

Adventures in Building Reliable Distributed Systems with Liquid Haskell

Lindsey Kuper
University of California, Santa Cruz
FLOPS 2022 Keynote
May 10, 2022

Adventures in Building Reliable Distributed Systems with Liquid Haskell

Lindsey Kuper
University of California, Santa Cruz
FLOPS 2022 Keynote
May 10, 2022

Adventures in Building Reliable Distributed Systems with Liquid Haskell

Lindsey Kuper
University of California, Santa Cruz
FLOPS 2022 Keynote
May 10, 2022

Adventures in Building Reliable Distributed Systems with Liquid Haskell

Lindsey Kuper
University of California, Santa Cruz
FLOPS 2022 Keynote
May 10, 2022







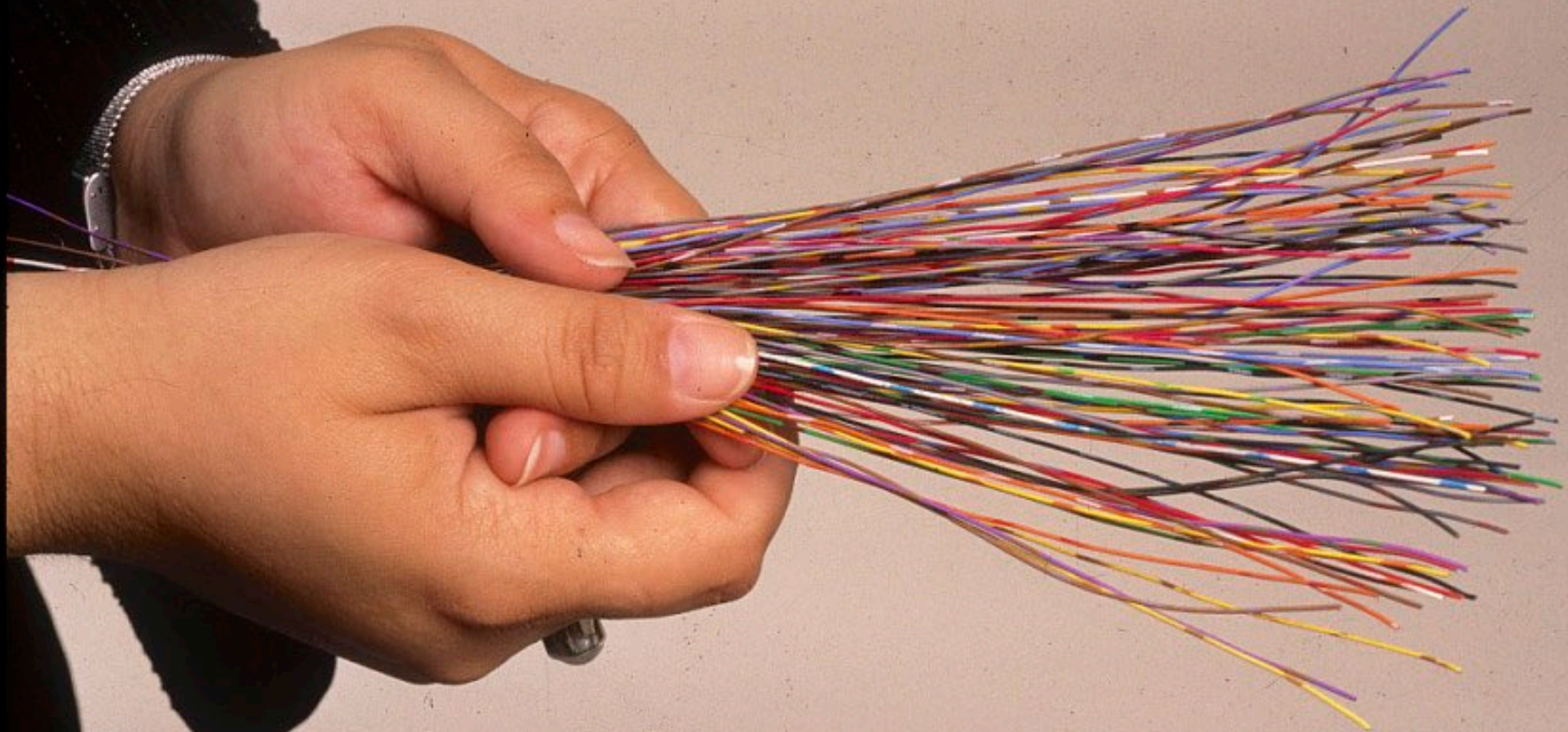


“Everything fails, all the time.”

— Werner Vogels, Amazon CTO

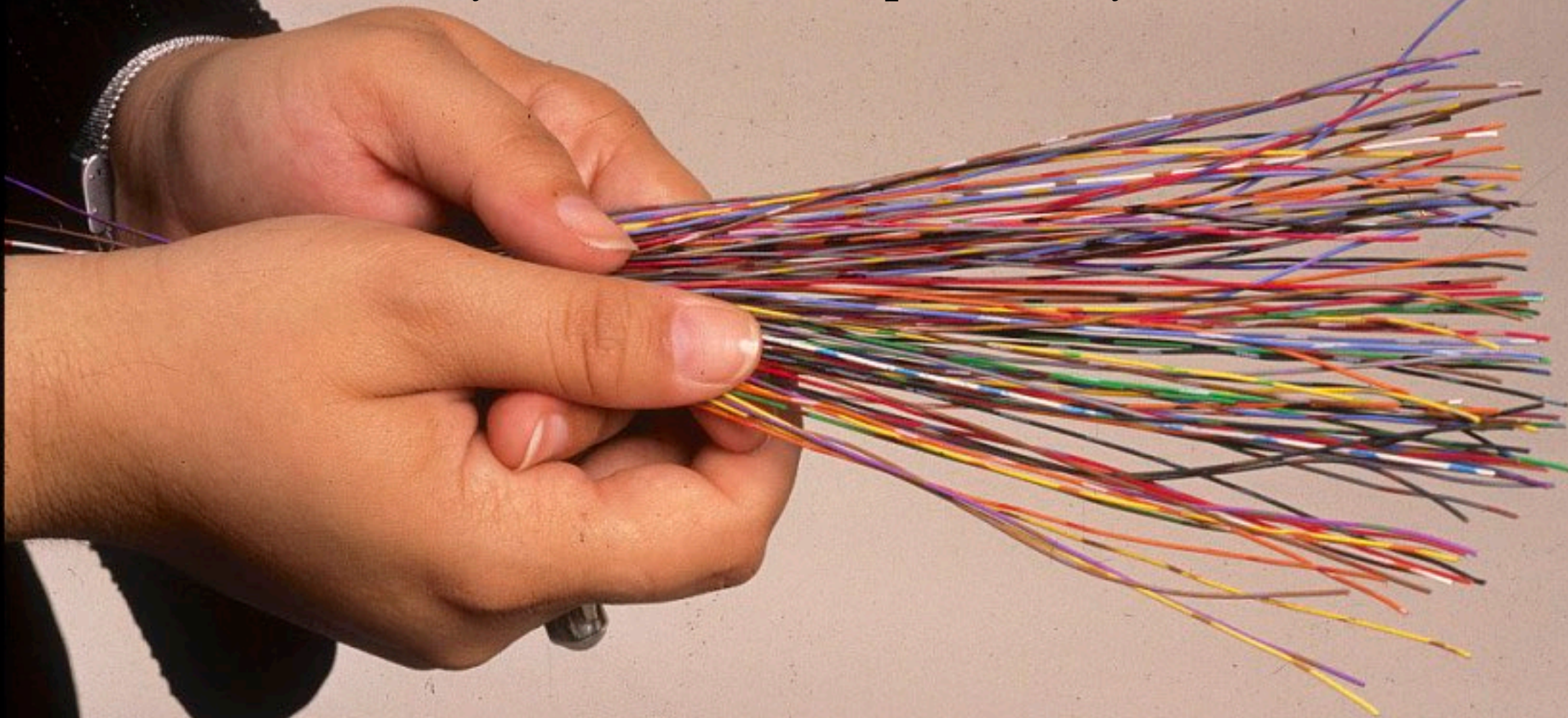


The speed of light is slow.



The speed of light is slow.

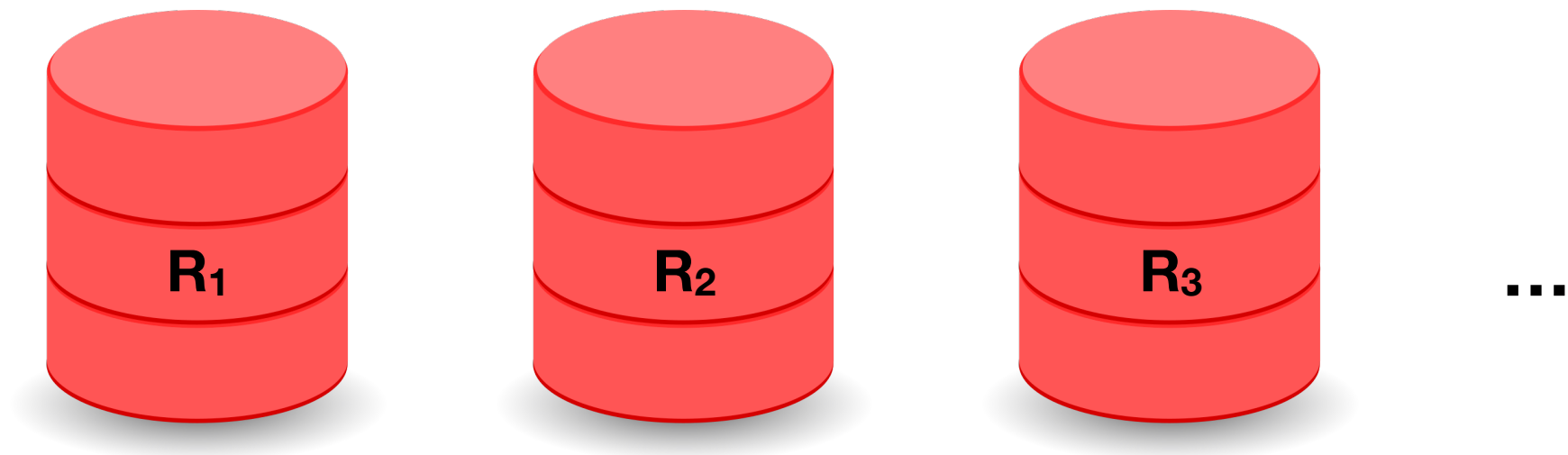
“It’s only about four inches per clock cycle.” — Mae Milano





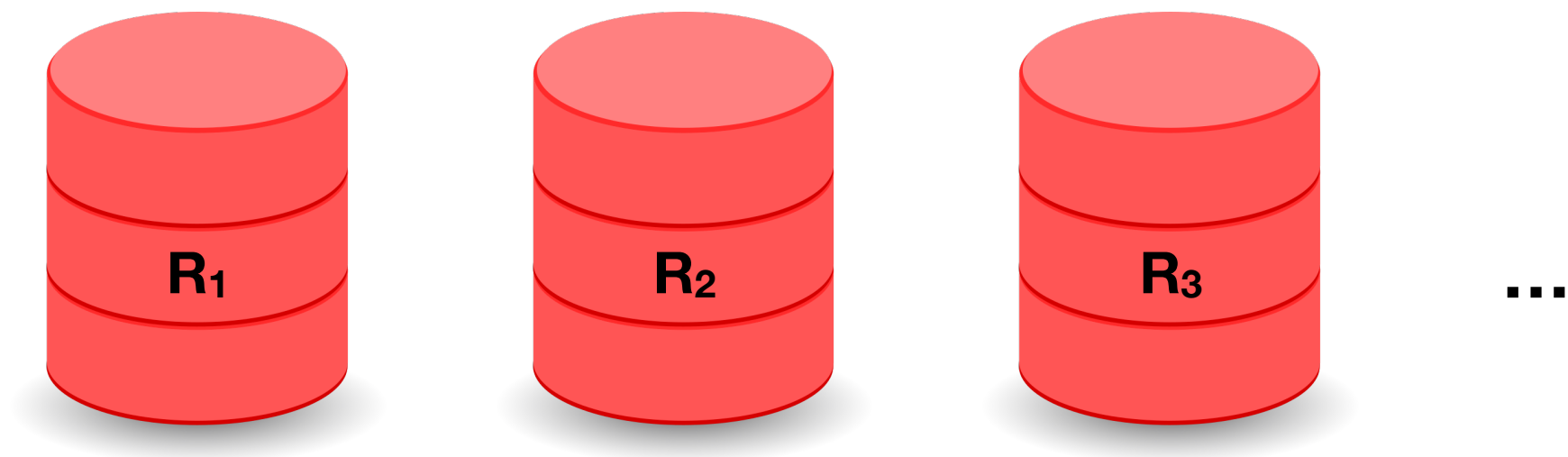


REPLICATION

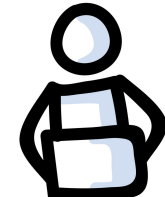
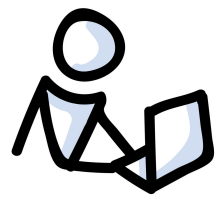


the cause of, and solution to,
most distributed systems problems

REPLICATION



the cause of, and solution to,
most distributed systems problems

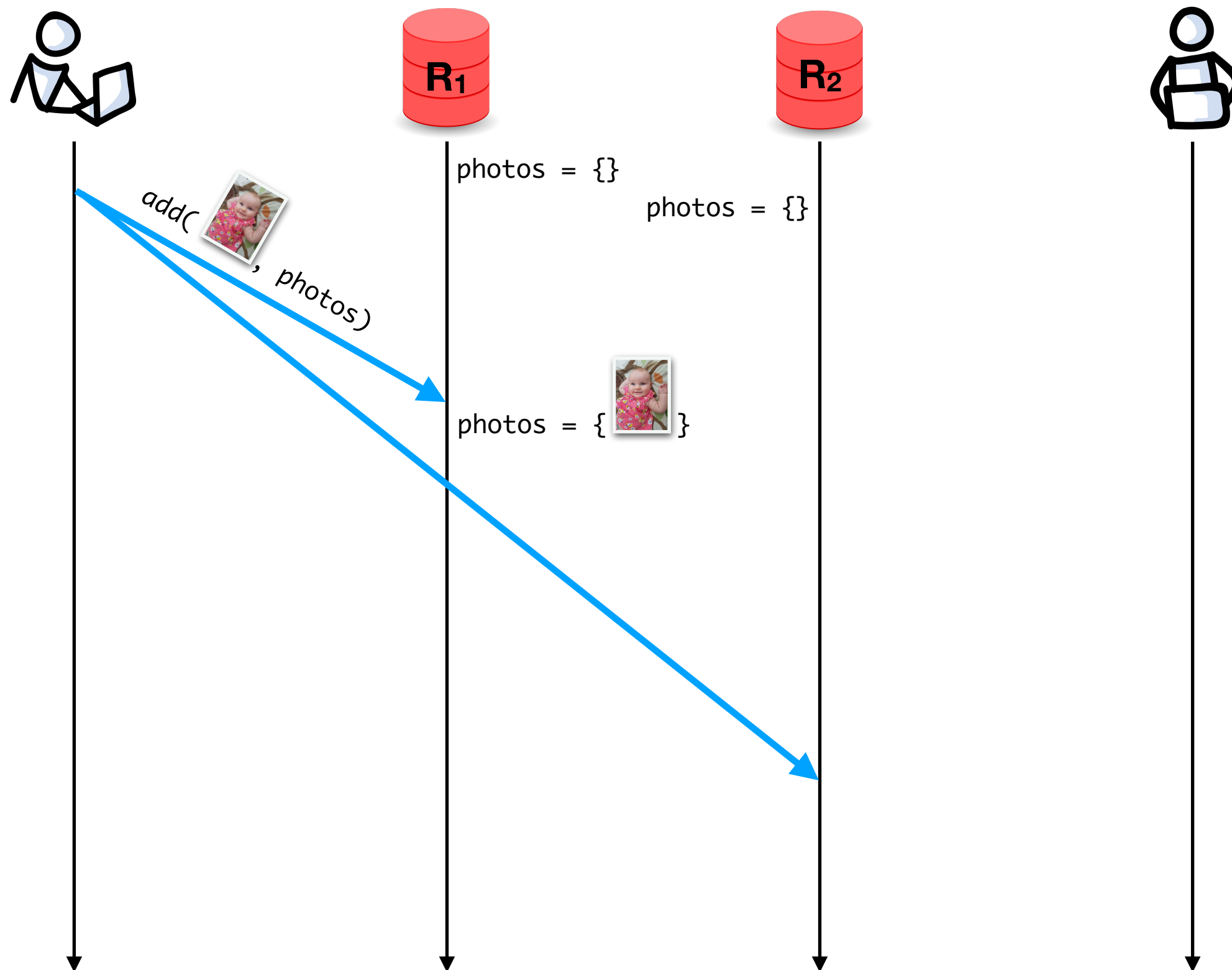


photos = {}

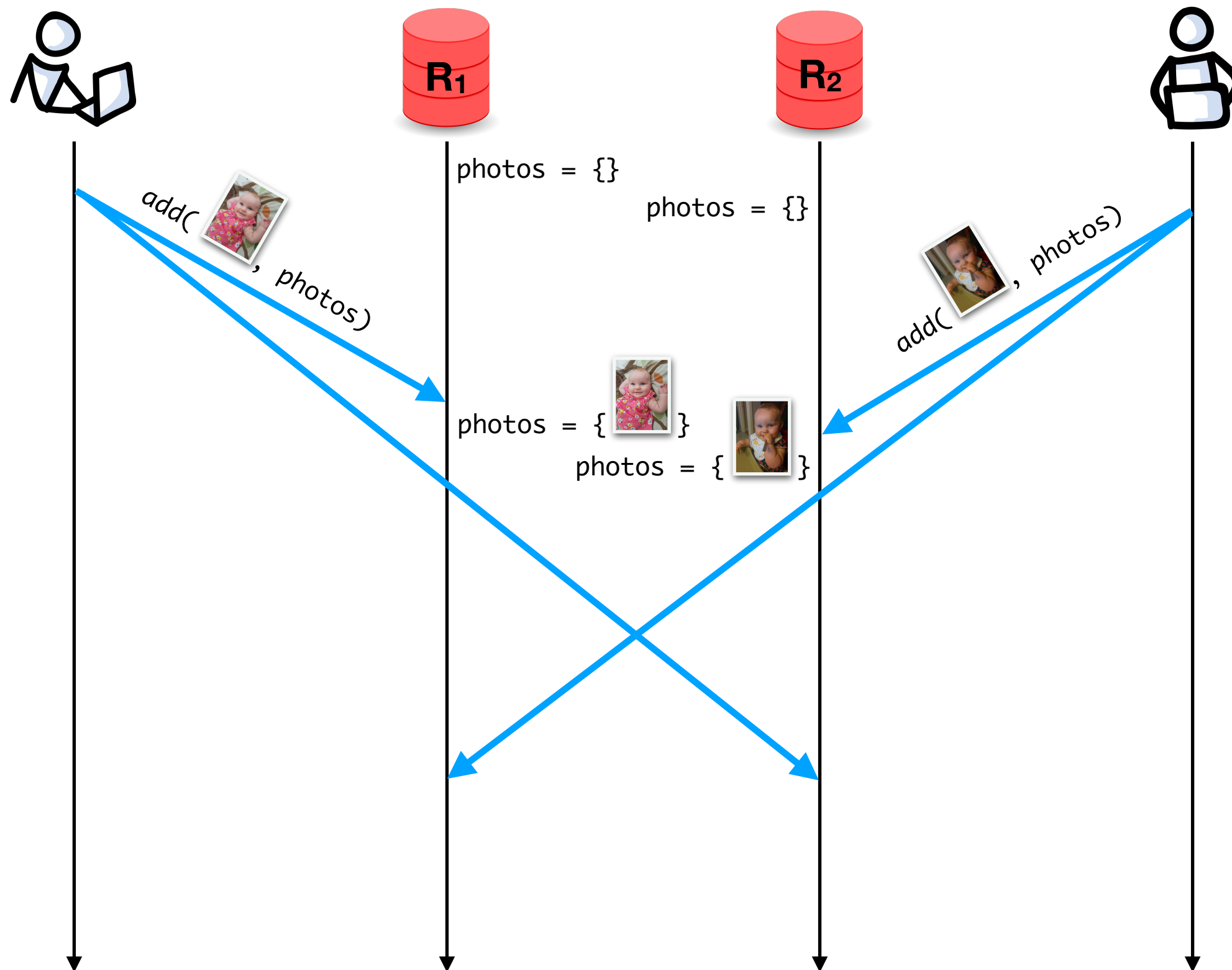
photos = {}

Strong convergence

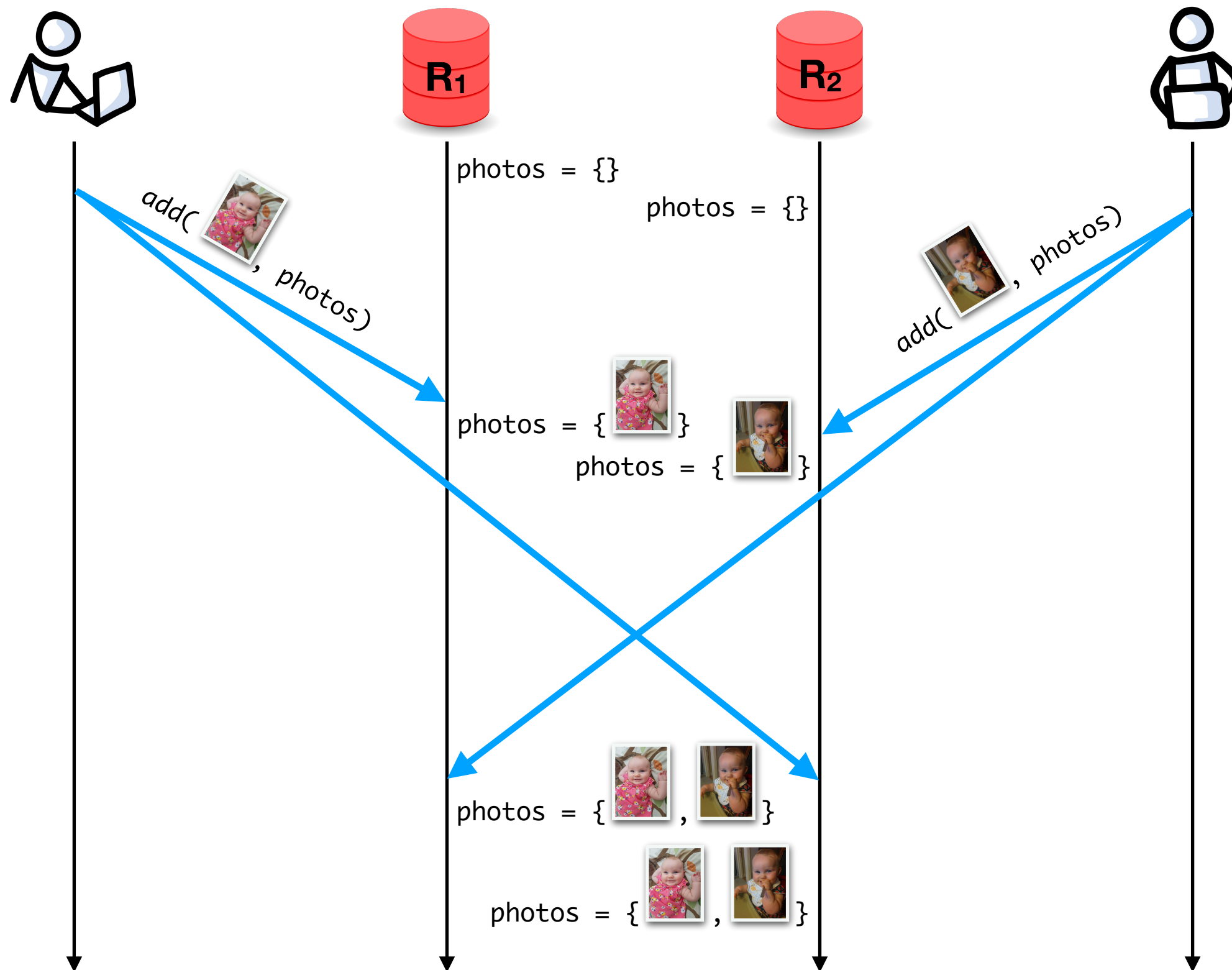
[Shapiro et al., 2011]



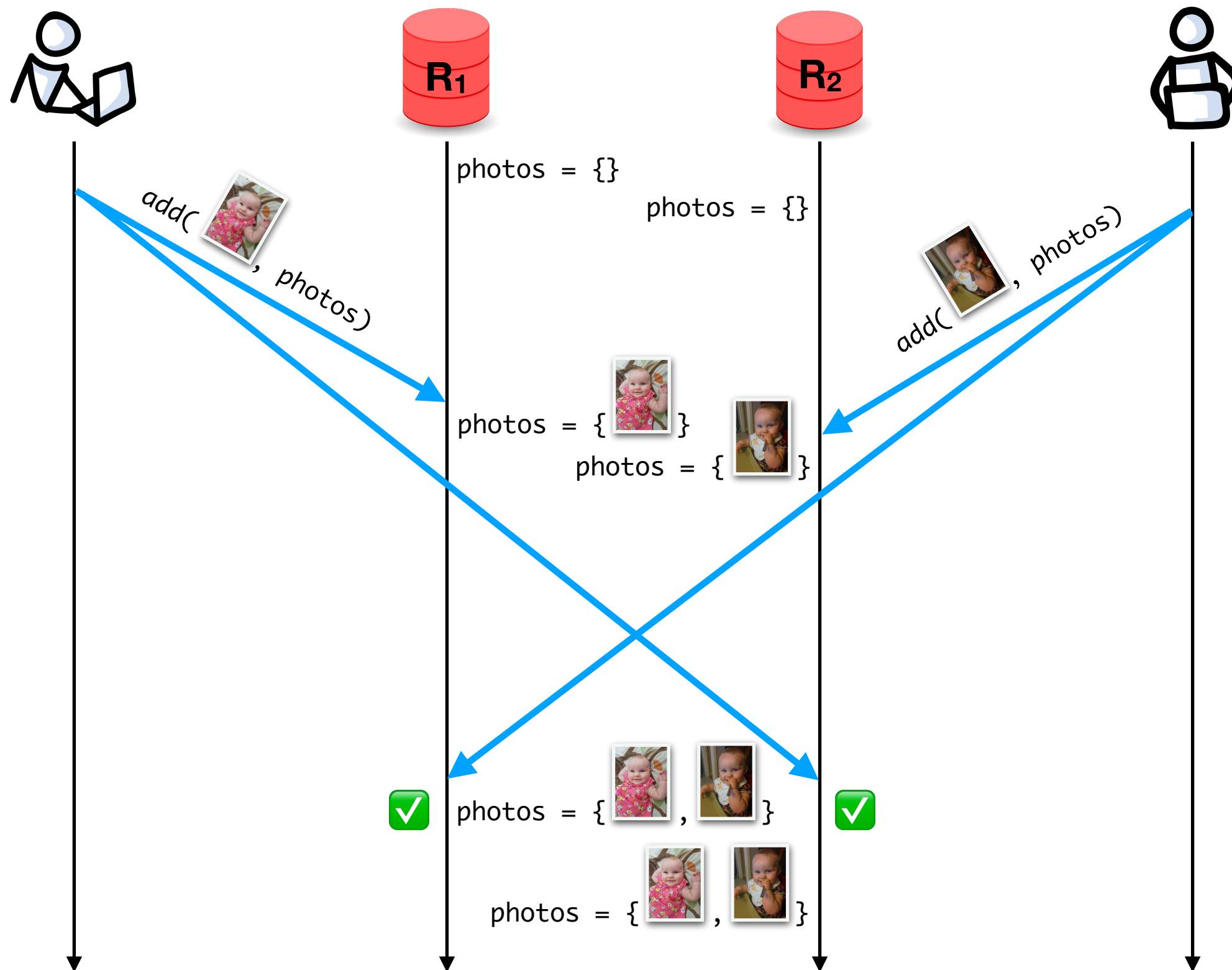
Strong convergence
[Shapiro et al., 2011]



Strong convergence
[Shapiro et al., 2011]

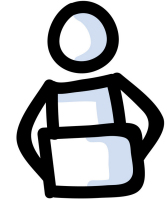
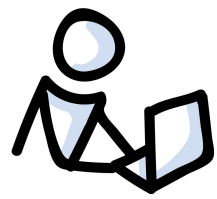



Strong convergence
[Shapiro et al., 2011]




Strong convergence

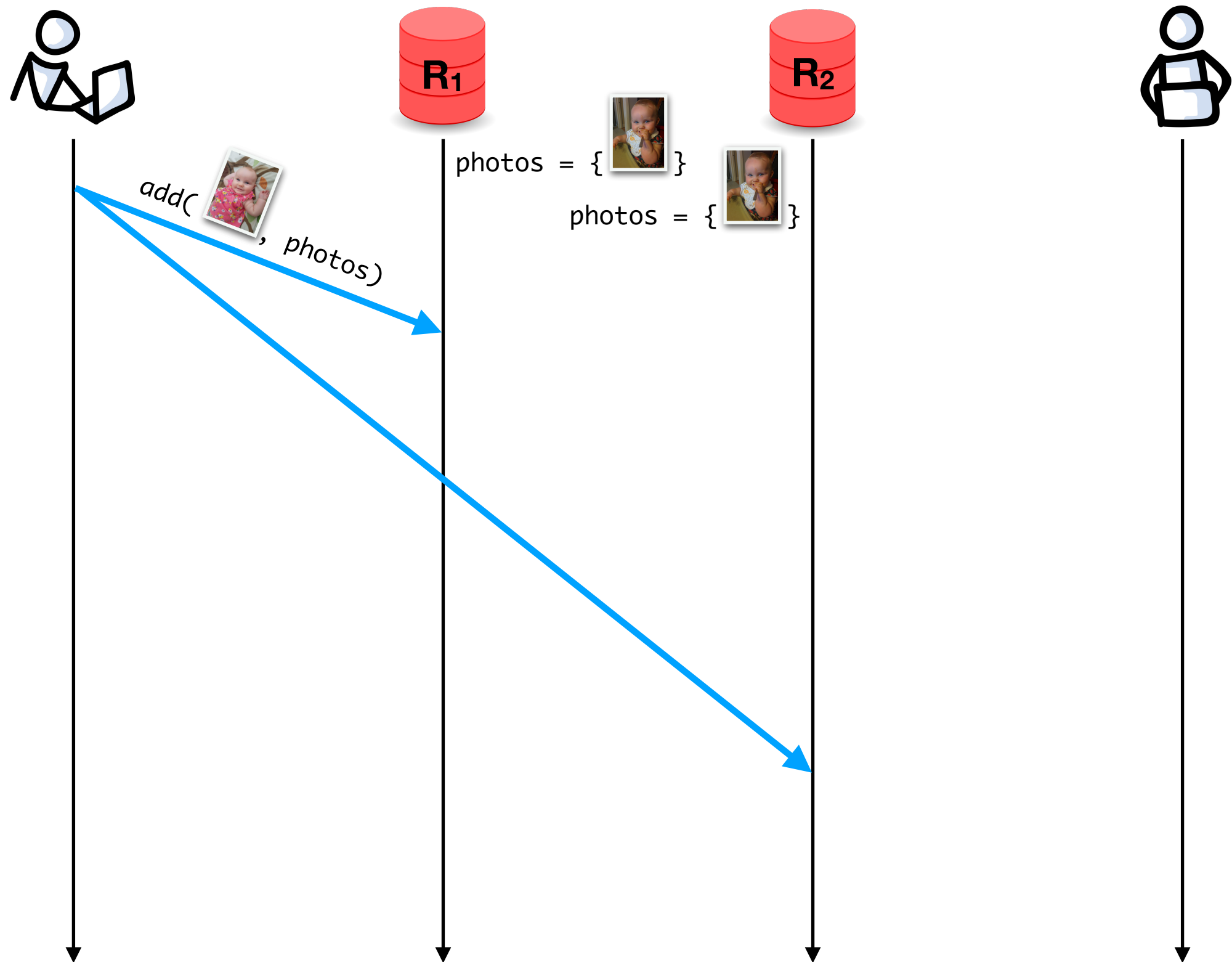
[Shapiro et al., 2011]

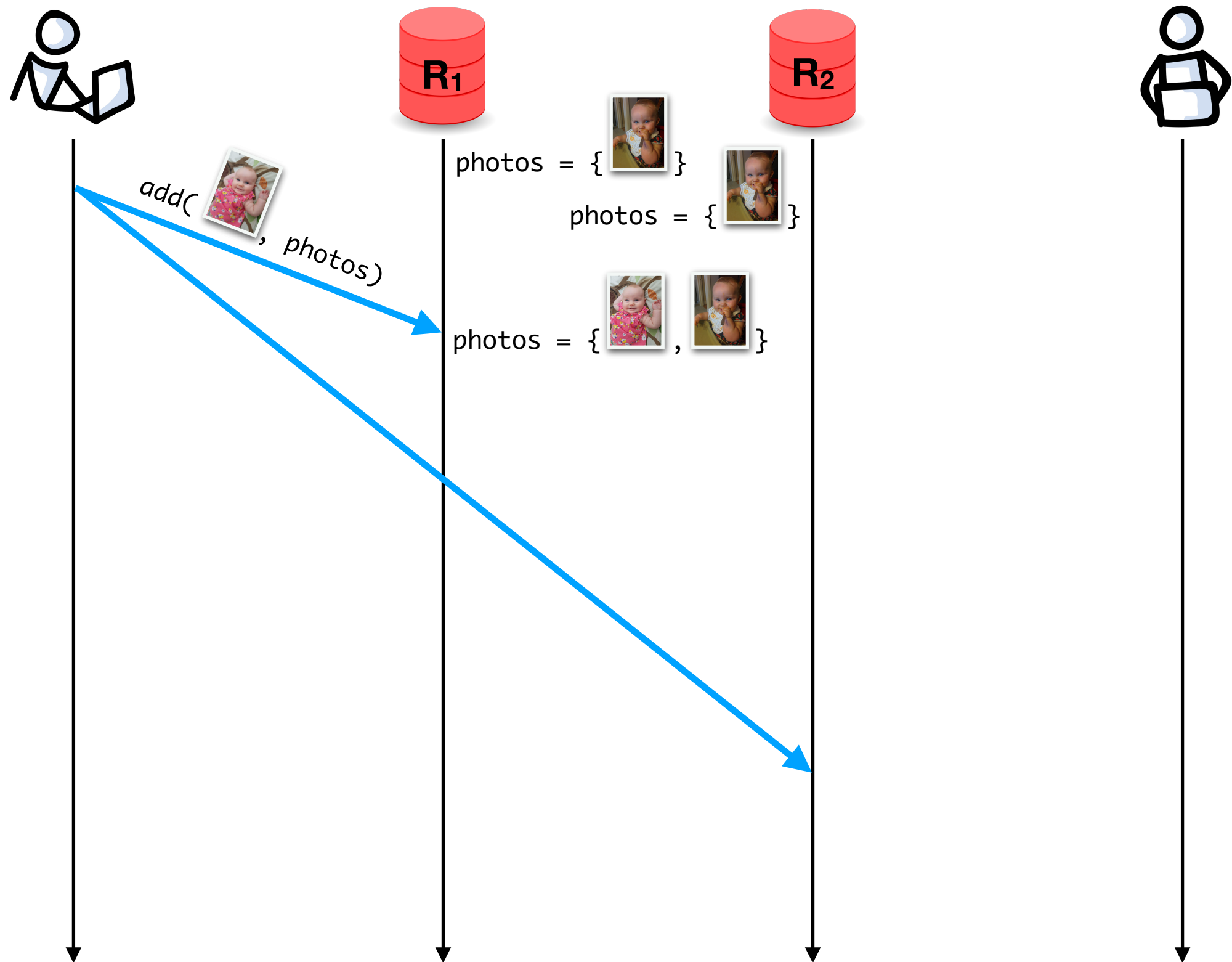


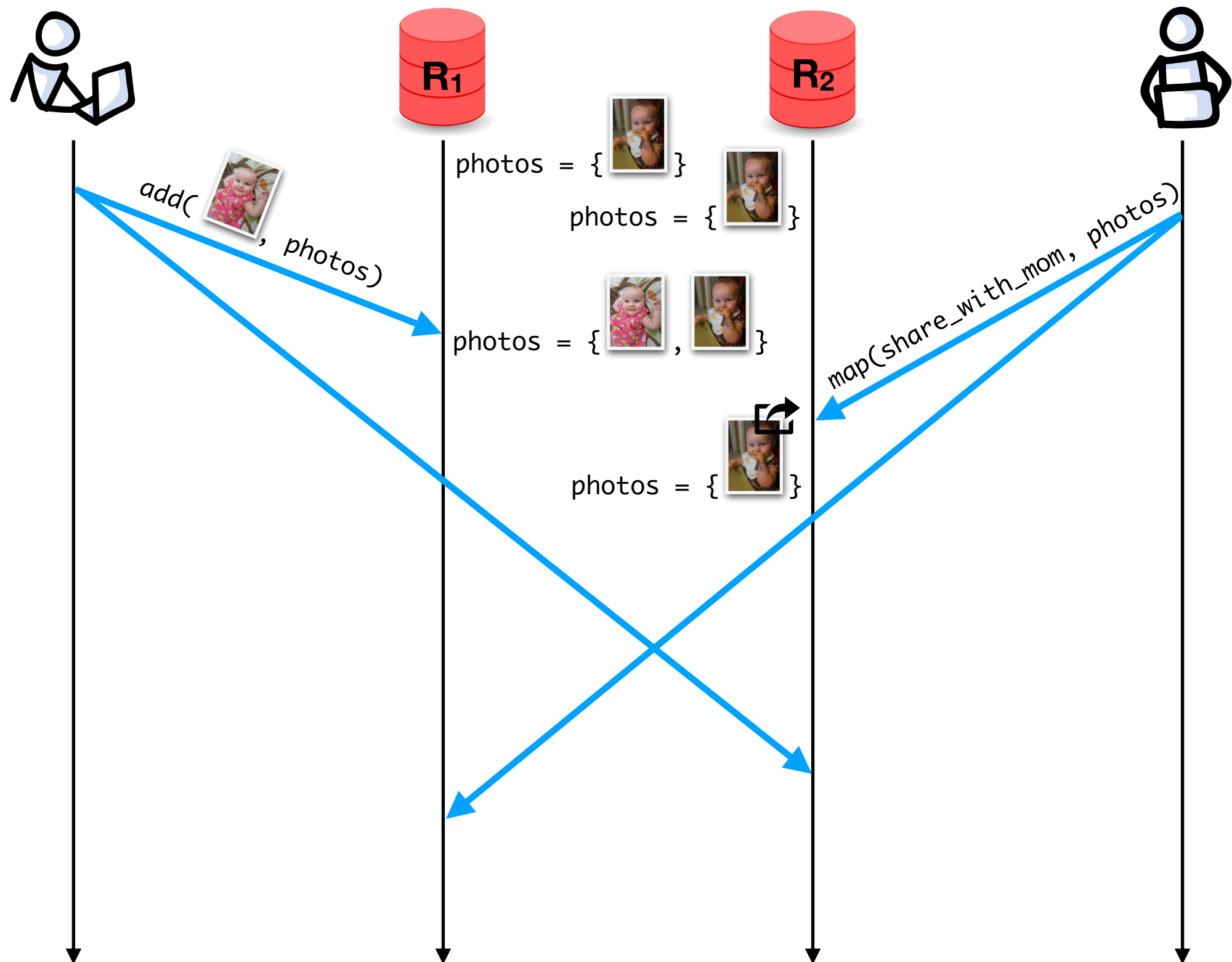
photos = {  }

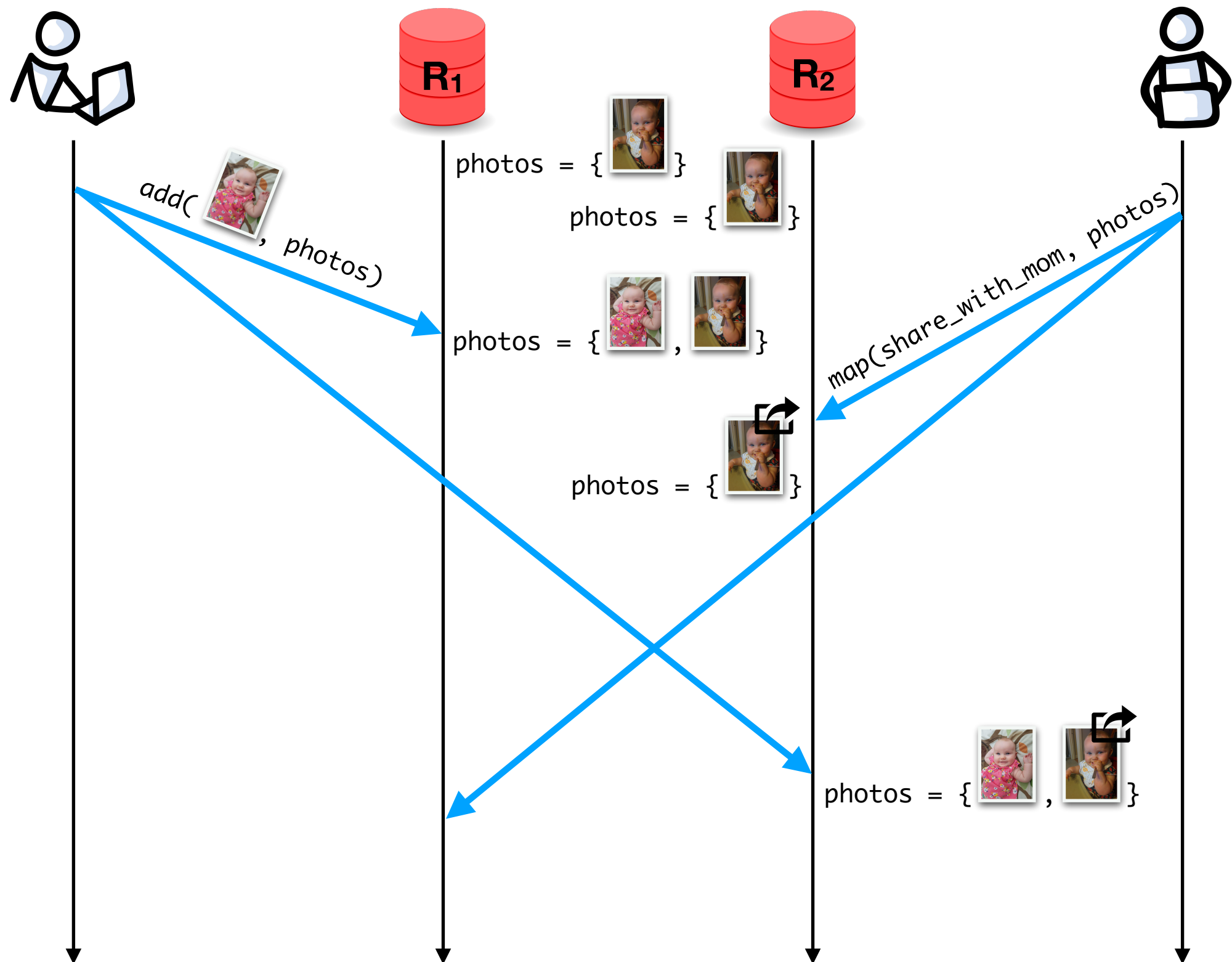
photos = {  }

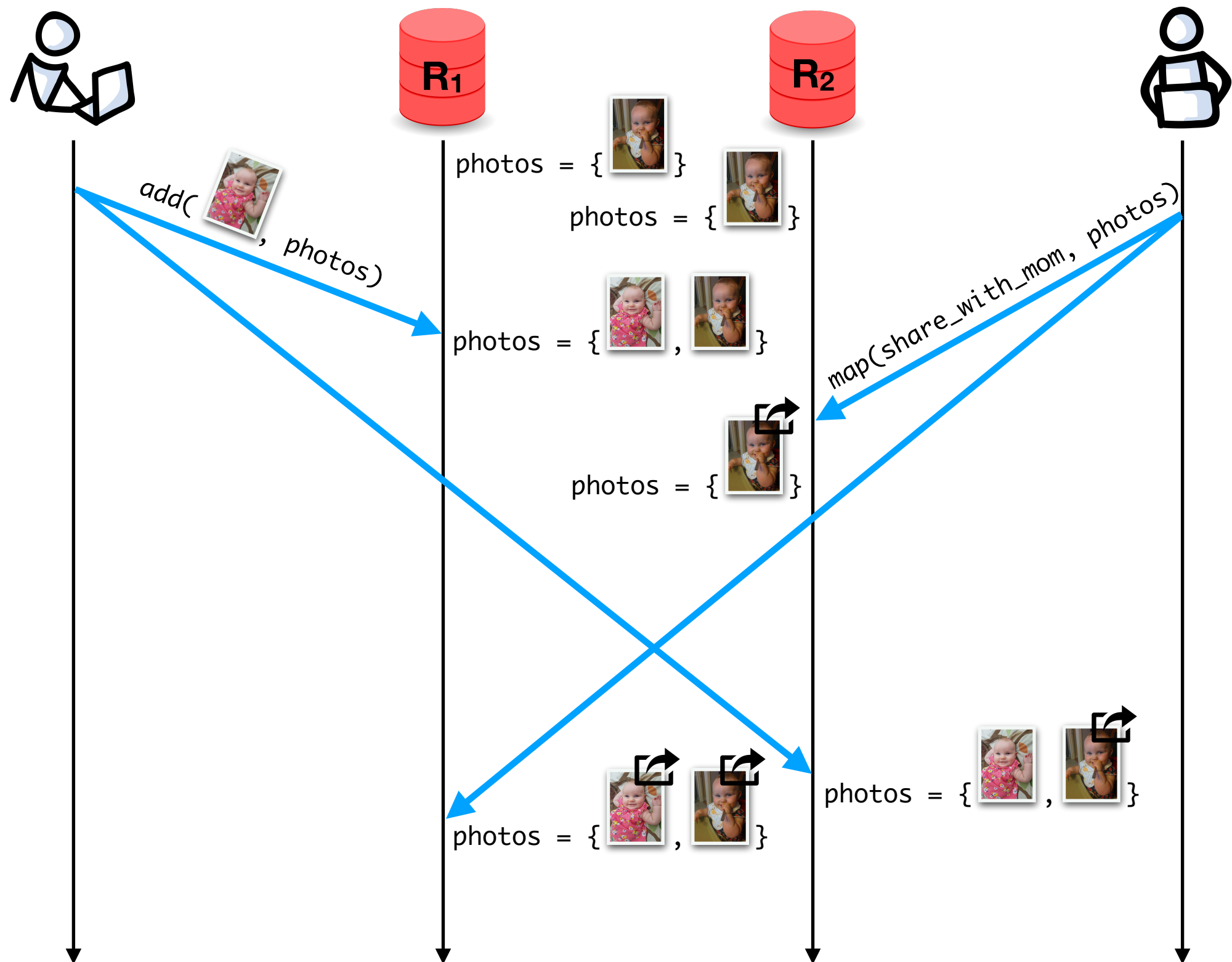


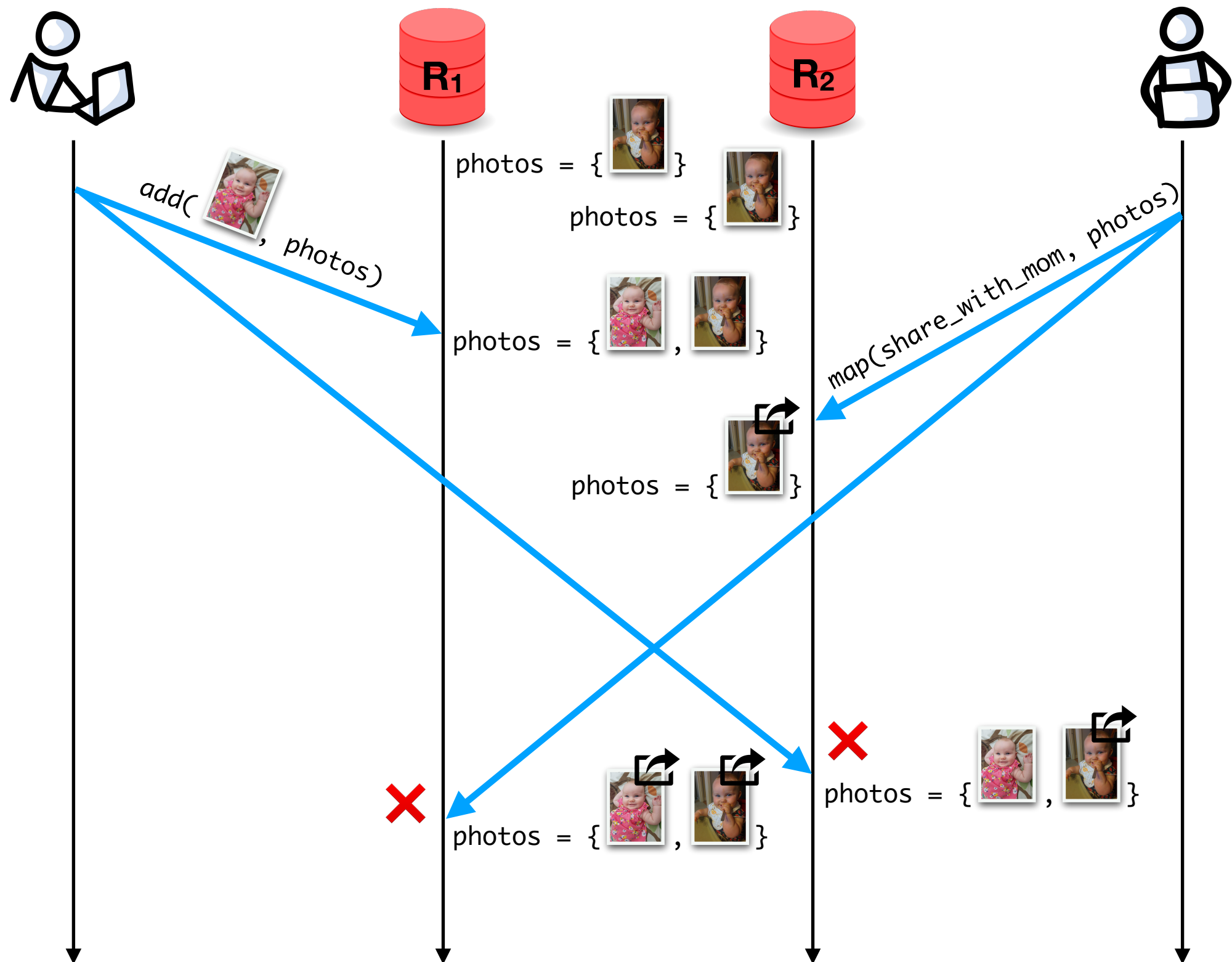


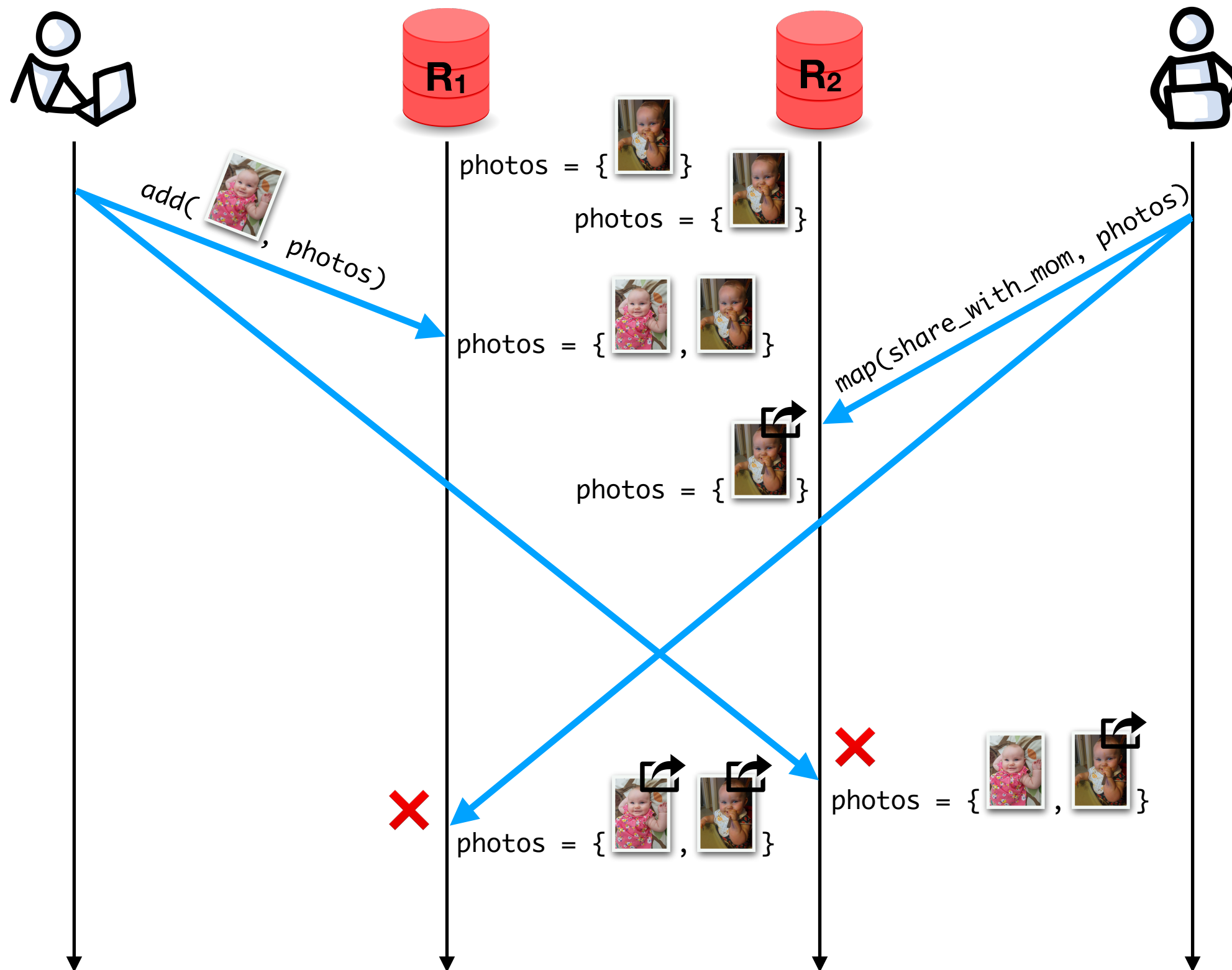






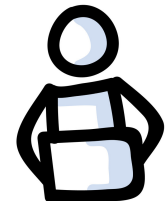
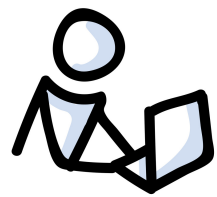








Conflict-free replicated data types

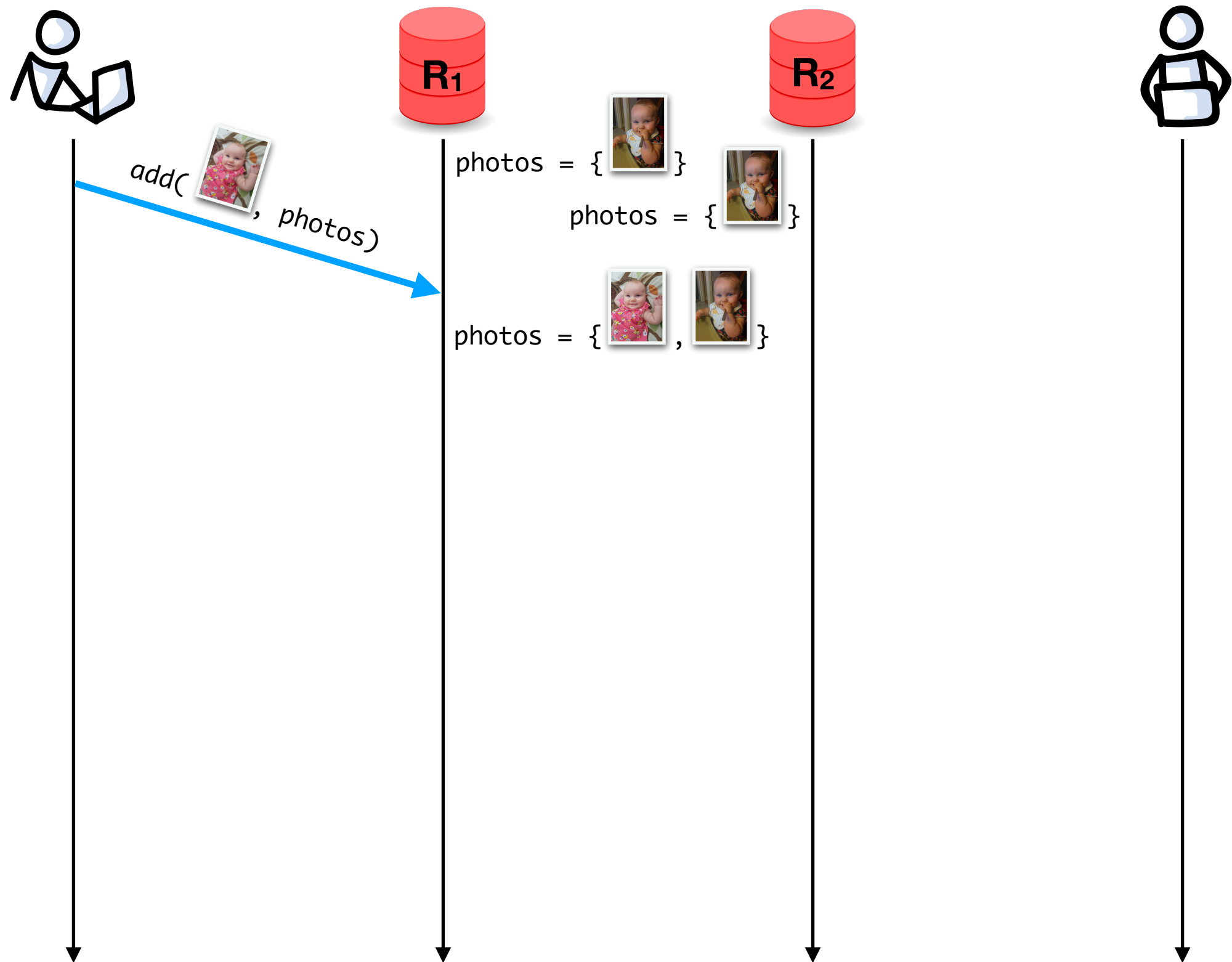
[Shapiro et al., 2011]

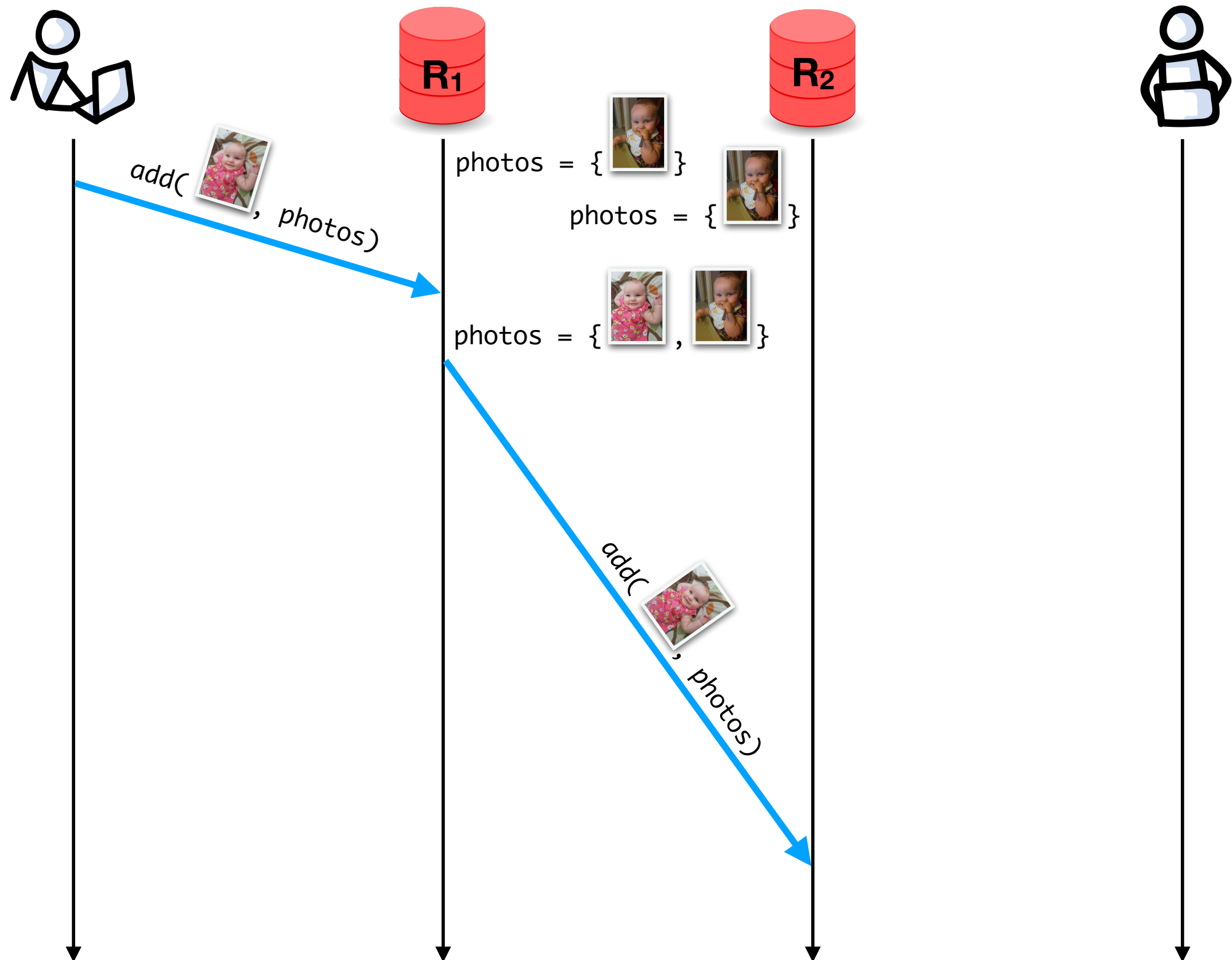


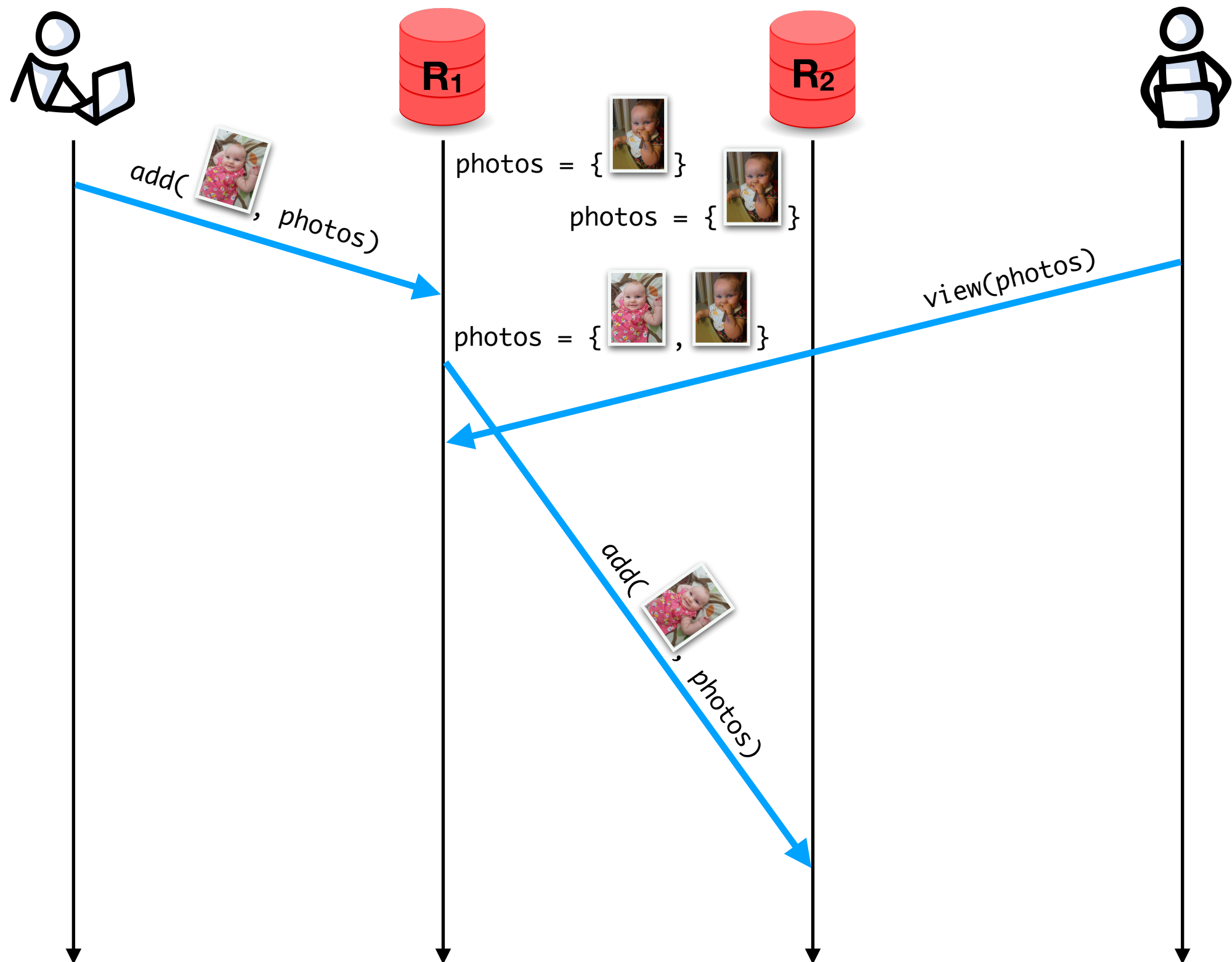
photos = {  }

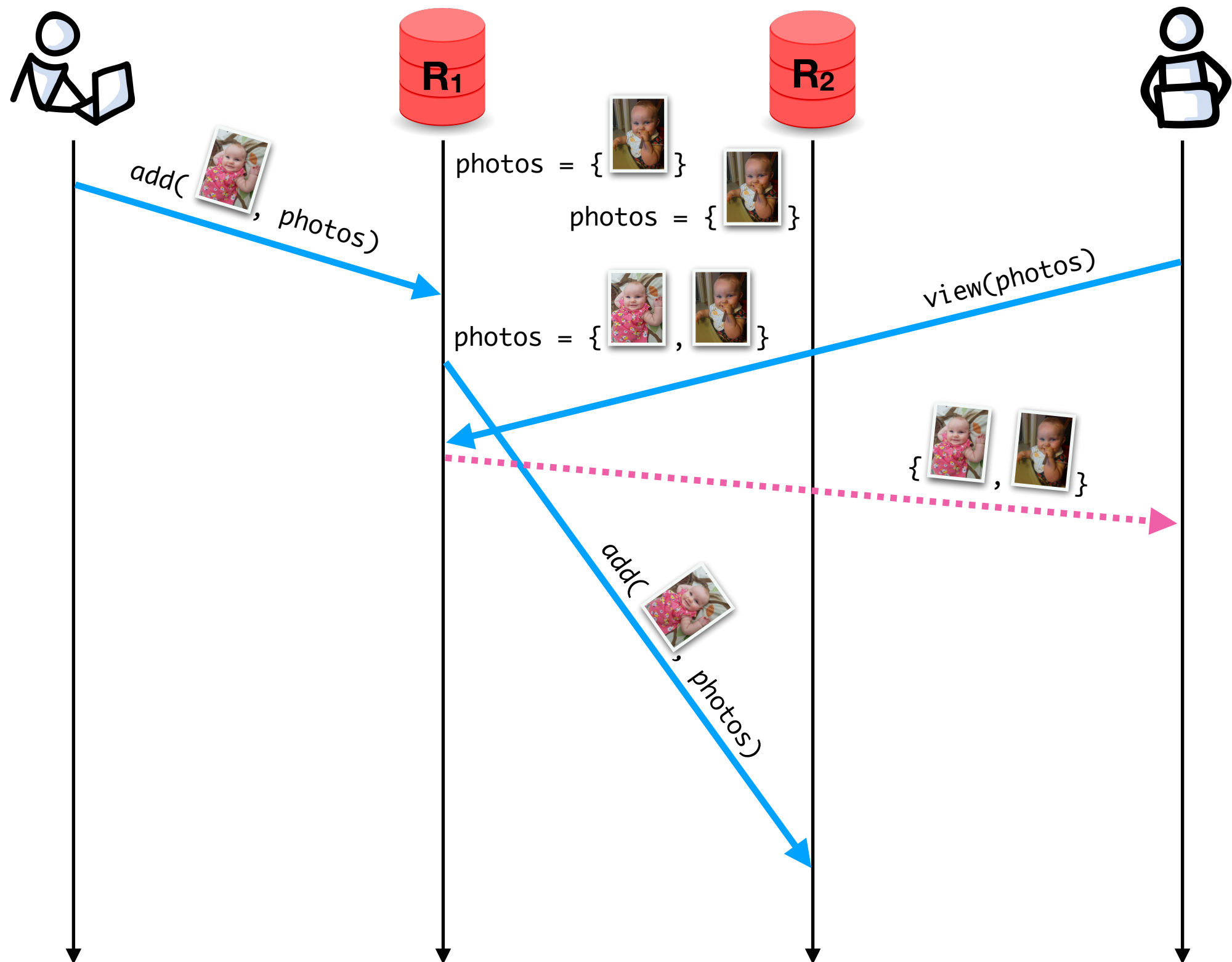
photos = {  }

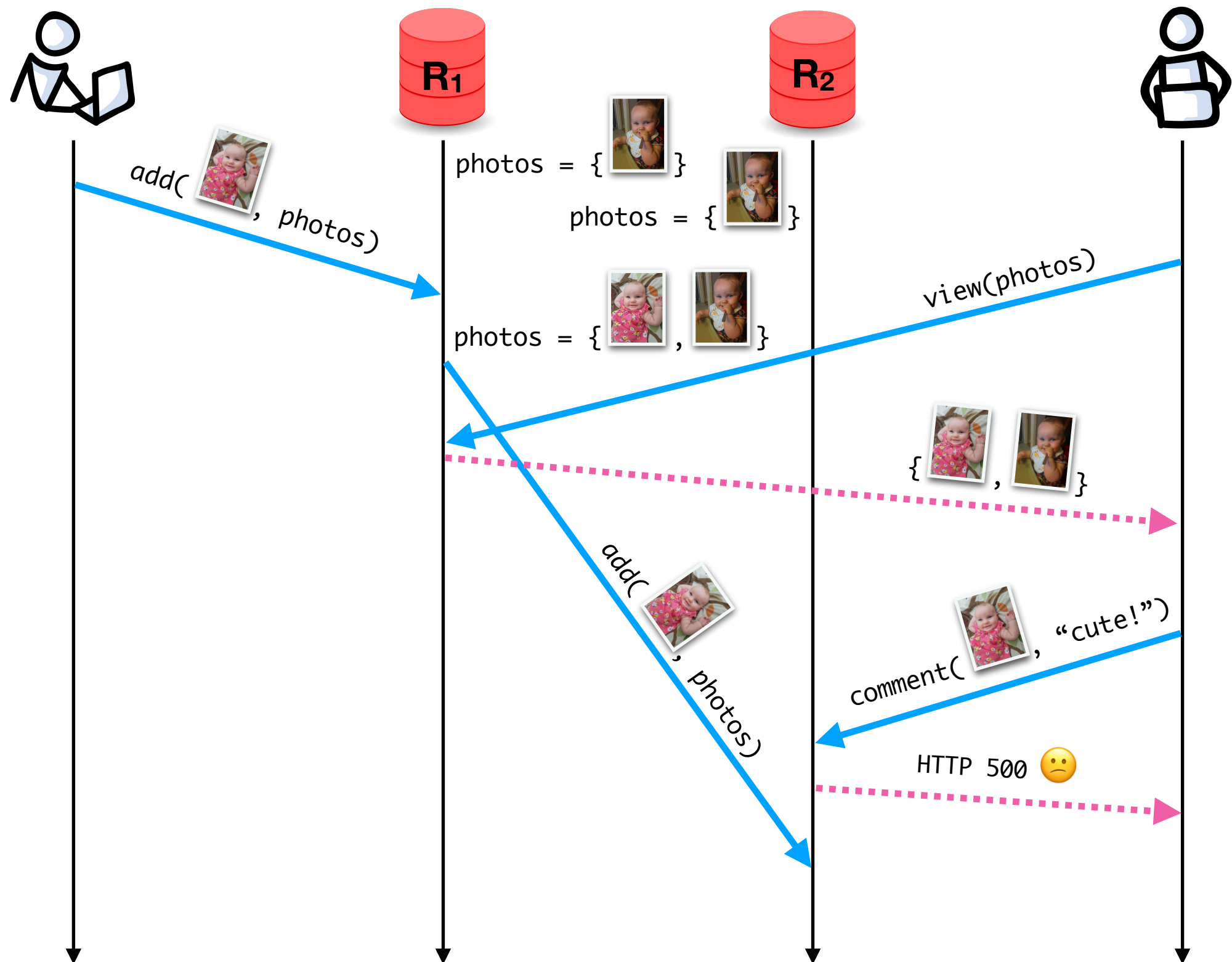


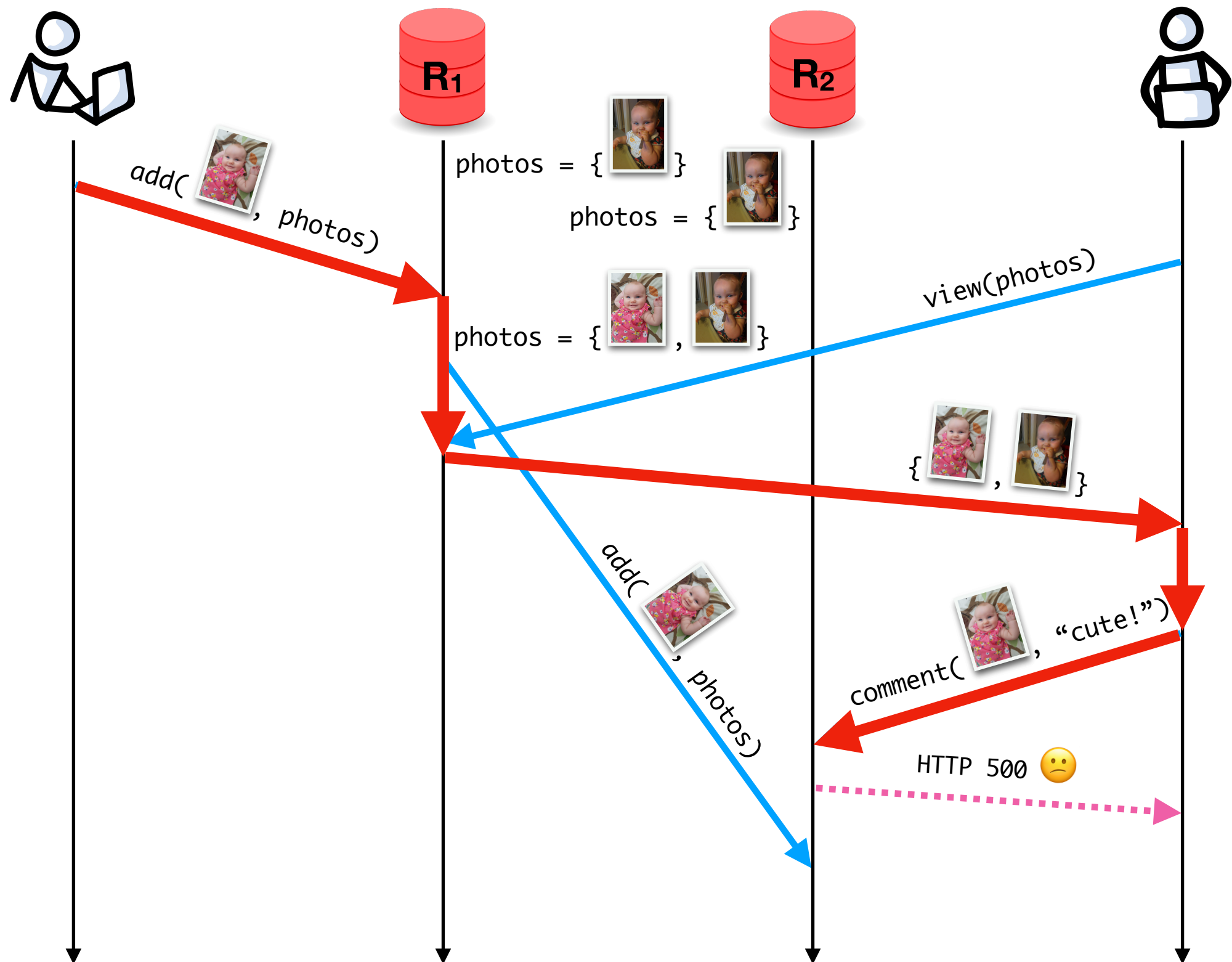


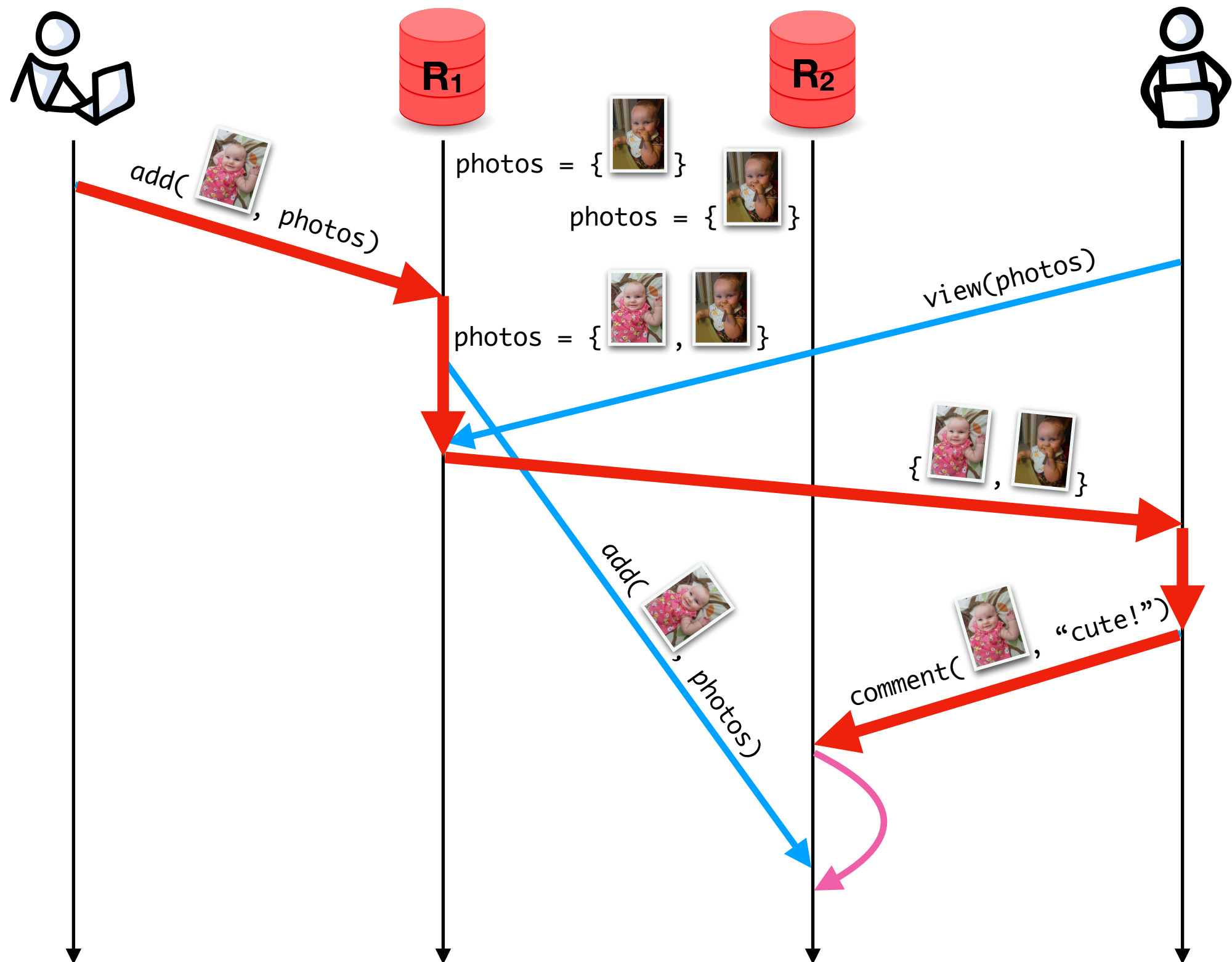


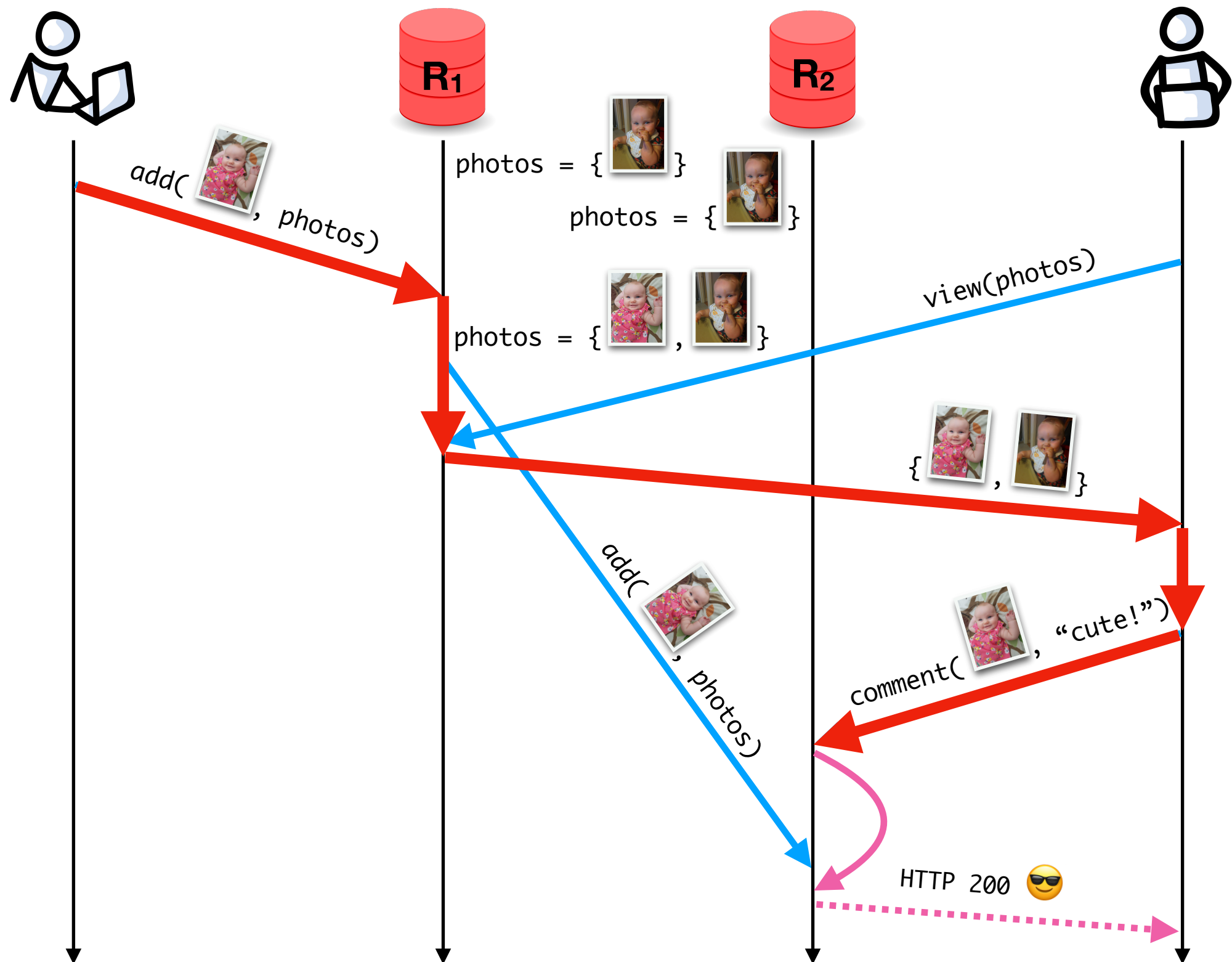


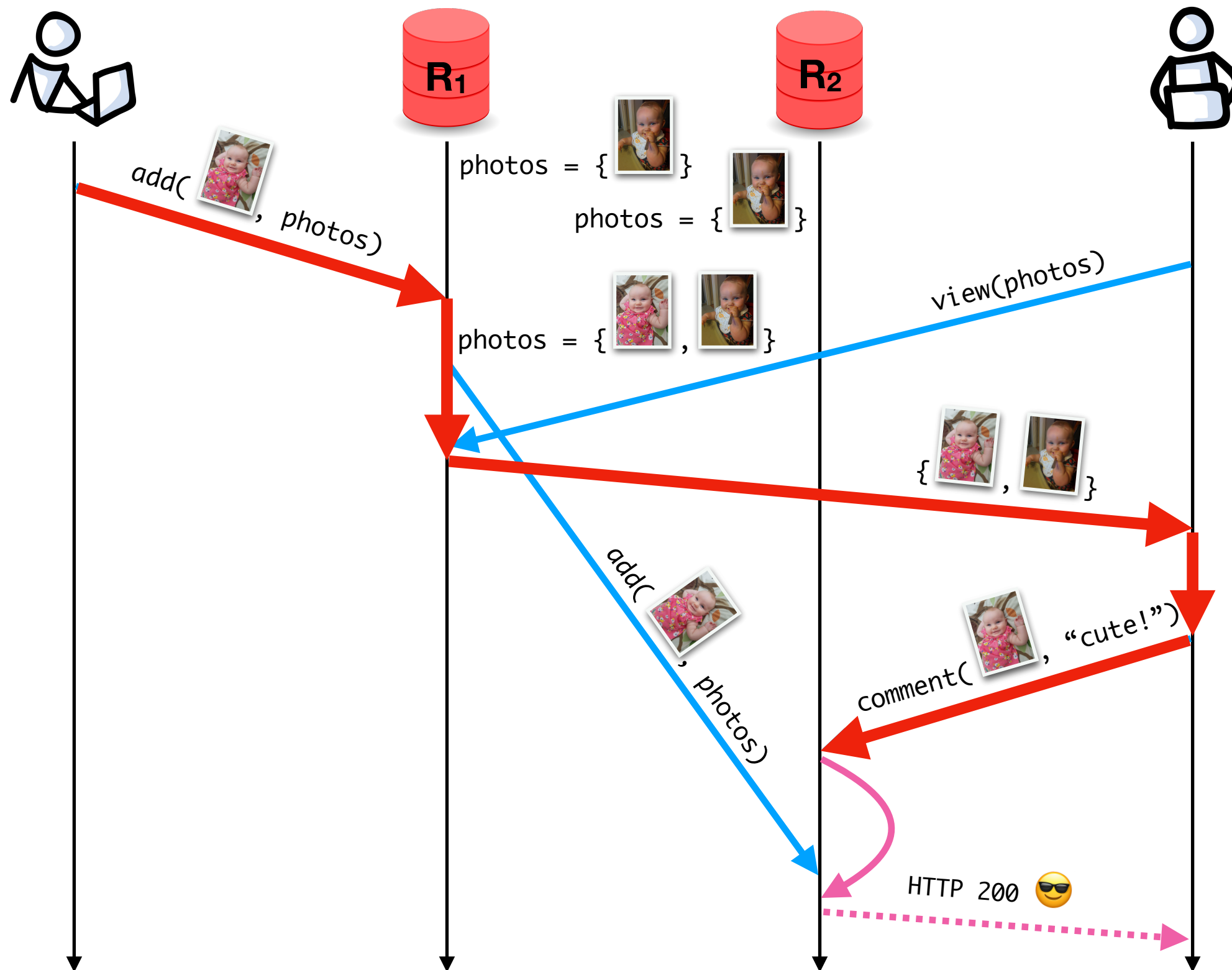




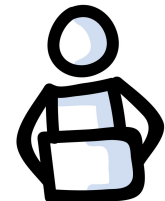
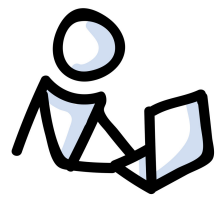




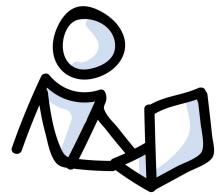




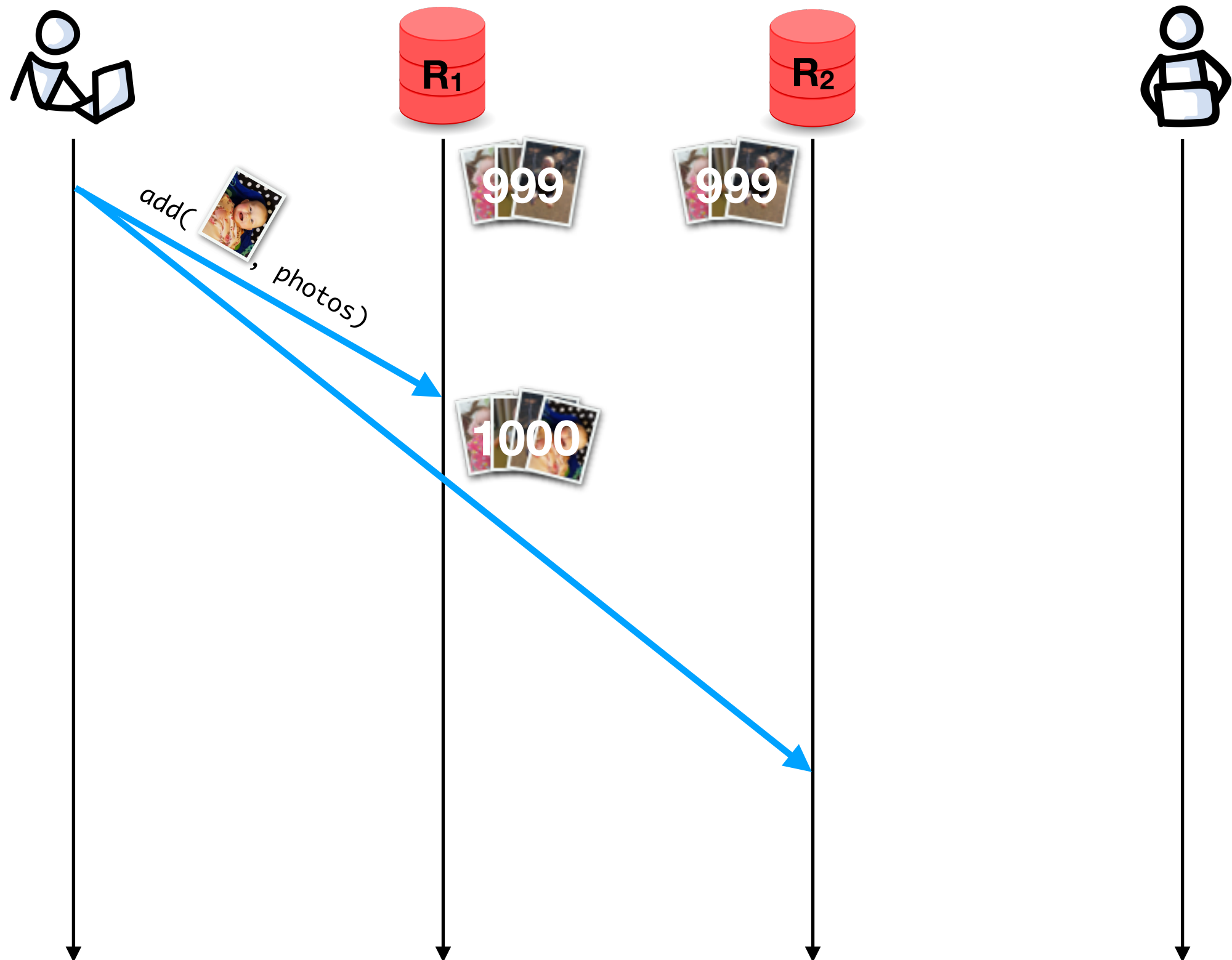
Causal delivery
[Birman and Joseph, 1987]



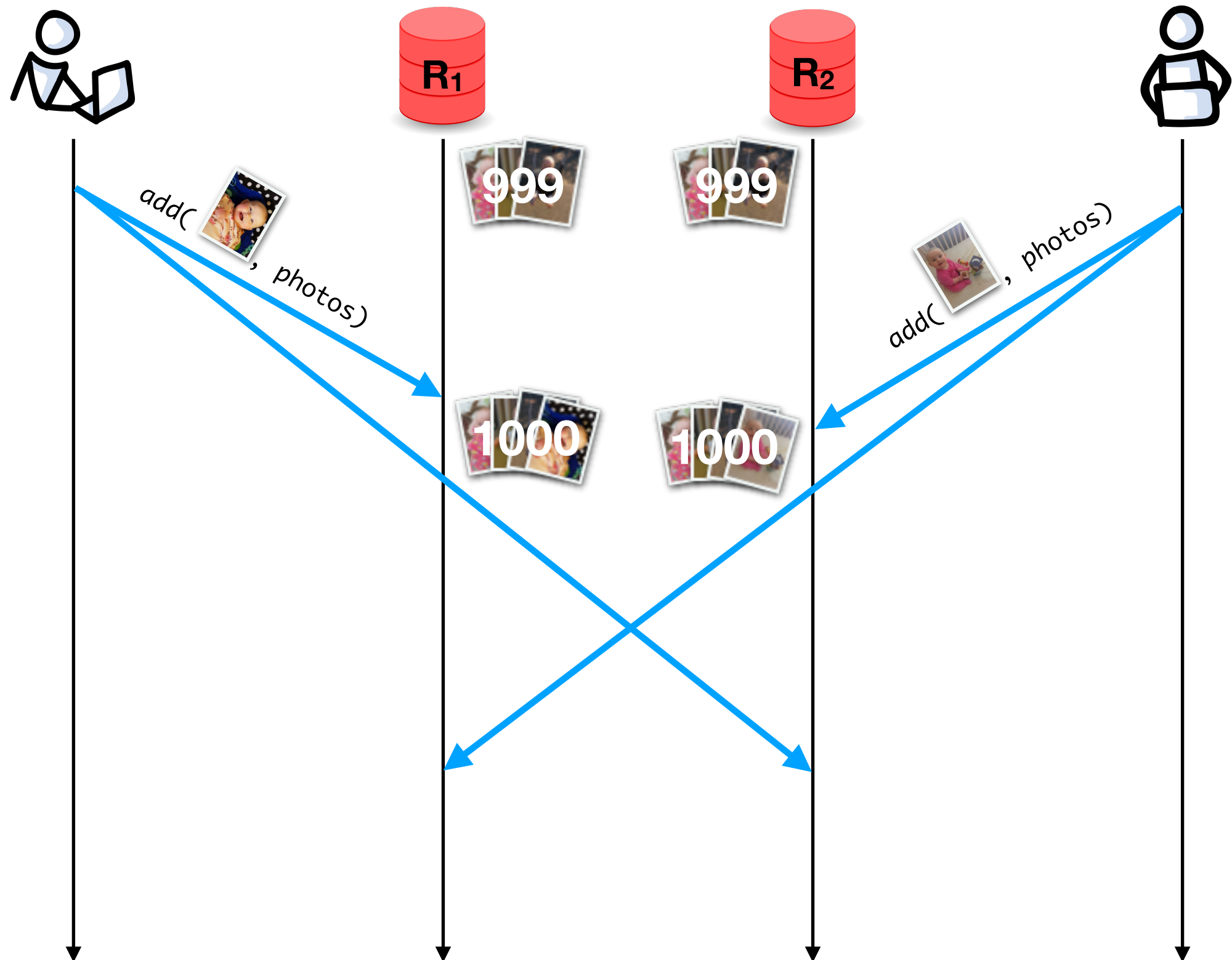
$\text{size}(\text{photos}) \leq 1000$



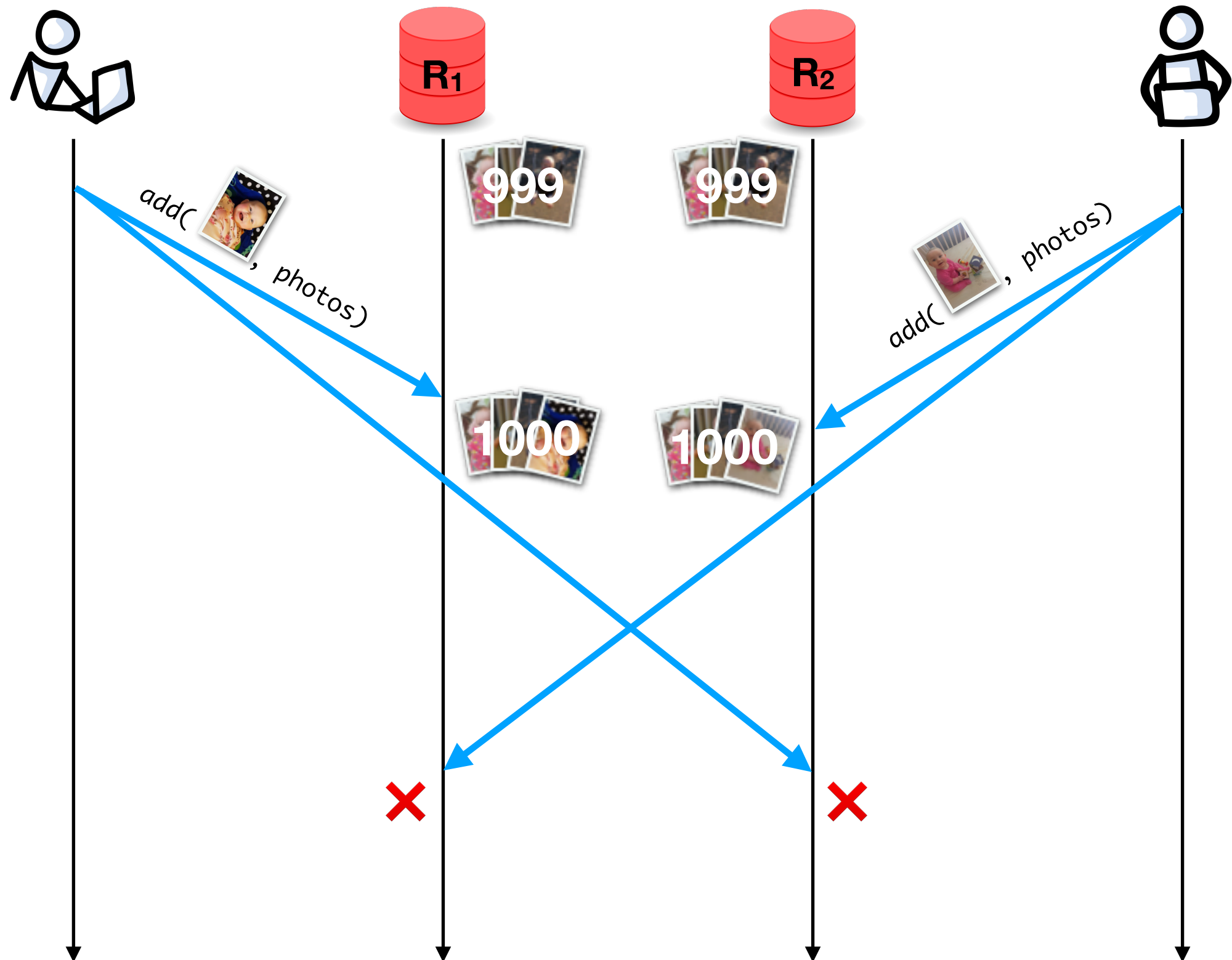
$\text{size}(\text{photos}) \leq 1000$



$\text{size}(\text{photos}) \leq 1000$



$\text{size}(\text{photos}) \leq 1000$

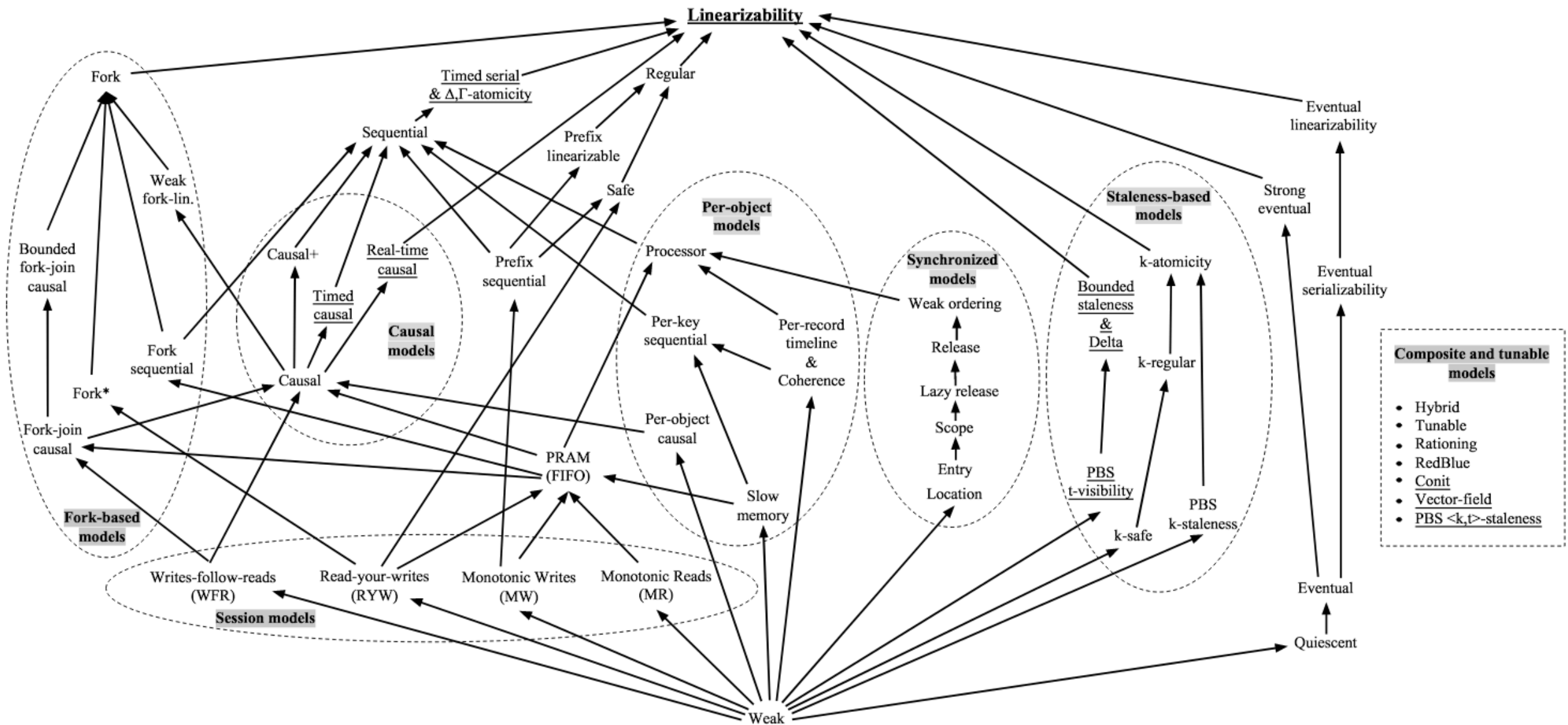


The distributed consistency model zoo:

“In which ways may replicas disagree?”

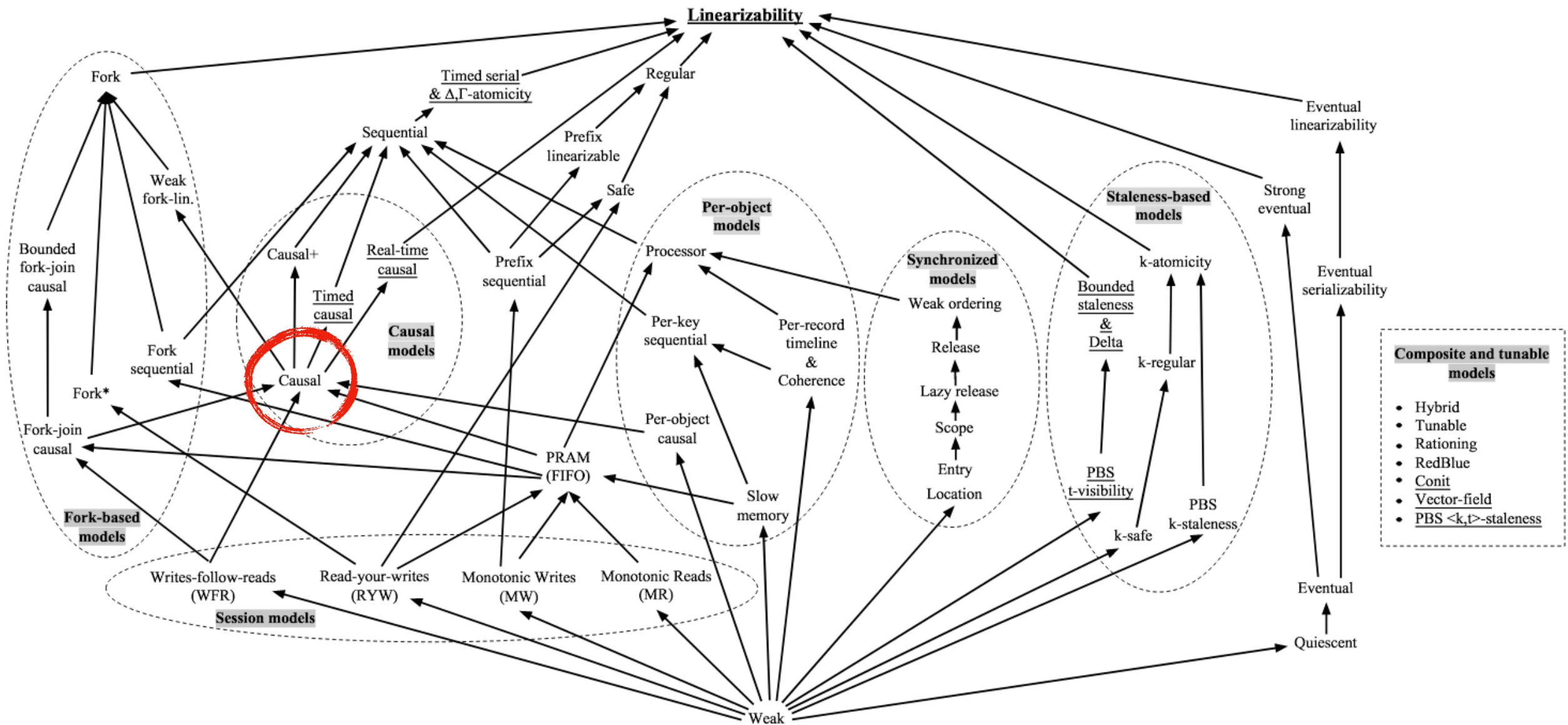
The distributed consistency model zoo:

“In which ways may replicas disagree?”



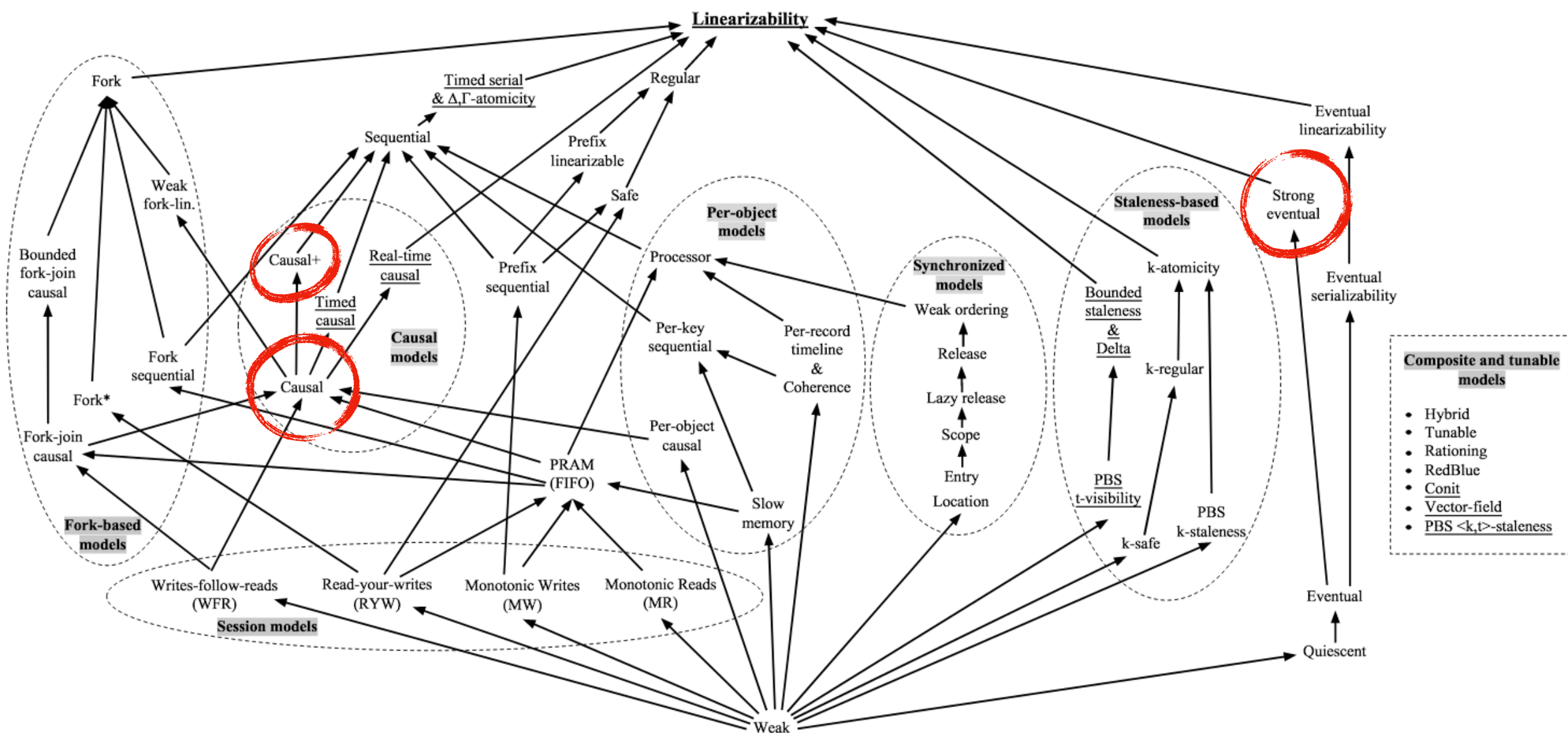
The distributed consistency model zoo:

“In which ways may replicas disagree?”



The distributed consistency model zoo:

“In which ways may replicas disagree?”





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

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!**)

Adventures in Building Reliable Distributed Systems with Liquid Haskell

Adventures in Building Reliable Distributed Systems with Liquid Haskell

Refinement types

[Rushby et al., 1998; Xi and Pfenning, 1998; Rondon et al., 2008; ...]

Refinement types

[Rushby et al., 1998; Xi and Pfenning, 1998; Rondon et al., 2008; ...]

```
type EvenInt = { n:Int | n mod 2 == 0 }
```


Refinement types

[Rushby et al., 1998; Xi and Pfenning, 1998; Rondon et al., 2008; ...]

```
type EvenInt = { n:Int | n mod 2 == 0 }
```

Refinement types

[Rushby et al., 1998; Xi and Pfenning, 1998; Rondon et al., 2008; ...]

```
type EvenInt = { n:Int | n mod 2 == 0 }  
type OddInt  = { n:Int | n mod 2 == 1 }
```

Refinement types

[Rushby et al., 1998; Xi and Pfenning, 1998; Rondon et al., 2008; ...]

```
type EvenInt = { n:Int | n mod 2 == 0 }
```

```
type OddInt  = { n:Int | n mod 2 == 1 }
```

```
oddAdd :: OddInt -> OddInt -> EvenInt
```

```
oddAdd x y = x + y
```

Refinement types

[Rushby et al., 1998; Xi and Pfenning, 1998; Rondon et al., 2008; ...]

```
type EvenInt = { n:Int | n mod 2 == 0 }  
type OddInt  = { n:Int | n mod 2 == 1 }
```

```
oddAdd :: OddInt -> OddInt -> EvenInt  
oddAdd x y = x + y
```

```
tail :: [a] -> [a]  
tail (_:xs) = xs  
tail [] = error "oh no!"
```

Refinement types

[Rushby et al., 1998; Xi and Pfenning, 1998; Rondon et al., 2008; ...]

```
type EvenInt = { n:Int | n mod 2 == 0 }  
type OddInt  = { n:Int | n mod 2 == 1 }
```

```
oddAdd :: OddInt -> OddInt -> EvenInt  
oddAdd x y = x + y
```

```
tail :: { l:[a] | l /= [] } -> [a]  
tail (_:xs) = xs  
tail [] = error "oh no!"
```

Refinement types

[Rushby et al., 1998; Xi and Pfenning, 1998; Rondon et al., 2008; ...]

```
type EvenInt = { n:Int | n mod 2 == 0 }  
type OddInt  = { n:Int | n mod 2 == 1 }
```

```
oddAdd :: OddInt -> OddInt -> EvenInt  
oddAdd x y = x + y
```

```
tail :: { l:[a] | l /= [] } -> [a]  
tail (_:xs) = xs
```


Refinement types

[Rushby et al., 1998; Xi and Pfenning, 1998; Rondon et al., 2008; ...]

```
type EvenInt = { n:Int | n mod 2 == 0 }  
type OddInt  = { n:Int | n mod 2 == 1 }
```

```
oddAdd :: OddInt -> OddInt -> EvenInt  
oddAdd x y = x + y
```

```
tail :: { l:[a] | l /= [] } -> { l':[a] | len l - 1 == len l' }  
tail (_:xs) = xs
```

Refinement *reflection*

[Vazou et al., 2018]

```
type EvenInt = { n:Int | n mod 2 == 0 }
```

```
type OddInt  = { n:Int | n mod 2 == 1 }
```

```
oddAdd :: OddInt -> OddInt -> EvenInt
```

```
oddAdd x y = x + y
```

Refinement *reflection*

[Vazou et al., 2018]

```
type EvenInt = { n:Int | n mod 2 == 0 }
```

```
type OddInt  = { n:Int | n mod 2 == 1 }
```

```
oddAdd :: OddInt -> OddInt -> EvenInt
```

```
oddAdd x y = x + y
```

```
sumEven :: x:OddInt -> y:OddInt
```

```
    -> { _:Proof | (oddAdd x y) mod 2 == 0 }
```

Refinement *reflection*

[Vazou et al., 2018]

```
type EvenInt = { n:Int | n mod 2 == 0 }
```

```
type OddInt  = { n:Int | n mod 2 == 1 }
```

```
oddAdd :: OddInt -> OddInt -> EvenInt
```

```
oddAdd x y = x + y
```

```
sumEven :: x:OddInt -> y:OddInt
```

```
-> { _:Proof | (oddAdd x y) mod 2 == 0 }
```

Refinement *reflection*

[Vazou et al., 2018]

```
type EvenInt = { n:Int | n mod 2 == 0 }
```

```
type OddInt  = { n:Int | n mod 2 == 1 }
```

```
oddAdd :: OddInt -> OddInt -> EvenInt
```

```
oddAdd x y = x + y
```

```
sumEven :: x:OddInt -> y:OddInt
```

```
-> { _:Proof | (oddAdd x y) mod 2 == 0 }
```

Refinement *reflection*

[Vazou et al., 2018]

```
type EvenInt = { n:Int | n mod 2 == 0 }
```

```
type OddInt  = { n:Int | n mod 2 == 1 }
```

```
oddAdd :: OddInt -> OddInt -> EvenInt
```

```
oddAdd x y = x + y
```

```
sumEven :: x:OddInt -> y:OddInt
```

```
      -> { _:Proof | (oddAdd x y) mod 2 == 0 }
```

```
sumEven _ _ = ()
```


Refinement *reflection*

[Vazou et al., 2018]

```
type EvenInt = { n:Int | n mod 2 == 0 }
```

```
type OddInt  = { n:Int | n mod 2 == 1 }
```

```
oddAdd :: OddInt -> OddInt -> EvenInt
```

```
oddAdd x y = x + y
```

```
sumEven :: x:OddInt -> y:OddInt
```

```
    -> { _:Proof | (oddAdd x y) mod 2 == 0 }
```

```
sumEven _ _ = ()
```

Refinement *reflection*

[Vazou et al., 2018]

```
type EvenInt = { n:Int | n mod 2 == 0 }
```

```
type OddInt  = { n:Int | n mod 2 == 1 }
```

```
oddAdd :: OddInt -> OddInt -> EvenInt
```

```
oddAdd x y = x + y
```

```
sumEven :: x:OddInt -> y:OddInt
```

```
    -> { _:Proof | (oddAdd x y) mod 2 == 0 }
```

```
sumEven _ _ = ()
```

```
oddAddComm :: x:OddInt -> y:OddInt
```

```
    -> { _:Proof | oddAdd x y == oddAdd y x }
```

Refinement *reflection*

[Vazou et al., 2018]

```
type EvenInt = { n:Int | n mod 2 == 0 }
```

```
type OddInt  = { n:Int | n mod 2 == 1 }
```

```
oddAdd :: OddInt -> OddInt -> EvenInt
```

```
oddAdd x y = x + y
```

```
sumEven :: x:OddInt -> y:OddInt
```

```
    -> { _:Proof | (oddAdd x y) mod 2 == 0 }
```

```
sumEven _ _ = ()
```

```
oddAddComm :: x:OddInt -> y:OddInt
```

```
    -> { _:Proof | oddAdd x y == oddAdd y x }
```

Refinement *reflection*

[Vazou et al., 2018]

```
type EvenInt = { n:Int | n mod 2 == 0 }
```

```
type OddInt  = { n:Int | n mod 2 == 1 }
```

```
oddAdd :: OddInt -> OddInt -> EvenInt
```

```
oddAdd x y = x + y
```

```
sumEven :: x:OddInt -> y:OddInt
```

```
    -> { _:Proof | (oddAdd x y) mod 2 == 0 }
```

```
sumEven _ _ = ()
```

```
oddAddComm :: x:OddInt -> y:OddInt
```

```
    -> { _:Proof | oddAdd x y == oddAdd y x }
```

Refinement *reflection*

[Vazou et al., 2018]

```
type EvenInt = { n:Int | n mod 2 == 0 }
```

```
type OddInt  = { n:Int | n mod 2 == 1 }
```

```
oddAdd :: OddInt -> OddInt -> EvenInt
```

```
oddAdd x y = x + y
```

```
sumEven :: x:OddInt -> y:OddInt
```

```
    -> { _:Proof | (oddAdd x y) mod 2 == 0 }
```

```
sumEven _ _ = ()
```

```
oddAddComm :: x:OddInt -> y:OddInt
```

```
    -> { _:Proof | oddAdd x y == oddAdd y x }
```

```
oddAddComm _ _ = ()
```

Refinement *reflection*

[Vazou et al., 2018]

```
type EvenInt = { n:Int | n mod 2 == 0 }
```

```
type OddInt  = { n:Int | n mod 2 == 1 }
```

```
oddAdd :: OddInt -> OddInt -> EvenInt
```

```
oddAdd x y = x + y
```

```
sumEven :: x:OddInt -> y:OddInt
```

```
    -> { _:Proof | (oddAdd x y) mod 2 == 0 }
```

```
sumEven _ _ = ()
```

```
oddAddComm :: x:OddInt -> y:OddInt
```

```
    -> { _:Proof | oddAdd x y == oddAdd y x }
```

```
oddAddComm _ _ = ()
```


Refinement *reflection*

[Vazou et al., 2018]

```
type EvenInt = { n:Int | n mod 2 == 0 }  
type OddInt  = { n:Int | n mod 2 == 1 }
```

```
oddAdd :: OddInt -> OddInt -> EvenInt  
oddAdd x y = x + y
```

application code

```
sumEven :: x:OddInt -> y:OddInt  
         -> { _:Proof | (oddAdd x y) mod 2 == 0 }  
sumEven _ _ = ()
```

```
oddAddComm :: x:OddInt -> y:OddInt  
           -> { _:Proof | oddAdd x y == oddAdd y x }  
oddAddComm _ _ = ()
```

Refinement *reflection*

[Vazou et al., 2018]

```
type EvenInt = { n:Int | n mod 2 == 0 }  
type OddInt  = { n:Int | n mod 2 == 1 }
```

```
oddAdd :: OddInt -> OddInt -> EvenInt  
oddAdd x y = x + y
```

application code

```
sumEven :: x:OddInt -> y:OddInt  
         -> { _:Proof | (oddAdd x y) mod 2 == 0 }  
sumEven _ _ = ()
```

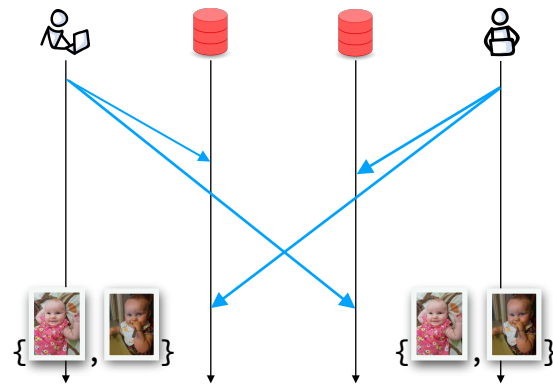
verification code

```
oddAddComm :: x:OddInt -> y:OddInt  
           -> { _:Proof | oddAdd x y == oddAdd y x }  
oddAddComm _ _ = ()
```

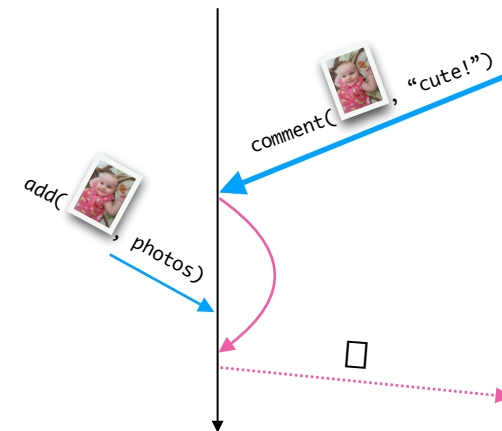
Adventures in Building Reliable Distributed Systems with Liquid Haskell

Adventures in Building Reliable Distributed Systems with Liquid Haskell

Adventures in Building Reliable Distributed Systems with Liquid Haskell

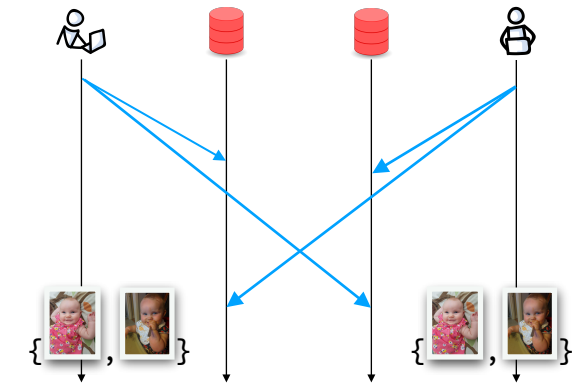


verified **strongly convergent**
replicated data structures



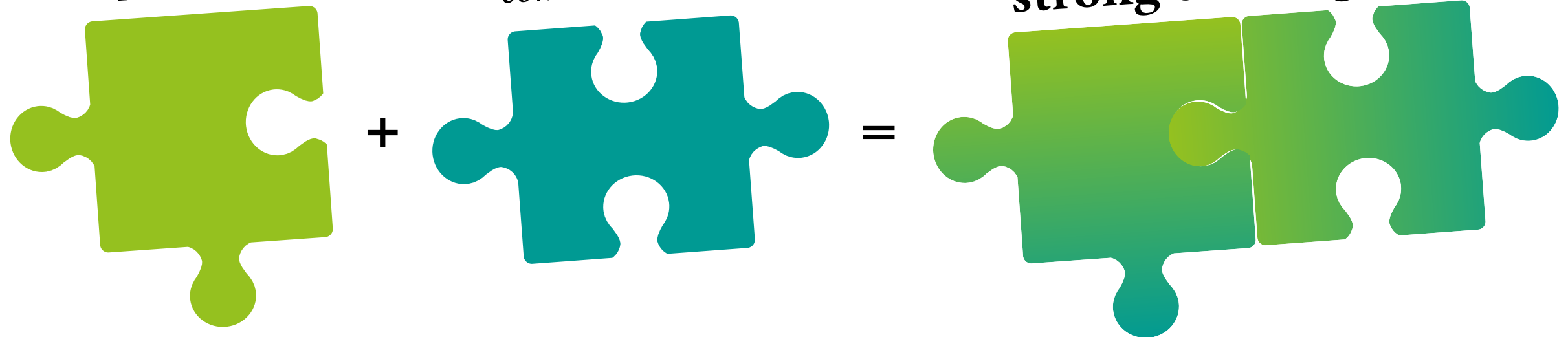
verified **causal delivery**

Verified **strong convergence** of CRDTs [OOPSLA 2020]



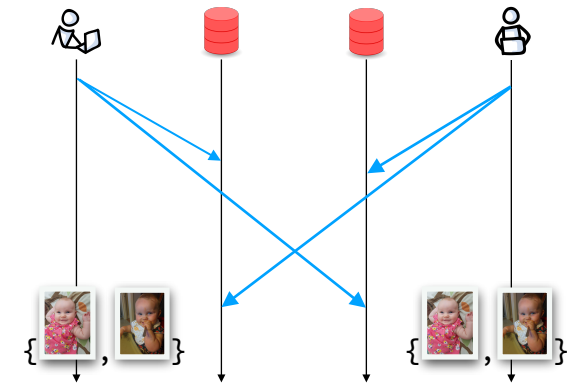
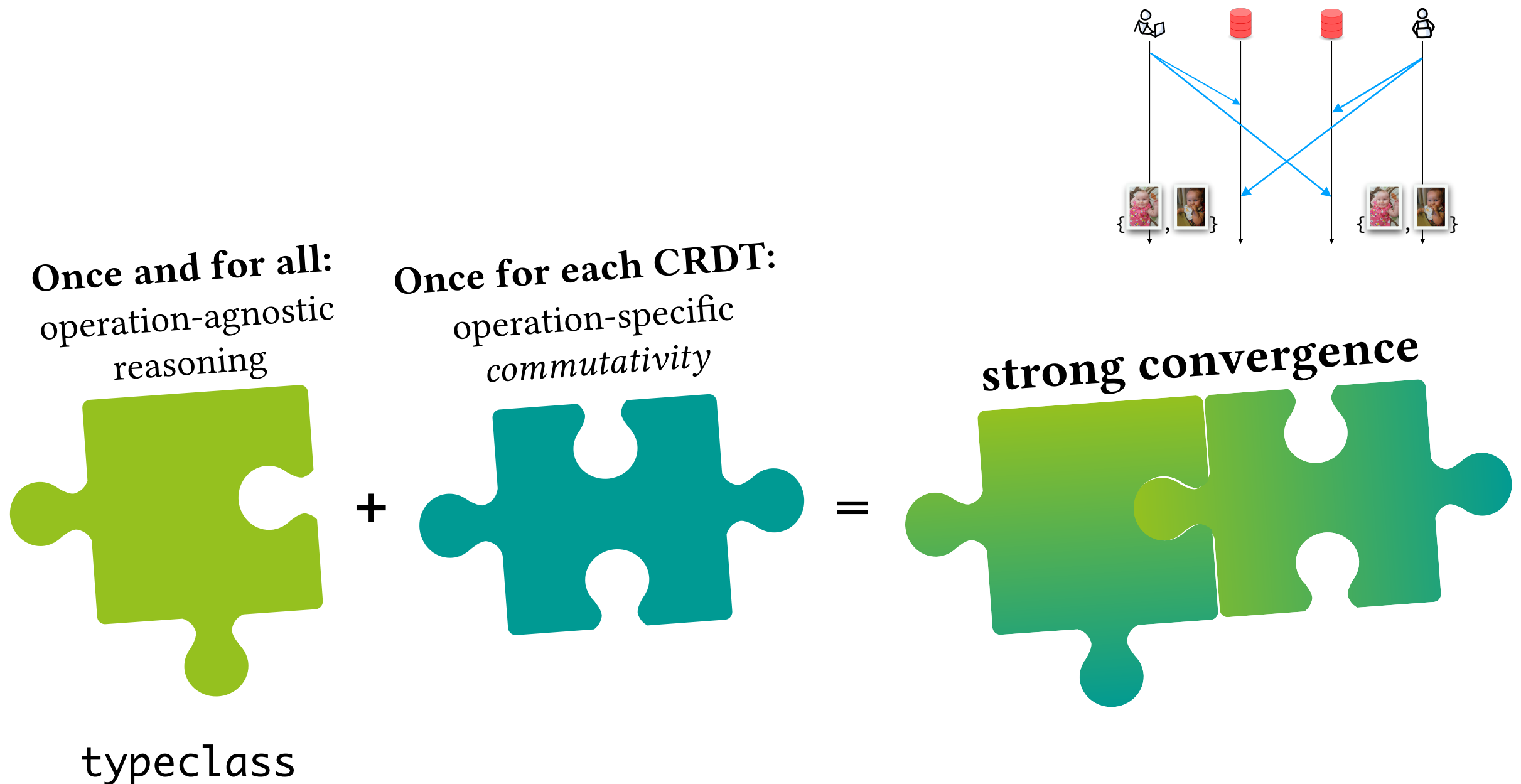
Once and for all:
operation-agnostic
reasoning

Once for each CRDT:
operation-specific
commutativity



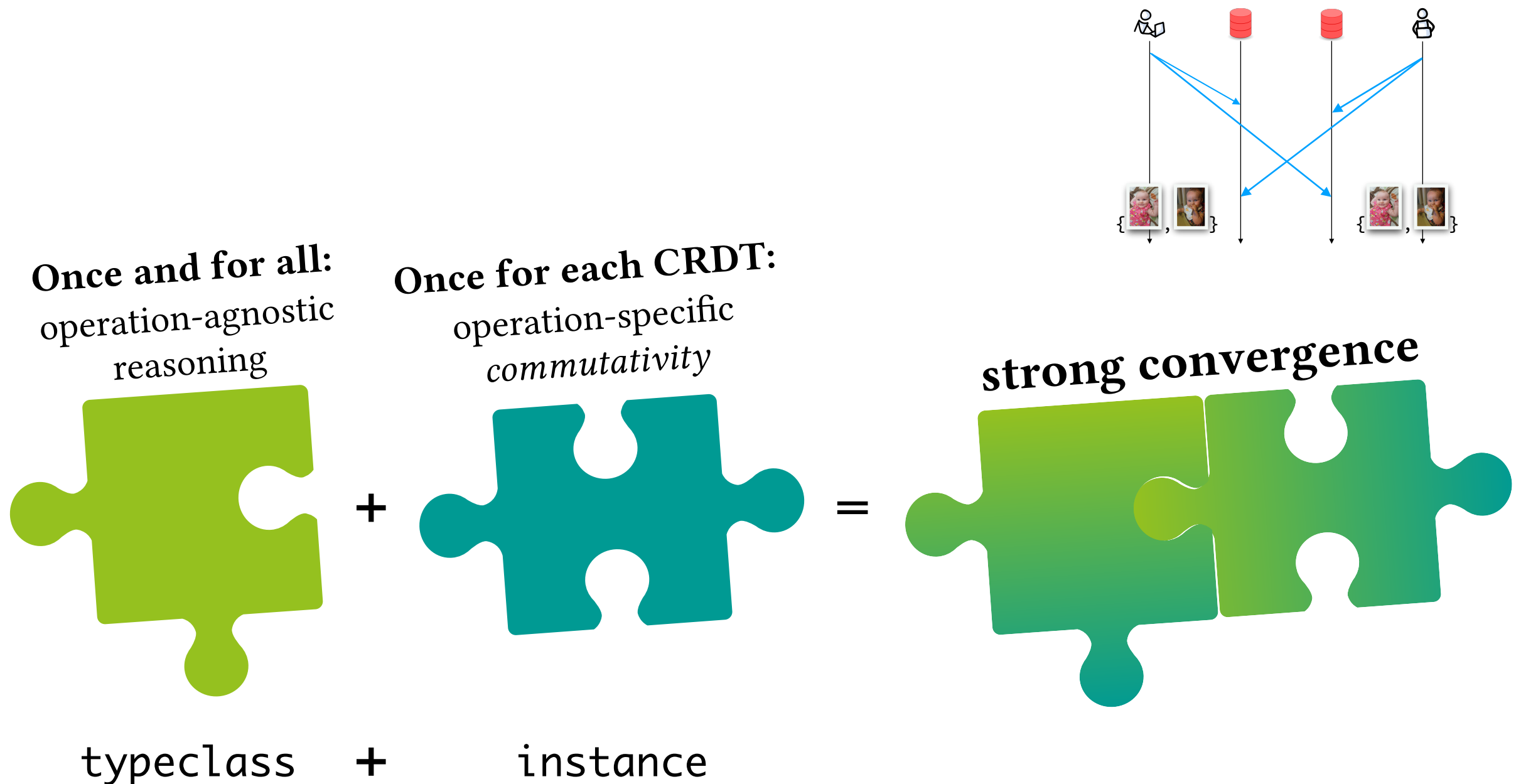
Joint work with: Yiyun Liu, James Parker, Patrick Redmond, Michael Hicks, and Niki Vazou

Verified **strong convergence** of CRDTs [OOPSLA 2020]



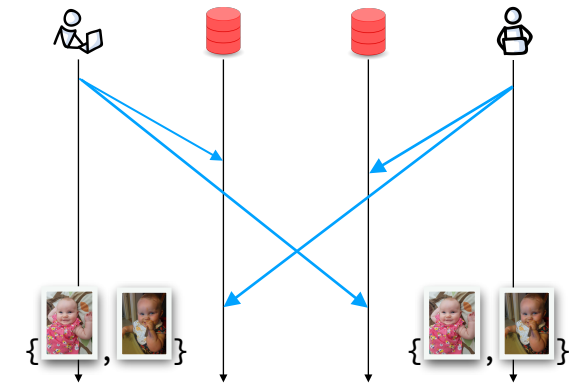
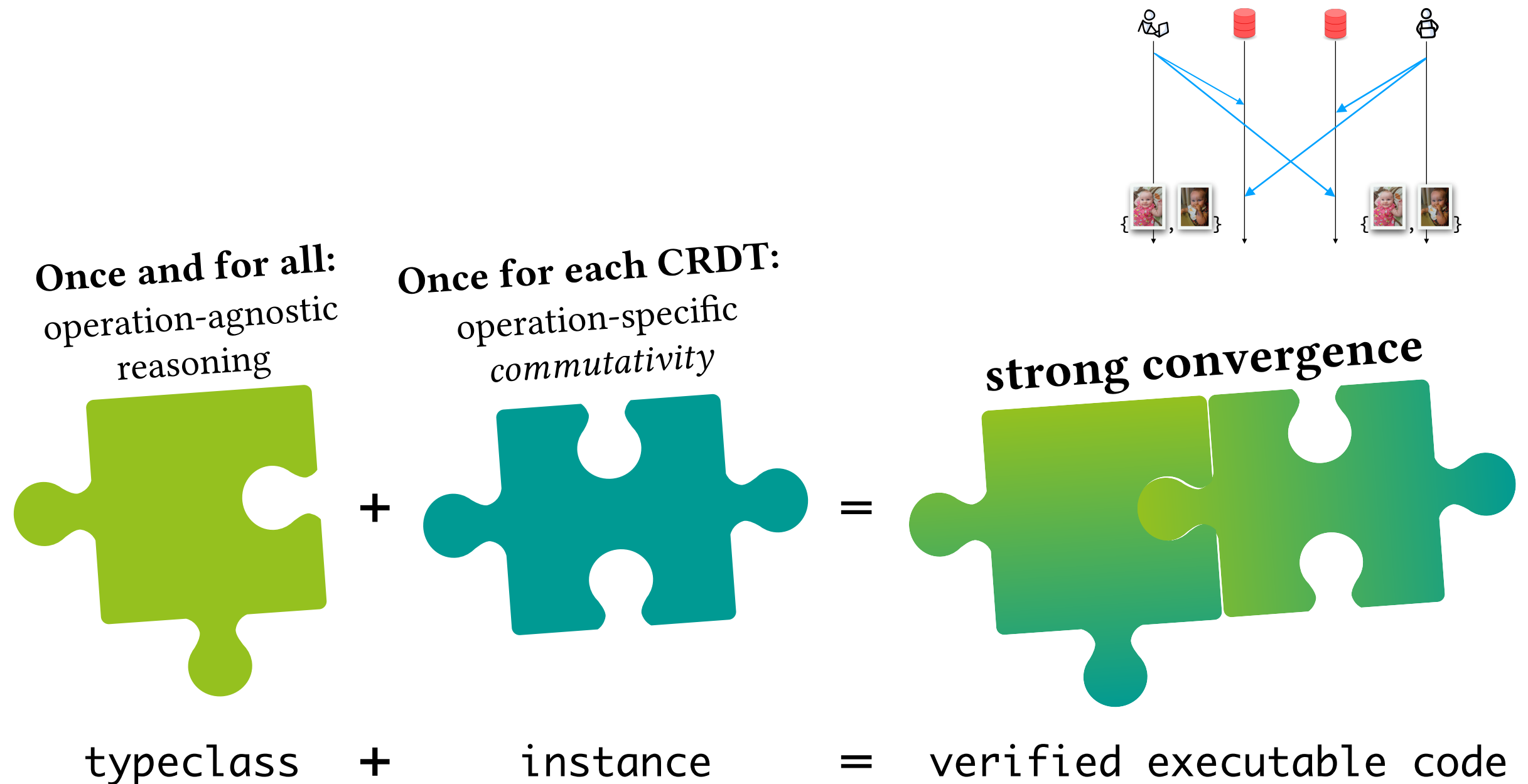
Joint work with: Yiyun Liu, James Parker, Patrick Redmond, Michael Hicks, and Niki Vazou

Verified **strong convergence** of CRDTs [OOPSLA 2020]



Joint work with: Yiyun Liu, James Parker, Patrick Redmond, Michael Hicks, and Niki Vazou

Verified **strong convergence** of CRDTs [OOPSLA 2020]



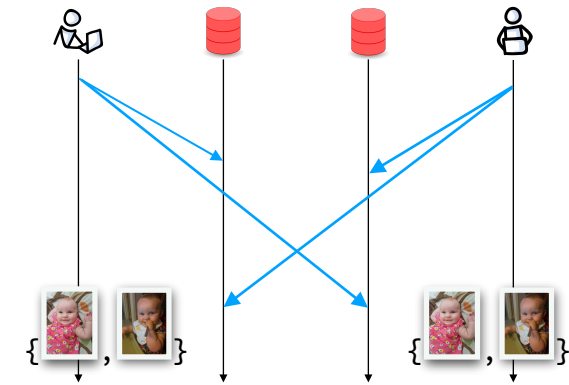
Joint work with: Yiyun Liu, James Parker, Patrick Redmond, Michael Hicks, and Niki Vazou

Verified **strong convergence** of CRDTs [OOPSLA 2020]

```
class CRDT t where  
  type Op t
```

```
apply :: t -> Op t -> t
```

```
lawCommut :: x:t -> op1:Op t -> op2:Op t  
  -> { _ : Proof | (apply (apply x op1) op2  
    = apply (apply x op2) op1) }
```



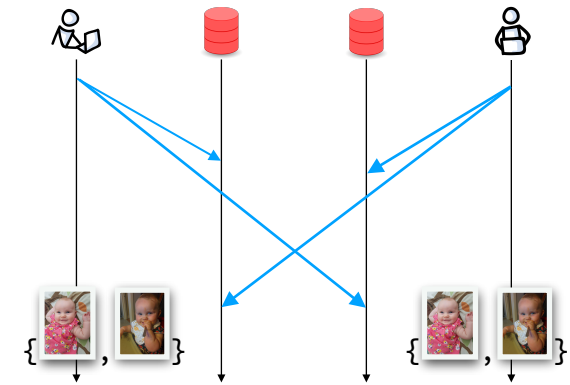
Verified **strong convergence** of CRDTs [OOPSLA 2020]

```
class CRDT t where  
  type Op t
```

```
apply :: t -> Op t -> t
```

```
lawCommut :: x:t -> op1:Op t -> op2:Op t  
  -> { _ : Proof | (apply (apply x op1) op2  
    = apply (apply x op2) op1) }
```

```
strongConvergence :: (Eq (Op a), CRDT a) =>  
  s0:a -> ops1:[Op a] -> ops2:[Op a] ->  
  { _ : Proof | isPermutation ops1 ops2 =>  
    applyAll s0 ops1 = applyAll s0 ops2 }  
strongConvergence s0 ops1 ops2 = ... -- ~300 lines
```



Once and for all:
operation-agnostic
reasoning



Verified **strong convergence** of CRDTs [OOPSLA 2020]

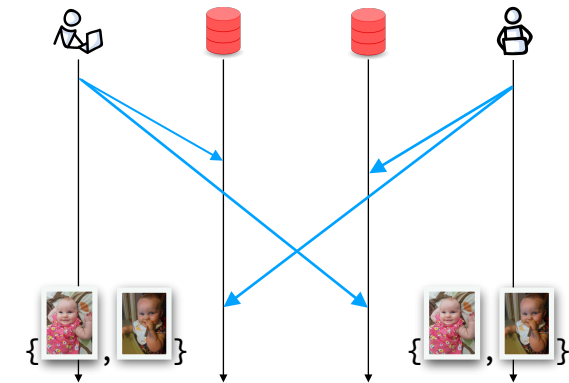
```
class CRDT t where  
  type Op t
```

```
apply :: t -> Op t -> t
```

application code

```
lawCommut :: x:t -> op1:Op t -> op2:Op t  
  -> { _ : Proof | (apply (apply x op1) op2  
    = apply (apply x op2) op1) }
```

```
strongConvergence :: (Eq (Op a), CRDT a) =>  
  s0:a -> ops1:[Op a] -> ops2:[Op a] ->  
  { _ : Proof | isPermutation ops1 ops2 =>  
    applyAll s0 ops1 = applyAll s0 ops2 }  
strongConvergence s0 ops1 ops2 = ... -- ~300 lines
```

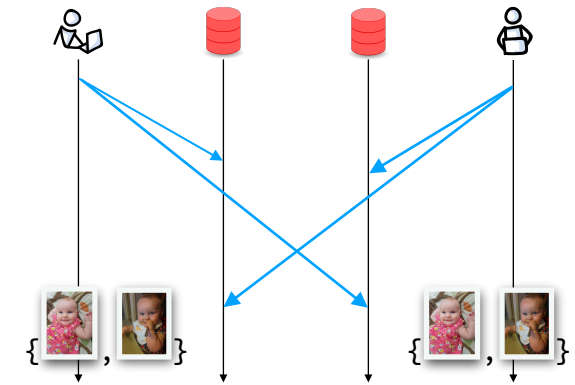


Once and for all:
operation-agnostic
reasoning



Verified **strong convergence** of CRDTs [OOPSLA 2020]

```
class CRDT t where
  type Op t
```



```
apply :: t -> Op t -> t
```

application code

```
lawCommut :: x:t -> op1:Op t -> op2:Op t
  -> { _ : Proof | (apply (apply x op1) op2
    = apply (apply x op2) op1) }
```

verification code

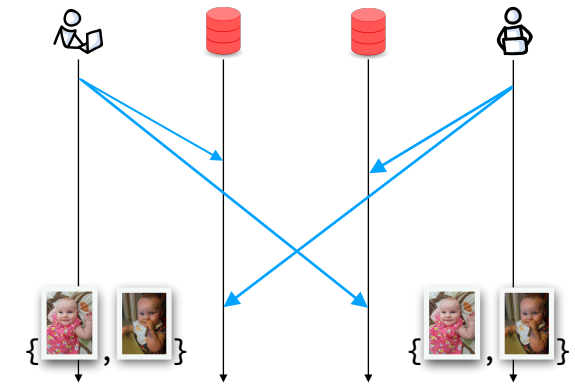
```
strongConvergence :: (Eq (Op a), CRDT a) =>
  s0:a -> ops1:[Op a] -> ops2:[Op a] ->
  { _ : Proof | isPermutation ops1 ops2 =>
    applyAll s0 ops1 = applyAll s0 ops2 }
strongConvergence s0 ops1 ops2 = ... -- ~300 lines
```

Once and for all:
operation-agnostic
reasoning



Verified **strong convergence** of CRDTs [OOPSLA 2020]

```
class CRDT t where
  type Op t
```



```
apply :: t -> Op t -> t
```

application code

```
lawCommut :: x:t -> op1:Op t -> op2:Op t
  -> { _ : Proof | (apply (apply x op1) op2
    = apply (apply x op2) op1) }
```

verification code

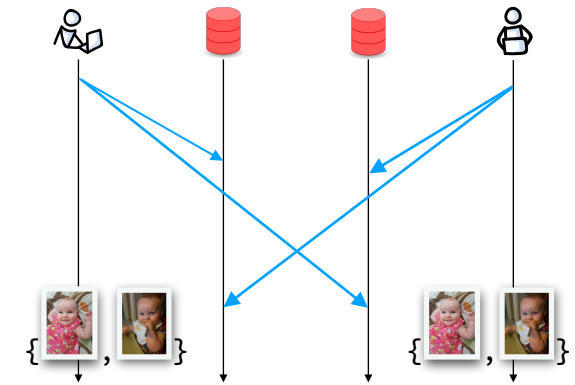
```
strongConvergence :: (Eq (Op a), CRDT a) =>
  s0:a -> ops1:[Op a] -> ops2:[Op a] ->
  { _ : Proof | isPermutation ops1 ops2 =>
    applyAll s0 ops1 = applyAll s0 ops2 }
strongConvergence s0 ops1 ops2 = ... -- ~300 lines
```

Once and for all:
operation-agnostic
reasoning



Verified **strong convergence** of CRDTs [OOPSLA 2020]

```
class CRDT t where
  type Op t
```



```
apply :: t -> Op t -> t
```

application code

```
lawCommut :: x:t -> op1:Op t -> op2:Op t
  -> { _ : Proof | (apply (apply x op1) op2
                        = apply (apply x op2) op1) }
```

verification code

```
strongConvergence :: (Eq (Op a), CRDT a) =>
  s0:a -> ops1:[Op a] -> ops2:[Op a] ->
  { _ : Proof | isPermutation ops1 ops2 =>
    applyAll s0 ops1 = applyAll s0 ops2 }
strongConvergence s0 ops1 ops2 = ... -- ~300 lines
```

Once and for all:
operation-agnostic
reasoning



Verified **strong convergence** of CRDTs [OOPSLA 2020]

```
instance Num a => CRDT (Counter a) where
  type Op (Counter a) = Counter a

  apply (Counter a) (Counter b) = Counter (a + b)

  lawCommut x op1 op2 = ()
```

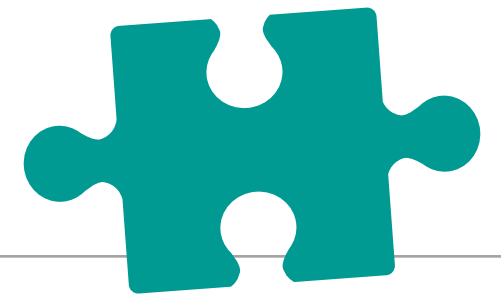
Verified **strong convergence** of CRDTs [OOPSLA 2020]

```
instance Num a => CRDT (Counter a) where
  type Op (Counter a) = Counter a

  apply (Counter a) (Counter b) = Counter (a + b)

  lawCommut x op1 op2 = ()
```

Once for each CRDT:
operation-specific
commutativity



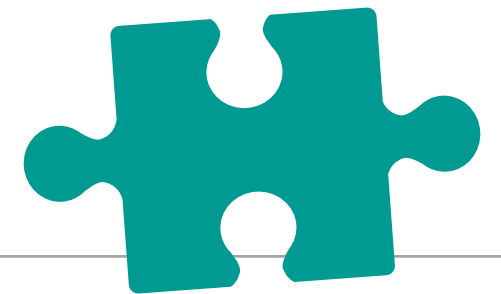
Verified **strong convergence** of CRDTs [OOPSLA 2020]

```
instance Num a => CRDT (Counter a) where
  type Op (Counter a) = Counter a

  apply (Counter a) (Counter b) = Counter (a + b)

  lawCommut x op1 op2 = ( ) 😎
```

Once for each CRDT:
operation-specific
commutativity



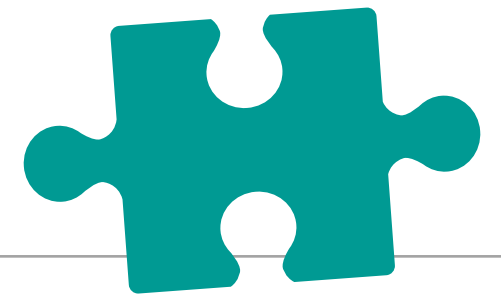
Verified **strong convergence** of CRDTs [OOPSLA 2020]

```
instance Num a => CRDT (Counter a) where
  type Op (Counter a) = Counter a

  apply (Counter a) (Counter b) = Counter (a + b)

  lawCommut x op1 op2 = () 😎
```

Once for each CRDT:
operation-specific
commutativity

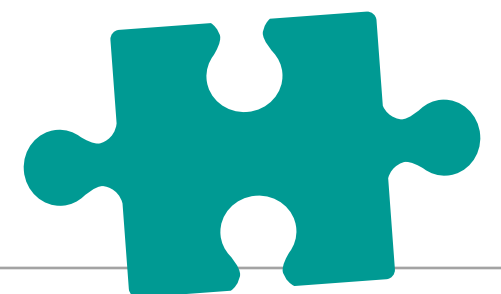


```
data DictOp k v = Insert k v | Update k (Op v) | Delete k

instance (Ord k, Ord (Op v), CRDT v) => CRDT (Dict k v) where
  type Op (Dict k v) = DictOp k v

  apply x op = ... -- ~50 lines

  lawCommut x op1 op2 = ... -- ~1200 lines
```



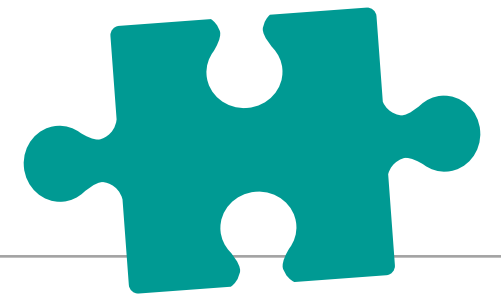
Verified **strong convergence** of CRDTs [OOPSLA 2020]

```
instance Num a => CRDT (Counter a) where
  type Op (Counter a) = Counter a

  apply (Counter a) (Counter b) = Counter (a + b)

  lawCommut x op1 op2 = () 😎
```

Once for each CRDT:
operation-specific
commutativity

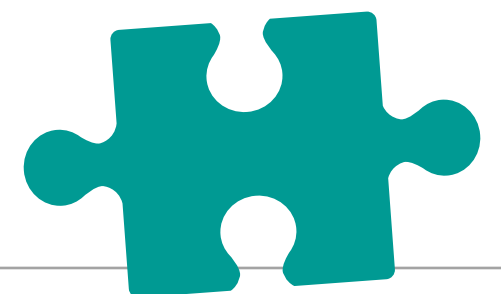


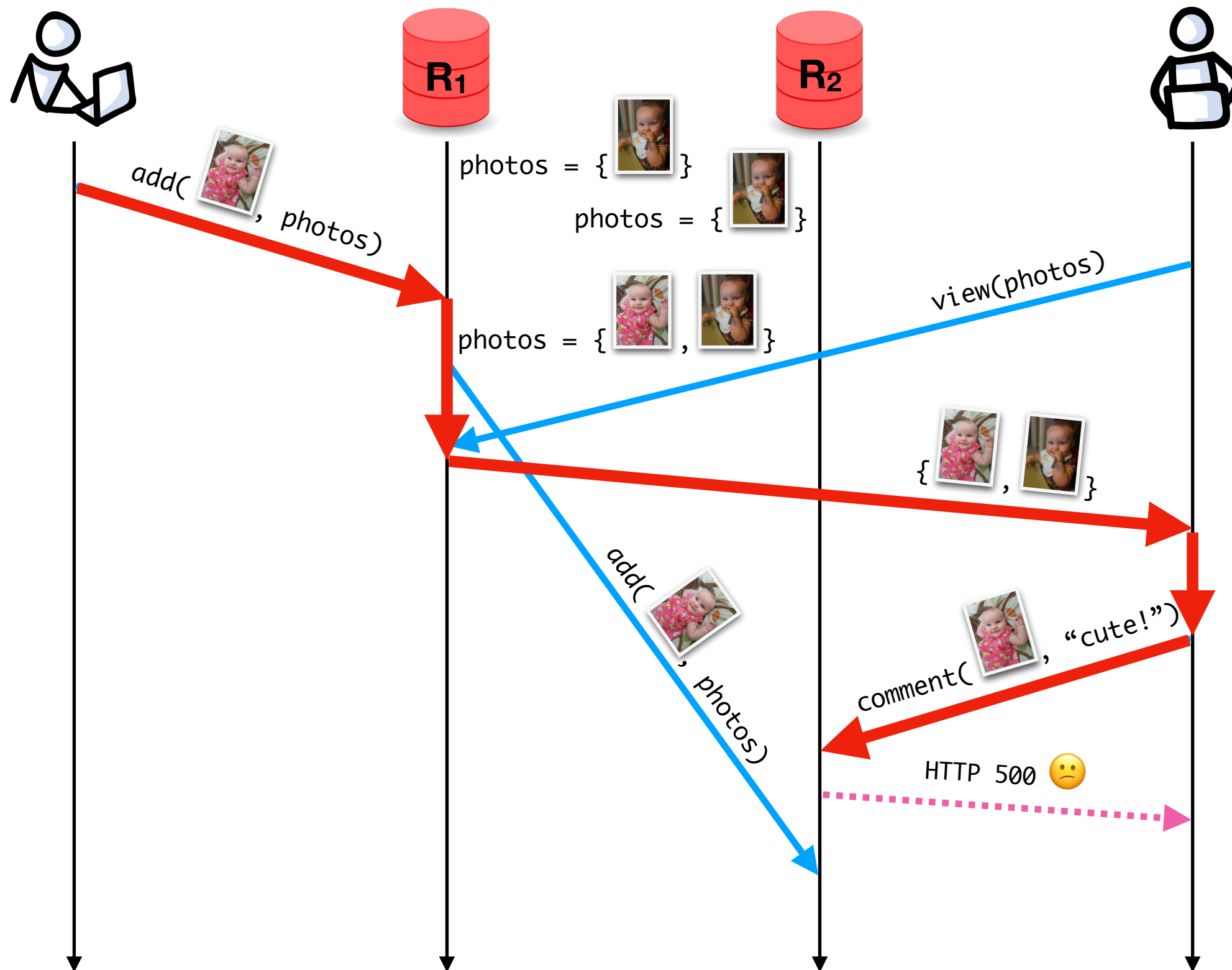
```
data DictOp k v = Insert k v | Update k (Op v) | Delete k

instance (Ord k, Ord (Op v), CRDT v) => CRDT (Dict k v) where
  type Op (Dict k v) = DictOp k v

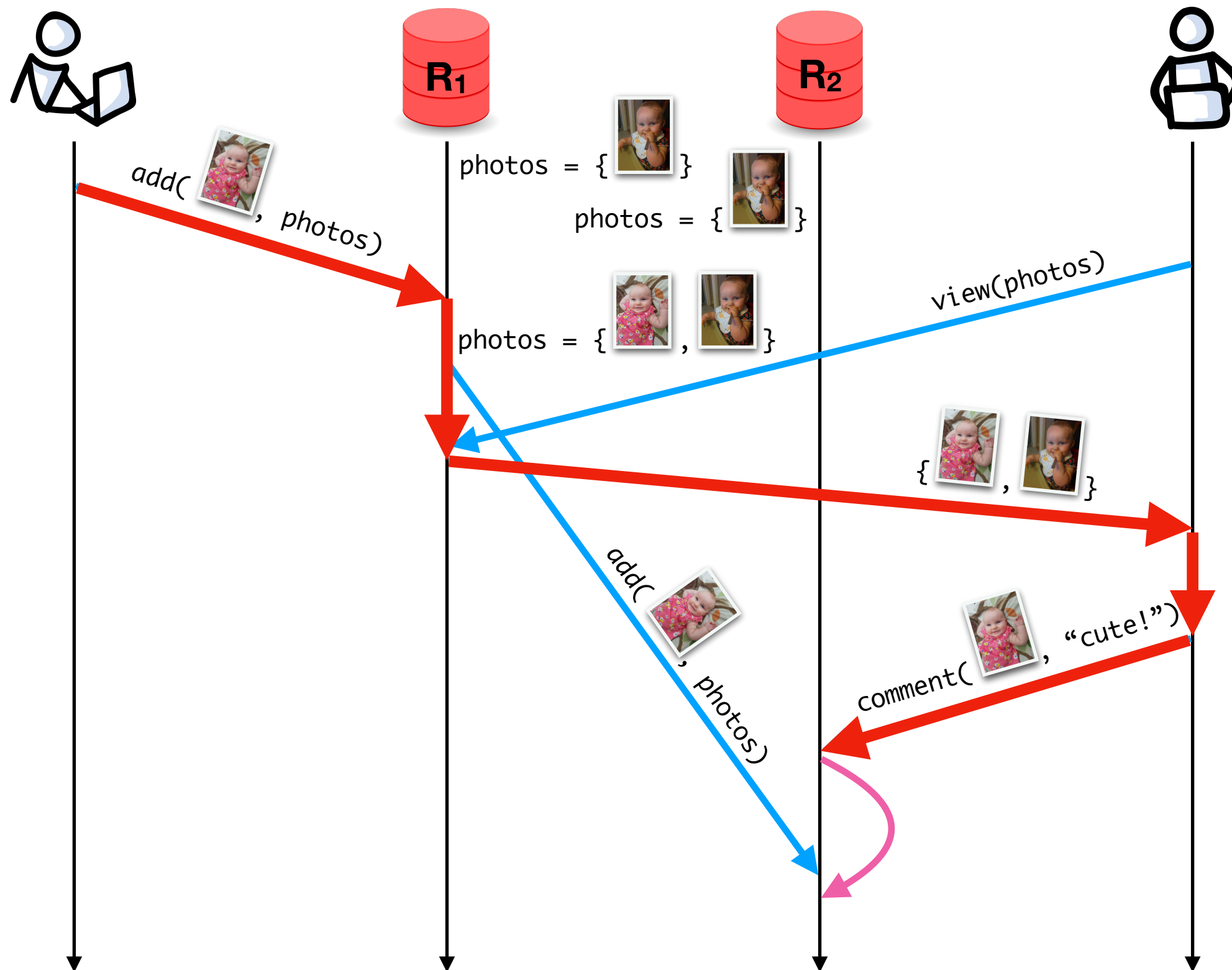
  apply x op = ... -- ~50 lines

  lawCommut x op1 op2 = ... -- ~1200 lines 😓
```

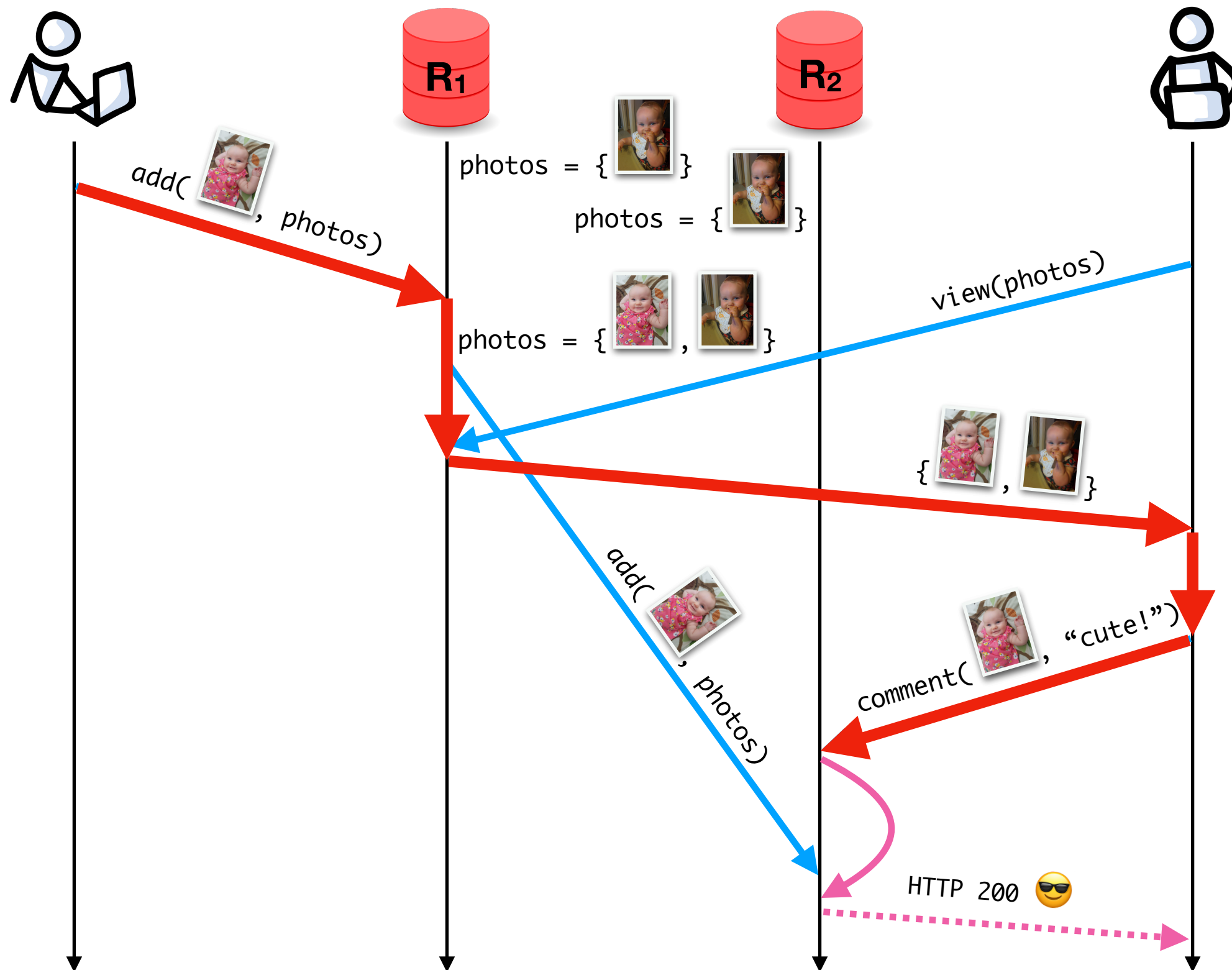




Causal delivery
[Birman and Joseph, 1987]



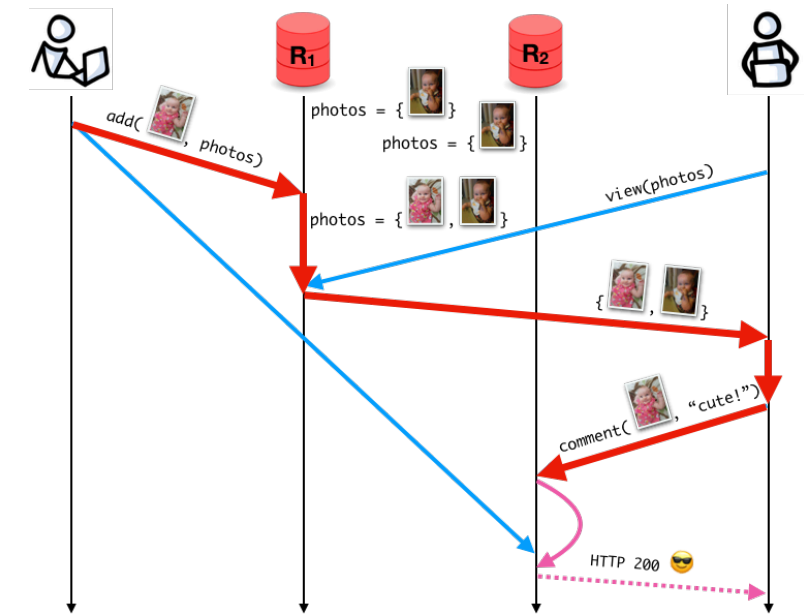
Causal delivery
[Birman and Joseph, 1987]



Causal delivery
[Birman and Joseph, 1987]

Verified causal message delivery

Collaborators: Patrick Redmond, Gan Shen, Niki Vazou



5.1 CBCAST Protocol

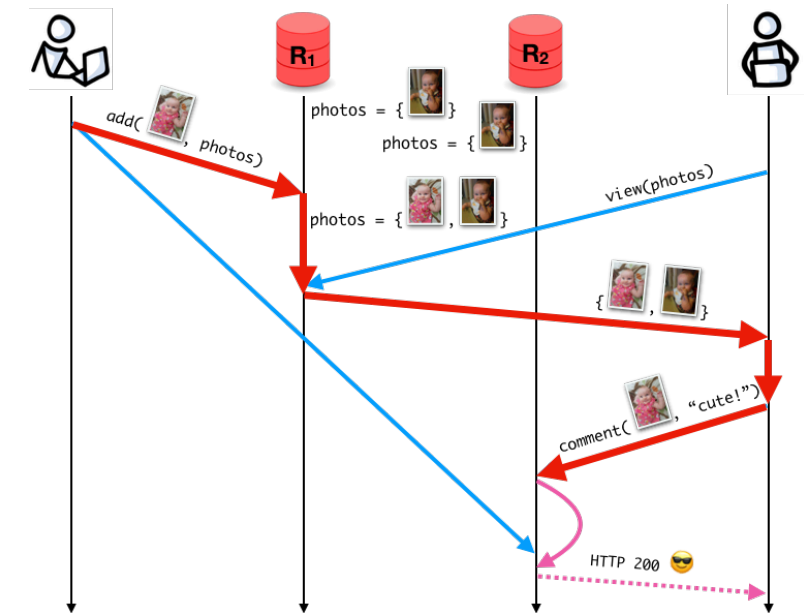
Suppose that a set of processes P communicate using only broadcasts to the full set of processes in the system; that is, $\forall m: \text{dests}(m) = P$. We now develop a *delivery protocol* by which each process p receives messages sent to it, delays them if necessary, and then delivers them in an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

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

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

Verified causal message delivery

Collaborators: Patrick Redmond, Gan Shen, Niki Vazou



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:

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

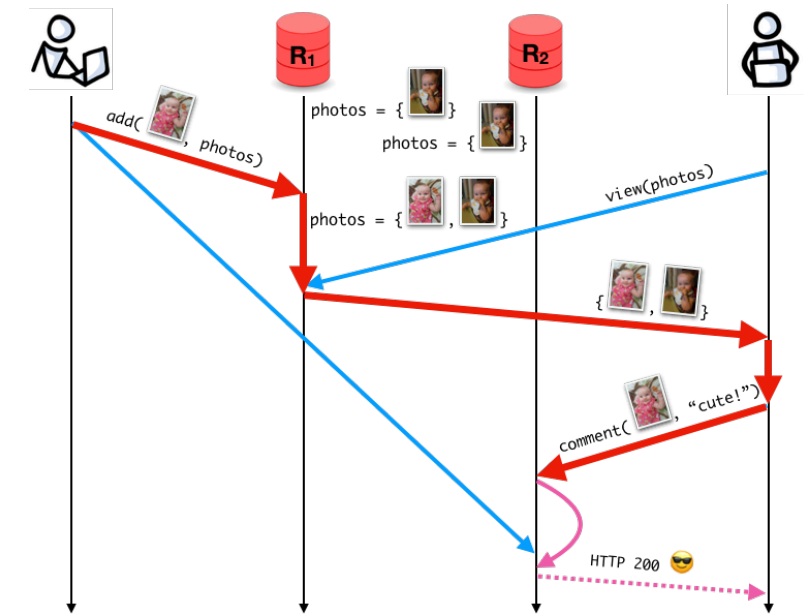
Birman et al., “Lightweight Causal and Atomic Group Multicast”
ACM TOCS, 1991

```

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') }
    
```


Verified causal message delivery

Collaborators: Patrick Redmond, Gan Shen, Niki Vazou



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:

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

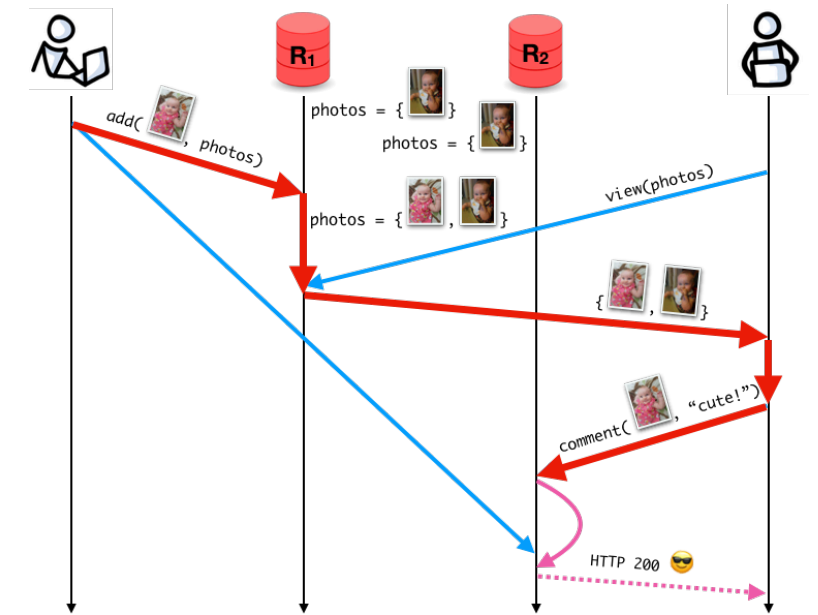
Birman et al., “Lightweight Causal and Atomic Group Multicast”
ACM TOCS, 1991

```

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') }
    
```

Verified causal message delivery

Collaborators: Patrick Redmond, Gan Shen, Niki Vazou

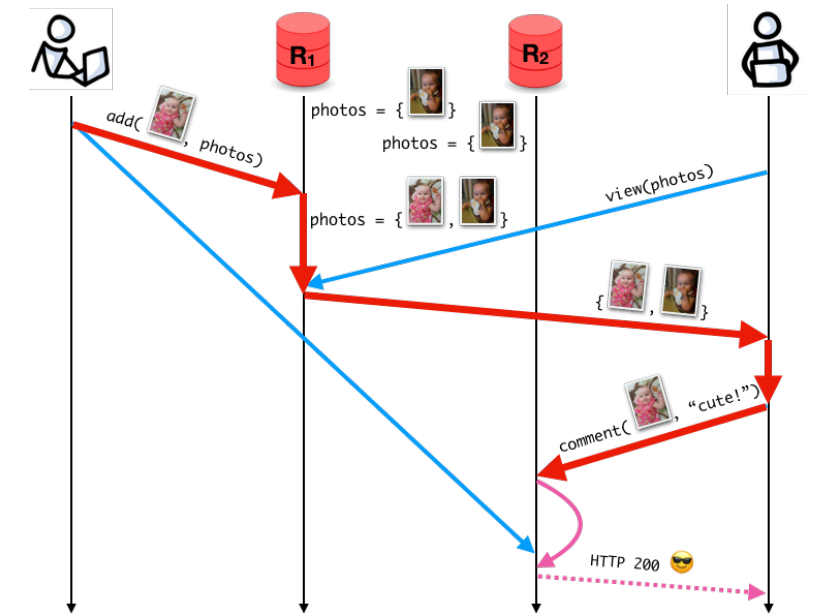


```

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') }
  
```

Verified causal message delivery

Collaborators: Patrick Redmond, Gan Shen, Niki Vazou



```

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') }
    
```

```

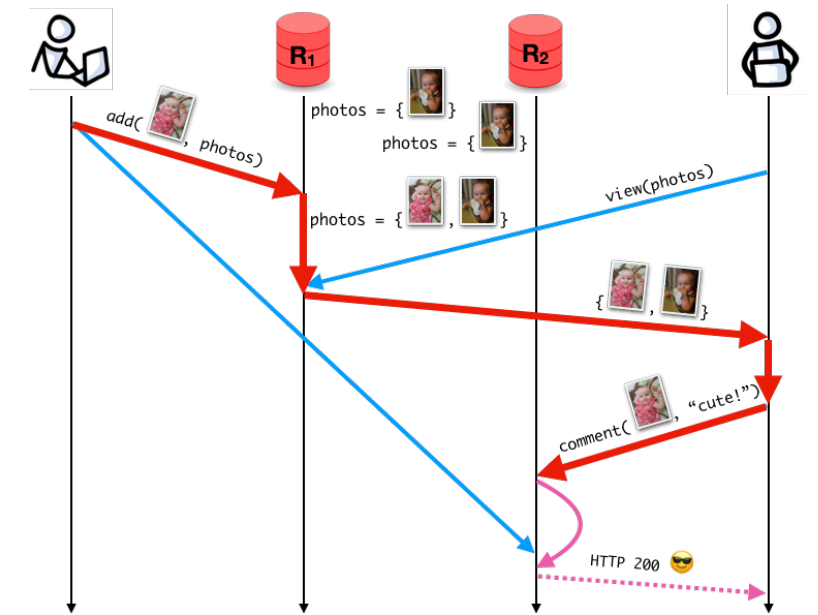
data Op = OpReceive (Message) | OpDeliver | OpBroadcast
    
```

```

step :: Op -> Process -> Process
    
```

Verified causal message delivery

Collaborators: Patrick Redmond, Gan Shen, Niki Vazou



```

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

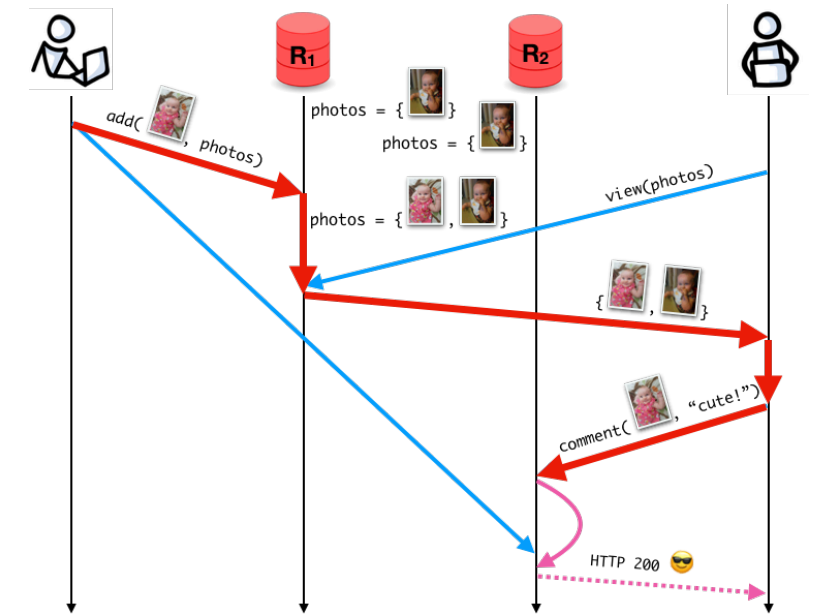
step :: Op -> Process -> Process

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

causalDeliveryPreservation = ... -- ~300 lines
  
```

Verified causal message delivery

Collaborators: Patrick Redmond, Gan Shen, Niki Vazou



```

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 = ... -- ~300 lines
  
```

What's next?

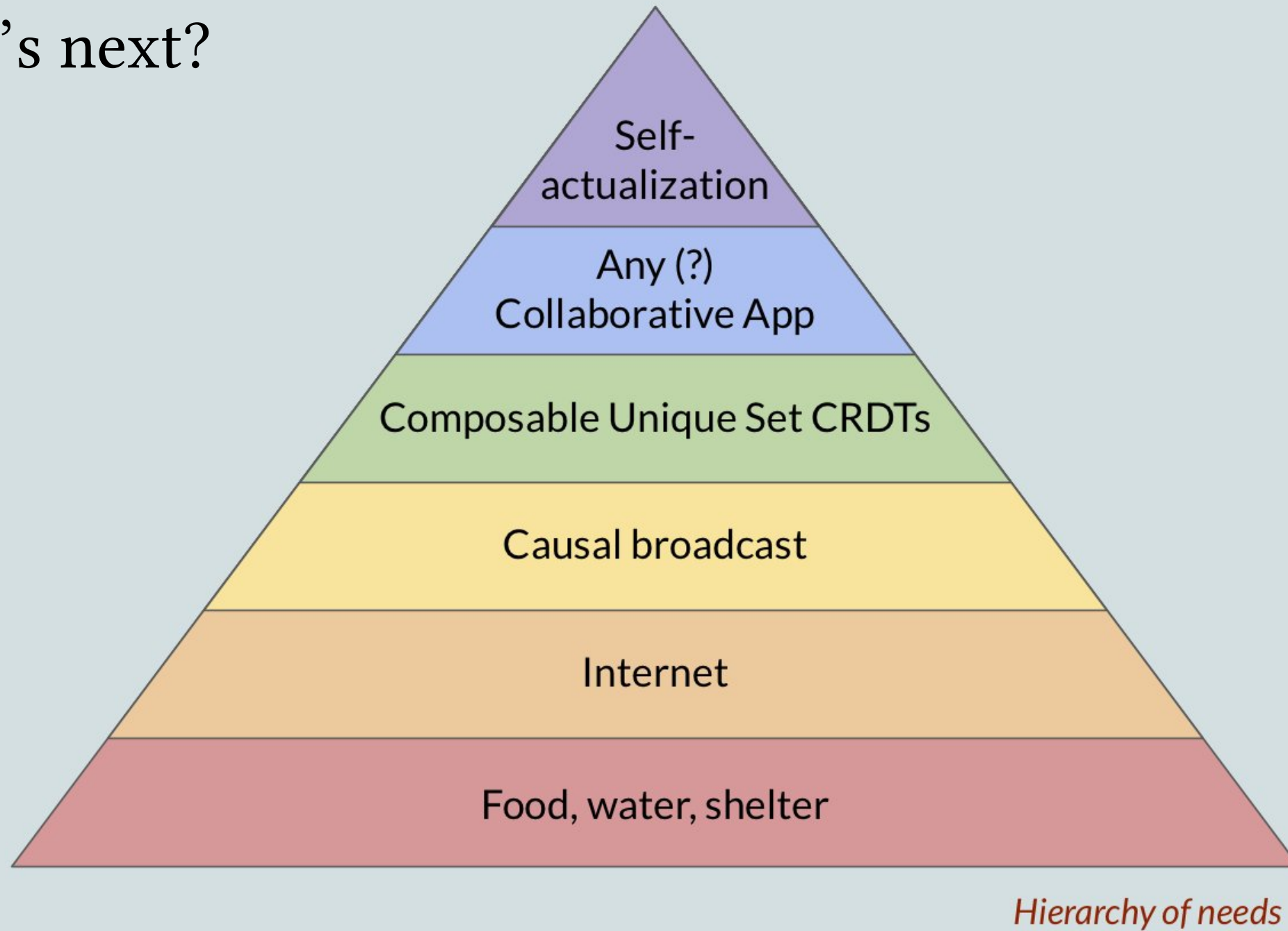


Image credit: Matthew Weidner

What's next?

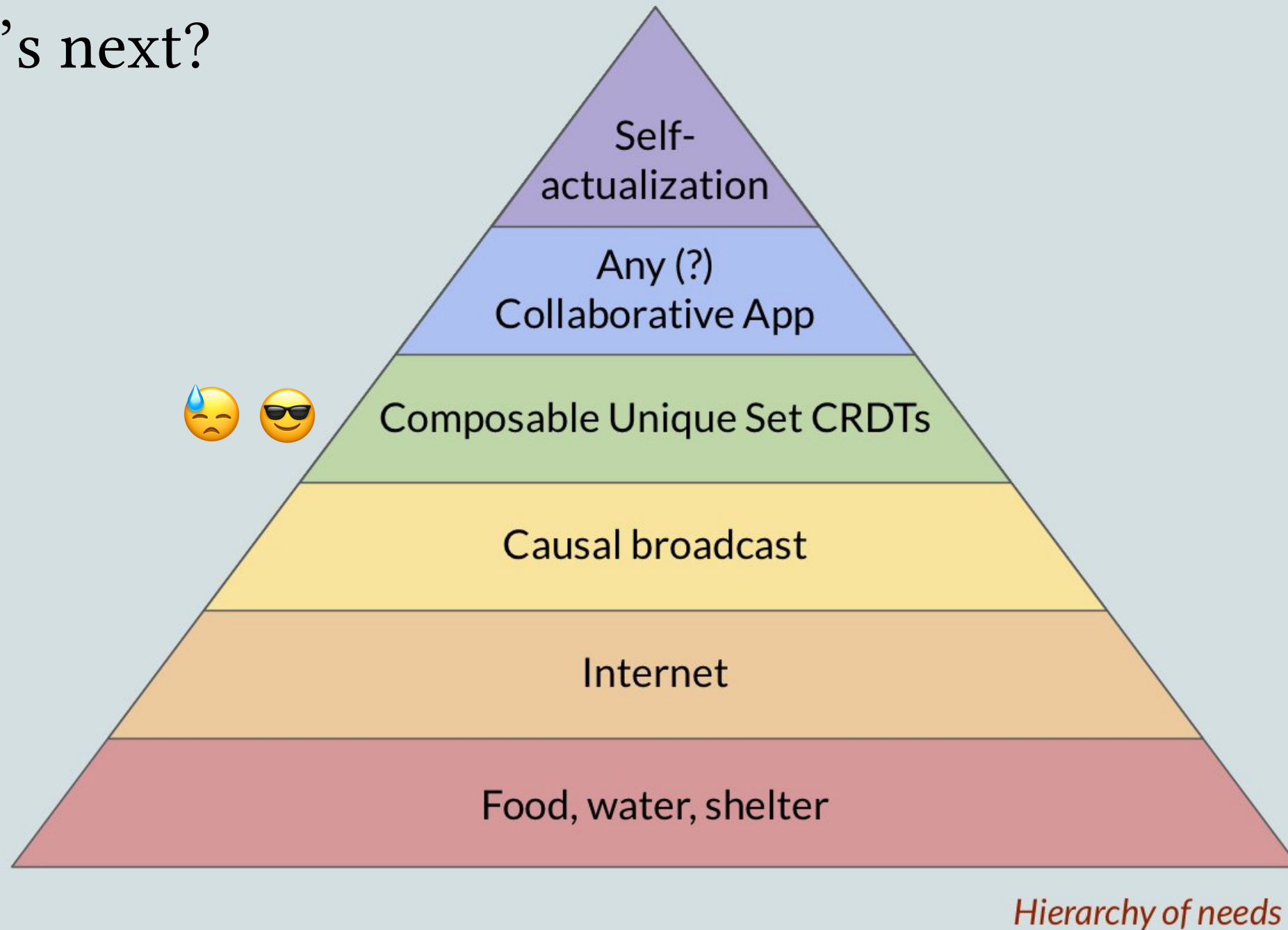


Image credit: Matthew Weidner

What's next?

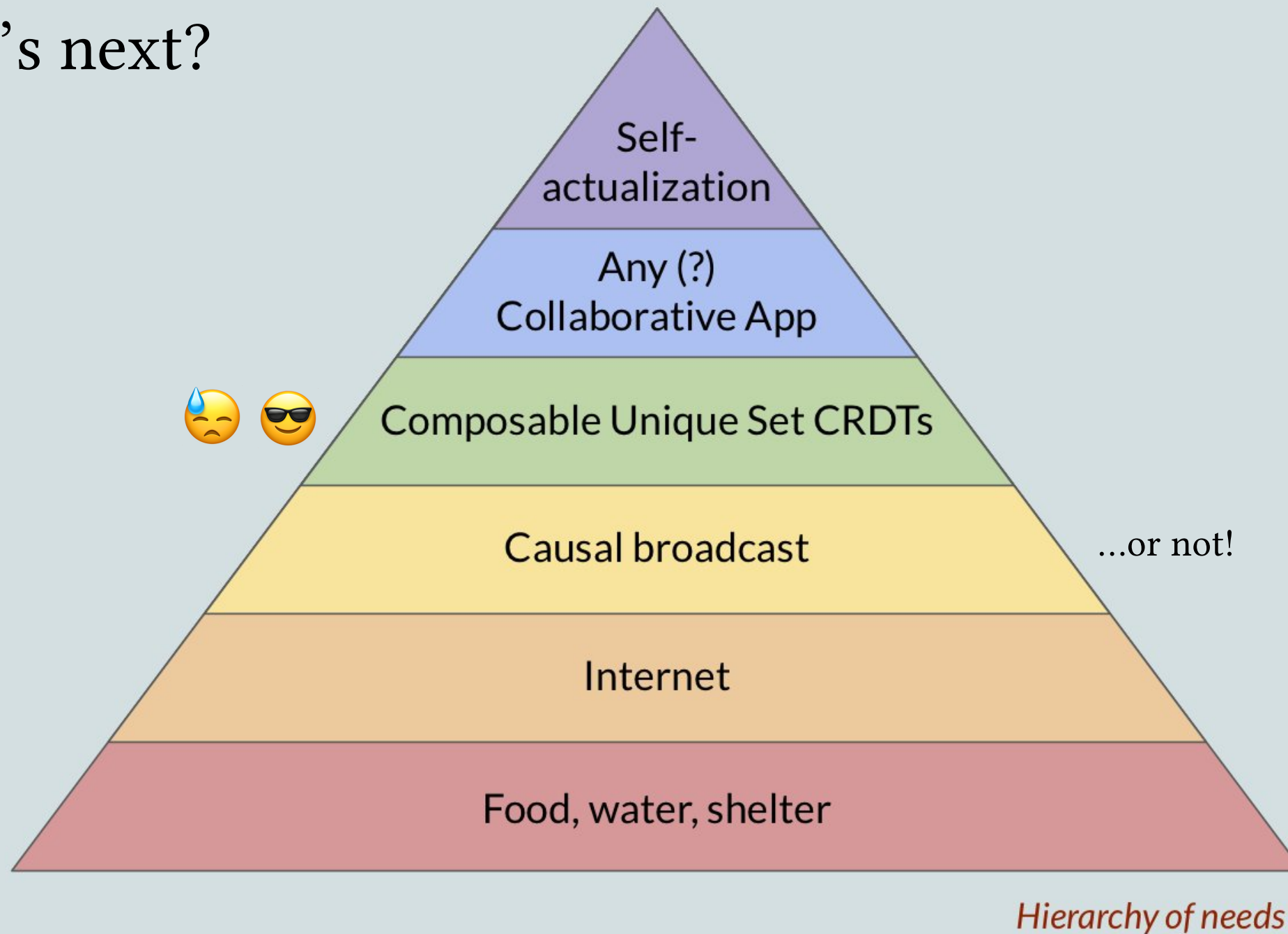


Image credit: Matthew Weidner

What's next?

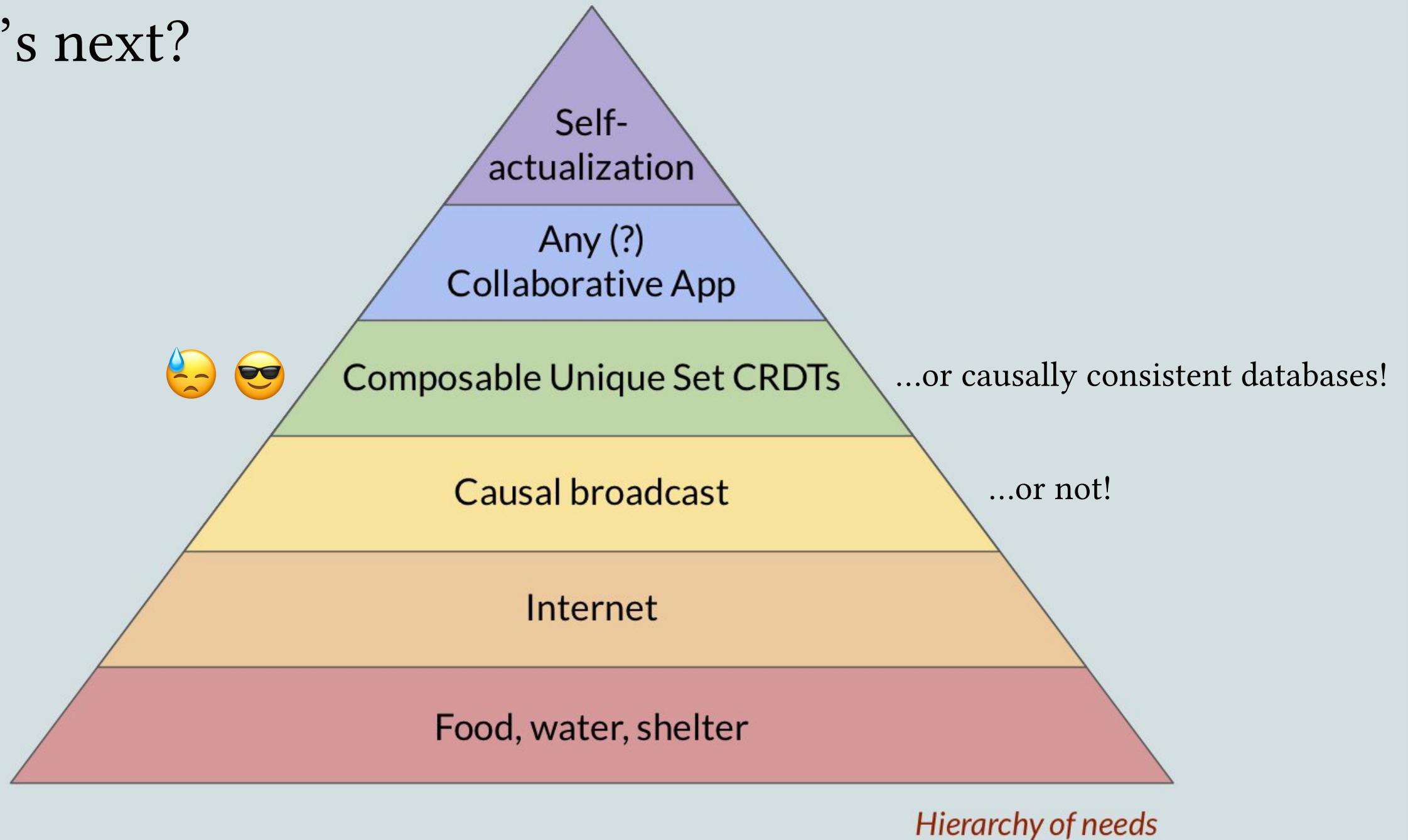


Image credit: Matthew Weidner

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

Thank you!

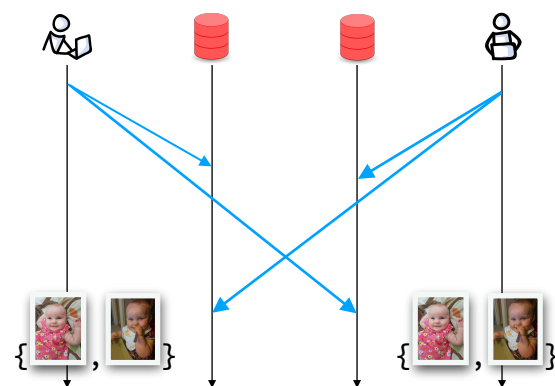
Languages, Systems, and Data Lab: lsd.ucsc.edu

Blog: composition.al

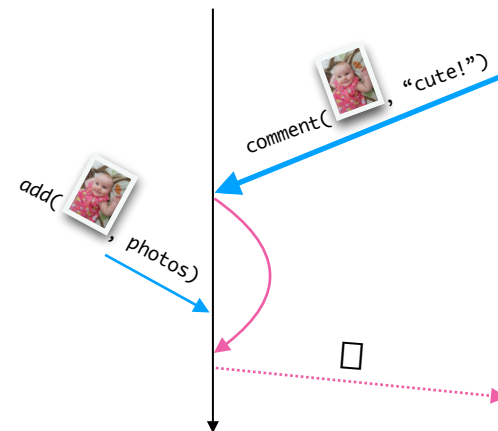
Thank you!

Languages, Systems, and Data Lab: lsd.ucsc.edu

Blog: composition.al



verified **strongly convergent**
replicated data structures



verified **causal delivery**