A gentle introduction to *Conflict-free replicated data types*

Guillaume Bogard - guillaumebogard.dev

Bonjour! 👋

My name is Guillaume Bogard

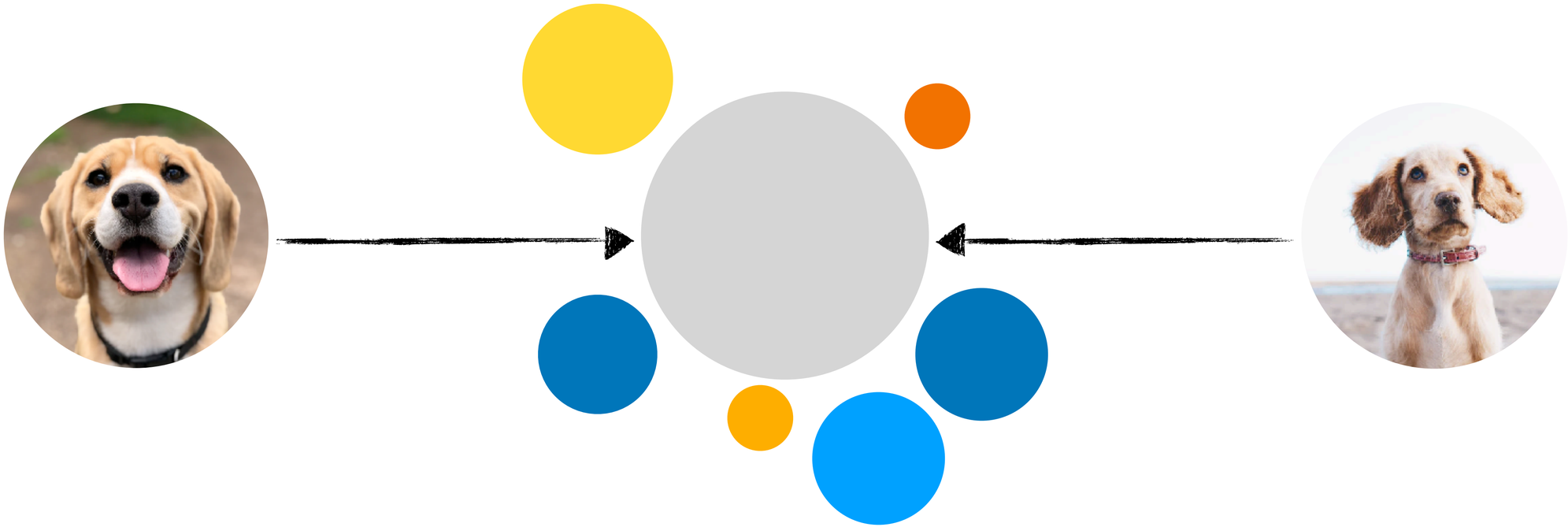
I’m building a *media asset management* platform @Canal+

Scala by day, Haskell by night

Addicted to League of Legends since 16, please send help

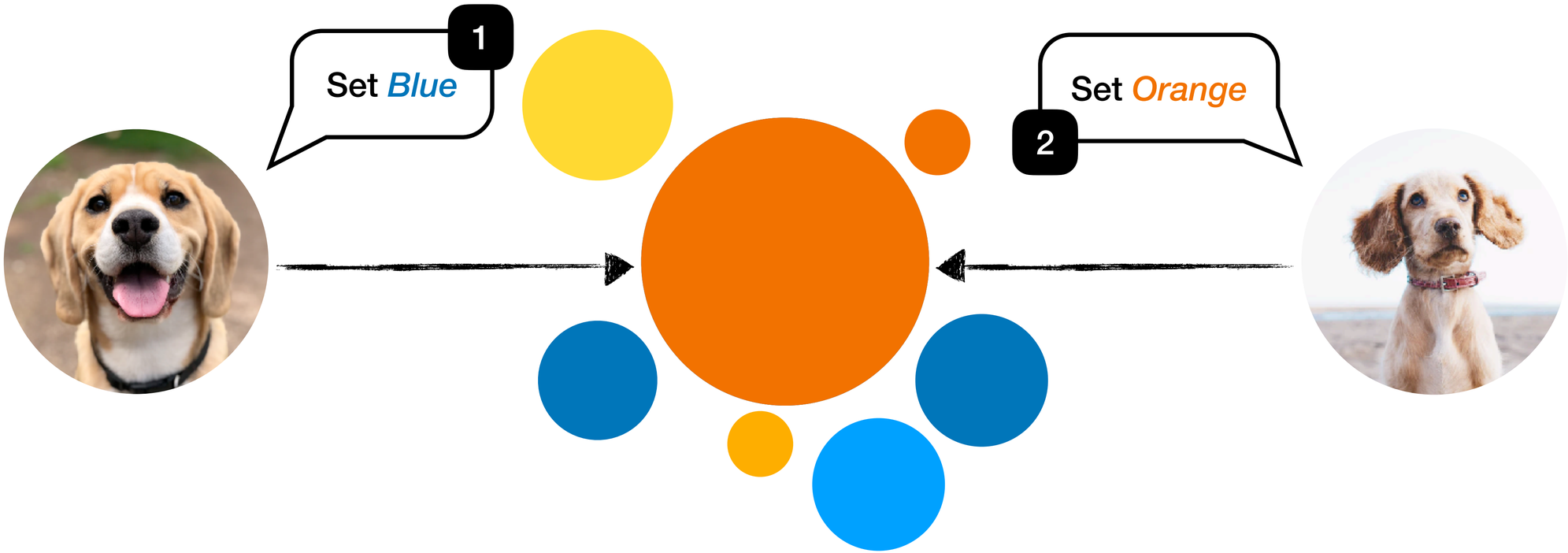
Why CRDTs ?

Motivation and terminology



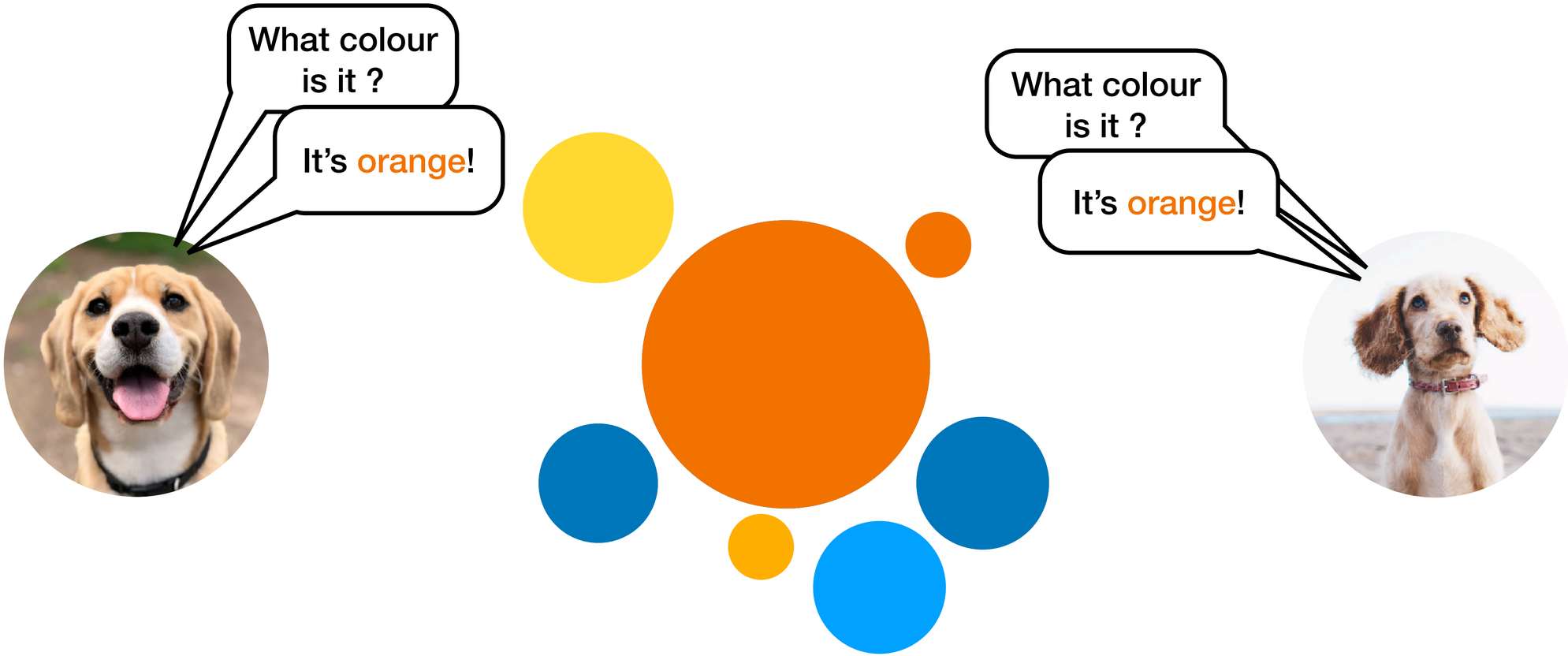
Stanley and Lily are both editing the same piece of data, in the same system

Strong consistency



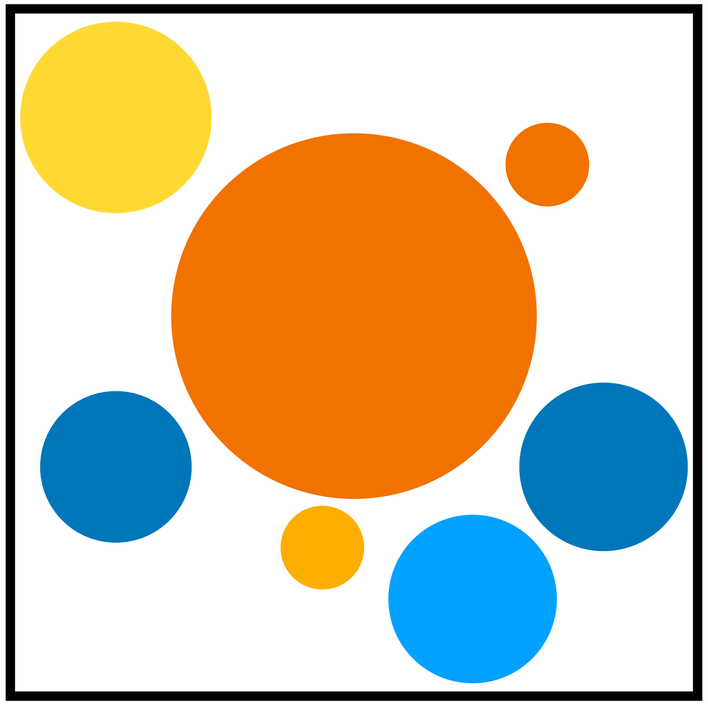
Operations are strictly sequential

All participants see the same order



As a result, all read requests always return the same, consistent state

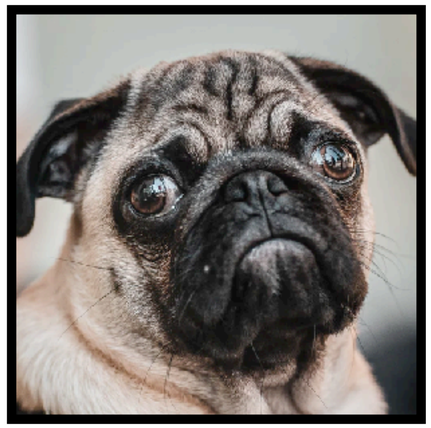
Strong consistency in a single-node system is *relatively easy*



Many RDMS (e.g Postgres) already offer
Atomic, Consistent, Isolated and Durable (ACID) transactions

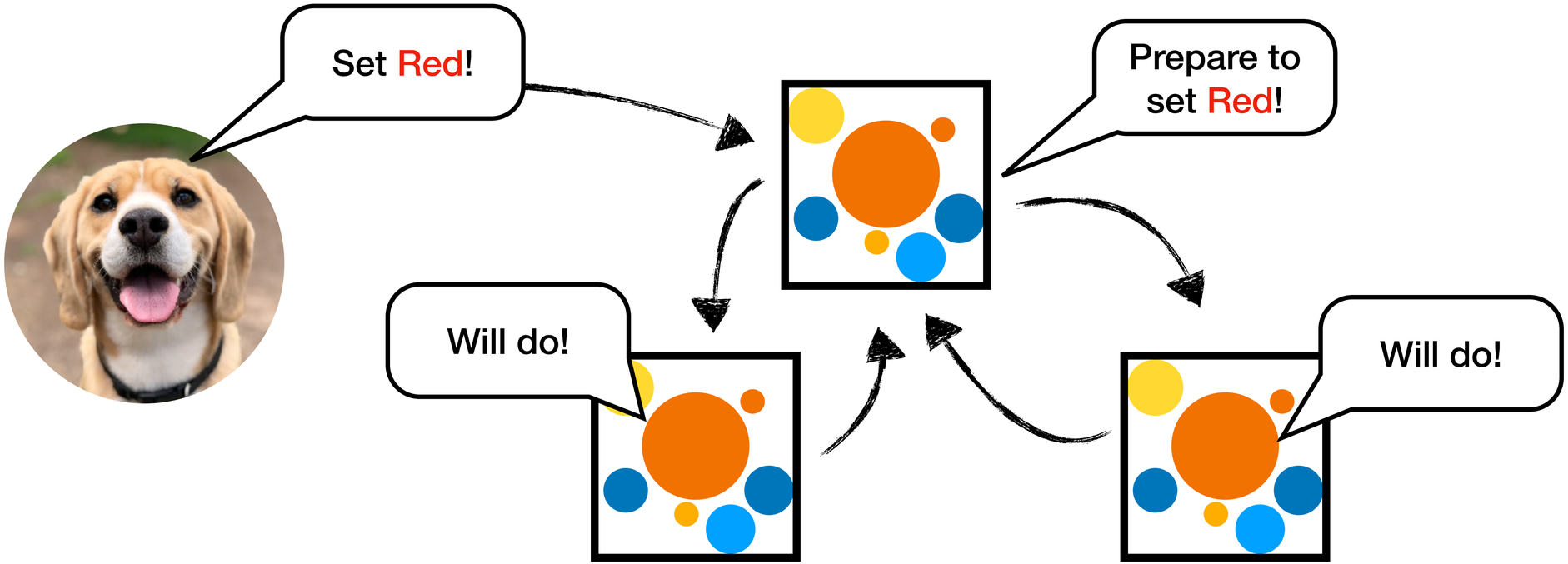
But a single-node system is also way more likely to fail

⇒



The only way to achieve
high-availability is with a distributed system

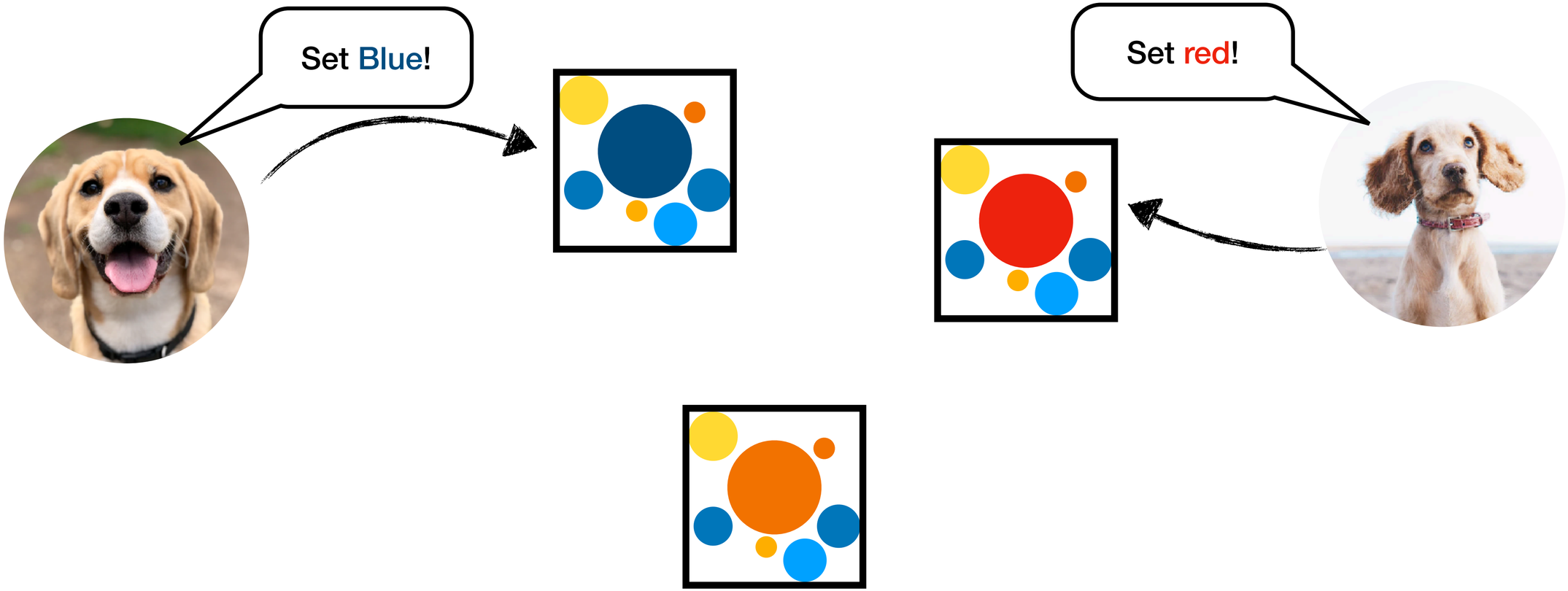
Strong consistency in a distributed system can be obtained through *consensus protocols* (e.g. Raft, Paxos, PBFT)



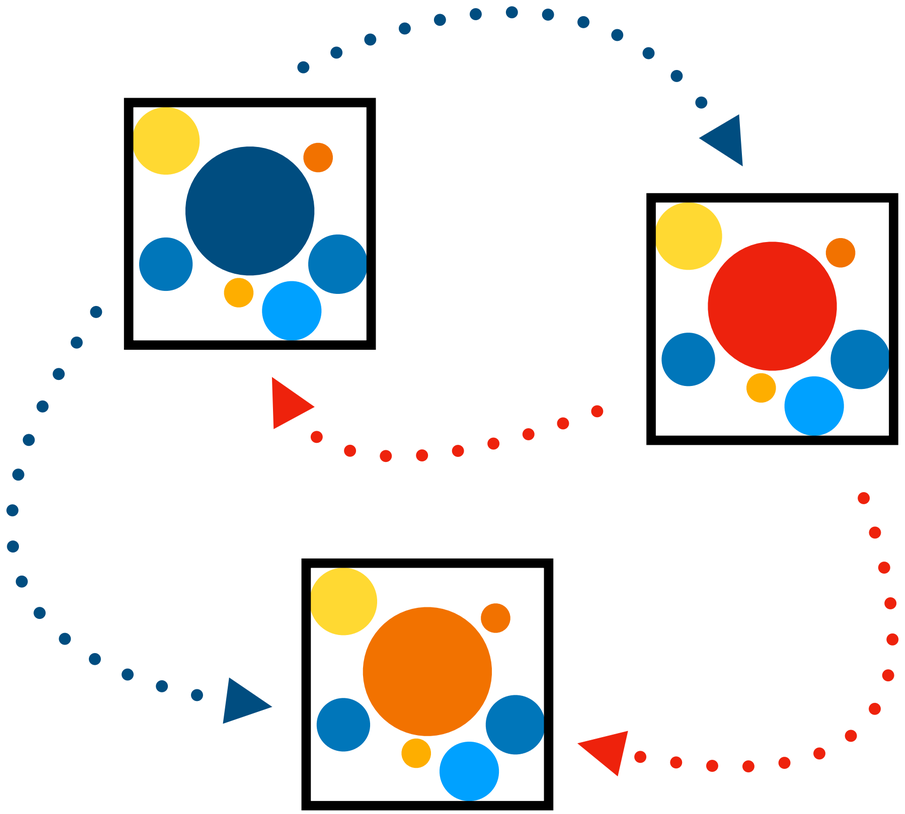
- The system will continue to work as long as *(n / 2) + 1* nodes are alive
- Consensus algorithms typically require a lot of interaction between the nodes
- Consensus algorithms offer high availability and data consistency, but relatively poor throughput and relatively high latency

Eventual consistency

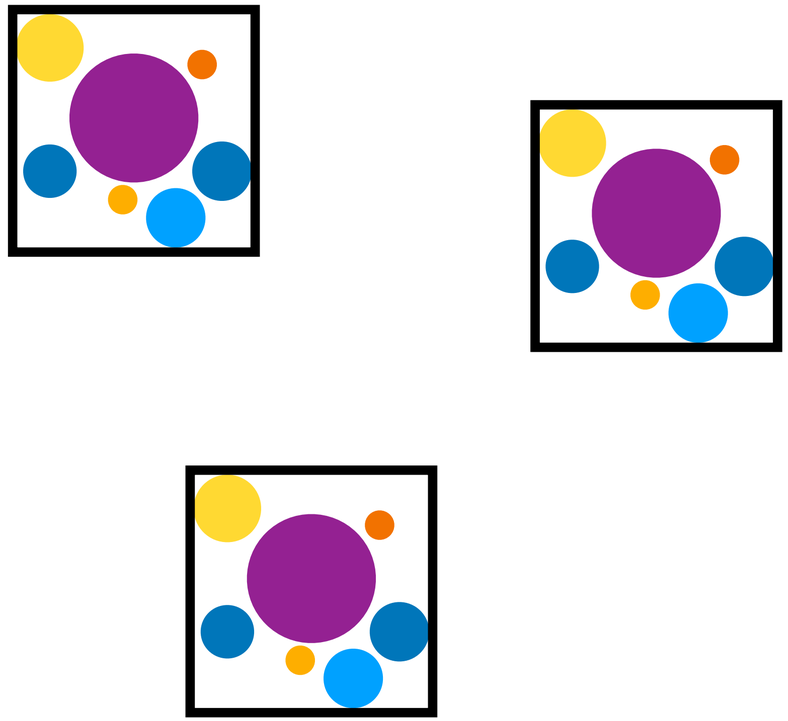
Updates are performed locally,
without any coordination with other nodes



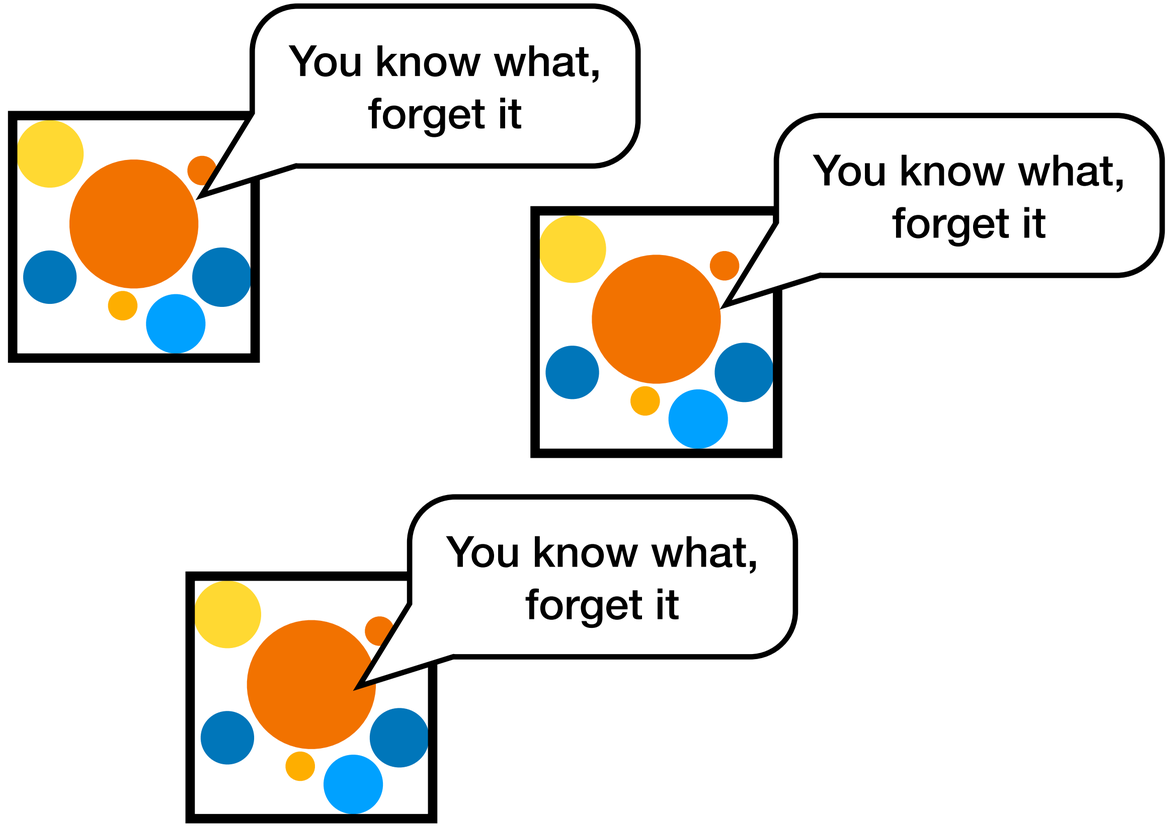
Changes are *eventually* propagated across the system



Conflicting changes are *somehow* resolved (● + ● = ●) …



Or abandoned altogethe



Since updates are asynchronously propagated,
users can observe different states at the same time

		✅	❌
Single node		Consistent data Operational simplicity	Failure-prone Vertical scalability only
Distributed	Strong consistency (Real-time consensus)	Consistent data High-availability	Higher latency Relatively low throughput Harder to implement (assuming you want to roll your own system)
	Eventual Consistency (Deferred consensus)	High-availability Highest throughput Lowest latency Suitable for offline-first apps	State isn’t immediately consistent Possible data losses due to conflicts

		✅	❌
Single node		Consistent data Operational simplicity	Failure-prone Vertical scalability only
Distributed	Strong consistency (Real-time consensus)	Consistent data High-availability	Higher latency Relatively low throughput Harder to implement (assuming you want to roll your own system)
	Eventual Consistency (Deferred consensus)	High-availability Highest throughput Lowest latency Suitable for offline-first apps	State isn't immediately consistent Possible data losses due to conflicts
	Something else?		

Strong eventual consistency

Eventual consistency

\+ *Automatic conflict resolution that always works*

= ***Strong eventual consistency***

Automatic conflict resolution that always works
is exactly what CRDTs are designed to achieve.

Two families of CRDTs

State-based and event-based

State-based CRDTs

Objects that can be merged with objects of the same type to produce a new object of that type.

The merge operation can be executed in any order, and any number of times.

State-based CRDTs

More formally

A tuple ($\mathbf{S}$, $\mathbf{s0}$, $\mathbf{q}$, $\mathbf{u}$, $\mathbf{m}$) where

- S is the set of possible states for the CRDT

- $\mathrm{s0}$ is the initial state of the CRDT

- q is the *query* function that lets a client read the current state of the object

- u is the *update* function that lets a client alter the state of the object

- m is a binary *merge* function

State-based CRDTs

More formally

m needs to be **associative**

$$\forall x, y, z : merge(x, (merge(y, z))) = merge(merge(x, y), z)$$

State-based CRDTs

More formally

m needs to be **commutative**

$$\forall x, y : merge(x, y) = merge(y, x)$$

State-based CRDTs

More formally

m needs to be **idempotent**

$$\forall x : merge(x, x) = x$$

State-based CRDTs

From a programmer's perspective

```haskell
-- | A state-based CRDT is a structure that has a binary 'merge' operation that is
-- | * associative
-- | * commutative
-- | * idempotent
class StateBasedCRDT t where
  merge :: t → t → t
```

Operation-based CRDTs

Objects whose modifications are described by *operations*

Operations can be sent over the network and applied locally by each replica, using a function called the *effector*

Any two concurrent operations must commute, meaning they can be applied in any order

Operation-based CRDTs

Concurrent operations are commutative
but not necessarily idempotent.

We need to keep that in mind when choosing the transport protocol in a real distributed system

Operation-based CRDTs

From a programmer's perspective

```haskell
-- | An effector applies an operation on a state to produce a new state
class Effector t op | t → op where
  applyOperation :: op → t → t

-- | A generator is a way of expressing state modifications
-- | as  operations that can be sent over the network
class Generator t op | t → op where
  generateOperations :: t → t → Array op
```

Operation-based CRDTs

From a programmer's perspective

```
-- | An effector applies an operation on a state to produce a new state
class Effector t op | t → op where
  applyOperation :: op → t → t

-- | A generator is a way of expressing state modifications
-- | as  operations that can be sent over the network
class Generator t op | t → op where
  generateOperations :: t → t → Array op
```

This talk mainly focuses
State-based CRDTS, however …

"Any SEC state-based object can be emulated by a SEC op-based object of a corresponding interface"

Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski. Conflict-free Replicated Data Types (2014)

"Any SEC op-based object can be emulated by a SEC state-based object of a corresponding interface"

Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski. Conflict-free Replicated Data Types (2014)

Choosing between state-based & event-based mostly depends how your updates are transported

(State-based use more bandwidth, op-based require exactly once delivery …)

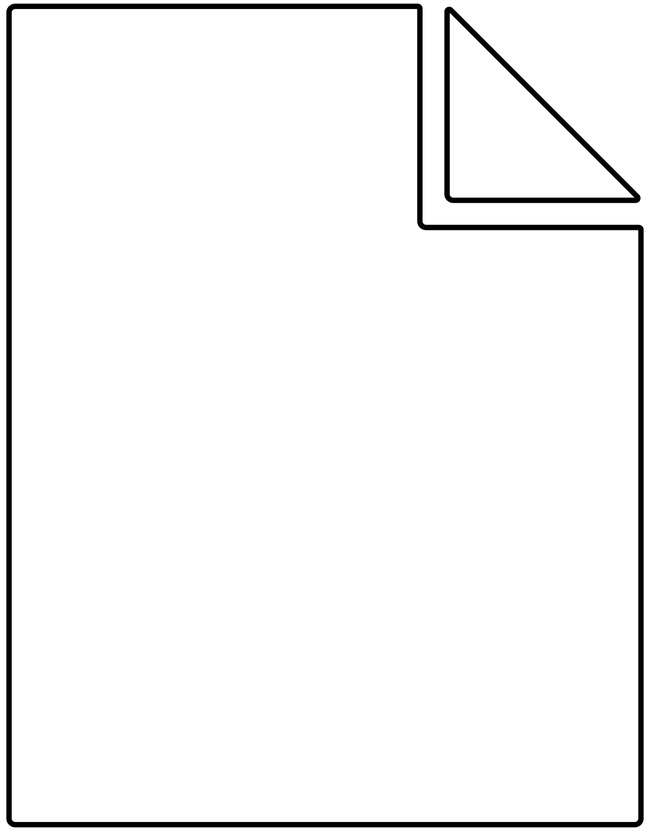
The simplest known CRDTs

Not necessarily the most useful though

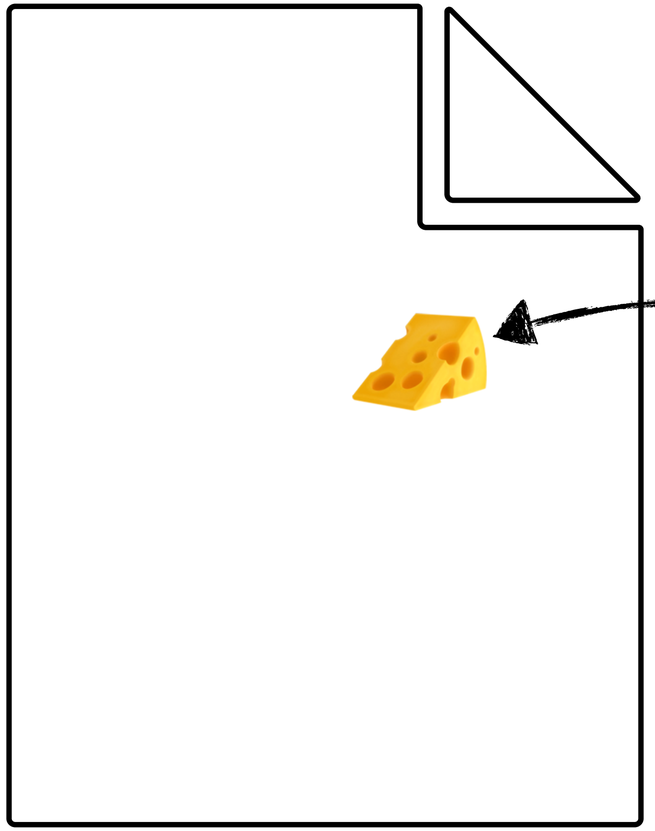
The G-Set

Grow-only set

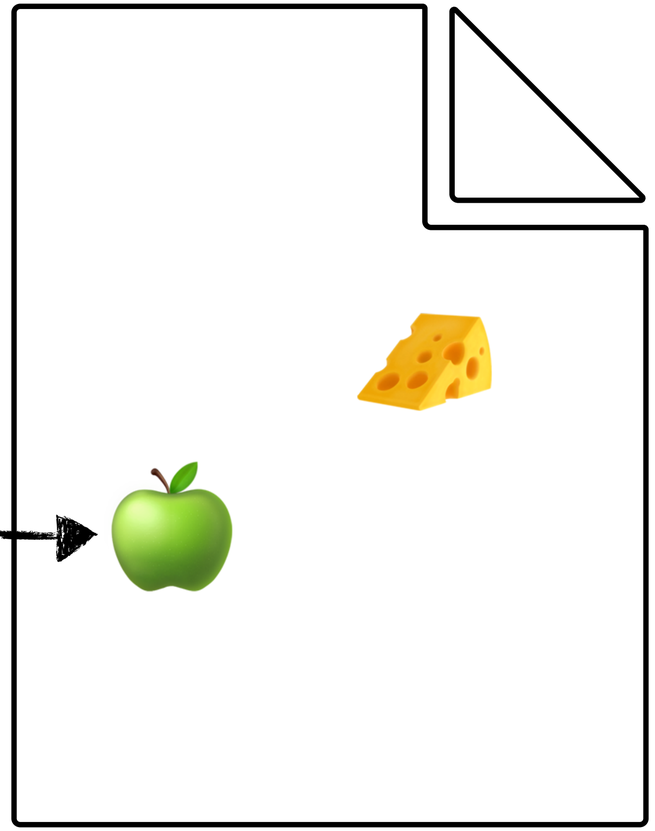
Stanley and Lily are working online on the same list



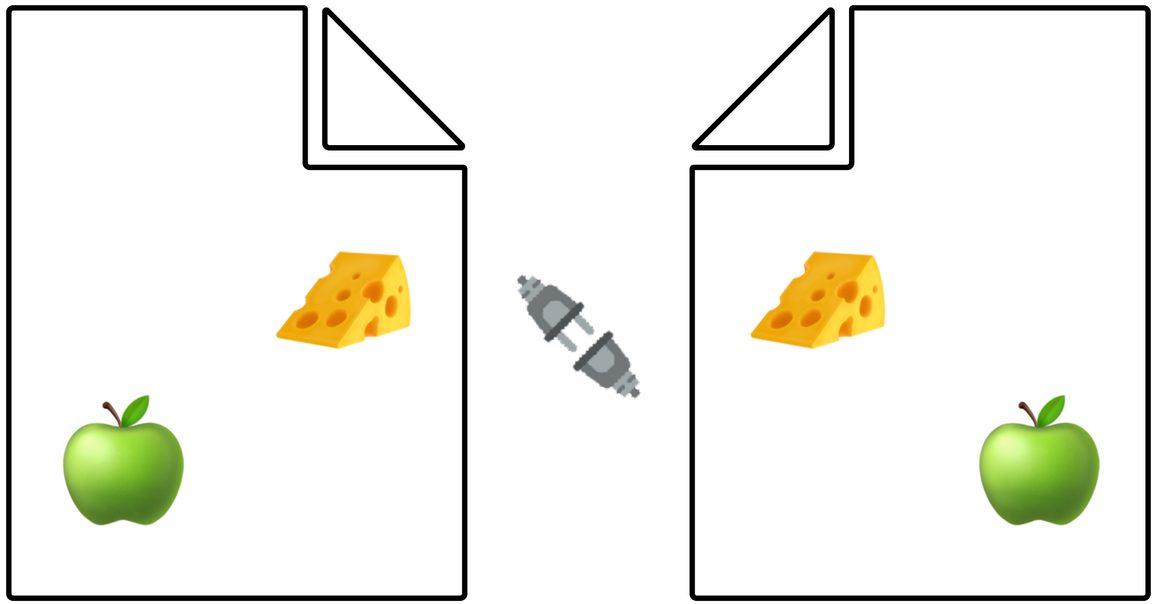
Lily adds 🧀



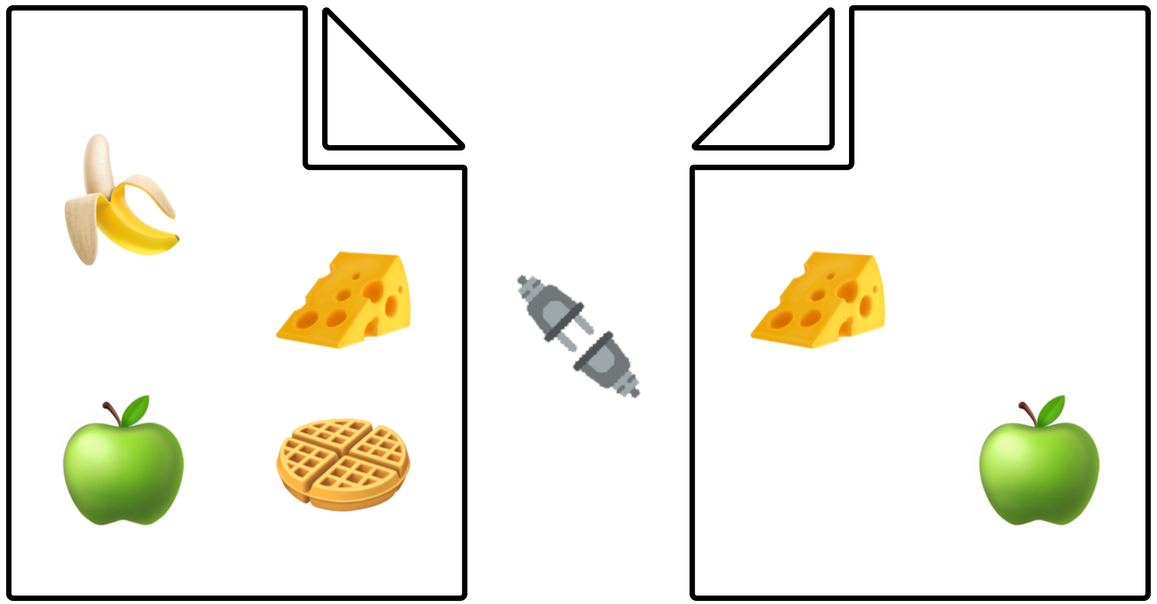
Stanley adds 🍏



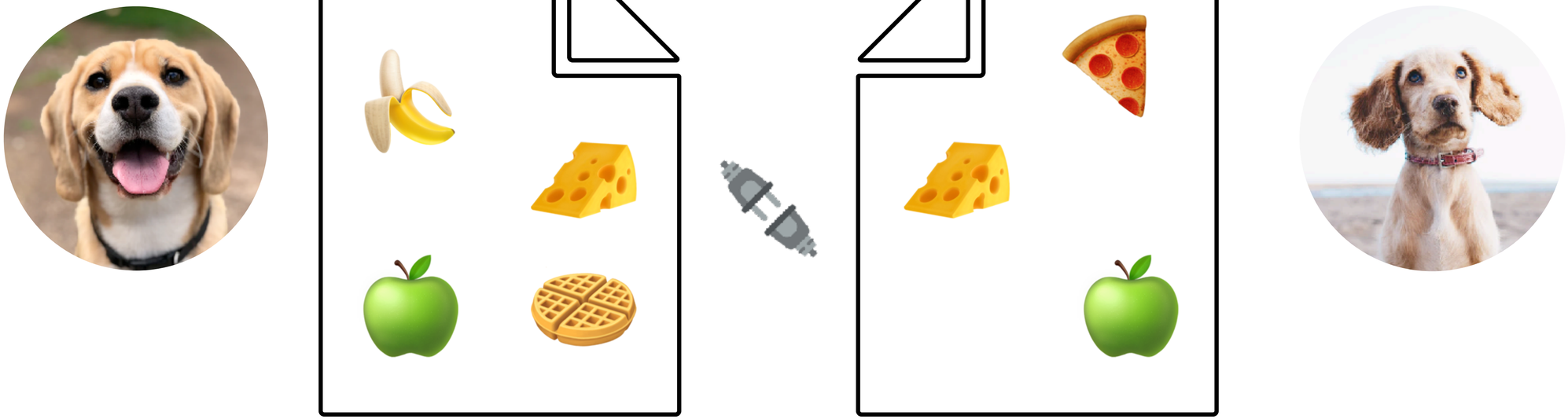
Stanley and Lily are disconnected
They continue working offline on their own



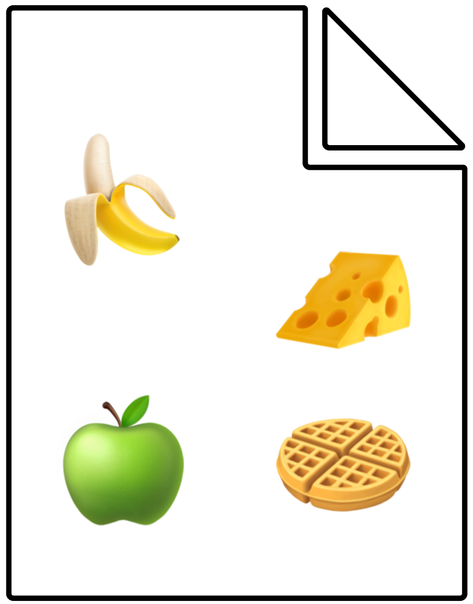
Stanley adds 🍌 and 🧇

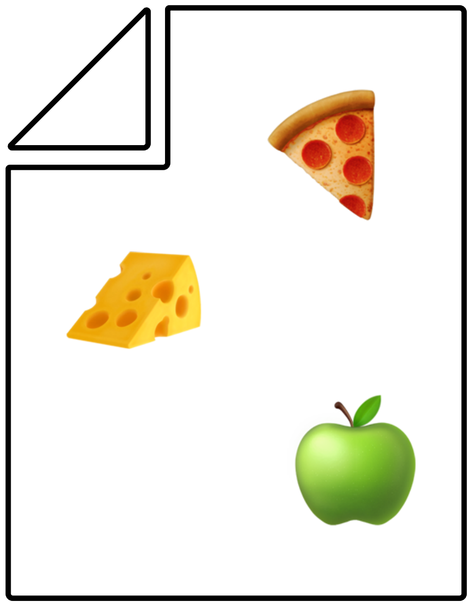


Lily adds 🍕

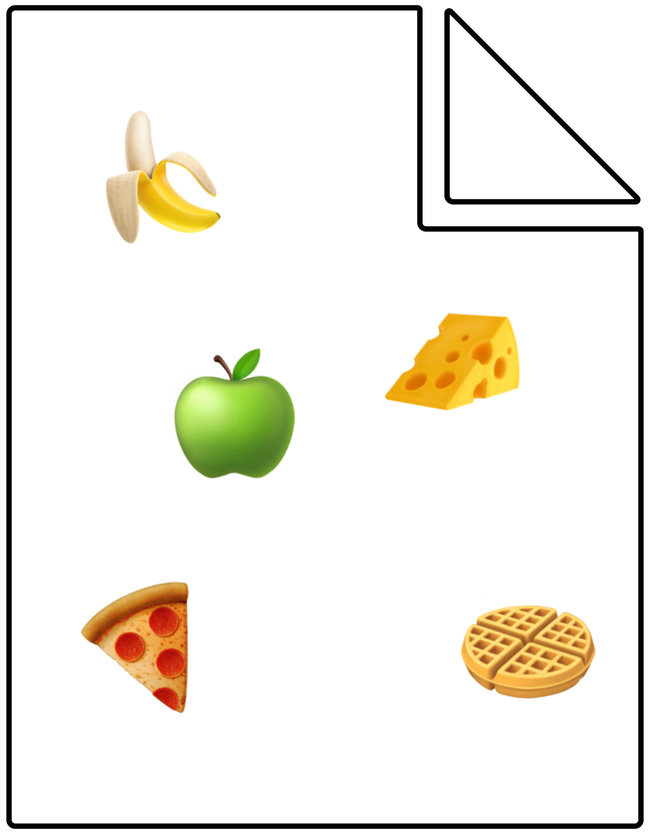


Their connection goes back on
What should the list look like ?





The G-Set is a very simple state-based CRDT, where
elements can only be added, and *merge* is defined as the union of the sets



- Because *merge* is **commutative**,
  Stanley and Lily can sync their local states in any order

- Because *merge* is **idempotent**, in the event of a network failure,
  Stanley and Lily can sync their local states multiple times for the same updates, and still end up with a consistent state

The G-Set

From a programmer's perspective

```haskell
-- | A simple grow-only set, where the merge function is just a union of the sets
newtype GSet t = GSet (Set.HashSet t)

instance stateBasedCRDTGSet :: (Hashable t) ⇒ StateBasedCRDT (GSet t) where
   merge (GSet a) (GSet b) = GSet $ a <> b

query :: forall t. GSet t → Set.HashSet t
query (GSet set) = set

insert :: forall t. Hashable t ⇒ t → GSet t → GSet t
insert value (GSet set) = GSet $ Set.insert value set
```

The G-Counter

Can you guess what it does?

The G-Counter is a vector of integers V with the size of the number of replicas

Each replica i is allowed to increment $\mathrm{V}[\mathrm{i}]$

The value of the counter is the sum of its components

Merging two counters takes the maximum value for each component

The G-Counter

From a programmer's perspective

```haskell
newtype GCounter
  = GCounter (Map.HashMap ReplicaId Int)

instance gCounterStateBasedCRDT :: StateBasedCRDT GCounter where
  merge (GCounter a) (GCounter b) = GCounter $ Map.unionWith max a b

query :: GCounter → Int
query (GCounter map) = sum map

increment :: ReplicaId → GCounter → GCounter
increment replica (GCounter map) = GCounter $ Map.insertWith add replica 1 map
```

Use case: hits / like buttons like the clap button on Medium
Page views counters

Demo time!

I told you there would be demos

Testing our implementation

It works, but is it really correct?

CRDTs have laws

State-based CRDTs

- Recall state-based CRDT definition
  - A binary *merge* operation that is
    - Associative
    - Commutative
    - Idempotent
  - A neutral element that also serves as the initial state for a new replica

Property-based testing verifies laws by ensuring they hold under a large number of randomly generated tests

- If I have a way of generating arbitrary members of S …
- A $\mathrm{merge(a,\ b)}$ function
- And a law, for instance commutativity, defined as
  $\forall x \in S, y \in S : merge(x, y) = merge(y, x)$
- Then I could generate a large number of inputs, and make sure the law holds of every one of them

Introducing QuickCheck

```haskell
-- In QuickCheck, the Arbitrary class associates a random value generator to a type
class Arbitrary t where
  arbitrary :: Gen t -- Gen t is a way of generating random values of type t


-- Let's define Arbitrary instances of our CRDTs
instance arbitraryGSet :: (Arbitrary t, Hashable t) ⇒ Arbitrary (GSet t) where
  arbitrary = GSet <<< Set.fromArray <$> arbitrary

instance gCounterArbitrary :: Arbitrary GCounter where
  arbitrary = GCounter <<< Map.fromArray <$> arrayOf kvGen
    where
    kvGen :: Gen (Tuple ReplicaId Int)
    kvGen = Tuple <$> arbitrary <*> suchThat arbitrary (\x → greaterThanOrEq x 0
```

Introducing QuickCheck

```
-- In QuickCheck, the Arbitrary class associates a random value generator to a type
class Arbitrary t where
  arbitrary :: Gen t -- Gen t is a way of generating random values of type t


-- Let's define Arbitrary instances of our CRDTs
instance arbitraryGSet :: (Arbitrary t, Hashable t) ⇒ Arbitrary (GSet t) where
  arbitrary = GSet <<< Set.fromArray <$> arbitrary

instance gCounterArbitrary :: Arbitrary GCounter where
  arbitrary = GCounter <<< Map.fromArray <$> arrayOf kvGen
    where
    kvGen :: Gen (Tuple ReplicaId Int)
    kvGen = Tuple <$> arbitrary <*> suchThat arbitrary (\x → greaterThanOrEq x 0
```

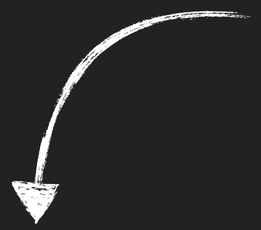
```haskell
stateBasedCRDTLaws ::
  forall t. StateBasedCRDT t ⇒ Arbitrary t ⇒ String → Eq t ⇒ Show t ⇒ Proxy t → Spec Unit
stateBasedCRDTLaws name _ =
  describe name do
    it "should be associative"
      $ quickCheck associativity
    it "should be commutative" do
      quickCheck commutativity
    it "should be idempotent" do
      quickCheck idempotence
    it "should have a neutral element" do
      quickCheck identity'
  where
  associativity a b c = (merge (merge a b) c) ≡≡ (merge a (merge b c))

  commutativity a b = merge a b ≡≡ merge b a

  idempotence a = merge a a ≡≡ a

  identity' a = (merge a mempty) ≡≡ a
```

Since the laws are the same for every state-based CRDT, I can reuse my tests

```
spec =
  describe "State-based CRDTs" do
    stateBasedCRDTLaws "GSet of integers" (Proxy :: Proxy (GSet Int))
    stateBasedCRDTLaws "GSet of strings" (Proxy :: Proxy (GSet String))
    stateBasedCRDTLaws "GCounter" (Proxy :: Proxy GCounter)
    stateBasedCRDTLaws "2PSet of integers" (Proxy :: Proxy (TwoPhaseSet Int))
    stateBasedCRDTLaws "2PSet of strings" (Proxy :: Proxy (TwoPhaseSet String))
    -- and so on
```

```
❯ spago test
GSet - Grow-only set » Int laws
  √ should be associative
  √ should be commutative
  √ should be idempotent
  √ should have a neutral element
GSet - Grow-only set » String laws
  √ should be associative
  √ should be commutative
  √ should be idempotent
  √ should have a neutral element
2PSet - Two-phase Set » Int laws
  √ should be associative
  √ should be commutative
  √ should be idempotent
  √ should have a neutral element
2PSet - Two-phase Set » String laws
  √ should be associative
  √ should be commutative
  √ should be idempotent
  √ should have a neutral element
GCounter - Grow-only counter » laws
  √ should be associative
  √ should be commutative
  √ should be idempotent
  √ should have a neutral element
OPCounter - Operation-based counter » laws
  √ should be commutative
  √ should have complementary generateOperations / applyOperation functions
```

A single test run generates hundreds of test cases, giving us a reasonable level of confidence that our implementation is correct



Operation-based CRDTs can be tested the same way

Revisiting the `StateBasedCRDT` class

- A state-based CRDT is an *algebraic structure*: a set + lawful operations on that set
- Different algebraic structures have different laws
  - A set + an associative binary operation is a **semigroup**
  - A semigroup + a neutral element, such that $\forall x : merge(e, x) = e$, is a **monoid**
  - A semigroup that is also commutative and idempotent is a **semilattice**
  - A monoid, that is also commutative and idempotent is a **bounded semilattice**
- So a state-based CRDT is a commutative and idempotent monoid, or bounded semilattice

A set + an associative binary operation is a **semigroup**

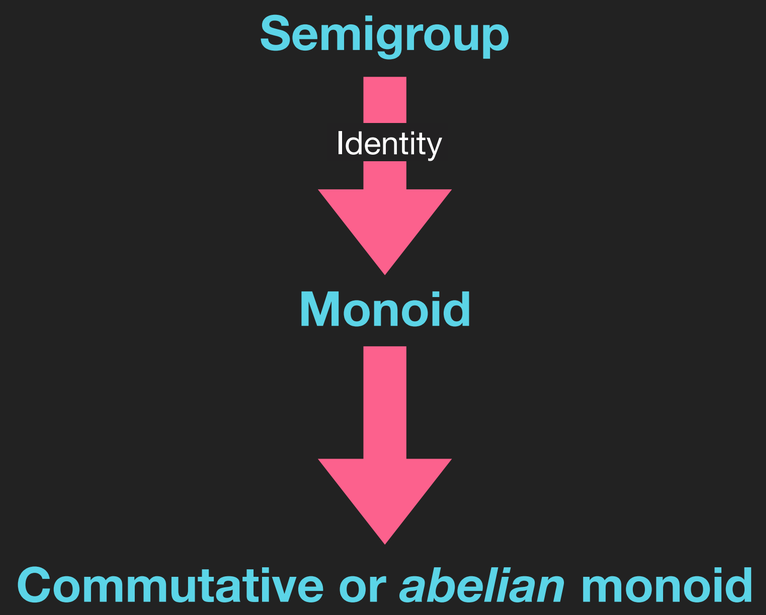
```
class Semigroup a where
  append :: a → a → a
```

A semigroup + an identity element is **monoid**

```haskell
class Semigroup a where
  append :: a → a → a

class Semigroup m ⩽ Monoid m where
  mempty :: m
```

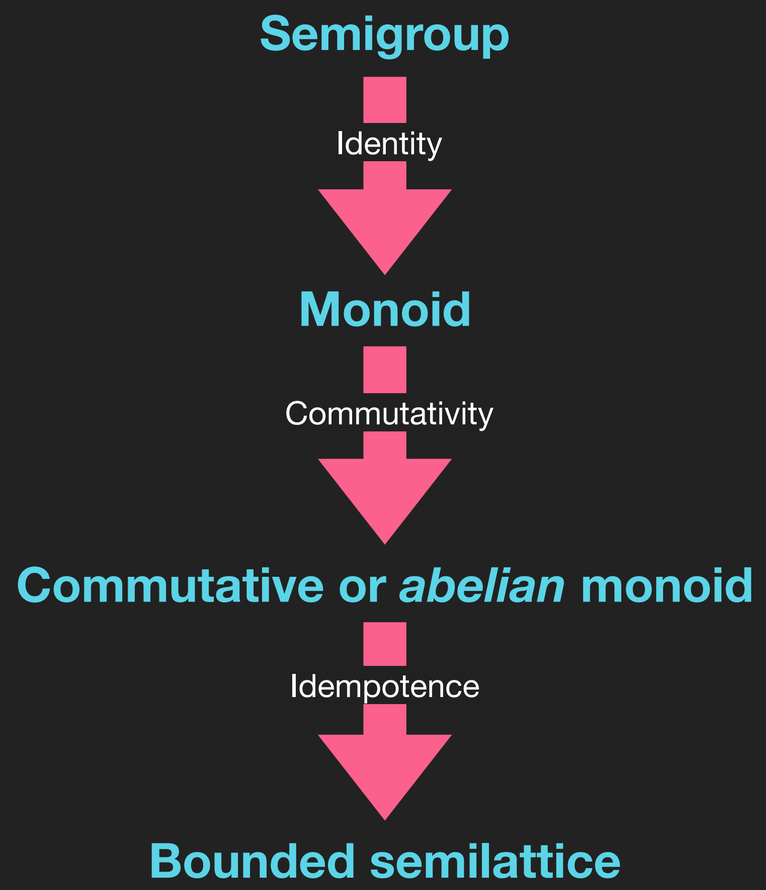
Additionally, a monoid can be commutative

```haskell
class Semigroup a where
  append :: a → a → a

class Semigroup m ⩽ Monoid m where
  mempty :: m

class Monoid m ⩽ CommutativeMonoid m
```



A monoid, which is also commutative and idempotent is a **bounded semilattice**

```
class Semigroup a where
  append :: a → a → a

class Semigroup m ⩽ Monoid m where
  mempty :: m

class Monoid m ⩽ CommutativeMonoid m

class CommutativeMonoid m
  ⩽ BoundedSemilattice m
```



I’m a programmer, why do I care ?

- Semigroups and monoid are common in Haskell & PureScript
- They also show up in Haskell-inspired libraries like *Cats* for Scala, *fp-ts* for TypeScript, *Arrow* for Kotlin …
- A state-based CRDT is a special monoid with more laws
- Knowing this lets us refactor our code to simplify it, and take advantage of existing libraries

```haskell
-- StateBasedCRDT is a bounded semilattice, i.e. a monoid with additional laws
-- Laws don't appear here, but they will in tests
class Monoid t ⩽ StateBasedCRDT t

-- In that case, merge is a synonym for append,
-- but the compiler will only let us use it for CRDTs, not just any semigroup
merge :: forall t. StateBasedCRDT t ⇒ t → t → t
merge = append
```

```haskell
-- Merging multiple states is a simple as calling 'fold'
mergeAll :: forall t. StateBasedCRDT t ⇒ Array t → t
mergeAll = fold

-- fold :: forall f t. Foldable f ⇒ Monoid t → f t → t
-- Other utility functions are ready to use, such as foldMap
```

Compound CRDTs

Unity makes strength

"A common strategy in CRDT development is to combine multiple CRDTs to make a more complex CRDT"

“Conflict-free replicated data type” from Wikipedia, the free encyclopedia (accessed 11 April 2021)

By combining to G-Counters,
we create a counter that supports increments and decrements.

We call it a **PN-Counter**

PN-Counter = A G-Counter for increments + A G-Counter for decrements
The value of the counter is the difference of the two G-Counters

```haskell
-- | A counter that supports increments and decrements
data PNCounter = PNCounter GCounter.GCounter GCounter.GCounter

query :: PNCounter → Int
query (PNCounter a b) = GCounter.query a - GCounter.query b

increment :: ReplicaId → PNCounter → PNCounter
increment rid (PNCounter a b) = PNCounter (GCounter.increment rid a) b

decrement :: ReplicaId → PNCounter → PNCounter
decrement rid (PNCounter a b) = PNCounter a (GCounter.increment rid b)
```

Similarly, we can create a set that supports removals
by combining two G-Sets

We call it a **Two-phase-Set**

2P-Set = A G-Set for adds + a G-Set for removals,
which we call a *tombstone set*

The members of the 2P-Set are elements of the first set that
don’t appear in the tombstone set

Use case: count active users on a P2P network

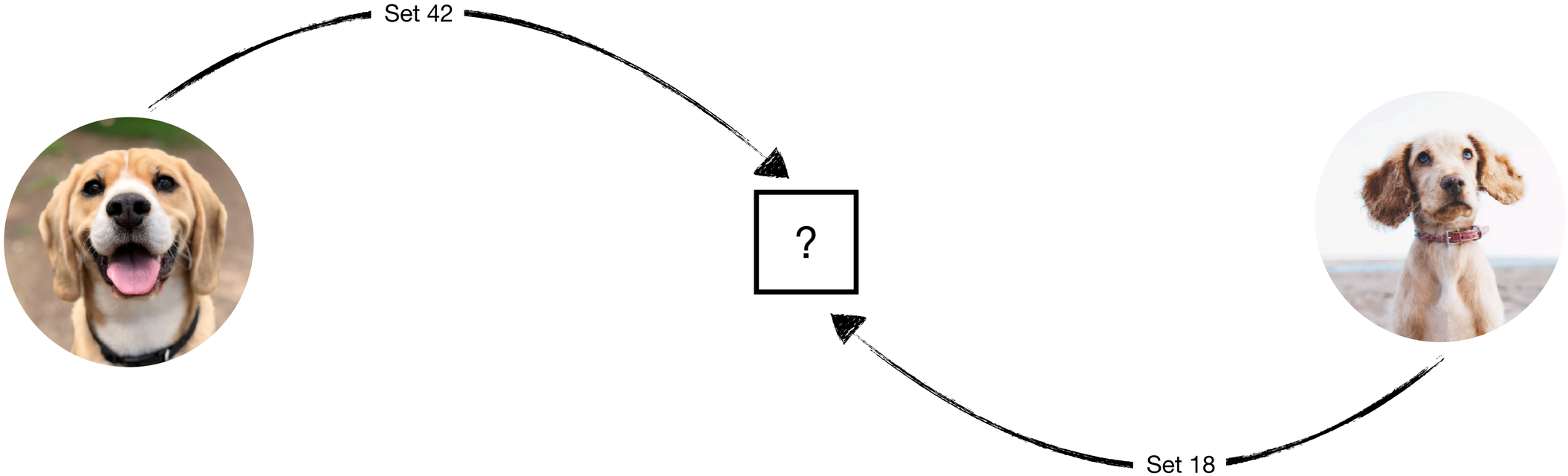
```haskell
-- | A set that supports adding elements, or removing it definitively
-- | Removed elements are kept around in a special *tombstone* set.
data TwoPhaseSet t
  = TwoPhaseSet (GSet.GSet t) (GSet.GSet t)

insert :: forall t. Hashable t ⇒ t → TwoPhaseSet t → TwoPhaseSet t
insert value (TwoPhaseSet a b) = TwoPhaseSet (GSet.insert value a) b

remove :: forall t. Hashable t ⇒ t → TwoPhaseSet t → TwoPhaseSet t
remove value (TwoPhaseSet a b) = TwoPhaseSet a $ GSet.insert value b

query :: forall t. Hashable t ⇒ TwoPhaseSet t → Set.HashSet t
query (TwoPhaseSet a b) = Set.difference (GSet.query a) (GSet.query b)
```

Demo time!

Time-based CRDTs

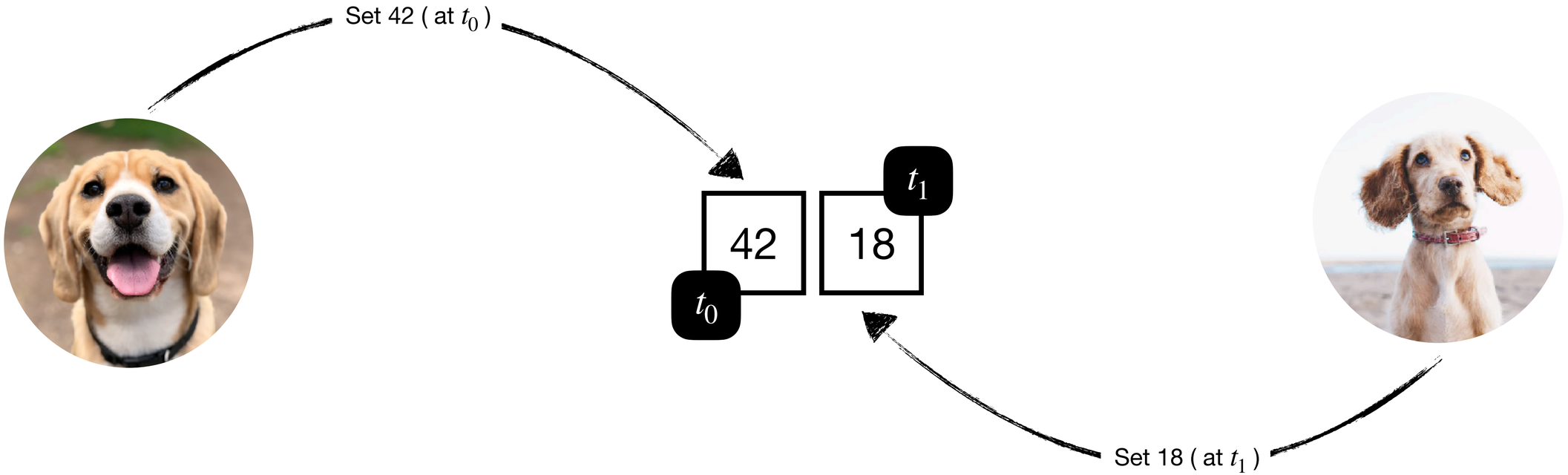
Where things get real messy, real quick

A register is a structure that can hold a single value,
and supports *get* and *set* operations



How can we reconcile conflicting replicas?

Let's create a *total order* of operations, by associating every *set* with a timestamp



The *merge* operation always picks the value with the greatest timestamp

We call this structure a *Last Writer Wins Register*

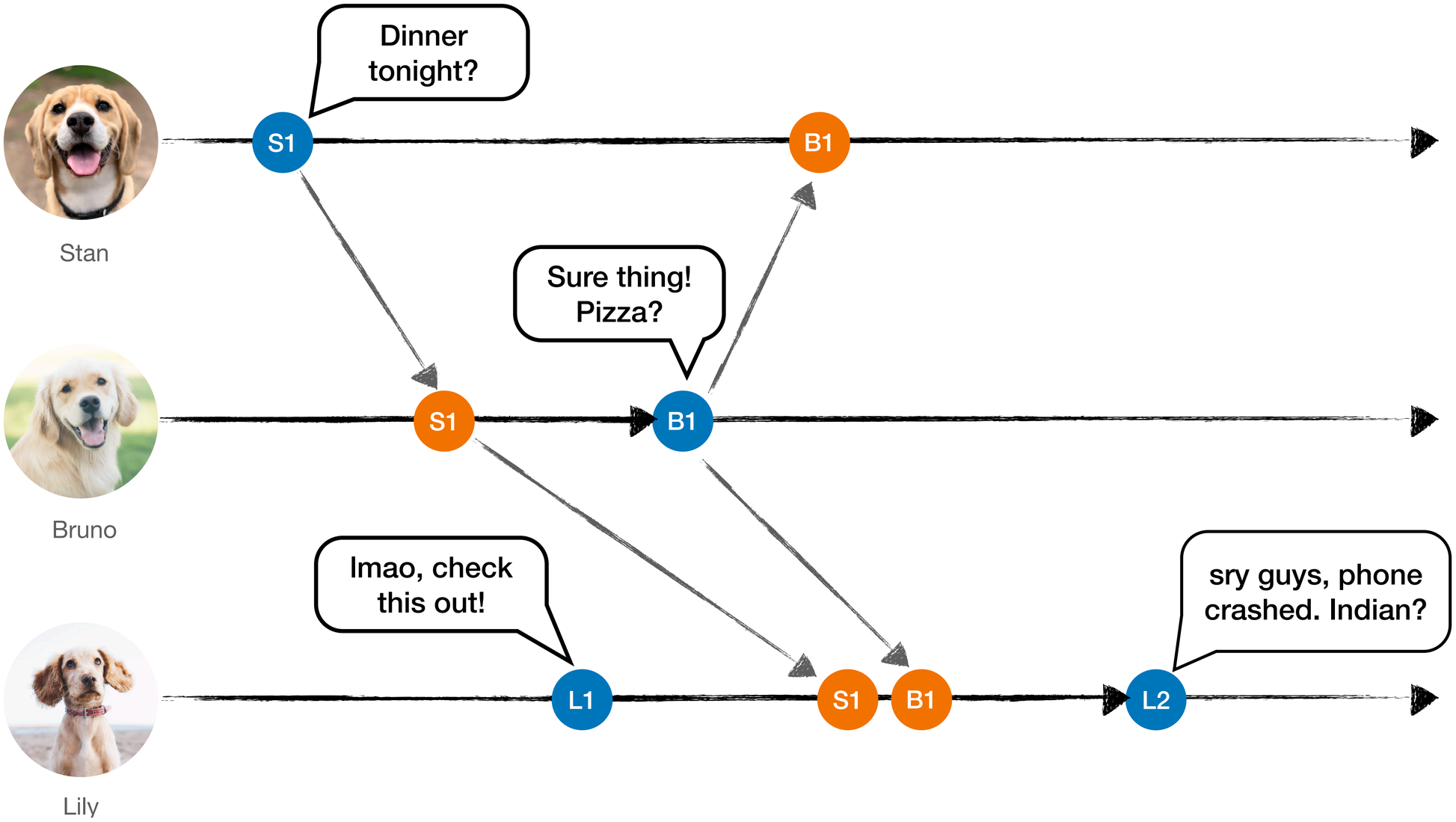
Several issues with LWW-Registers

Poor Stanley probably won’t be too happy to see his changes overridden,
even if it's by more recent changes

Several issues with LWW-Registers

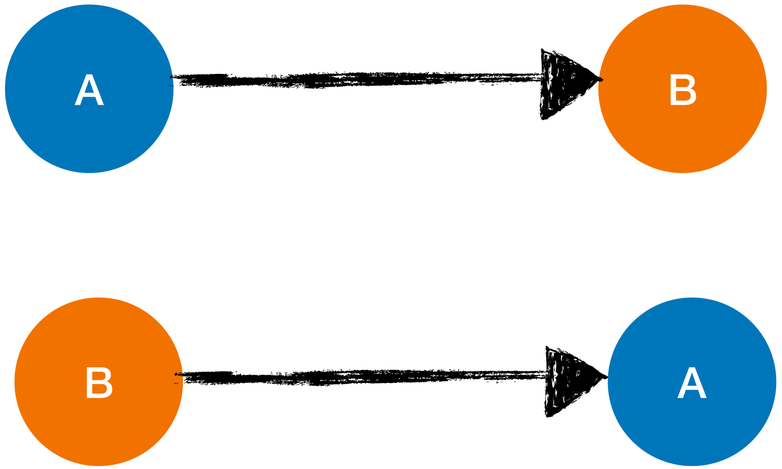
Defining a total order of events in a distributed system isn’t trivial

We know S1 hasn’t caused L1, even if it happened physically before L1,
simply because it hadn’t been received by Lily when she sent L1

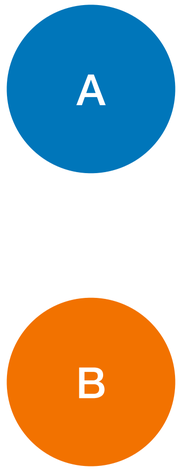


What if we didn’t need physical time? 

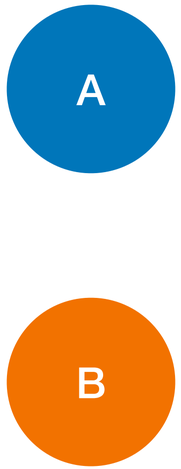
Suppose two events *A* and *B*



We want to know if one was possibly caused by the other

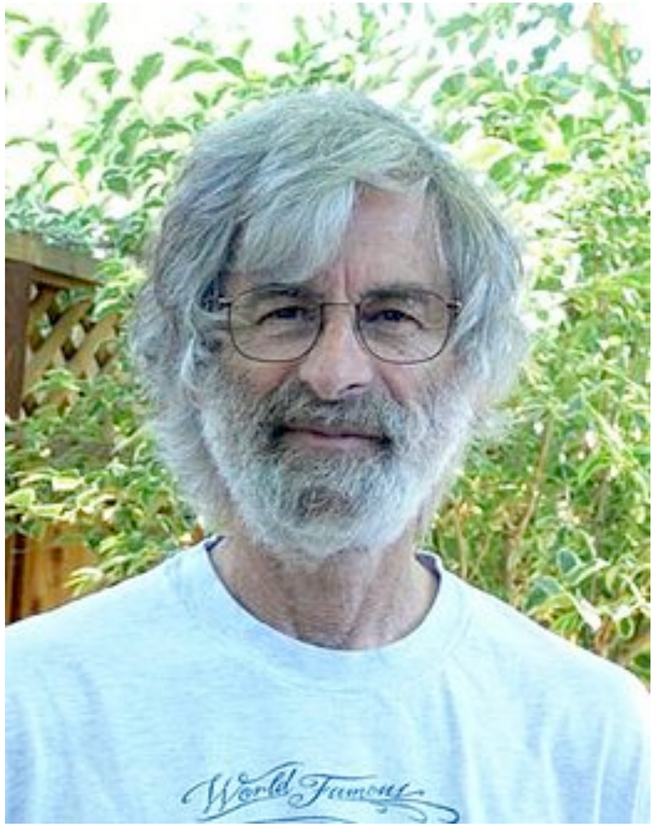


Or if they are concurrent



Regardless of their physical time, which doesn’t imply causation
(and which we don’t reliably know anyway)

Logical clocks



Leslie Lamport, inventor of the *happened-before* relation

- Logical clocks define a partial order of events in a distributed system
- They capture the causation (or lack thereof) between two events
- They don’t capture physical time
- They exist in many forms (go check em’ out!)
  - Lamport timestamps
  - Vector clocks
  - Hybrid logical clocks
  - …

Alright,

we have partial ordering, now what ?

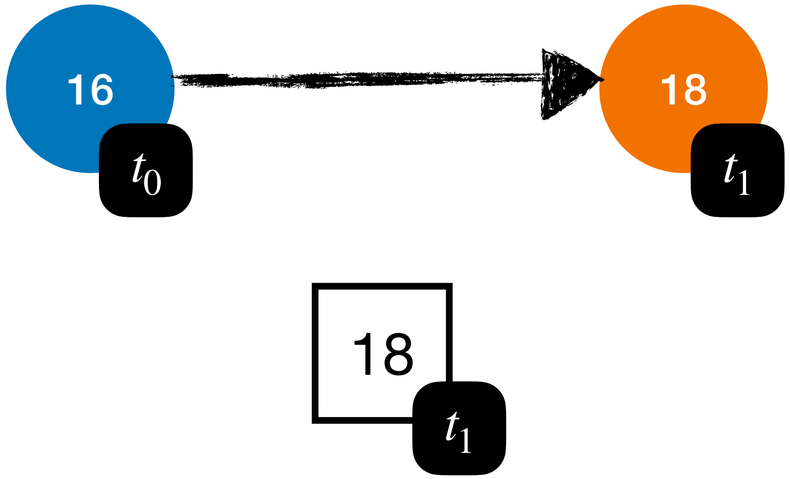
Let’s say you have read everything about logical clocks, and implemented one for your system.

You have two routes to go from there:

- Add an arbitrary component to form a total order of events, then implement LWW
- Embrace the partially-ordered nature of distributed system, and implement *multi-value registers*

A multi-value register
contains either one or many values

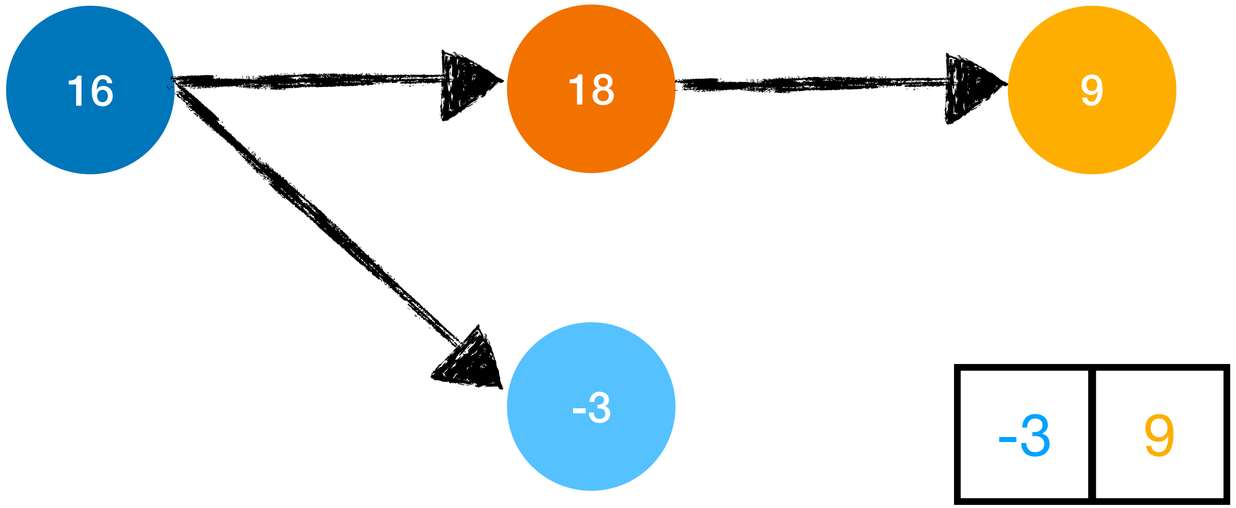
When one state precedes the other*, the CRDT behaves like a LWW-Register



*with regard to the *happened-before* relation,
symbolised by →

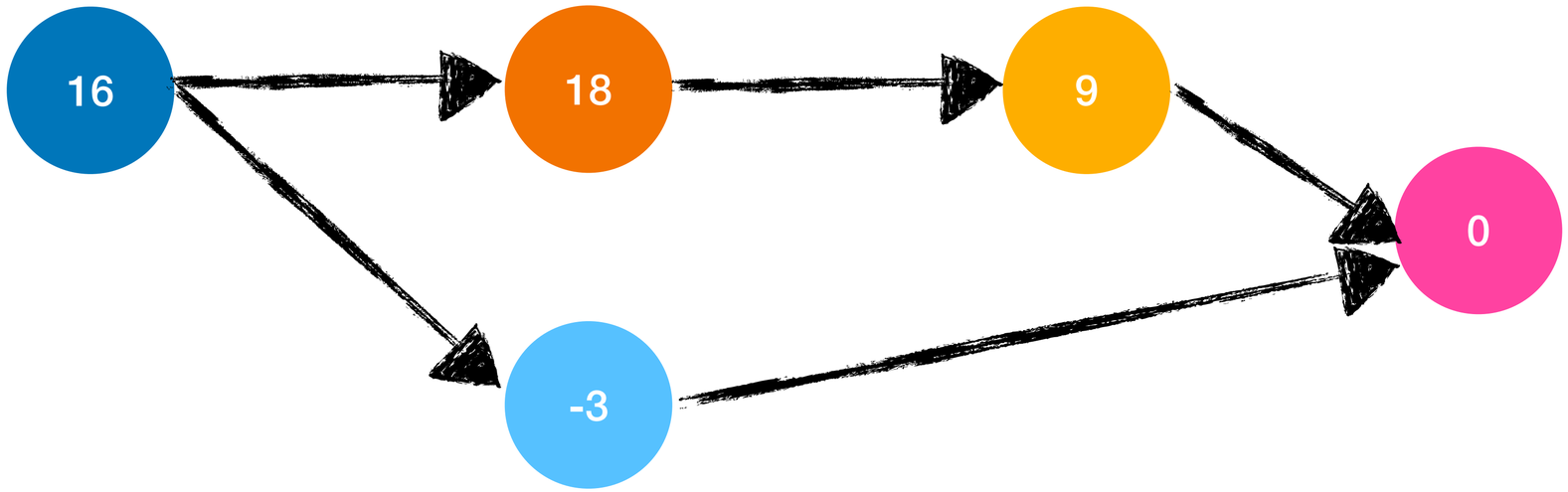
When two states are independent, they are both kept around

This could happen if one participant was offline at some point



-3	9

Eventually, someone will overwrite the values with a single one
The logical clock tells us wether this is intended or not



Let’s recap

This is getting too long

- Replicating data in a distributed system is hard
- Different replications strategies have different pros and cons
  - Strong consistency: consistent state achieved with complex consensus protocols, at the expense of write performance
  - Eventual consistency: states can diverge for the benefit of write performance and offline availability
  - Strong eventual consistency (SEC) : eventual consistency + additional safety guarantees
- Conflict-free replicated data types are one way to achieve SEC

- CRDTs exist in state-based and event-based variants
- Known CRDTs include counters, sets, maps and sequences
- CRDTs can be combined to form more complex structures, e.g. a map of CRDT to represent a complex object
- CRDTs work because of their obedience to algebraic laws: associativity, commutativity, identity and idempotence

- Physical time is not a substitue for causal ordering
- Logical clocks allow us to capture the causality (or lack thereof) between to events
- If you’re not sure which state to keep, try to keep both

Thank you 🙏

crdt.guillaumebogard.dev

for references, demos and slides