Verified Causal Broadcast with Liquid Haskell

Patrick Redmond  Gan Shen  Niki Vazou  Lindsey Kuper

UC Berkeley Programming Systems Seminar
24 October 2022

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

Patrick Redmond  Gan Shen  Niki Vazou  Lindsey Kuper

UC Berkeley Programming Systems Seminar
24 October 2022

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

Patrick Redmond  Gan Shen  Niki Vazou  Lindsey Kuper

UC Berkeley Programming Systems Seminar
24 October 2022

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

Patrick Redmond  Gan Shen  Niki Vazou  Lindsey Kuper

UC Berkeley Programming Systems Seminar 24 October 2022

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

Patrick Redmond  Gan Shen  Niki Vazou  Lindsey Kuper

UC Berkeley Programming Systems Seminar
24 October 2022

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

Patrick Redmond  Gan Shen  Niki Vazou  Lindsey Kuper

UC Berkeley Programming Systems Seminar
24 October 2022

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

Patrick Redmond  Gan Shen  Niki Vazou  Lindsey Kuper

UC Berkeley Programming Systems Seminar
24 October 2022

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

Found it!
Lost my 😞 …

Found it!
Lost my...

Found it!

😕

happens-before

2
Lost my ...  😕

Found it!

happens-before
Lost my…

🙂

FIFO delivery

Found it!

Lost my…

happens-before

Found it!
Lost my …

Found it!

🙂

Yay!

FIFO delivery

happens-before

2
Lost my ...

Found it!

FIFO delivery

Yay!

😮

😮

happens-before

2
Lost my …

Found it!

😊

😮

Yay!

FIFO delivery

Lost my …

Found it!

😮

happens-before
Lost my

Found it!

FIFO delivery

Yay!

happens-before
Lost my …

Found it!

🙂

Yay!

FIFO delivery

happens-before

Lost my …

Found it!

🙂

🙂

happens-before

FIFO delivery

Yay!
Causal broadcast with vector clocks [Birman et al., 1991]
Causal broadcast with vector clocks [Birman et al., 1991]
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
- \( \leq \) 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
- ≤ 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 \[\text{[Birman et al., 1991]}\]
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]
Programmers should be able to...

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

express and prove interesting correctness properties

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

express and prove interesting correctness properties
...of deployable implementations of distributed systems
...using language-integrated verification tools
Programmers should be able to...

express and prove interesting correctness properties
...of deployable implementations of distributed systems
...using language-integrated verification tools (i.e., types!)
Refinement types
type Nat = \{ v : Int \mid v \geq 0 \}
type Nat = \{ v : \text{Int} \mid v \geq 0 \}
type Nat = { v: Int | v >= 0 }
type VectorClock = [Nat]

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

type VectorClock = [Nat]

\text{vcMerge} :: \text{VectorClock} \to \text{VectorClock} \to \text{VectorClock}

\textbf{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 VC sized N = { vc:VectorClock | len vc == N }
type Nat = { v:Int | v >= 0 }

type VectorClock = [Nat]

cMerge :: VectorClock -> VectorClock -> VectorClock
cMerge = zipWith max

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

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

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


type VCsameLength V = VCsized {len V}  

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

cMerge = zipWith max  

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

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

vcMerge :: v: VectorClock \rightarrow VCsameLength \; \{ v \} \rightarrow VCsameLength \; \{ v \} \\
vcMerge = \text{zipWith max} \\

Refinement \, \text{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

Refinement reflection
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: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

Refinement reflection

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

type VectorClock = [\text{Nat}]

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

type VCssameLength V = VCssized \{\text{len} \ V\}

vcMerge :: v:\text{VectorClock} \to VCssameLength \{v\} \to VCssameLength \{v\}
vcMerge = \text{zipWith max}

type Commutative a A = x:a \to y:a \to \{ \text{\_}:\text{Proof} \mid A \ x \ y == A \ y \ x \}

vcMergeComm :: n:\text{Nat} \to \text{Commutative} (\text{VC sized} \ n) \ vcMerge

Reﬁnement \text{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

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

vcMergeComm :: n: Nat -> Commutative (VCSized n) vcMerge

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 :: vc: 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
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


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

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 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 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

Re
fl
en
men

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

vcMergeComm :: n: Nat -> Commutative (VCsized n) vcMerge

vcMergeComm _n [] [] = ()
vcMergeComm n (_x:xs) (_y:ys) = vcMergeComm (n - 1) xs ys

Re
fl
en
men

Refinement 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

application code

verification code

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

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

Refinement reflection
(Local) causal delivery as a refinement type

’s process history (pHist):
[(Deliver “Lost my 📱”),
(Deliver “Found it!”),
(Broadcast “Yay!”),
…]
(Local) causal delivery as a refinement type

```
type LocalCausalDelivery P =
    { m1 : Message | elem (Deliver (pID P) m1) (pHist P) }
  -> { m2 : Message | elem (Deliver (pID P) m2) (pHist P)
         && vcLess (mVC m1) (mVC m2) }
  -> { _ : Proof | processOrder (pHist P) (Deliver (pID P) m1)
                                  (Deliver (pID P) m2) }
```

Ages process history (pHist):
[(Deliver “Lost my 📱”),
 (Deliver “Found it!”),
 (Broadcast “Yay!”),
 ...]
(Local) causal delivery as a refinement type

type LocalCausalDelivery P =
  { m1 : Message | elem (Deliver (pID P) m1) (pHist P) }
-> { m2 : Message | elem (Deliver (pID P) m2) (pHist P)
    && vcLess (mVC m1) (mVC m2) }
-> { _ : Proof | processOrder (pHist P) (Deliver (pID P) m1)
    (Deliver (pID P) m2) }

’s process history (pHist):
[(Deliver “Lost my 📱”),
(Deliver “Found it!”),
(Broadcast “Yay!”),
...]
(Local) causal delivery as a refinement type

```plaintext
type LocalCausalDelivery P =
  { m1 : Message | elem (Deliver (pID P) m1) (pHist P) }
-> { m2 : Message | elem (Deliver (pID P) m2) (pHist P) 
  && vcLess (mVC m1) (mVC m2) }
-> { _: Proof | processOrder (pHist P) (Deliver (pID P) m1) 
  (Deliver (pID P) m2) }
```

His process history (pHist):
[(Deliver “Lost my 📱”),
 (Deliver “Found it!”),
 (Broadcast “Yay!”),
 ...]
(Local) causal delivery as a refinement type

```
type LocalCausalDelivery P =
    { m1 : Message | elem (Deliver (pID P) m1) (pHist P) }
  -> { m2 : Message | elem (Deliver (pID P) m2) (pHist P)
             && vcLess (mVC m1) (mVC m2) }
  -> { _ : Proof | processOrder (pHist P) (Deliver (pID P) m1)
                      (Deliver (pID P) m2) }
```

`s process history (pHist):
[(Deliver “Lost my 📱”),
 (Deliver “Found it!”),
 (Broadcast “Yay!”),
 ...]`
(Local) causal delivery as a refinement type

```haskell
type LocalCausalDelivery P =
    { m1 : Message | elem (Deliver (pID P) m1) (pHist P) }
-> { m2 : Message | elem (Deliver (pID P) m2) (pHist P)
    && vcLess (mVC m1) (mVC m2) }
-> { _: Proof | processOrder (pHist P) (Deliver (pID P) m1)
    (Deliver (pID P) m2) }
```

A process's history (pHist):
```
[(Deliver 🎈 “Lost my 🎈”),
  (Deliver 🎈 “Found it!”),
  (Broadcast “Yay!”),
  …]
```
Running the protocol preserves (local) causal delivery
Running the protocol preserves (local) causal delivery

\[
\text{data } \text{Op } r = \text{OpBroadcast } r \mid \text{OpReceive } (\text{Message } r) \mid \text{OpDeliver}
\]

\[
\text{step } :: \text{Op } r \rightarrow \text{Process} \rightarrow \text{Process}
\]

\[
\begin{align*}
\text{step } (\text{OpBroadcast } r) & \quad p = \ldots \\
\text{step } (\text{OpReceive } m) & \quad p = \ldots \\
\text{step } (\text{OpDeliver}) & \quad p = \ldots
\end{align*}
\]

application code

verification code
Running the protocol preserves (local) causal delivery

data Op r = OpBroadcast r | OpReceive (Message r) | OpDeliver

step :: Op r -> Process -> Process
step (OpBroadcast r) p = ...
step (OpReceive m) p = ...
step (OpDeliver) p = ...

application code

lcdStep :: op : Op r
        -> p : Process
        -> LocalCausalDelivery p
        -> LocalCausalDelivery (step p op)
lcdStep op p lcdp =
case op ? step op p of
  OpBroadcast r -> ... -- short proof
  OpReceive m -> ... -- short proof
  OpDeliver -> ... -- long proof

verification code
Running the protocol preserves (local) causal delivery

data Op r = OpBroadcast r | OpReceive (Message r) | OpDeliver

step :: Op r -> Process -> Process
step (OpBroadcast r) p = ...
step (OpReceive m) p = ...
step (OpDeliver) p = ...

application code

lcdStep :: op : Op r
-> p : Process
-> LocalCausalDelivery p
-> LocalCausalDelivery (step p op)

lcdStep op p lcdp =
case op ? step op p of
  OpBroadcast r -> ... -- short proof
  OpReceive m -> ... -- short proof
  OpDeliver -> ... -- long proof

verification code
Running the protocol preserves (local) causal delivery

```
data Op r = OpBroadcast r | OpReceive (Message r) | OpDeliver

step :: Op r -> Process -> Process
step (OpBroadcast r) p = ...  
step (OpReceive m)   p = ...  
step (OpDeliver)      p = ...
```

application code

```
lcdStep :: op : Op r
         -> p : Process
         -> LocalCausalDelivery p
         -> LocalCausalDelivery (step p op)

lcdStep op p lcdp =
  case op ? step op p of
     OpBroadcast r   -> ... -- short proof
     OpReceive m     -> ... -- short proof
     OpDeliver      -> ... -- long proof
```

verification code
Running the protocol preserves (local) causal delivery

data Op r = OpBroadcast r | OpReceive (Message r) | OpDeliver

step :: Op r -> Process -> Process
step (OpBroadcast r) p = ... 
step (OpReceive m) p = ... 
step (OpDeliver) p = ...

application code

lcdStep :: op : Op r 
       -> p : Process
       -> LocalCausalDelivery p
       -> LocalCausalDelivery (step p op)

lcdStep op p lcdn =
case op ? step op p of
  OpBroadcast r -> ... -- short proof
  OpReceive m -> ... -- short proof
  OpDeliver    -> ... -- long proof

verification code
Running the protocol for one step preserves local causal delivery

\[ \downarrow = \text{“relies on”} \]
Running the protocol for *one* step preserves *local* causal delivery.
Running the protocol for one step preserves local causal delivery.

broadcast, receive, deliver
each preserve local causal delivery
(deliver is the hard part)

\[ \downarrow = \text{“relies on”} \]
Running the protocol for one step preserves causal delivery

Running the protocol for one step preserves local causal delivery

broadcast, receive, deliver each preserve local causal delivery
(deliver is the hard part)

\[ \downarrow = \text{“relies on”} \]
Running the protocol for one step preserves causal delivery.

Running the protocol for one step preserves local causal delivery.

broadcast, receive, deliver each preserve local causal delivery (deliver is the hard part)

Causal delivery [Birman et al., 1991]:  
\[ m \rightarrow m' = \forall p: \text{deliver}_p(m) \xrightarrow{p} \text{deliver}_p(m') \]
Running the protocol for one step preserves causal delivery

Running the protocol for one step preserves local causal delivery

broadcast, receive, deliver each preserve local causal delivery (deliver is the hard part)

Causal delivery [Birman et al., 1991]: \( m \rightarrow m' = \forall p: \text{deliver}_p(m) \xrightarrow{p} \text{deliver}_p(m') \)

```
type CausalDelivery X =
    pid : PID -- any pid in the domain of execution X
-> { m : Message | elem (Deliver pid m) (pHist (X pid)) }
-> { m' : Message | elem (Deliver pid m') (pHist (X pid))
    && happensBefore X (Broadcast m) (Broadcast m') }
-> { _ : Proof | procOrder (pHist (X pid)) (Deliver pid m) (Deliver pid m') }
```
Running the protocol for one step preserves causal delivery

Running the protocol for one step preserves local causal delivery

broadcast, receive, deliver
each preserve local causal delivery
(deliver is the hard part)

Causal delivery [Birman et al., 1991]: $m \rightarrow m' = \forall p: \text{deliver}_p(m) \xRightarrow{p} \text{deliver}_p(m')$

```
@inproceedings{Birman1991CausalDelivery,
  author = {Paulo V. M. Birman and David Verhoek and Akim Doyen},
  title = {Causal Delivery},
  pages = {185-196},
  year = {1991},
}
```

**type** CausalDelivery $X =$

<table>
<thead>
<tr>
<th>pid : PID -- any pid in the domain of execution $X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rightarrow { m : \text{Message}</td>
</tr>
<tr>
<td>$\rightarrow { m' : \text{Message}</td>
</tr>
<tr>
<td>$&amp;&amp; \text{happensBefore X (Broadcast m) (Broadcast m')} }$</td>
</tr>
<tr>
<td>$\rightarrow { _ : \text{Proof}</td>
</tr>
</tbody>
</table>

**verification code**
Running the protocol for one step preserves causal delivery

broadcast, receive, deliver each preserve local causal delivery (deliver is the hard part)

Causal delivery [Birman et al., 1991]: \[ m \rightarrow m' = \forall p: \text{deliver}_p(m) \xrightarrow{p} \text{deliver}_p(m') \]

type CausalDelivery X = pid: PID -- any pid in the domain of execution X

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>`m : Message</td>
<td>elem (Deliver pid m ) (pHist (X pid)) }`</td>
</tr>
<tr>
<td>`m' : Message</td>
<td>elem (Deliver pid m') (pHist (X pid))</td>
</tr>
<tr>
<td><code>&amp;\&amp; \text{happensBefore X (Broadcast m) (Broadcast m')} \} </code></td>
<td></td>
</tr>
<tr>
<td>`=: Proof</td>
<td>procOrder (pHist (X pid)) (Deliver pid m) (Deliver pid m') `</td>
</tr>
</tbody>
</table>

\[ \downarrow \text{ = "relies on" } \]
Causal delivery [Birman et al., 1991]: \( m \rightarrow m' = \forall p: \text{deliver}_p(m) \rightarrow \text{deliver}_p(m') \)

**type** CausalDelivery X =

```plaintext
pid : PID -- any pid in the domain of execution X

-> { m : Message | elem (Deliver pid m) (pHist (X pid)) }

-> { m' : Message | elem (Deliver pid m') (pHist (X pid))
    && happensBefore X (Broadcast m) (Broadcast m') }

-> { _: Proof | procOrder (pHist (X pid)) (Deliver pid m) (Deliver pid m') }
```

\( \downarrow = \text{“relies on”} \)
Running the protocol for one step preserves causal delivery

whole execution observes causal delivery → each process observes **local** causal delivery

vector clocks **reflect** happens-before

Running the protocol for one step preserves **local** causal delivery

broadcast, receive, deliver each preserve **local** causal delivery (deliver is the hard part)

Causal delivery [Birman et al., 1991]:

\[ m \rightarrow m' = \forall p: \text{deliver}_p(m) \overset{p}{\Rightarrow} \text{deliver}_p(m') \]

type CausalDelivery X =

\[
\text{pid: PID} -- \text{any pid in the domain of execution X}
\]

\[
\rightarrow \{ \text{m: Message} \mid \text{elem (Deliver pid m) (pHist (X pid))} \}
\rightarrow \{ \text{m': Message} \mid \text{elem (Deliver pid m') (pHist (X pid))} \}
\rightarrow \{ \_ : \text{Proof} \mid \text{procOrder (pHist (X pid)) (Deliver pid m) (Deliver pid m')} \}
\]

\[ \downarrow = "\text{relies on}" \]
Running the protocol for one step preserves causal delivery

whole execution observes causal delivery →
each process observes local causal delivery

vector clocks reflect happens-before
broadcast, receive, deliver each preserve local causal delivery (deliver is the hard part)

Causal delivery [Birman et al., 1991]: \[ m \rightarrow m' = \forall p: \text{deliver}_p(m) \xrightarrow{p} \text{deliver}_p(m') \]

**type** CausalDelivery X =

\begin{align*}
\text{pid} : \text{PID} & \quad \text{-- any pid in the domain of execution X} \\
\rightarrow \{ m : \text{Message} \mid \text{elem (Deliver pid m) (pHist (X pid))} \} \\
\rightarrow \{ m' : \text{Message} \mid \text{elem (Deliver pid m') (pHist (X pid))} \} \\
& \quad \text{&& \ happenBefore X (Broadcast m) (Broadcast m')} \\
\rightarrow \{ _ : \text{Proof} \mid \text{procOrder (pHist (X pid)) (Deliver pid m) (Deliver pid m')} \}
\end{align*}

\[ = \text{“relies on”} \]
Running the protocol for *one* step preserves causal delivery

whole execution observes causal delivery $\rightarrow$

each process observes local causal delivery

vector clocks reflect happens-before

broadcast, receive, deliver

each preserve local causal delivery
(deliver is the hard part)

Causal delivery [Birman et al., 1991]:

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

**type** CausalDelivery $X =$

```haskell
pid : PID -- any pid in the domain of execution $X$

-> { m : Message | elem (Deliver pid m) (pHist (X pid)) }

-> { m' : Message | elem (Deliver pid m') (pHist (X pid))

  && happensBefore X (Broadcast m) (Broadcast m') }

-> { _ : Proof | procOrder (pHist (X pid)) (Deliver pid m) (Deliver pid m') }
```

= “relies on”
Running the protocol for any number of steps preserves causal delivery

(+ induction)

Running the protocol for one step preserves causal delivery

whole execution observes causal delivery $\rightarrow$ each process observes local causal delivery

vector clocks reflect happens-before

broadcast, receive, deliver each preserve local causal delivery (deliver is the hard part)

each process observes local causal delivery $\rightarrow$ whole execution observes causal delivery

Causal delivery [Birman et al., 1991]: $m \rightarrow m' = \forall p: \overrightarrow{\text{deliver}}_p(m) \overrightarrow{\text{deliver}}_p(m')$

**type** CausalDelivery $X =$

- $\text{pid : PID}$ -- any pid in the domain of execution $X$

- $\rightarrow \{ m : \text{Message} | \text{elem } (\text{Deliver } \text{pid } m) (\text{pHist } (X \text{ pid})) \}$

- $\rightarrow \{ m' : \text{Message} | \text{elem } (\text{Deliver } \text{pid } m') (\text{pHist } (X \text{ pid}))$

  && \text{happensBefore } X (\text{Broadcast } m) (\text{Broadcast } m') \}$

- $\rightarrow \{ \_ : \text{Proof} | \text{procOrder } (\text{pHist } (X \text{ pid})) (\text{Deliver } \text{pid } m) (\text{Deliver } \text{pid } m') \}$

**verification code**
Building apps with causal broadcast
Building apps with causal broadcast

- **App logic**
  - **App state**
  - **Delay queue**
    - **Deliverable?**
    - **Broadcast**
    - **Deliver**
    - **Receive**

- **Message transport**
  - **WAN**
  - **Node**
    - **Node**
    - **Node**
    - **Node**
Building apps with causal broadcast

- **Node**
  - App logic
  - App state
  - Delay queue
    - deliverable?
  - deliver
  - receive
  - broadcast

- **Message transport**

- **WAN**

- **Node**
Building apps with causal broadcast
Building apps with causal broadcast
Building apps with causal broadcast

Credit: Matthew Weidner
Programmers should be able to...

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

Programmers should be able to...

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

[HATRA 2021]
Thank you!

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

github.com/lsd-ucsc/cbcast-lh
Thank you!

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

github.com/lsd-ucsc/cbcast-lh