Verified Causal Broadcast with Liquid Haskell

Patrick Redmond Gan Shen Niki Vazou Lindsey Kuper

IFL 2022
Copenhagen, Denmark
31 August 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

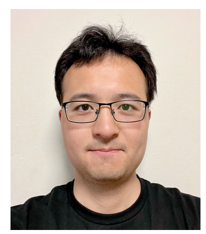
IFL 2022
Copenhagen, Denmark
31 August 2022

github.com/lsd-ucsc/cbcast-lh

Verified Causal Broadcast with Liquid Haskell

Patrick Redmond

Gan Shen



Niki Vazou

Lindsey Kuper

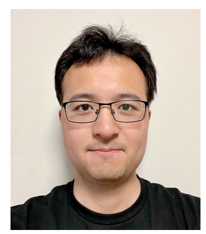
IFL 2022
Copenhagen, Denmark
31 August 2022

github.com/lsd-ucsc/cbcast-lh

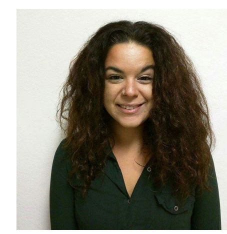
Verified Causal Broadcast with Liquid Haskell

Patrick Redmond

Gan Shen



Niki Vazou



Lindsey Kuper

IFL 2022
Copenhagen, Denmark
31 August 2022

UNIVERSITY OF CALIFORNIA SANTA CRUZ

institute imdea software

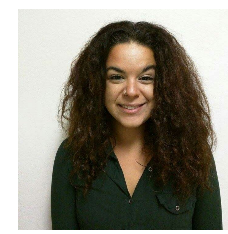
github.com/lsd-ucsc/cbcast-lh

Verified Causal Broadcast with Liquid Haskell

Patrick Redmond

Gan Shen

Niki Vazou



Lindsey Kuper

IFL 2022
Copenhagen, Denmark
31 August 2022

UNIVERSITY OF CALIFORNIA SANTA CRUZ

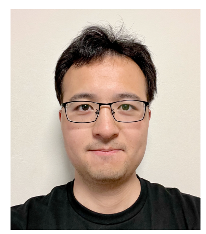
institute imdea software

github.com/lsd-ucsc/cbcast-lh

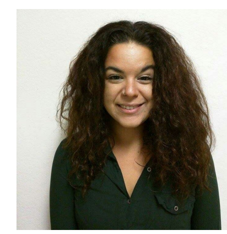
Verified Causal Broadcast with Liquid Haskell

Patrick Redmond

Gan Shen



Niki Vazou



Lindsey Kuper

IFL 2022
Copenhagen, Denmark
31 August 2022

UNIVERSITY OF CALIFORNIA SANTA CRUZ

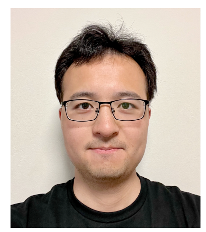
institute imdea software

github.com/lsd-ucsc/cbcast-lh

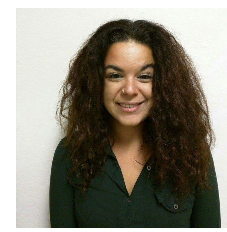
Verified Causal Broadcast with Liquid Haskell

Patrick Redmond

Gan Shen



Niki Vazou



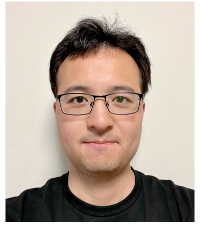
Lindsey Kuper

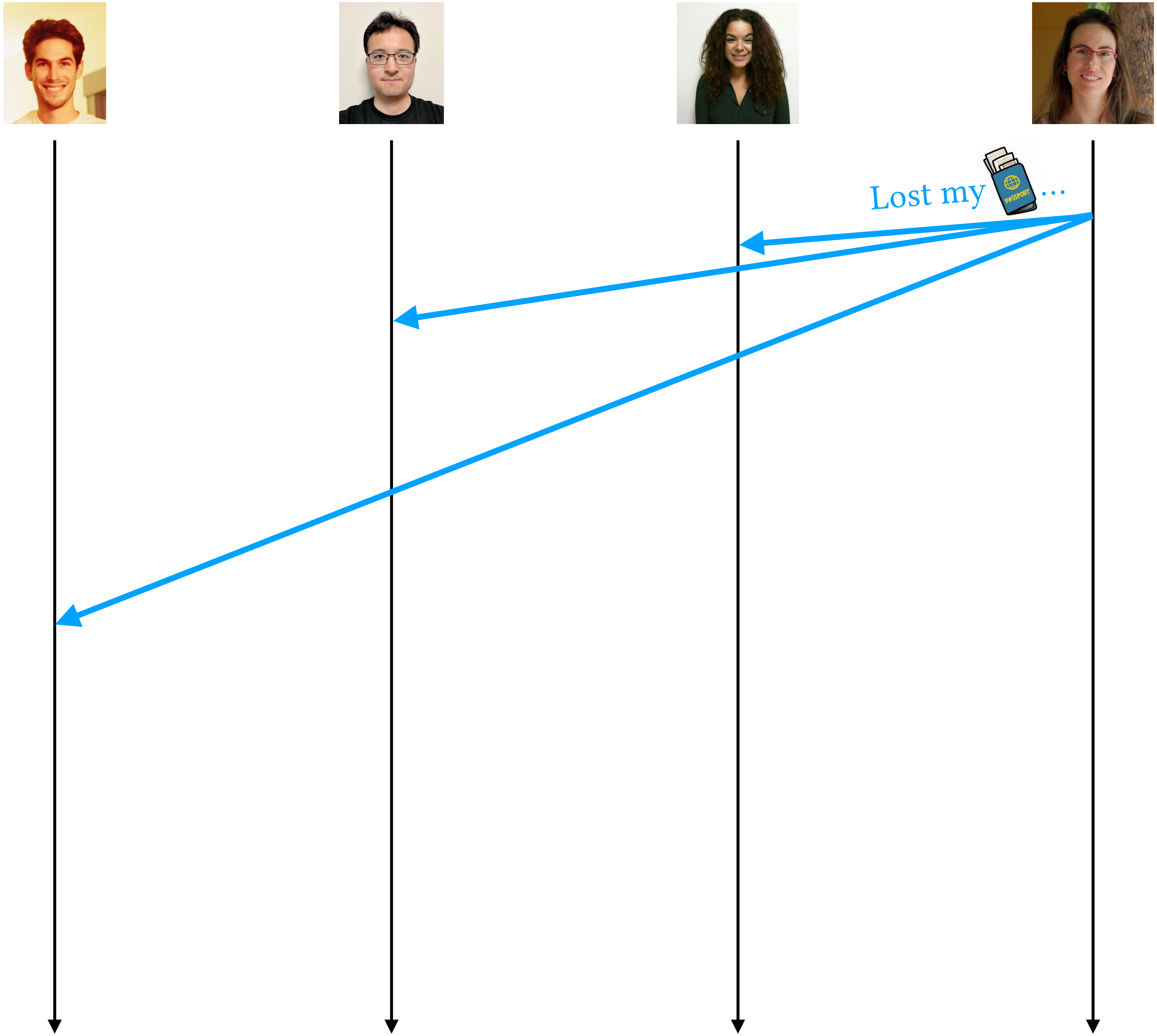
IFL 2022
Copenhagen, Denmark
31 August 2022

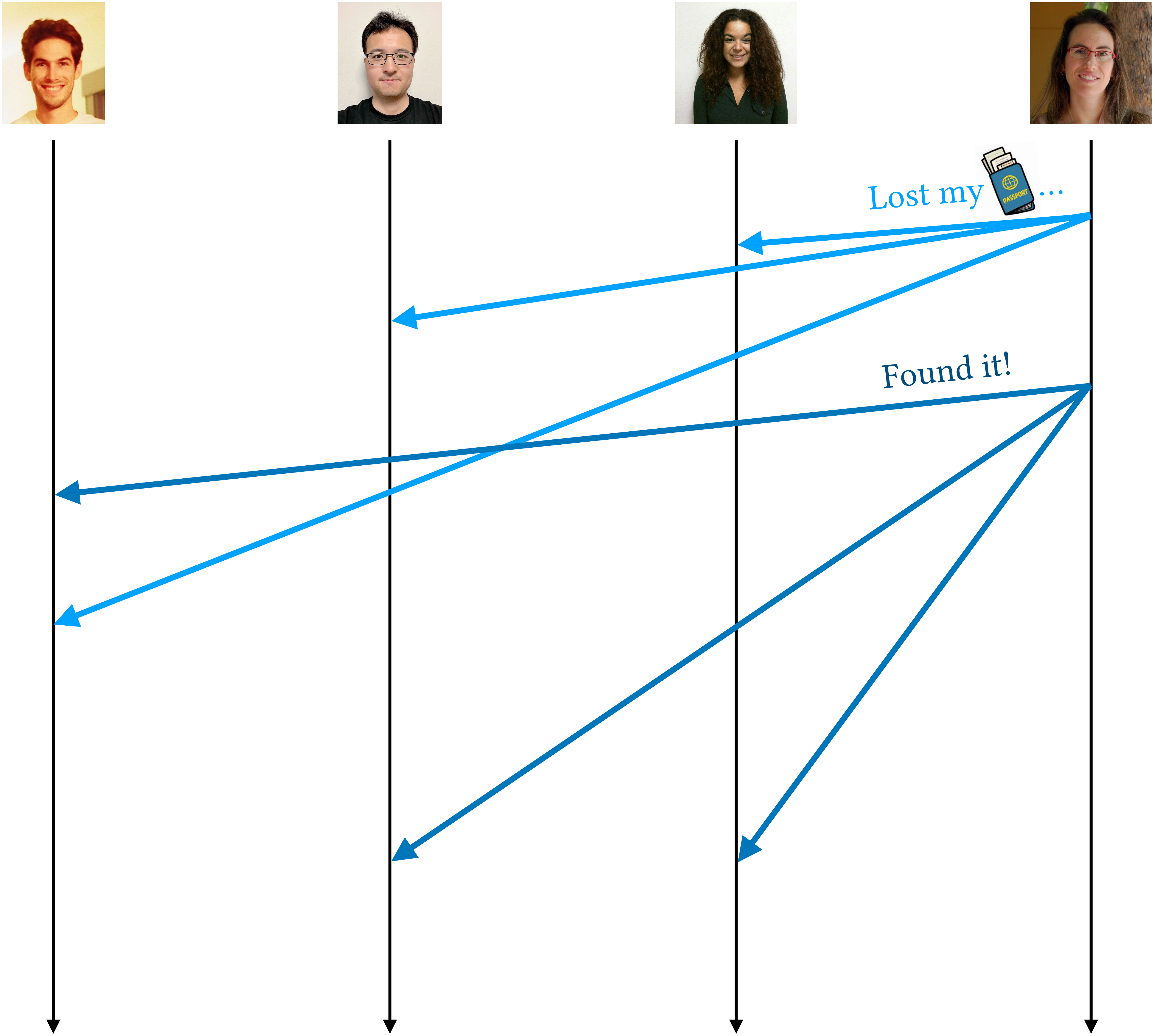
UNIVERSITY OF CALIFORNIA SANTA CRUZ

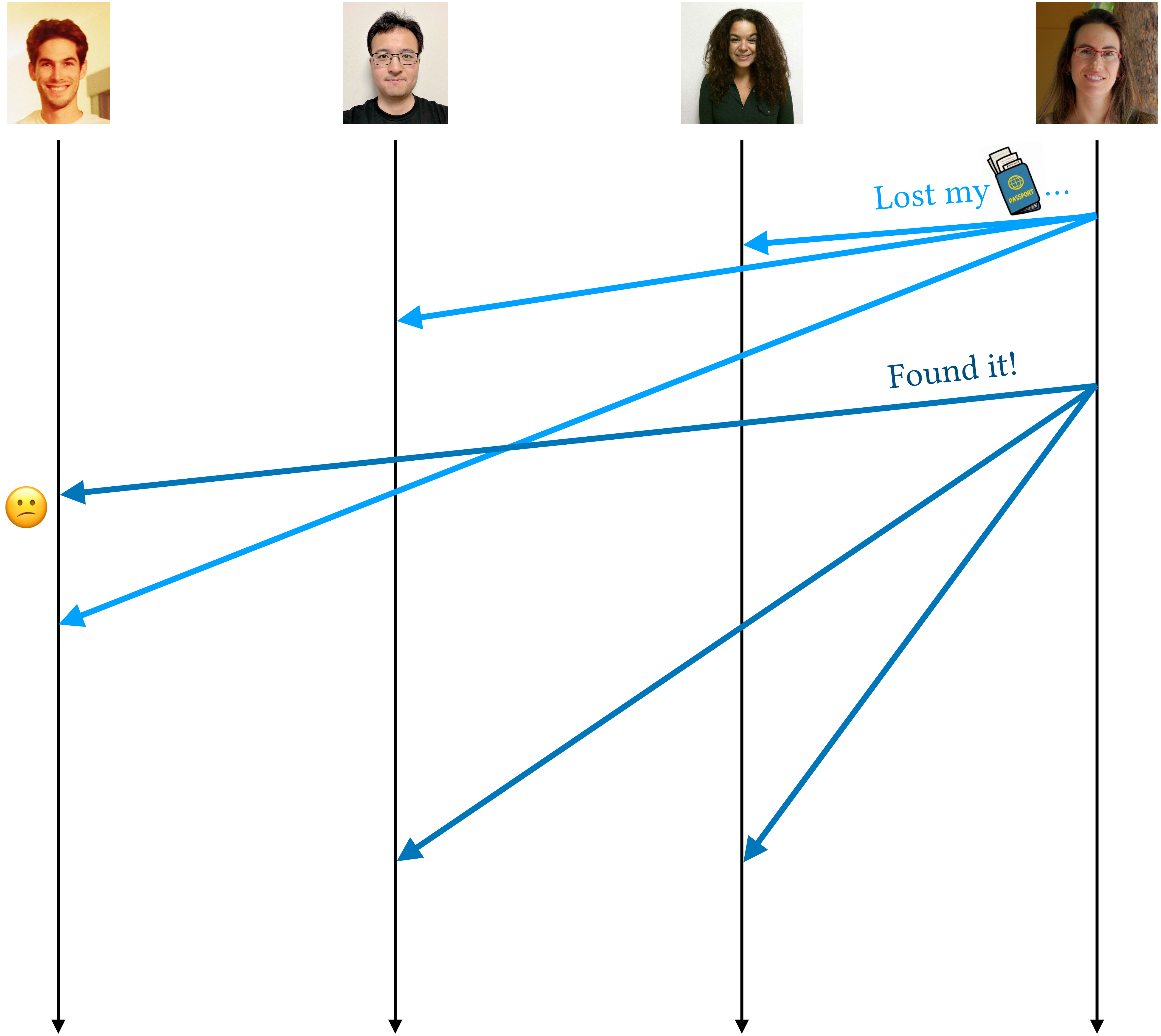
institute imdea software

github.com/lsd-ucsc/cbcast-lh

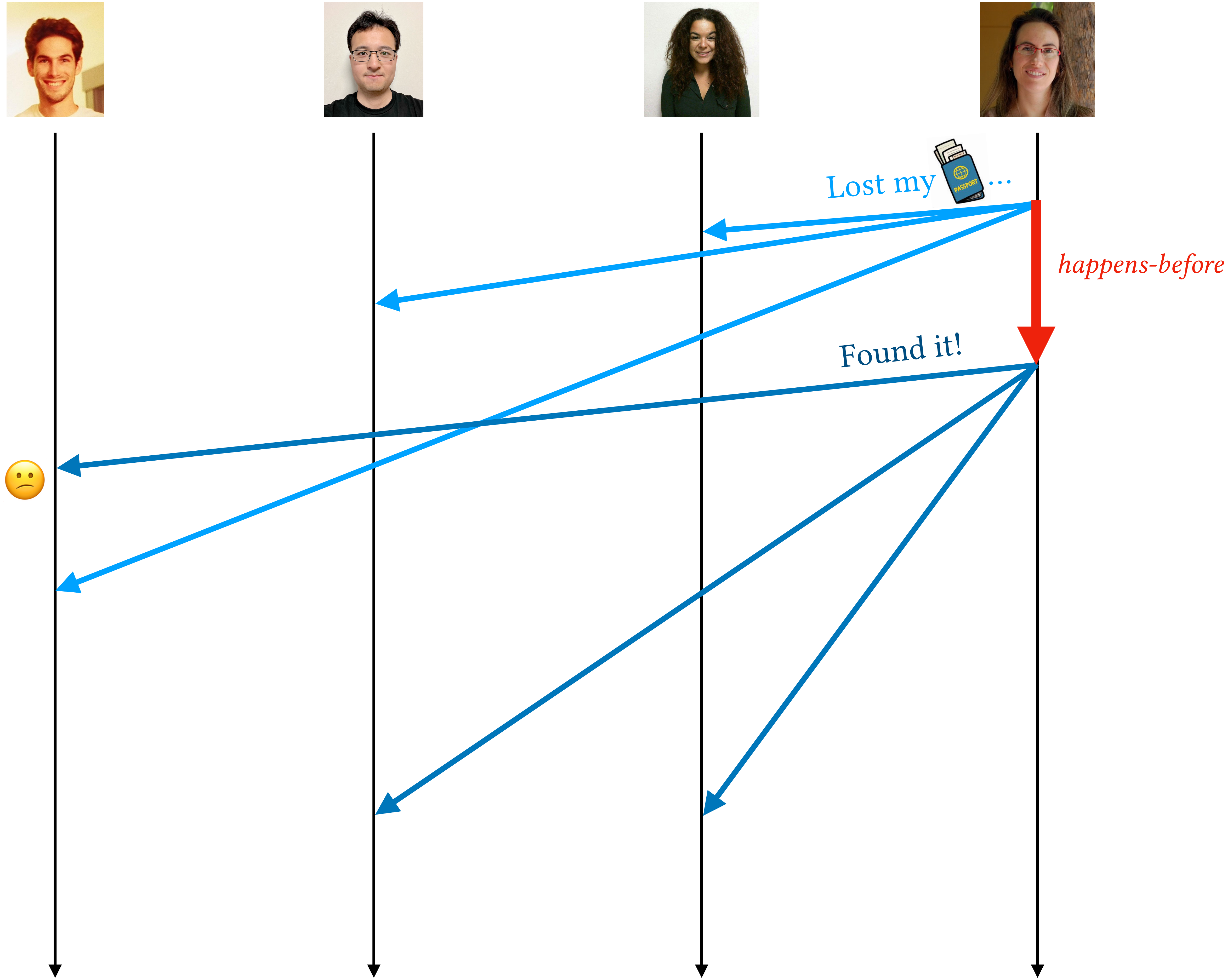
   

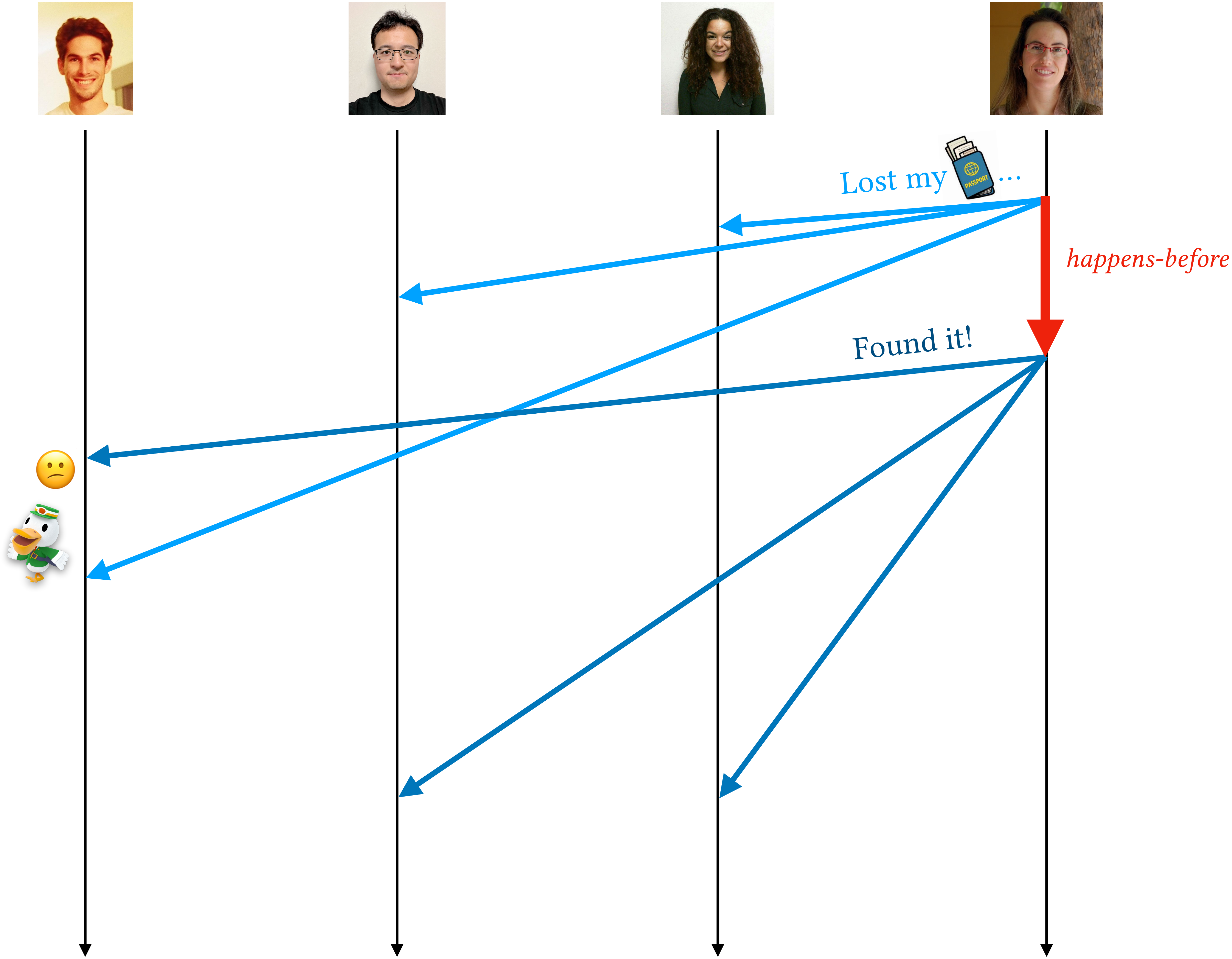

Lost my ...

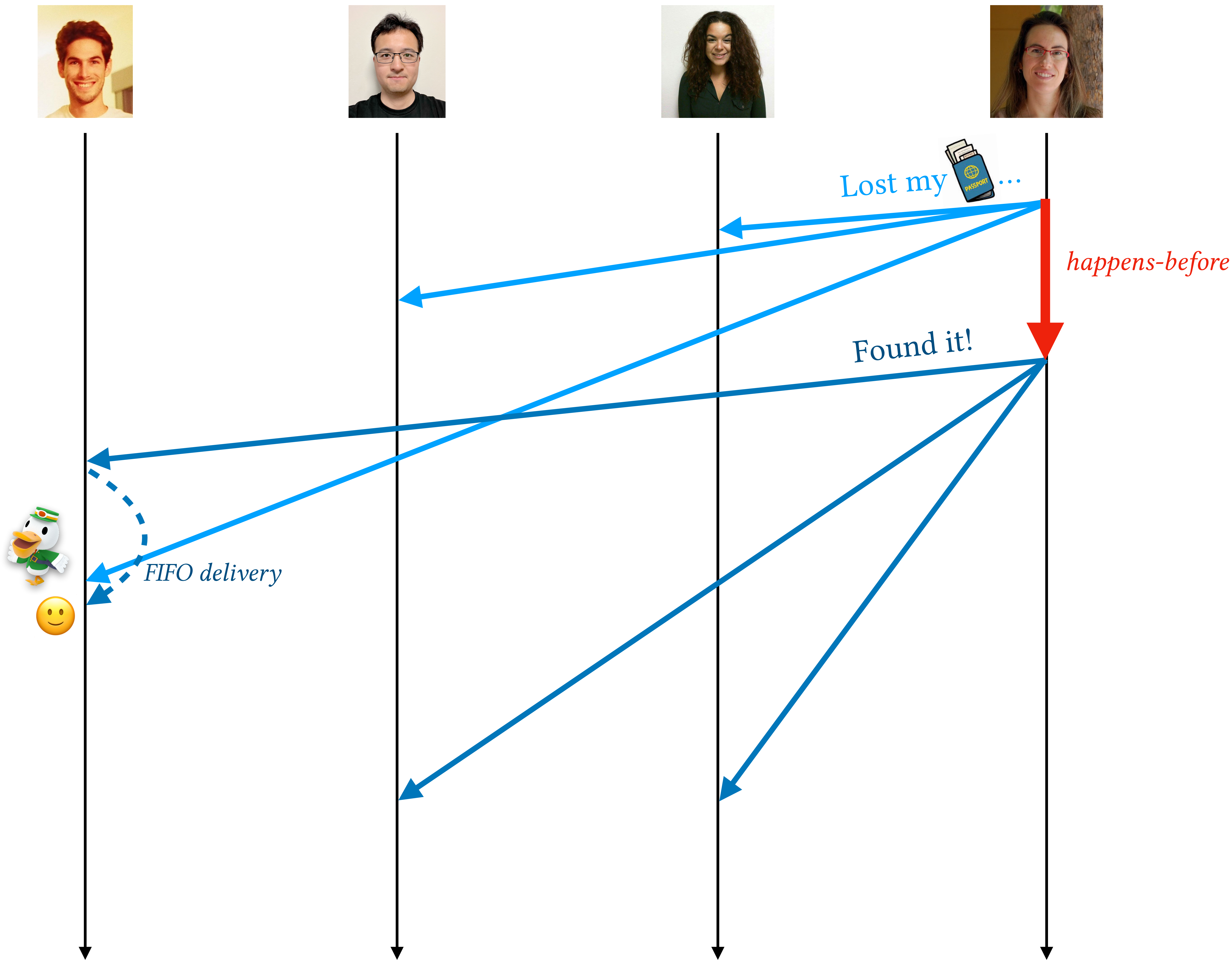

Lost my ...
Found it!

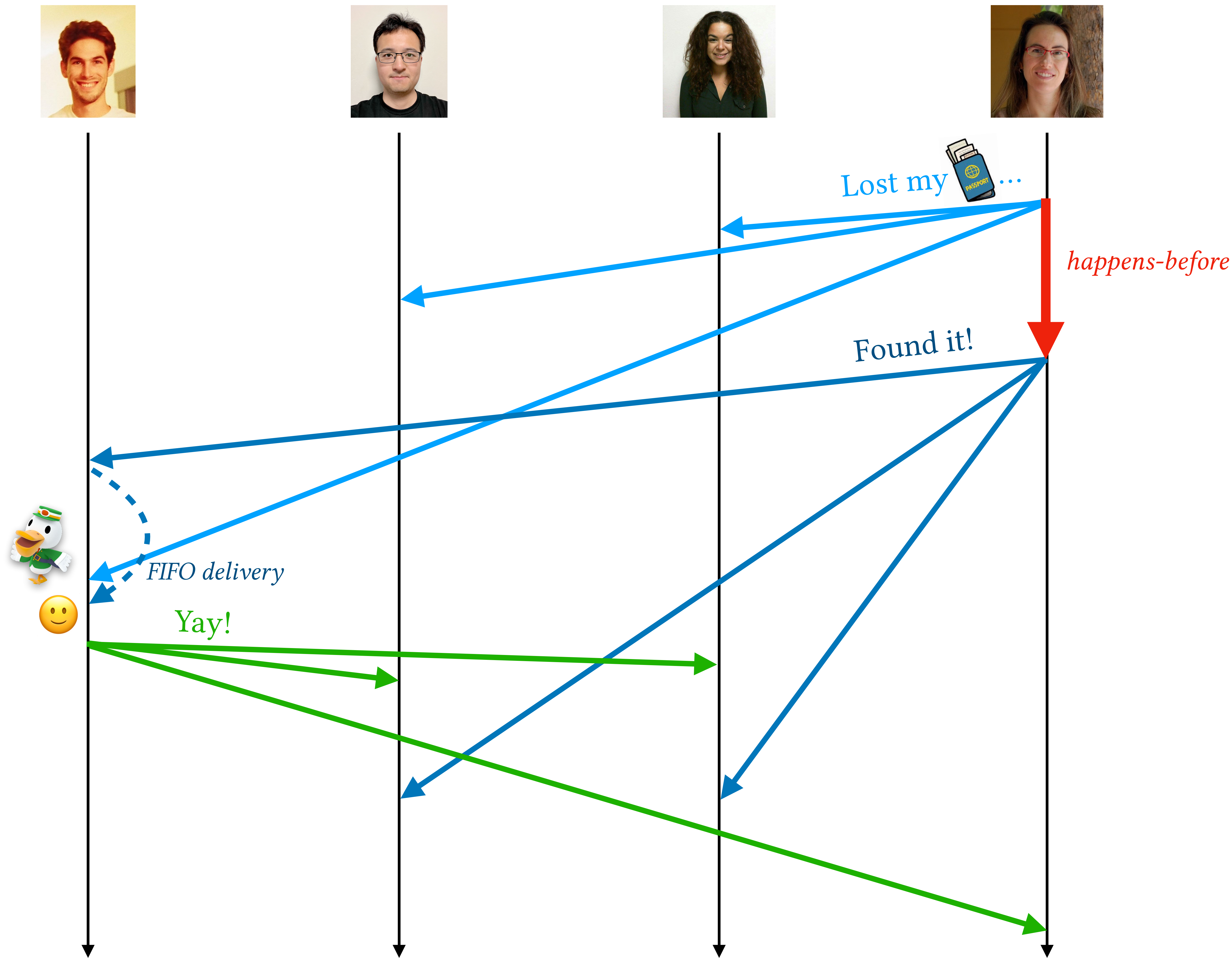


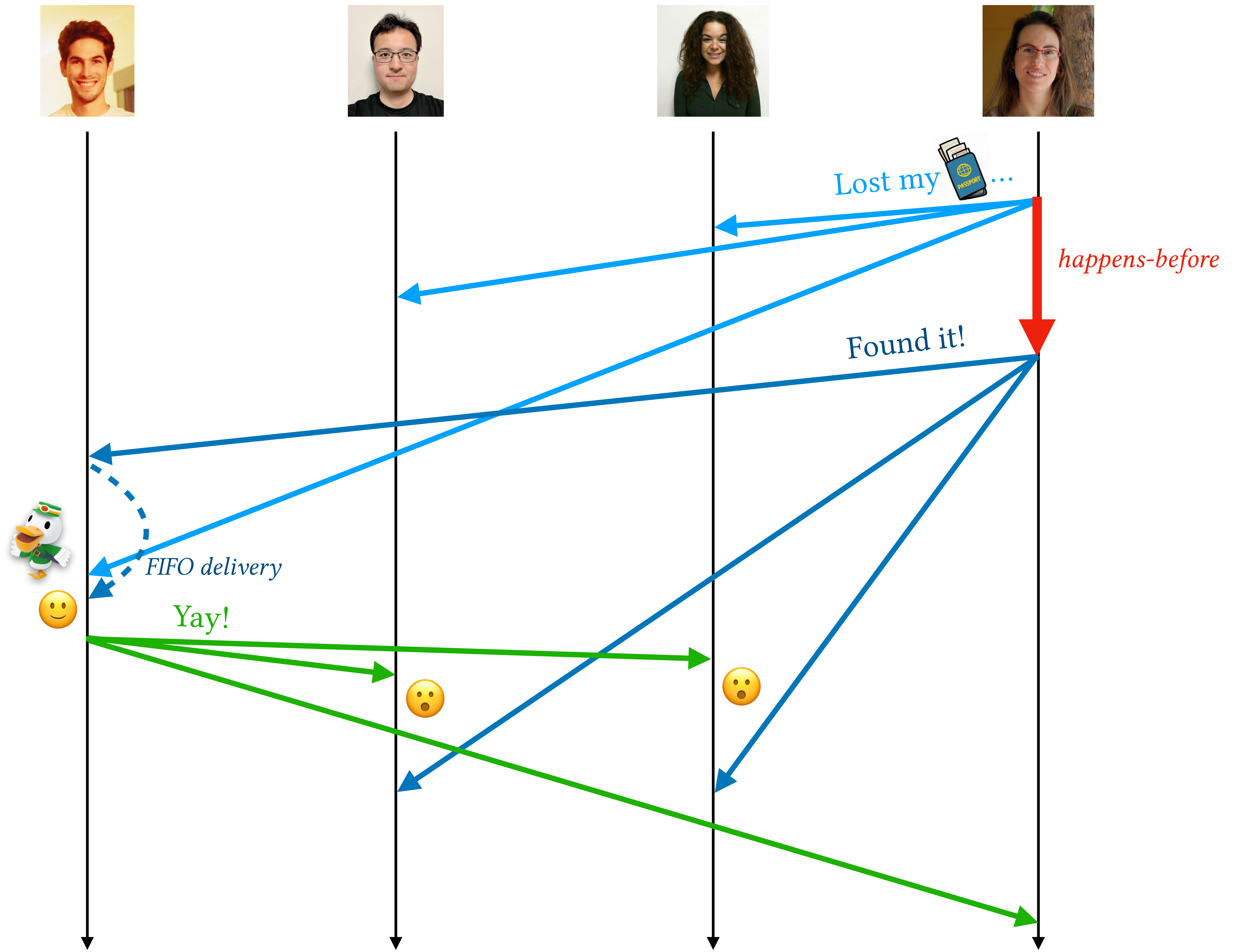
Lost my ...
Found it!


Lost my …
happens-before
Found it!

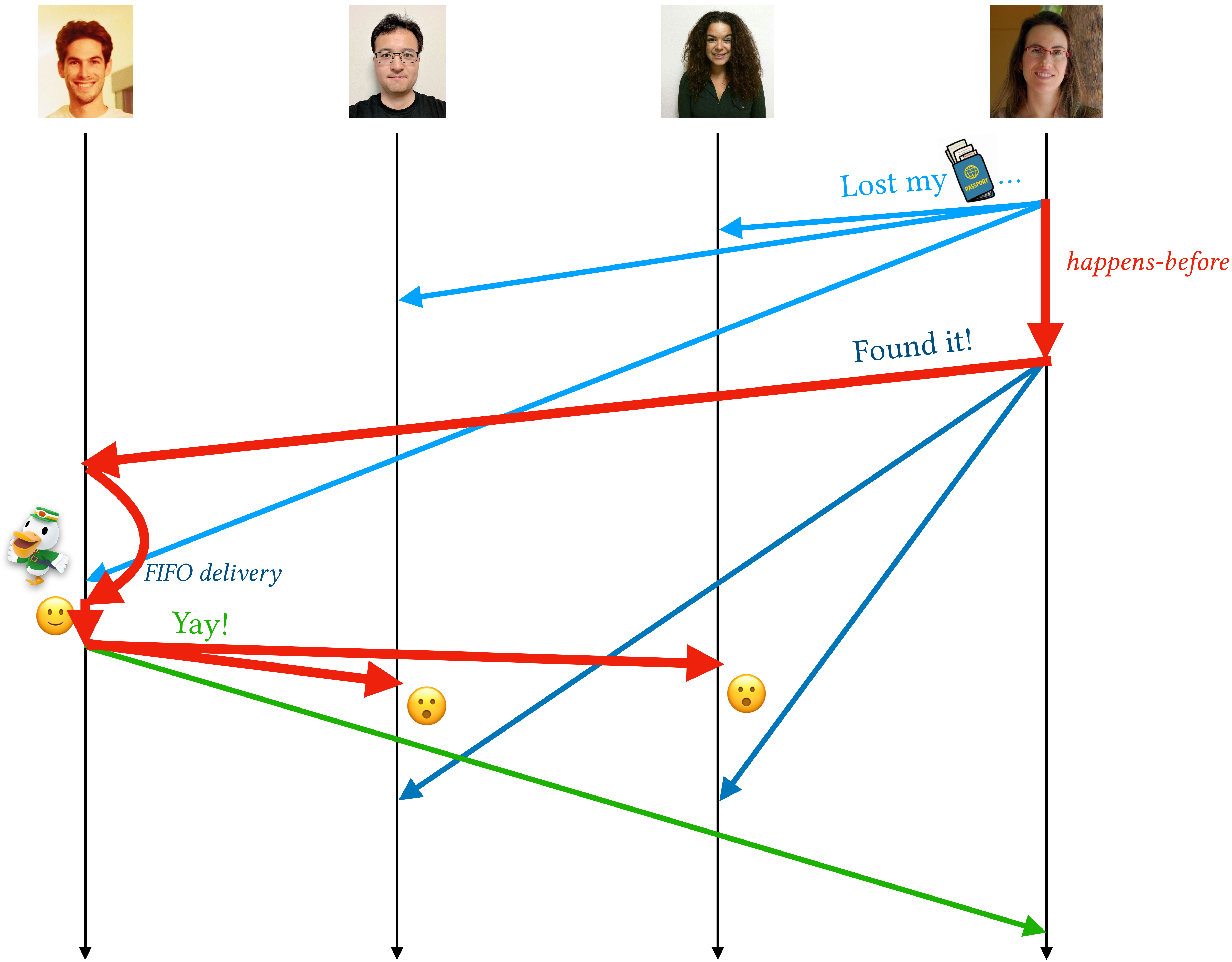

Lost my …
happens-before
Found it!


Lost my …
happens-before
Found it!
FIFO delivery

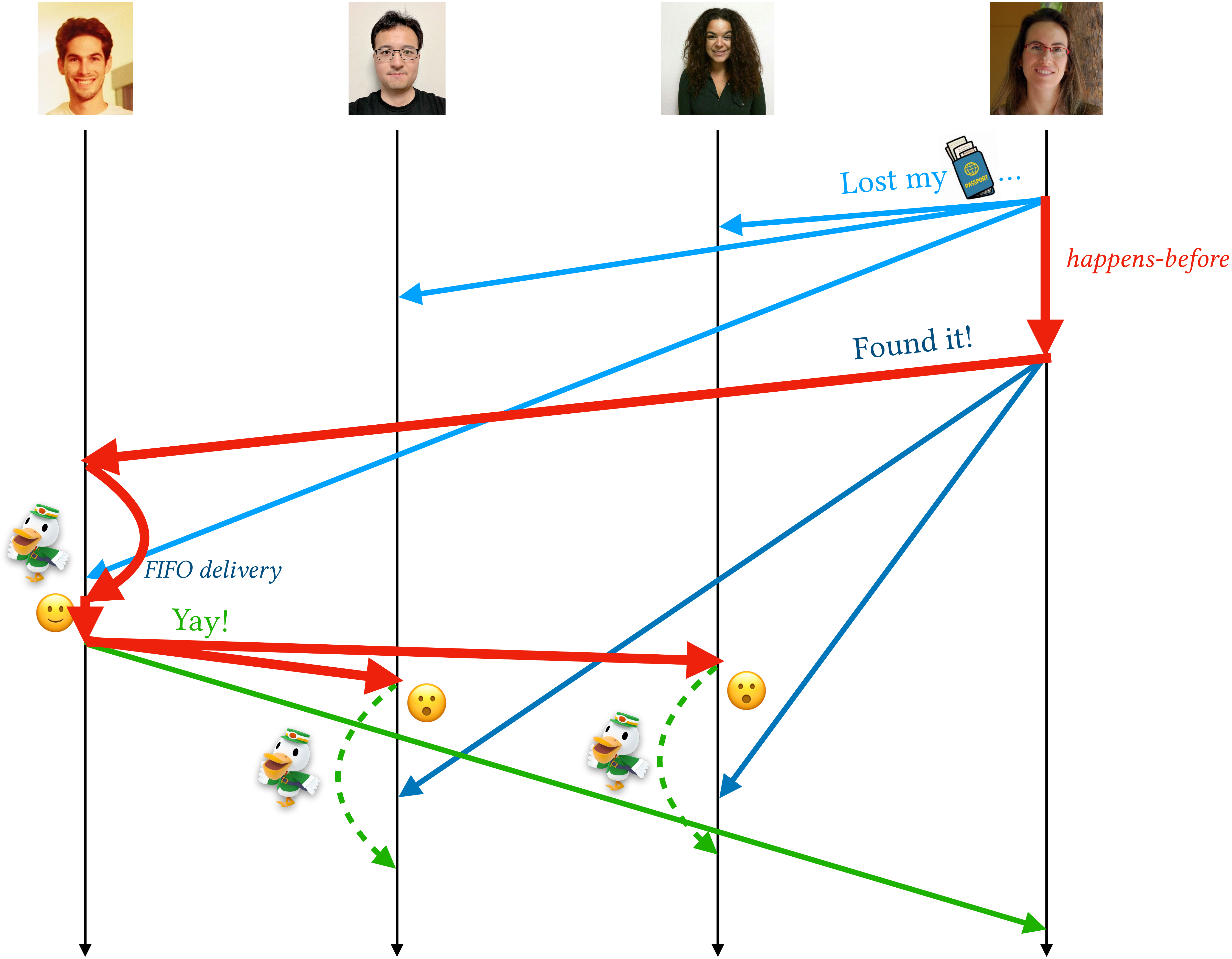

Lost my …
happens-before
Found it!
FIFO delivery
Yay!



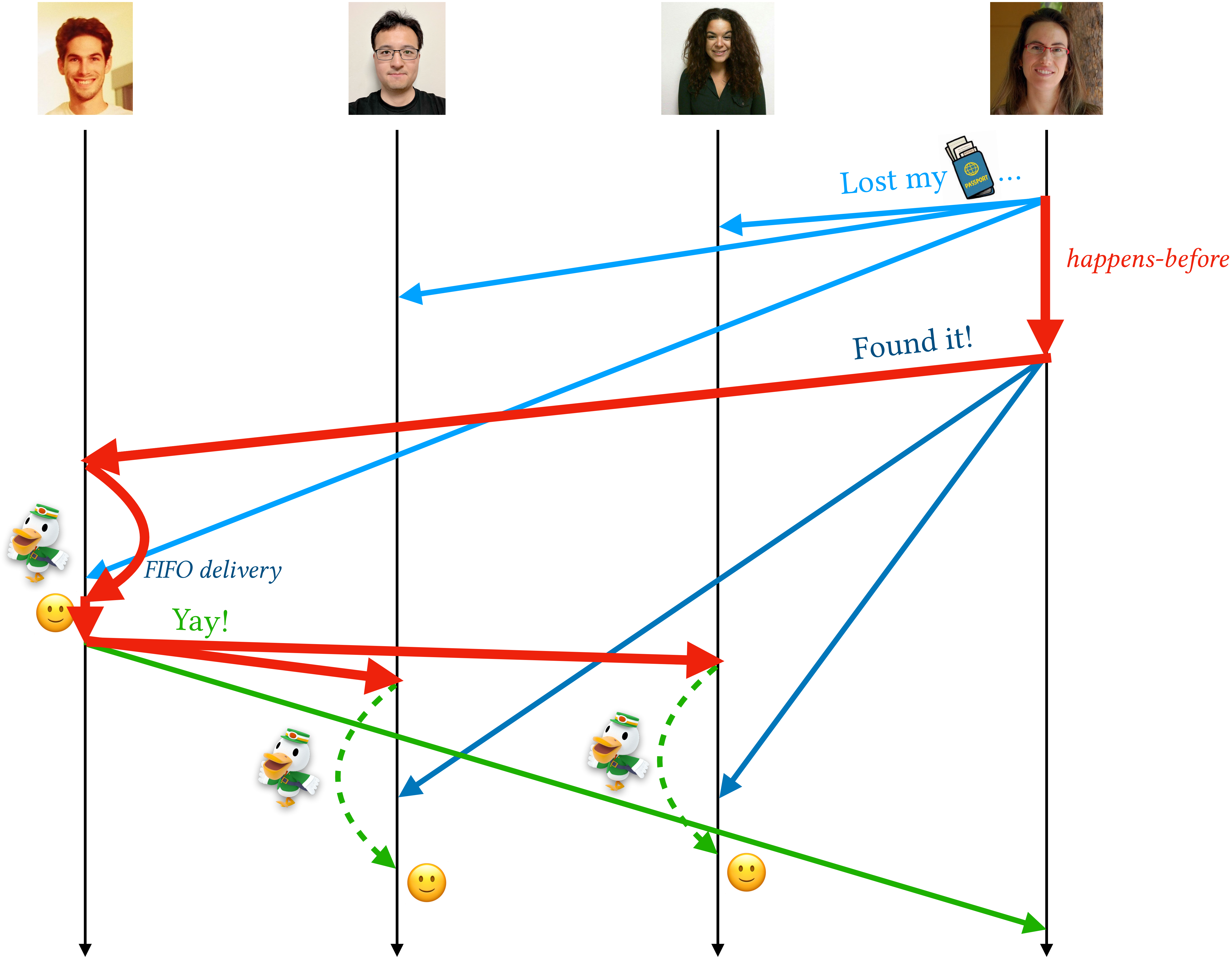
Lost my …
happens-before
Found it!
FIFO delivery
Yay!

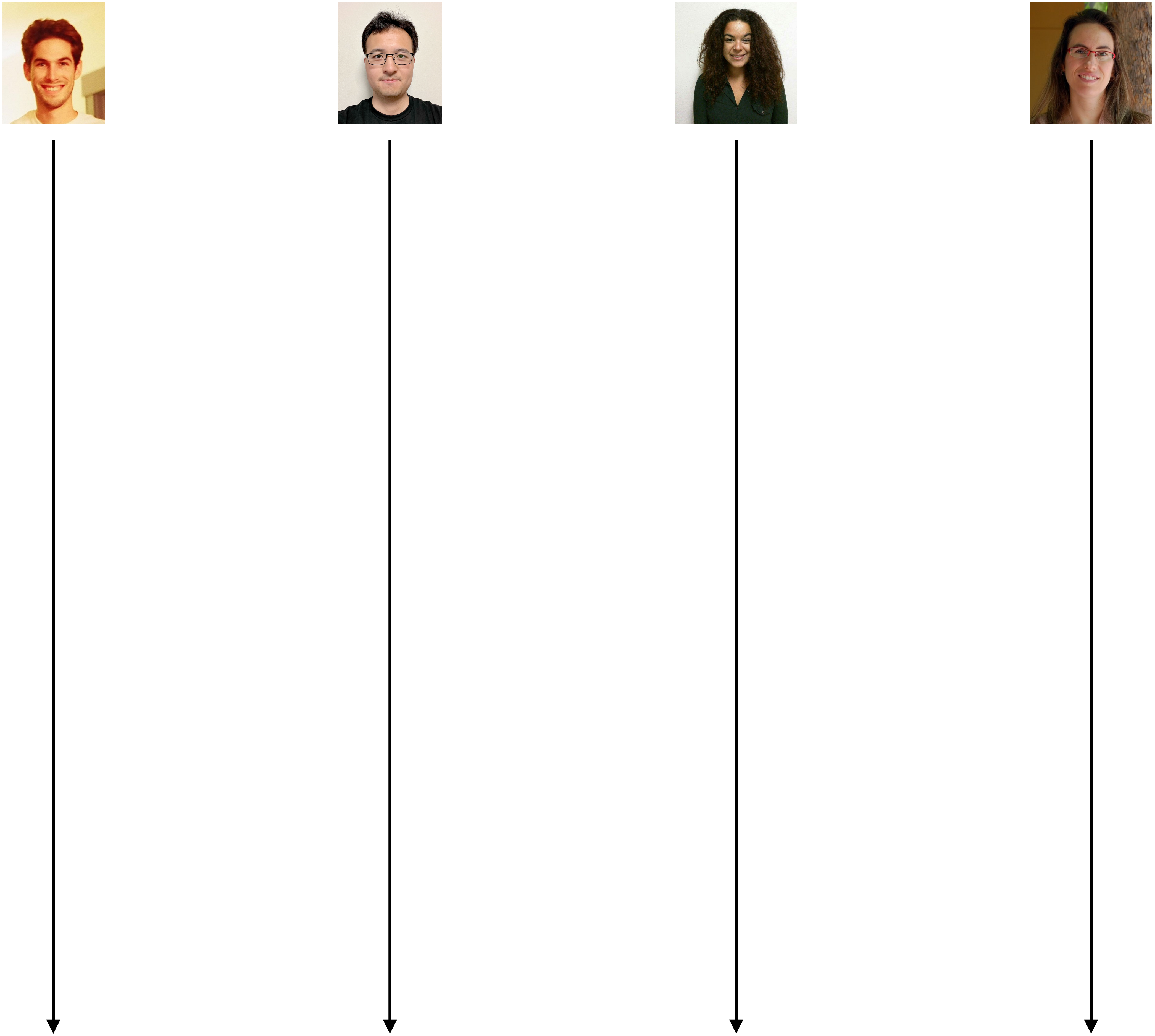


Lost my ...
happens-before
Found it!
FIFO delivery
Yay!

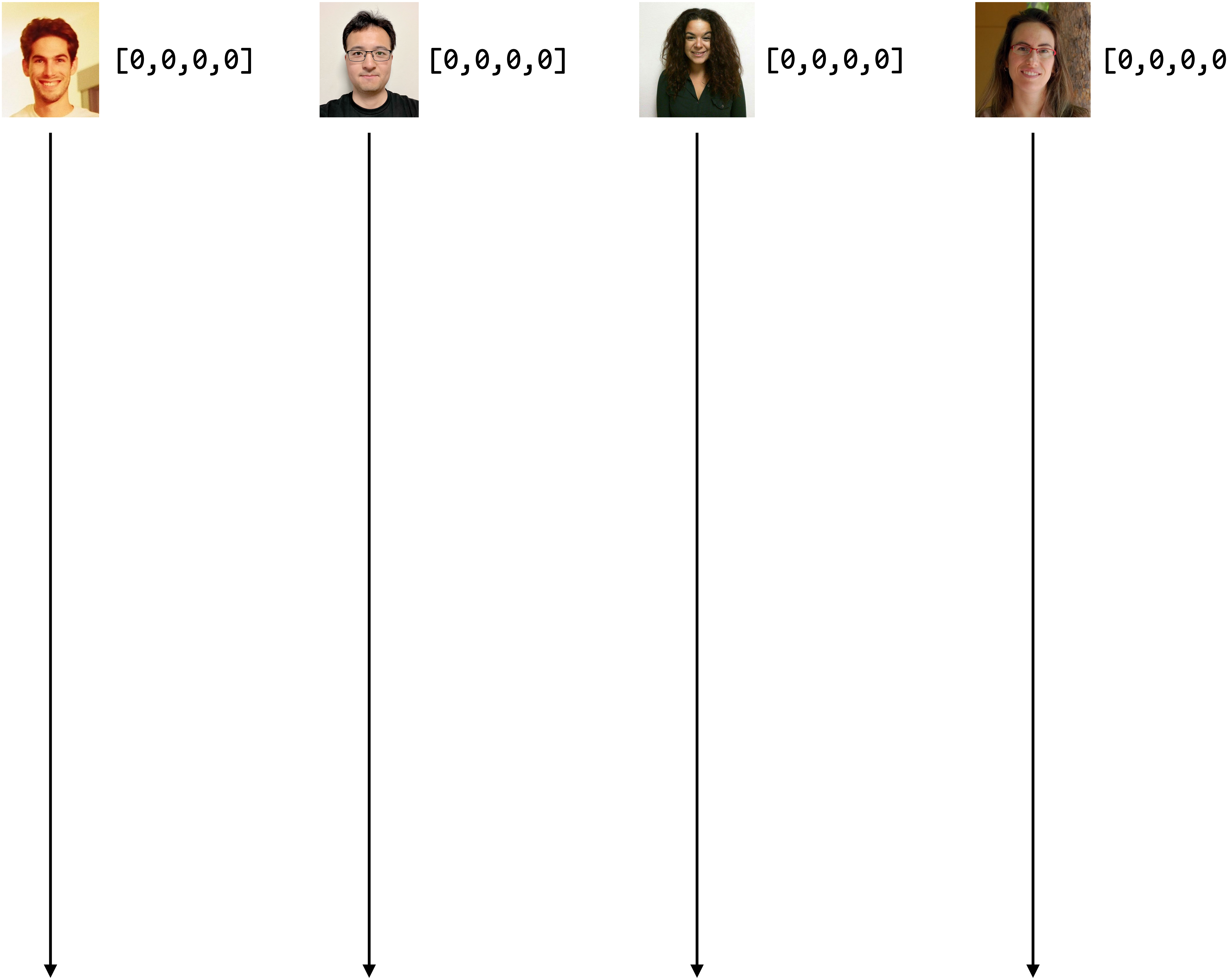


Lost my …
happens-before
Found it!
FIFO delivery
Yay!

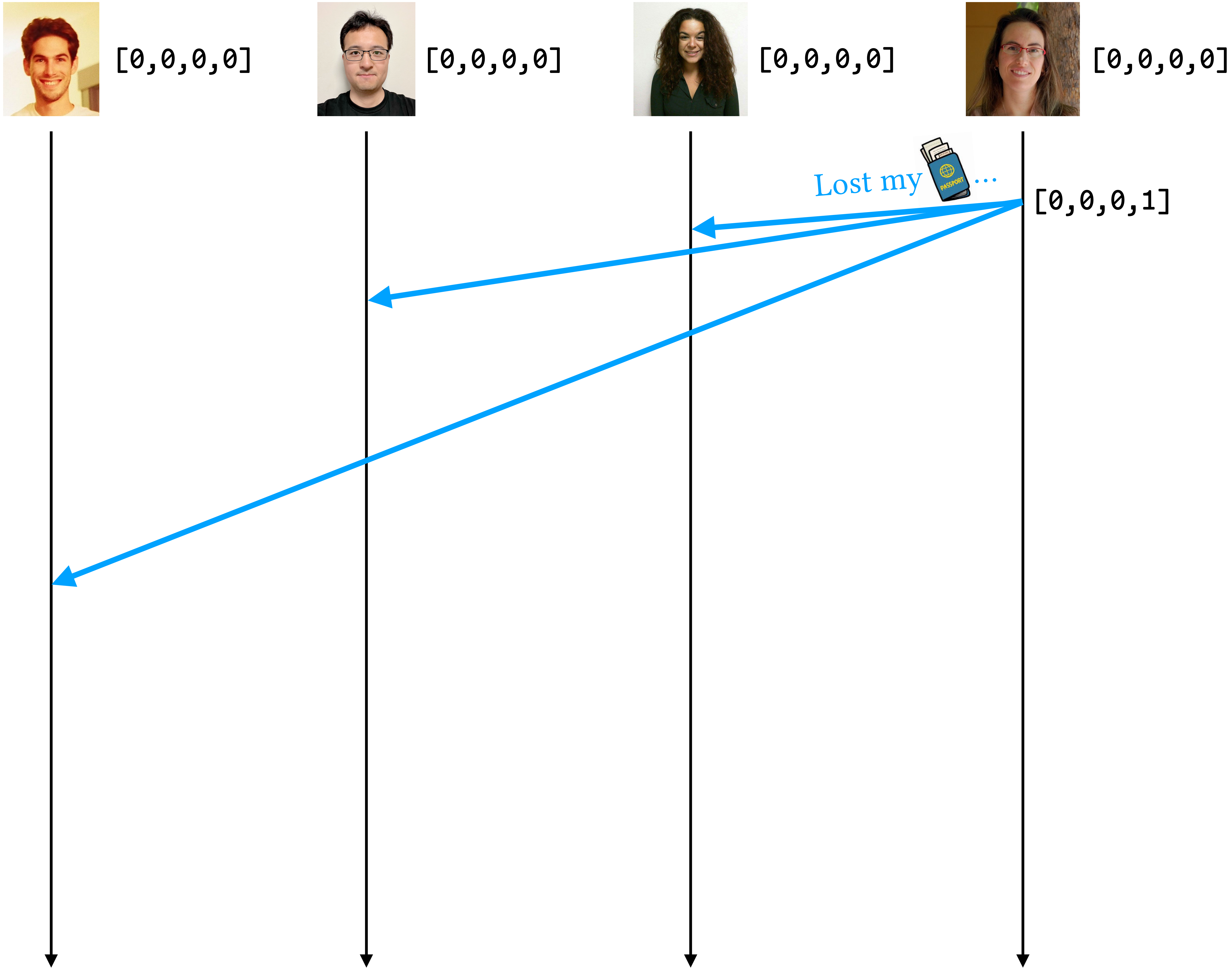

Lost my …
happens-before
Found it!
FIFO delivery
Yay!



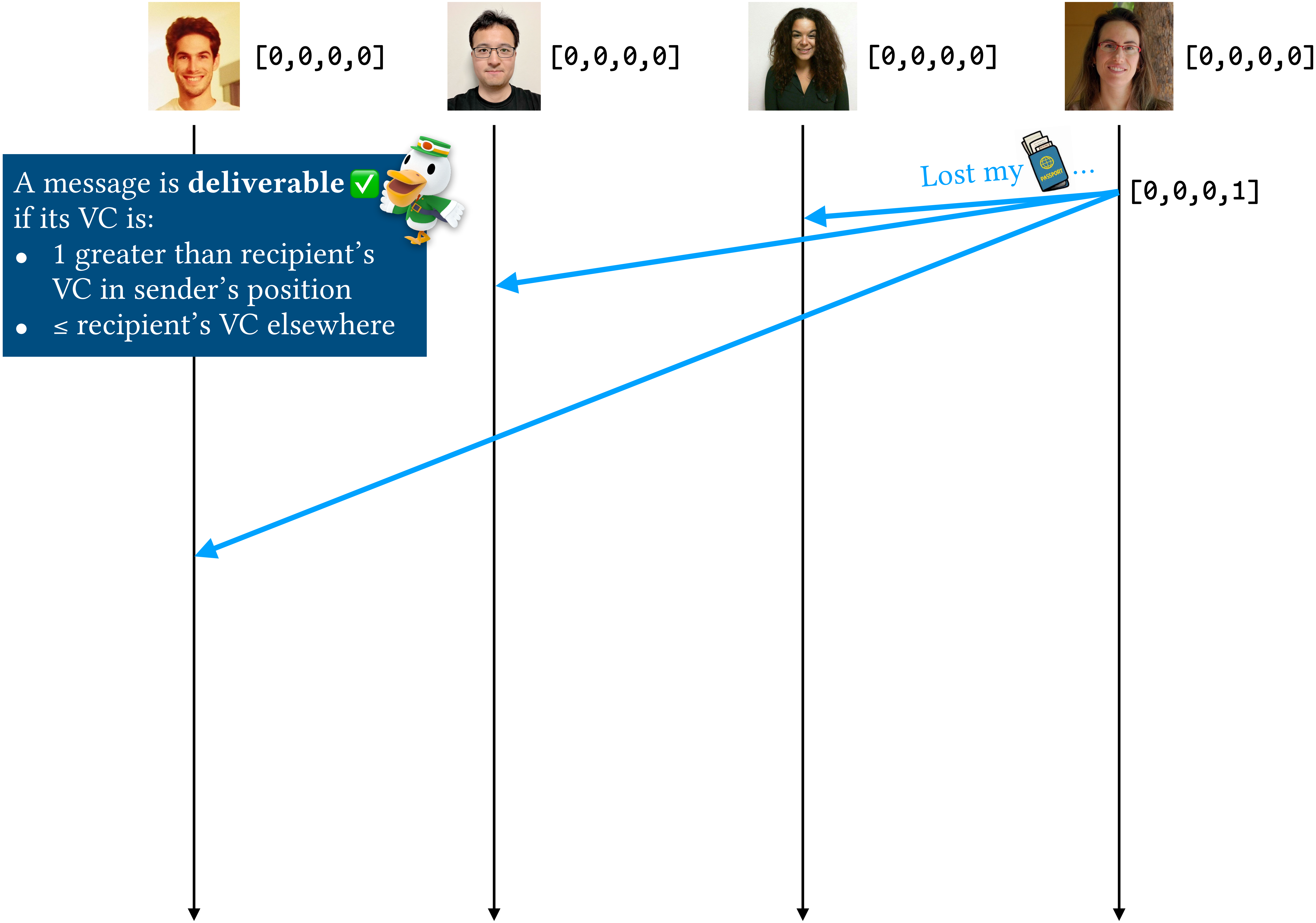
Causal broadcast with vector clocks [Birman et al., 1991]



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



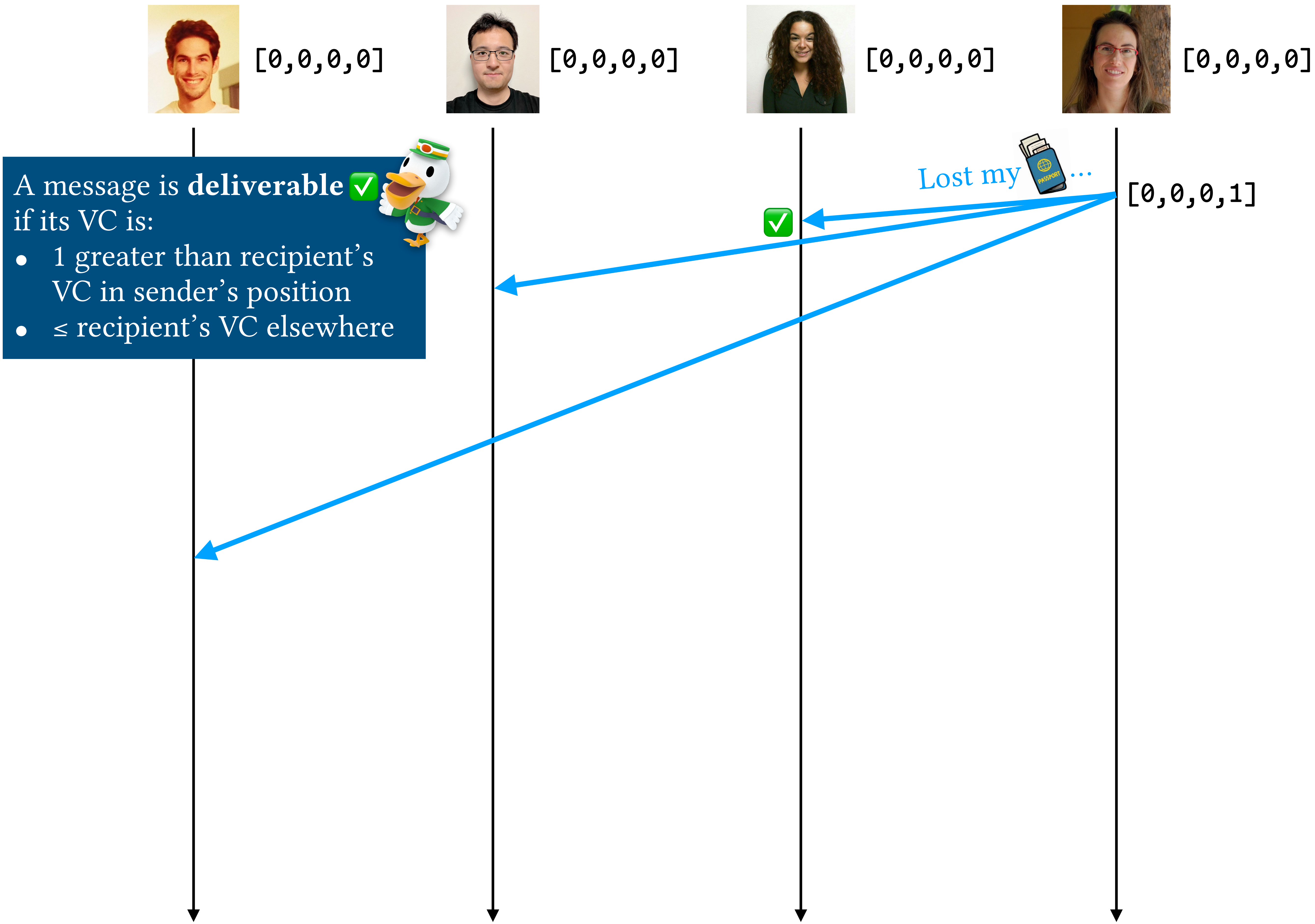
Causal broadcast with vector clocks [Birman et al., 1991]



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

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

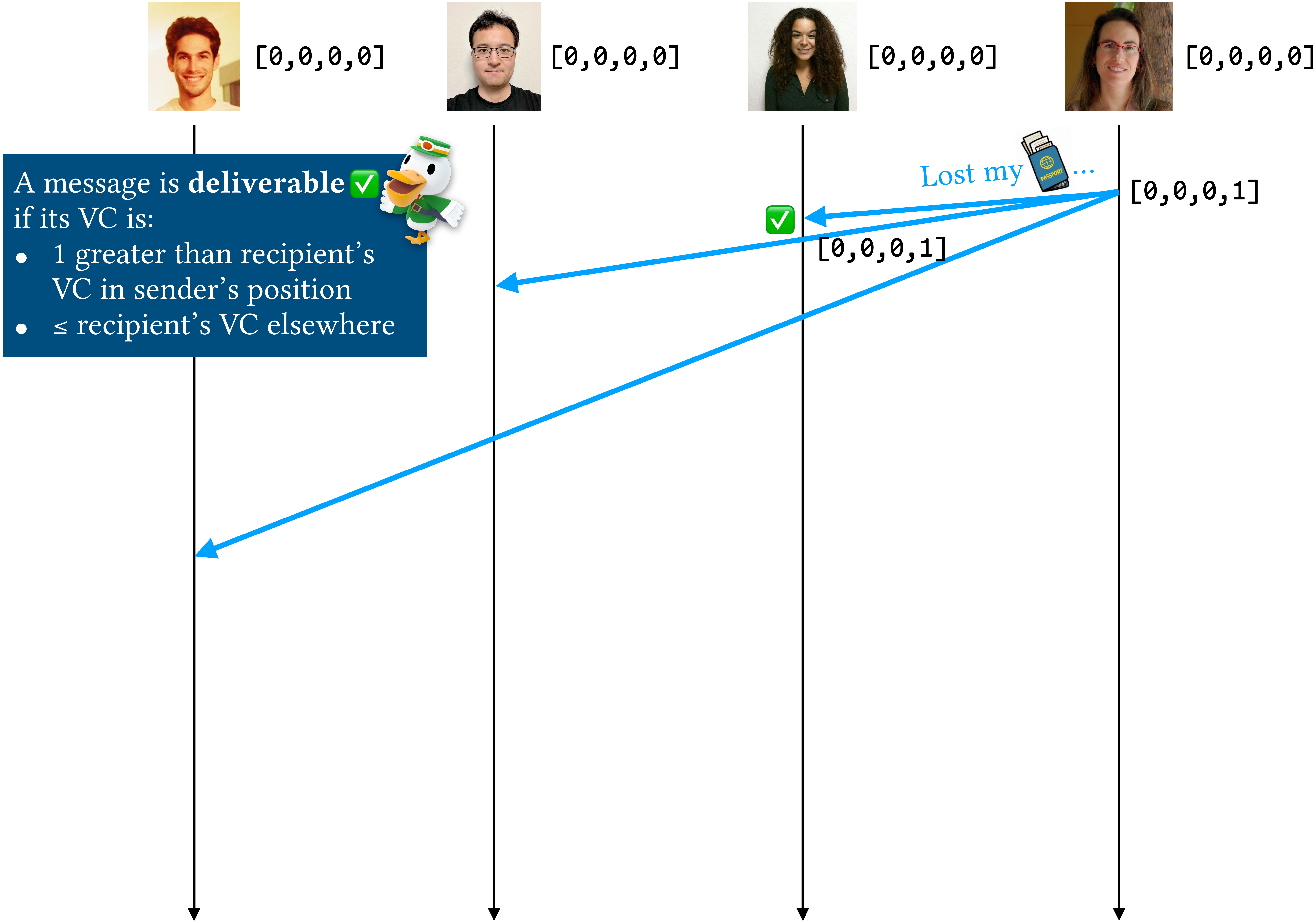
Causal broadcast with vector clocks [Birman et al., 1991]



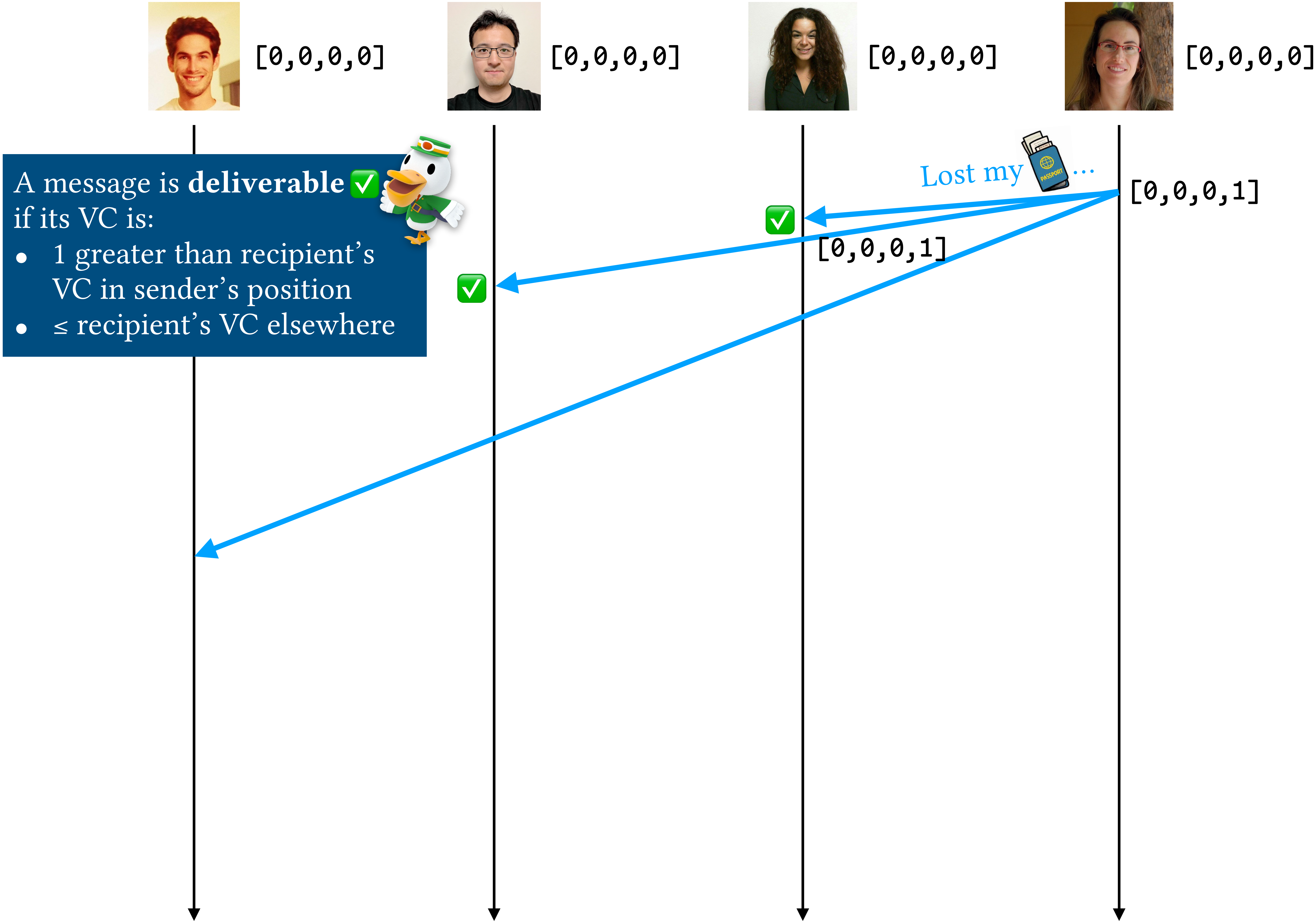
A message is **deliverable** ✅ if its VC is:

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

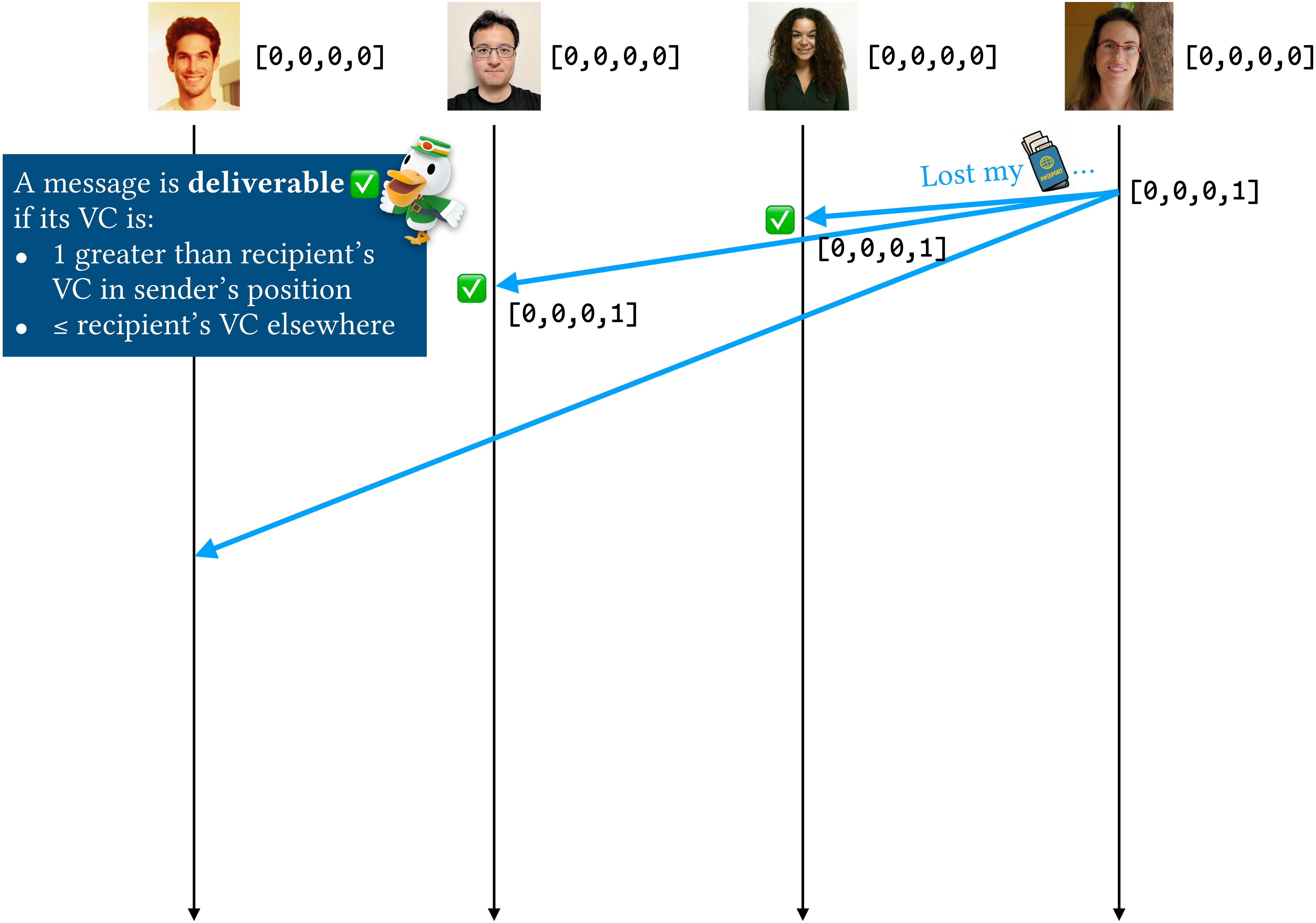
Causal broadcast with vector clocks [Birman et al., 1991]



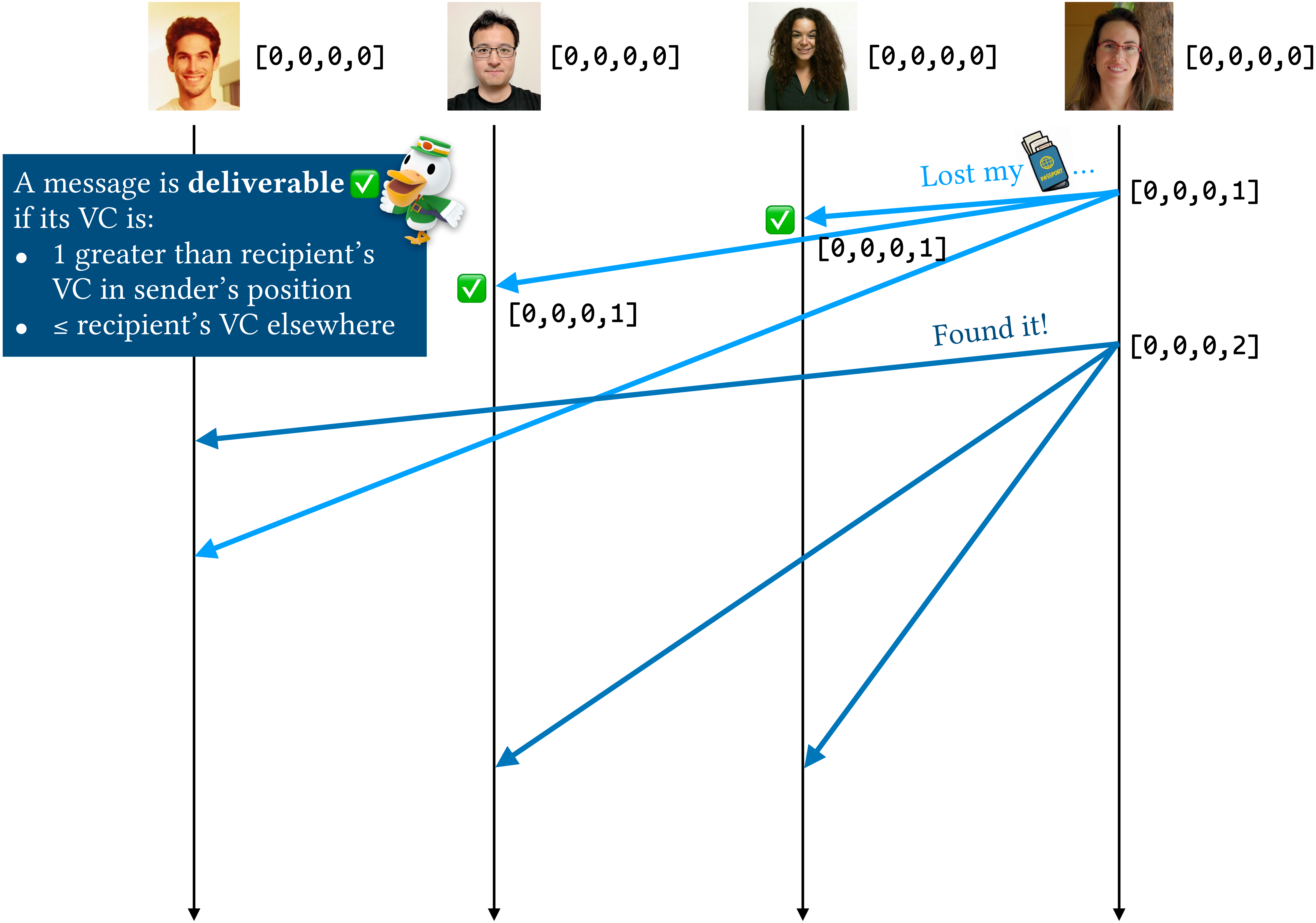
Causal broadcast with vector clocks [Birman et al., 1991]



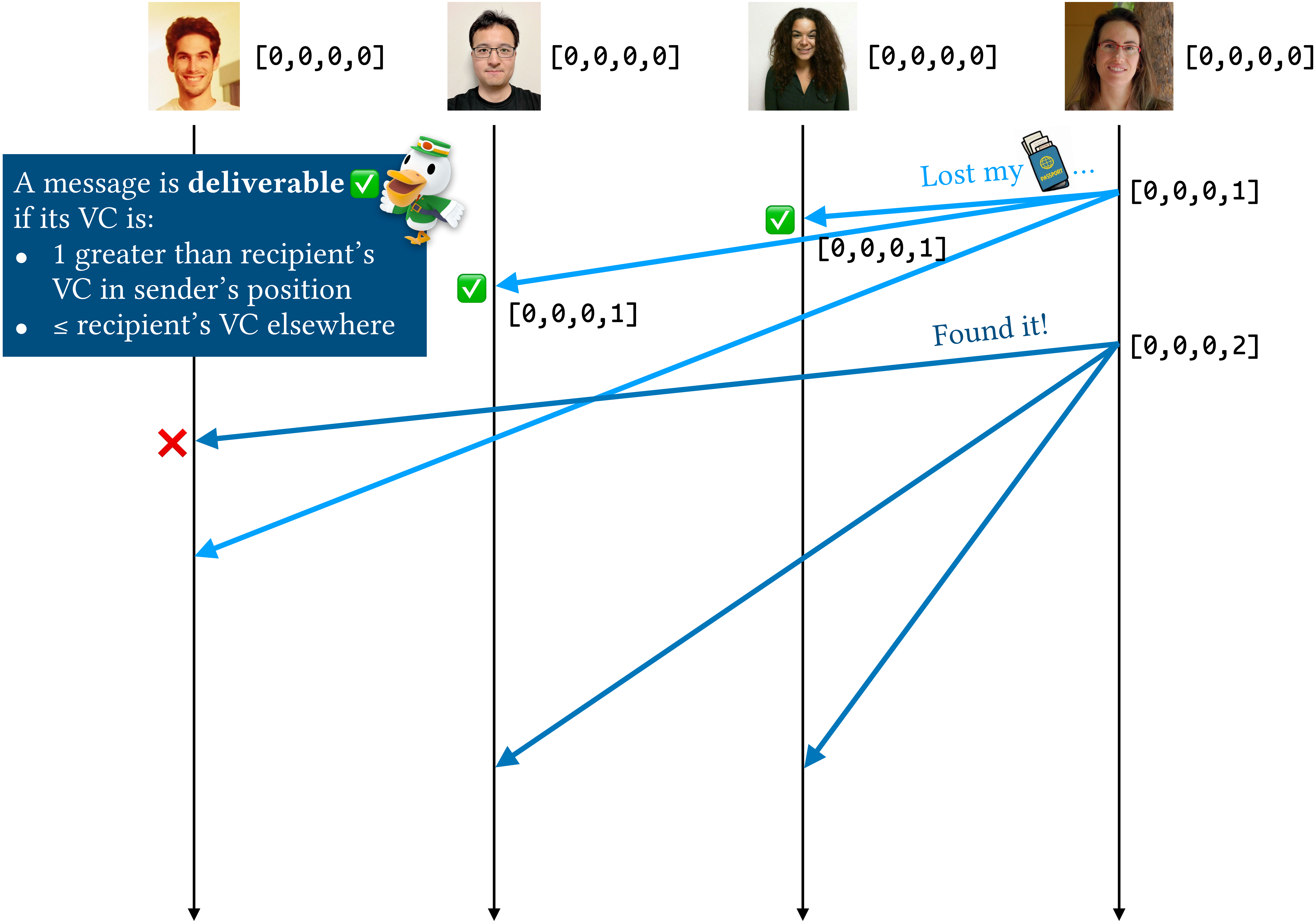
Causal broadcast with vector clocks [Birman et al., 1991]



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



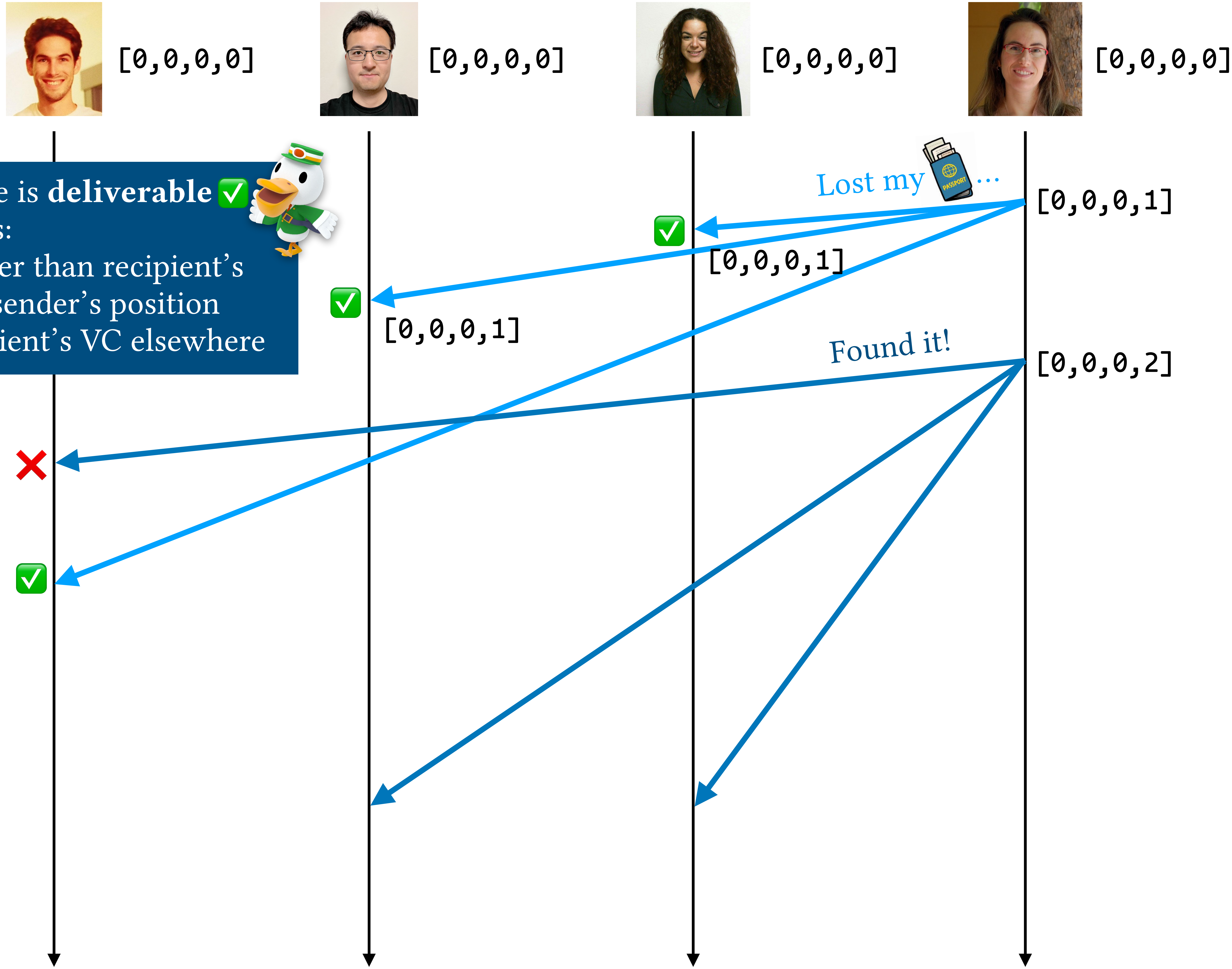
Causal broadcast with vector clocks [Birman et al., 1991]



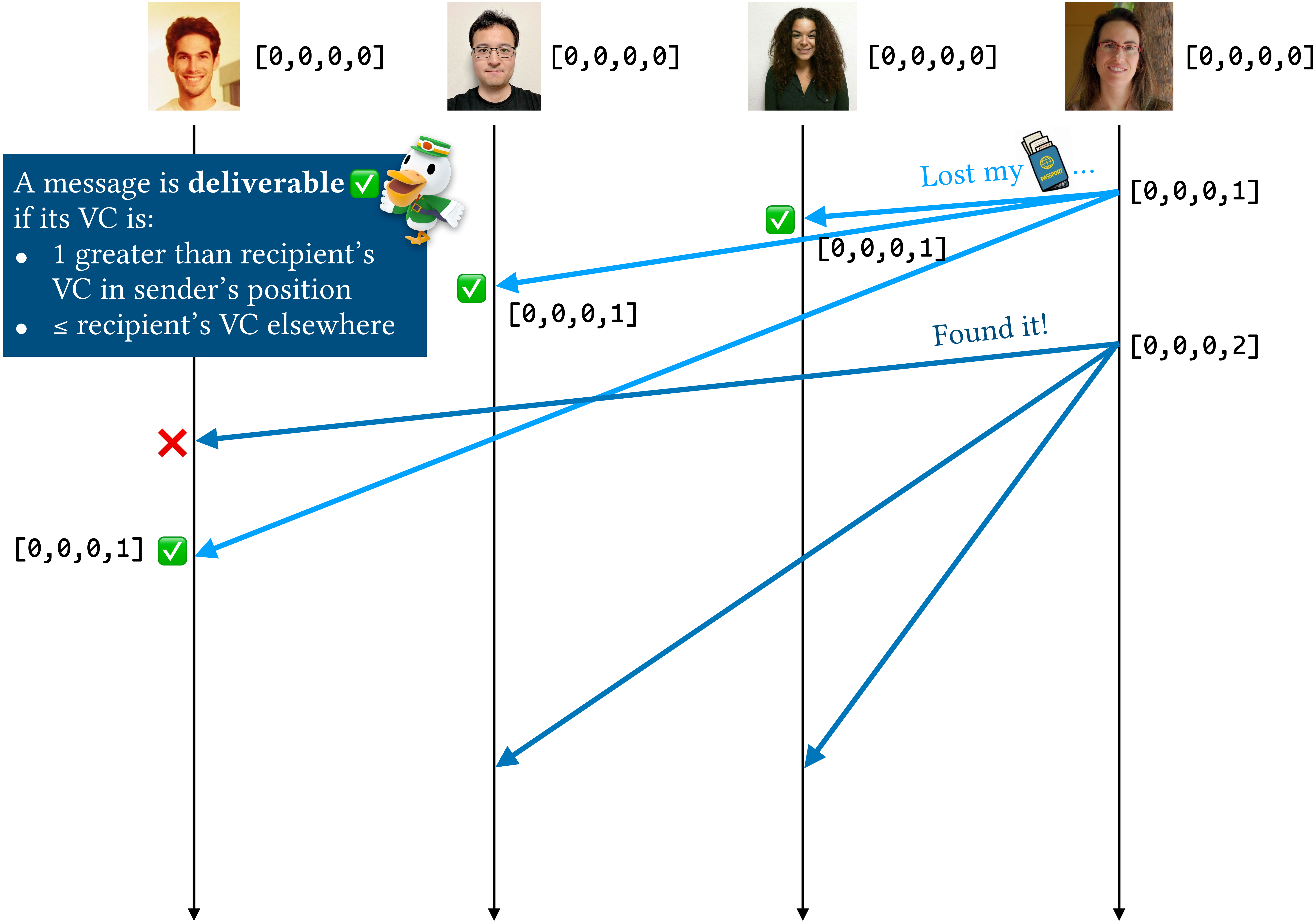
Causal broadcast with vector clocks [Birman et al., 1991]

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

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



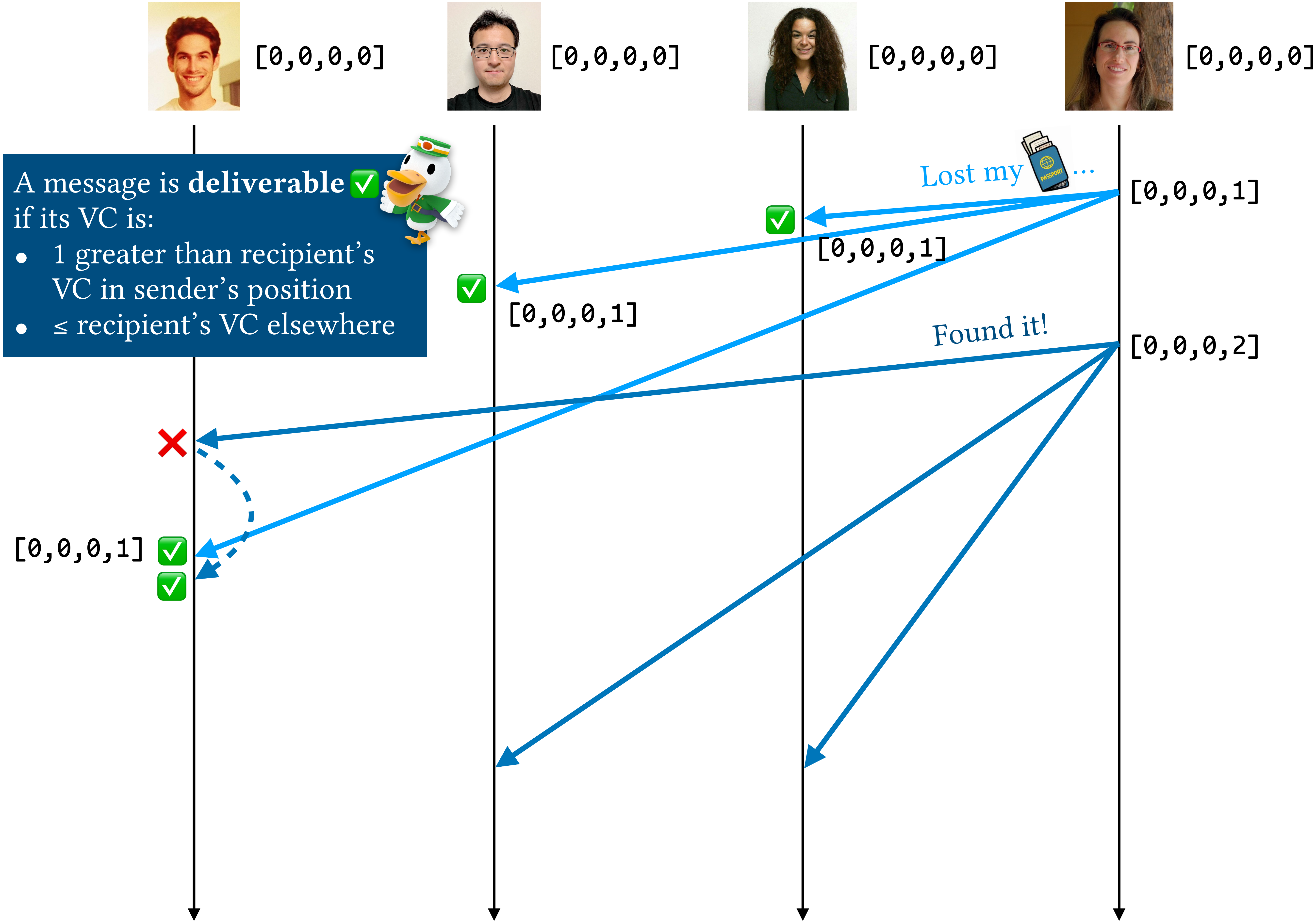
Causal broadcast with vector clocks [Birman et al., 1991]



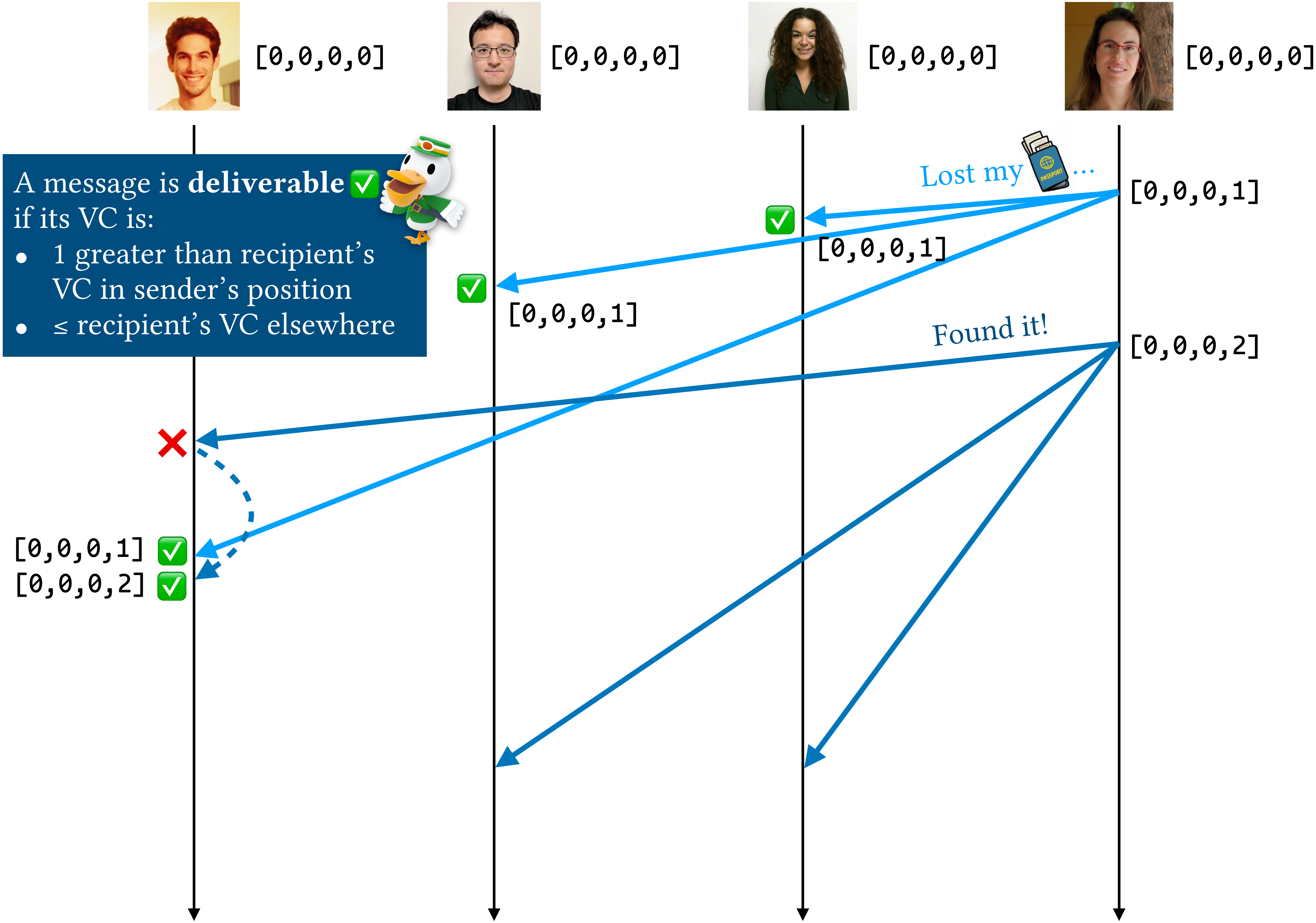
Causal broadcast with vector clocks [Birman et al., 1991]

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

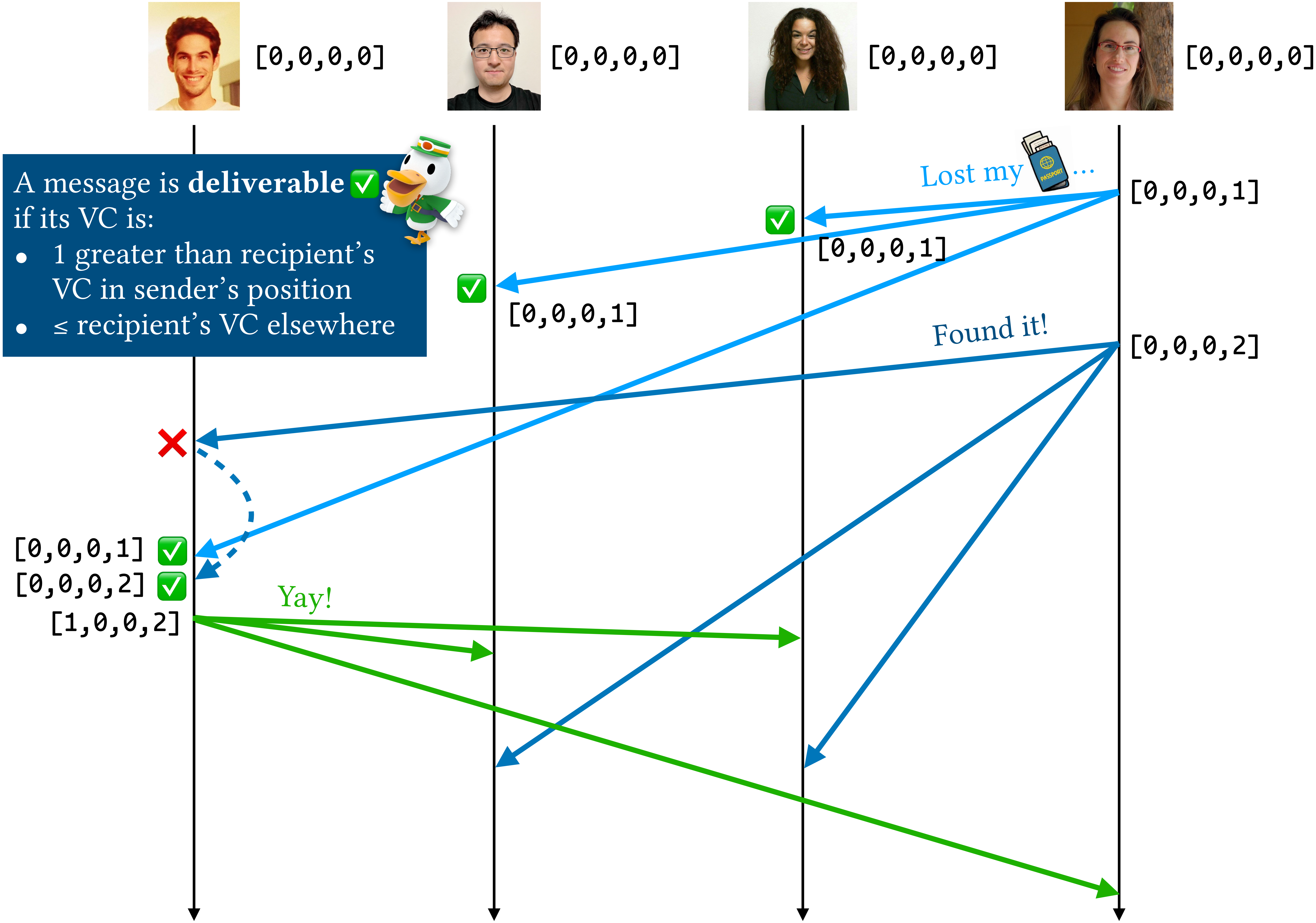
- 1 greater than recipient's VC in sender's position
- ≤ recipient's VC elsewhere



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



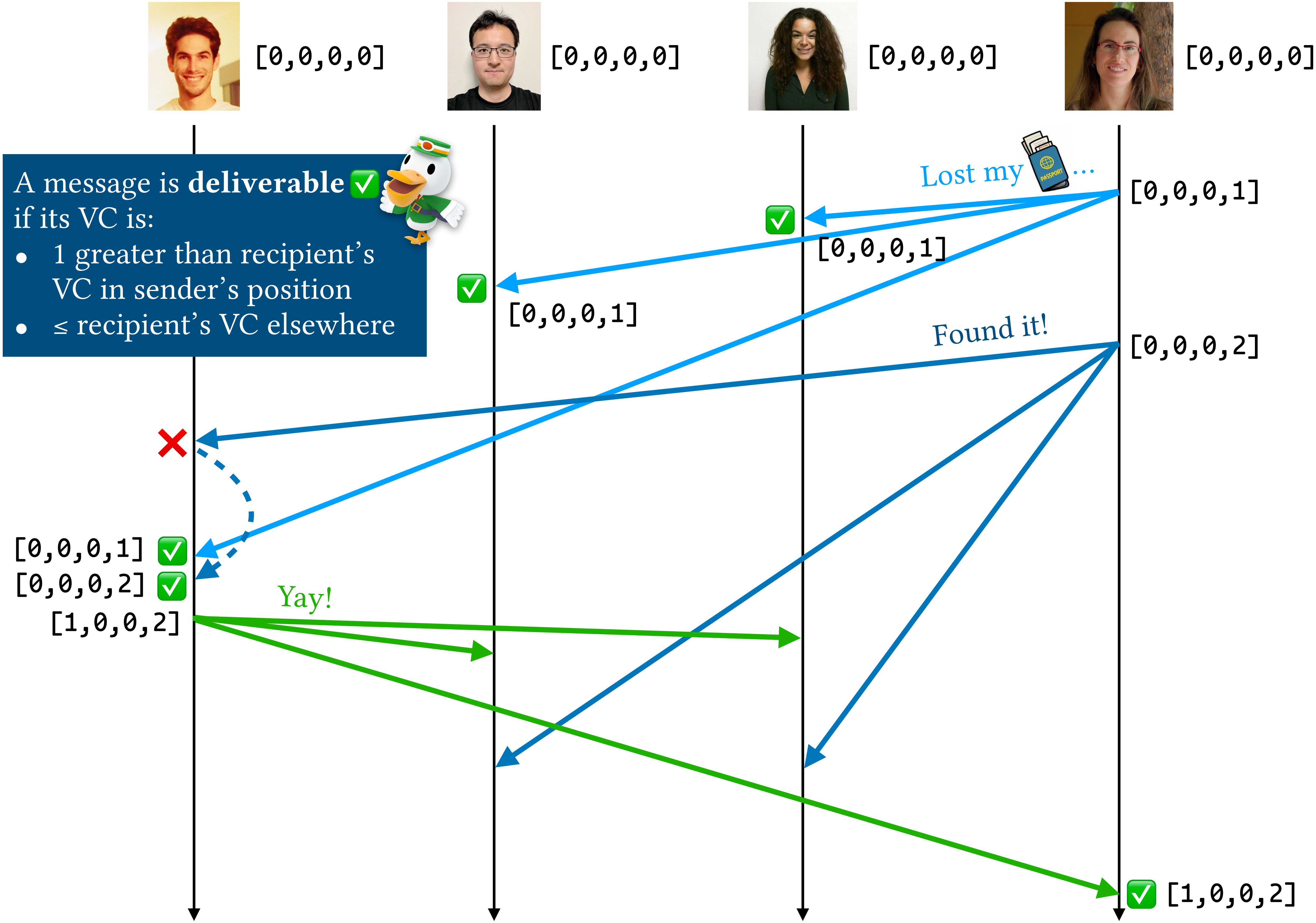
Causal broadcast with vector clocks [Birman et al., 1991]



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

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

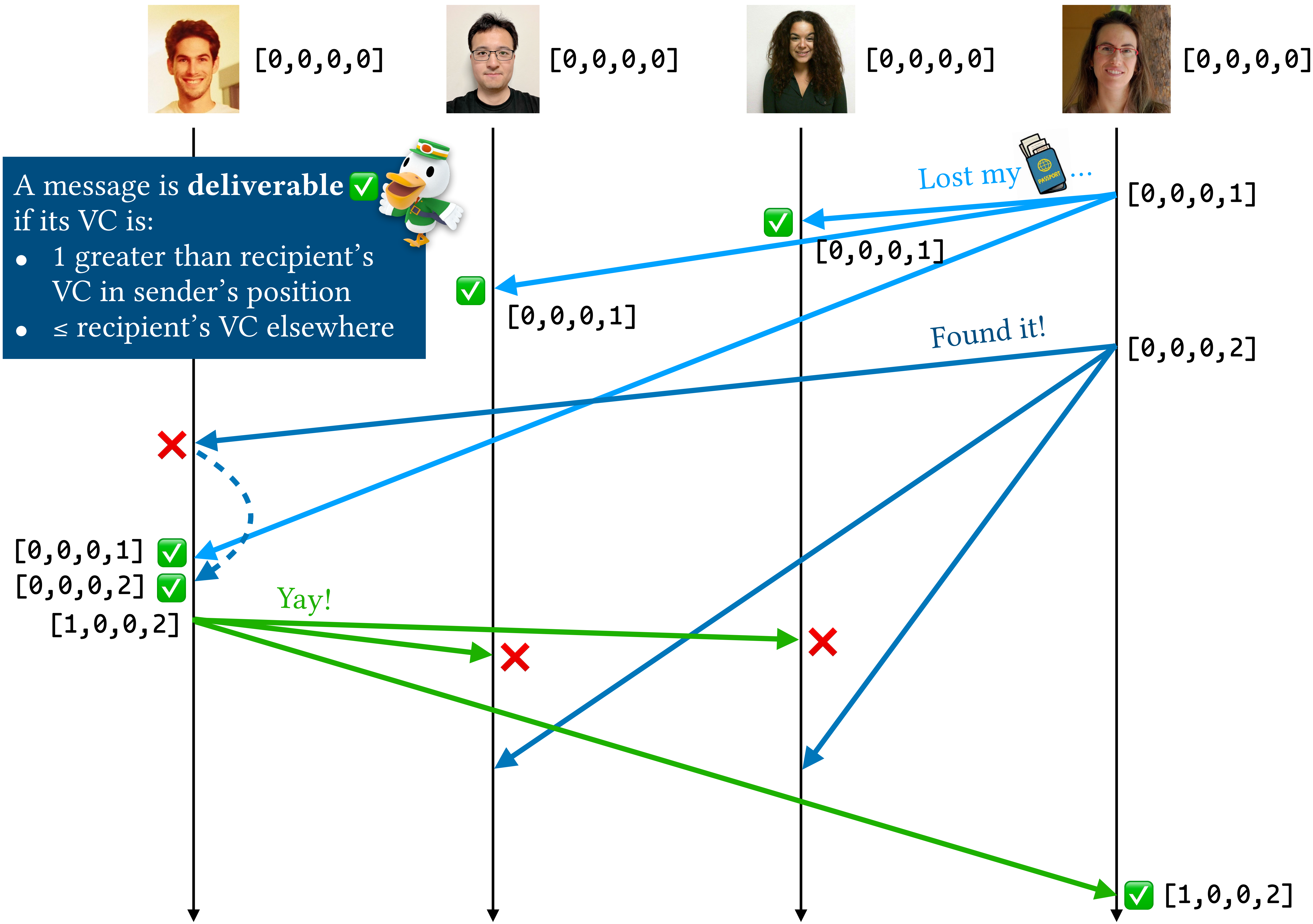
- 1 greater than recipient's VC in sender's position
- ≤ recipient's VC elsewhere



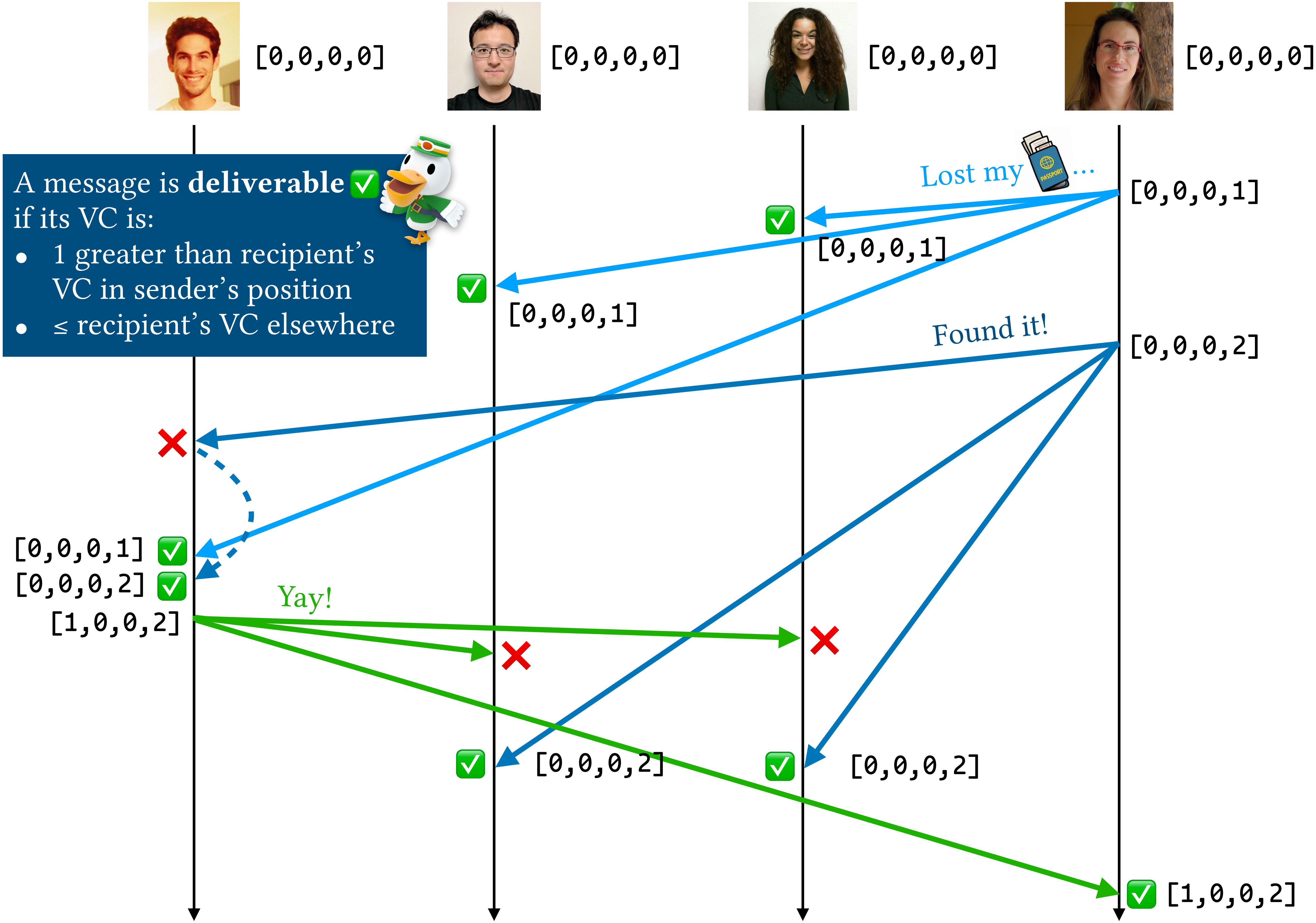
Causal broadcast with vector clocks [Birman et al., 1991]

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

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



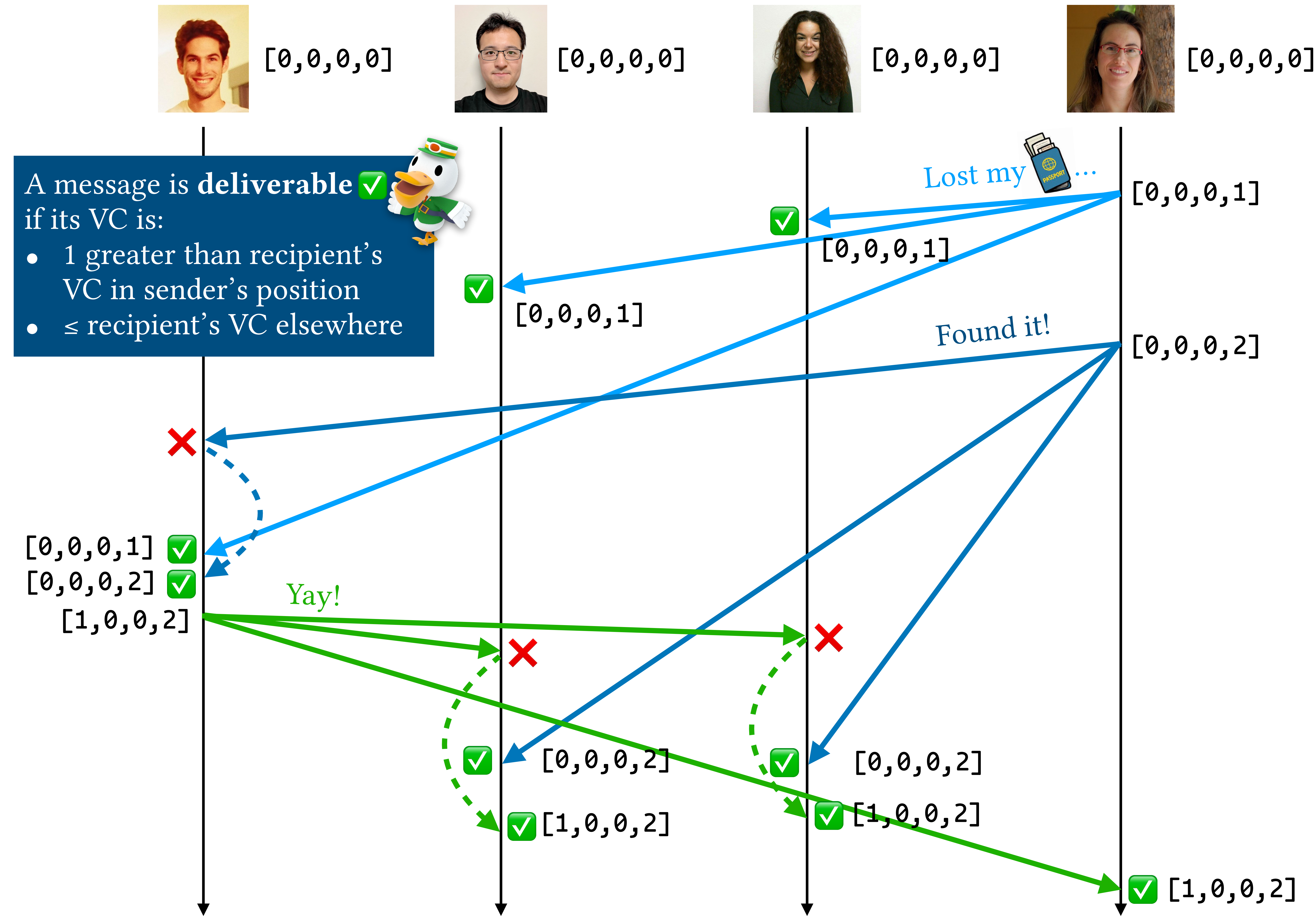
Causal broadcast with vector clocks [Birman et al., 1991]



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

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]

```
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

```
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

```haskell
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
```

```
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
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
```

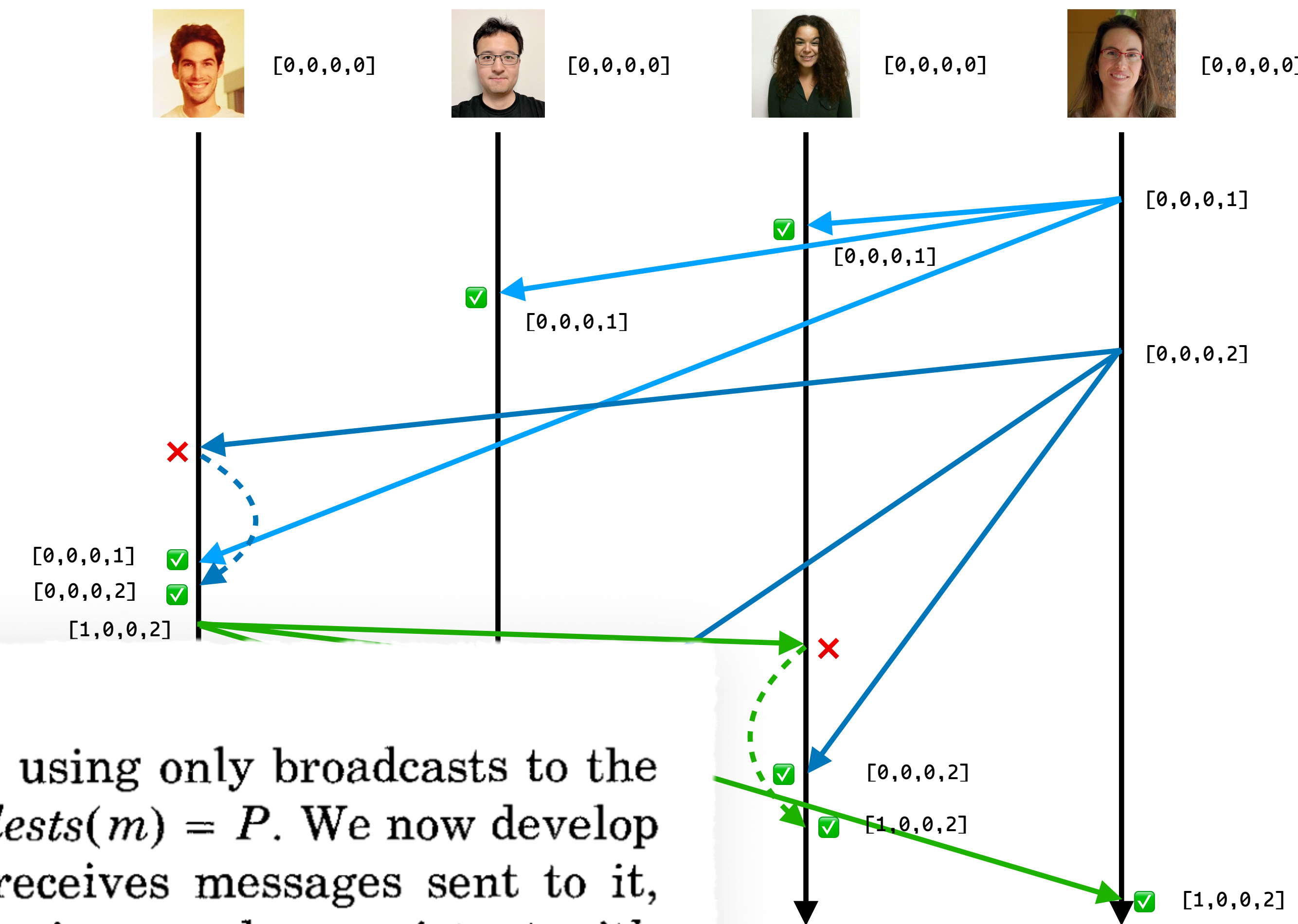
application code

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*

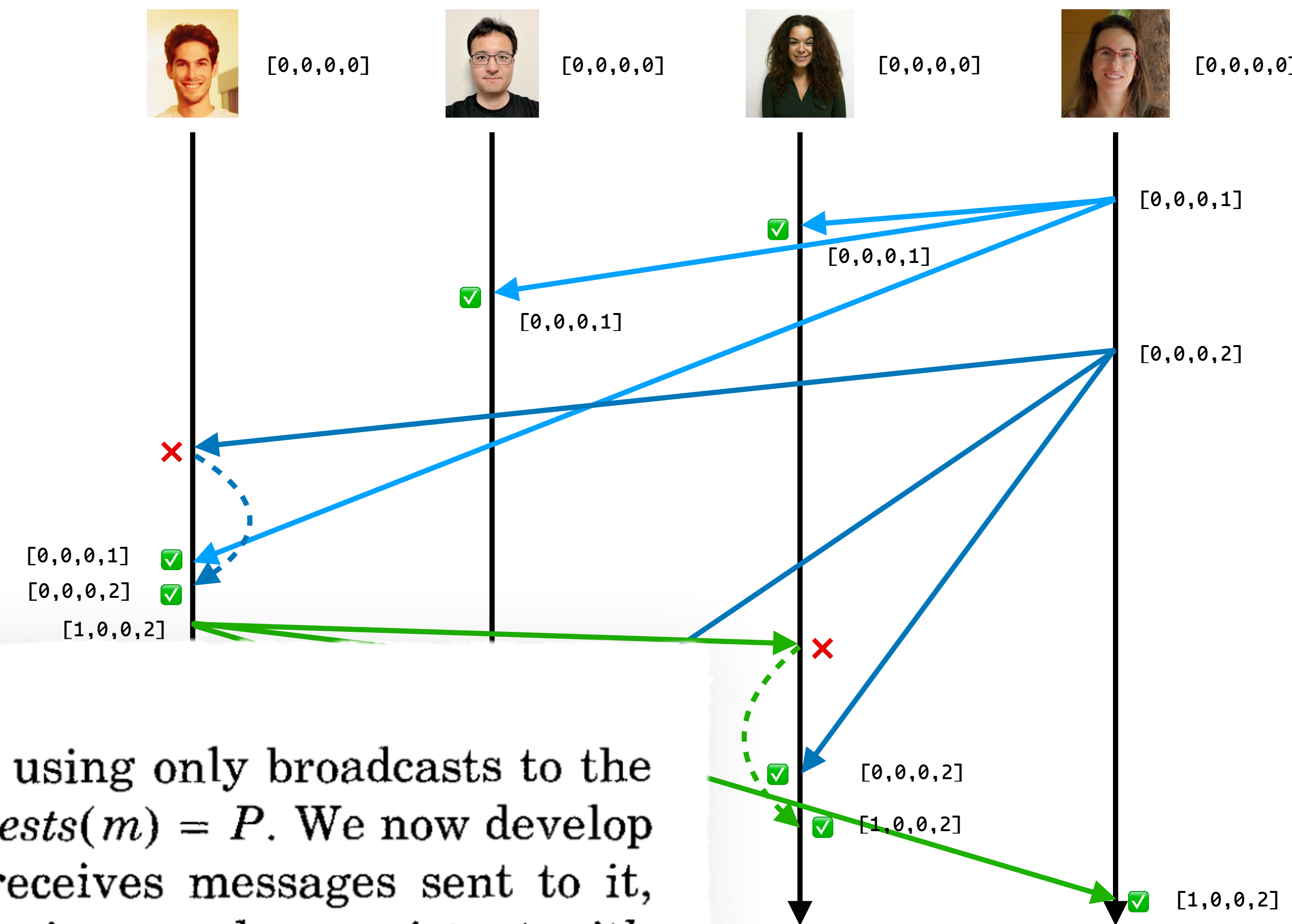


5.1 CBCAST Protocol

Suppose that a set of processes P communicate using only broadcasts to the full set of processes in the system; that is, $\forall m$: *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:

$$m \rightarrow m' \Rightarrow \forall p\colon \mathit{deliver}_p(m) \xrightarrow{p} \mathit{deliver}_p(m').$$

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, $\forall m$: $dests(m) = P$. 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:

$$m \rightarrow m' \Rightarrow \forall p\colon deliver_p(m) \xrightarrow{p} deliver_p(m').$$

Birman et al., "Lightweight Causal and Atomic Group Multicast"
ACM *TOCS*, 1991

```
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') }
```

```
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') }

data Op = OpReceive (Message) | OpDeliver | OpBroadcast

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') }

data Op = OpReceive (Message) | OpDeliver | OpBroadcast

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') }

data Op = OpReceive (Message) | OpDeliver | OpBroadcast

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]

Toward Hole-Driven Development in Liquid Haskell

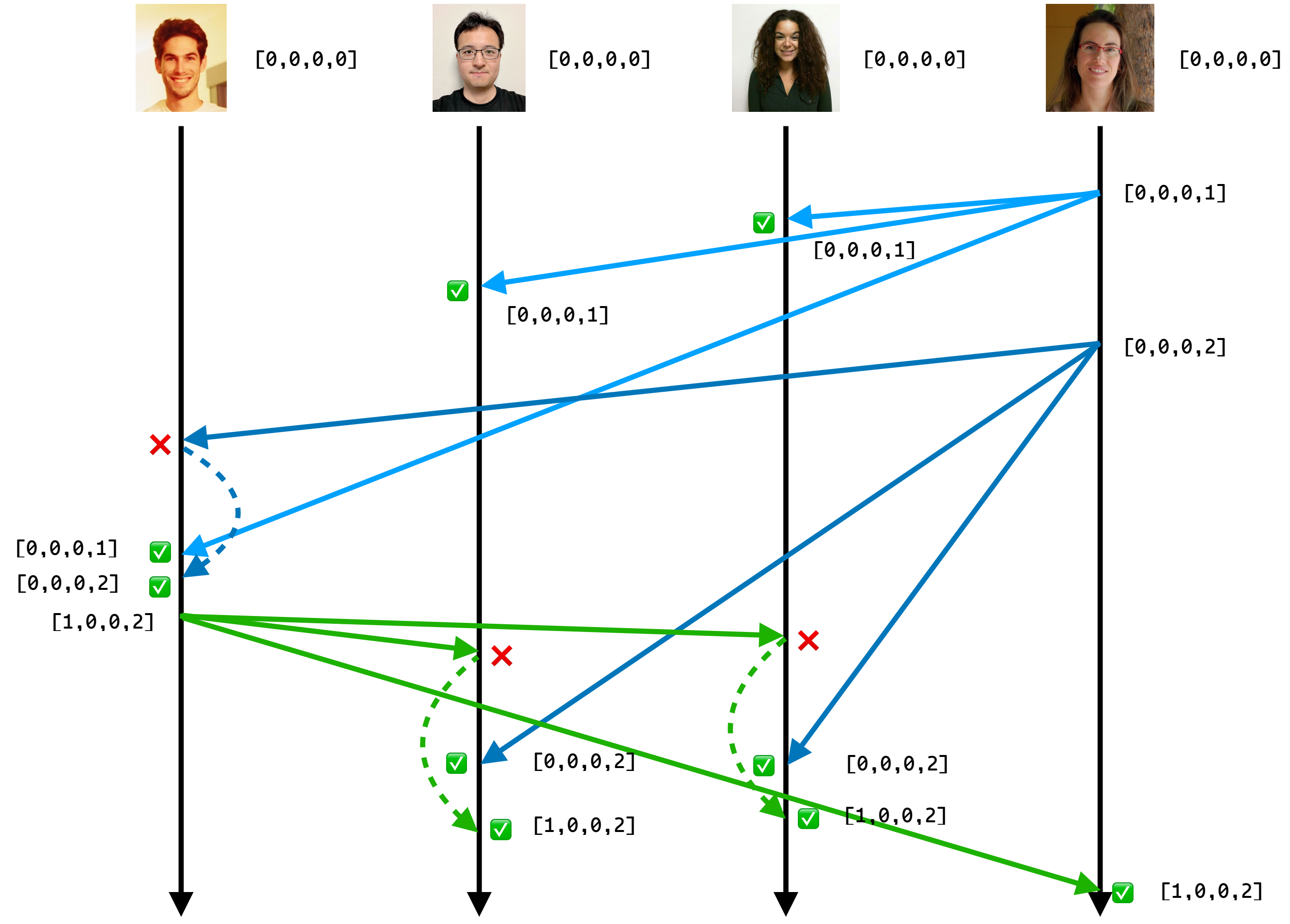
PATRICK REDMOND, University of California, Santa Cruz, USA
GAN SHEN, University of California, Santa Cruz, USA
LINDSEY KUPER, University of California, Santa Cruz, USA

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

Tak!

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



github.com/lsd-ucsc/cbcast-lh