdealizedProbability where}
\]
\[
exAdd = (+)
\]

Given a definition of exclusive sum for single events, and existence neutral element of addition, we can easily expand the definition to the latency distributions:

\[
\text{instance} \ \text{ExclusiveSum a}
\]
\[
\Rightarrow \text{ExclusiveSum (Series a) where}
\]
\[
\text{Series a `exAdd` Series b} = \text{Series}
\]
\[
\text{zipWithExpanding exAdd a b}
\]
\[
\text{instance} \ \text{ExclusiveSum (LatencyDistribution a) where}
\]
\[
\text{LatencyDistribution a `exAdd}`
\]

Proc. ACM Program. Lang., Vol. 1, No. POPL, Article 1. Publication date: January 2021.
LatencyDistribution $ a \prec \text{`exAdd` $ b$

6.2 \textit{K-out-of-N synchronization of series of events $a_k$.}

For histogramming a fraction of events that have been delivered within given time, we use generalization of recursive formula \( \left( \begin{array}{c} n \\ k \end{array} \right) \).

\[
F_{\left( \begin{array}{c} a_n \\ k \end{array} \right)}(t) = a_n \wedge F_{\left( \begin{array}{c} a_{n-1} \\ k \end{array} \right)}(t) \oplus (\overline{a_n} \wedge F_{\left( \begin{array}{c} a_{n-1} \\ k \end{array} \right)}(t))
\]

Here $a_{n-1}$ is a finite series without its last term (ending at index $n - 1$).

It is more convenient to treat $F_{\left( \begin{array}{c} a_n \\ k \end{array} \right)}(t)$ as a Series with indices ranging over $k$: $F_{\left( \begin{array}{c} a_n \\ k \end{array} \right)}(t)$. Then we see the following equation:

\[
F_{\left( \begin{array}{c} a_n \\ k \end{array} \right)} = [\overline{a_n}, a_n] \otimes F_{\left( \begin{array}{c} a_{n-1} \\ k \end{array} \right)}
\]

Where:

- \( \otimes \) 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.

We implement it as a series $k_{\text{of}}_n$ with parameter given as series $a_n$, and indices ranging over $k$:

\[
kOfN :: \text{TimeToCompletion} \ a \rightarrow \text{ExclusiveSum} \ a \rightarrow \text{Complement} \ a \rightarrow \text{Series} \ a
\]

\[
kOfN \ (\text{Series} \ []) = \text{error} \ "kOfN \ of \ empty \ series"
\]

\[
kOfN \ (\text{Series} \ [\ x \ ]) = \text{[complement} \ x, \ x]\]

\[
kOfN \ (\text{Series} \ ([\ x:xs\ ])) = \text{[x] `convolution` Series xs}
\]

\[
\text{where}
\]

\[
\text{convolution} = \text{convolve}_\text{exAdd} \ \text{lastToFinish} \ \ `\text{on}` \ \text{kOfN}
\]

6.2.1 \textit{Fraction of reached nodes.} Now, for a connection matrix $A$, each row corresponds to a vector of latency distribution for individual nodes. Naturally source node is indicated as a \textit{unit} on a diagonal. Now we can use $kOfN$ to transform the series corresponding to a single row vector into a distribution of latencies for reaching $k$-\textit{out-of-$n$} nodes. Note that this new series will have \textit{indices} corresponding to \textit{number of nodes reached} instead of node indices:

\[
nodesReached :: \text{Probability} \ a \rightarrow \text{ExclusiveSum} \ a \rightarrow \text{Series} \ (\text{LatencyDistribution} \ a)
\]

\[
nodesReached = kOfN
\]

For this we need to define complement for \textit{LatencyDistribution}:

\[
\text{instance} \ \text{Complement} \ a \rightarrow \text{Complement} \ (\text{LatencyDistribution} \ a) \ 
\text{where}
\]

\[
\text{complement} \ (\text{LatencyDistribution} \ s) = \text{LatencyDistribution} \ (\text{complement} \ <$\ > \ s)
\]
6.3 Averaging broadcast from different nodes

Given that we have a connection matrix $A$ of broadcast iterated $n$ times, we might want histogram of distribution of a fraction of nodes reached for a random selection of source node.

We can perform this averaging with exclusive sum operator, pointwise division of elements by the number of distributions summed:

```haskell
averageKOutOfN :: (KnownNat n, ExclusiveSum a, Probability a, Show a) => SMatrix n (LatencyDistribution a) -> Series (LatencyDistribution a)
averageKOutOfN m = average (nodesReached . Series <$> rows m)
```

```haskell
average :: (ExclusiveSum a, Probability a, Show a) => [Series (LatencyDistribution a)] -> Series (LatencyDistribution a)
average aList = scaleLD <$> exSum aList
  where
    scaleLD :: Probability a -> LatencyDistribution a
            -> LatencyDistribution a
    scaleLD = scaleProbability
              (1/fromIntegral (length aList))
```

7 SUMMARY

We have shown how few lines of Haskell code can be used to accurately model network latency and get n-hop approximations of packet propagation.

It turns out they can also be used to model task completion distributions for a all-around estimation of software completion time.

7.1 What for capacity-limited networks?

It turns out that most of the real network traffic is latency limited and focused on mice connections: that is connections that never have bandwidth-latency product that would be greater than 12kB’s.

That means that our approximation is actually useful for most of the flows in real networks, even though the real connections have limited capacity!

7.2 Relation of latency to other plausible metrics of network performance

One can imagine other key properties that network must satisfy:

- That absent permanent failures, network will reach full connectivity. That corresponds to the situation where given $\Delta Q(t)$ iCDF ultimately reaches 100% delivery probability for some delay, $\Delta R(t)$ will also always reach 100%. Moreover $\Delta R(t)$ metric allows to put deadline for reaching full connectivity in a convenient way.
- That given a fixed limit on rate of nodes joining and leaving the network, we also will have deadline on when $\Delta R(t)$ reaches a fixed delivery rate $R_{SLA}$ within deadline $t_{SLA}$.
• That given conditions on minimum average quality of connections $\Delta Q(t)$, and fixed rate of adversary nodes $r_{adv}$ we can still guarantee networks reaches reachability $R_{SLA}$.
• That there are conditions for which $\Delta R(t)$ always reaches almost optimal reachability defined by given ratio $\alpha \in (0.9, 1.0)$, such that $\Delta R(\alpha \times t) \geq max(\Delta R_{optimal}(t)/\alpha)$. In other words: there is a deadline $\alpha$-times longer than time to reach optimal reachability in an optimal network, we reach connectivity of no less than $\alpha$-times connectivity of the optimal network.

7.3 Interesting properties
We note that moduli representing latency distributions have properties that allow for efficient estimation by bounds that conform to the same laws.

Square matrices of these distributions or their estimations can be used to estimate network propagation and reachability properties.
That makes for an interesting class of algebras that can be used as a demonstration of moduli to undergraduates, and also allows to introduce them to latency-limited performance which is characteristic to most of modern internetworking.

7.4 Future work
We would like to apply these methods of latency estimation to modelling a most adverse scenarios: when hostile adversary aims to issue denial-of-service attack by delaying network packets[1].

BIBLIOGRAPHY
[1] Anderson, C. et al. 2005. Towards type inference for javascript. ECOOP 2005 - object-oriented programming (Berlin, Heidelberg, 2005), 428–452.
[2] Azzana, Y. et al. 2009. Adaptive algorithms for identifying large flows in ip traffic.
[3] Bradley, J.T. 1999. Towards reliable modelling with stochastic process algebras. Department of Computer Science, University of Bristol.
[4] Ellens, W. and Kooij, R.E. 2013. Graph measures and network robustness. CoRR abs/1311.5064, (2013).
[5] Ellens, W. et al. 2011. Effective graph resistance. Linear Algebra and its Applications. 435, 10 (2011), 2491–2506. DOI:https://doi.org/https://doi.org/10.1016/j.laa.2011.02.024.
[6] Jean-Yves Le Boudec; Thiran, Patrick 2001. Network Calculus: A Theory of Deterministic Queuing Systems for the Internet. Lecture Notes in Computer Science. 2050.
[7] McIlroy, M.D. 1999. Power series, power serious. Journal of Functional Programming. (1999), 323–335.
[8] Nielson, F. et al. 1999. Principles of program analysis.
[9] O’Connor, R. 2011. A very general method of computing shortest paths. Russell O’Connor’s Blog. http://r6.ca/.
[10] Wikipedia contributors 2019. Finite difference — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Finite_difference&oldid=903566420.
[11] Wikipedia contributors 2019. Indefinite sum — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Indefinite_sum&oldid=900662530.

GLOSSARY
• $t \in T$ - time since sending the message, or initiating a process
• $\Delta Q(t)$ - response rate of a single connection after time $t$ (chance that message was received until time $t$)
• $\Delta R(t)$ - completion rate of broadcast to entire network (rate of nodes expected to have seen the result until time $t$)
\( \epsilon \) - rate of packets that are either dropped or arrive after latest reasonable deadline we chose