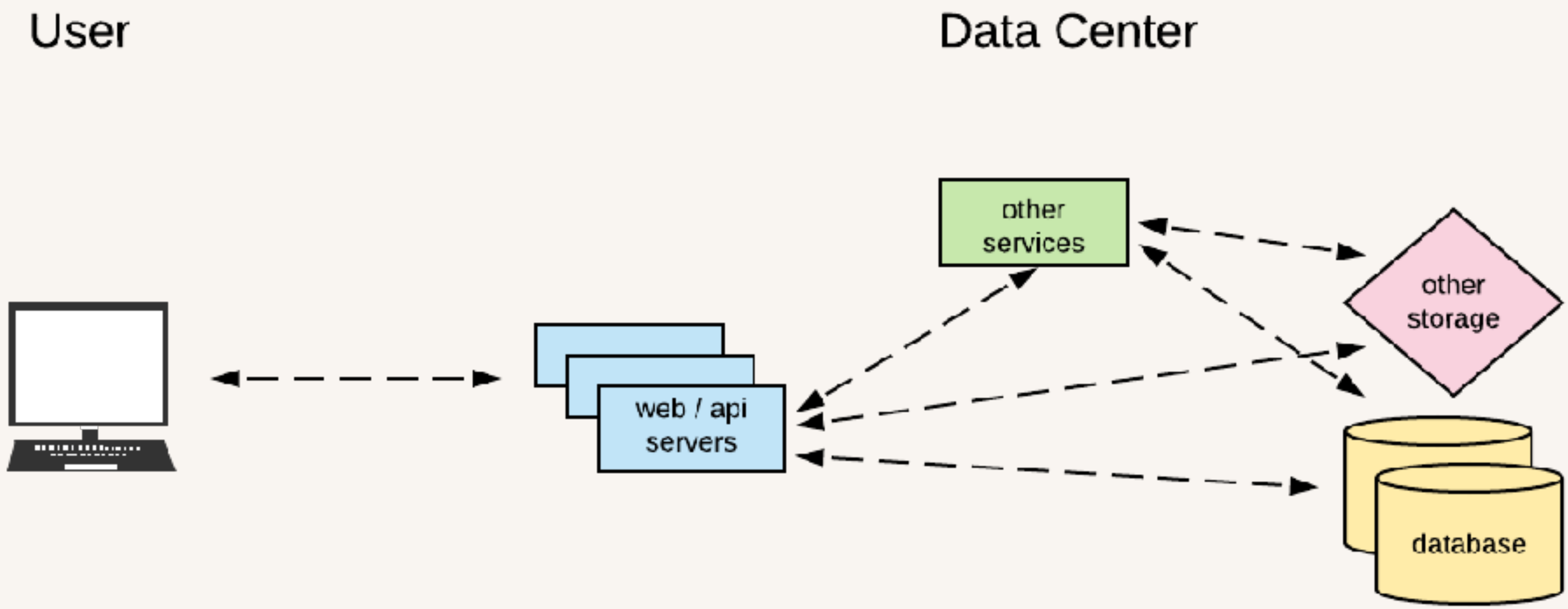
What Came First?

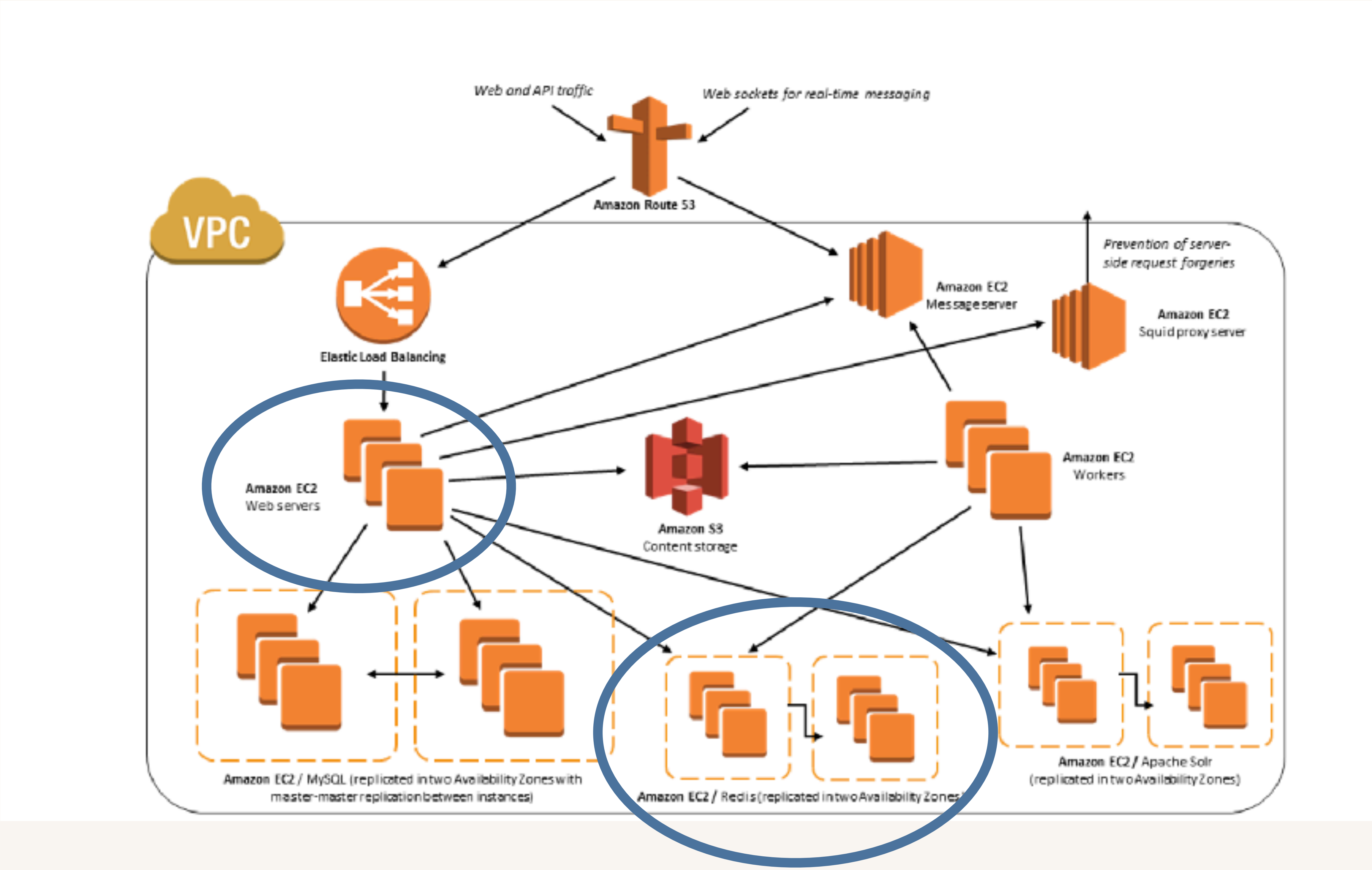
The Ordering of Events in Systems

@kavya719

kavya

the design of concurrent systems


User
Data Center
web / api
servers
other
services
other
storage
database

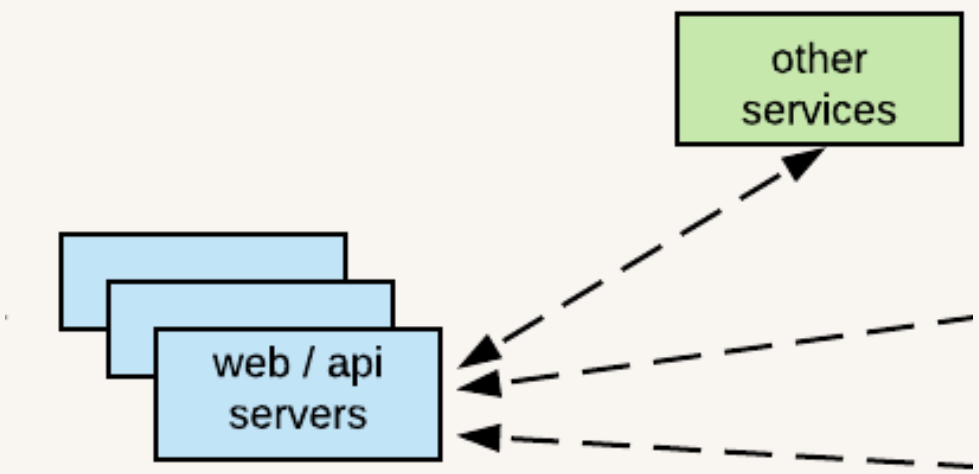


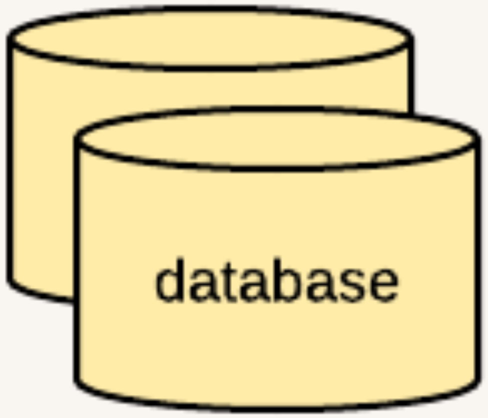
Slack architecture on AWS

systems with **multiple independent actors**.

threads
in a multithreaded program.

nodes
in a distributed system.





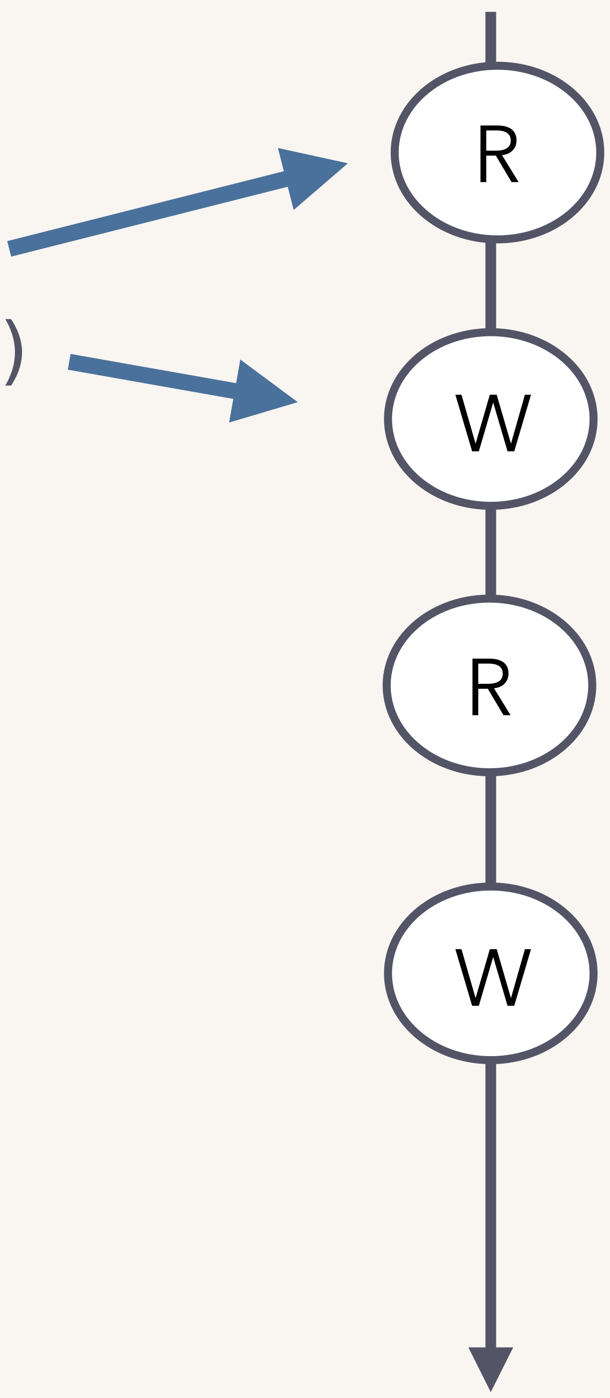
concurrent actors

threads → user-space or system threads

threads → user-space or system threads

```
var tasks []Task

func main() {
  for {
    if len(tasks) > 0 {
      task := dequeue(tasks)
      process(task)
    }
  }
}
```



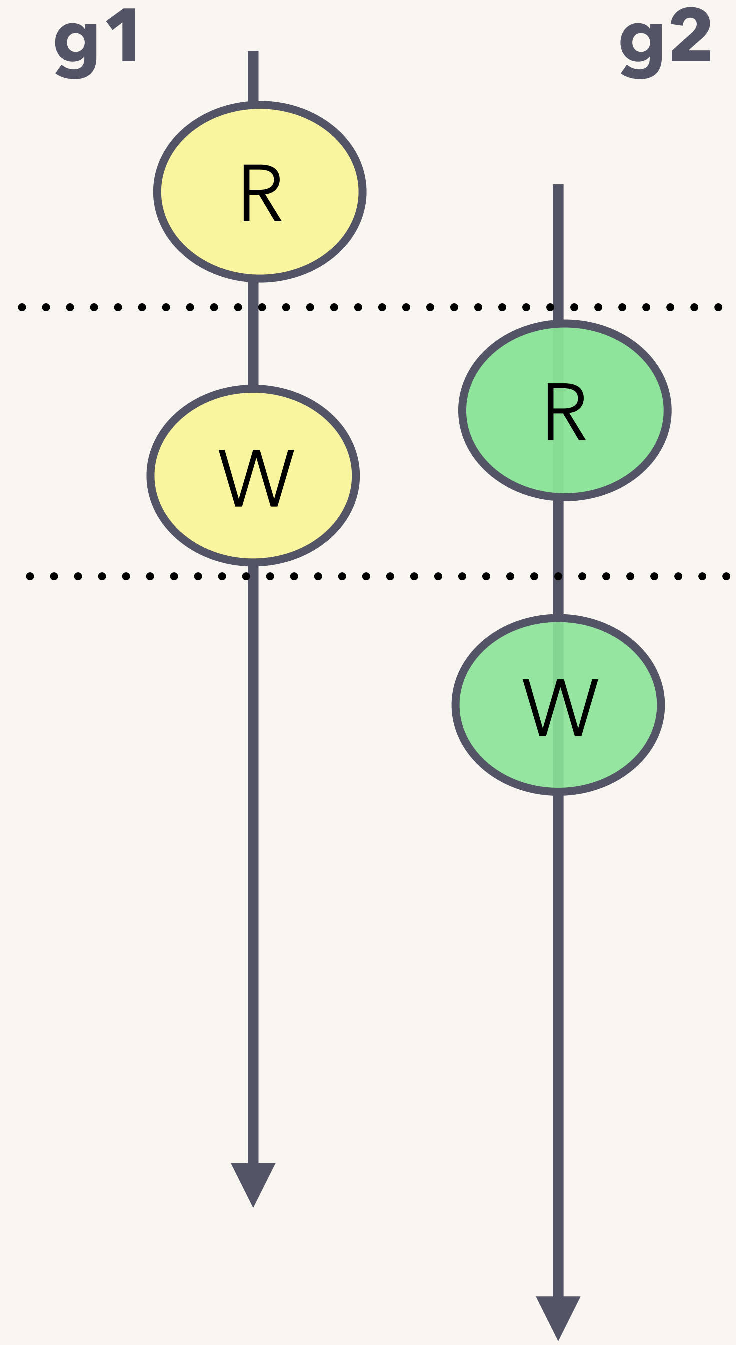
multiple threads:

```
// Shared variable
var tasks []Task

func worker() {
   for len(tasks) > 0 {
      task := dequeue(tasks)
      process(task)
   }
}

func main() {
   // Spawn fixed-pool of worker threads.
   startWorkers(3, worker)

   // Populate task queue.
   for _, t := range hellaTasks {
      tasks = append(tasks, t)
   }
}
```



data race

"when two+ threads concurrently access a shared memory location, at least one access is a write."

…many threads provides concurrency,
may introduce data races.

nodes

→ processes i.e. logical nodes
(but term can also refer to machines i.e. physical nodes).

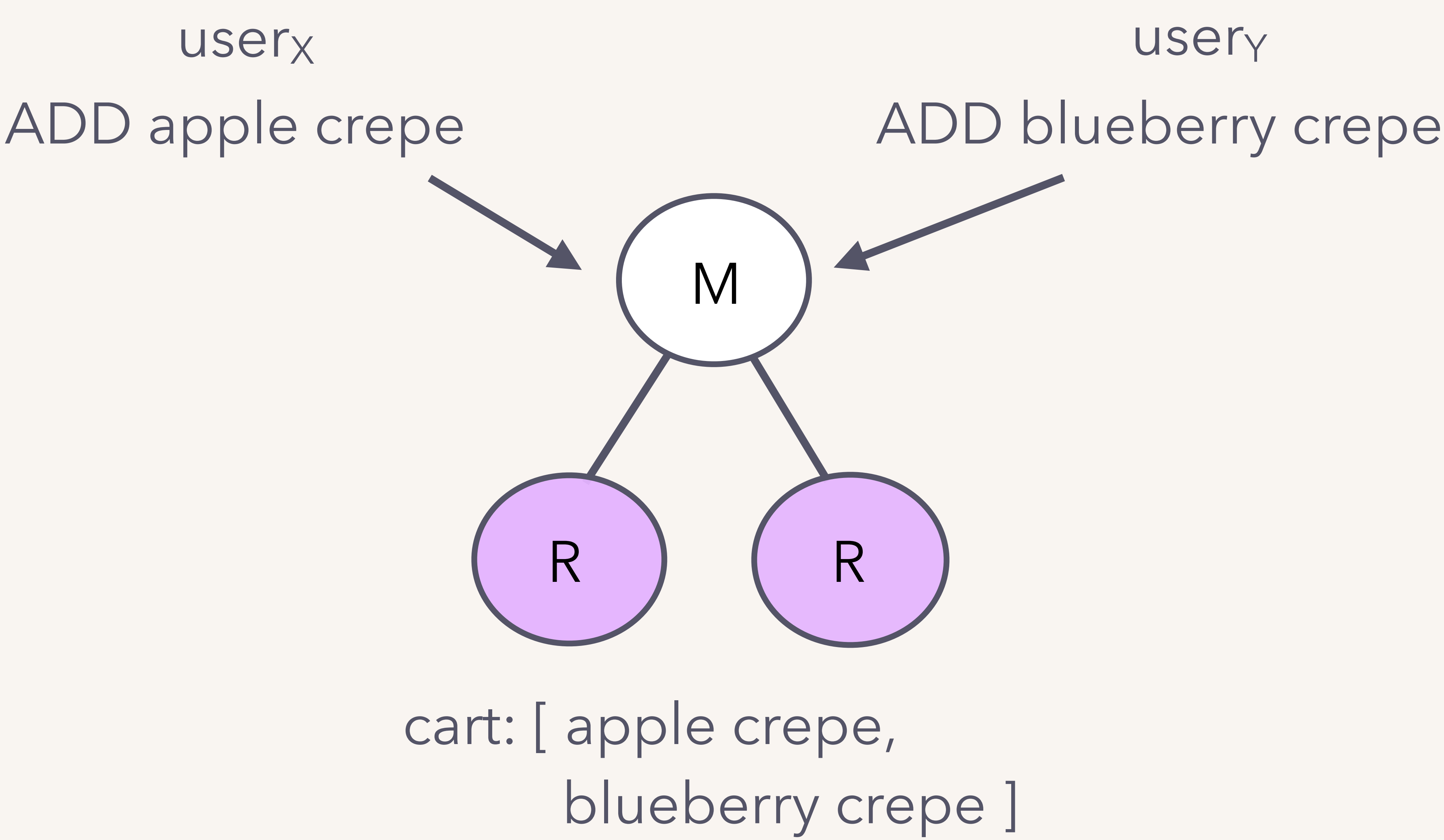
communicate by message-passing i.e. connected by unreliable network, no shared memory.

are sequential.

no global clock.

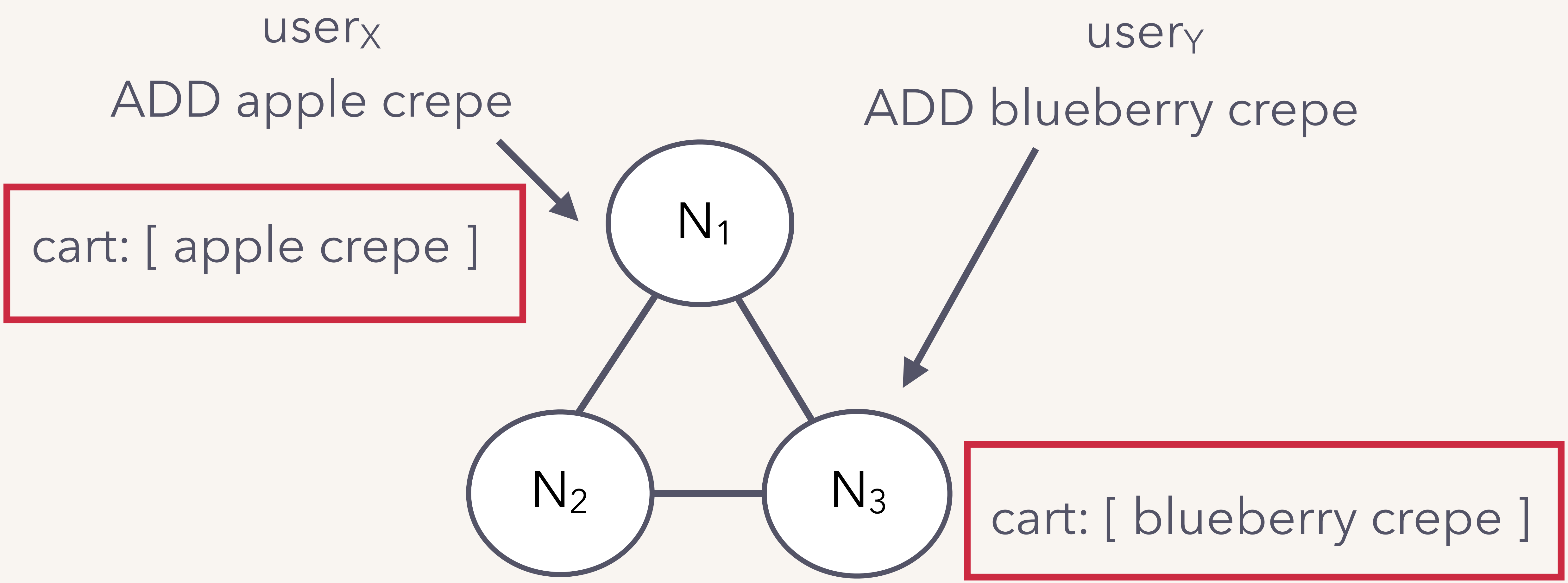
distributed key-value store.
three nodes with master and two replicas.

cart: [ ]



distributed key-value store.
three nodes with three **equal replicas**.
read_quorum = write_quorum = 1.
eventually consistent.

cart: [ ]



…multiple nodes accepting writes
provides availability,
may introduce conflicts.

given we want
concurrent systems,
we need to deal with
data races,
conflict resolution.

riak:

distributed
key-value store

channels:

Go concurrency primitive

stepping back:

similarity,
meta-lessons

riak

a distributed datastore

riak

- **Distributed key-value database**:

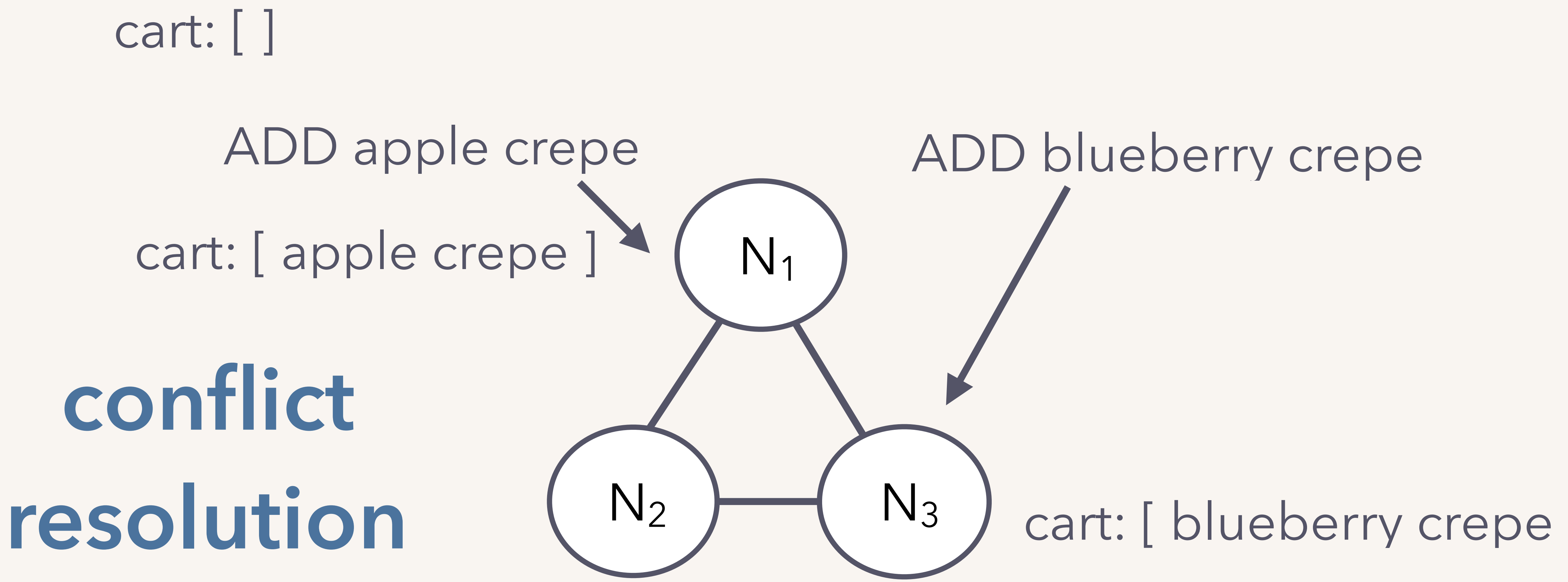
```
// A data item = <key: blob>
{"uuid1234": {"name":"ada"}}
```

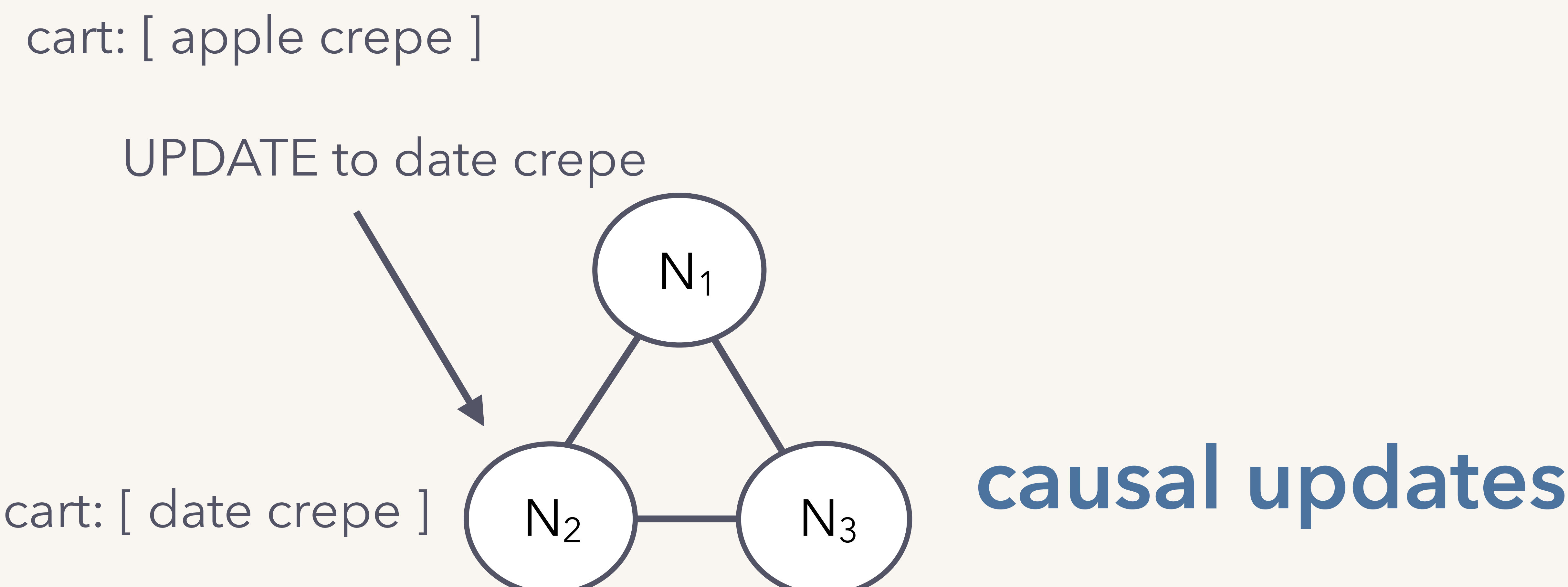
- v1.0 released in 2011.
  Based on Amazon's Dynamo.

- **Eventually consistent**:
  uses optimistic replication i.e.
  replicas can temporarily diverge,
  will eventually converge.

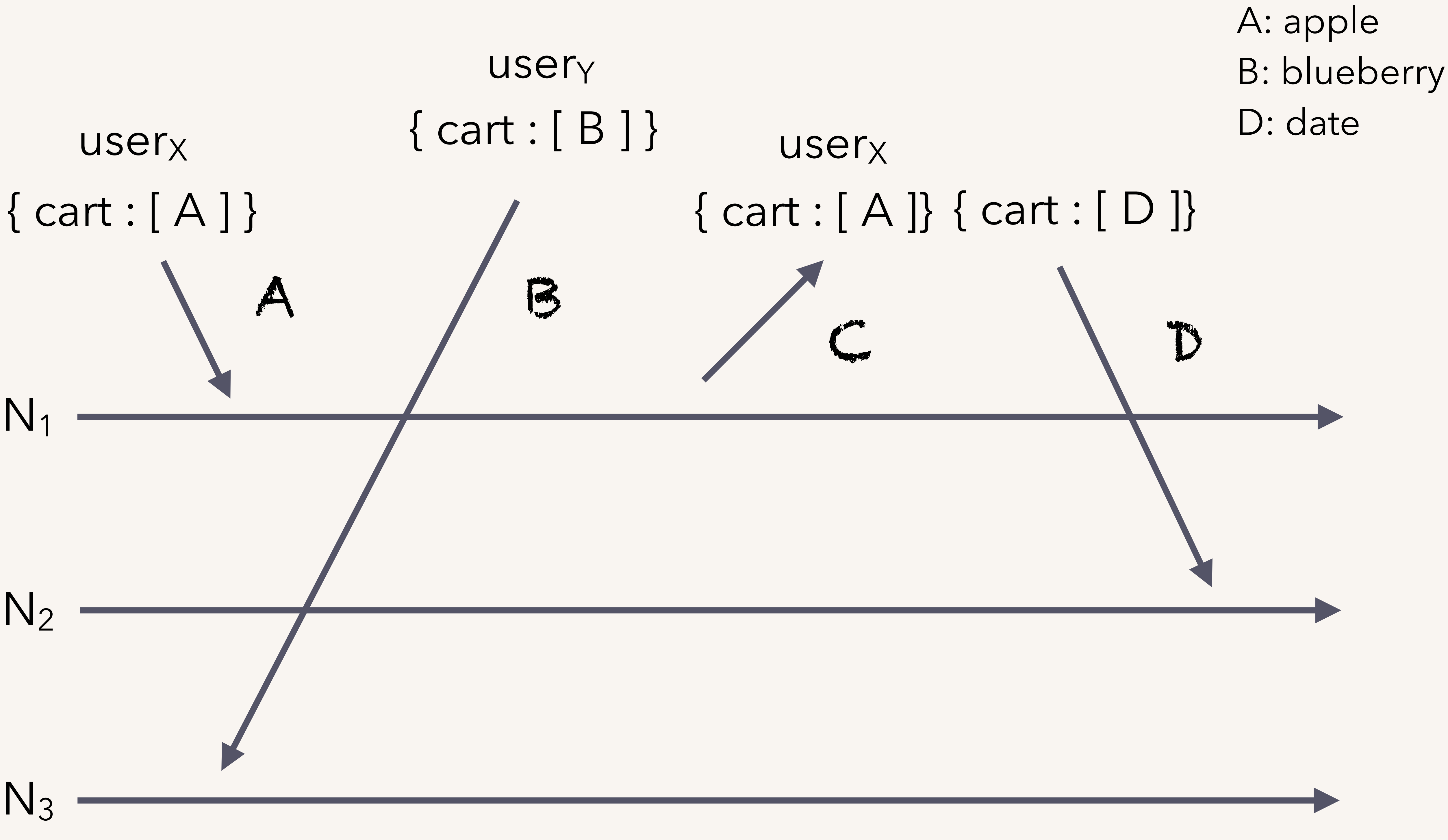
- **Highly available**:
  data partitioned and replicated,
  decentralized,
  sloppy quorum.

AP system
(CAP theorem)

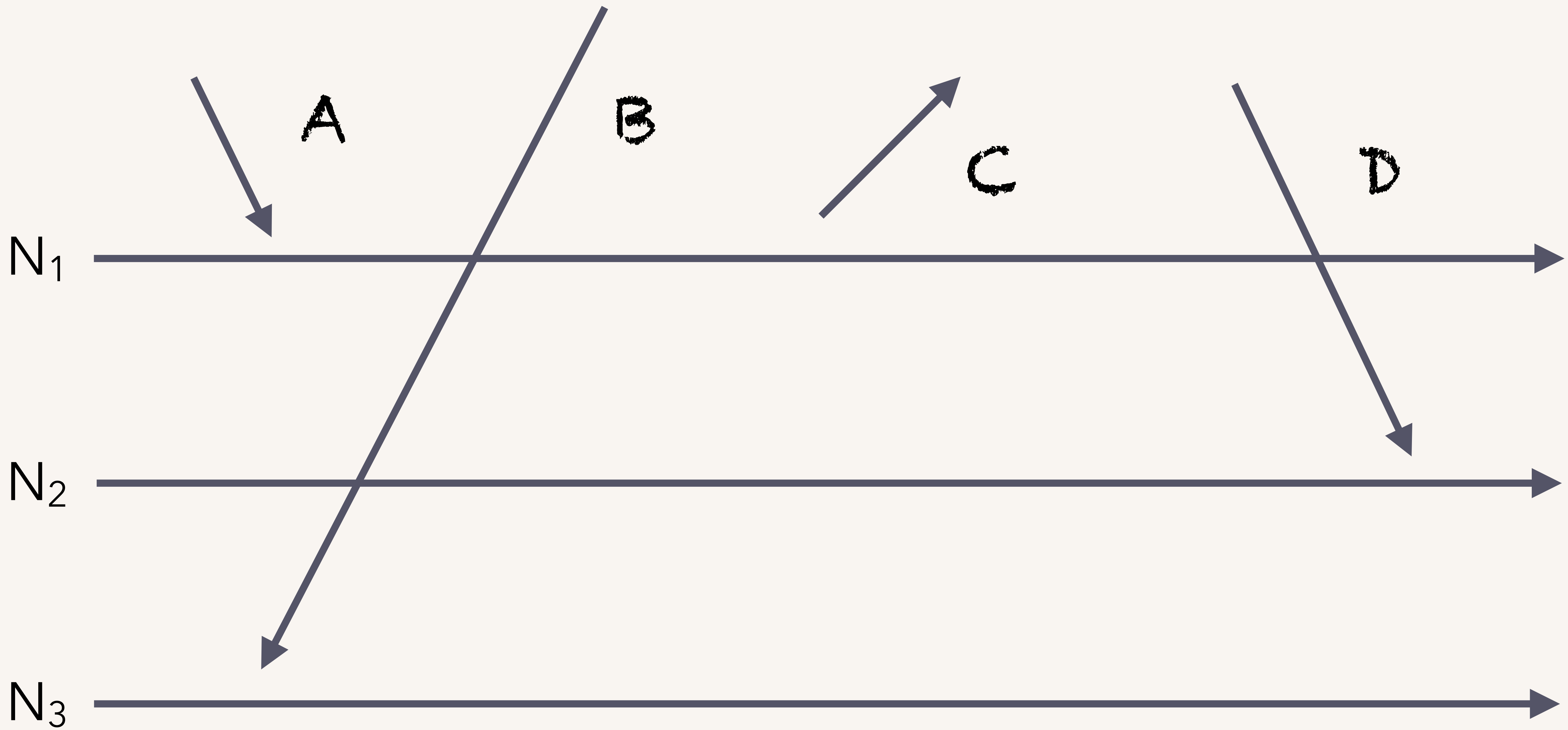

cart: [ ]
ADD apple crepe
ADD blueberry crepe
cart: [apple crepe]
N_1
conflict resolution
N_2
N_3
cart: [blueberry crepe]


cart: [apple crepe]
UPDATE to date crepe
N_1
cart: [date crepe]
N_2
N_3
causal updates

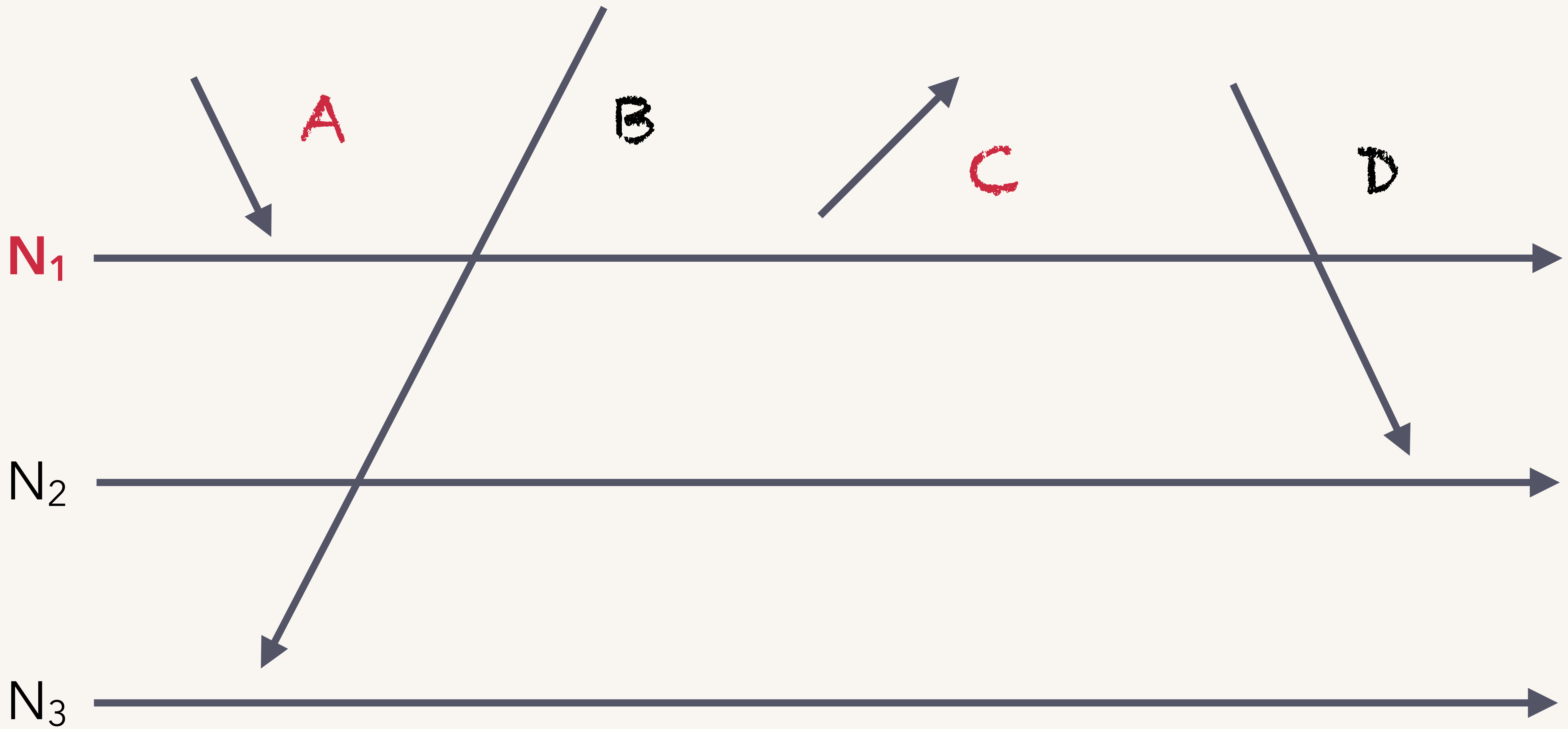
how do we determine
causal vs. concurrent
updates?



concurrent events?

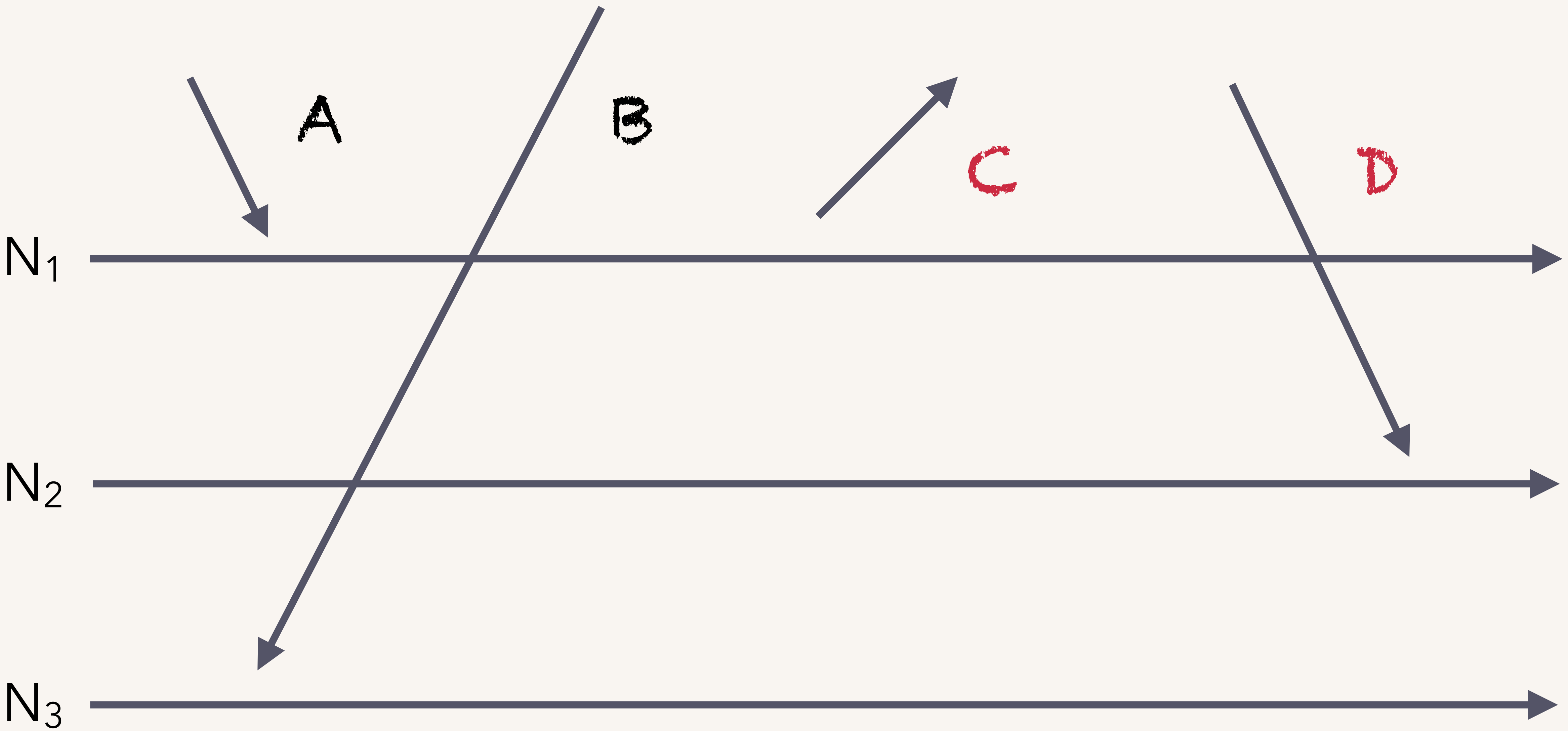


concurrent events?



A, C:

not concurrent – same sequential actor



A, C:

not concurrent – same sequential actor

C, D:

not concurrent – fetch/ update pair

happens-before

orders events <u>across actors</u>.

(threads or nodes)

Formulated in Lamport's *Time, Clocks, and the Ordering of Events* paper in 1978.

establishes causality and concurrency.

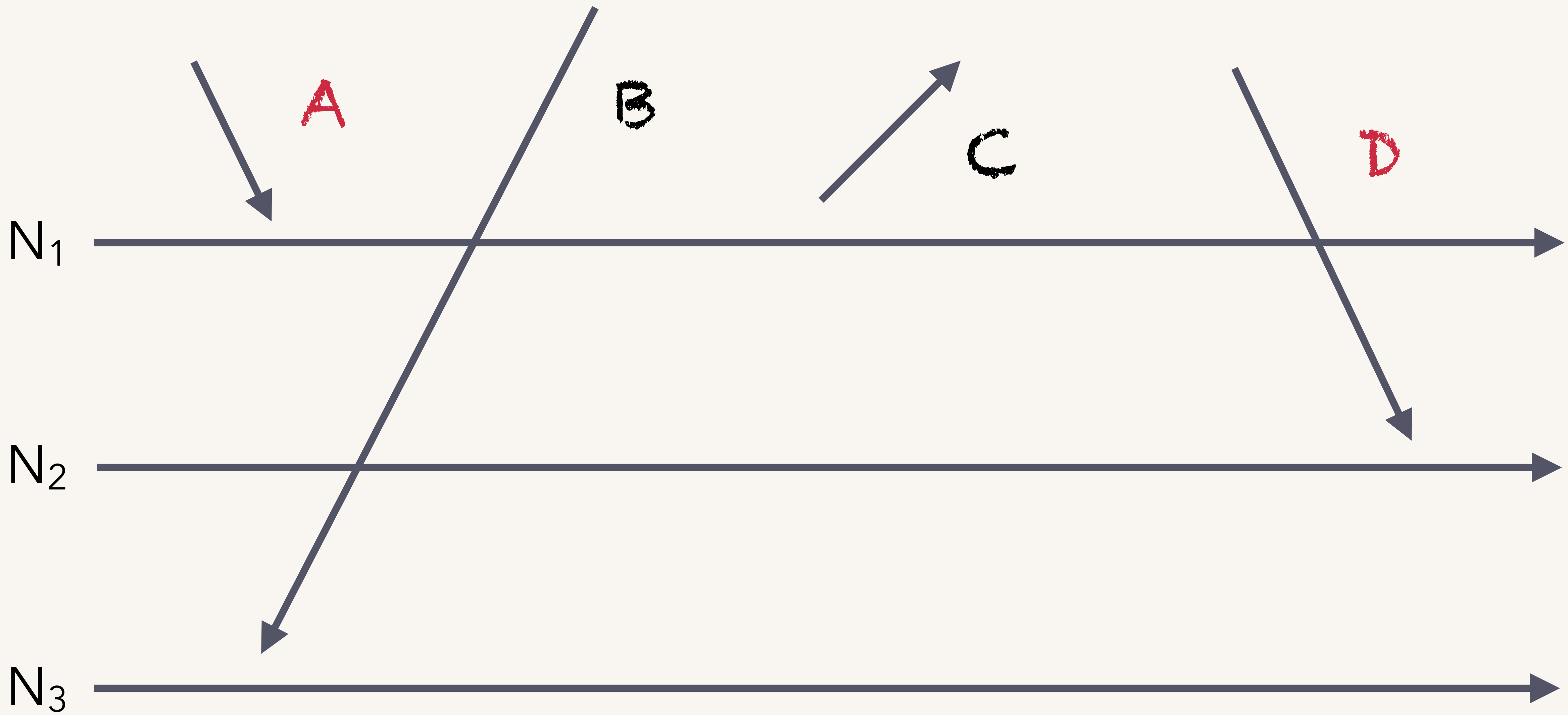
$X < Y$ IF one of:

– same actor
– are a synchronization pair
– $X < E < Y$

IF X <u>not</u> $<$ Y and Y <u>not</u> $<$ X ,

concurrent!

causality

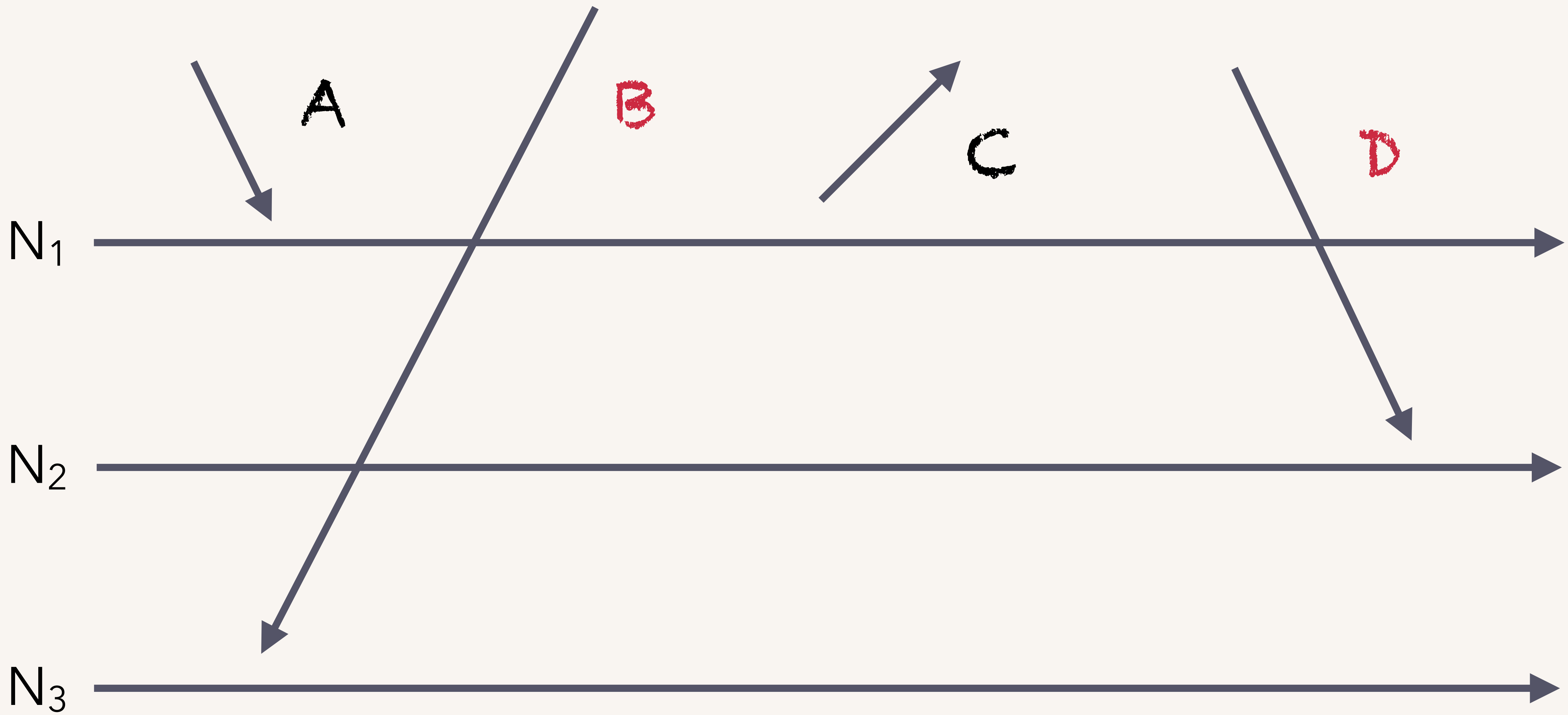


A < C (same actor)

C < D (synchronization pair)

So, A < D (transitivity)

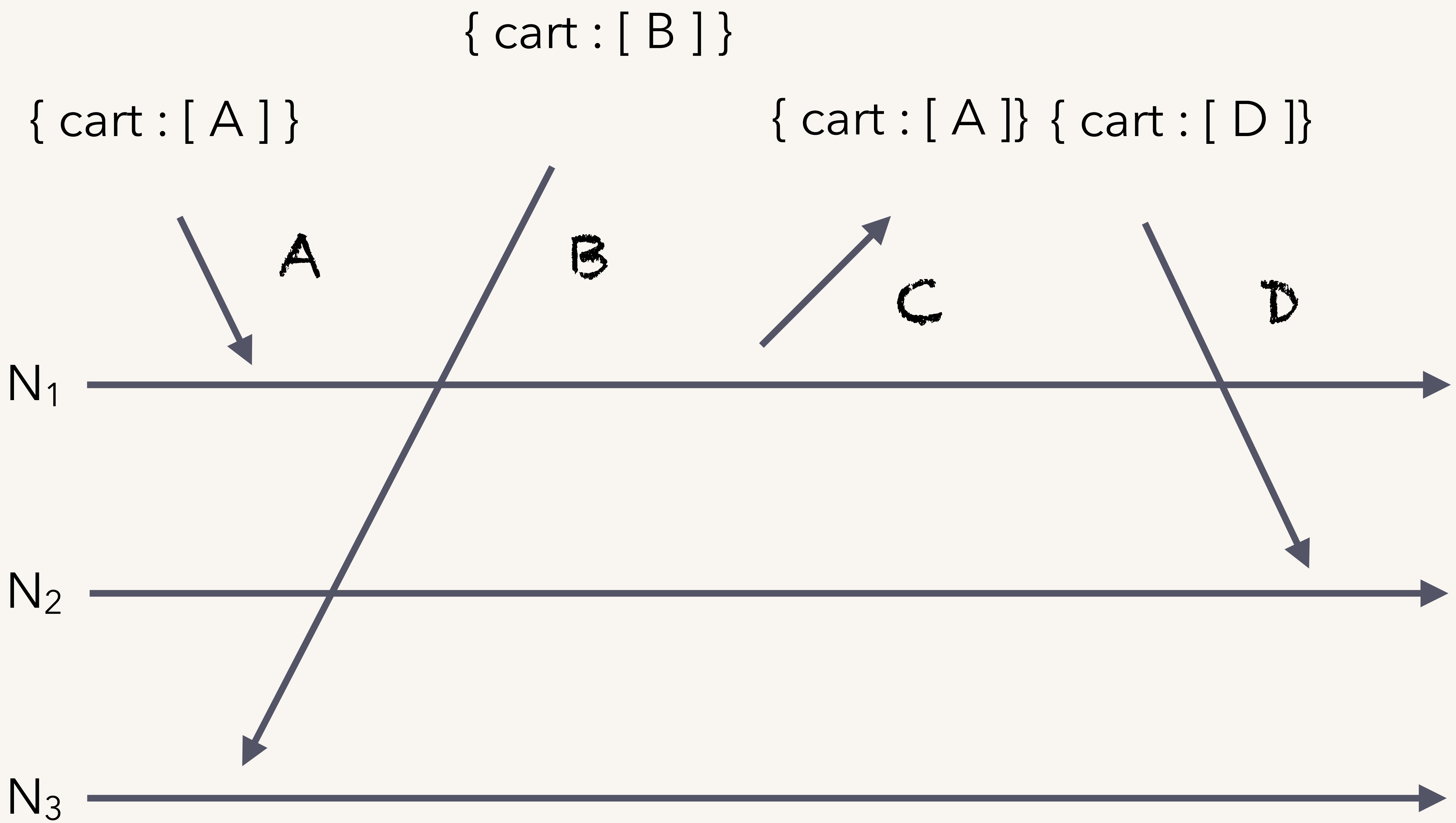
causality and concurrency



…but B ? D

D ? B

So, B, D concurrent!



A < D

D should update A

B, D concurrent

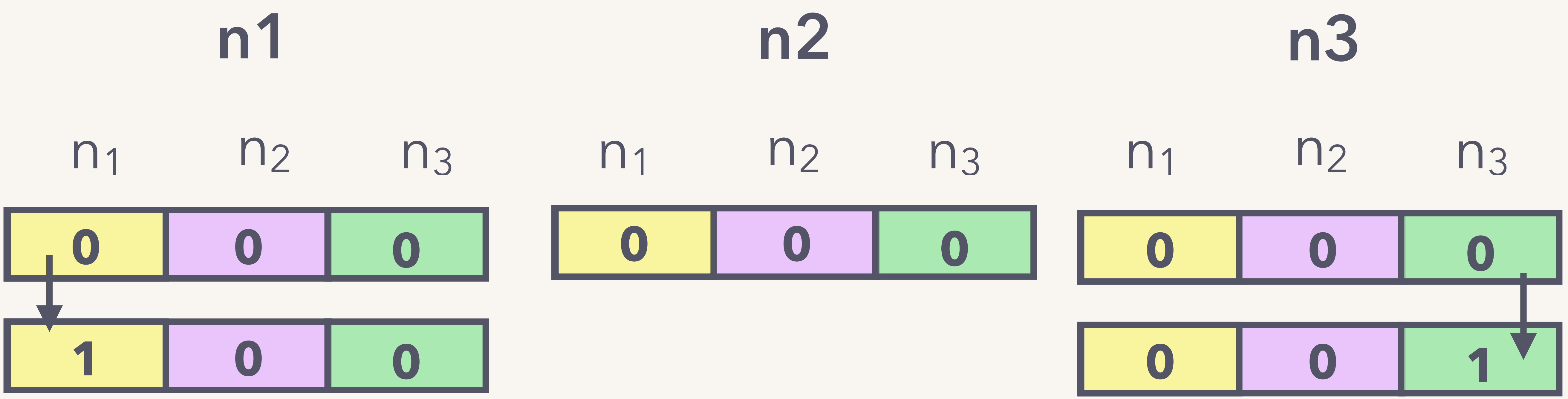
B, D need resolution

how do we implement

happens-before?

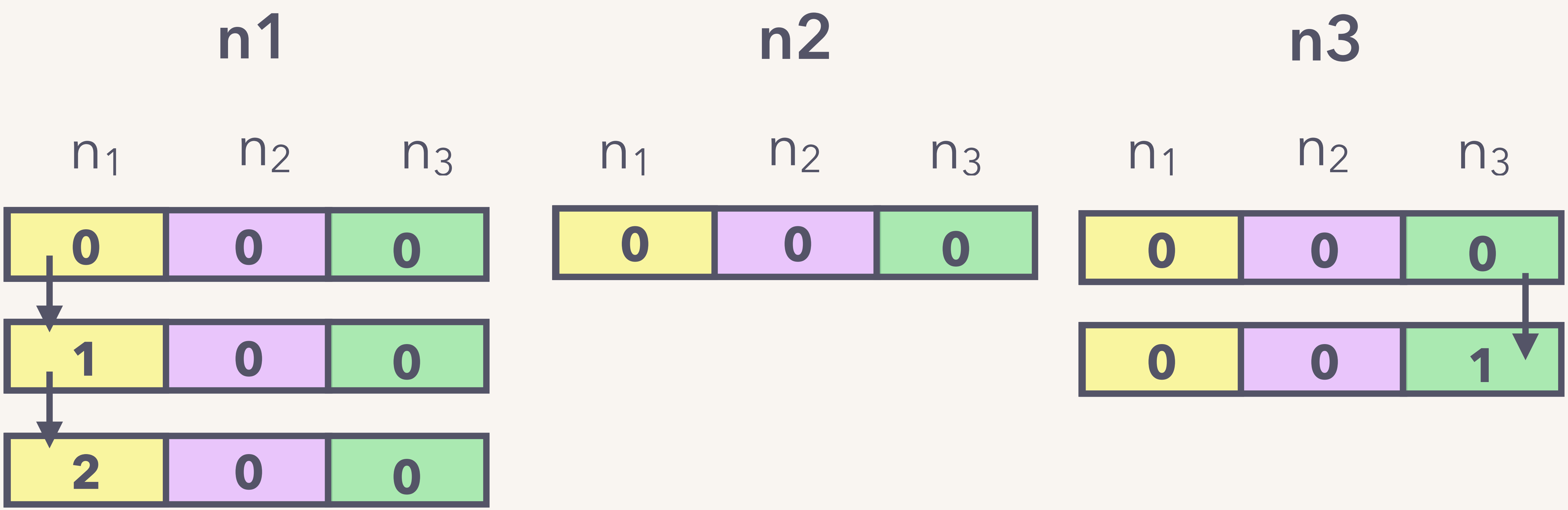
vector clocks

means to establish happens-before edges.



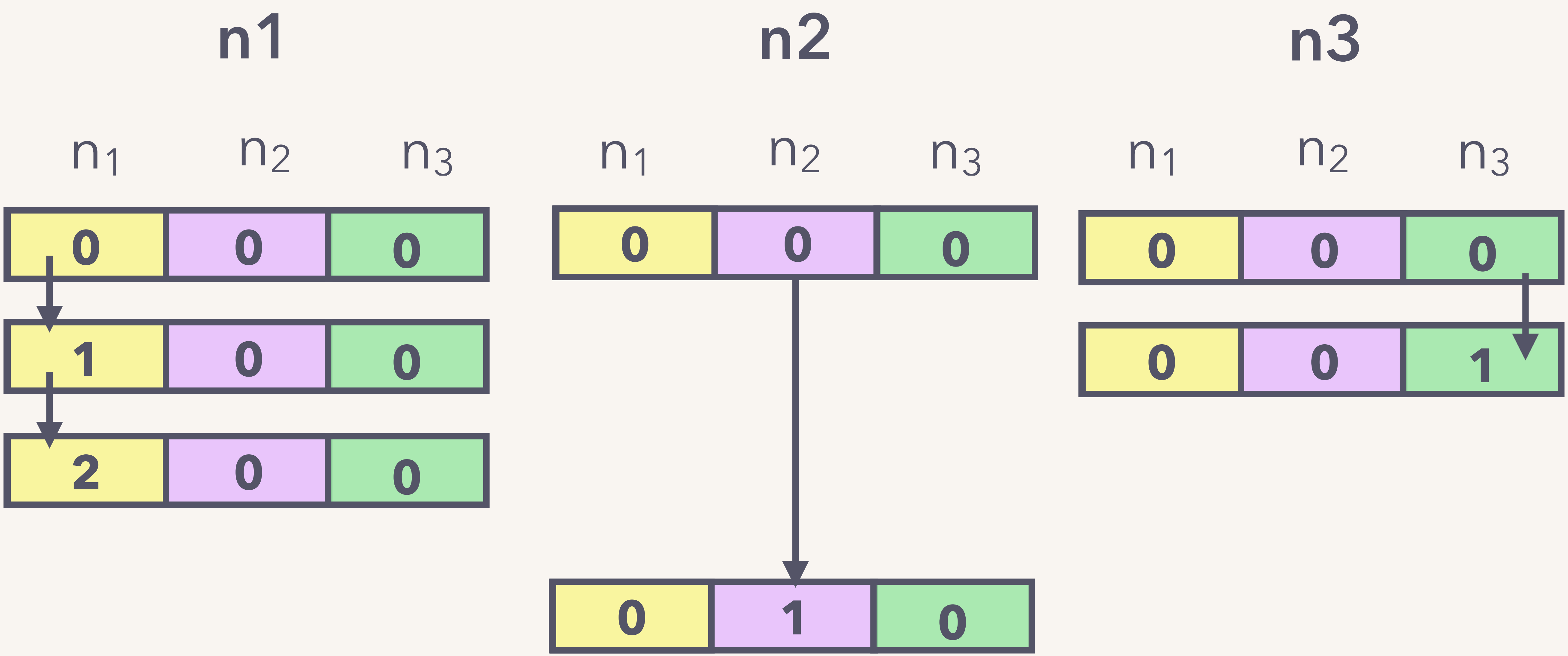
vector clocks

means to establish happens-before edges.



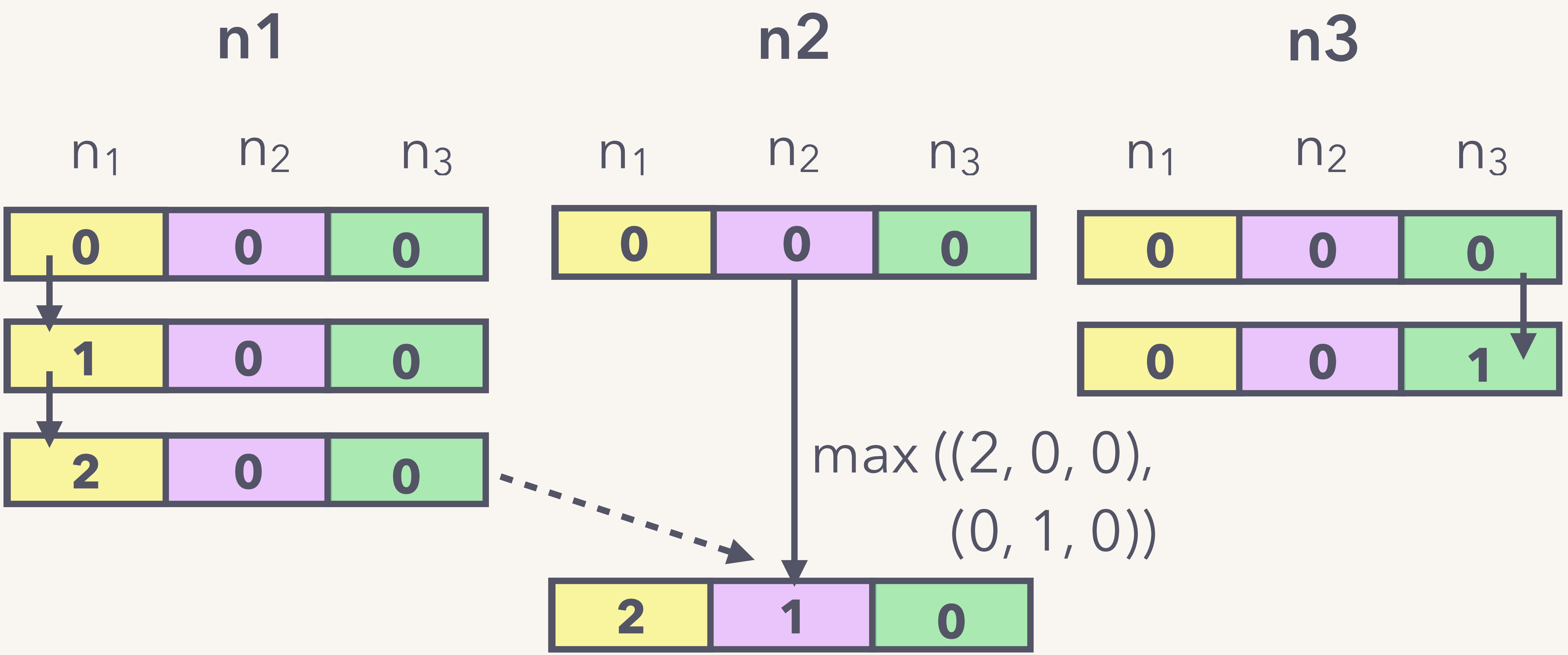
vector clocks

means to establish happens-before edges.



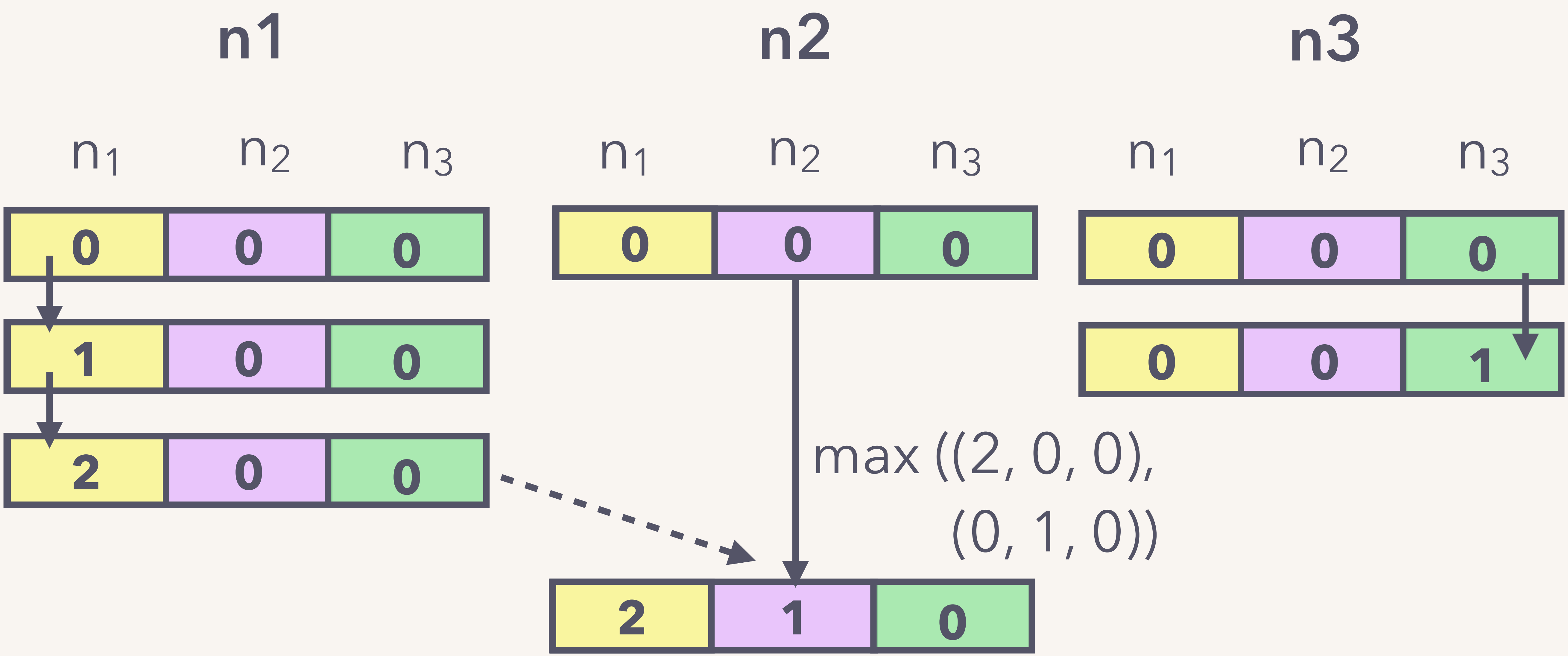
vector clocks

means to establish happens-before edges.

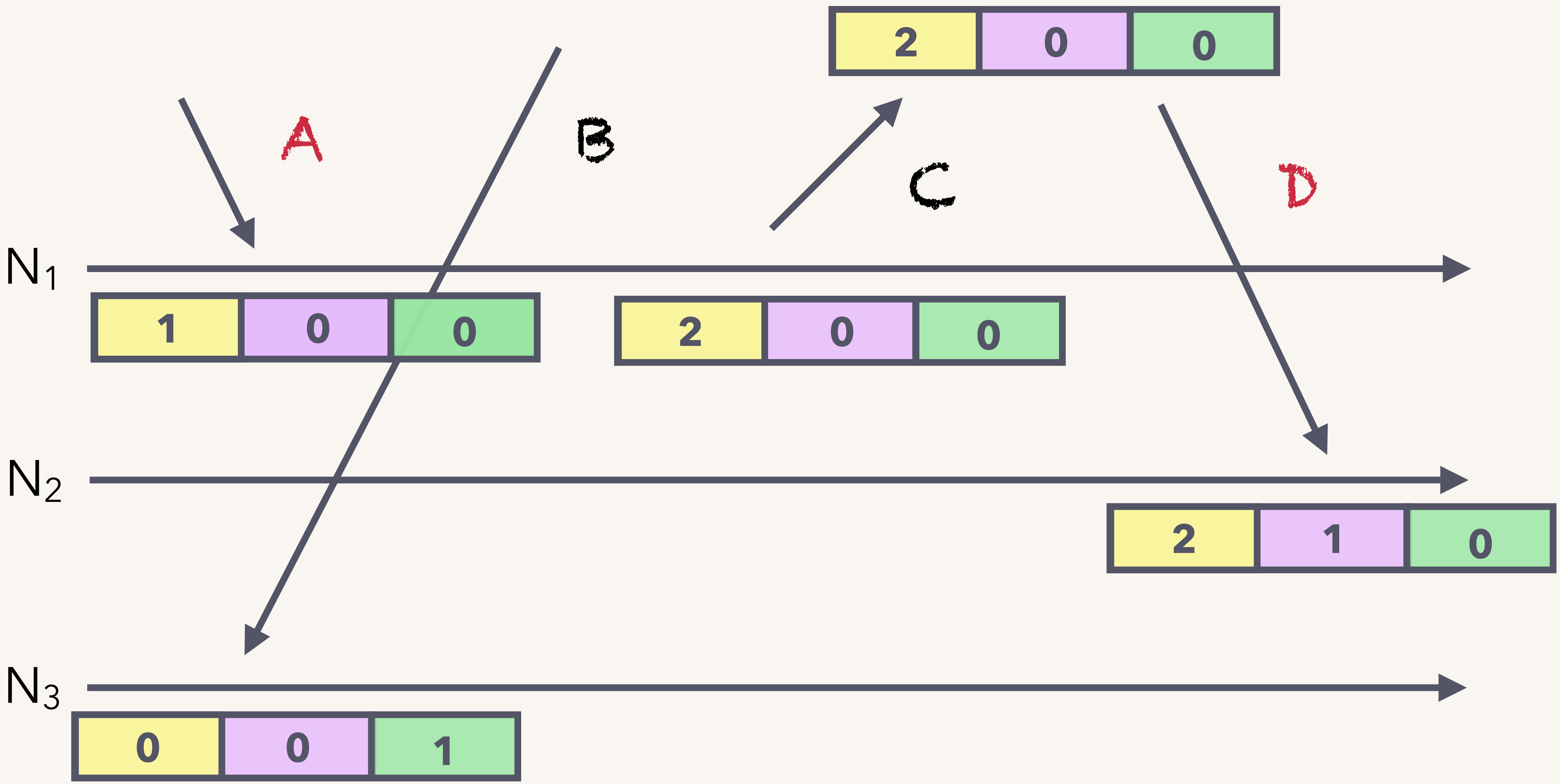


vector clocks

means to establish happens-before edges.



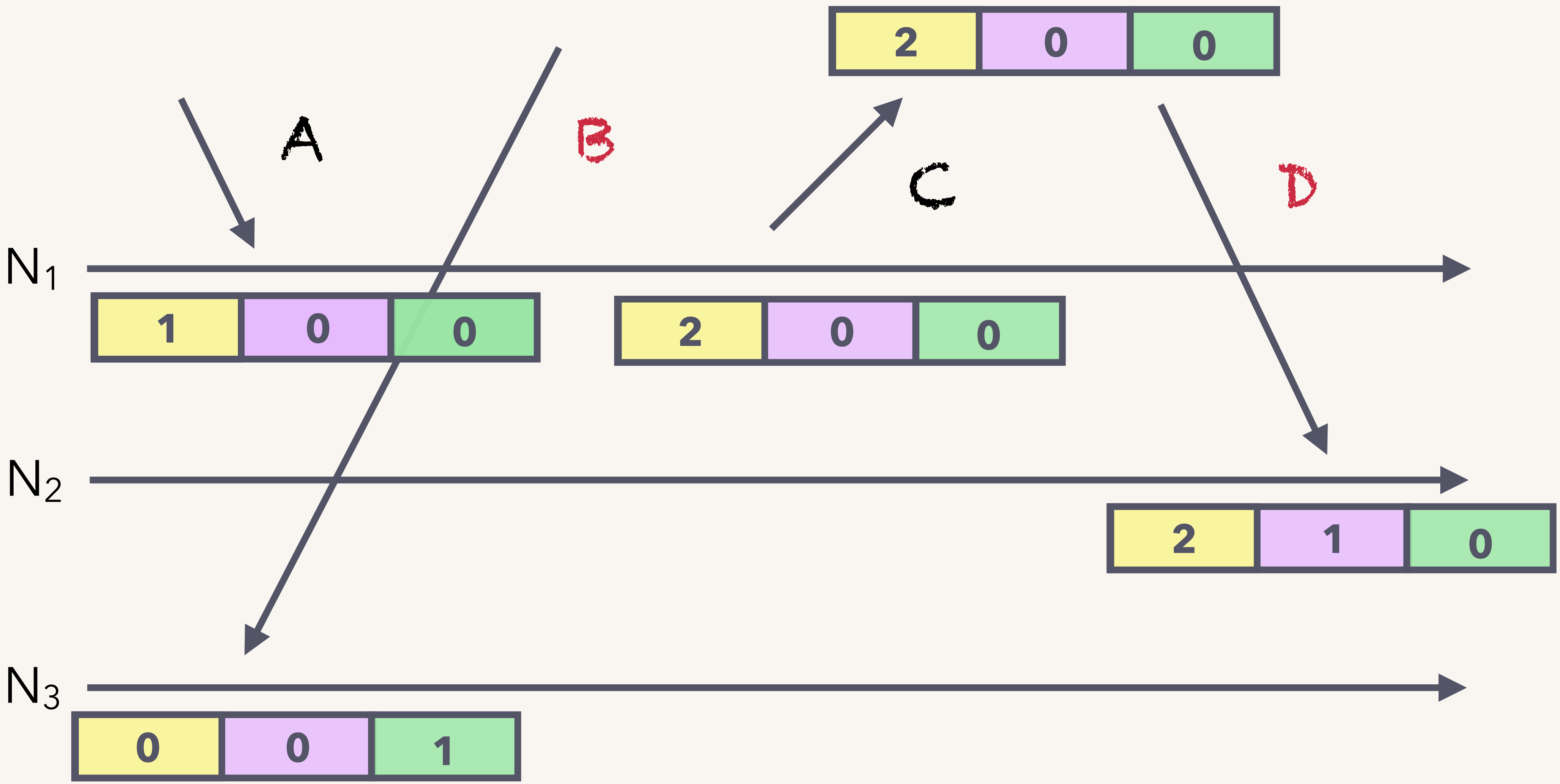
happens-before comparison: X ≺ Y iff VCx < VCy



VC at A: | 1 | 0 | 0 |

VC at D: | 2 | 1 | 0 |

So, A < D



VC at B:	0	0	1
VC at D:	2	1	0

So, B, D concurrent

causality tracking in riak

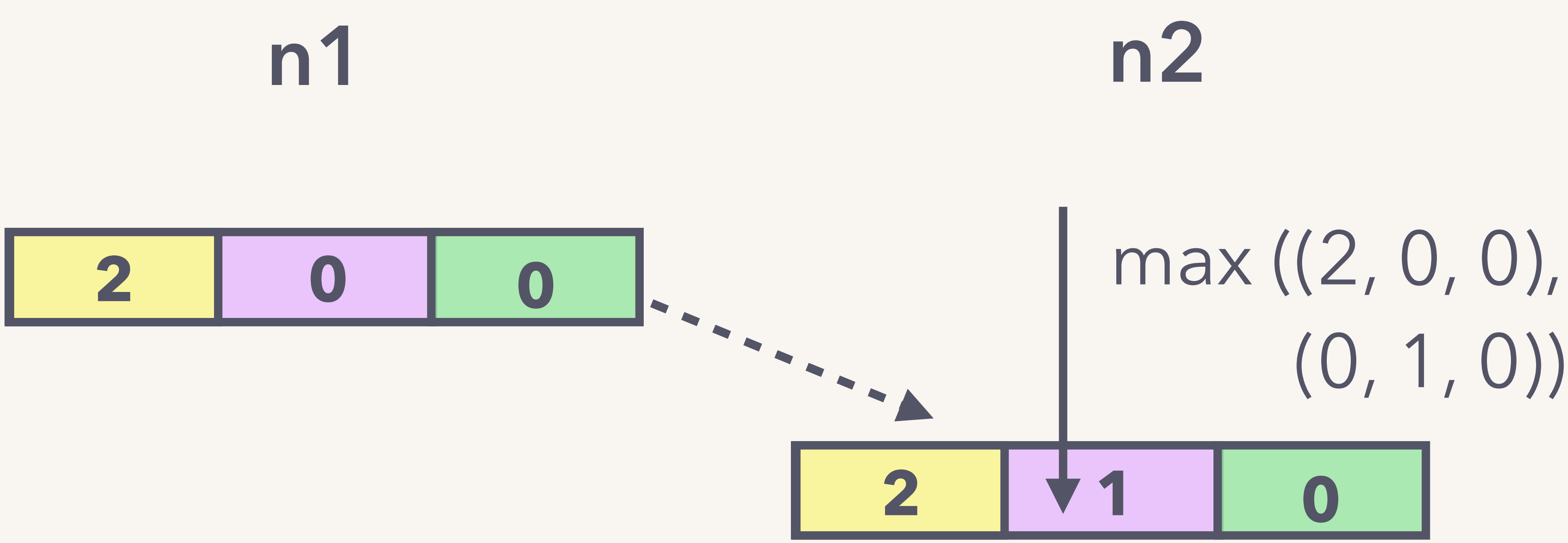
Riak stores a vector clock with each version of the data.

a more precise form,

"dotted version vector"

GET, PUT operations on a key pass around a casual context object,

that contains the vector clocks.



causality tracking in riak

Riak stores a vector clock with each version of the data.

a more precise form,

"dotted version vector"

GET, PUT operations on a key pass around a casual context object,

that contains the vector clocks.

Therefore, able to detect conflicts.

...what about resolving those conflicts?

conflict resolution in riak

Behavior is configurable.

Assuming vector clock analysis enabled:

- **last-write-wins**

  i.e. version with higher timestamp picked.

- **merge, iff the underlying data type is a CRDT**

- **return conflicting versions to application**

  riak stores "siblings" or conflicting versions, returned to application for resolution.

return conflicting versions to application:

Riak stores both versions

B: { cart: [“blueberry crepe”] }	0	0	1
D: { cart: [“date crepe”] }	2	1	0

next op returns both to application

<u>application</u> must resolve conflict

{ cart: [“blueberry crepe”, “date crepe”] }

which creates a causal update

{ cart: [“blueberry crepe”, “date crepe”] }	2	1	1

…what about resolving those conflicts?

doesn't

(default behavior).

instead, **exposes happens-before graph** to the application for conflict resolution.

riak:

uses

vector clocks

to track causality and conflicts.

exposes

happens-before graph

to the user for conflict resolution.

channels

Go concurrency primitive

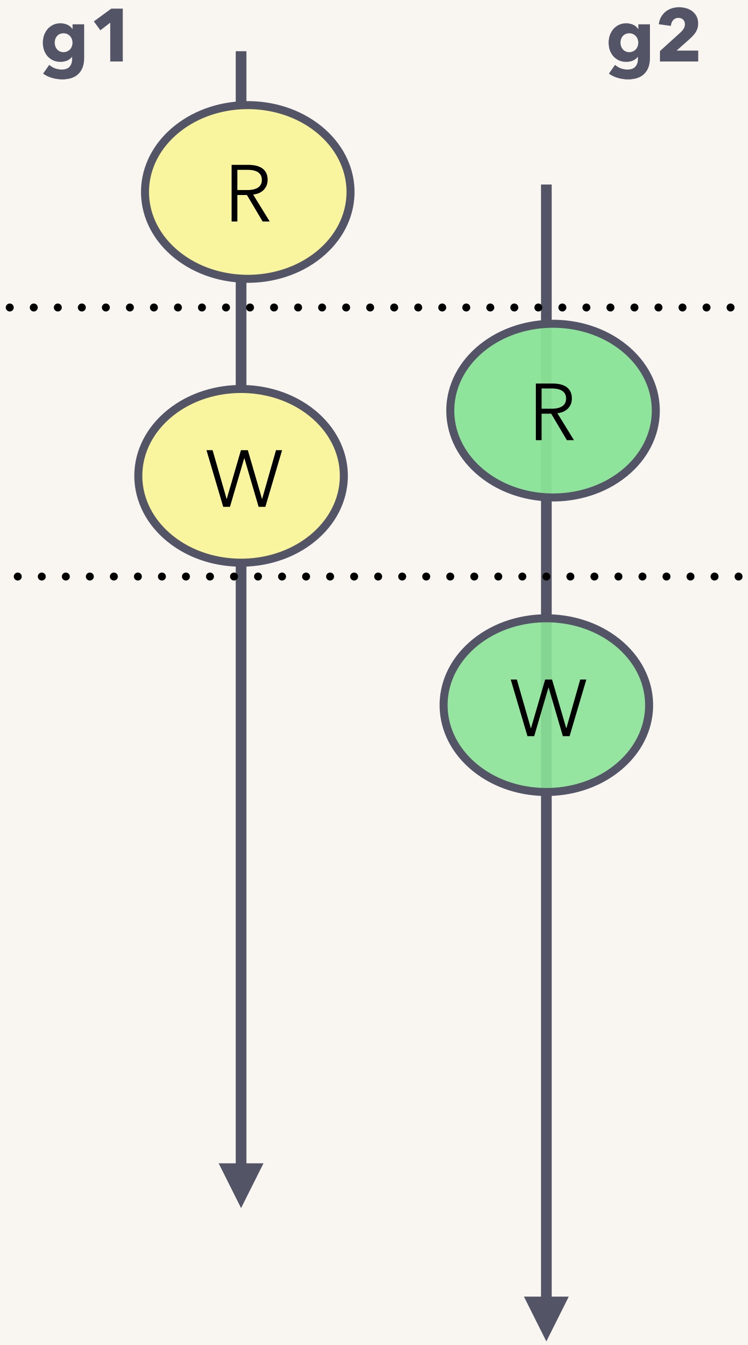
multiple threads:

```go
// Shared variable
var tasks []Task

func worker() {
   for len(tasks) > 0 {
      task := dequeue(tasks)
      process(task)
    }
}

func main() {
   // Spawn fixed-pool of worker threads.
   startWorkers(3, worker)

   // Populate task queue.
   for _, t := range hellaTasks {
      tasks = append(tasks, t)
   }
}
```



data race

"when two+ threads <u>concurrently</u> access a <u>shared memory location</u>, at least one access is a <u>write</u>."

memory model

specifies when an event **happens before** another.

X	x = 1
Y	print(x)

X < Y IF one of:

– same thread
– are a synchronization pair → unlock/ lock on a mutex, send / recv on a channel, spawn/ first event of a thread. etc.
– X < E < Y

IF X <u>not</u> < Y and Y <u>not</u> < X , concurrent!

goroutines

The unit of concurrent execution: goroutines

- user-space threads
- use as you would threads

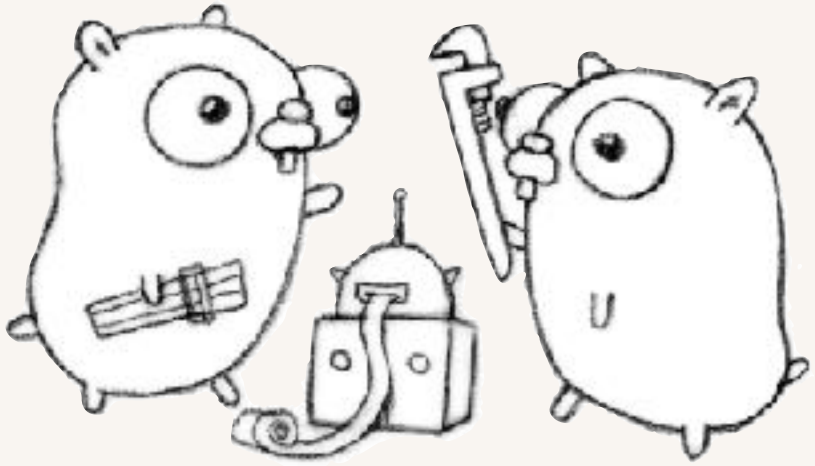
  ```
  > go handle_request(r)
  ```

- Go memory model specified in terms of goroutines
  - within a goroutine: reads + writes are ordered
  - with multiple goroutines: shared data must be synchronized…else data races!

synchronization

The synchronization primitives are:

- mutexes, conditional vars, …
  - > `import “sync”`
  - > `mu.Lock()`
- atomics
  - > `import “sync/ atomic"`
  - > `atomic.AddUint64(&myInt, 1)`
- channels



channels

"Do not communicate by sharing memory;
instead, share memory by communicating."

- standard type in Go – `chan`
  safe for concurrent use.

- mechanism for goroutines to communicate, and synchronize.

- Conceptually similar to Unix pipes:

```
> ch := make(chan int) // Initialize
> go func() { ch <- 1 } () // Send
> <-ch // Receive, blocks until sent.
```

```go
// Shared variable
var tasks []Task

func worker() {
   for len(tasks) > 0 {
      task := dequeue(tasks)
      process(task)
    }
}

func main() {
   // Spawn fixed-pool of workers.
   startWorkers(3, worker)

   // Populate task queue.
   for _, t := range hellaTasks {
      tasks = append(tasks, t)
   }
}
```

want:

worker:

* get a task.
* process it.
* repeat.

main:

* give tasks to workers.

```go
var taskCh = make(chan Task, n)

func main() {
   // Spawn fixed-pool of workers.
   startWorkers(3, worker)

   // Populate task queue.
   for _, t := range hellaTasks {
      taskCh <- t
   }




}
```

```go
func worker() {
   for {
      // Get a task.
      t := <-taskCh
      process(t)


   }
}
```

mutex?

mu

mu

mu

```
// Shared variable
var tasks []Task

func worker() {
   for len(tasks) > 0 {
      task := dequeue(tasks)
      process(task)
    }
}

func main() {
   // Spawn fixed-pool of workers.
   startWorkers(3, worker)

   // Populate task queue.
   for _, t := range hellaTasks {
      tasks = append(tasks, t)
   }
}
```

…but workers can exit early.

want:

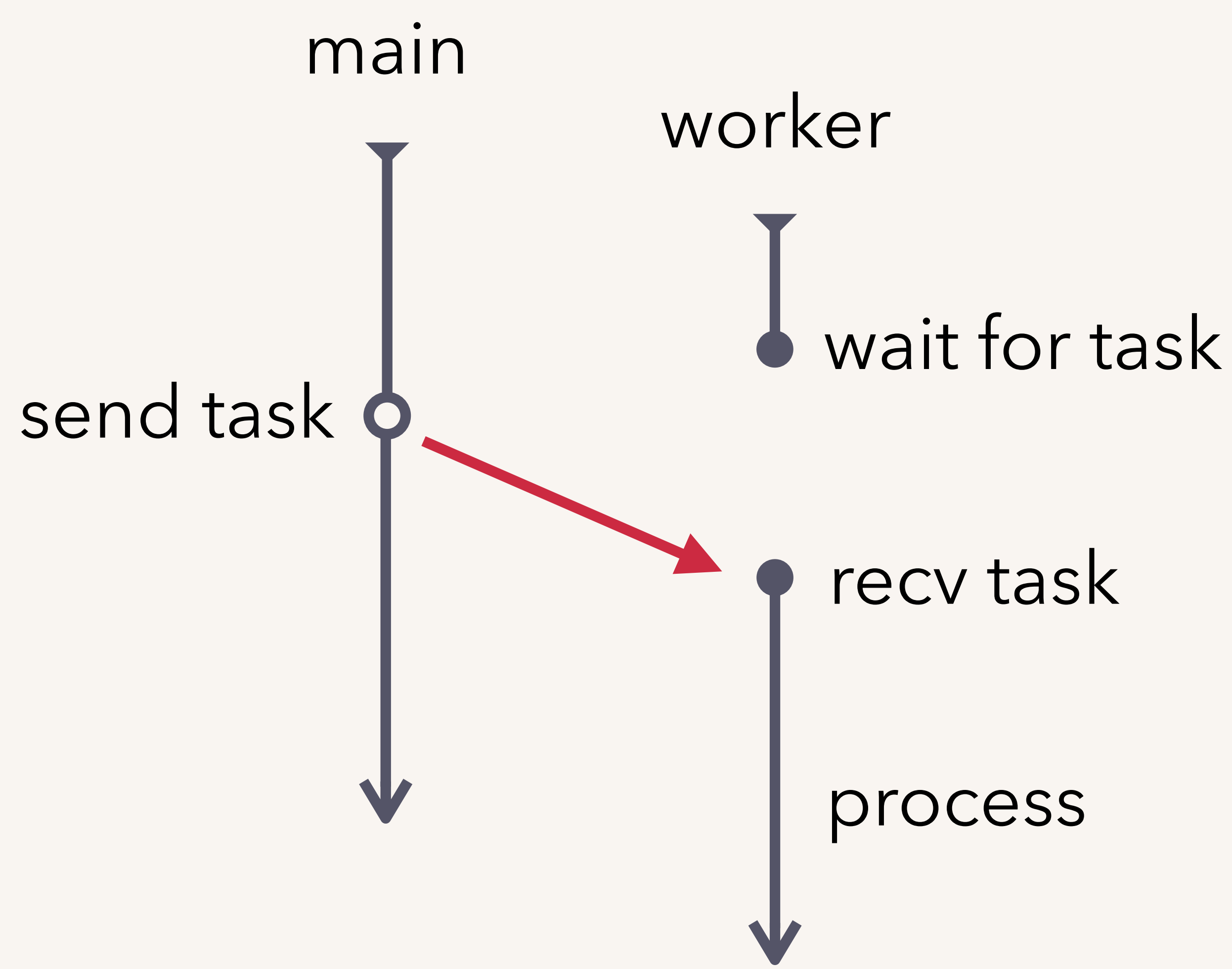
main:

*   send tasks

worker:

*   wait for task
*   process it
*   repeat

channel semantics (as used):



send task to happen before worker runs.

…channels allow us to express happens-before constraints.

channels:

allow, and force, the user
to express
happens-before
constraints.

stepping back…

similarities

surface happens-before to the user

riak:
distributed
key-value store

channels:
Go
concurrency primitive

first principle:
happens-before

meta-lessons

new technologies
cleverly decompose
into
old ideas

the “right” boundaries
for abstractions
are flexible.

$\prec$

happens-before

riak

channels

@kavya719

https://speakerdeck.com/kavya719/what-came-first

riak: a note (or two)...

nodes in Riak:

> virtual nodes ("vnodes")

> key-space partitioning by consistent hashing,1 vnode per partition.

> sequential because Erlang processes, use message queues.

replicas:

> N, R, W, etc. configurable by key.

> on network partition, defaults to sloppy quorum w/ hinted-handoff.

conflict-resolution:

> by read-repair, active anti-entropy.

riak: dotted version vectors

problem with standard vector clocks: false concurrency.

userX: PUT “cart”:”A”, {} –> (1, 0); “A”

userY: PUT “cart”:”B”, {} –> (2, 0); [“A”, “B”]

userX: PUT “cart”:”C”, {(1, 0); “A”} –> (1, 0) !< (2, 0) –> (3, 0); [“A”, “B”, “C”]

This is false concurrency; leads to “sibling explosion”.

dotted version vectors

fine-grained mechanism to detect causal updates.

decompose each vector clock into its set of discrete events, so:

userX: PUT “cart”:”A”, {} –> (1, 0); “A”

userY: PUT “cart”:”B”, {} –> (2, 0); [(1, 0)->”A”, (2, 0)->”B”]

userX: PUT “cart”:”C”, {} –> (3, 0); [(2, 0)->”B”, (3, 0)->”C”]

riak: CRDTs

Conflict-free / Convergent / Commutative Replicated Data Type

> data structure with property:
replicas can be updated concurrently without coordination, and it's mathematically possible to always resolve conflicts.

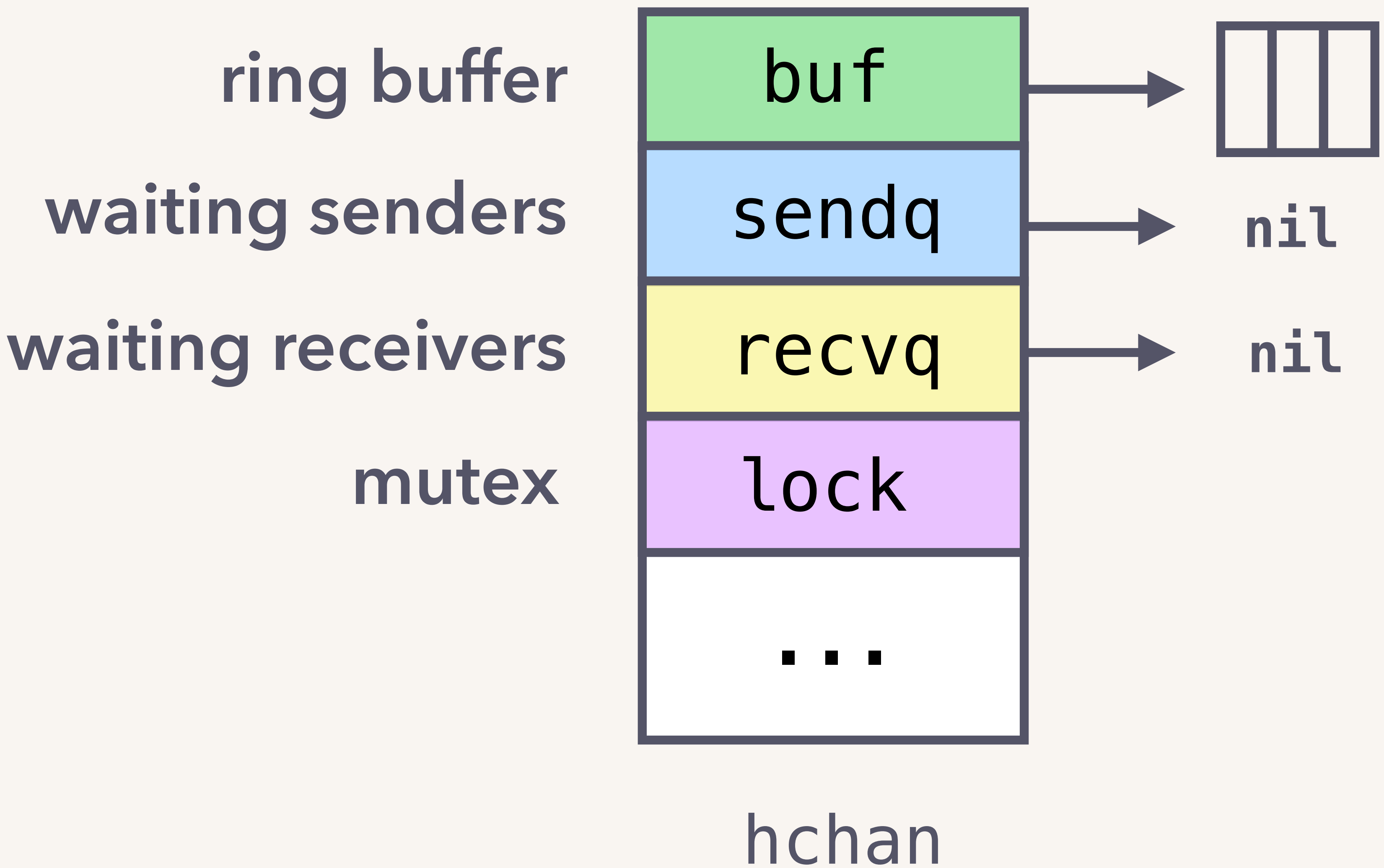
> two types: op-based (commutative) and state-based (convergent).

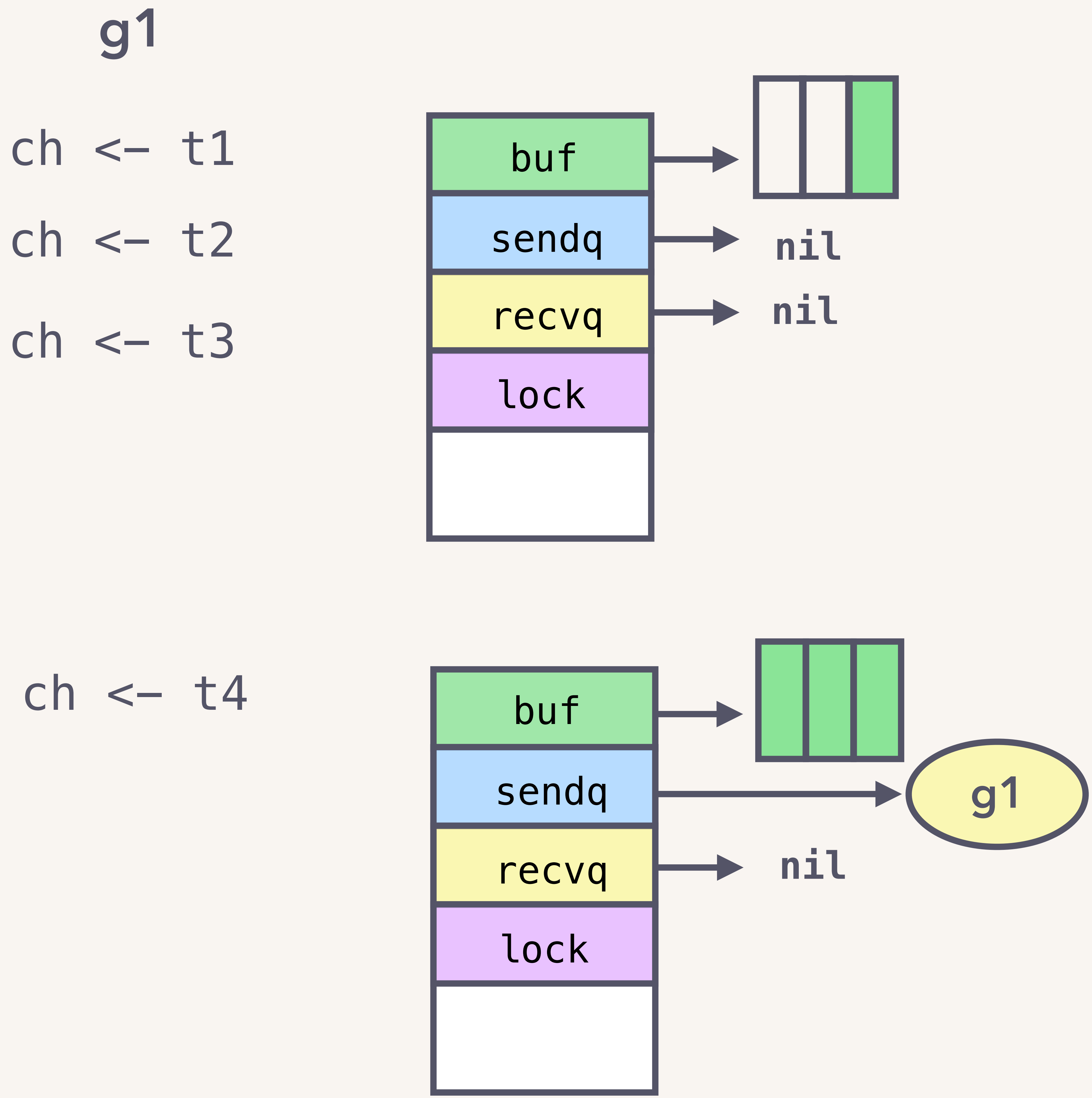
> examples: G-Set (Grow-Only Set), G-Counter, PN-Counter

> Riak DT is state-based CRDTs.

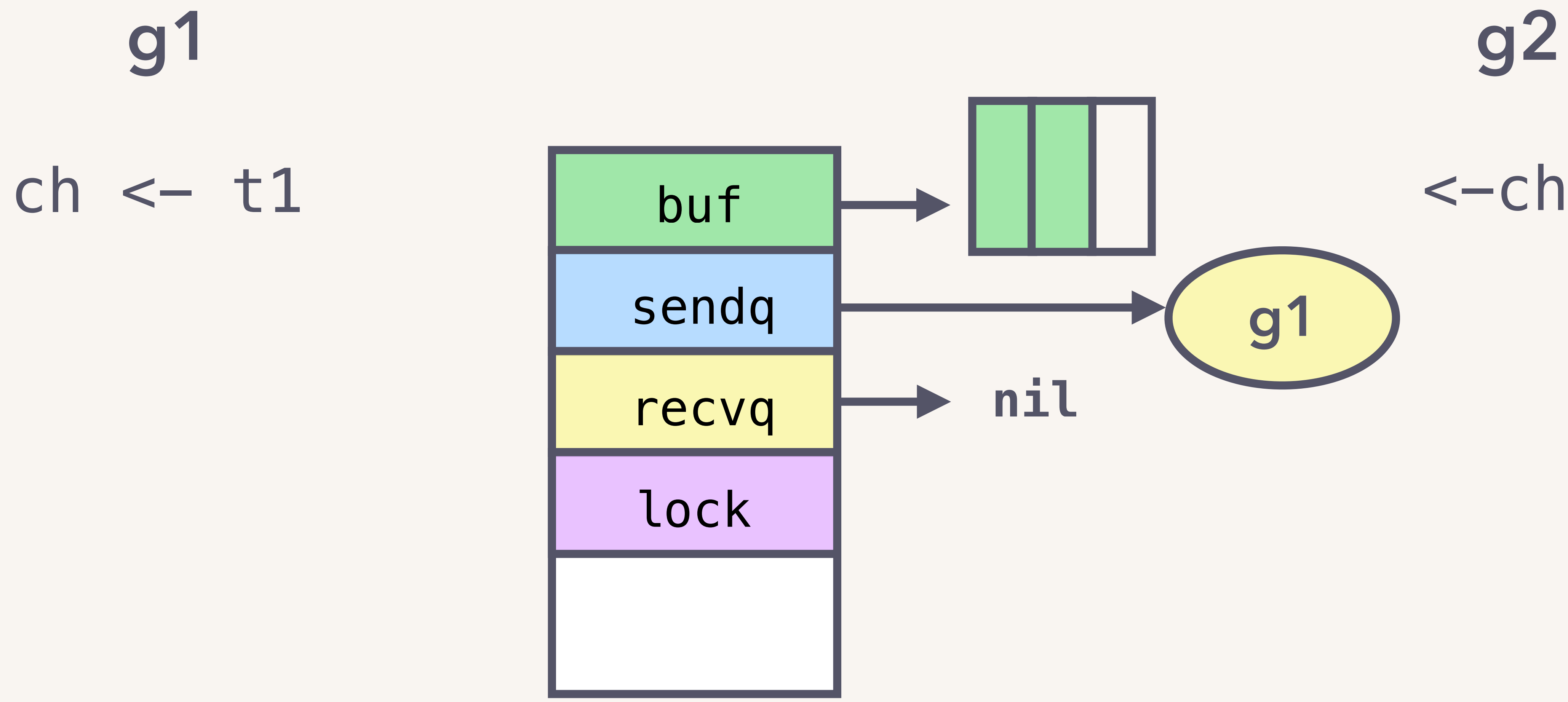
channels: implementation

```
ch := make(chan int, 3)
```

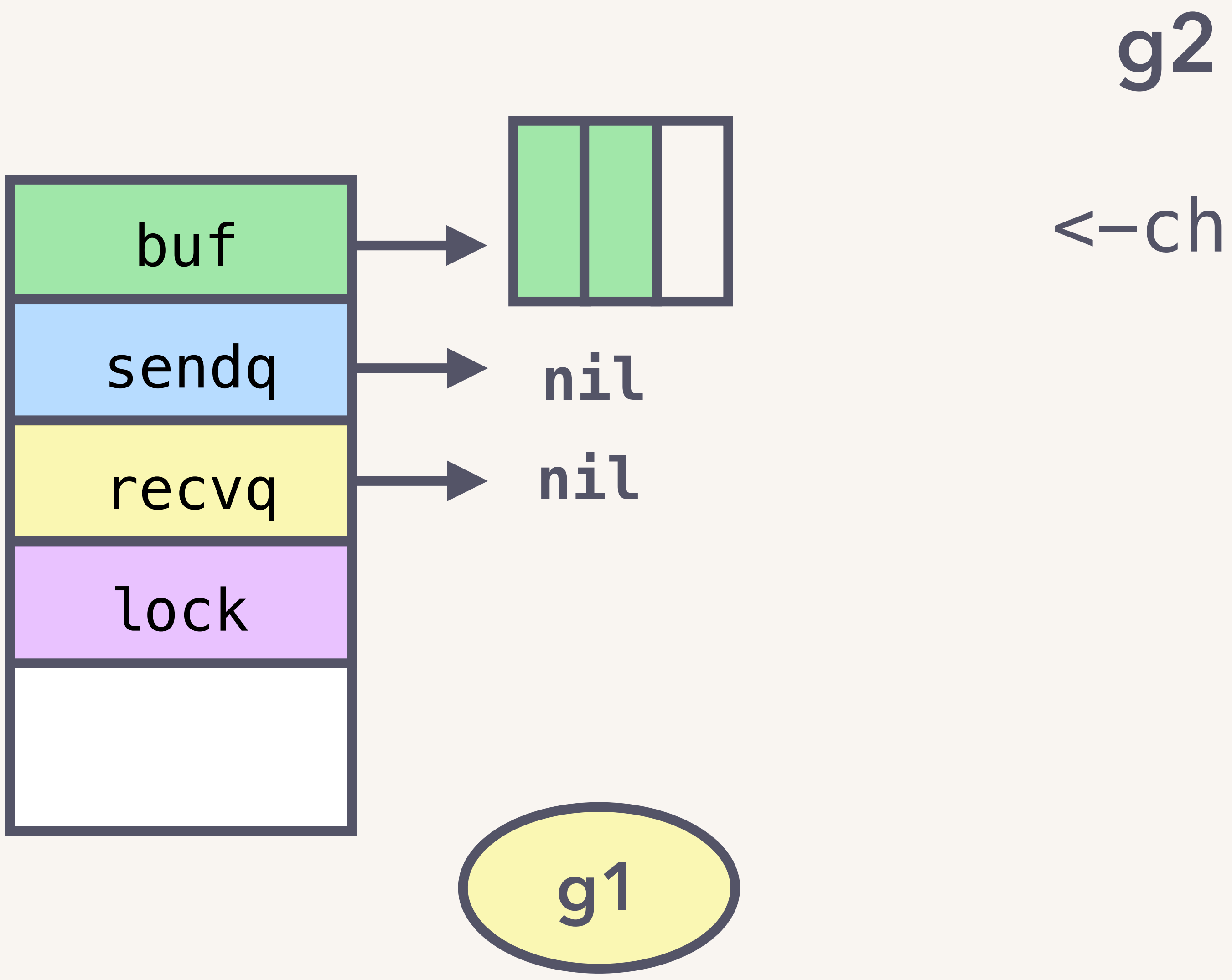


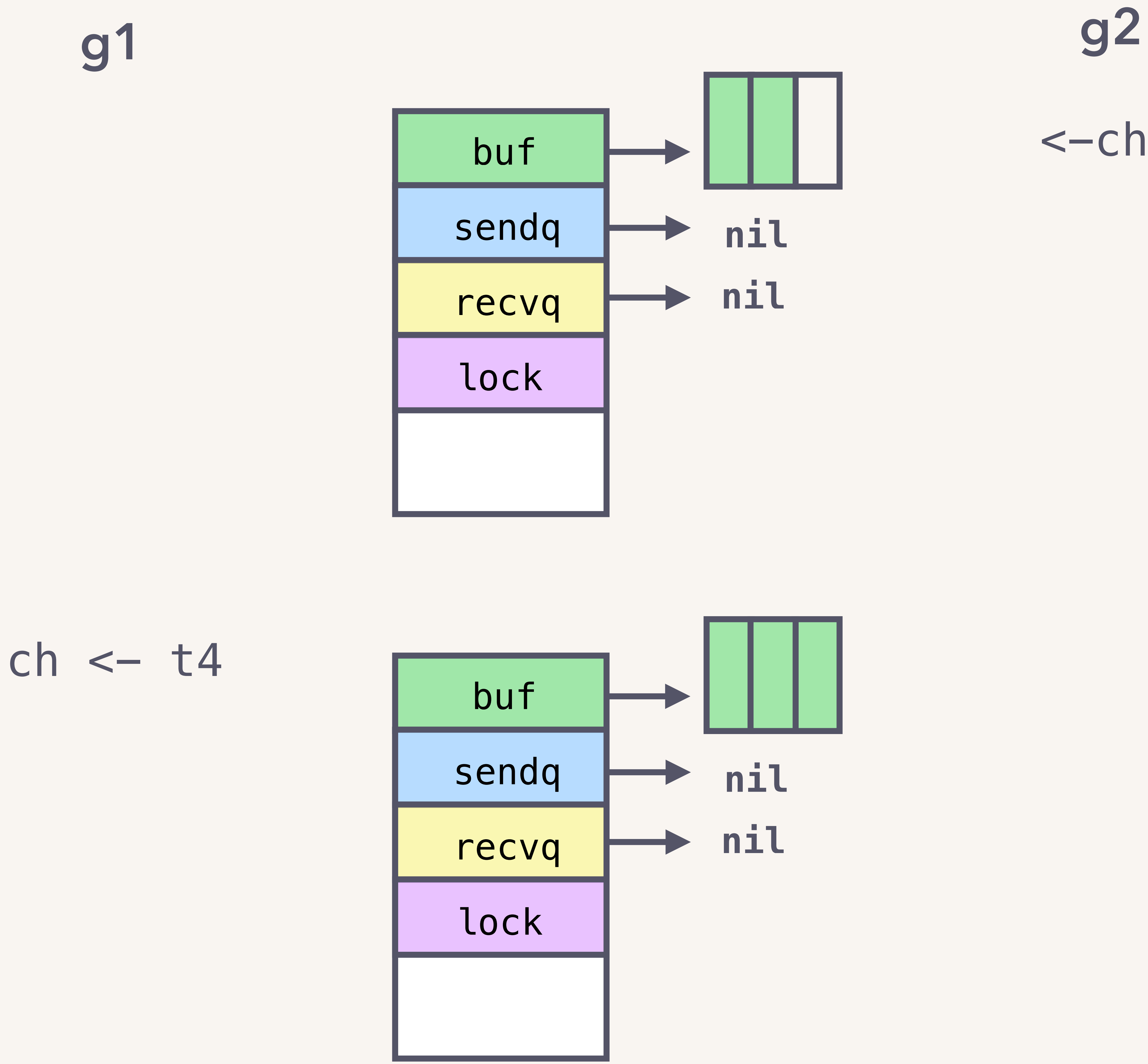


g1
ch <- t1
ch <- t2
ch <- t3
buf
sendq
recvq
lock
nil
nil
ch <- t4
buf
sendq
recvq
lock
g1
nil



g1
g2
ch <- t1
<-ch
buf
sendq
recvq
lock
g1
nil


g2
buf
sendq
recvq
lock
nil
nil
<-ch
g1


g1
g2
<-ch
buf
sendq
nil
recvq
nil
lock
ch <- t4
buf
sendq
nil
recvq
nil
lock

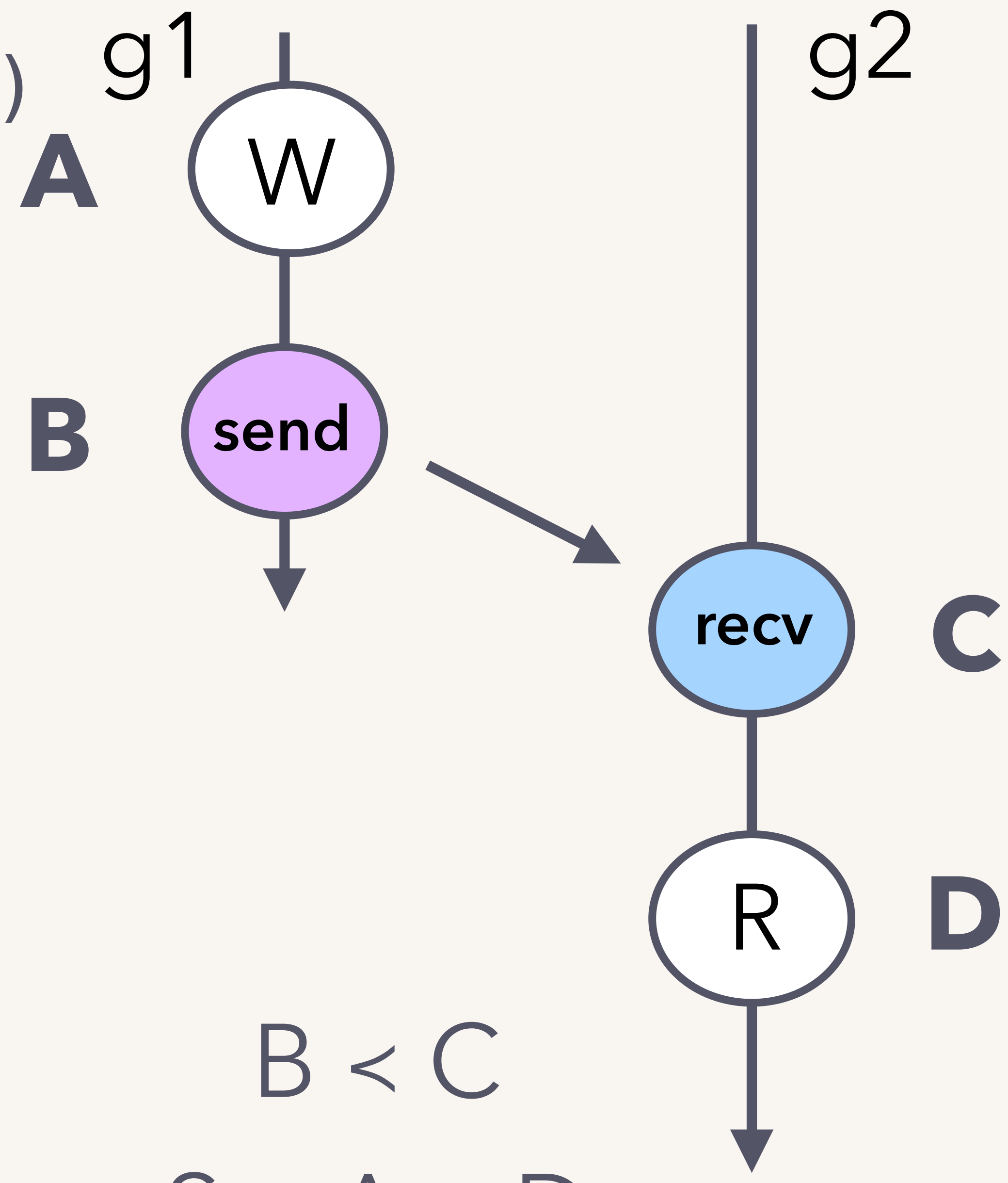
1. send happens-before corresponding receive

```
// Shared variable
var count = 0
var ch = make(chan bool, 1)

func setCount() {
  count++
  ch <- true
}

func printCount() {
  <- ch
  print(count)
}

go setCount()
go printCount()
```



B < C

So, A < D

2. n^{th} receive on a channel of size C happens-before $n+C^{th}$ send completes.

```
var maxOutstanding = 3
var taskCh = make(chan int, maxOutstanding)

func worker() {
  for {
    t := <-taskCh
    processAndStore(t)
  }
}

func main() {
  go worker()

  tasks := generateHellaTasks()
  for _, t := range tasks {
    taskCh <- t
  }
}
```

1. send happens-before corresponding receive.

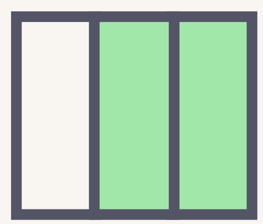
If channel empty:
receiver goroutine paused;
resumed after a channel send occurs.

If channel not empty:
receiver gets first unreceived element
i.e. buffer is a FIFO queue.

Sends must have completed due to mutex.

2. n^{th} receive on a channel of size C happens-before $n+C^{th}$ send completes.

“2nd receive happens-before 5th send.”



send #3 can occur.

send #4 can occur after receive #1.

send #5 can occur after receive #2.

Fixed-size, circular buffer.

2. n^{th} receive on a channel of size C happens-before $n+C^{th}$ send completes.

If channel full:
sender goroutine paused;
resumed after a channel recv occurs.

If channel not empty:
receiver gets first unreceived element
i.e. buffer is a FIFO queue.

Send of that element must have completed due to channel mutex