Verified Causal Broadcast with Liquid Haskell

Patrick Redmond     Gan Shen     Niki Vazou     Lindsey Kuper

IFL 2022
Copenhagen, Denmark
31 August 2022

github.com/lsd-ucsc/cbcast-lh
Verified Causal Broadcast with Liquid Haskell

Patrick Redmond  Gan Shen  Niki Vazou  Lindsey Kuper

IFL 2022
Copenhagen, Denmark
31 August 2022

github.com/lsd-ucsc/cbcast-lh
Verified Causal Broadcast with Liquid Haskell

Patrick Redmond  Gan Shen  Niki Vazou  Lindsey Kuper

IFL 2022
Copenhagen, Denmark
31 August 2022

github.com/lsd-ucsc/cbcast-lh
Verified Causal Broadcast with Liquid Haskell

Patrick Redmond  Gan Shen  Niki Vazou  Lindsey Kuper

IFL 2022
Copenhagen, Denmark
31 August 2022

github.com/lsd-ucsc/cbcast-lh
Verified Causal Broadcast with Liquid Haskell

Patrick Redmond  Gan Shen  Niki Vazou  Lindsey Kuper

IFL 2022
Copenhagen, Denmark
31 August 2022

github.com/lsd-ucsc/cbcast-lh
Verified Causal Broadcast with Liquid Haskell

Patrick Redmond  Gan Shen  Niki Vazou  Lindsey Kuper

IFL 2022
Copenhagen, Denmark
31 August 2022

github.com/lsd-ucsc/cbcast-lh
Verified Causal Broadcast with Liquid Haskell

Patrick Redmond  Gan Shen  Niki Vazou  Lindsey Kuper

IFL 2022
Copenhagen, Denmark
31 August 2022

github.com/lsd-ucsc/cbcast-lh
Lost my...

Found it!
Lost my …

Found it!

Lost my …

Found it!
Lost my …

Found it!

happens-before
Lost my...       ...happens-before

happens-before

Found it!
Lost my…

“FIFO delivery"

Found it!

happens-before
Lost my ... 

FIFO delivery

Yay!

Found it!

happens-before
Lost my …

Found it!

Yay!

FIFO delivery

happens-before
Lost my ... Found it!

Yay!

FIFO delivery

happens-before
Lost my...  

Found it!  

FIFO delivery  

happens-before
Lost my ...

Yay!

FIFO delivery

Found it!

happens-before
Causal broadcast with vector clocks [Birman et al., 1991]
Causal broadcast with vector clocks [Birman et al., 1991]
Lost my …

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:
- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:
- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:
- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:
- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:

- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:

- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:

- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:
- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:

- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:
- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:
- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:
- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:
- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

**Causal broadcast with vector clocks** [Birman et al., 1991]
A message is deliverable if its VC is:

- 1 greater than recipient’s VC in sender’s position
- \(\leq\) recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:
- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:

- 1 greater than recipient’s VC in sender’s position
- \( \leq \) recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
type Nat = { v:Int | v >= 0 }
type Nat = { v:Int | v >= 0 }
type Nat = { v: Int | v >= 0 }

type VectorClock = [Nat]

Refinement types
type Nat = { v:Int | v >= 0 }

type VectorClock = [Nat]

vcMerge :: VectorClock -> VectorClock -> VectorClock

Refinement types
type Nat = { v:Int | v >= 0 }

type VectorClock = [Nat]

vcMerge :: VectorClock -> VectorClock -> VectorClock
vcMerge = zipWith max

e.g., vcMerge [1,0,0,0] [0,2,0,1] = [1,2,0,1]

Refinement types
type Nat = { v:Int | v >= 0 }

type VectorClock = [Nat]

vcMerge :: VectorClock -> VectorClock -> VectorClock
vcMerge = zipWith max

e.g., vcMerge [1,0,0,0] [0,2,0,1] = [1,2,0,1]

type VCsized N = { vc:VectorClock | len vc == N }

Refinement types
\[
\text{type } \text{Nat} = \{ \text{v:Int} \mid \text{v} \geq 0 \} \\
\text{type } \text{VectorClock} = [\text{Nat}]
\]

\[\text{vcMerge} :: \text{VectorClock} \rightarrow \text{VectorClock} \rightarrow \text{VectorClock}\]
\[\text{vcMerge} = \text{zipWith max}\]

\[\text{e.g., } \text{vcMerge} [1,0,0,0] [0,2,0,1] = [1,2,0,1] \]

\[
\text{type } \text{VCsized N} = \{ \text{vc:VectorClock} \mid \text{len vc} = N \} \\
\text{type } \text{VCsameLength V} = \text{VCsized} \{\text{len V}\}
\]

**Refinement types**
type Nat = { v: Int | v >= 0 }
type VectorClock = [Nat]

vcMerge :: VectorClock -> VectorClock -> VectorClock
vcMerge = zipWith max

e.g., vcMerge [1,0,0,0] [0,2,0,1] = [1,2,0,1]

type VC sized N = { vc: VectorClock | len vc == N }
type VCsameLength V = VC sized {len V}

vcMerge :: v: VectorClock -> VCsameLength {v} -> VCsameLength {v}

Refinement types
type Nat = { v:Int | v >= 0 }

type VectorClock = [Nat]

type VCsize N = { vc:VectorClock | len vc == N }

type VCsameLength V = VCsize {len V}

vcMerge :: v:VectorClock -> VCsameLength {v} -> VCsameLength {v}
vcMerge = zipWith max

Refinement types
type Nat = { v:Int | v >= 0 }

type VectorClock = [Nat]

type VCsized N = { vc:VectorClock | len vc == N }
type VCsameLength V = VCsized {len V}

vcMerge :: v:VectorClock -> VCsameLength {v} -> VCsameLength {v}
vcMerge = zipWith max

Refinement reflection
type Nat = { v:Int | \( v \geq 0 \) }

type VectorClock = [Nat]

type VC sized N = { vc:VectorClock | len vc == N }

type VC sameLength V = VC sized {len V}

vcMerge :: v:VectorClock -> VC sameLength {v} -> VC sameLength {v}

vcMerge = zipWith max


type Commutative a A = x:a -> y:a -> { _:Proof | A x y == A y x }

Refinement reflection
type Nat = { v:Int | v >= 0 } 

type VectorClock = [Nat]

type VC sized N = { vc:VectorClock | len vc == N } 

type VCsameLength V = VC sized {len V}

vcMerge :: v:VectorClock -> VCsameLength {v} -> VCsameLength {v}
vcMerge = zipWith max


type Commutative a A = x:a -> y:a -> { _:Proof | A x y == A y x }
type Nat = { v:Int | v >= 0 }

type VectorClock = [Nat]

type VC sized N = { vc:VectorClock | len vc == N }

type VC same Length V = VC sized {len V}

vcMerge :: v:VectorClock -> VC same Length {v} -> VC same Length {v}

vcMerge = zipWith max

type Commutative a A = x:a -> y:a -> { _:Proof | A x y == A y x }

Refinement reflection
type Nat = \{ v: \text{Int} \mid v \geq 0 \}

(type VectorClock = [Nat]

type VCsize N = \{ vc: VectorClock \mid \text{len} \ vc == N \}

type VCsameLength V = VCsize {\text{len} V}

vcMerge :: \( \text{v: VectorClock} \rightarrow \text{VCsameLength \{v\}} \rightarrow \text{VCsameLength \{v\}} \)

vcMerge = zipWith max

(type Commutative a A = x:a -> y:a -> \{ _:Proof \mid A \times y == A y x \}

vcMergeComm :: n: \text{Nat} \rightarrow \text{Commutative (VCsize n) vcMerge}

Refinement \textit{reflection}
type Nat = { v:Int | v >= 0 }

type VectorClock = [Nat]

type VCsized N = { vc:VectorClock | len vc == N }

type VCsameLength V = VCsized {len V}

vcMerge :: v:VectorClock -> VCsameLength {v} -> VCsameLength {v}
vcMerge = zipWith max

vcMergeComm :: n:Nat -> Commutative (VCsized n) vcMerge
vcMergeComm _n [] [] = ()
vcMergeComm n (_x:xs) (_y:ys) = vcMergeComm (n - 1) xs ys

Refinement reflection
type Nat = { v:Int | v >= 0 }

type VectorClock = [Nat]

type VCsize N = { vc:VectorClock | len vc == N }

type VCsameLength V = VCsize {len V}

vcMerge :: v:VectorClock -> VCsameLength {v} -> VCsameLength {v}
vcMerge = zipWith max

vcMergeComm :: n:Nat -> Commutative (VCsize n) vcMerge
vcMergeComm _n [] [] = ()
vcMergeComm n (_x:xs) (_y:ys) = vcMergeComm (n - 1) xs ys

Refinement reflection
**Type**

```
type Nat = { v: Int | v >= 0 }
type VectorClock = [Nat]
```

**Type Definition**

```
type VC sized N = { vc: VectorClock | len vc == N }
type V CsameLength V = V C sized {len V}
```

```
vcMerge :: v: VectorClock -> V CsameLength {v} -> V CsameLength {v}
vcMerge = zipWith max
```

**Type**

```
type Commutative a A = x: a -> y: a -> { _ : Proof | A x y == A y x }
```

```
vcMergeComm :: n:Nat -> Commutative (V C sized n) vcMerge
vcMergeComm _n [] [] = ()
vcMergeComm n (_x:xs) (_y:ys) = vcMergeComm (n - 1) xs ys
```

**Refinement** *reflection*
type Nat = { v:Int | v >= 0 }

type VectorClock = [Nat]

type VCsize N = { vc:VectorClock | len vc == N }

type VCsameLength V = VCsize {len V}

vcMerge :: v:VectorClock -> VCsameLength {v} -> VCsameLength {v}
vcMerge = zipWith max

vcMergeComm :: n:Nat -> Commutative (VCsize n) vcMerge
vcMergeComm _n [] [] = ()
vcMergeComm n (_x:xs) (_y:ys) = vcMergeComm (n - 1) xs ys

Refinement reflection
type Nat = { v:Int | v >= 0 }
type VectorClock = [Nat]

type VC sized N = { vc:VectorClock | len vc == N }
type VCsameLength V = VCsized {len V}

vcMerge :: v:VectorClock -> VCsameLength {v} -> VCsameLength {v}
vcMerge = zipWith max

vcMergeComm :: n:Nat -> Commutative (VC sized n) vcMerge
vcMergeComm _n [] [] = ()
vcMergeComm n (_x:xs) (_y:ys) = vcMergeComm (n - 1) xs ys

Refinement reflection
5.1 CBCAST Protocol

Suppose that a set of processes $P$ communicate using only broadcasts to the full set of processes in the system; that is, $\forall m: \text{dests}(m) = P$. We now develop a delivery protocol by which each process $p$ receives messages sent to it, delays them if necessary, and then delivers them in an order consistent with causality:

$$m \rightarrow m' = \forall p: \text{deliver}_p(m) \rightarrow \text{deliver}_p(m').$$

Birman et al., “Lightweight Causal and Atomic Group Multicast”
ACM TOCS, 1991
5.1 CBCAST Protocol

Suppose that a set of processes $P$ communicate using only broadcasts to the full set of processes in the system; that is, $\forall m: \text{dests}(m) = P$. We now develop a \textit{delivery protocol} by which each process $p$ receives messages sent to it, delays them if necessary, and then delivers them in an order consistent with causality:

$$m \rightarrow m' = \forall p: \text{deliver}_p(m) \rightarrowP \text{deliver}_p(m').$$

Birman et al., “Lightweight Causal and Atomic Group Multicast”
ACM TOCS, 1991

\texttt{type CausalDelivery \ p =}
\begin{verbatim}
  \{ m : Message | elem (Deliver m ) (pHist p) \}
  \rightarrow \{ m': Message | elem (Deliver m') (pHist p)
  \&\& causallyBefore m m' \}
  \rightarrow \{ _ : Proof | ordered (pHist p) (Deliver m) (Deliver m') \}
\end{verbatim}
type CausalDelivery p =
    { m : Message | elem (Deliver m) (pHist p) }
  -> { m' : Message | elem (Deliver m') (pHist p)
      && causallyBefore m m' }
  -> { _ : Proof | ordered (pHist p) (Deliver m) (Deliver m') }
type CausalDelivery p =
  { m : Message | elem (Deliver m) (pHist p) }
  -> { m' : Message | elem (Deliver m') (pHist p)
                 && causallyBefore m m' }
  -> { _ : Proof | ordered (pHist p) (Deliver m) (Deliver m') }


step :: Op -> Process -> Process
type CausalDelivery p =
    { m : Message | elem (Deliver m) (pHist p) }
  -> { m' : Message | elem (Deliver m') (pHist p)
       && causallyBefore m m' }
  -> { _ : Proof | ordered (pHist p) (Deliver m) (Deliver m') }

step :: Op -> Process -> Process

causalDeliveryPreservation :: ops : [Op]
  -> p : Process
  -> CausalDelivery p
  -> CausalDelivery (foldr step p ops)
causalDeliveryPreservation = ... – a few hundred lines
type CausalDelivery p =
  { m : Message | elem (Deliver m) (pHist p) }
-> { m' : Message | elem (Deliver m') (pHist p)
    && causallyBefore m m' }
-> { _ : Proof | ordered (pHist p) (Deliver m) (Deliver m') }


step :: Op -> Process -> Process

causalDeliveryPreservation :: ops : [Op]
  -> p : Process
  -> CausalDelivery p
  -> CausalDelivery (foldr step p ops)

causalDeliveryPreservation = ... – a few hundred lines
Programmers should be able to...

mechanically express and prove correctness properties
Programmers should be able to...

mechanically express and prove correctness properties

...of executable implementations of distributed systems
Programmers should be able to...

mechanically express and prove correctness properties
...of executable implementations of distributed systems
...using language-integrated verification tools (i.e., types!)
Programmers should be able to...

mechanically express and prove correctness properties
...of executable implementations of distributed systems
...using language-integrated verification tools (i.e., types!)

[HATRA 2021]
Tak!

Languages, Systems, and Data Lab: lsd.ucsc.edu
Lindsey’s research blog: decomposition.al

github.com/lsd-ucsc/cbcast-lh