STATE

AND THE DISTRIBUTED DATA STRUCTURES

ARKADIUSZ GIL

@arkgil
@_arkgil

Erlang

STATE

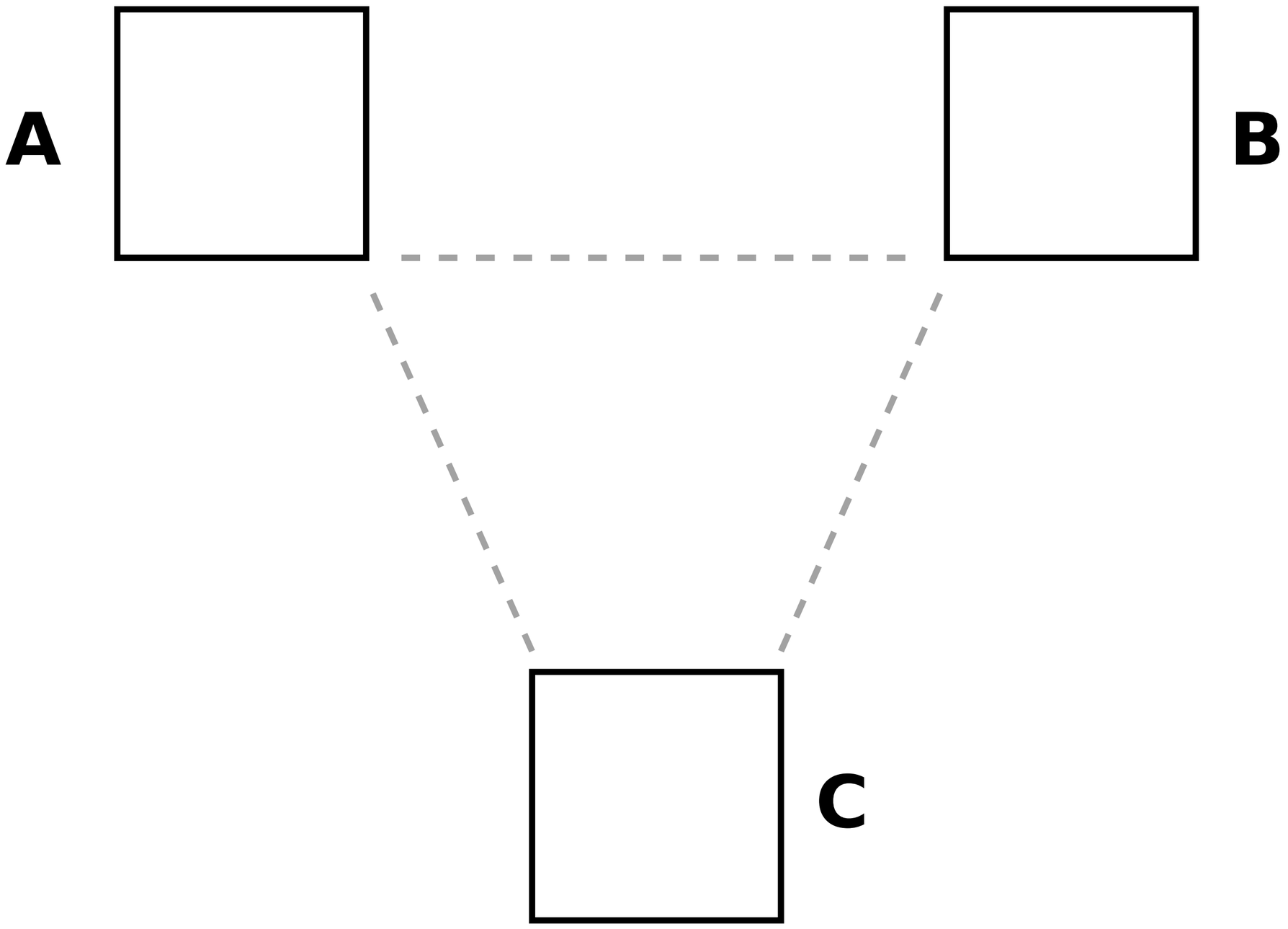
DISTRIBUTED STATE

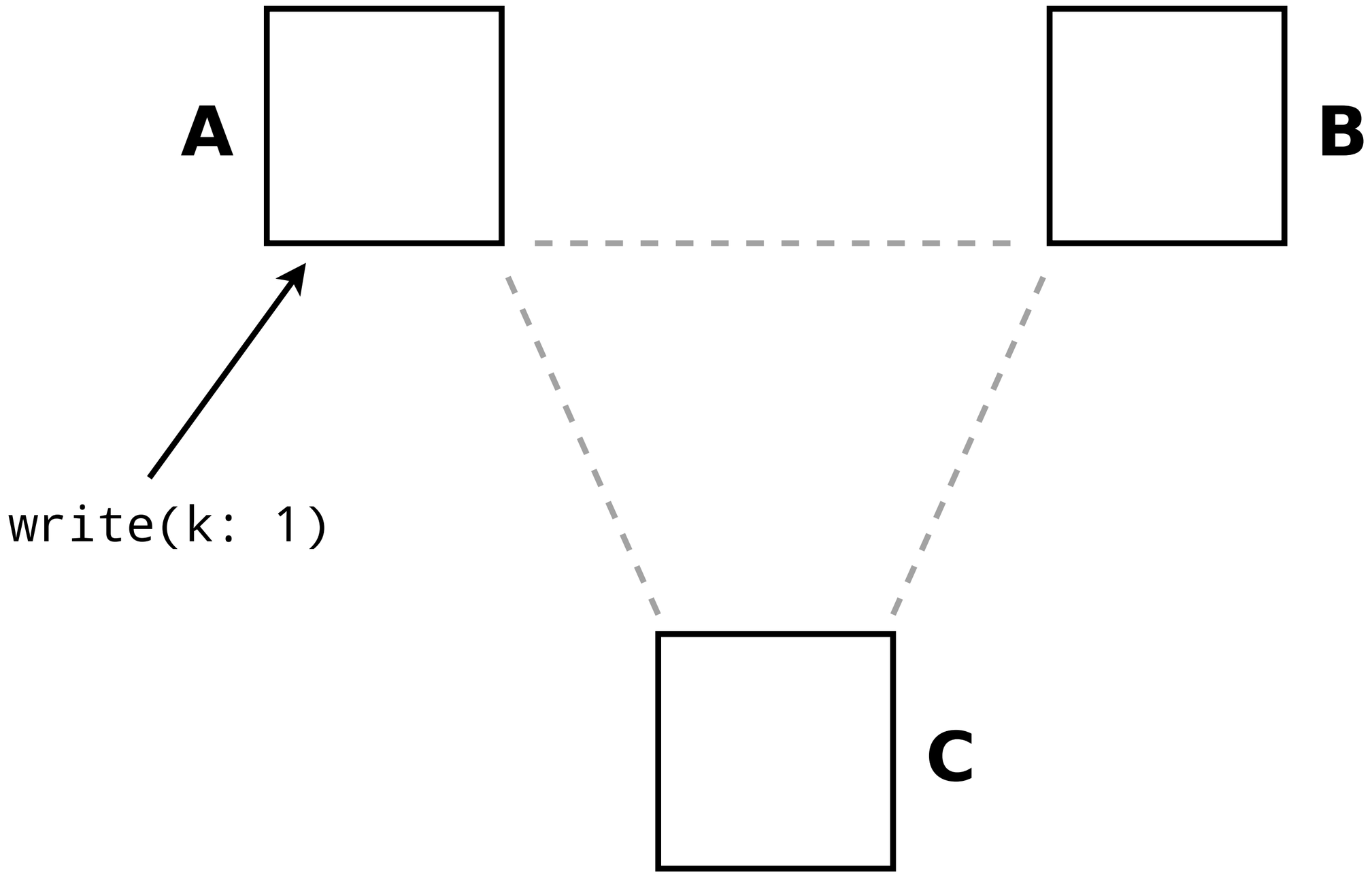
MNESIA

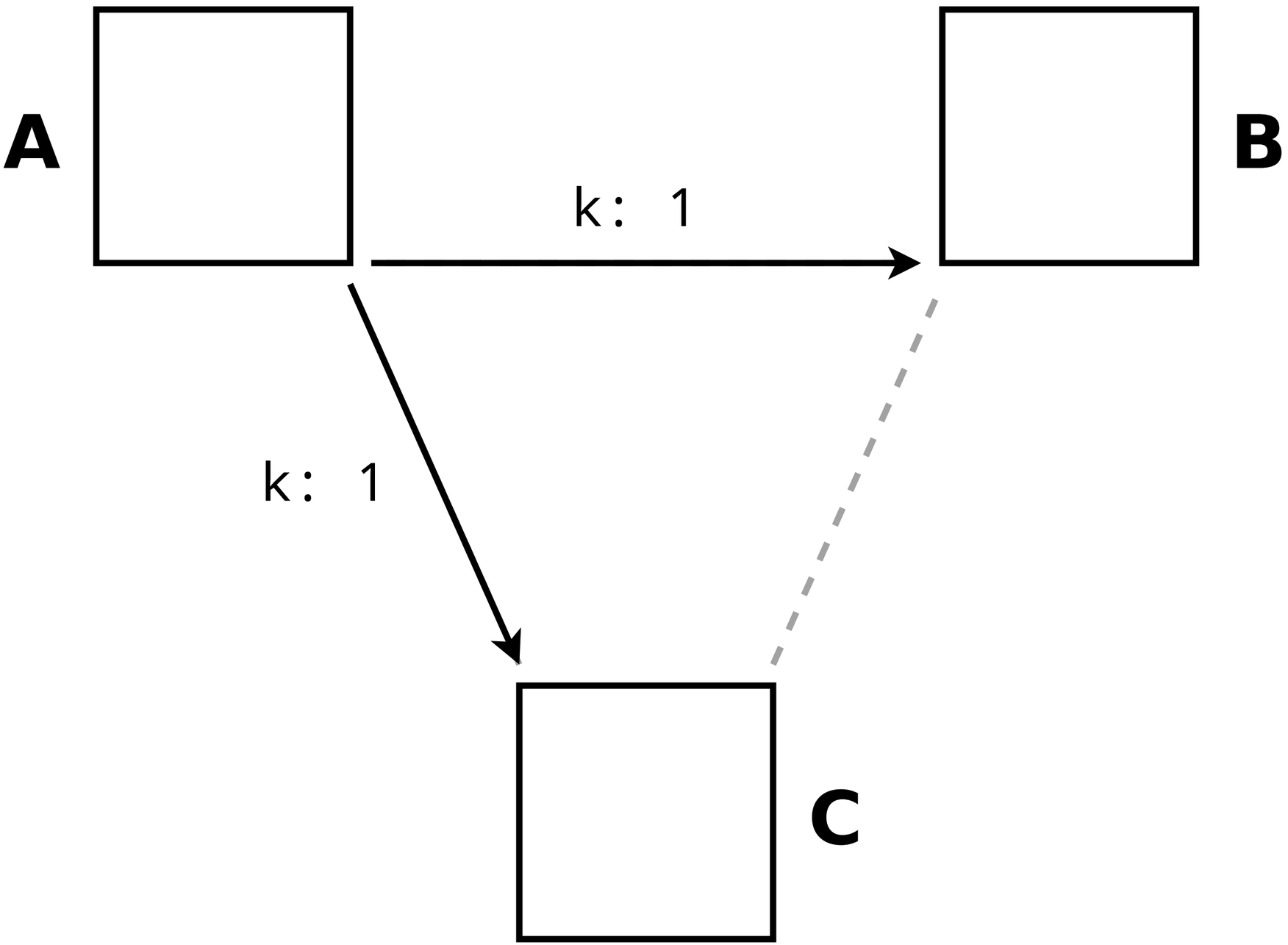
REPLICATION

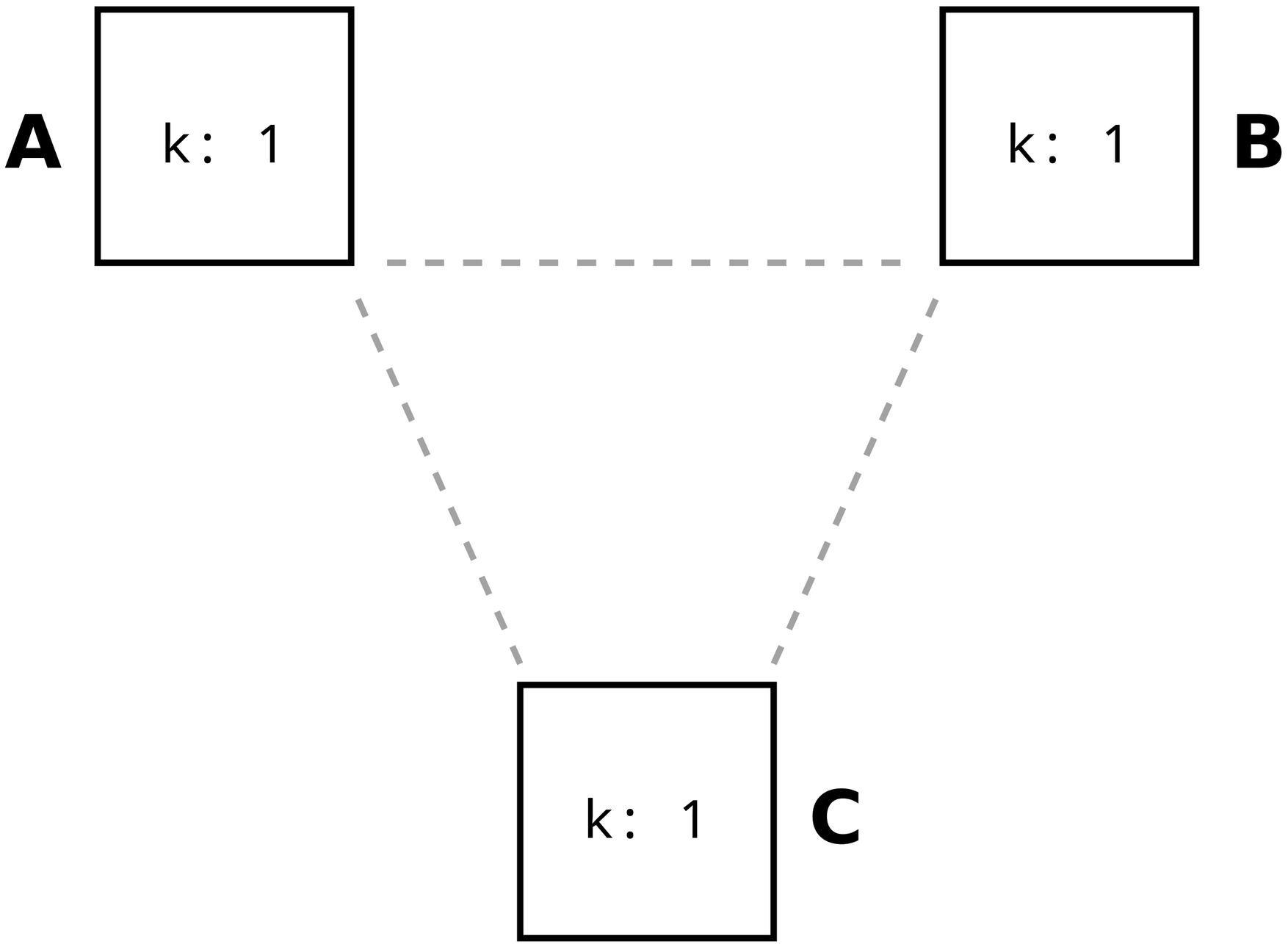
TRANSACTIONS

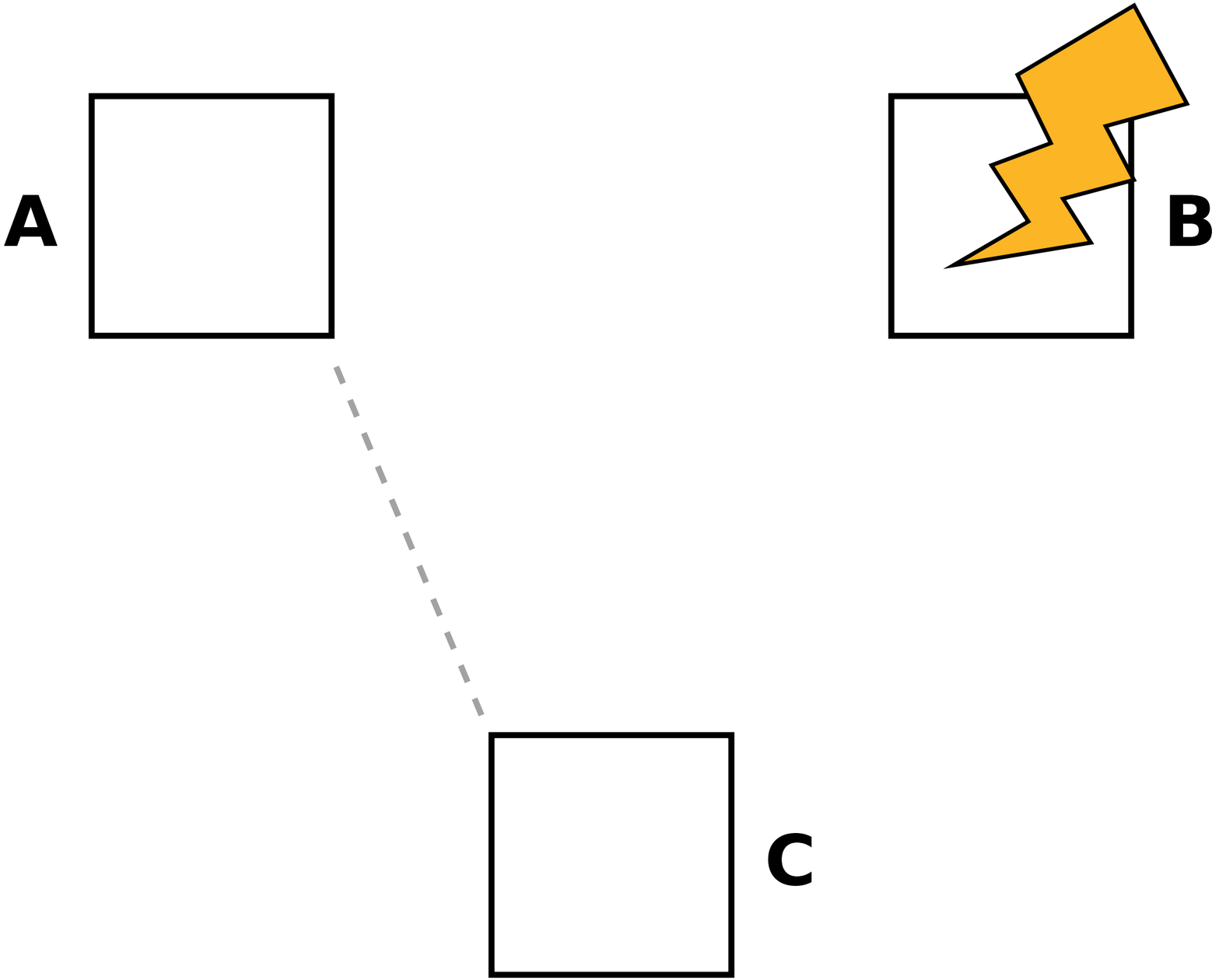
Guarantees

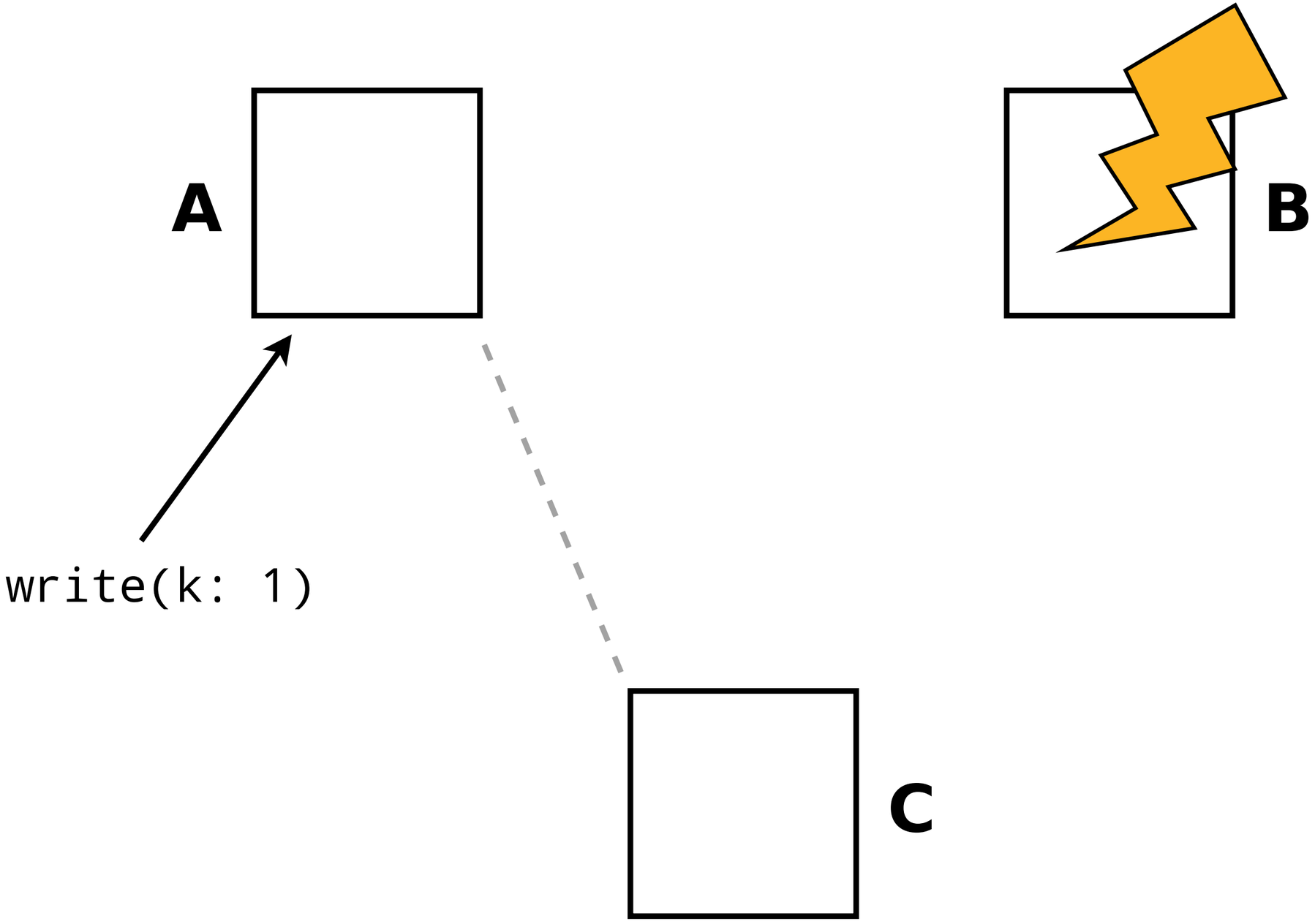

A
B
C

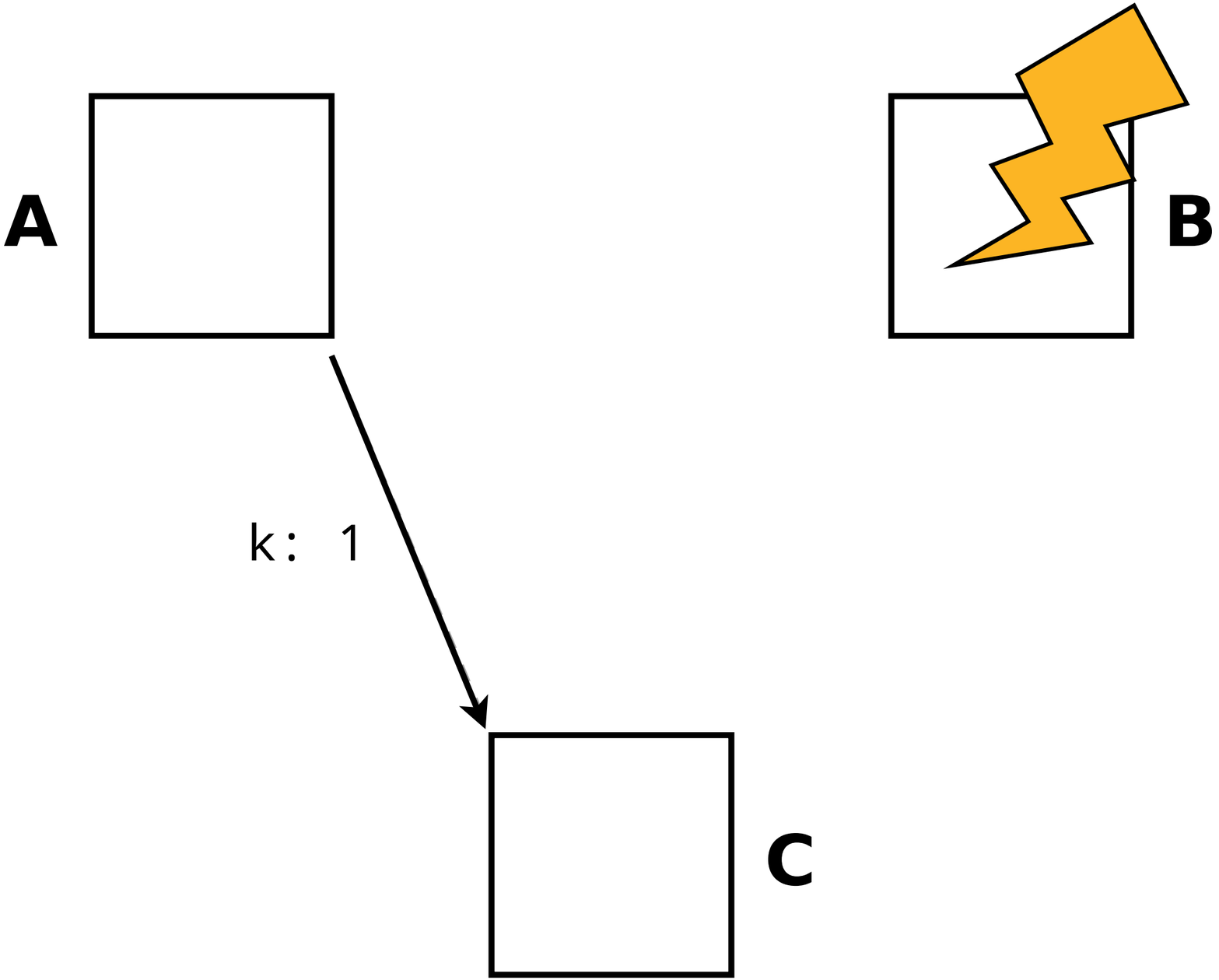

A
B
C
write(k: 1)

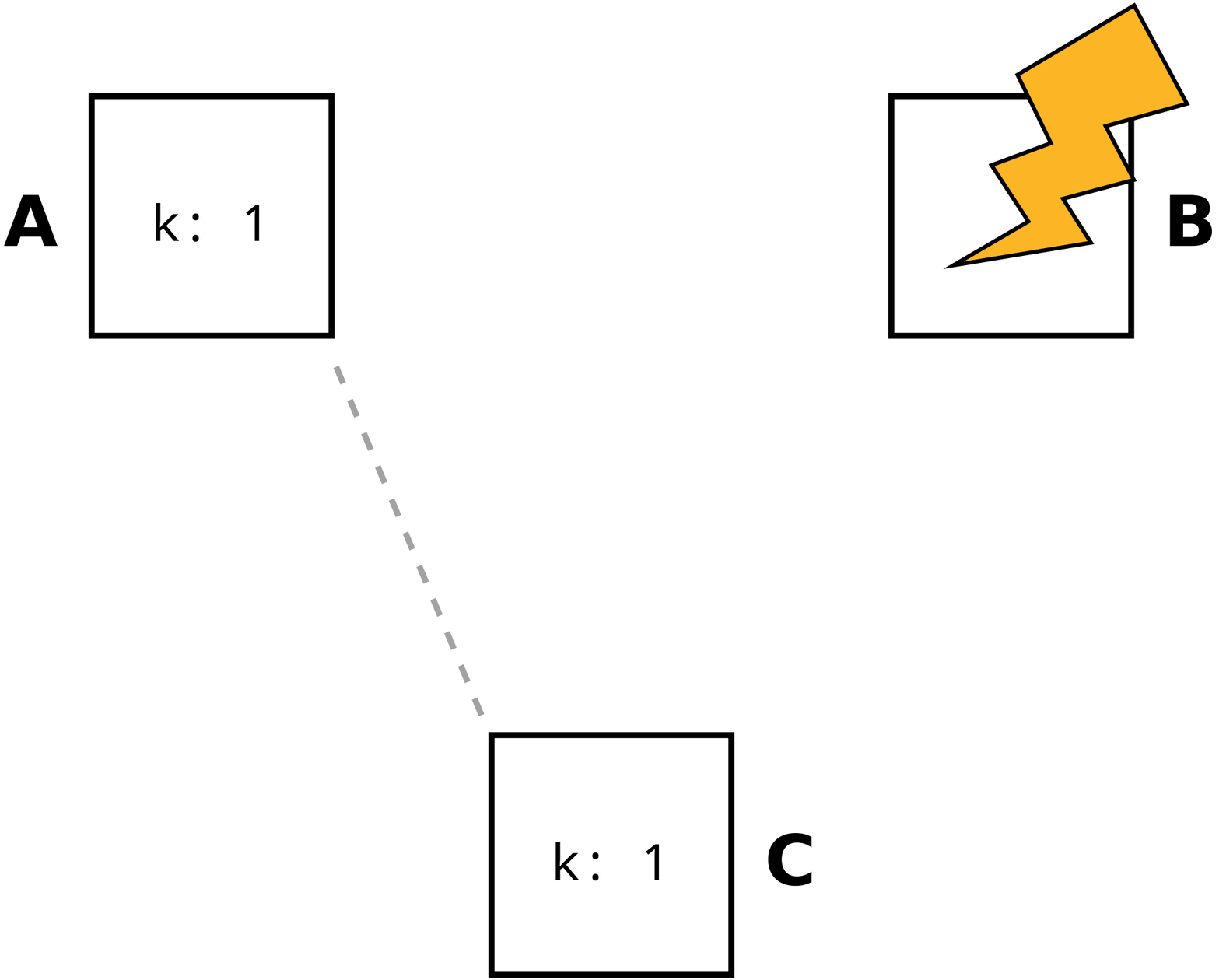

A
B
C
k: 1
k: 1

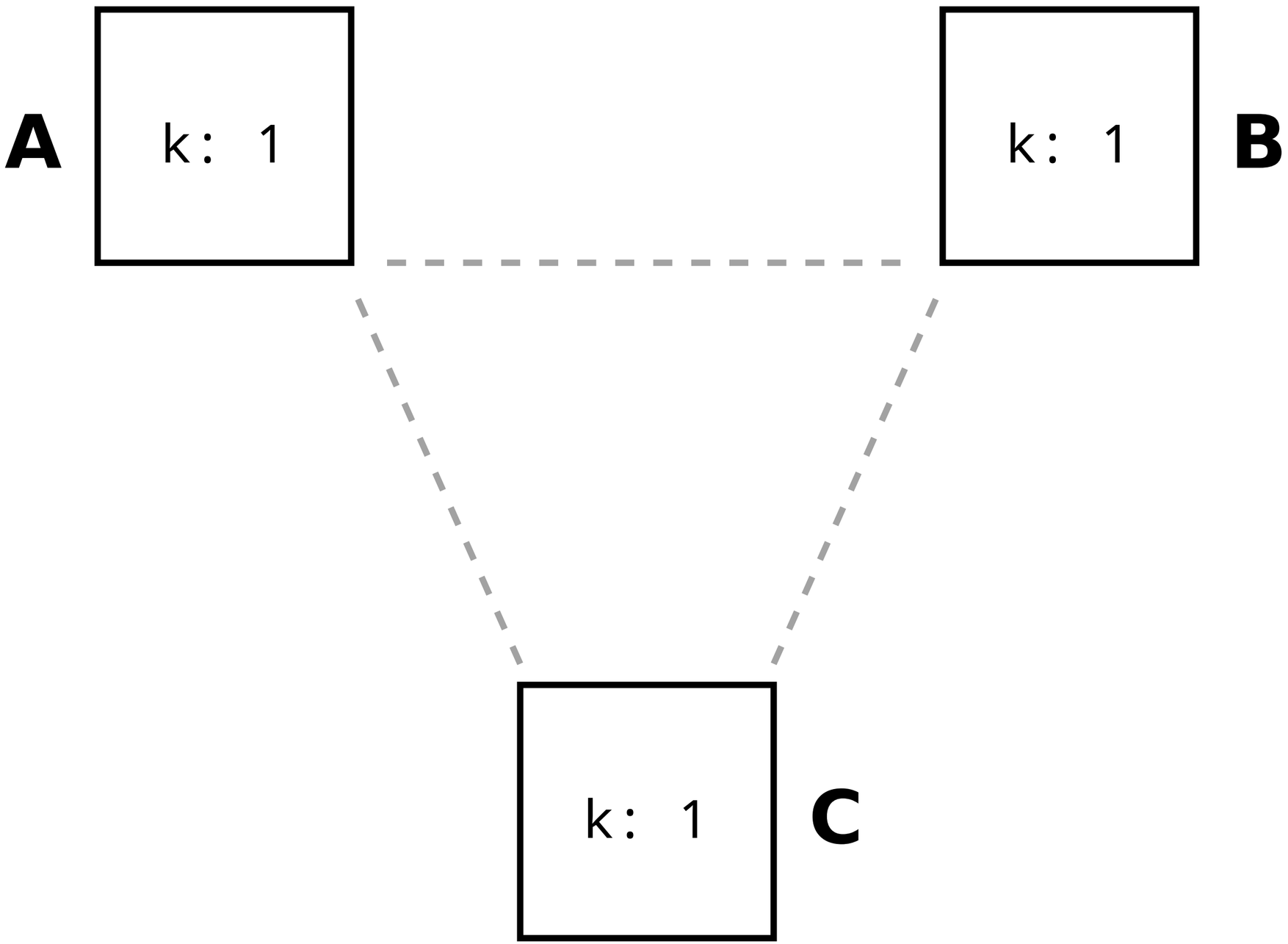

A
k: 1
k: 1
B
k: 1
C

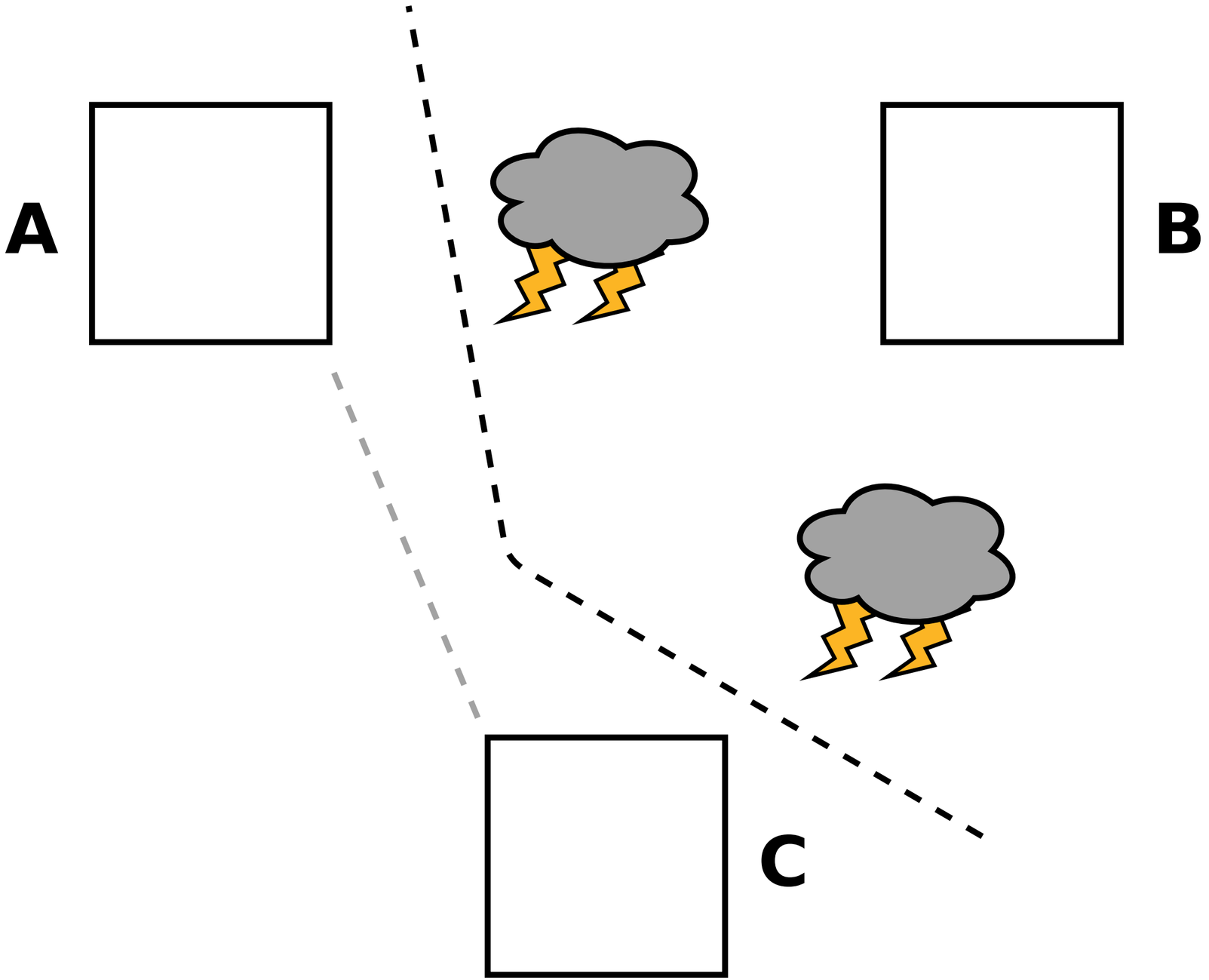

A
B
C

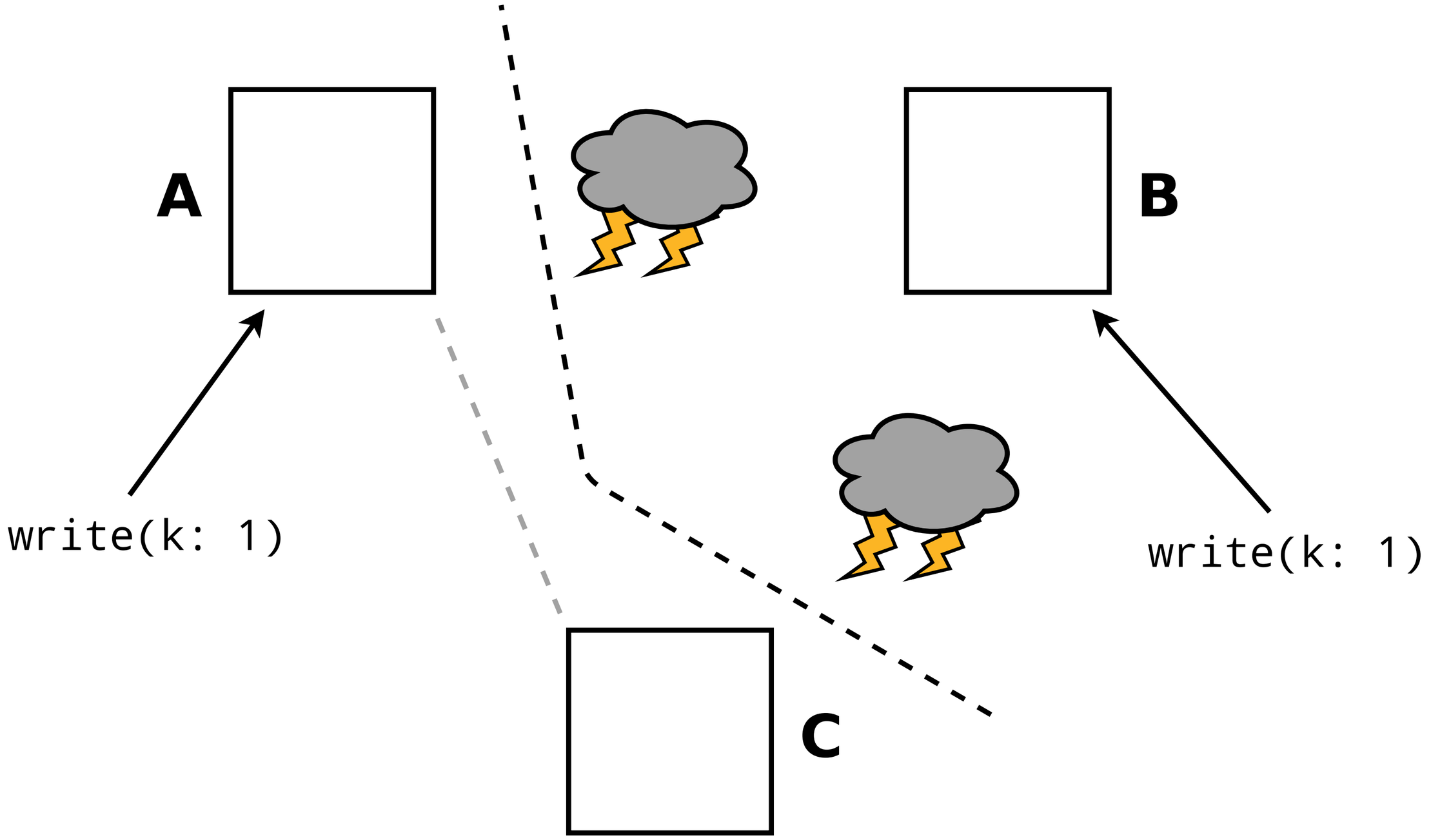

A
B
C
write(k: 1)

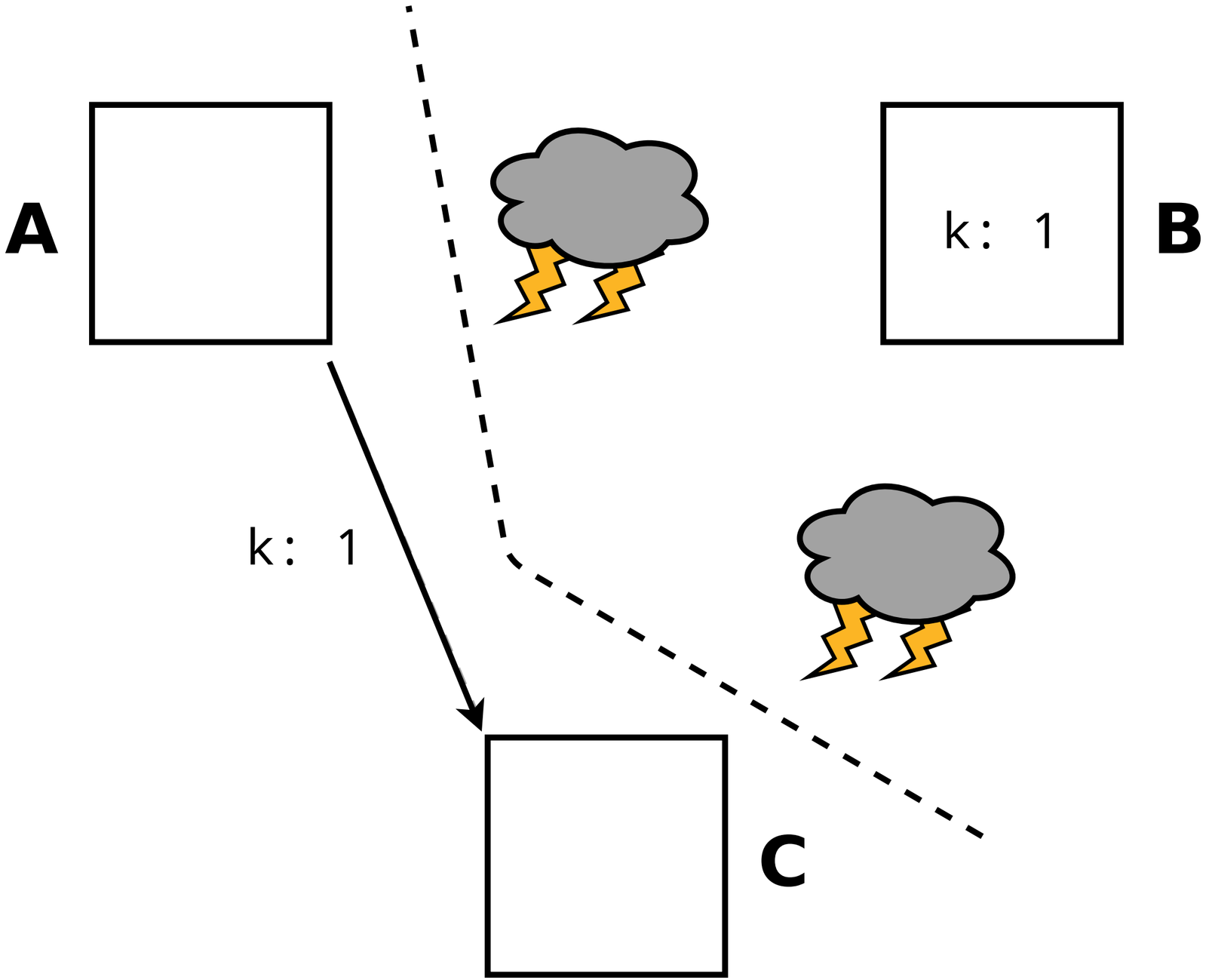

A
B
k: 1
C

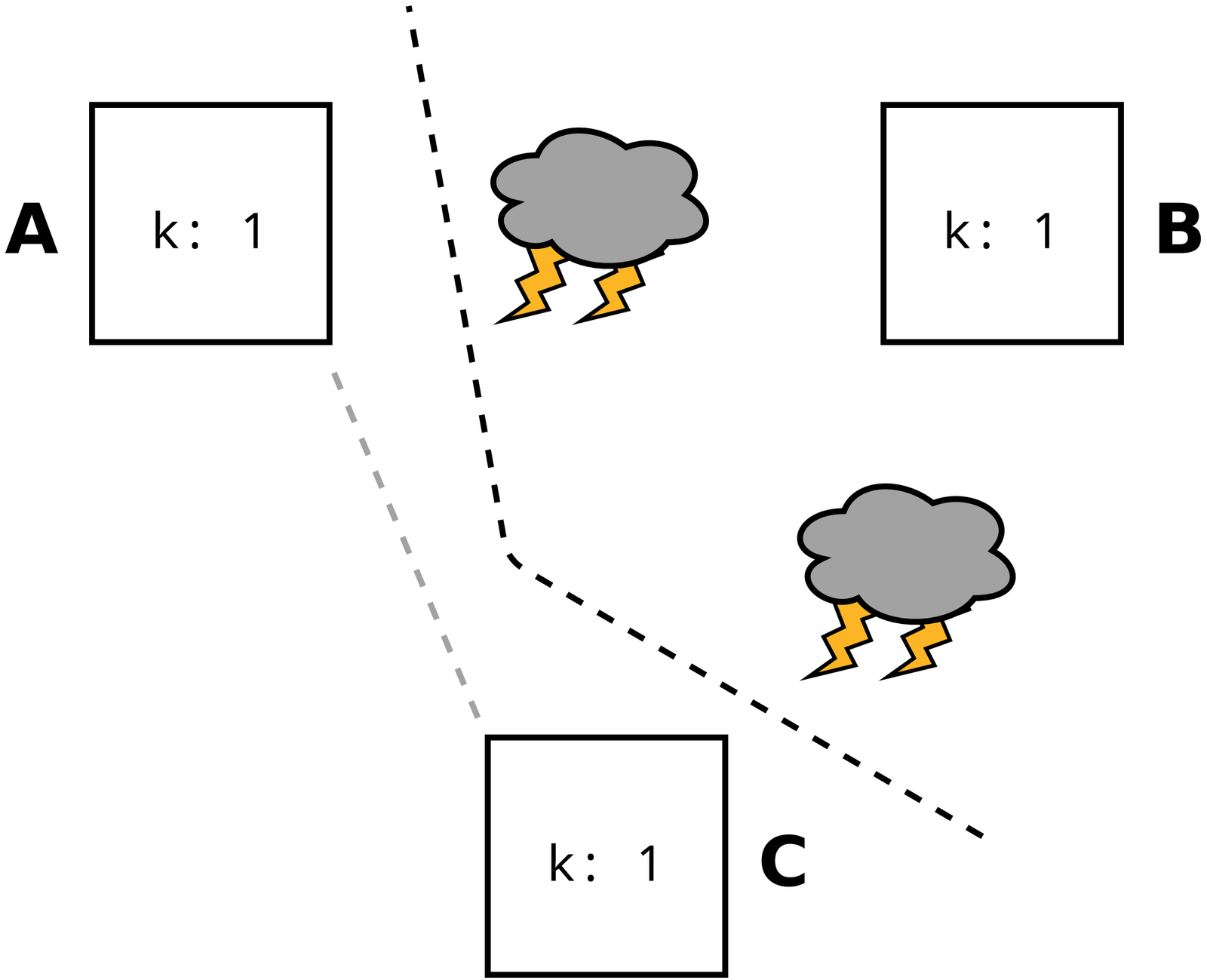

A
k: 1
B
k: 1
C

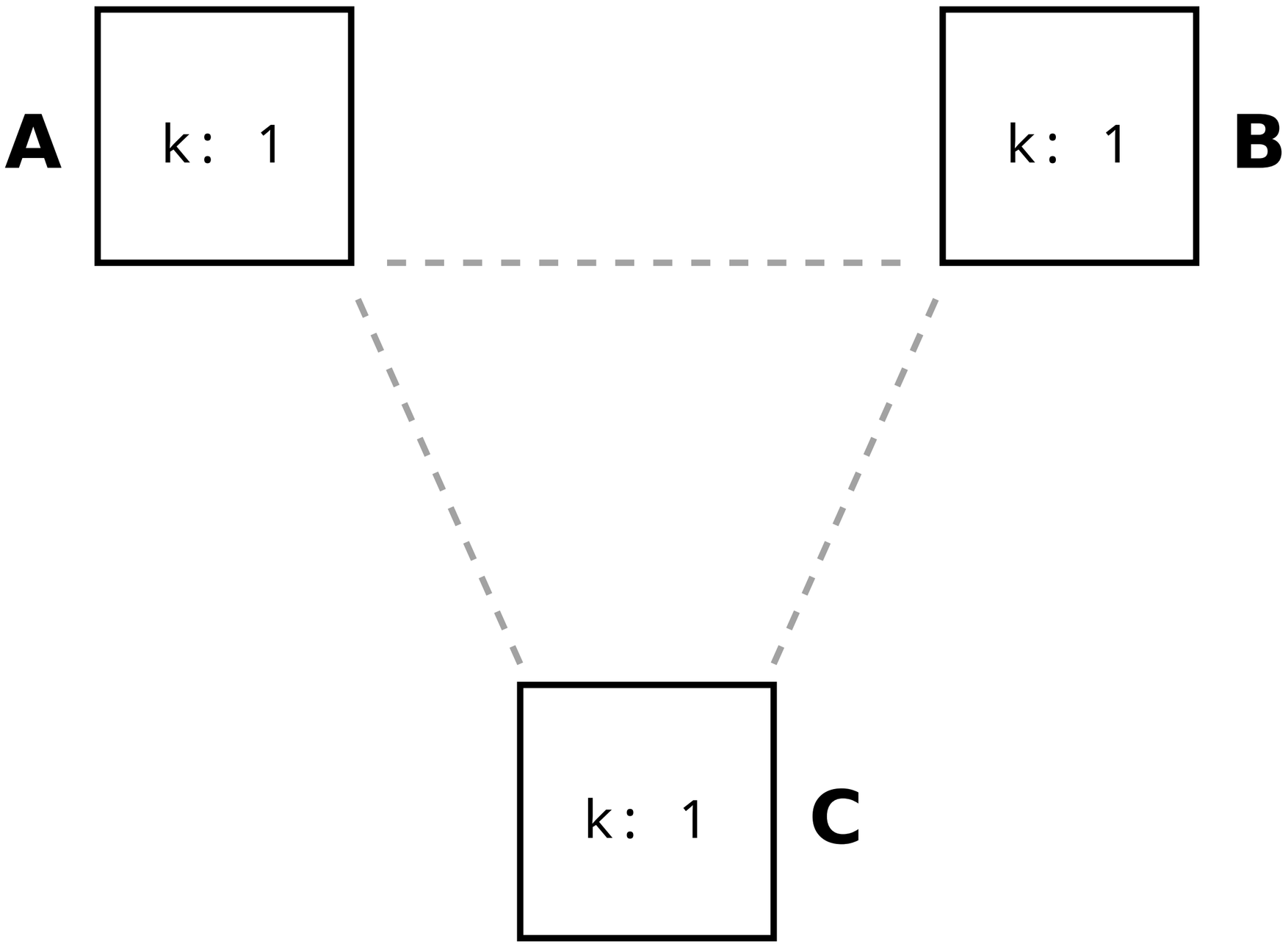

A
k: 1
k: 1
B
k: 1
C

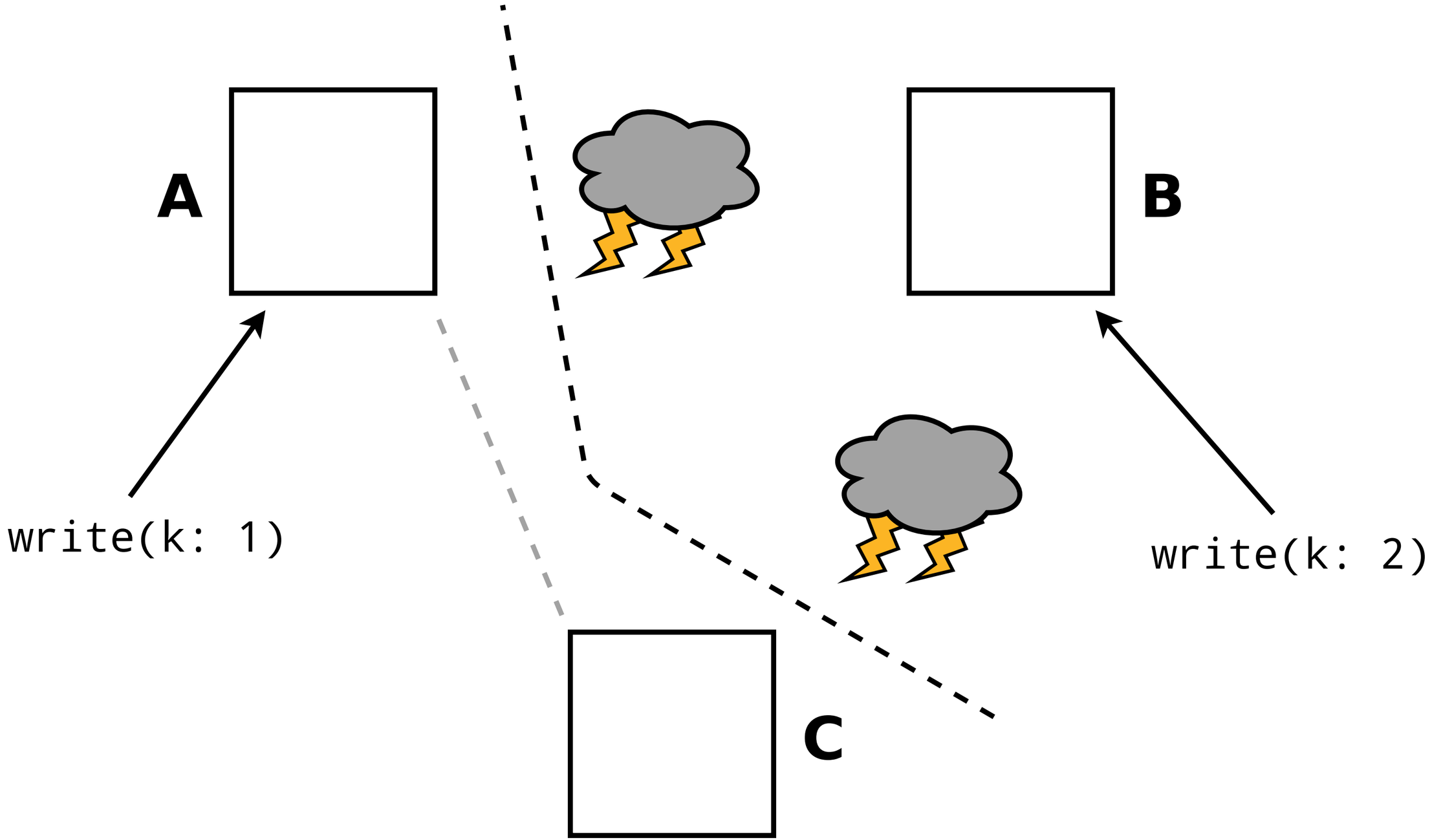

A
B
C

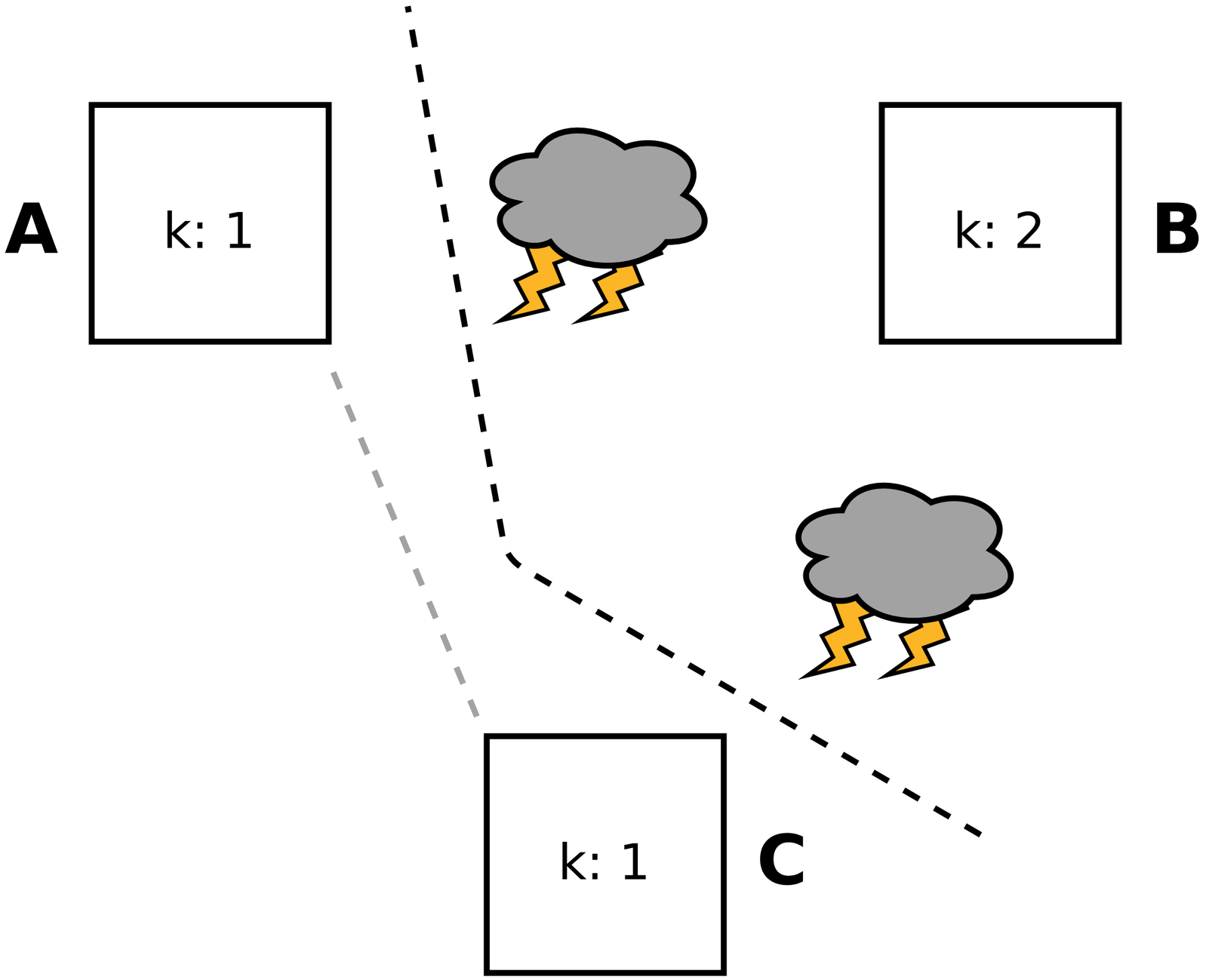

A
B
C
write(k: 1)
write(k: 1)

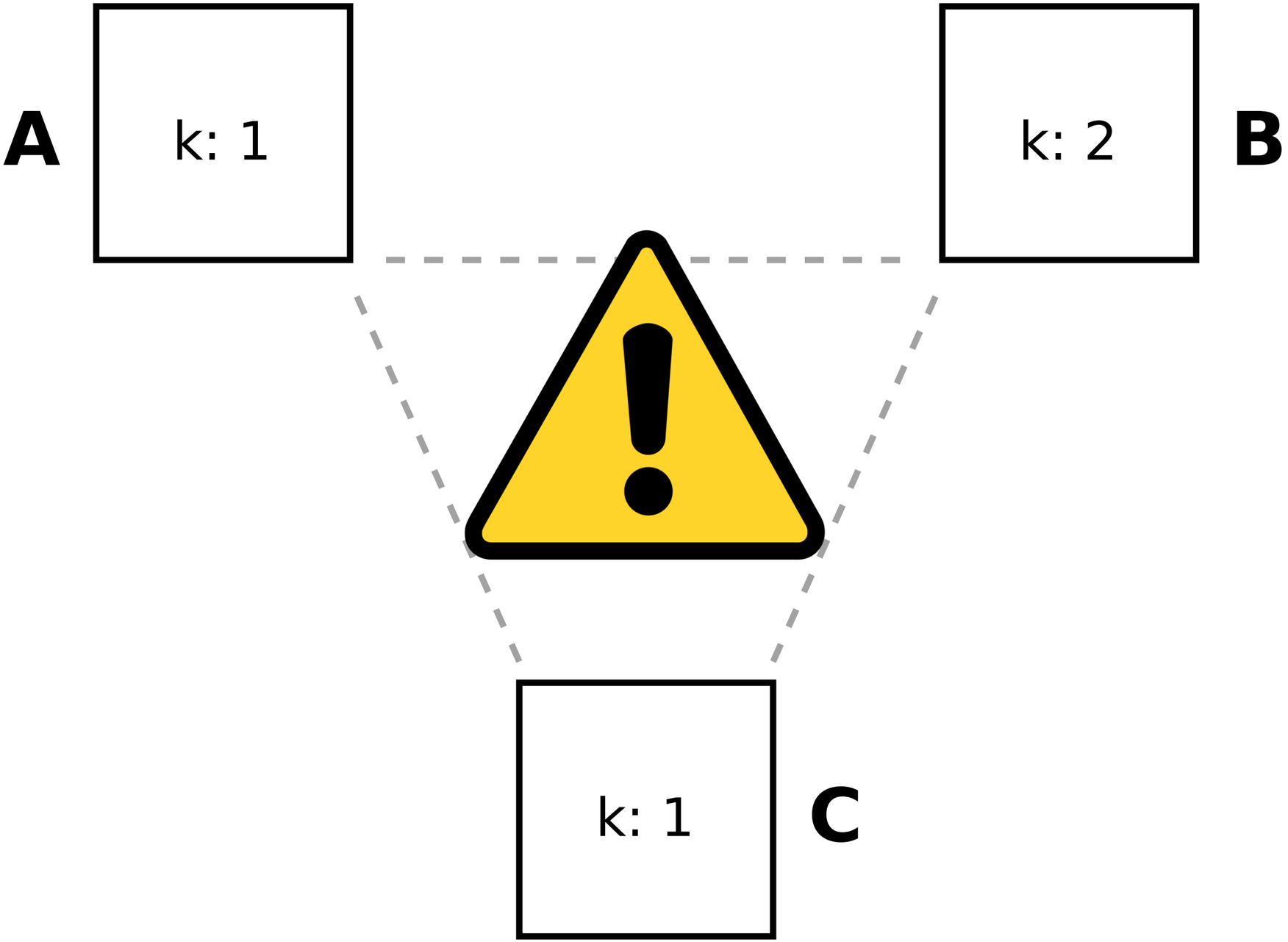

A
k: 1
B
k: 1
C


A
k: 1
k: 1
B
k: 1
C


A
k: 1
k: 1
B
k: 1
C


A
B
C
write(k: 1)
write(k: 2)


A
k: 1
k: 2
B
k: 1
C


A
k: 1
k: 2
B
k: 1
C

MNESIA

- built on BEAM tools
- transactions (AP or CP)
- has some flaws..

CAN WE DO BETTER?

CAN WE DO ~~BETTER~~ DIFFERENT?

CRDTs

CENTRAL REGIONAL DENTAL TESTING SERVICES [1]

[1] http://crdts.org

CONFLICT-FREE REPLICATED DATA TYPES

CRDT

- a data type
- a set of functions

Example

```
type t() :: integer()

@spec increment(t()) :: t()
@spec decrement(t()) :: t()
```

Example

```
type t() :: Set.t()

@spec add(t(), term()) :: t()
@spec remove(t(), term()) :: t()
```

EVENTUAL CONSISTENCY

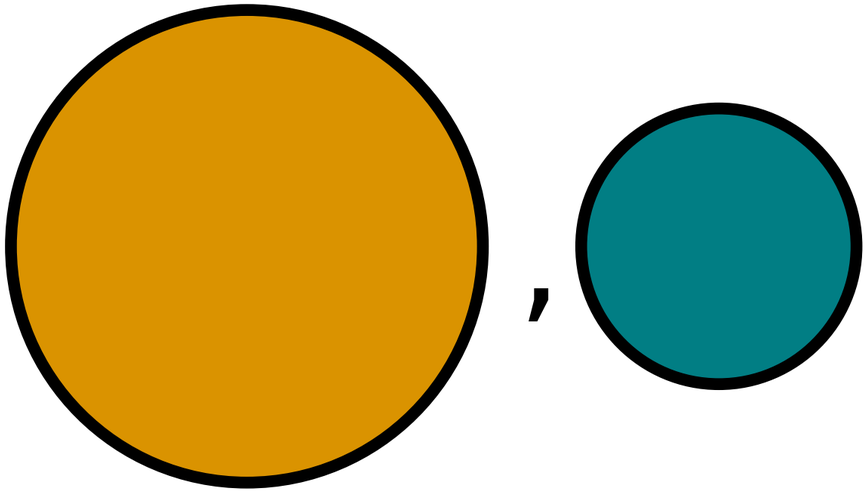
+

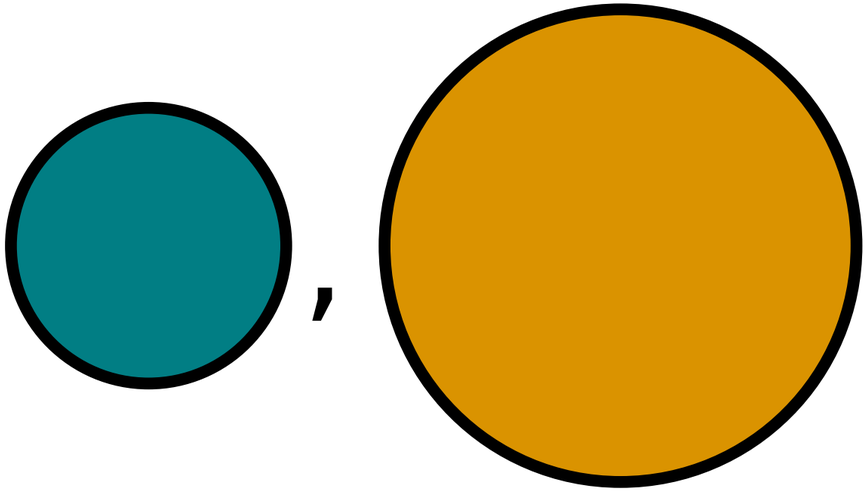
OPTIMISTIC REPLICATION

State-based CRDTs

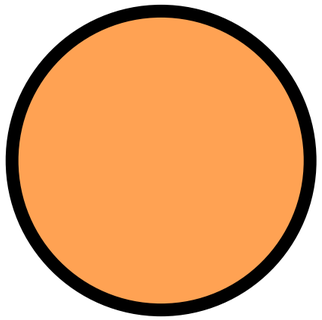
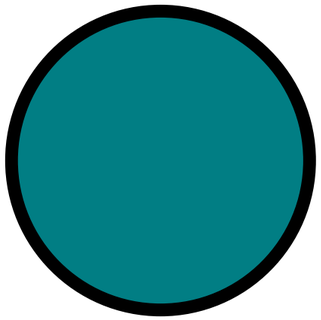
DATA TYPE

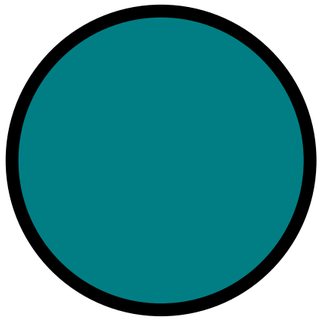
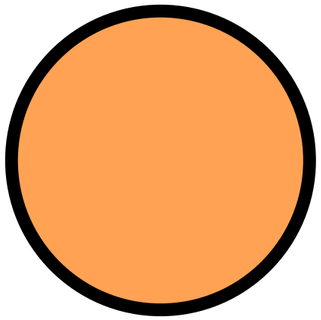
COMPARE

compare() = true

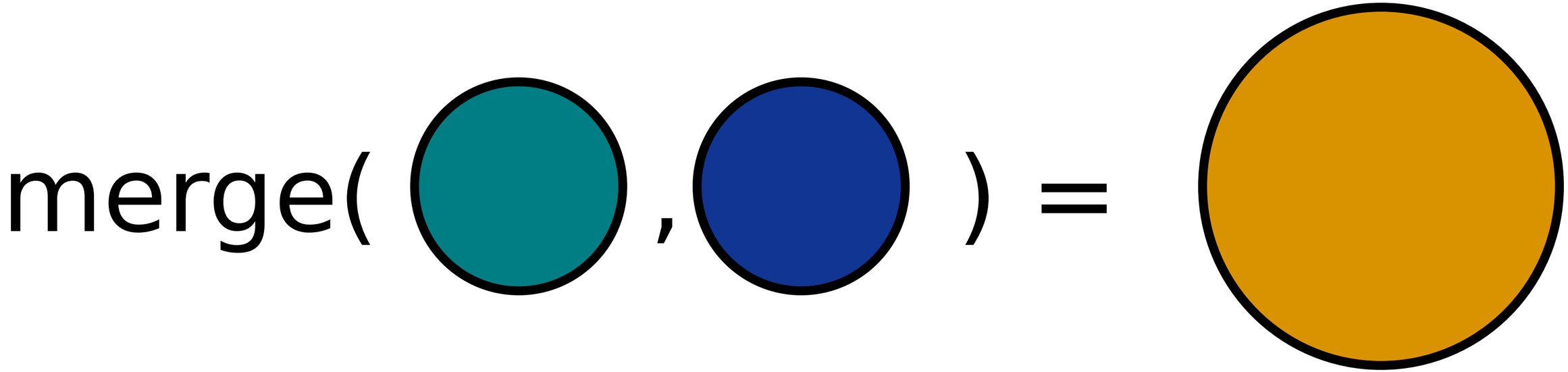
compare() = false

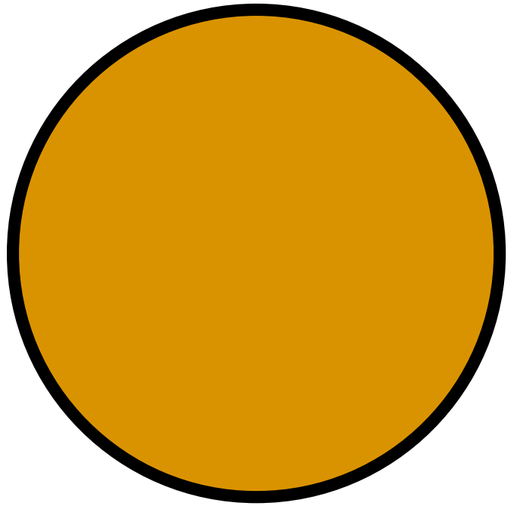
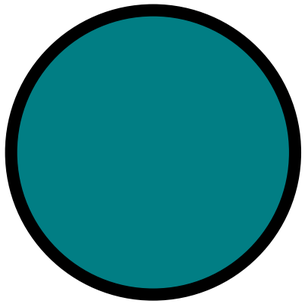
partial order

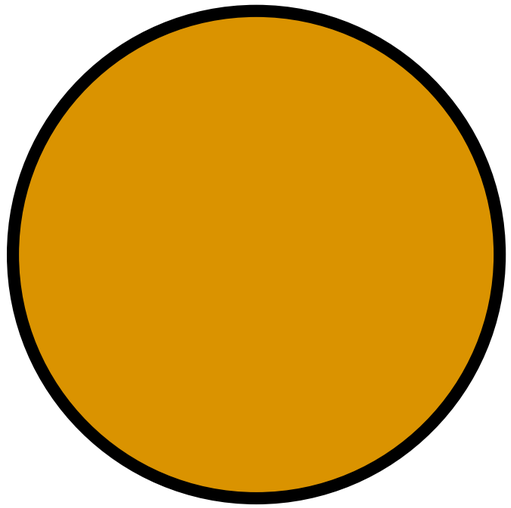
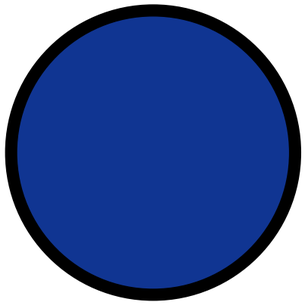
compare( , ) = false

compare( , ) = false

MERGE

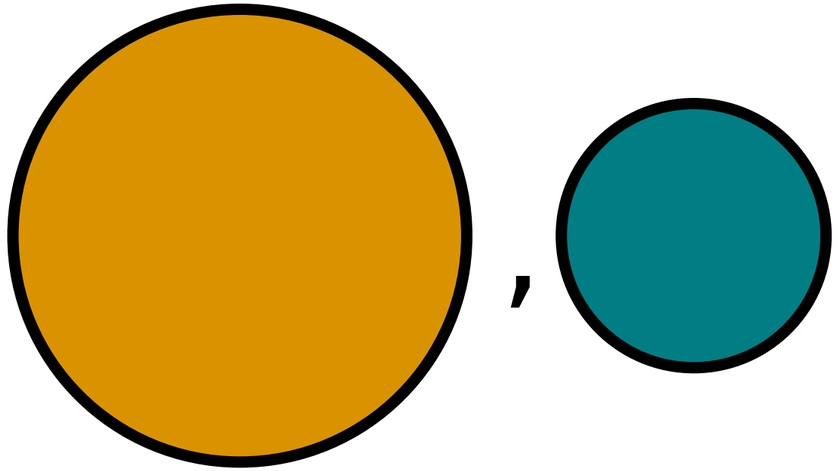

merge(,) =

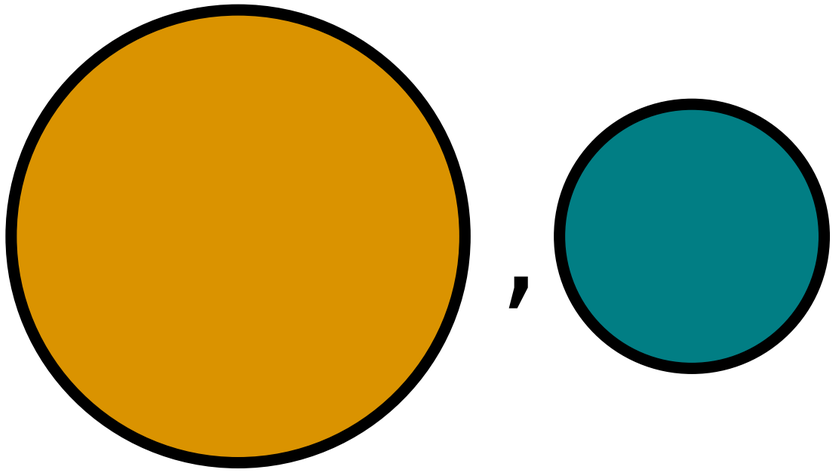
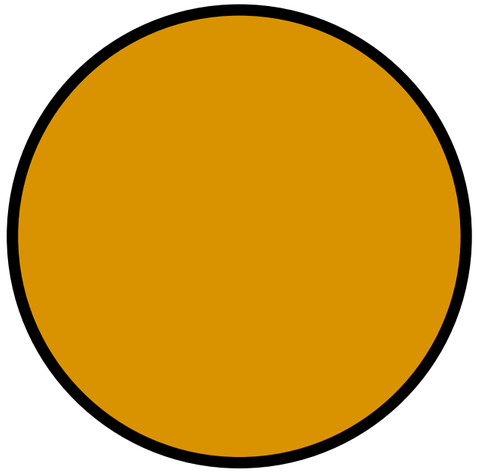
compare( , ) = true

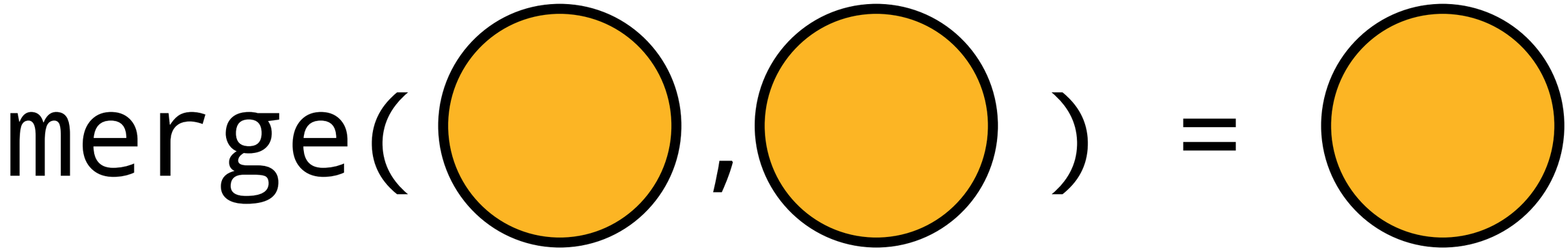
compare( , ) = true

least upper bound

OBSERVATIONS!

compare() = true

merge() = 


merge(,) =

join semilattice

Example

Example

```
@type t() :: Set.t()

def compare(s1, s2), do: ...

def merge(s1, s2), do: ...
```

Example

```
@type t() :: Set.t()

def compare(s1, s2), do: Set.super_or_eq?(s1, s2)

def merge(s1, s2), do: ...
```

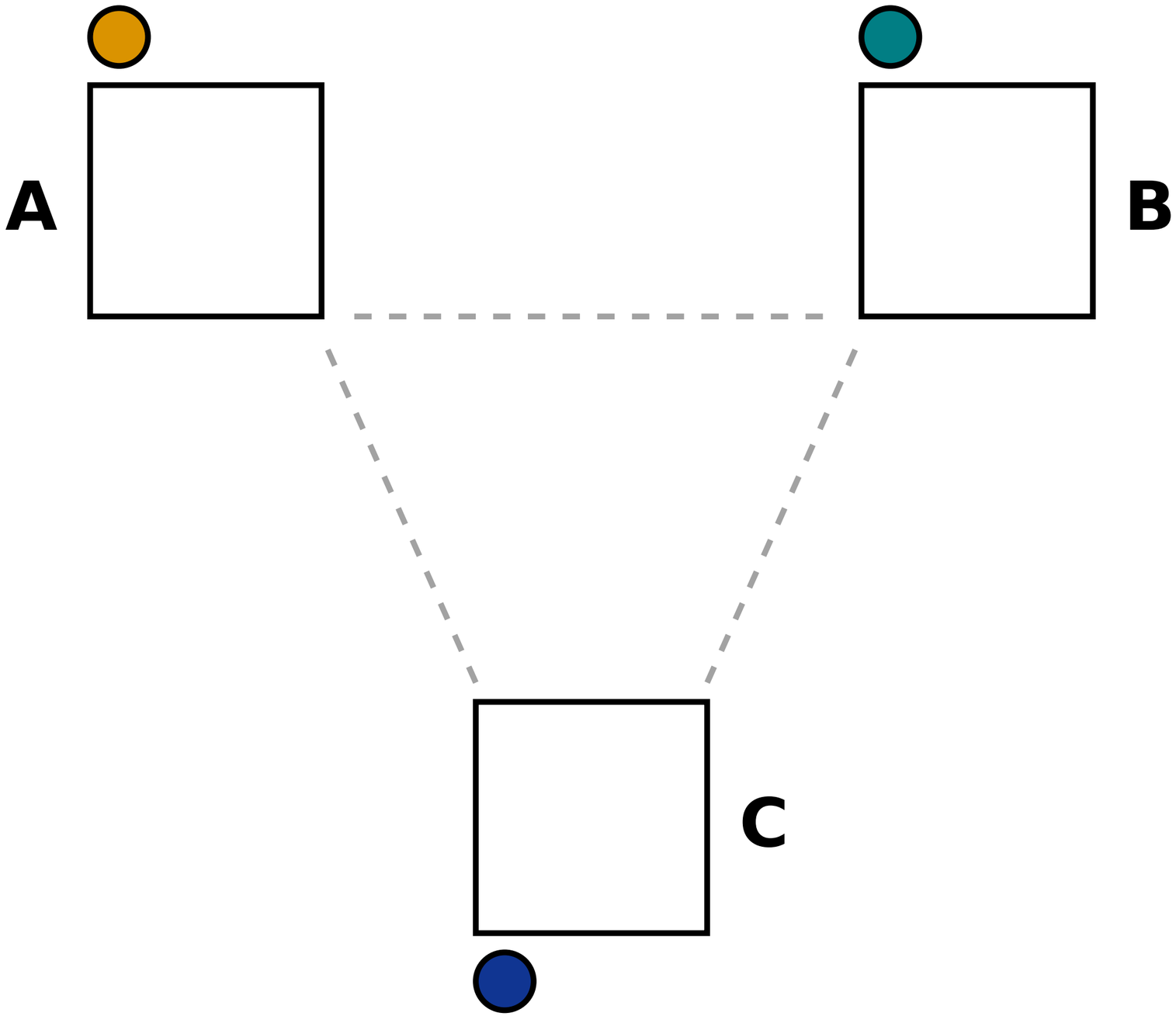
Example

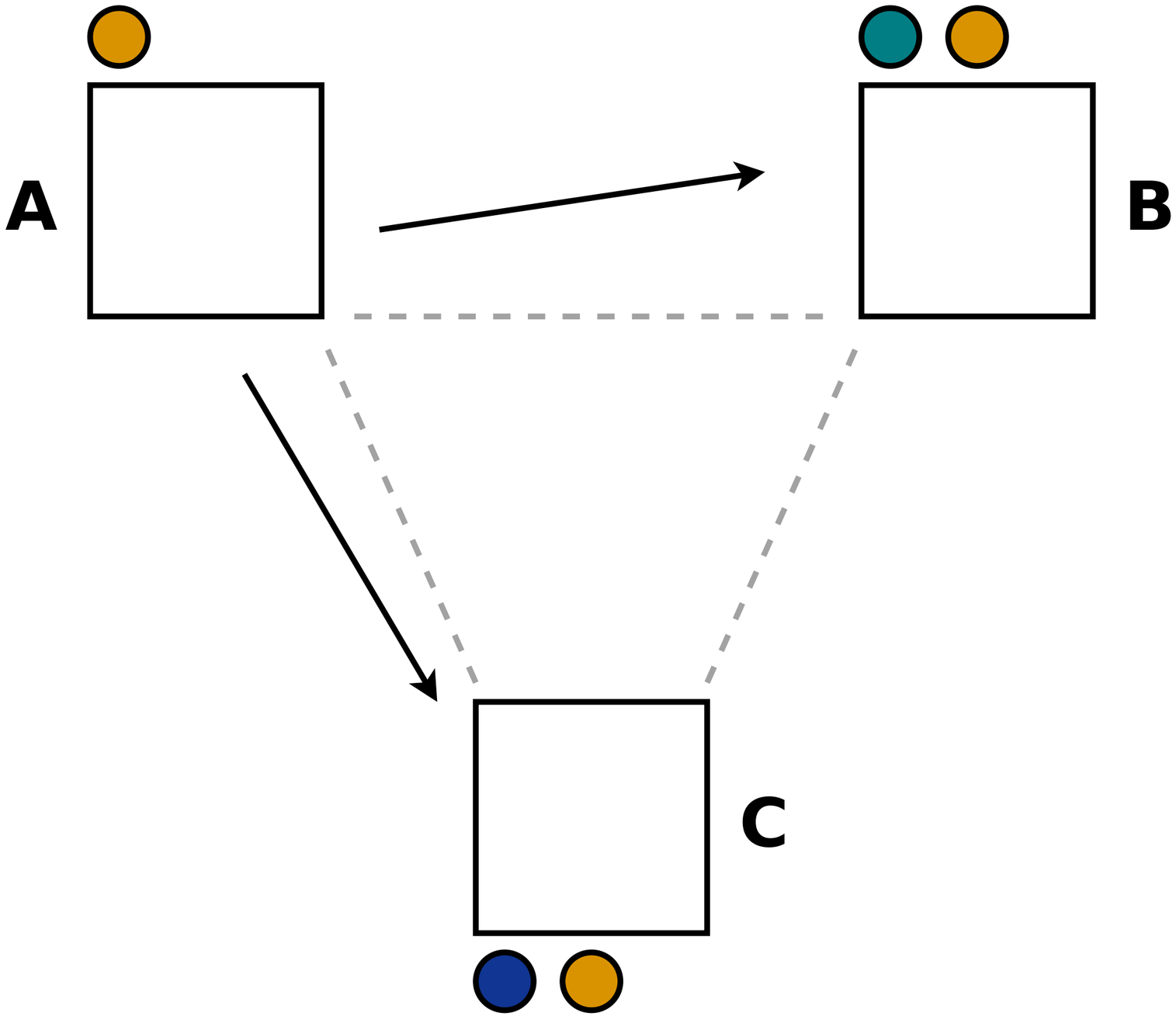
```
@type t() :: Set.t()

def compare(s1, s2), do: Set.super_or_eq?(s1, s2)

def merge(s1, s2), do: Set.union(s1, s2)
```

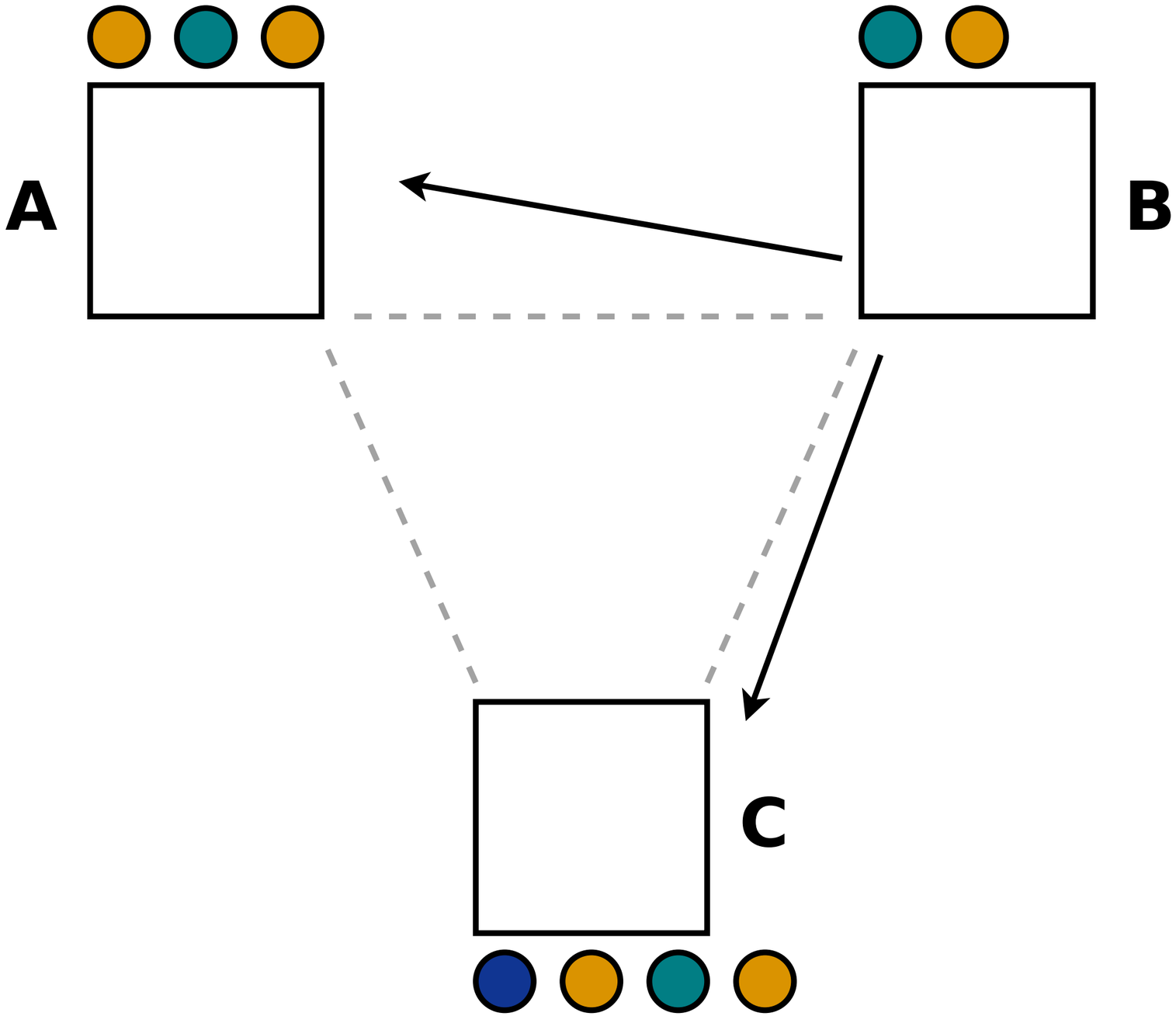
Example

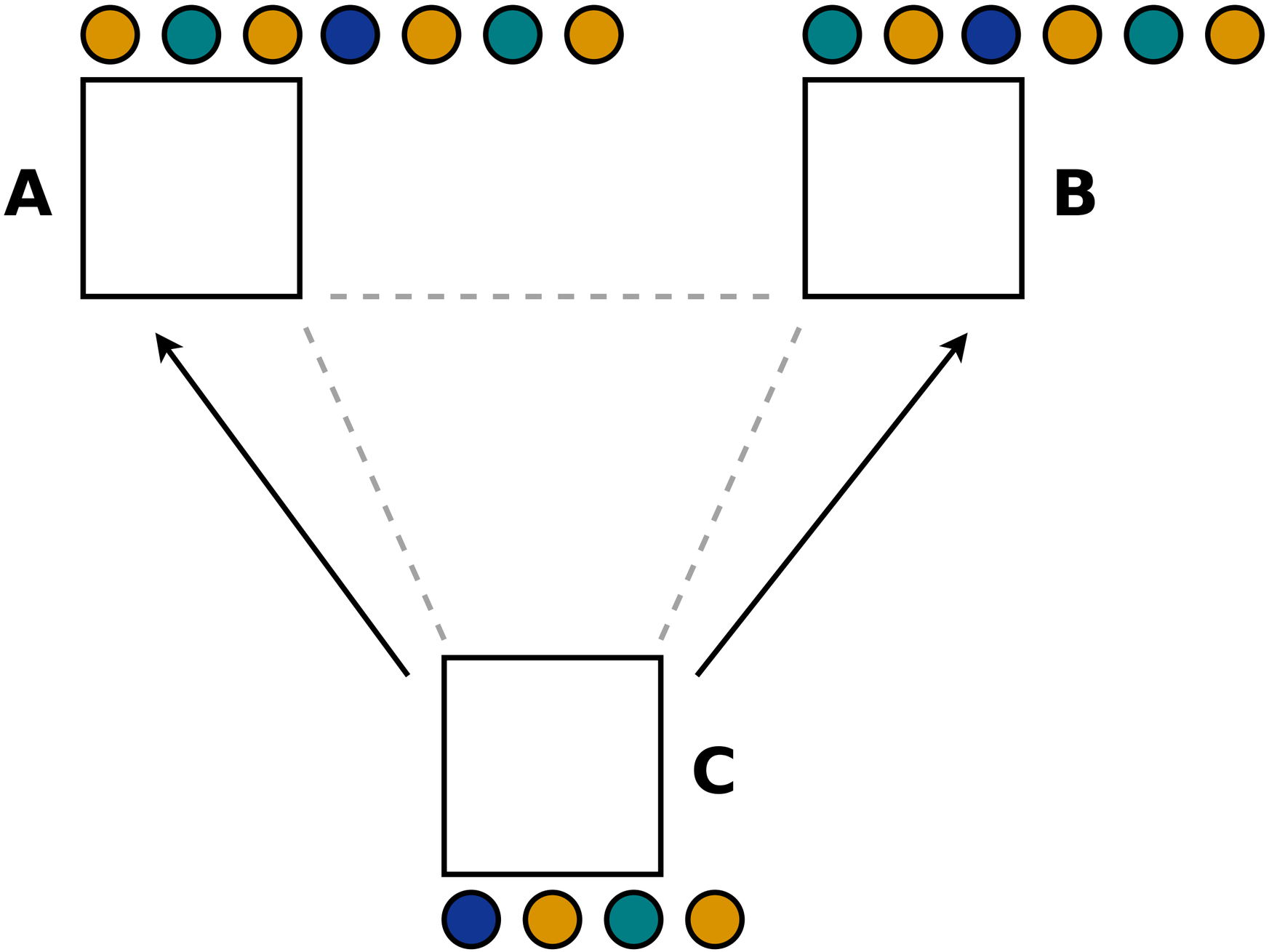
```
merge({1, 2}, {3})
#=> {1, 2, 3}

compare({1, 2, 3}, {1, 2})
#=> true

compare({1, 2, 3}, {3})
#=> true
```

Protocol

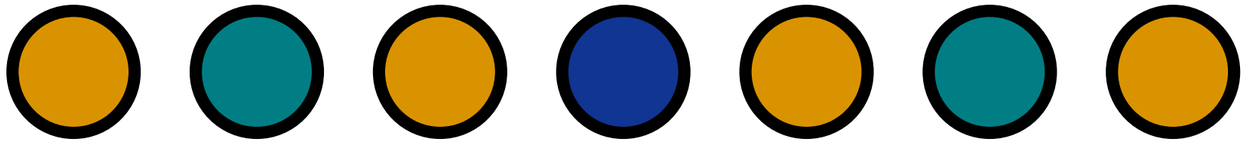

A
B
C


A
B
C

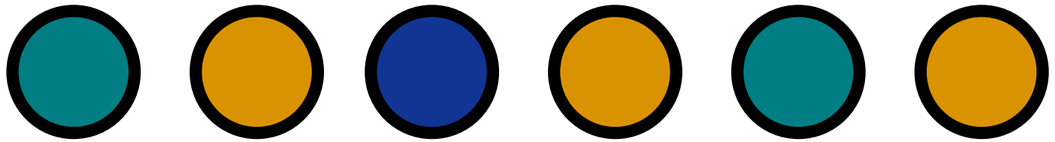

A
B
C


A
B
C

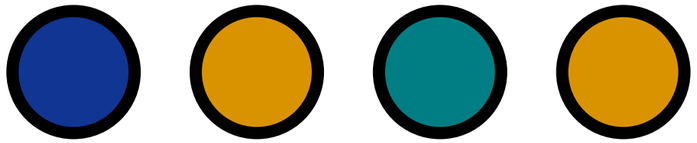
A



B



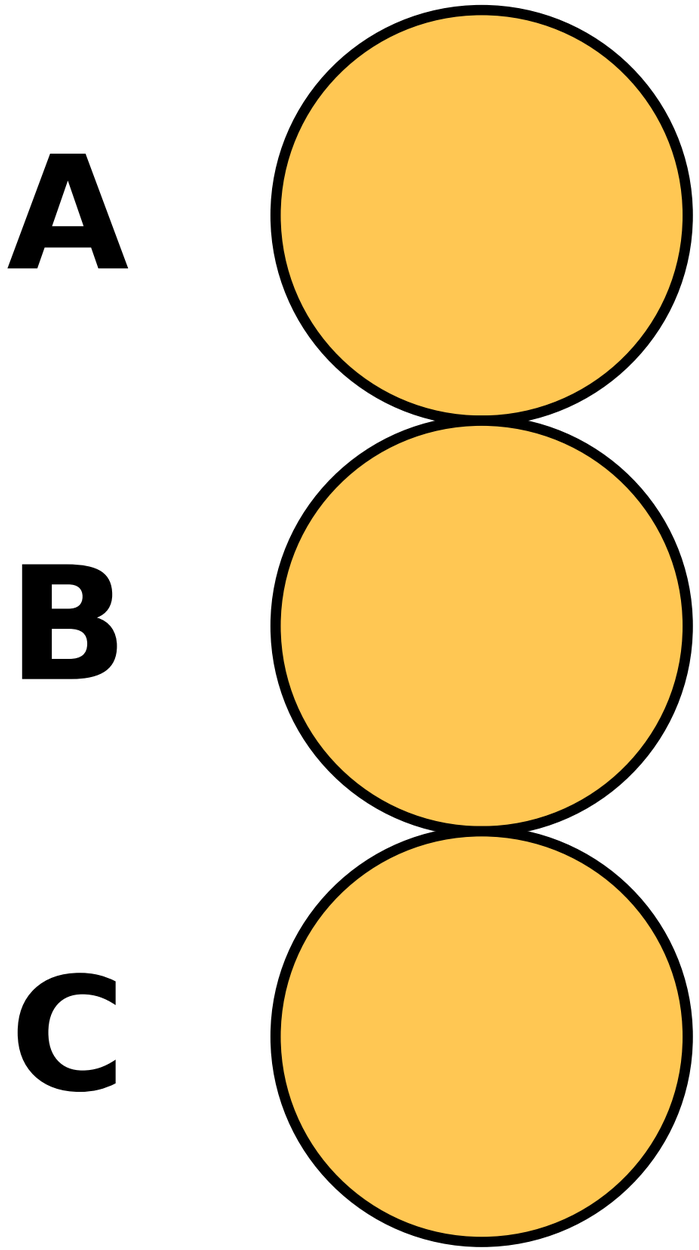
C

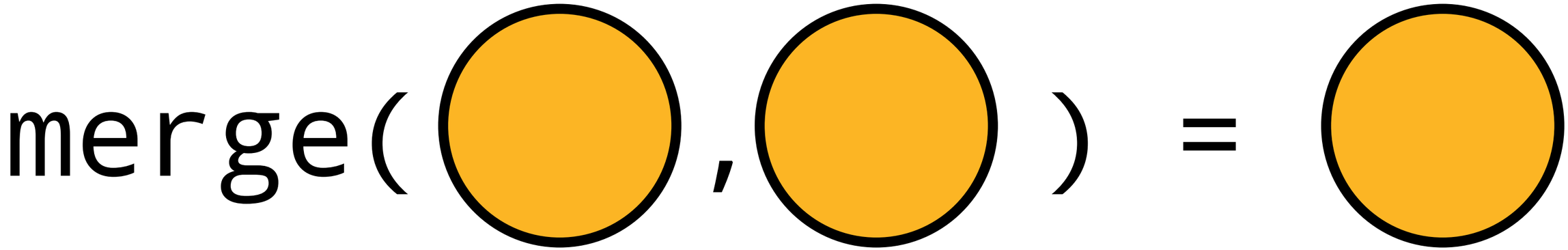


A

B

C


A
B
C


merge(,) =

FUNCTIONS

FUNCTIONS

MUTATORS

MUTATORS

```
@spec m(t(), [term()]) :: t()
```

MERGE, COMPARE

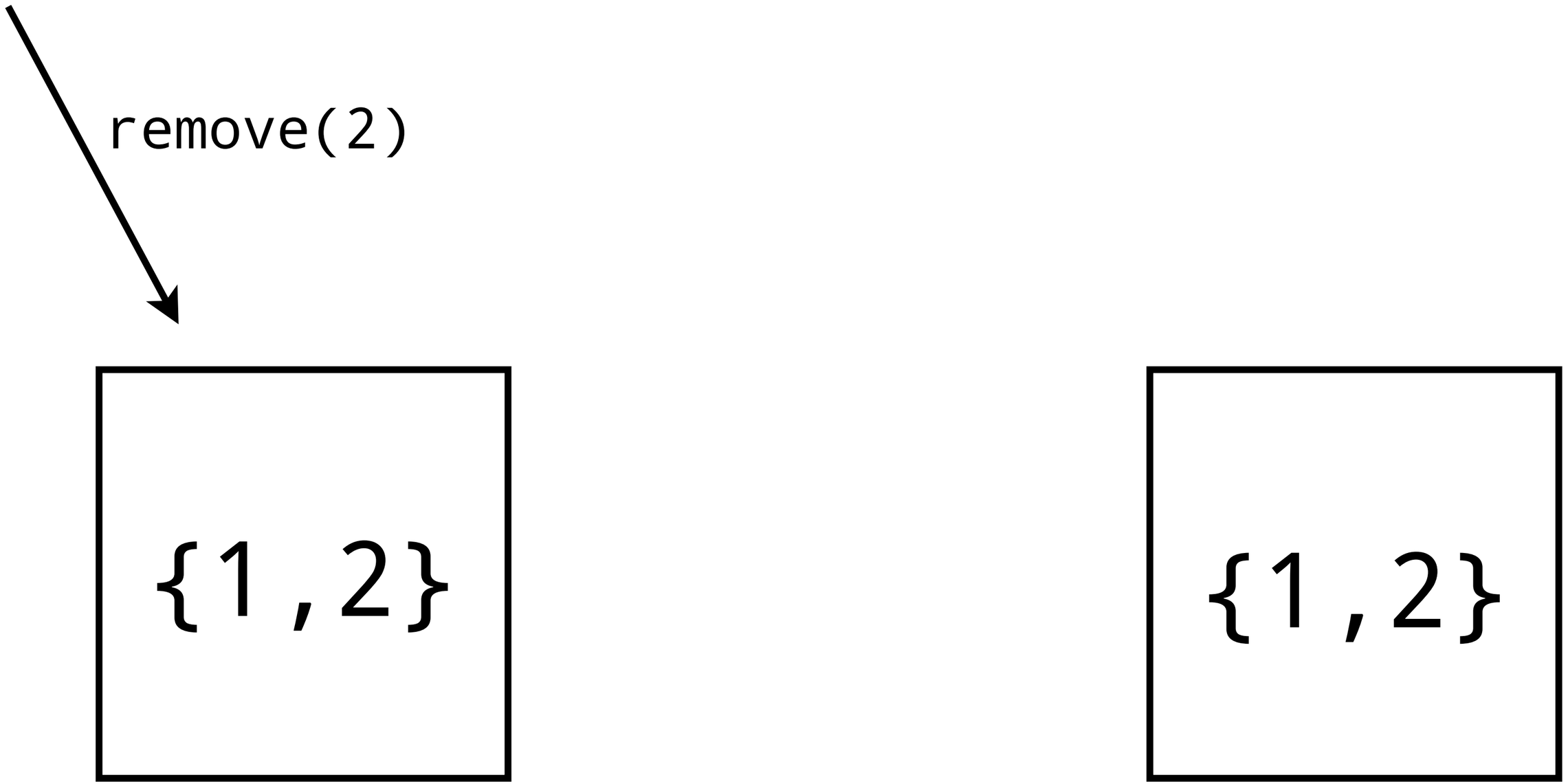
≠

MUTATOR

Example

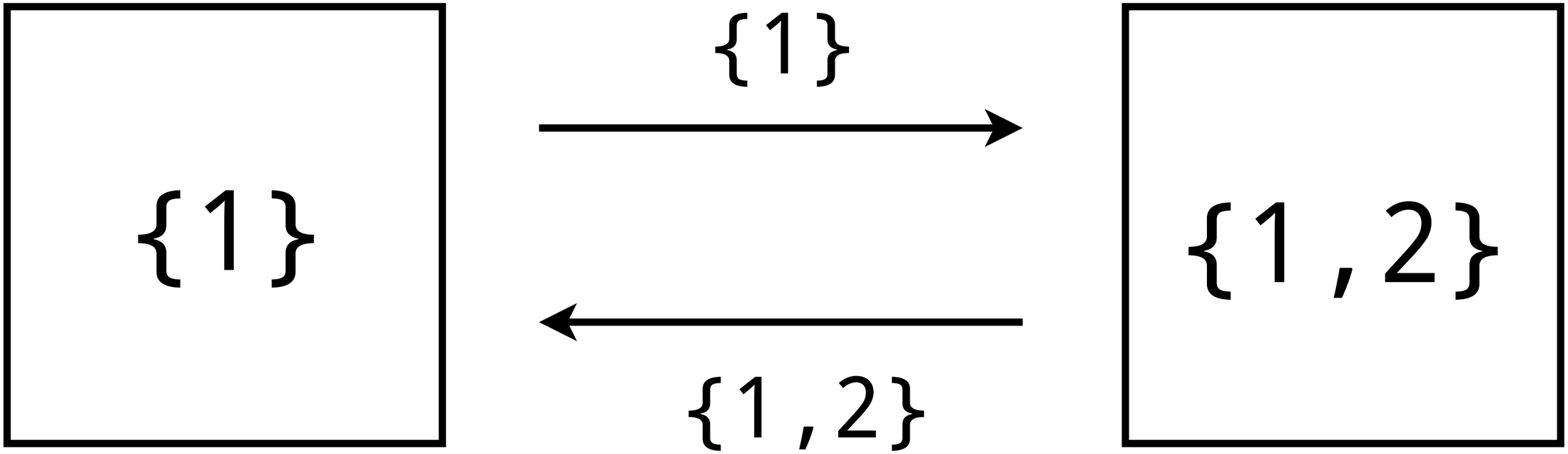
Example

```
def add(s, el), do: Set.union(s, {el})

def remove(s, el), do: Set.subtract(s, {el})
```


remove(2)
{1,2}
{1,2}

$\{1\}$

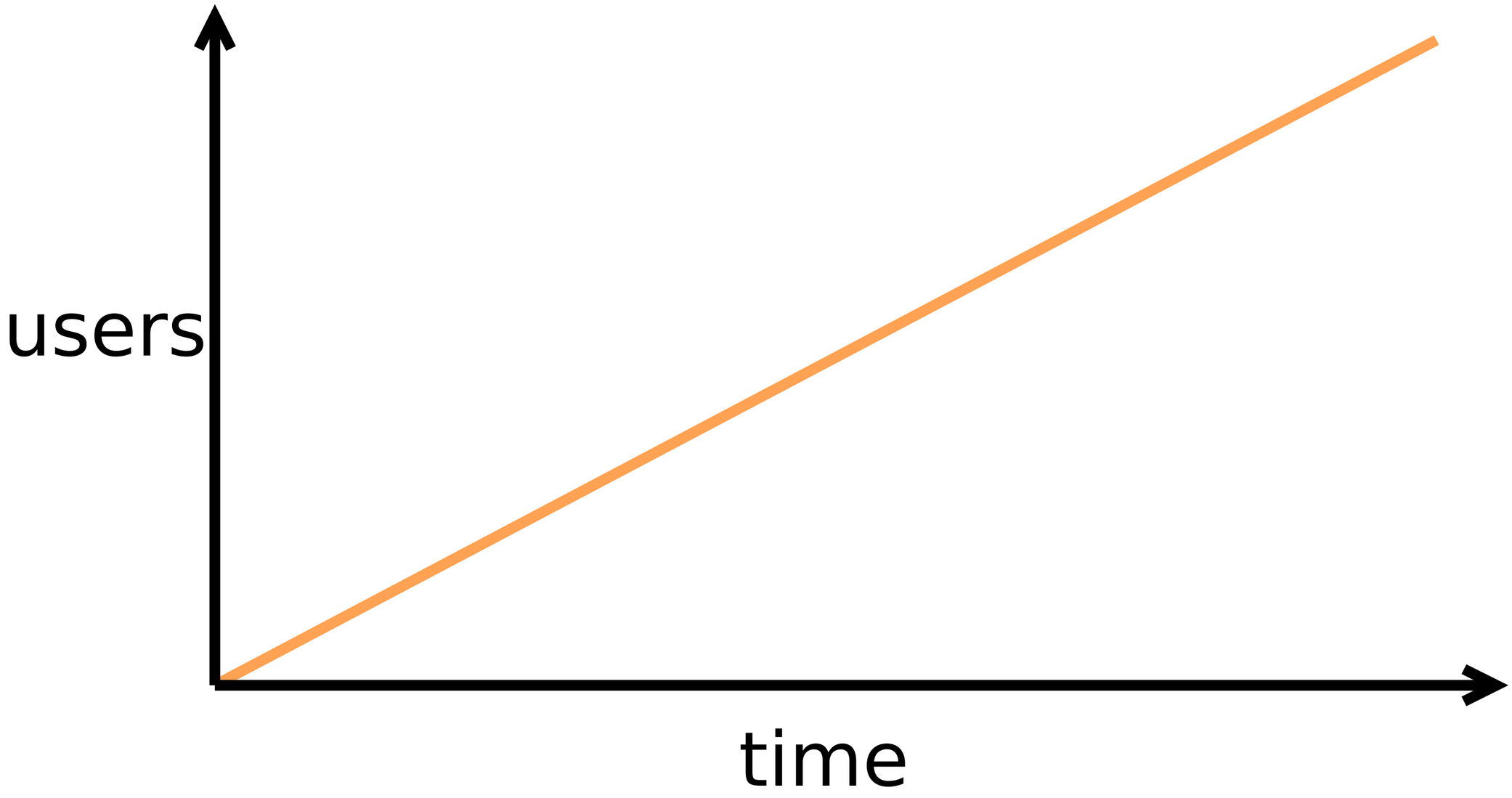
$\{1, 2\}$


{1}
{1}
{1,2}
{1,2}

$\{1, 2\}$

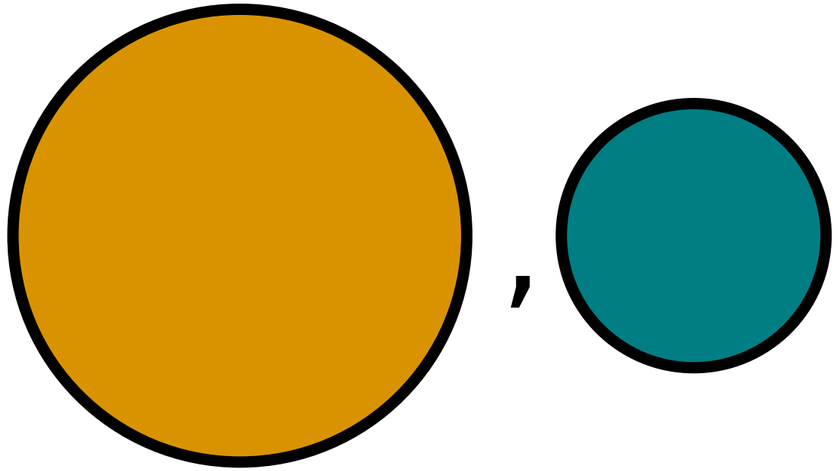
$\{1, 2\}$

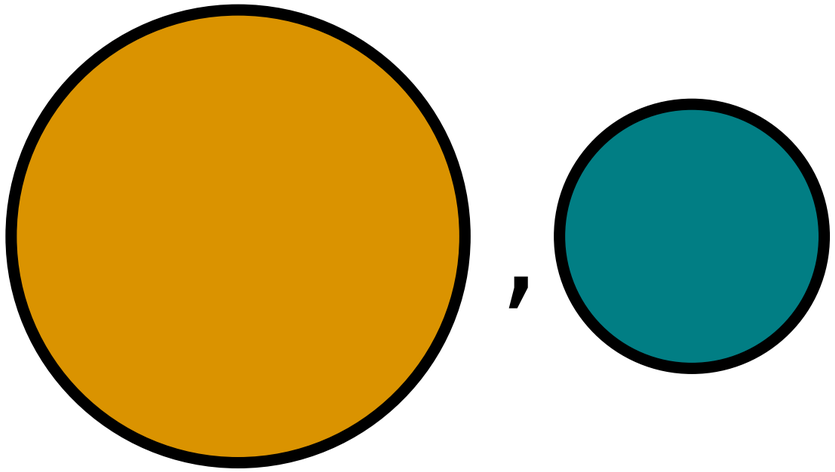
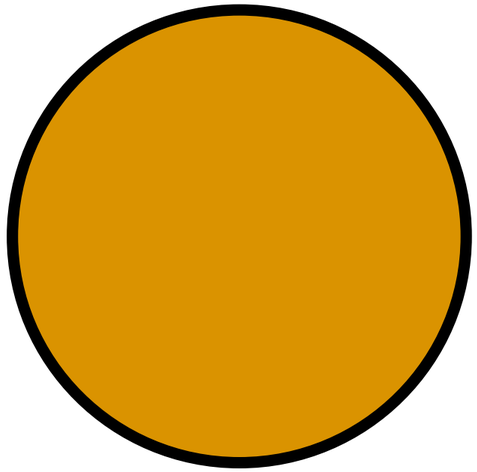
G-SET


users
time

Problem 1

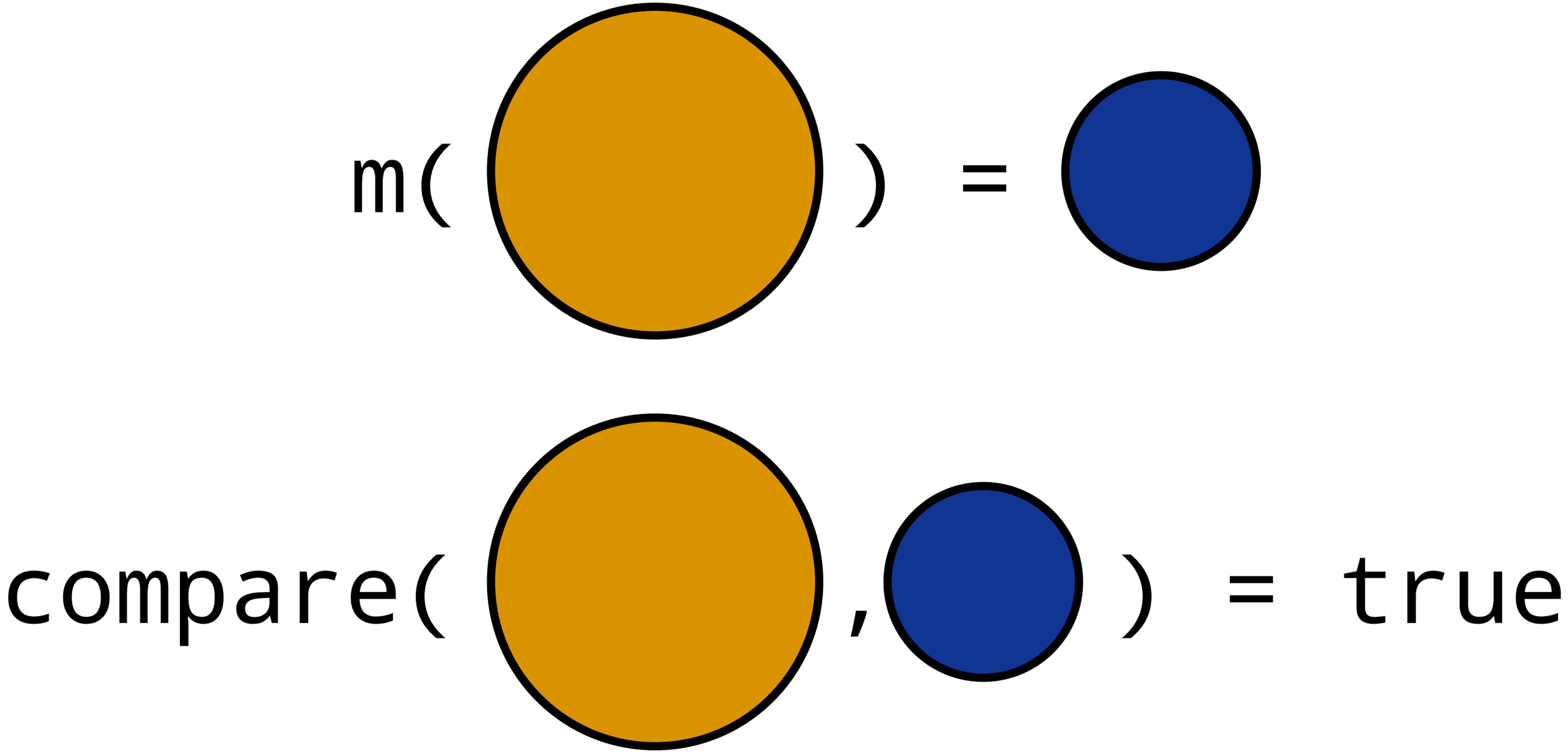
- "bigger" values dominate

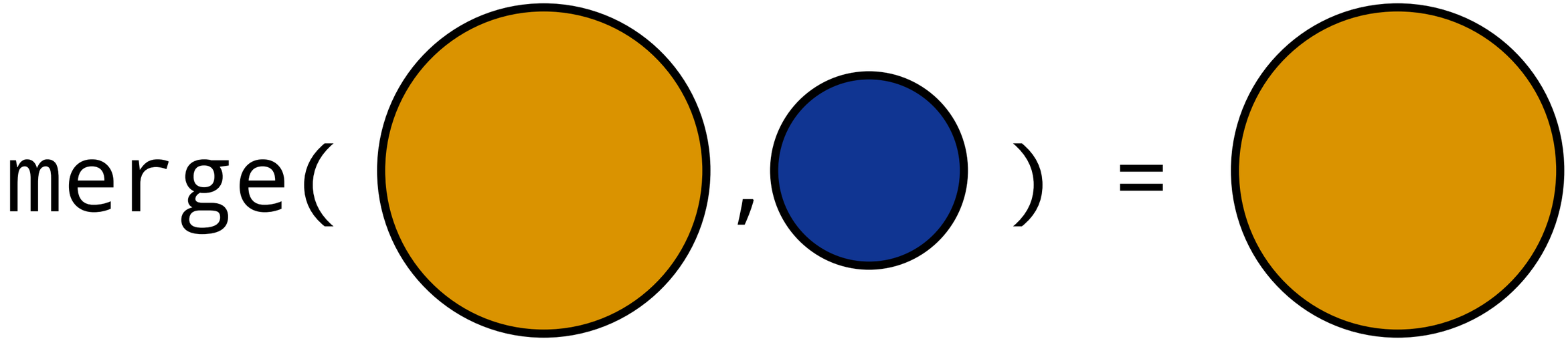
compare() = true

merge() = 

Problem 1

- "bigger" values dominate
- removing things "shrink"


m( ) =
compare(,) = true


merge(
,
) =

Removes must inflate!

SOLUTION 1
TOMBSTONES

TOMBSTONES

```
@type t :: {Set.t(), Set.t()}

def add({s, t}, el) do
  {Set.union(s, {el}), t}
end

def remove({s, t}, el) do
  {s, Set.union(t, {el})}
end
```

```
a = {{}, {}} |> add(1) #=> {{1}, {}}
b = a |> add(2)        #=> {{1, 2}, {}}
c = b |> remove(1)     #=> {{1, 2}, {1}}
```

```
a = {{}, {}} |> add(1) #=> {{1}, {}}
b = a |> add(2)        #=> {{1, 2}, {}}
c = b |> remove(1)     #=> {{1, 2}, {1}}

compare(b, a) = true
compare(c, b) = true
```

```elixir
def compare({s1, t1}, {s2, t2}) do
  Set.super_or_eq?(s1, s2) and Set.super_or_eq?(t1, t2)
end

def merge({s1, t1}, {s2, t2}) do
  {Set.union(s1, s2), Set.union(t1, t2)}
end
```

```elixir
def element?({s, t}, el) do
  s
  |> Set.subtract(t)
  |> Set.element?(el)
end
```

```
s =
  {{}, {}}

element?(s, 1) #=> false
element?(s, 2) #=> false
```

```elixir
s =
  {{}, {}}
  |> add(1)      #=> {{1}, {}}

element?(s, 1) #=> true
element?(s, 2) #=> false
```

```
s =
  {{}, {}}
  |> add(1)      #=> {{1}, {}}
  |> add(2)      #=> {{1, 2}, {}}

element?(s, 1) #=> true
element?(s, 2) #=> true
```

```
s =
  {{}, {}}
  |> add(1)      #=> {{1}, {}}
  |> add(2)      #=> {{1, 2}, {}}
  |> remove(1)   #=> {{1, 2}, {1}}

element?(s, 1) #=> false
element?(s, 2) #=> true
```

```elixir
s =
  {{}, {}}
  |> add(1)      #=> {{1}, {}}
  |> add(2)      #=> {{1, 2}, {}}
  |> remove(1)   #=> {{1, 2}, {1}}
  |> add(1)      #=> {{1, 2}, {1}}

element?(s, 1) #=> false
element?(s, 2) #=> true
```

elements can be removed, but never added again 😢

2P-SET

SOLUTION 2
CAUSAL CONTEXTS[2]

[2] https://arxiv.org/pdf/1603.01529.pdf

Problem 2

Problem 2

whole state sent to each replica, multiple times

SOLUTION

DELTA STATE REPLICATED DATA TYPES[2]

[2] https://arxiv.org/pdf/1603.01529.pdf

MUTATOR

```
@spec m(t(), [term()]) :: t()
```

DELTA MUTATOR

```
@spec d(t(), [term()]) :: t()
```

the rule

```
m(state) = merge(state, d(state))
```

Example

```
def add(s, el), do: union(s, {el})

def delta_add(s, el), do: {el}
```

WHY BOTHER?

CONFIDENCE

PURE FUNCTIONS &
DATA STRUCTURES

UNDERSTANDING

RIAK

RIAK
CASSANDRA

RIAK
CASSANDRA
DYNAMO

RIAK
CASSANDRA
DYNAMO
ANTIDOTE

You can use them, too!

LASP

DECLARE VARIABLE

```
:lasp.declare({"set", :state_gset}, :state_gset)
```

UPDATE VALUE

```
:lasp.update({"set", :state_gset}, {:add, 1}, self())
```

READ VALUE

```
:lasp.query({"set", :state_gset})
```

REFERENCES

- A comprehensive study of Convergent and Commutative Replicated Data Types
- Conflict-free Replicated Data Types
- Delta State Replicated Data Types
- Phoenix 1.2 and Beyond
- Efficient State-based CRDTs by Delta-Mutation

Thank you!

@arkgil @_arkgil