Verified Causal Broadcast with Liquid Haskell

Patrick Redmond Gan Shen Niki Vazou Lindsey Kuper

NUS PLSE Seminar
12 October 2022

github.com/lsd-ucsc/cbcast-lh

Verified Causal Broadcast with Liquid Haskell

Patrick Redmond

Gan Shen
Niki Vazou
Lindsey Kuper

NUS PLSE Seminar
12 October 2022

UNIVERSITY OF CALIFORNIA SANTA CRUZ

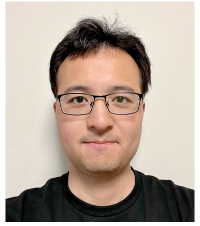
institute imdea software

github.com/lsd-ucsc/cbcast-lh

Verified Causal Broadcast with Liquid Haskell

Patrick Redmond

Gan Shen



Niki Vazou

Lindsey Kuper

NUS PLSE Seminar
12 October 2022

UNIVERSITY OF CALIFORNIA SANTA CRUZ

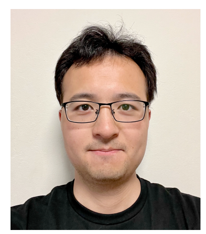
imdea software institute

github.com/lsd-ucsc/cbcast-lh

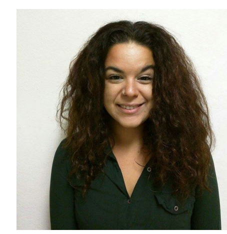
Verified Causal Broadcast with Liquid Haskell

Patrick Redmond

Gan Shen



Niki Vazou



Lindsey Kuper

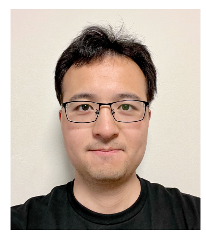
NUS PLSE Seminar
12 October 2022

github.com/lsd-ucsc/cbcast-lh

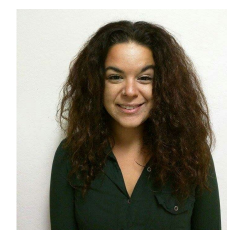
Verified Causal Broadcast with Liquid Haskell

Patrick Redmond

Gan Shen



Niki Vazou



Lindsey Kuper

NUS PLSE Seminar
12 October 2022

UNIVERSITY OF CALIFORNIA SANTA CRUZ

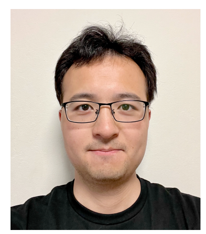
institute imdea software

github.com/lsd-ucsc/cbcast-lh

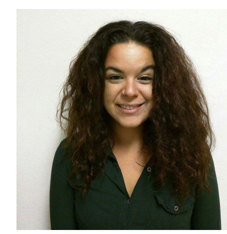
Verified Causal Broadcast with Liquid Haskell

Patrick Redmond

Gan Shen



Niki Vazou



Lindsey Kuper

NUS PLSE Seminar
12 October 2022

UNIVERSITY OF CALIFORNIA SANTA CRUZ

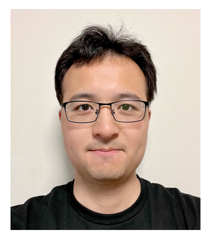
institute IMDEA software

github.com/lsd-ucsc/cbcast-lh

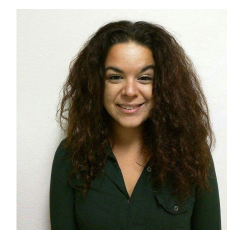
Verified Causal Broadcast with Liquid Haskell

Patrick Redmond

Gan Shen



Niki Vazou



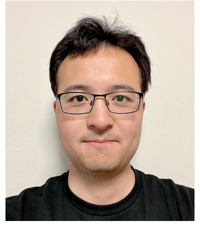
Lindsey Kuper

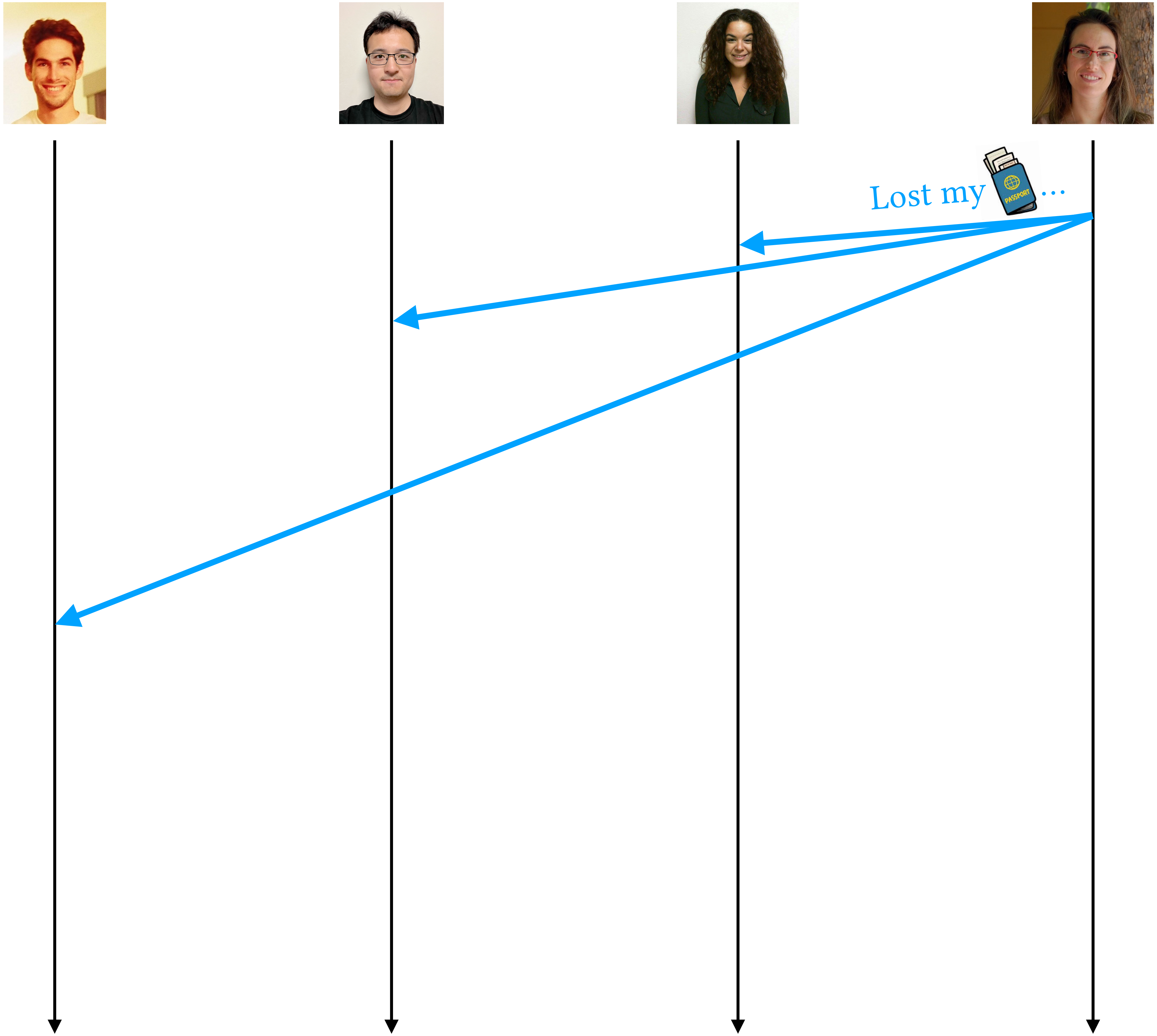
NUS PLSE Seminar
12 October 2022

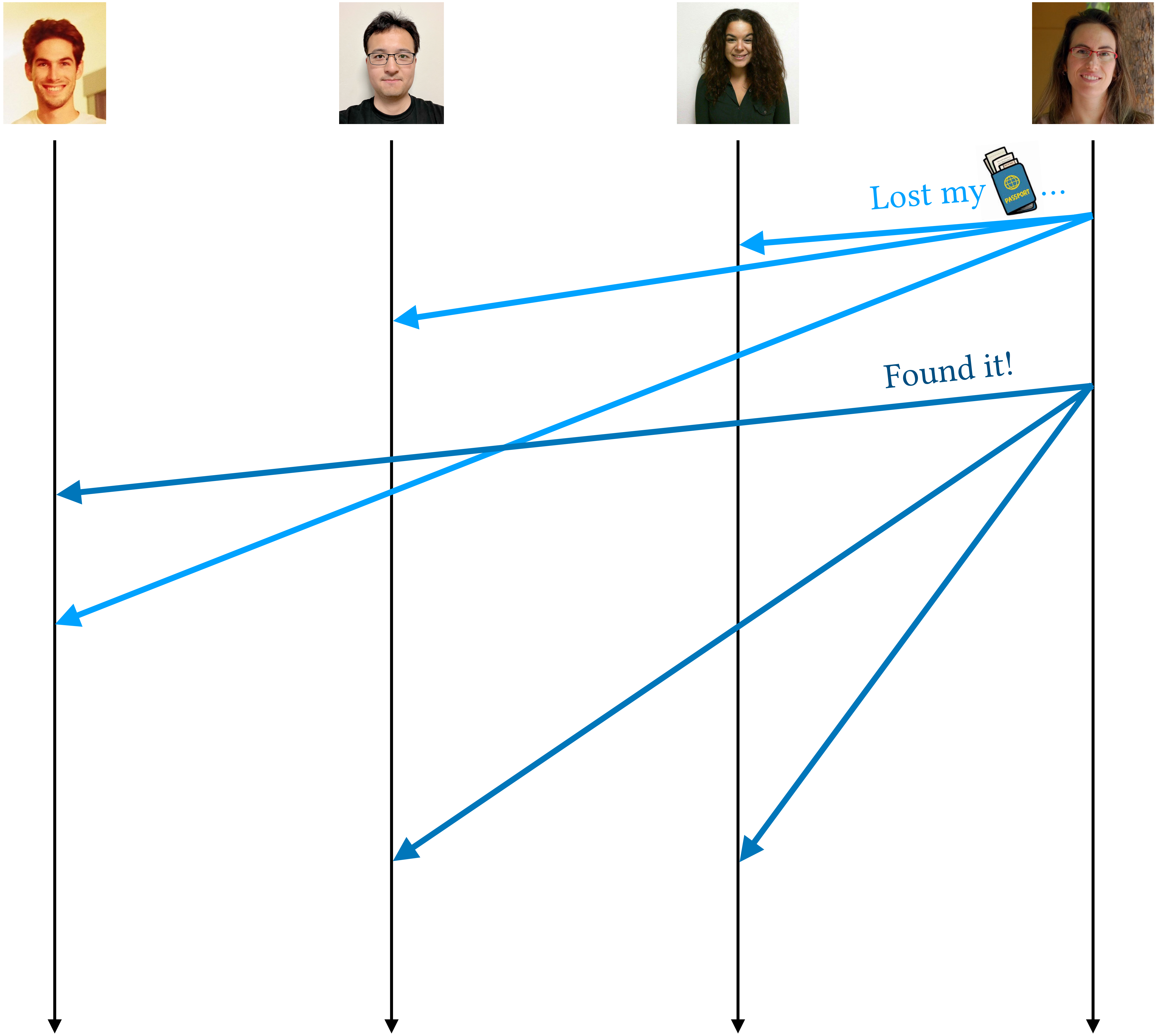
UNIVERSITY OF CALIFORNIA SANTA CRUZ

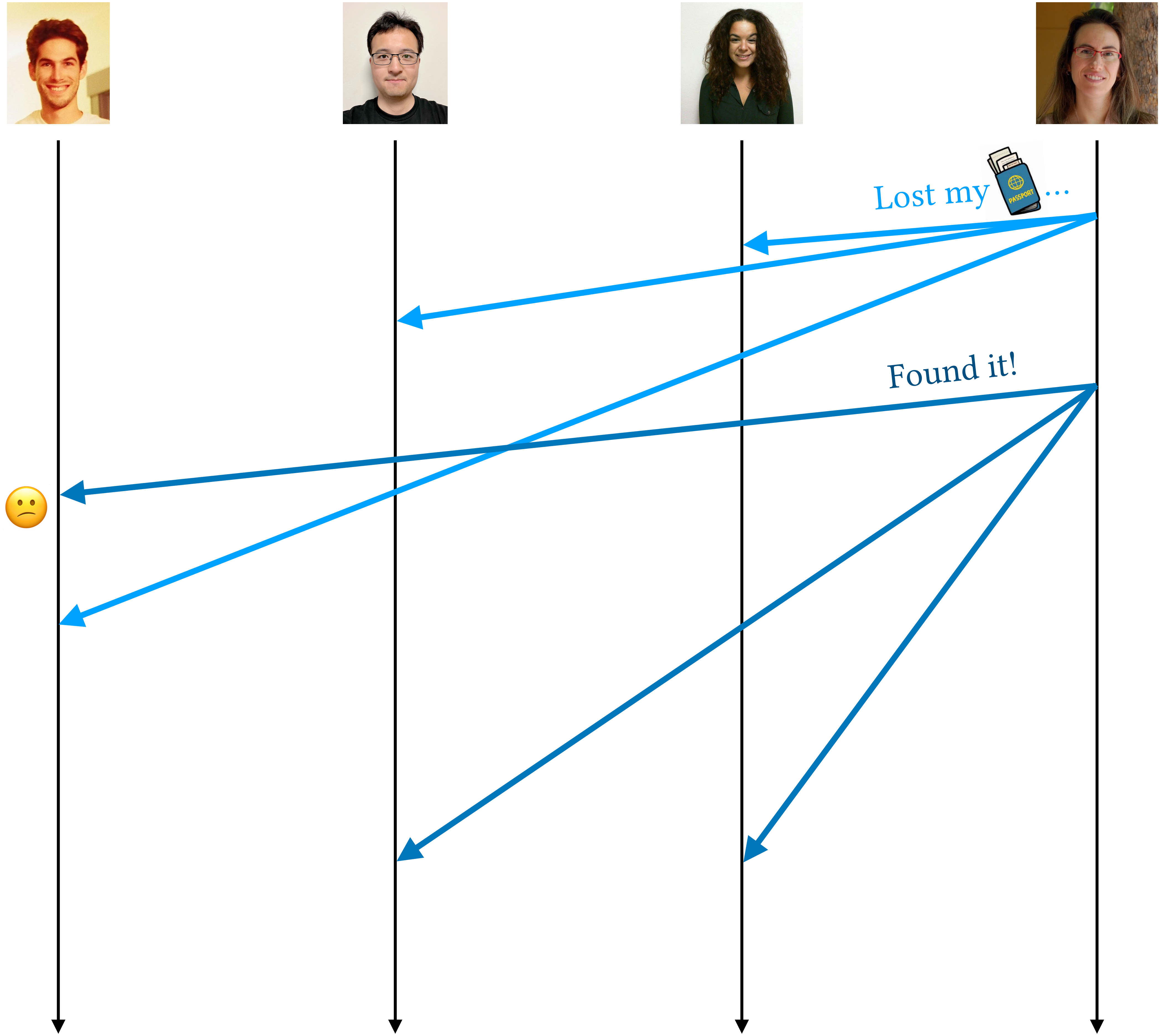
institute IMDEA software

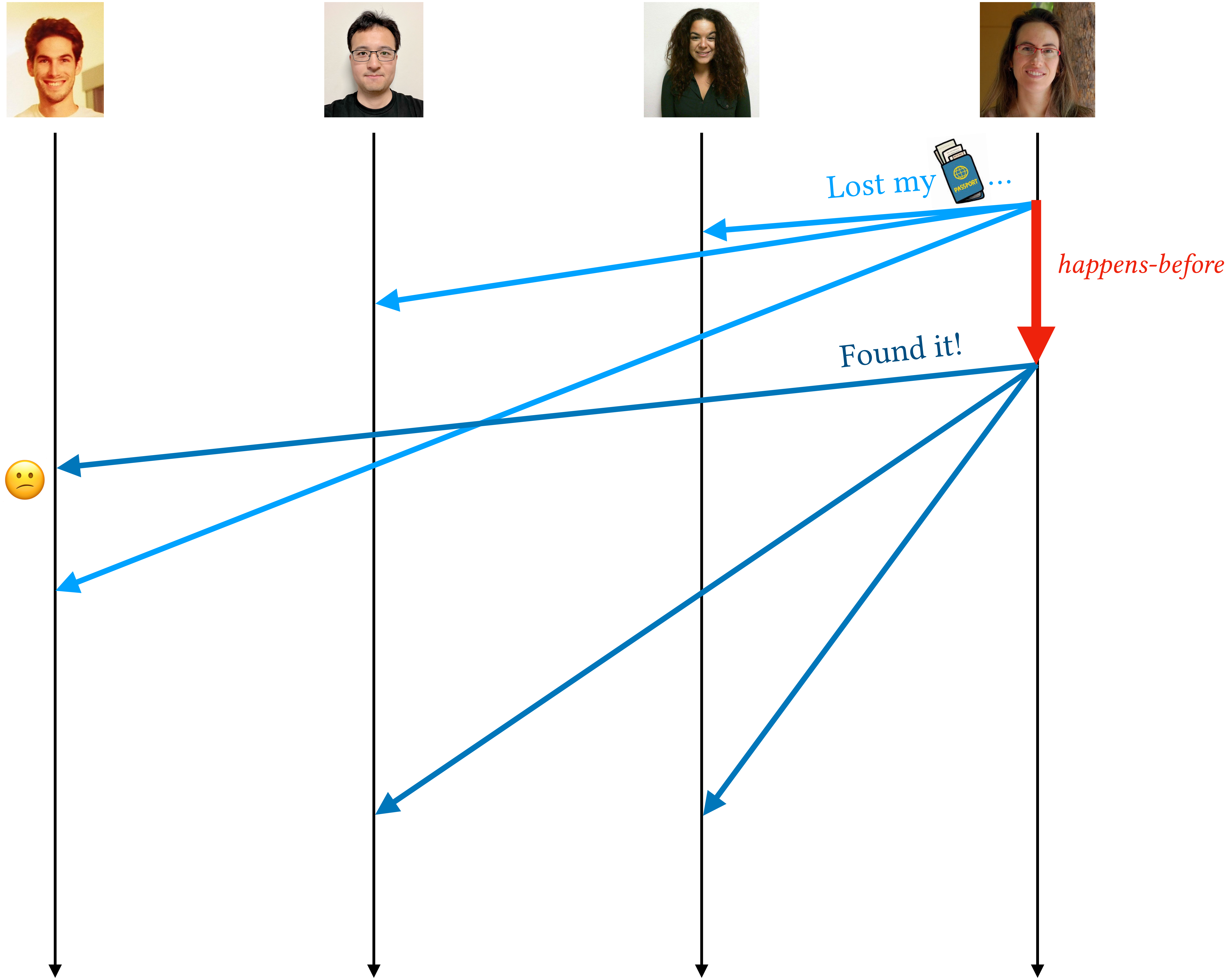
github.com/lsd-ucsc/cbcast-lh

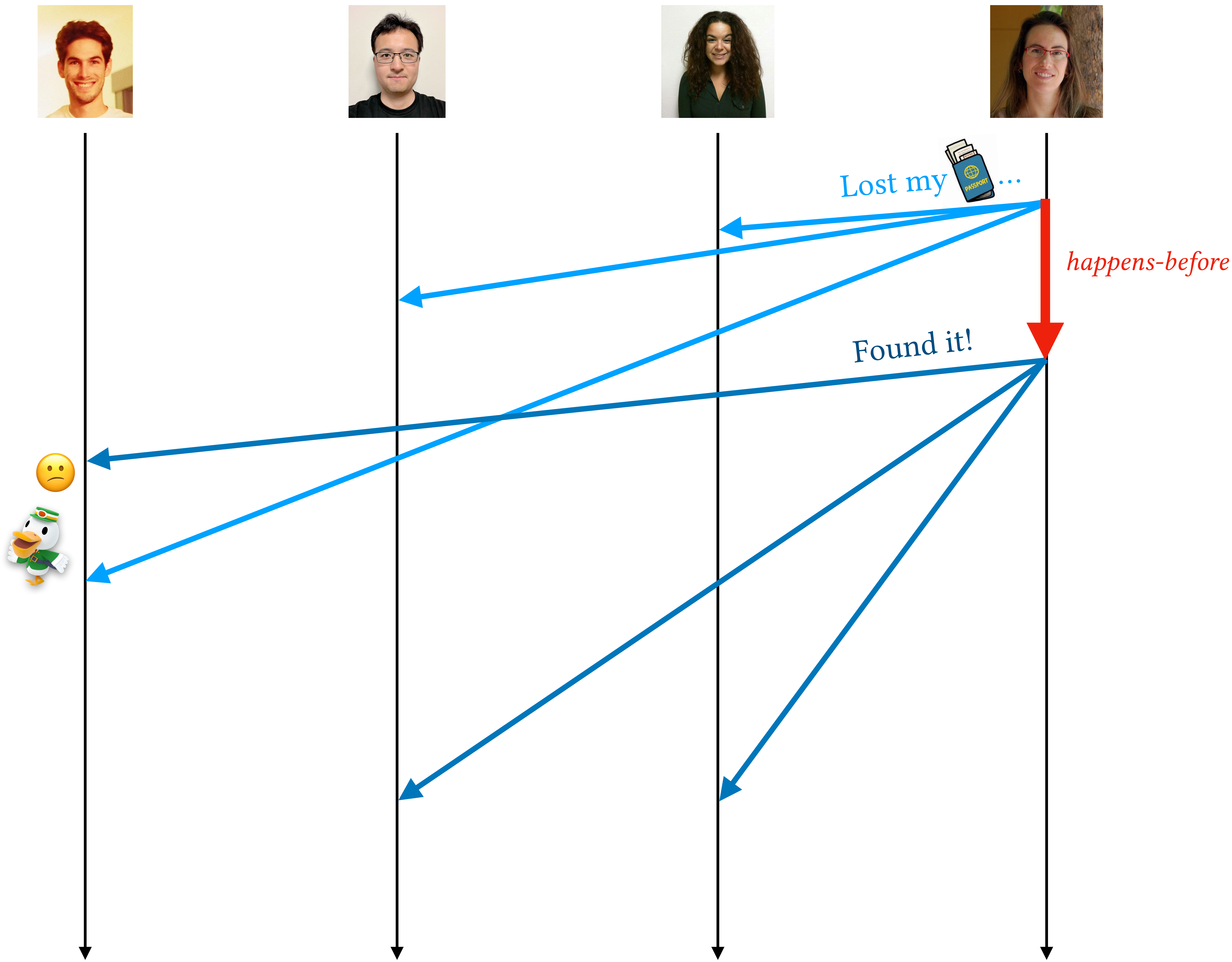



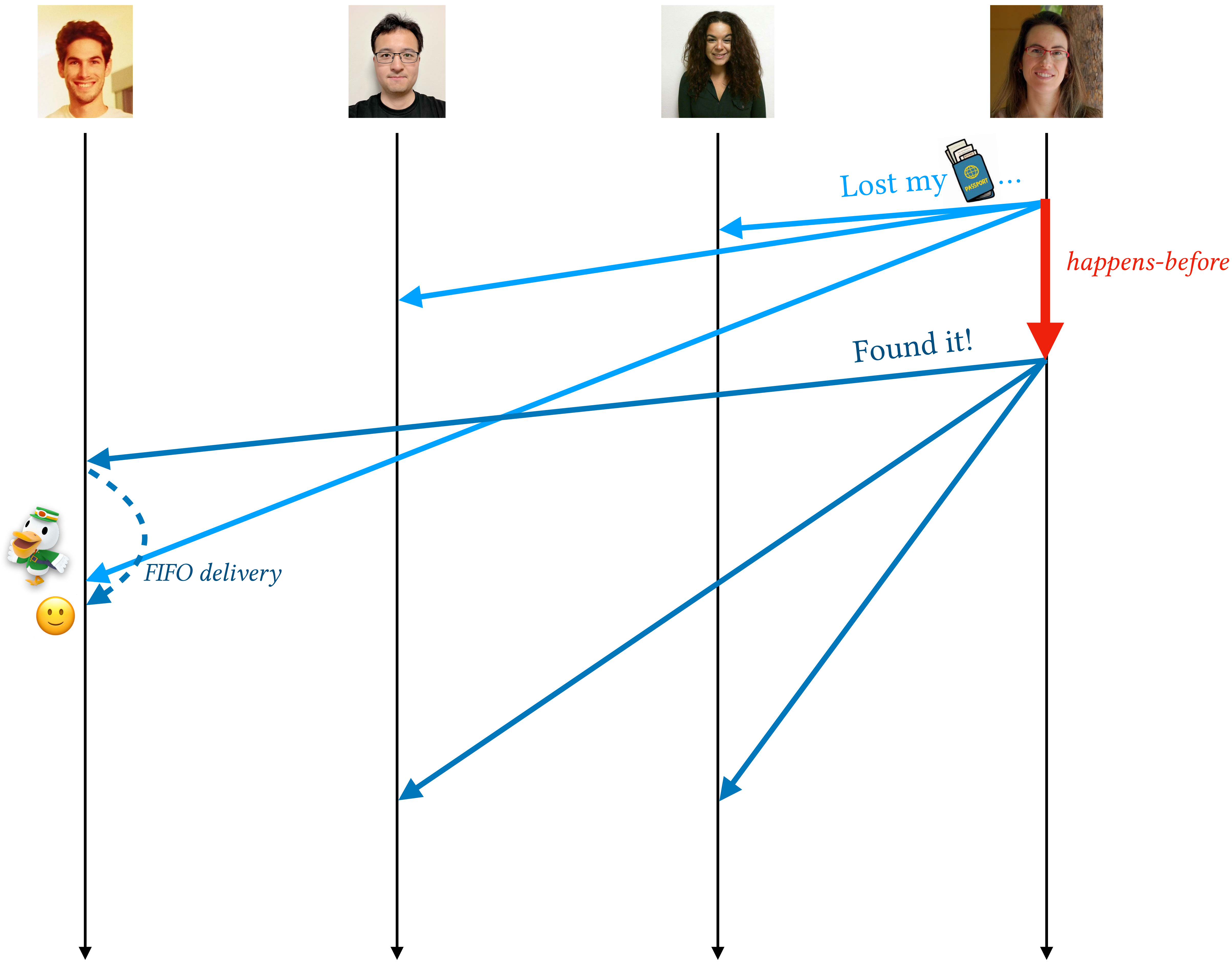

Lost my …

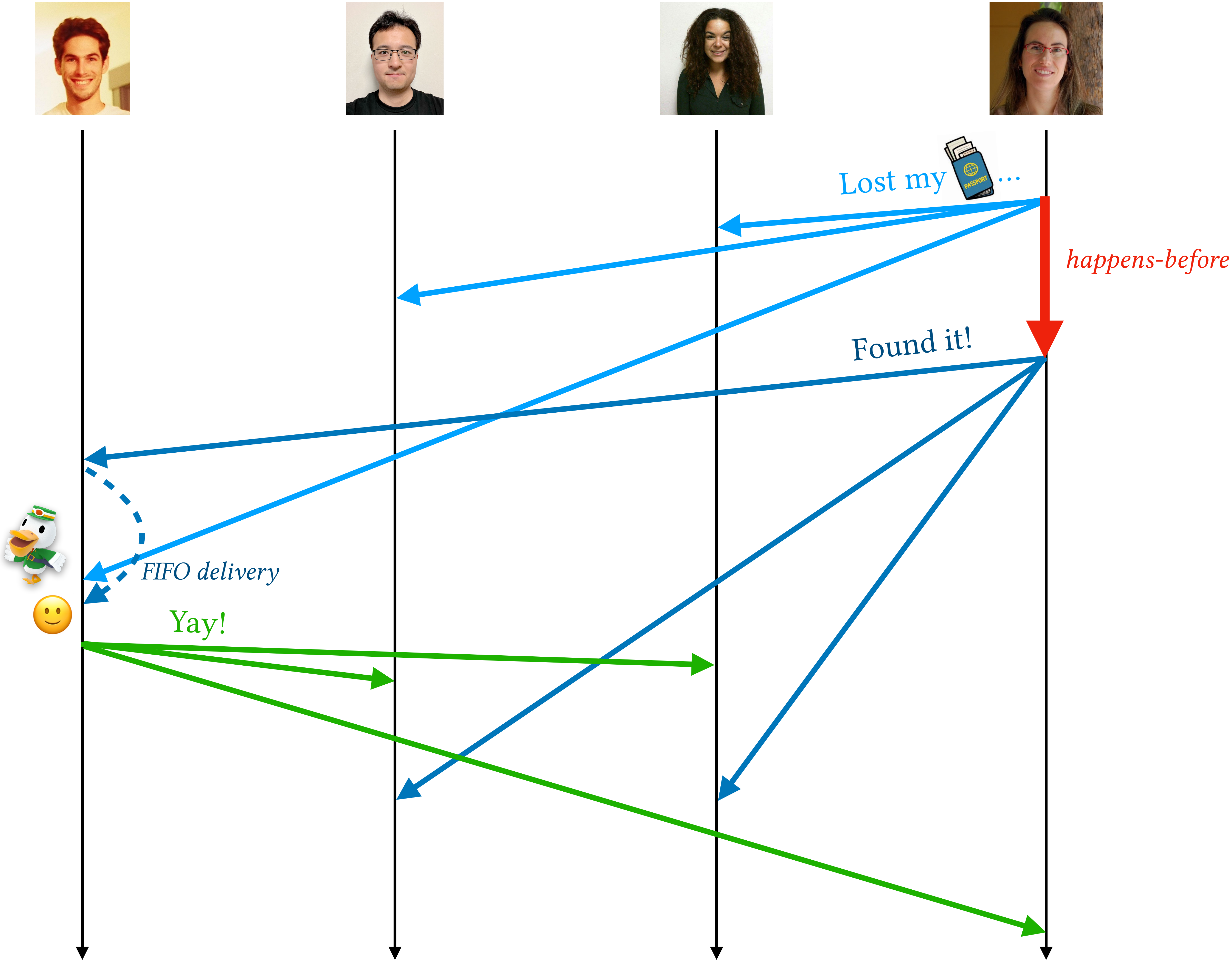

Lost my …
PASSPORT
Found it!

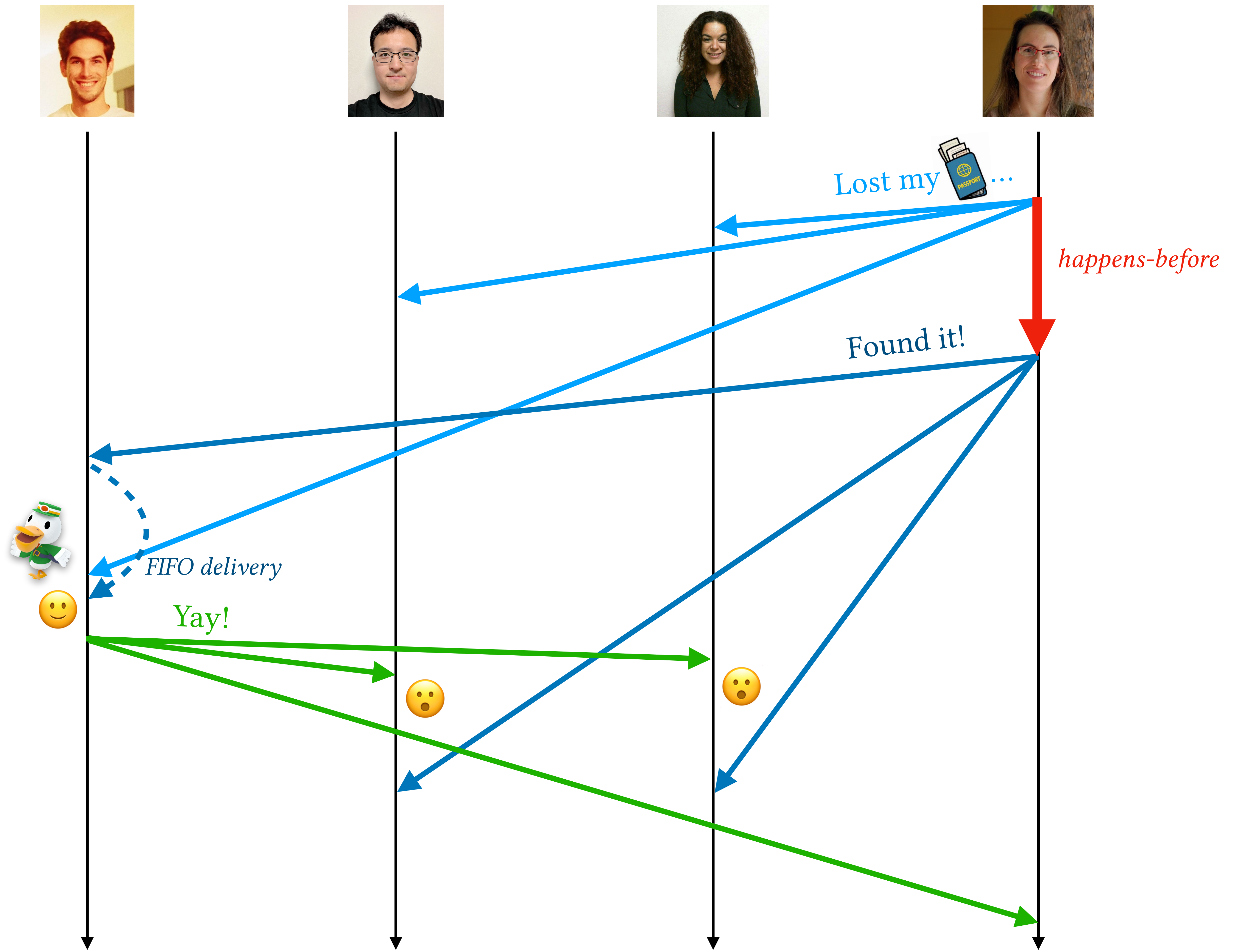

Lost my …
Found it!

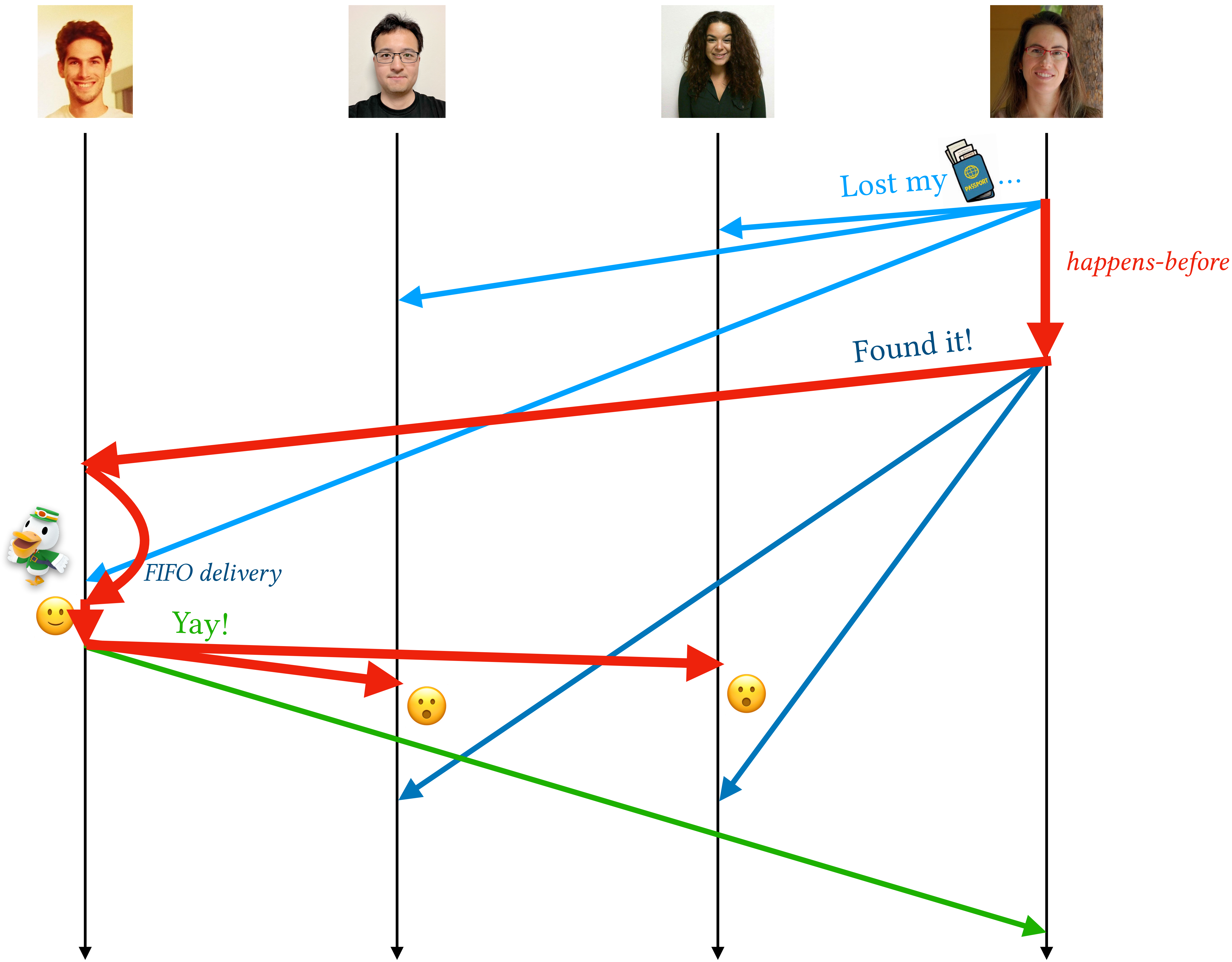

Lost my …
happens-before
Found it!

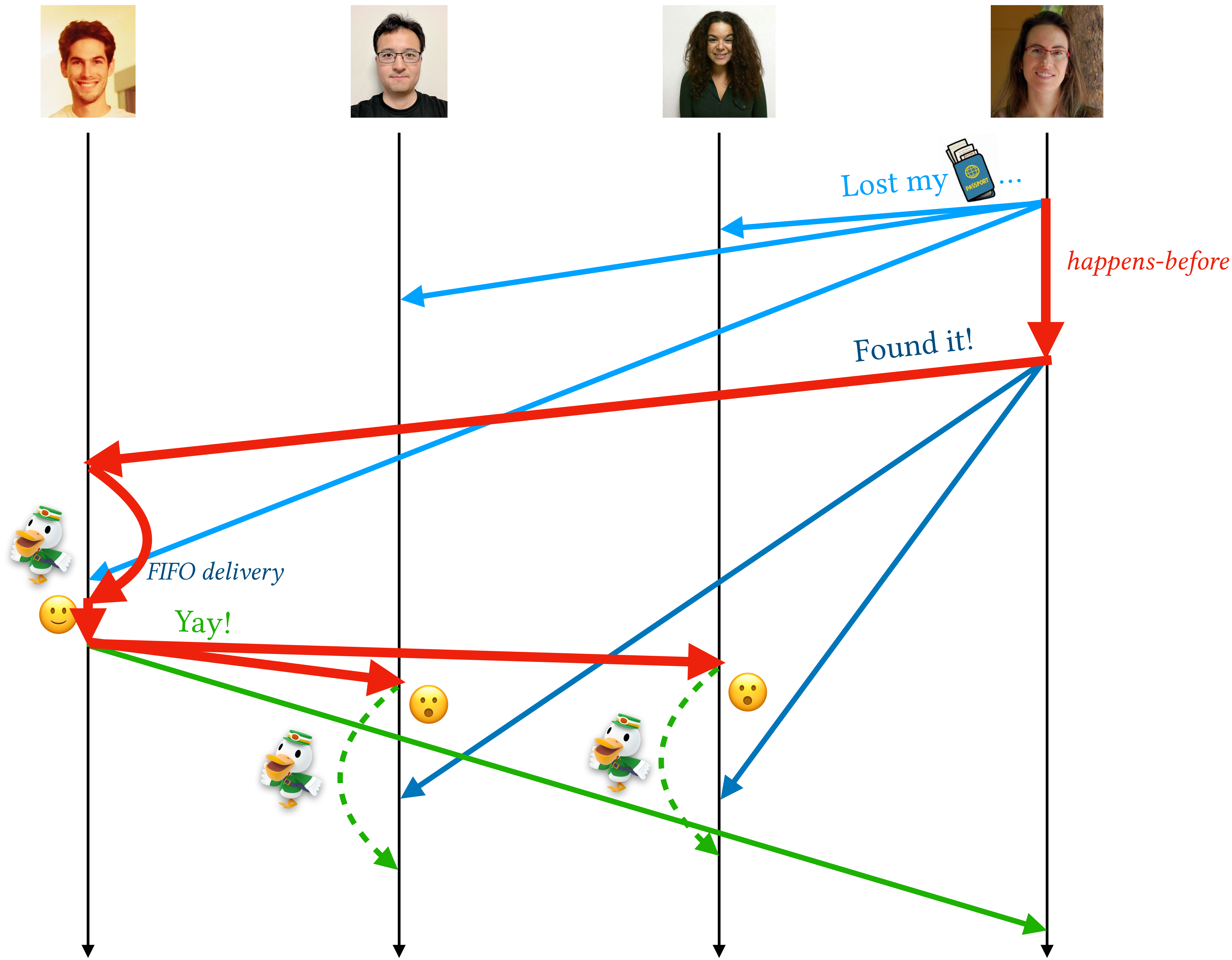

Lost my …
happens-before
Found it!

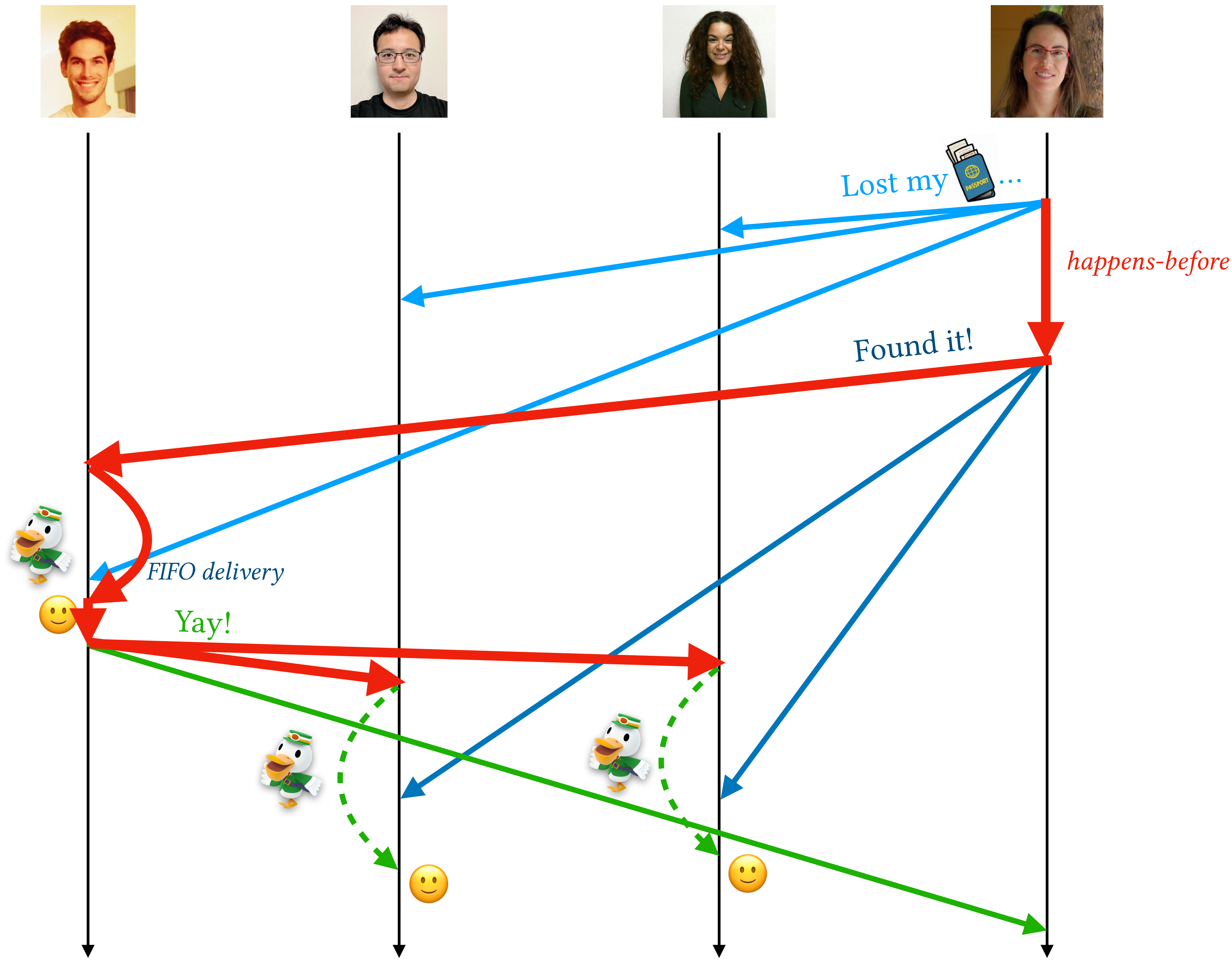

Lost my …
happens-before
Found it!
FIFO delivery

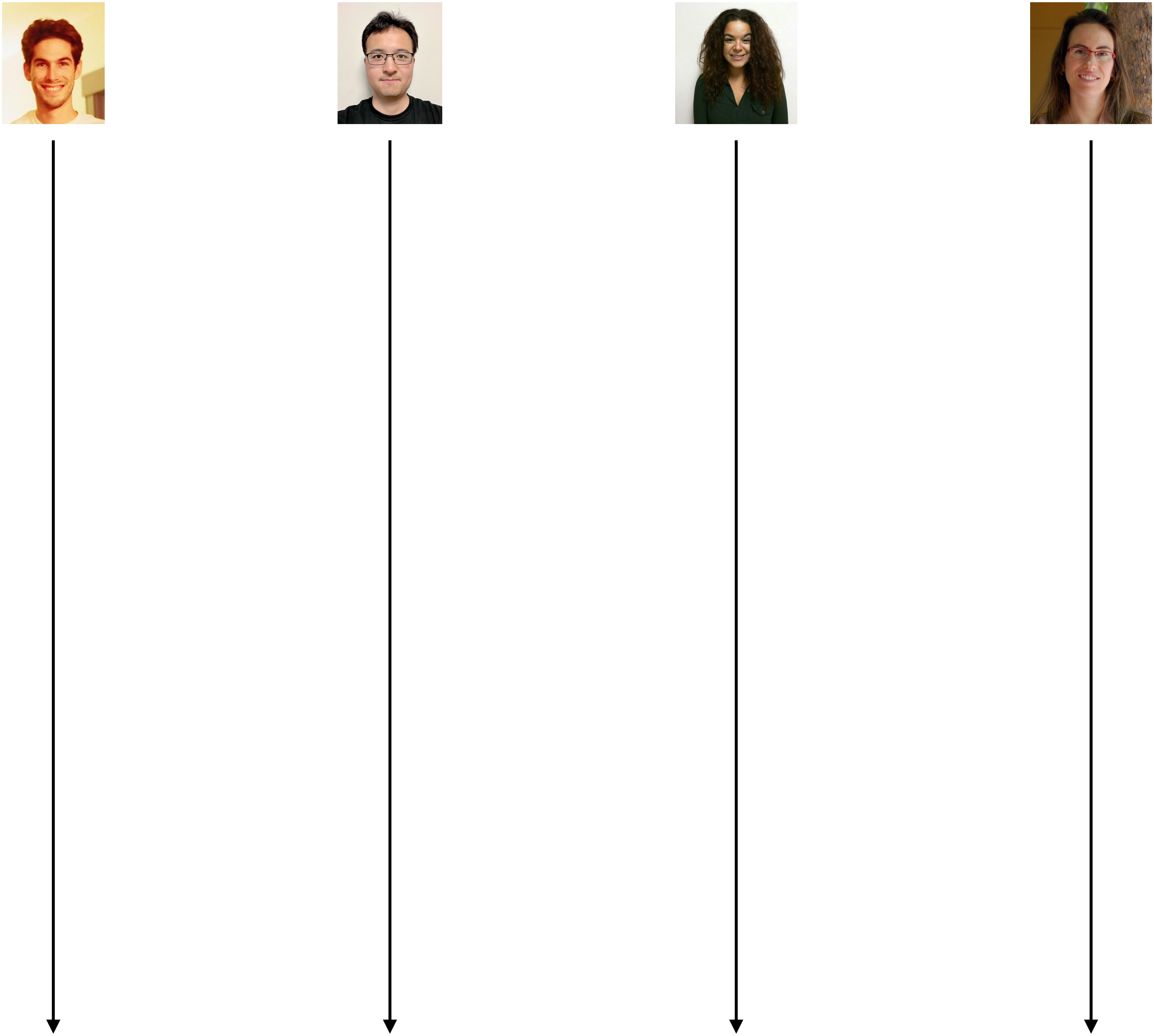

Lost my …
happens-before
Found it!
FIFO delivery
Yay!


Lost my …
happens-before
Found it!
FIFO delivery
Yay!

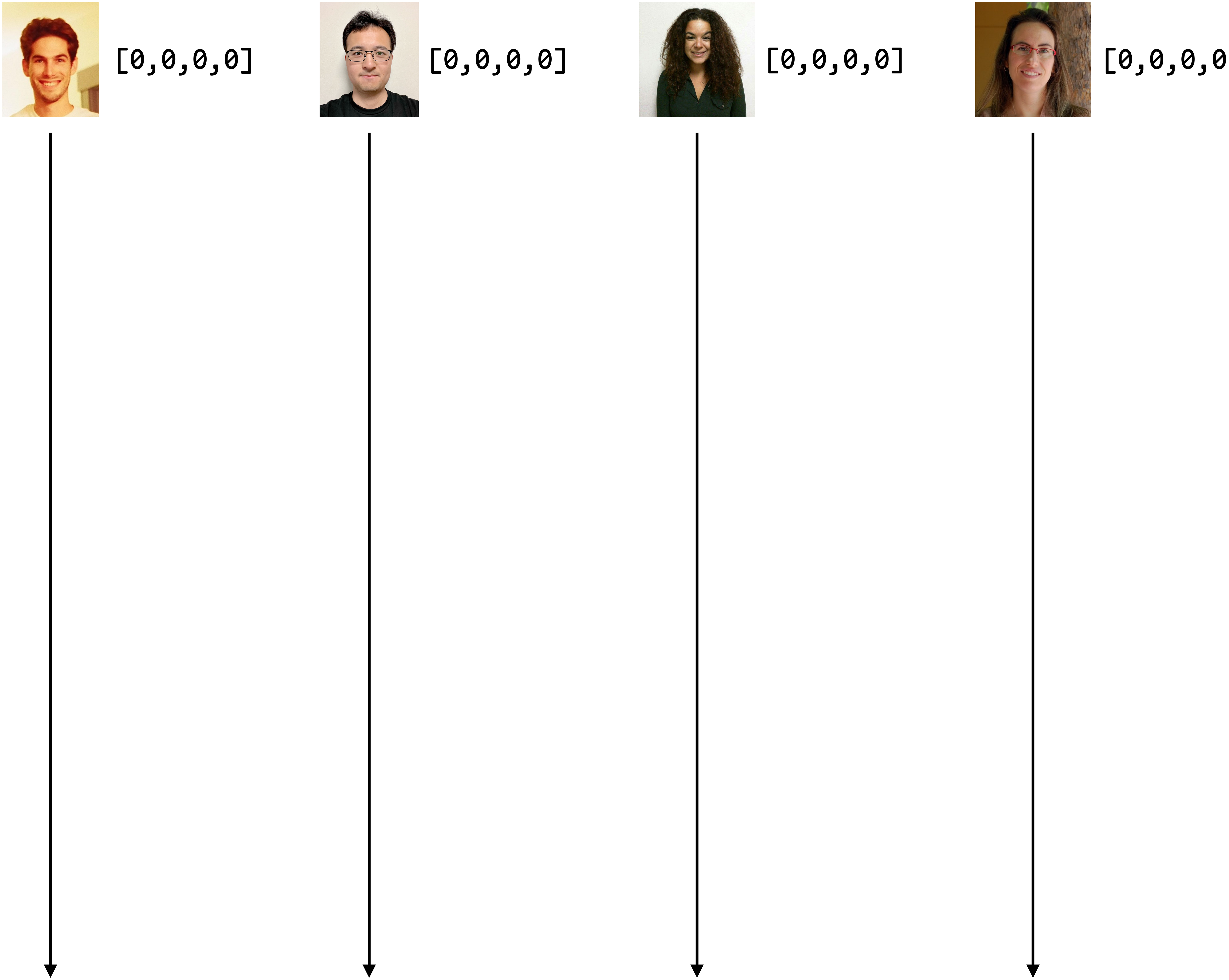

Lost my …
happens-before
Found it!
FIFO delivery
Yay!


Lost my …
happens-before
Found it!
FIFO delivery
Yay!

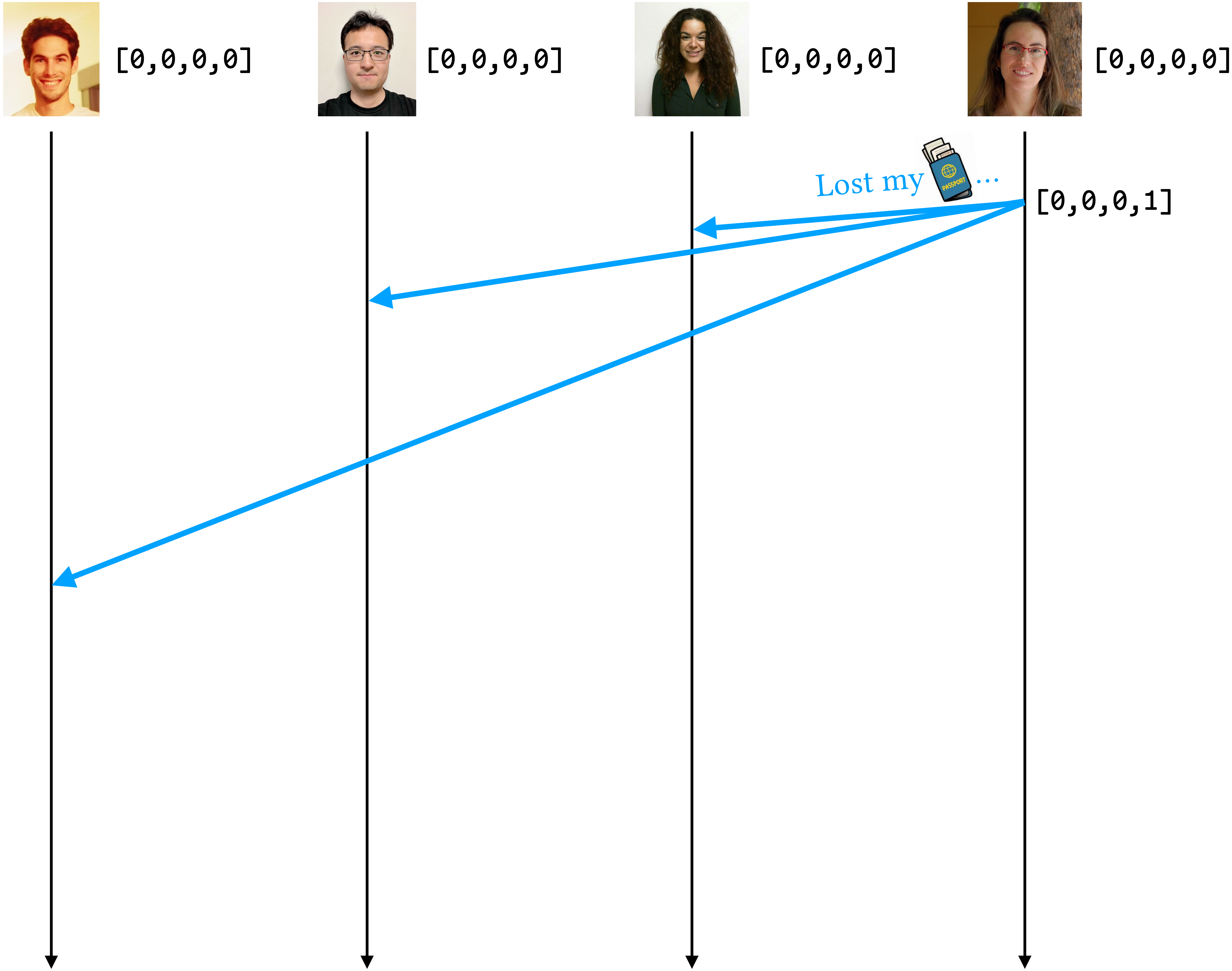

Lost my …
happens-before
Found it!
FIFO delivery
Yay!



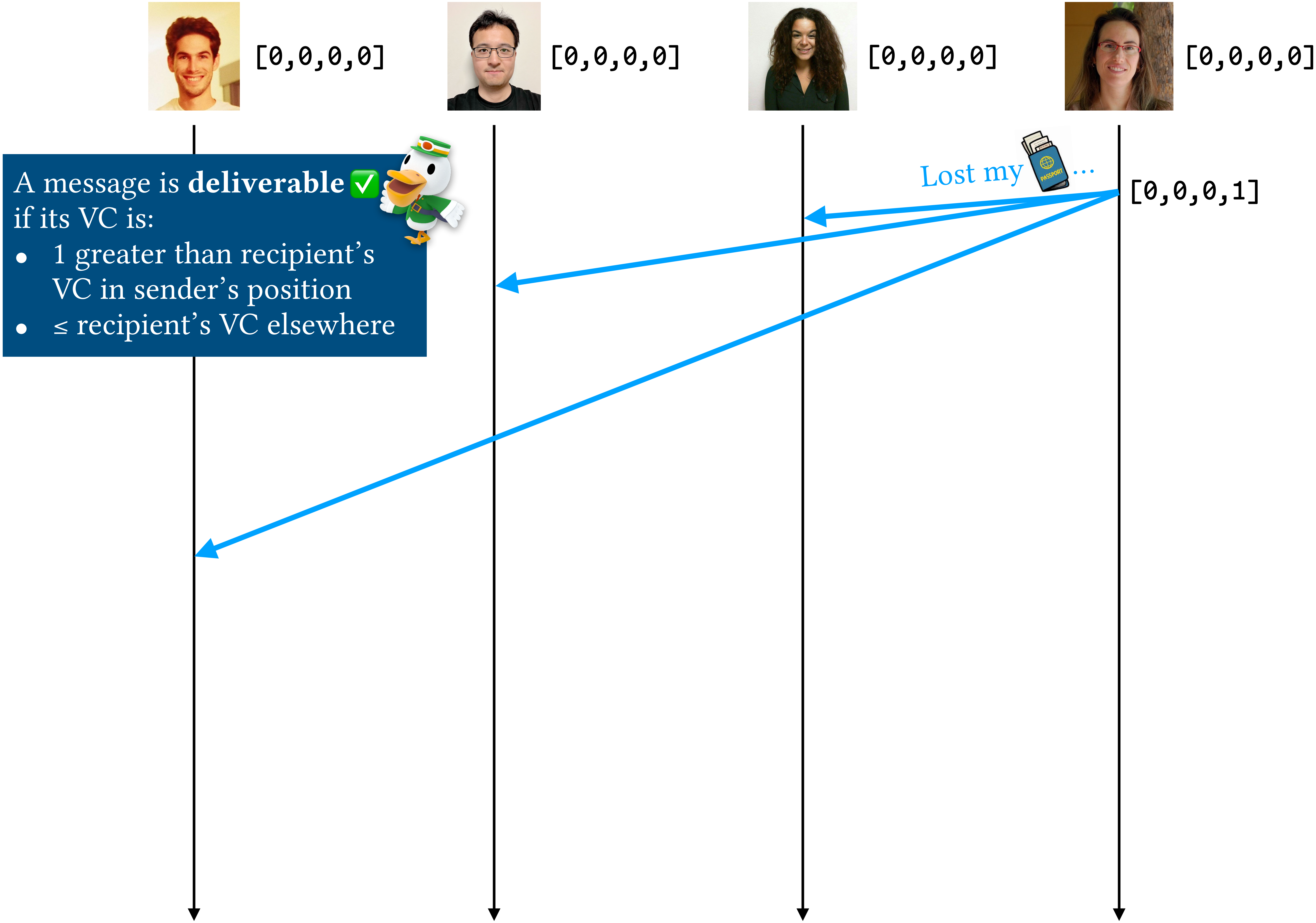
Causal broadcast with vector clocks [Birman et al., 1991]



Causal broadcast with vector clocks [Birman et al., 1991]



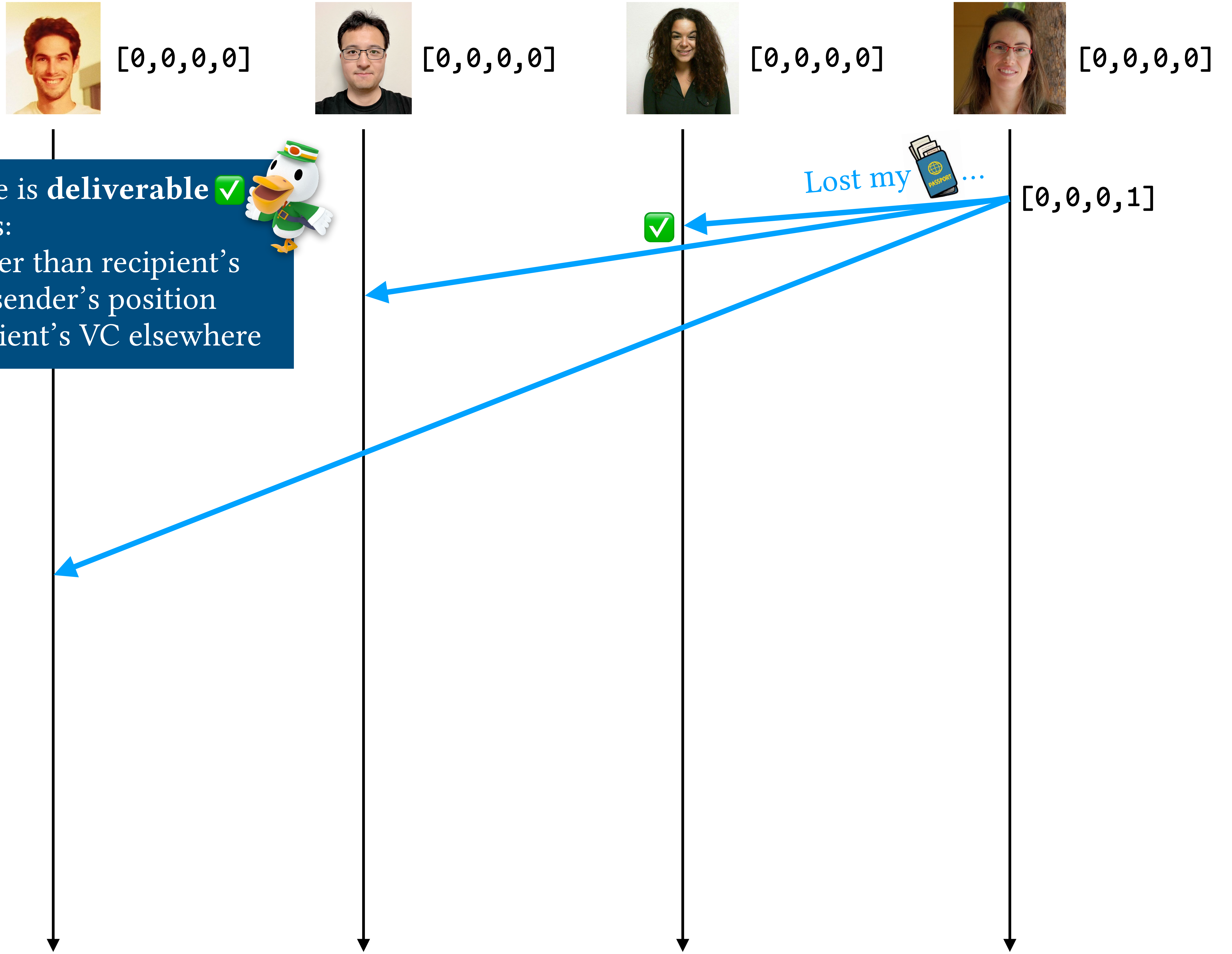
Causal broadcast with vector clocks [Birman et al., 1991]



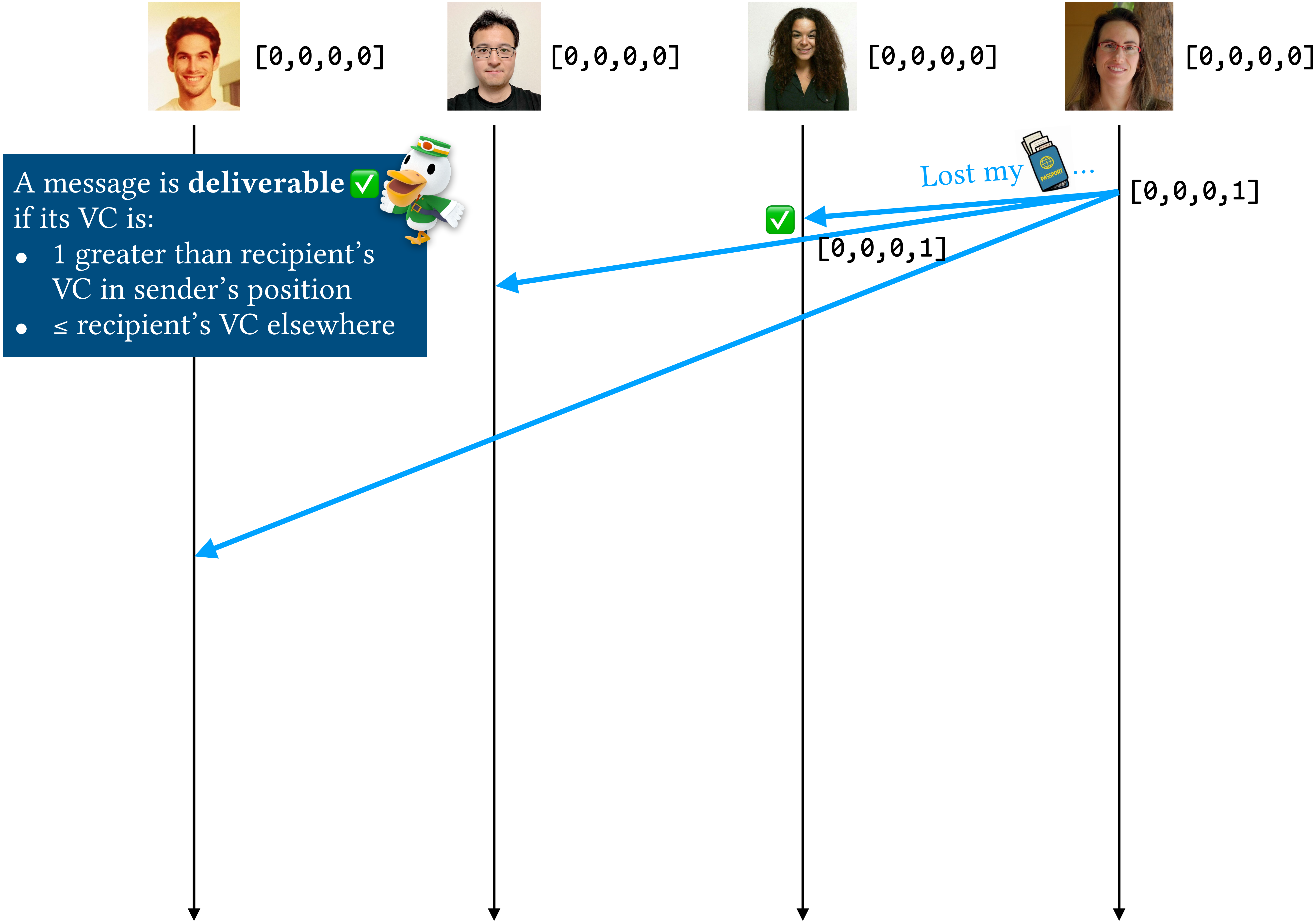
Causal broadcast with vector clocks [Birman et al., 1991]

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]



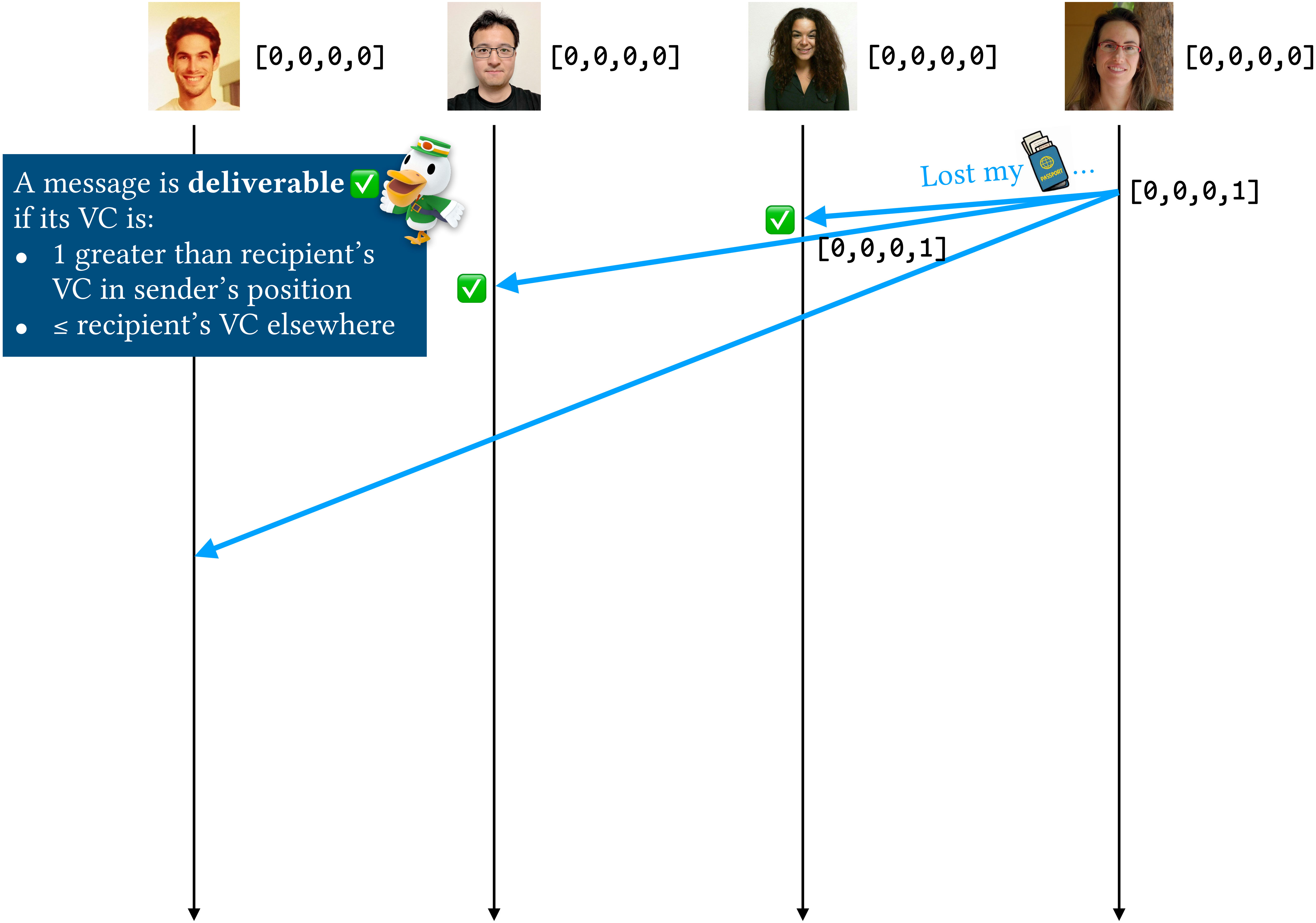
A message is **deliverable** ✅ if its VC is:

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

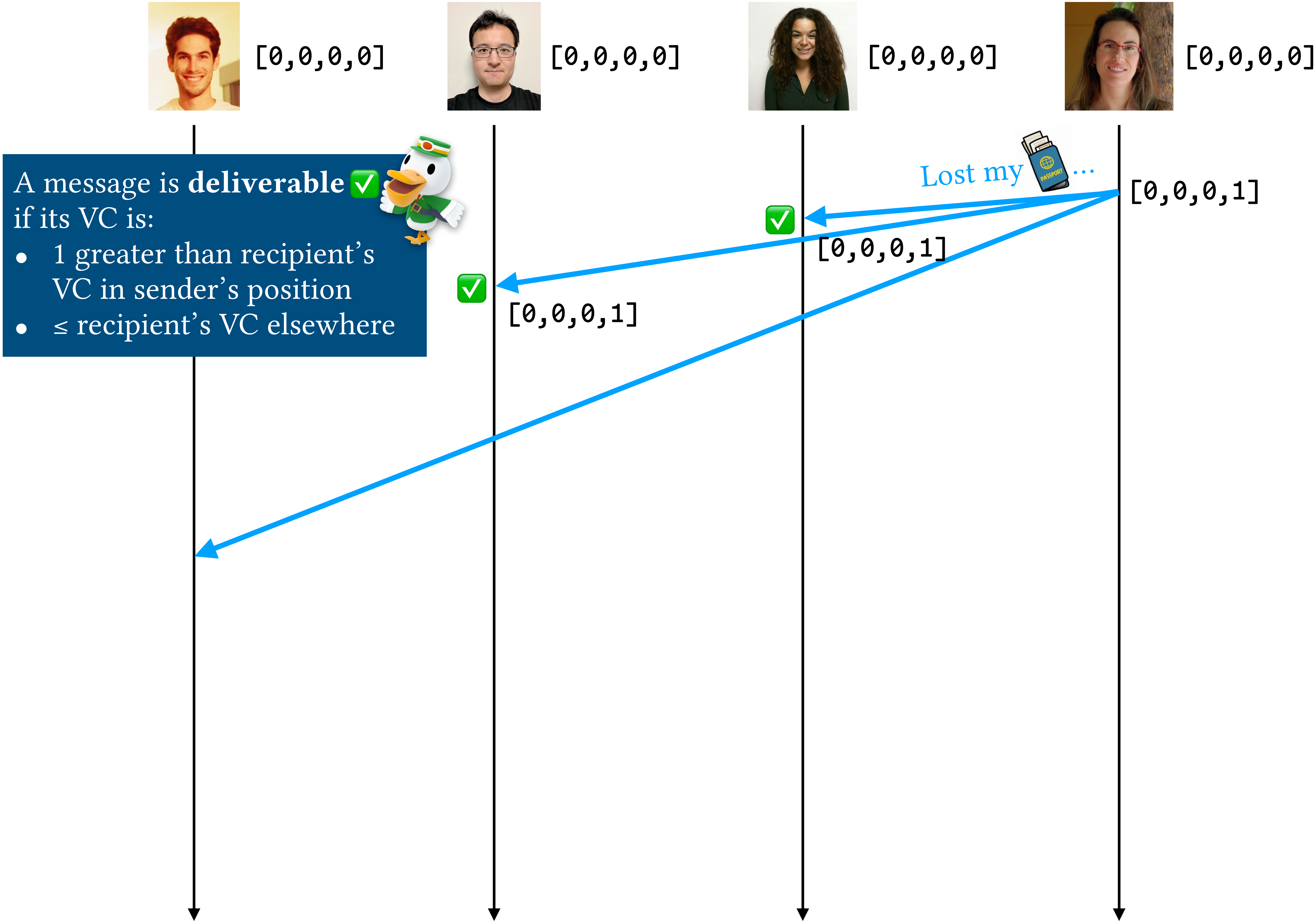
Causal broadcast with vector clocks [Birman et al., 1991]

A message is **deliverable** ✅ if its VC is:

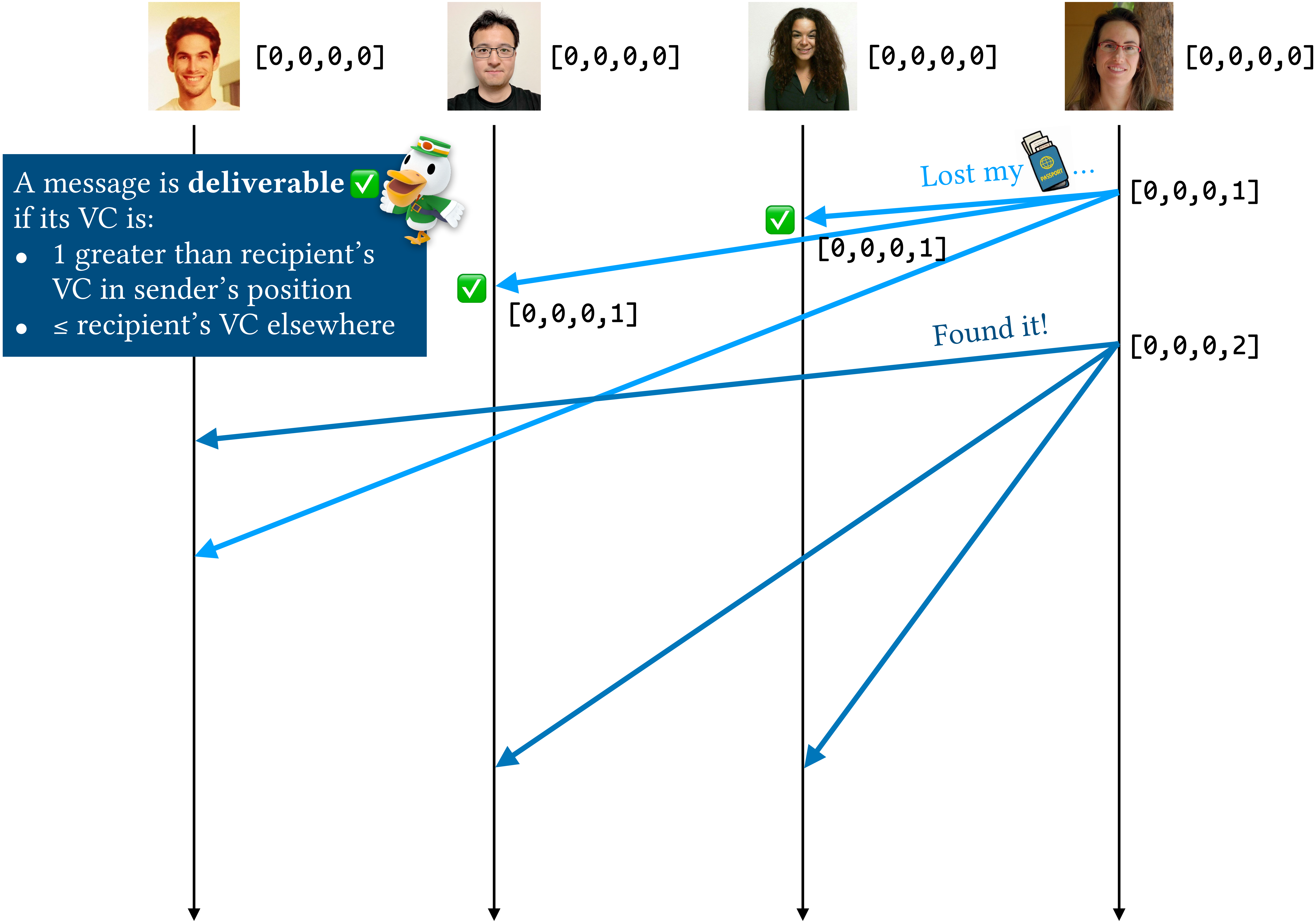
- 1 greater than recipient's VC in sender's position
- ≤ recipient's VC elsewhere



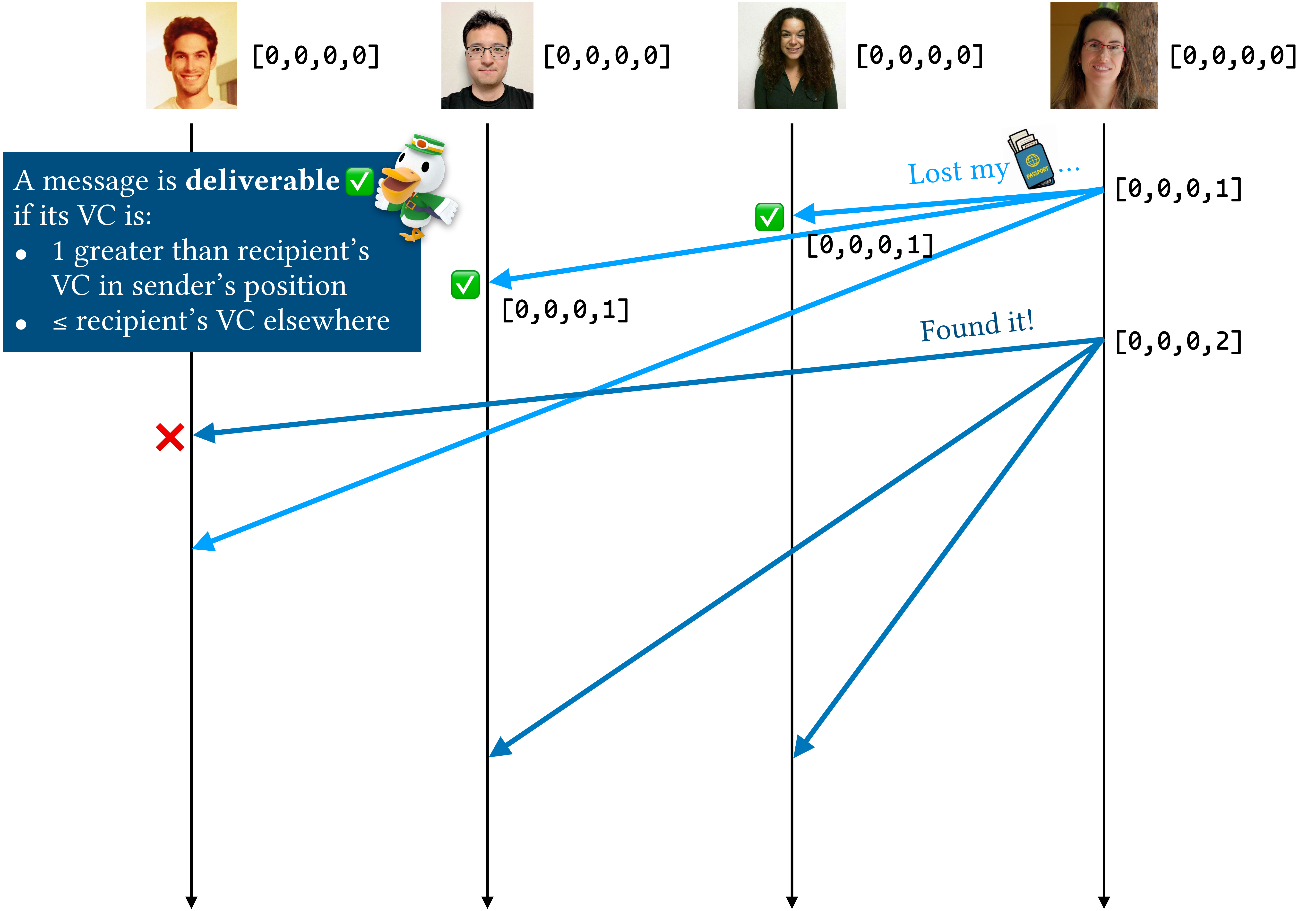
Causal broadcast with vector clocks [Birman et al., 1991]



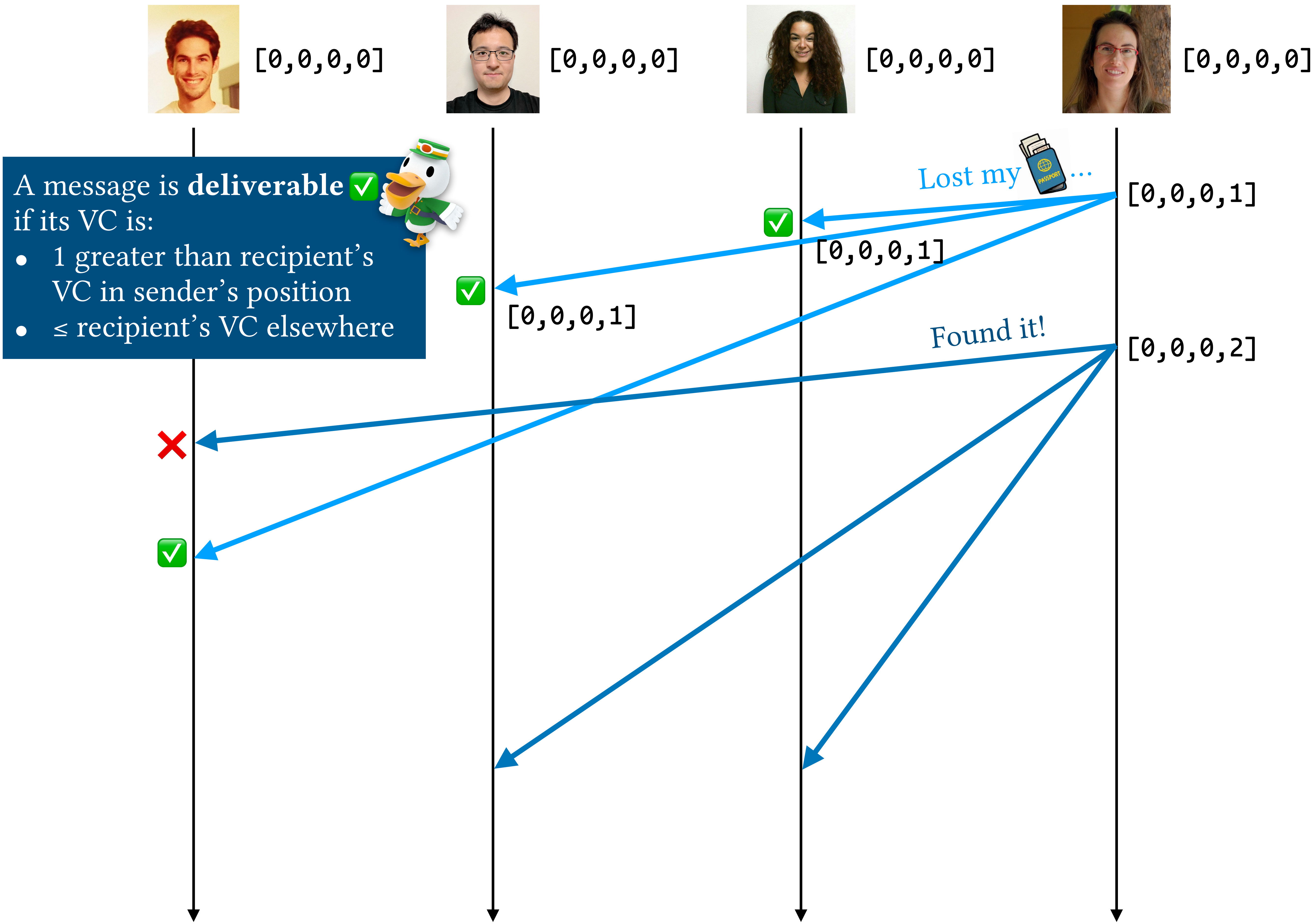
Causal broadcast with vector clocks [Birman et al., 1991]



Causal broadcast with vector clocks [Birman et al., 1991]



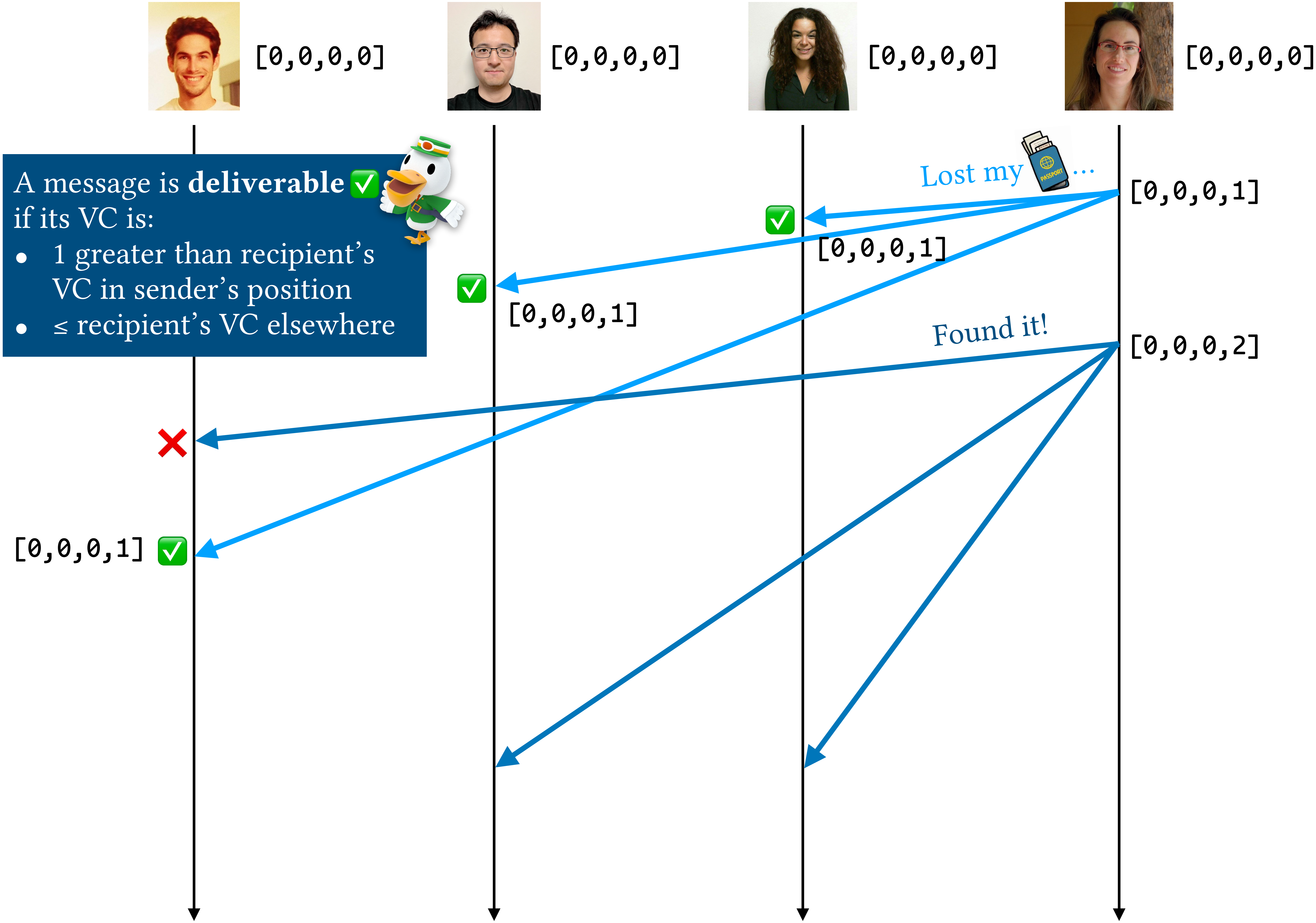
Causal broadcast with vector clocks [Birman et al., 1991]



Causal broadcast with vector clocks [Birman et al., 1991]

A message is **deliverable** ✅ if its VC is:

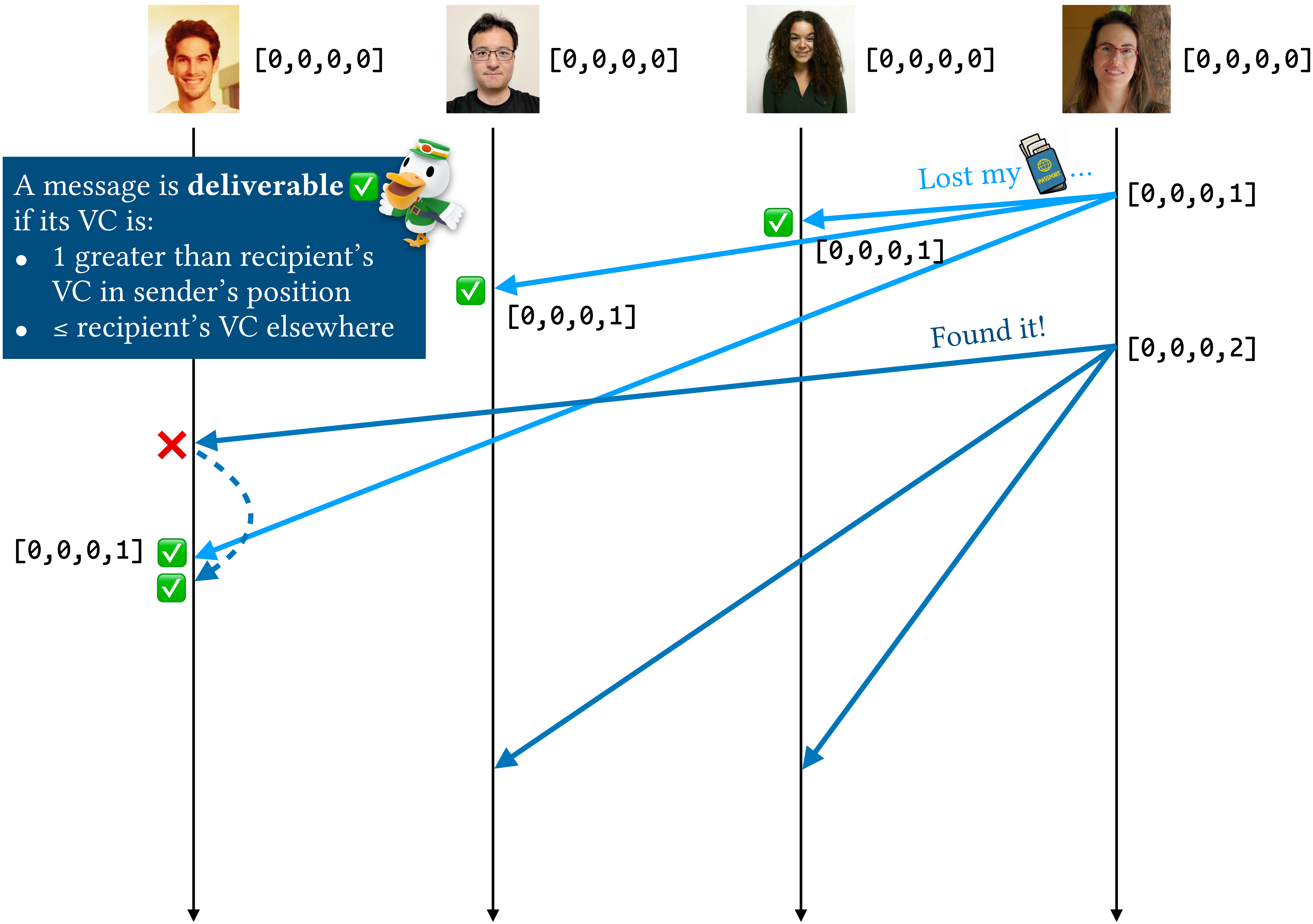
- 1 greater than recipient's VC in sender's position
- ≤ recipient's VC elsewhere



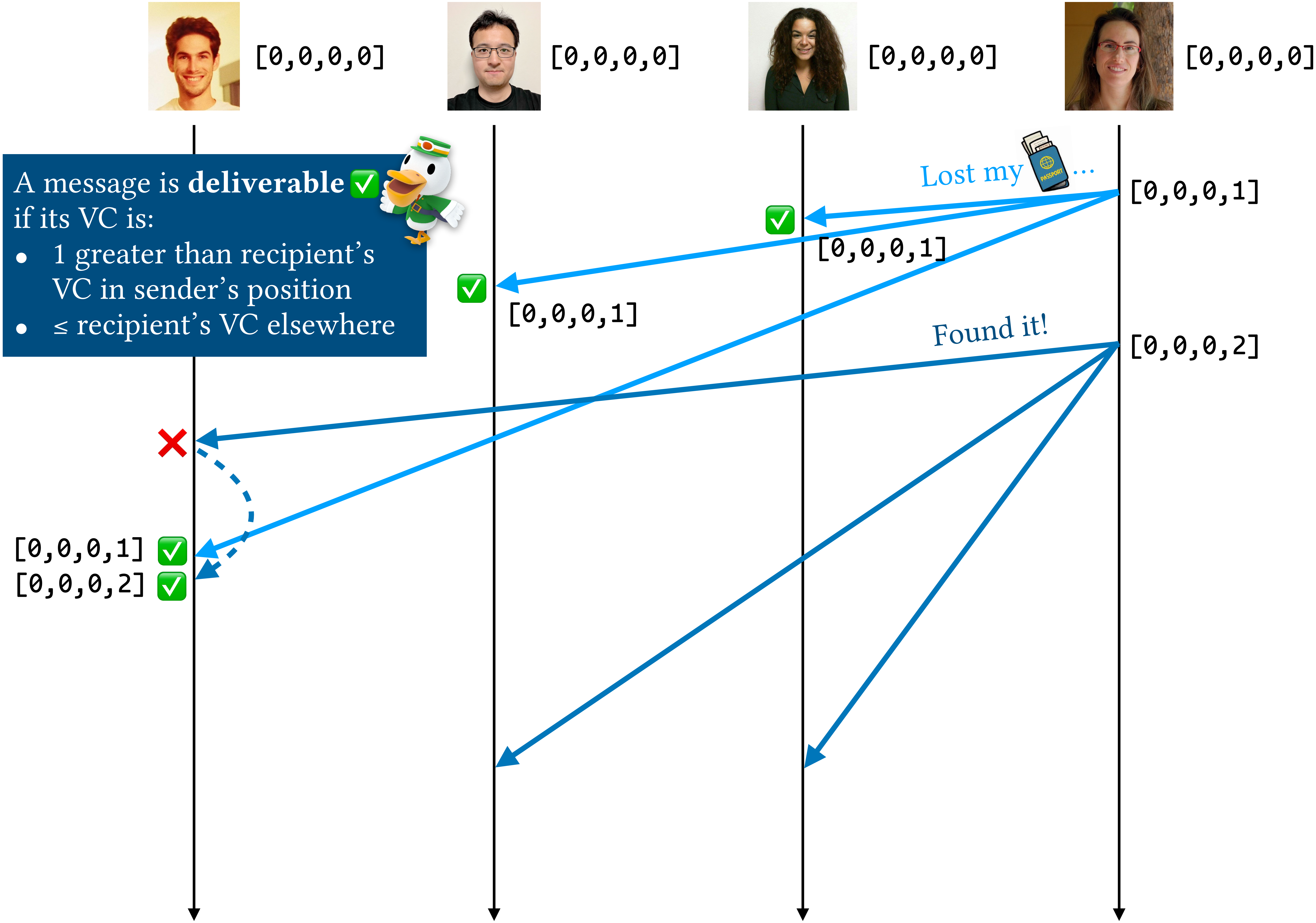
Causal broadcast with vector clocks [Birman et al., 1991]

A message is **deliverable** ✅ if its VC is:

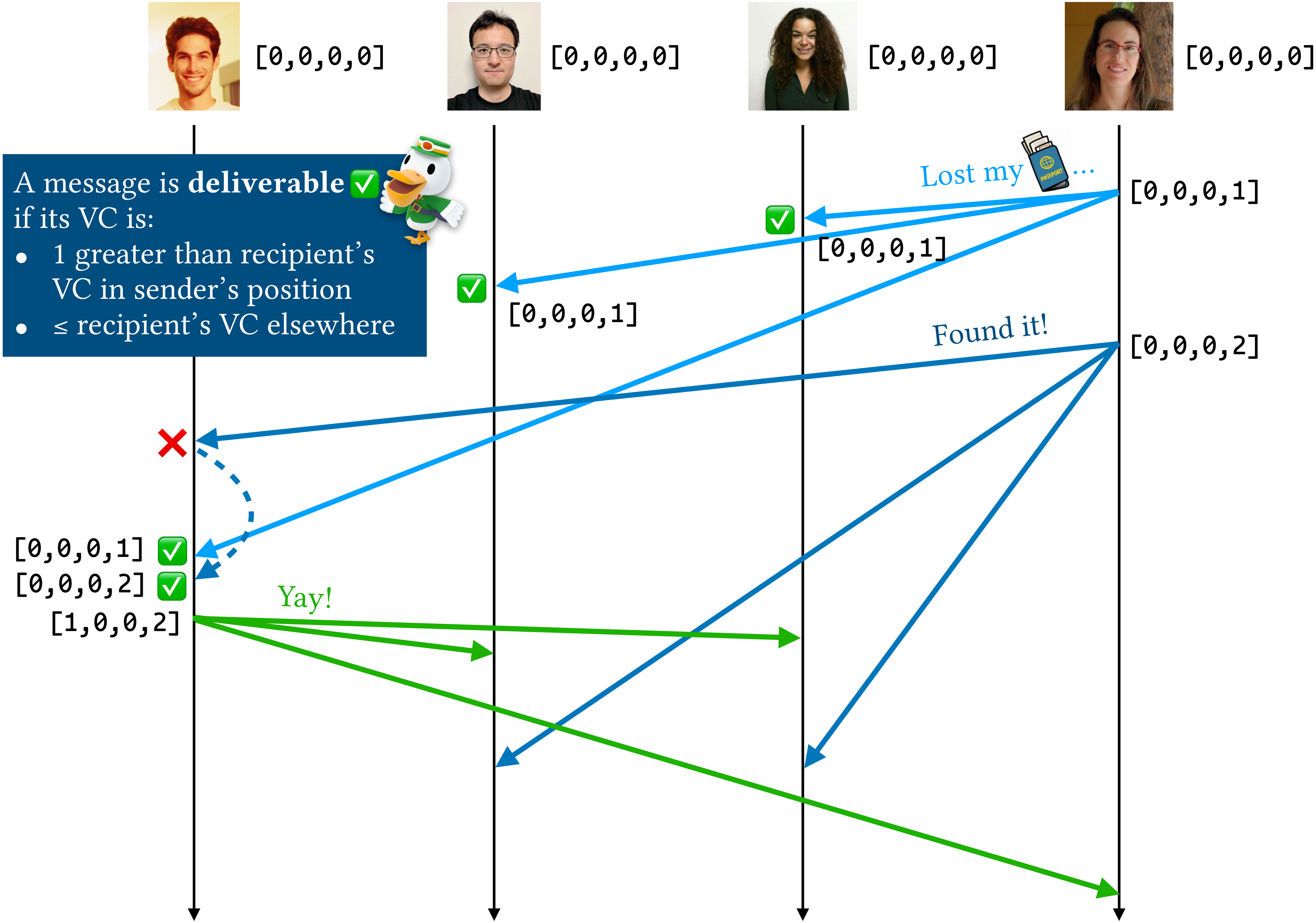
- 1 greater than recipient's VC in sender's position
- ≤ recipient's VC elsewhere



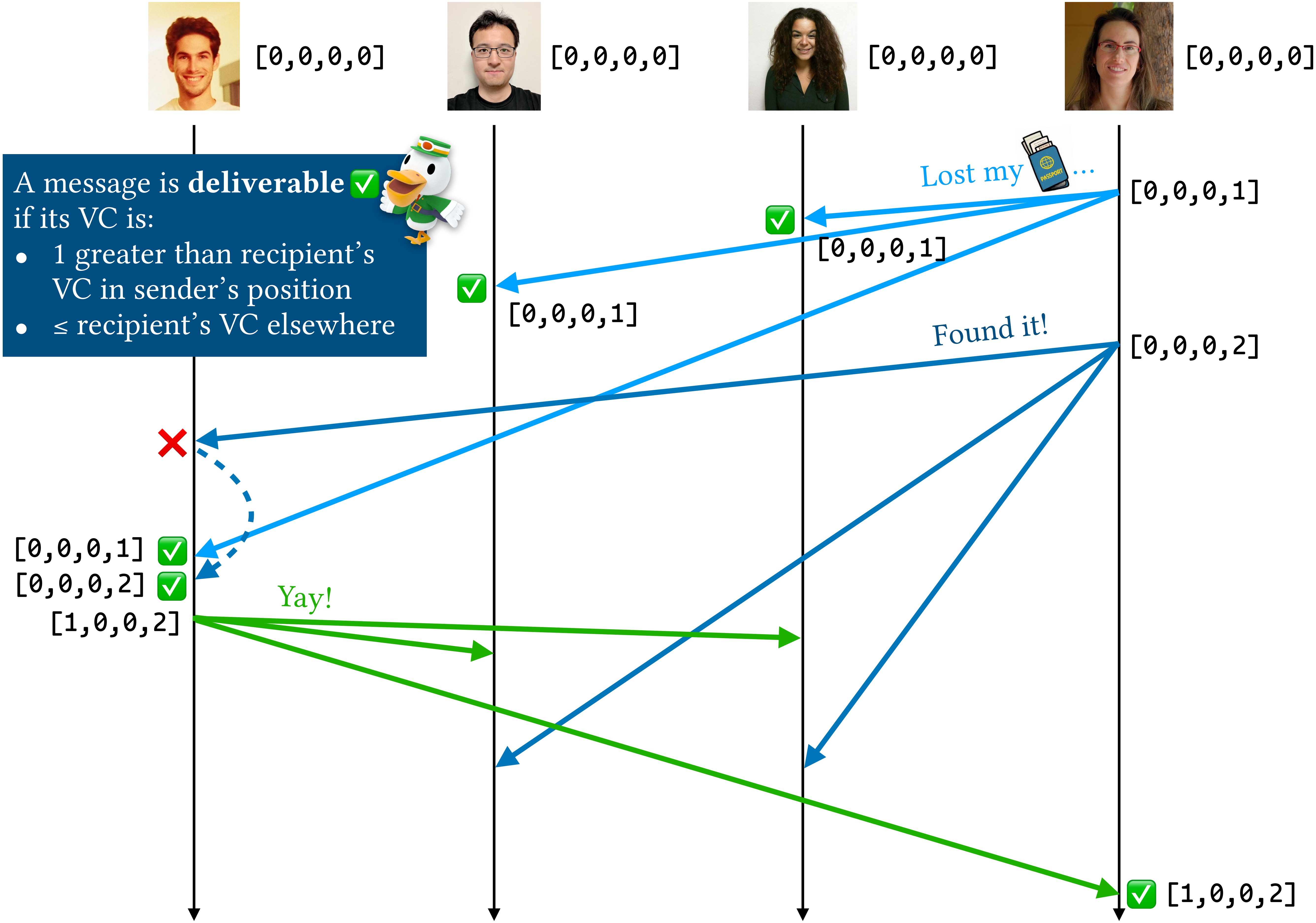
Causal broadcast with vector clocks [Birman et al., 1991]



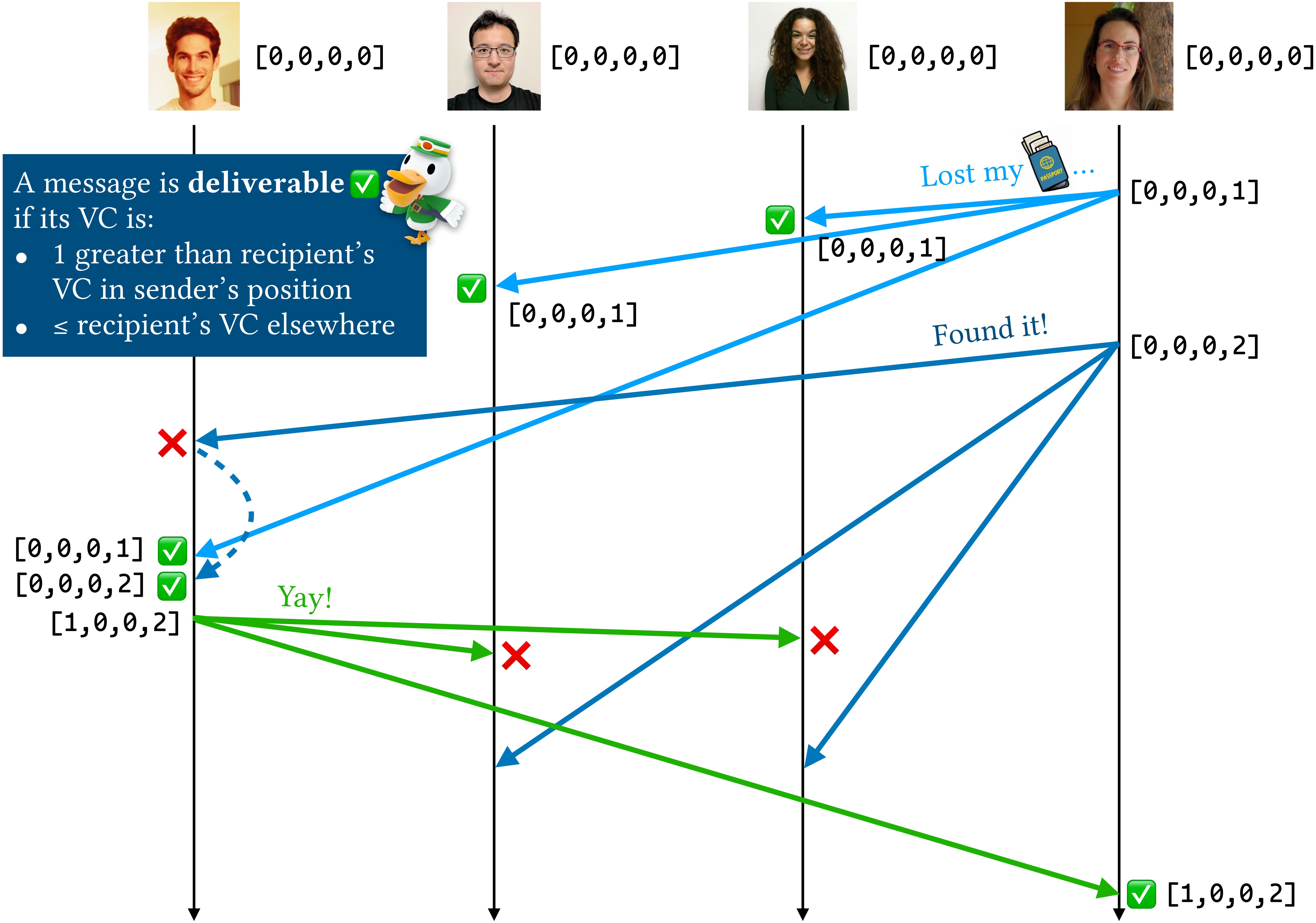
Causal broadcast with vector clocks [Birman et al., 1991]



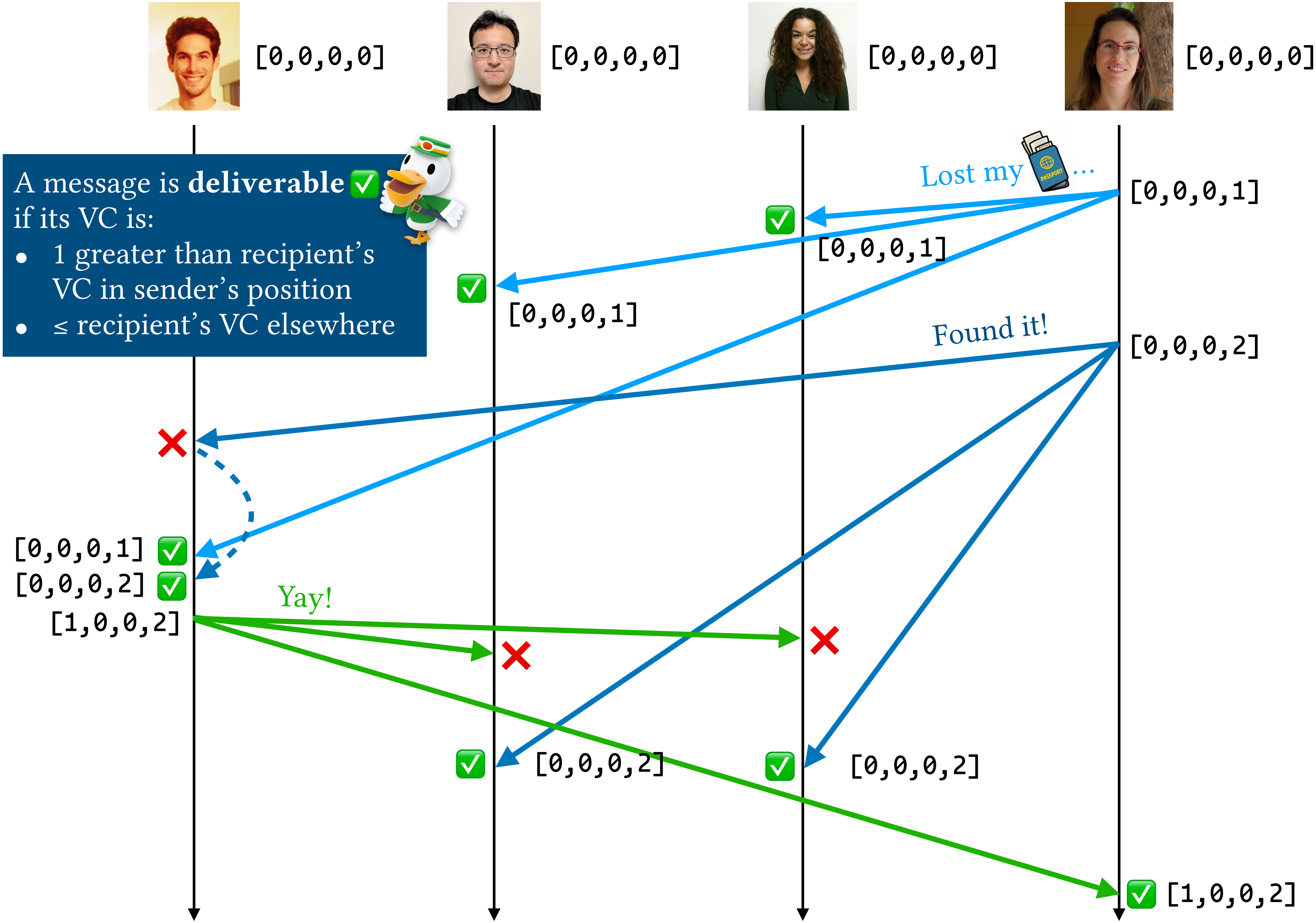
Causal broadcast with vector clocks [Birman et al., 1991]



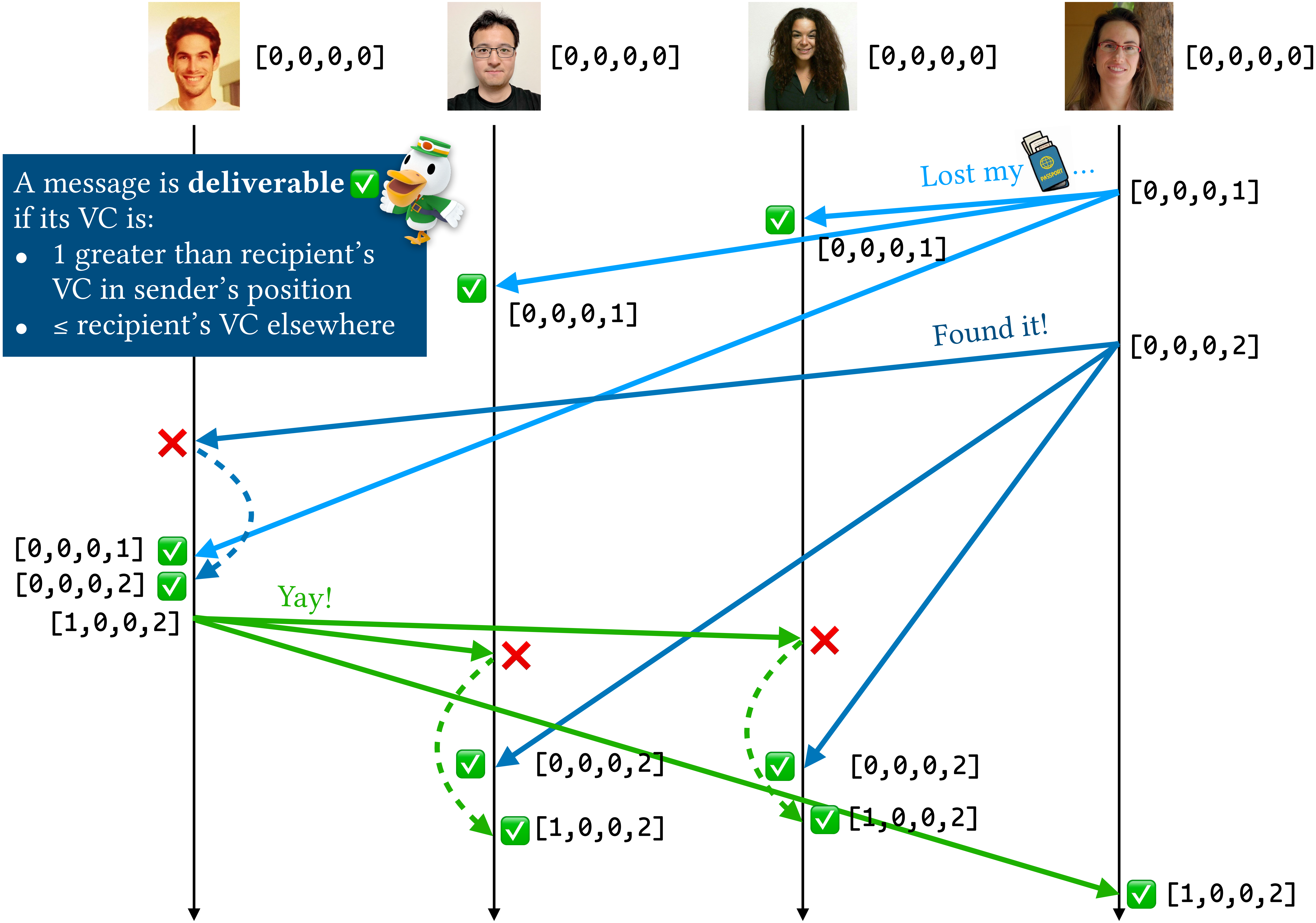
Causal broadcast with vector clocks [Birman et al., 1991]



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]

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 | v >= 0 }
```

Refinement types

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

Refinement types

```
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

```haskell
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

```haskell
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]`

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

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 >= 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 >= 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
```

```
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 VCsized 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 }
```

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
```

```
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 VCsized 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 VCsized 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 VCsized 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
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
```

```
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*

```
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
```

```
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*

```
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

```
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*

```
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
```

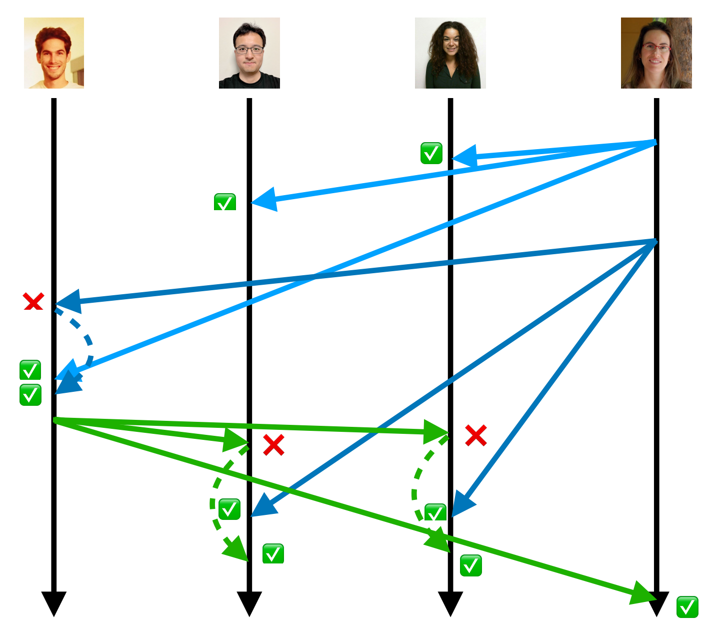
Refinement *reflection*

(Local) causal delivery as a refinement type

verification code

’s process history (`pHist` **):**

```
[(Deliver   “Lost my   ”),
 (Deliver   “Found it!”),
 (Broadcast “Yay!”),
 …]
```



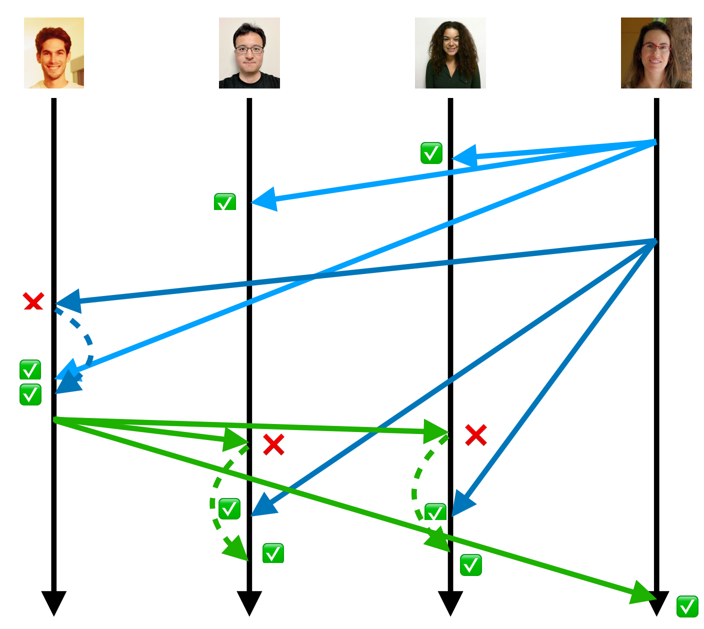
(Local) causal delivery as a refinement type

verification code

```
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` **):**

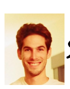
```
[(Deliver   “Lost my   ”),
 (Deliver   “Found it!”),
 (Broadcast “Yay!”),
 …]
```

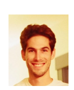
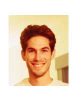


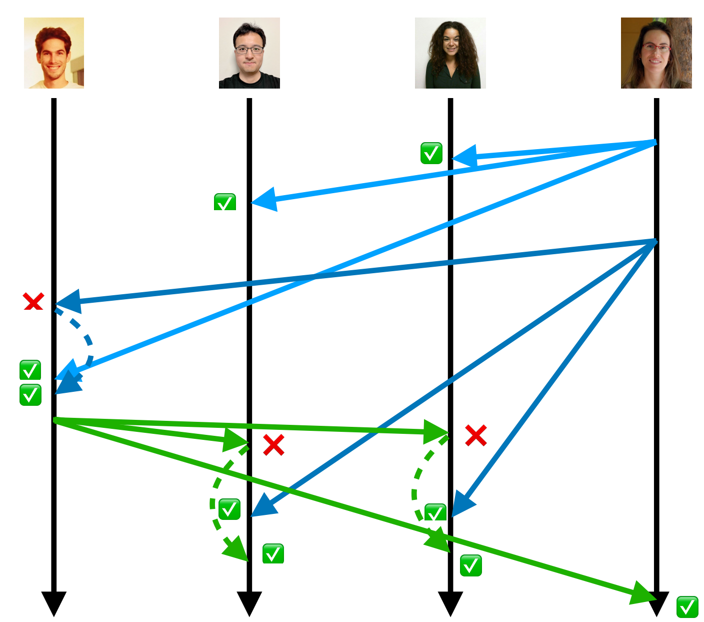
(Local) causal delivery as a refinement type

verification code

```
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 **):**

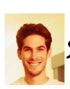
[(Deliver  “Lost my ”),
(Deliver  “Found it!”),
(Broadcast “Yay!”),
…]

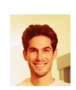
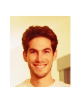


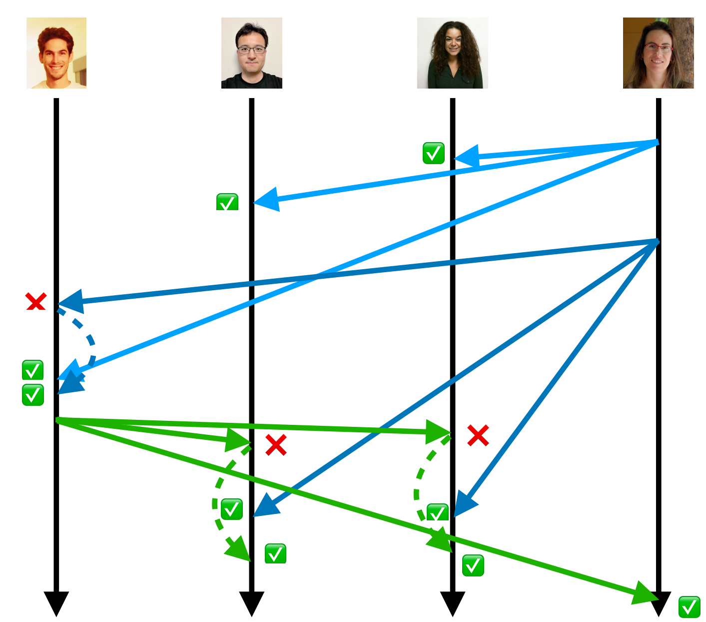
(Local) causal delivery as a refinement type

verification code

```
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` **):**

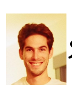
[(Deliver  "Lost my "),
 (Deliver  "Found it!"),
 (Broadcast "Yay!"),
 …]

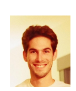
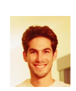


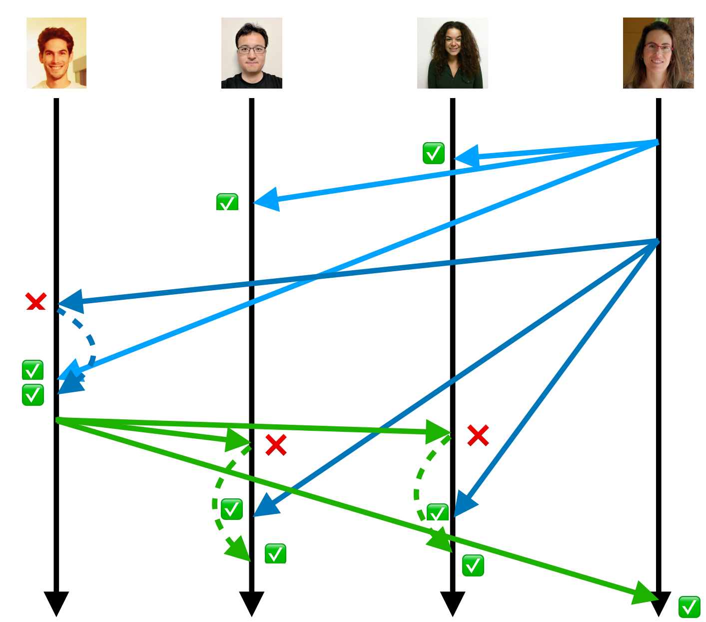
(Local) causal delivery as a refinement type

verification code

```
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` **):**

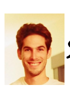
[(Deliver  "Lost my "),
 (Deliver  "Found it!"),
 (Broadcast "Yay!"),
 …]

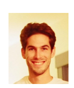
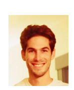


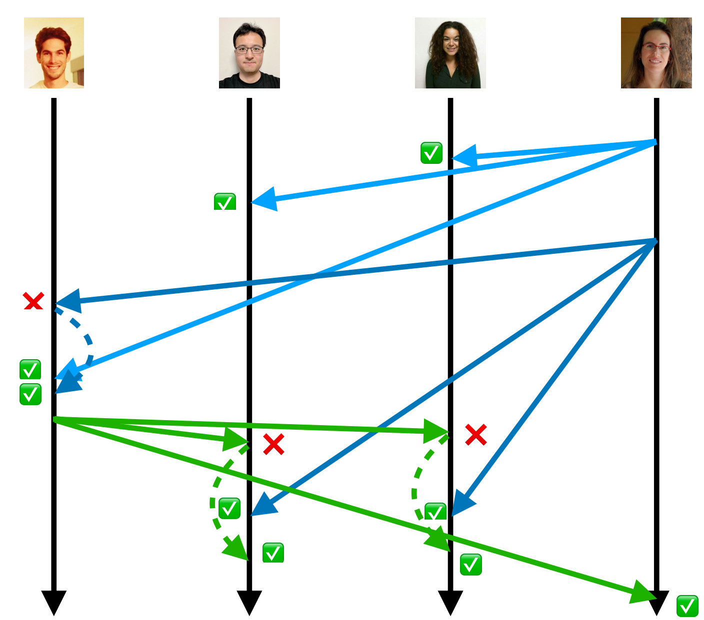
(Local) causal delivery as a refinement type

verification code

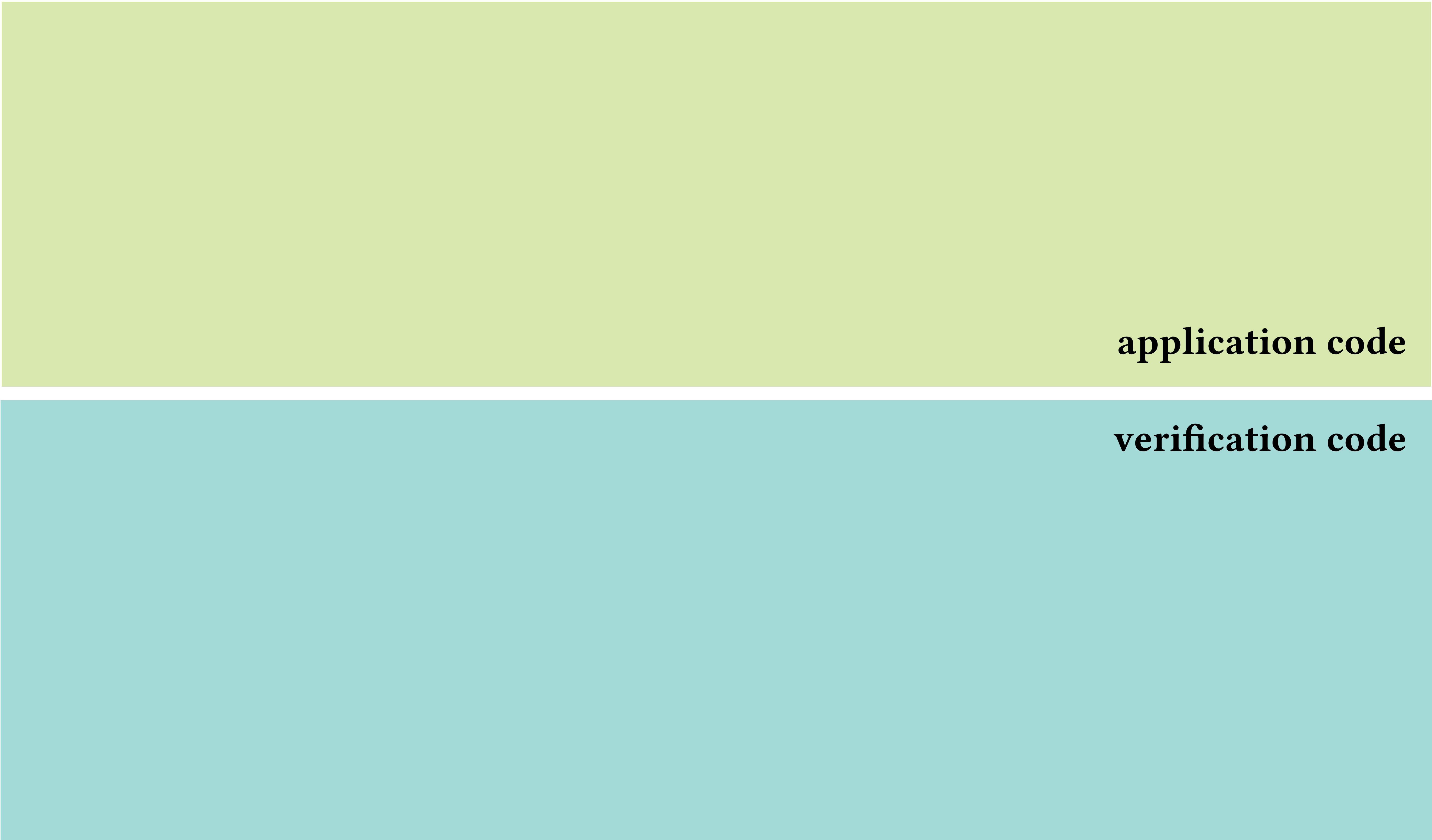
```
type LocalCausalDelivery P =
    =  { m1 : Message | elem (Deliver (pID P) m1) (pHist P) }
    -> { m2 : Message | elem (Deliver (pID P) m2) (pHist P)
                        && vcLess (mVC m1) (mVC m2) }
    -> { _: Proof | processOrder (pHist P) (Deliver (pID P) m1)
                           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

**’s process history (** `pHist` **):**

[(Deliver  “Lost my ”),
(Deliver  “Found it!”),
(Broadcast “Yay!”),
…]



Running the protocol preserves (local) causal delivery



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

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

verification 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
```

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

verification 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
```

Running the protocol preserves (local) causal delivery

```haskell
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

verification code

```haskell
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
```

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

verification 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
```

↓ = “relies on”

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

↓ = “relies on”

lcdStep

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

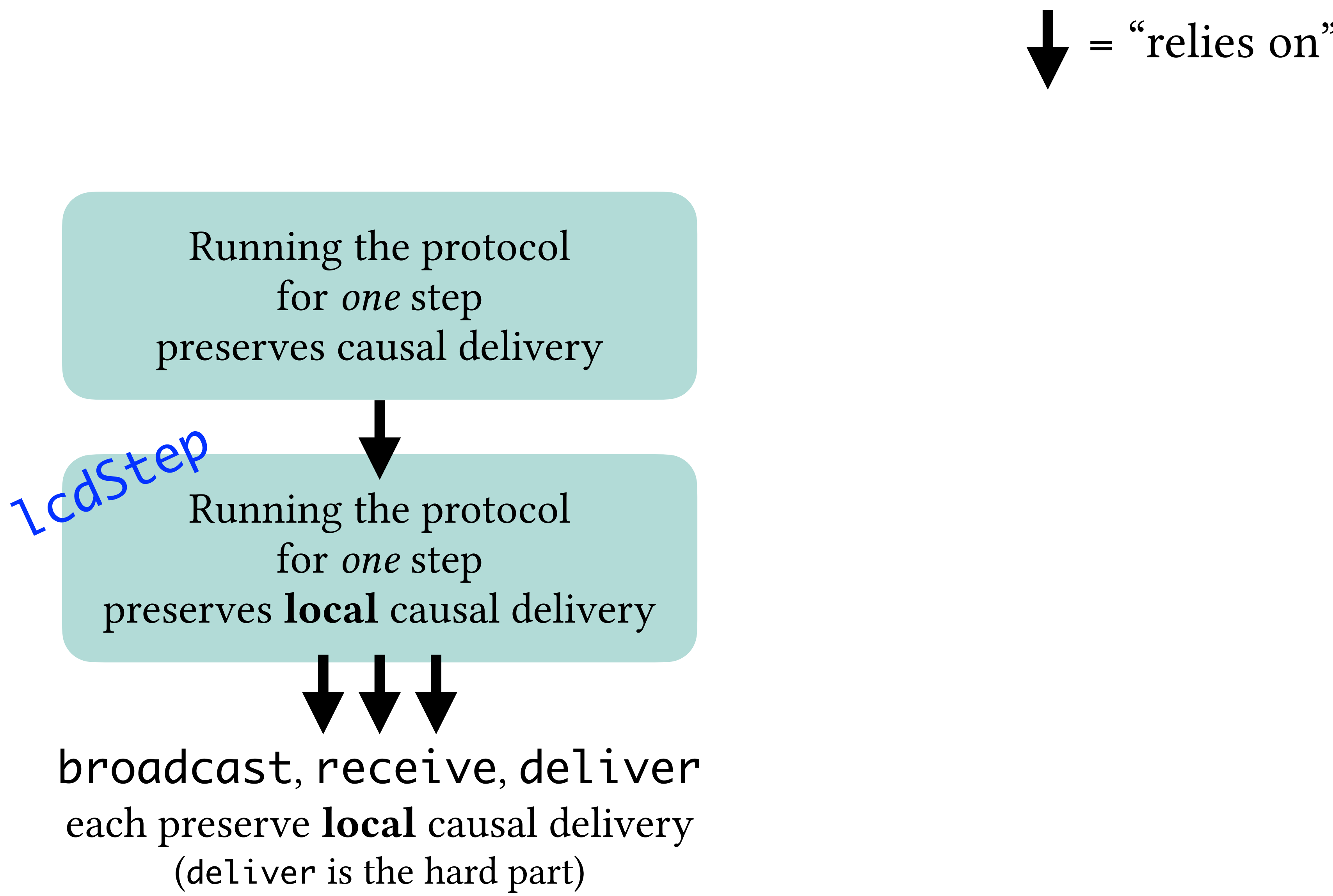
↓ = “relies on”

`lcdStep`

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

↓↓↓

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



= "relies on"
lcdStep
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)

↓ = “relies on”

Running the protocol
for *one* step
preserves causal delivery

↓

`lcdStep`

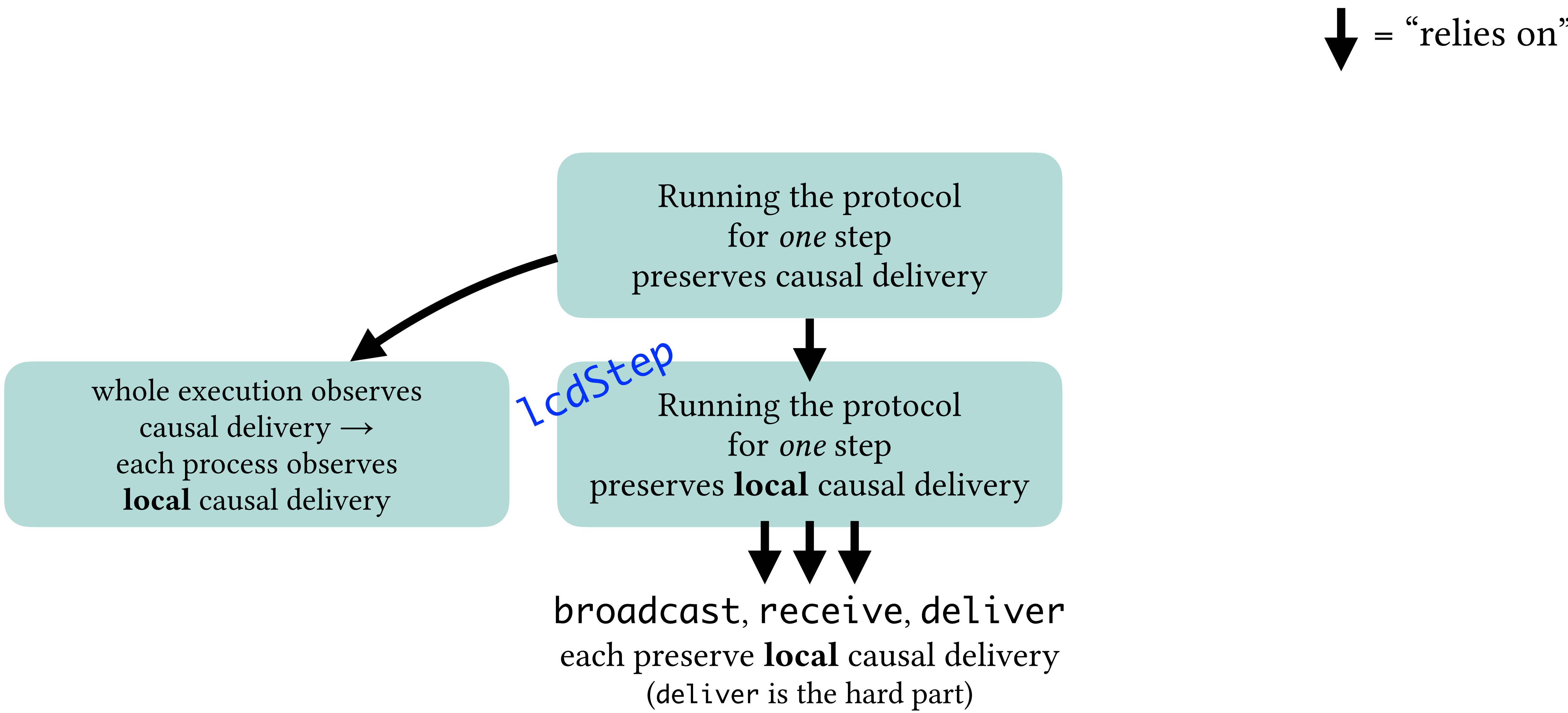
Running the protocol
for *one* step
preserves **local** causal delivery

↓ ↓ ↓

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

verification code

```
type CausalDelivery X =
    pid : PID -- a pid in the domain of execution X
  -> { m1 : Message | elem (Deliver pid m1) (pHist (X pid)) }
  -> { m2 : Message | elem (Deliver pid m2) (pHist (X pid))
                      && happensBefore X (Broadcast m1) (Broadcast m2) }
  -> { _: Proof | procOrder (pHist (X pid)) (Deliver pid m1) (Deliver pid m2) }
```



verification code

```
type CausalDelivery X =
    pid : PID -- a pid in the domain of execution X
  -> { m1 : Message | elem (Deliver pid m1) (pHist (X pid)) }
  -> { m2 : Message | elem (Deliver pid m2) (pHist (X pid))
                    && happensBefore X (Broadcast m1) (Broadcast m2) }
  -> { _: Proof | procOrder (pHist (X pid)) (Deliver pid m1) (Deliver pid m2) }
```

= “relies on”

Running the protocol
for *one* step
preserves causal delivery

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

`lcdStep`

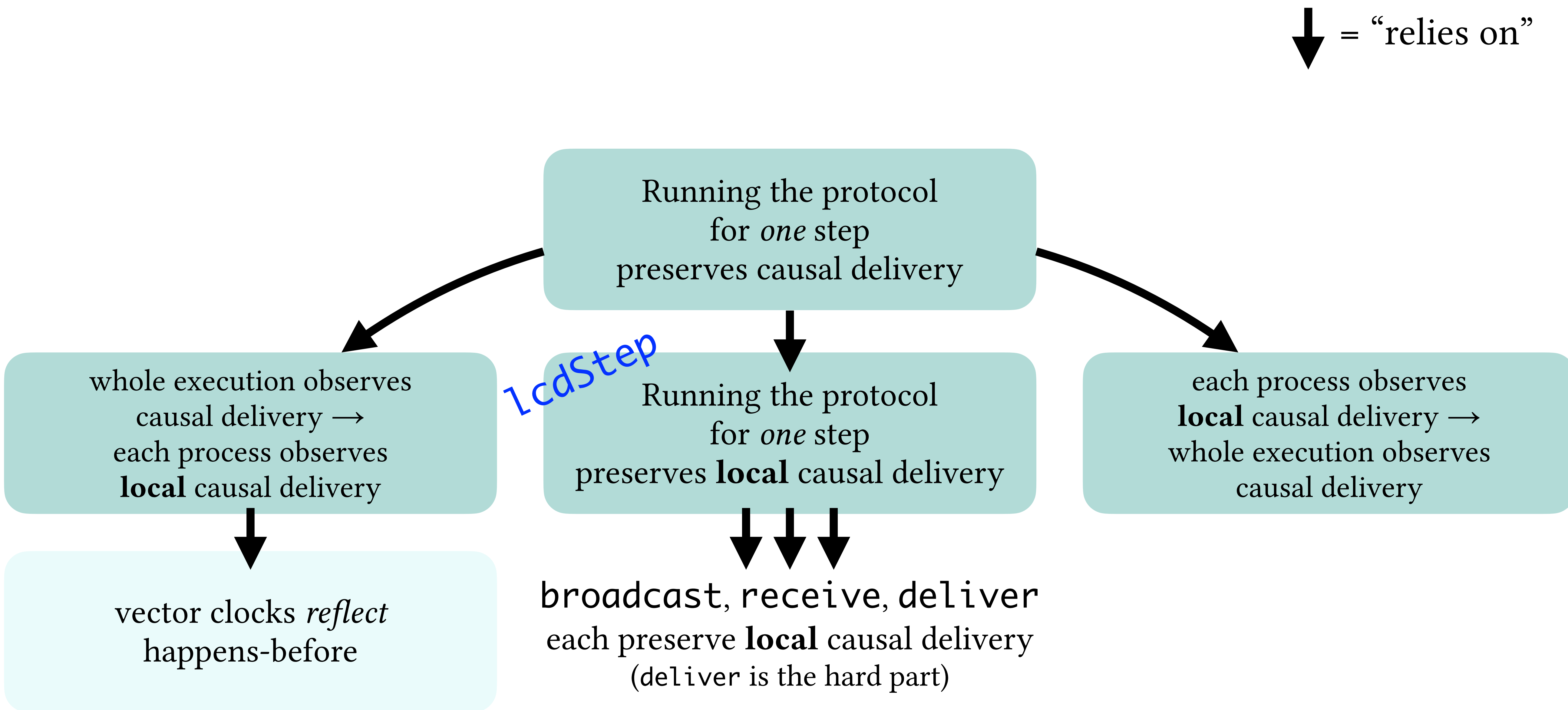
Running the protocol
for *one* step
preserves **local** causal delivery

vector clocks *reflect*
happens-before

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

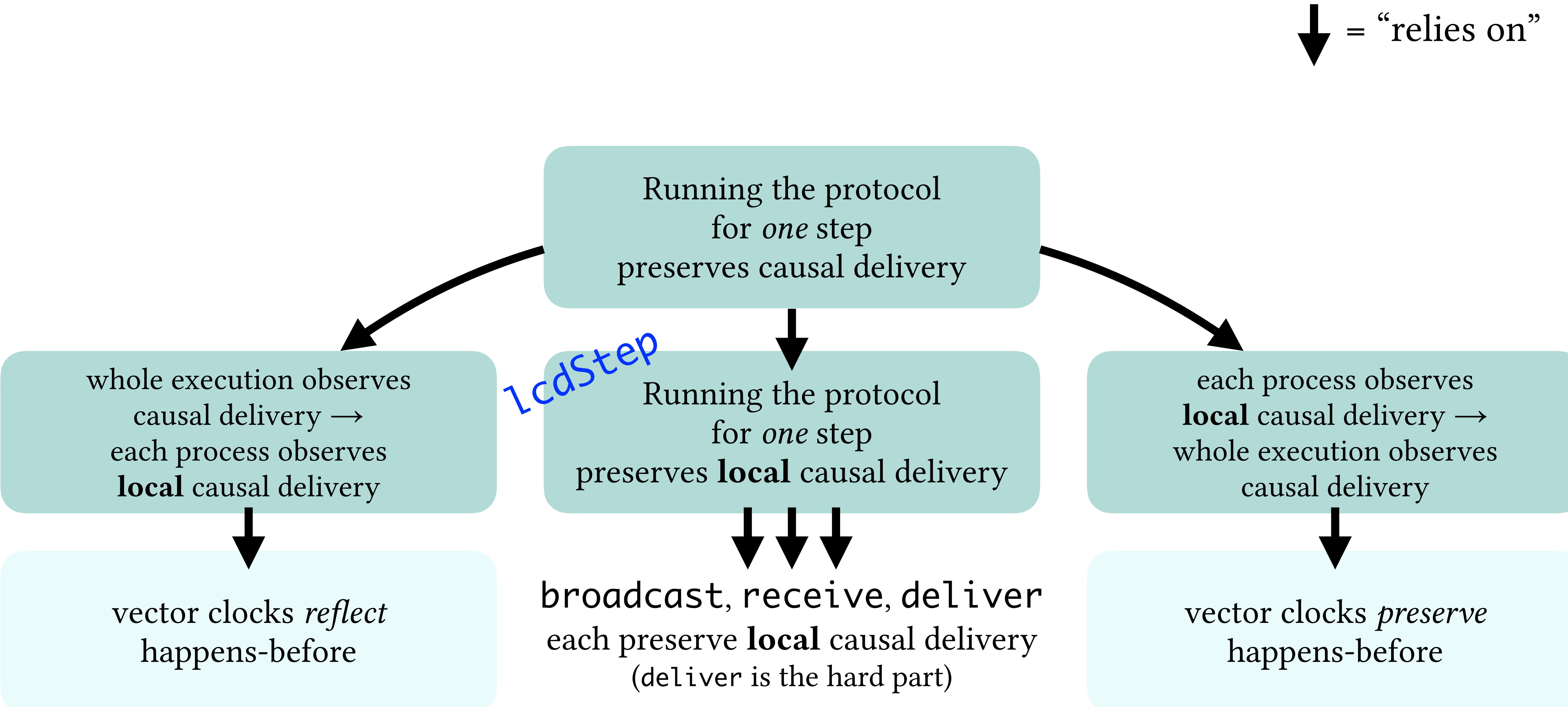
verification code

```
type CausalDelivery X =
    pid : PID -- a pid in the domain of execution X
  -> { m1 : Message | elem (Deliver pid m1) (pHist (X pid)) }
  -> { m2 : Message | elem (Deliver pid m2) (pHist (X pid))
                      && happensBefore X (Broadcast m1) (Broadcast m2) }
  -> { _: Proof | procOrder (pHist (X pid)) (Deliver pid m1) (Deliver pid m2) }
```



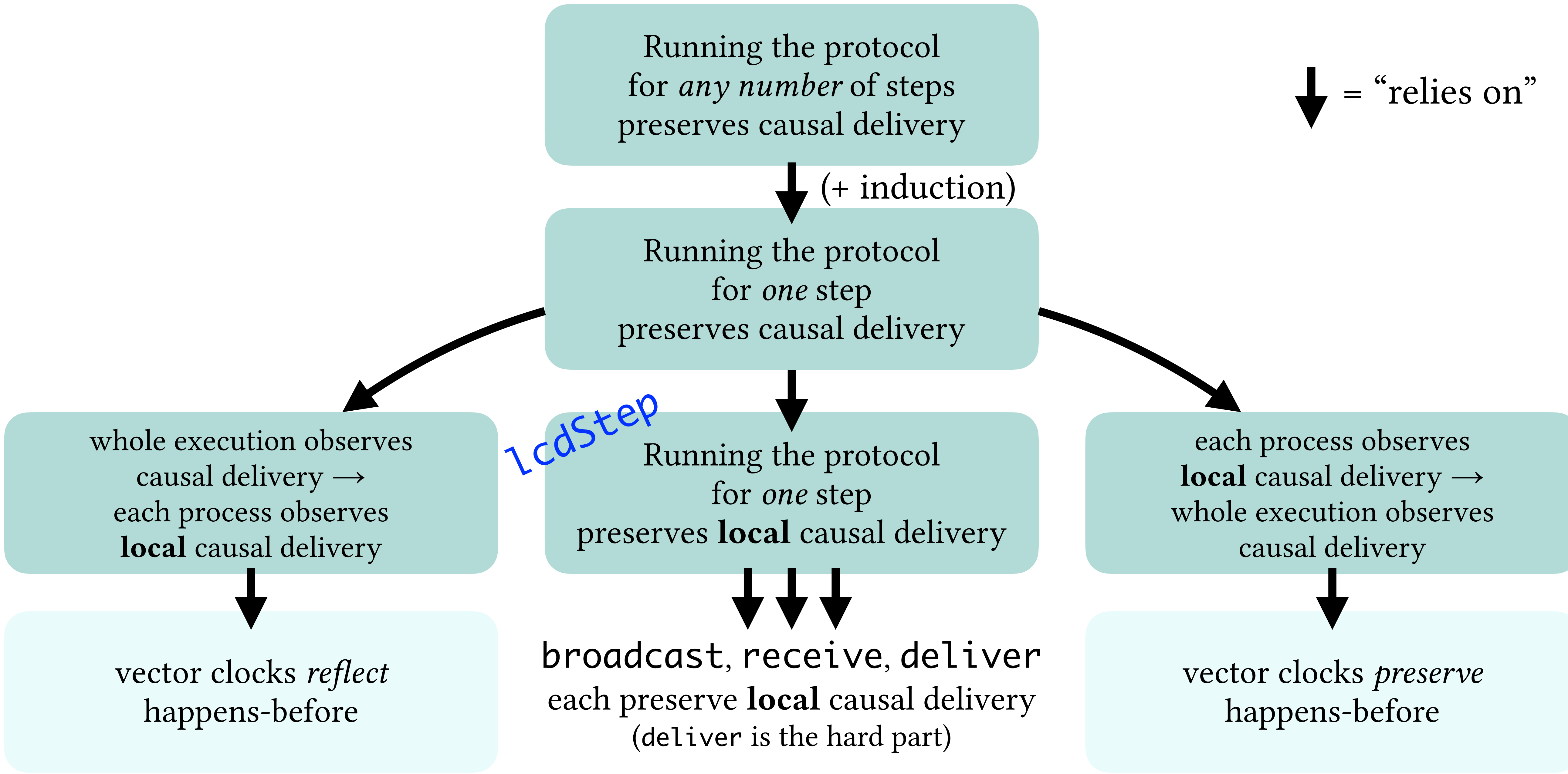
verification code

```
type CausalDelivery X =
    pid : PID -- a pid in the domain of execution X
  -> { m1 : Message | elem (Deliver pid m1) (pHist (X pid)) }
  -> { m2 : Message | elem (Deliver pid m2) (pHist (X pid))
                    && happensBefore X (Broadcast m1) (Broadcast m2) }
  -> { _: Proof | procOrder (pHist (X pid)) (Deliver pid m1) (Deliver pid m2) }
```



verification code

```
type CausalDelivery X =
    pid : PID -- a pid in the domain of execution X
  -> { m1 : Message | elem (Deliver pid m1) (pHist (X pid)) }
  -> { m2 : Message | elem (Deliver pid m2) (pHist (X pid))
                      && happensBefore X (Broadcast m1) (Broadcast m2) }
  -> { _: Proof | procOrder (pHist (X pid)) (Deliver pid m1) (Deliver pid m2) }
```



verification code

```
type CausalDelivery X =
    pid : PID -- a pid in the domain of execution X
  -> { m1 : Message | elem (Deliver pid m1) (pHist (X pid)) }
  -> { m2 : Message | elem (Deliver pid m2) (pHist (X pid))
                      && happensBefore X (Broadcast m1) (Broadcast m2) }
  -> { _: Proof | procOrder (pHist (X pid)) (Deliver pid m1) (Deliver pid m2) }
```

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!**)

Toward Hole-Driven Development in Liquid Haskell

PATRICK REDMOND, University of California, Santa Cruz, USA
GAN SHEN, University of California, Santa Cruz, USA
LINDSEY KUPER, University of California, Santa Cruz, USA

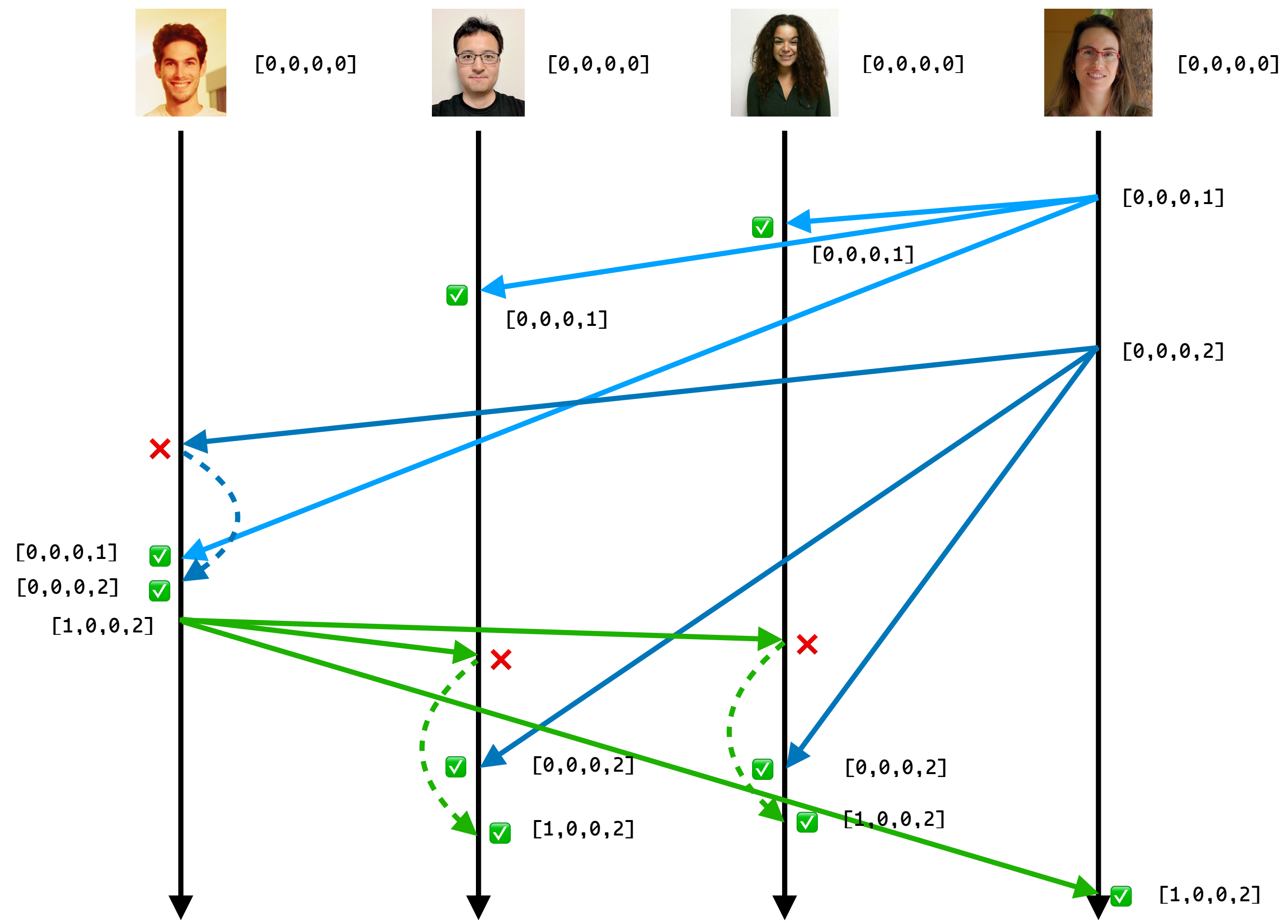
Liquid Haskell is an extension to the Haskell programming language that adds support for *refinement types*: data types augmented with SMT-decidable logical predicates that refine the set of values that can inhabit a type. Furthermore, Liquid Haskell's support for *refinement reflection* enables the use of Haskell for general-purpose mechanized theorem proving. A growing list of large-scale mechanized proof developments in Liquid Haskell take advantage of this capability. Adding theorem-proving capabilities to a "legacy" language like Haskell lets programmers directly verify properties of real-world Haskell programs (taking advantage of the existing highly tuned compiler, run-time system, and libraries), just by writing Haskell. However, more established proof assistants like Agda and Coq offer far better support for interactive proof development and insight into the proof state (for instance, what subgoals still need to be proved to finish a partially-complete proof). In contrast, Liquid Haskell provides only coarse-grained feedback to the user — either it reports a type error, or not — unfortunately hindering its usability as a theorem prover.

In this paper, we propose improving the usability of Liquid Haskell by extending it with support for Agda-style *typed holes* and interactive editing commands that take advantage of them. In Agda, typed holes allow programmers to indicate unfinished parts of a proof, and incrementally complete the proof in a dialogue with the compiler. While GHC Haskell already has its own Agda-inspired support for typed holes, we posit

[HATRA 2021]

Thank you!

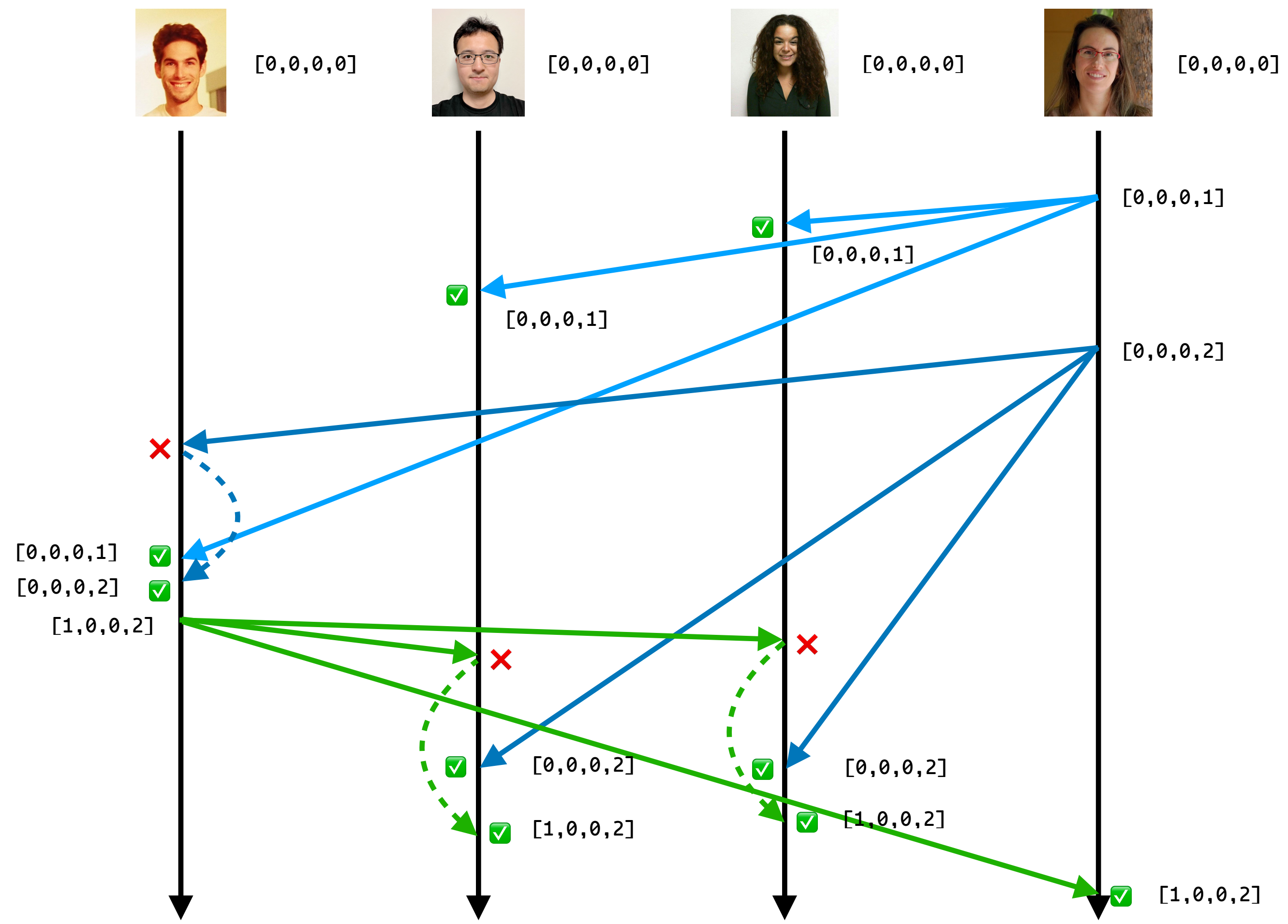
Languages, Systems, and Data Lab: lsd.ucsc.edu
Lindsey's research blog: decomposition.al



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