Reactive Programming: Distributed computing with actors - Eventual consistency

Jasper Cheung and Hannan Abid

Strong Consistency

- After an update completes, all reads will return the updated value.

```
private var field = 0
def update(f: Int => Int): Int = synchronized {
  field = f(field)
  field
}
def read(): Int = synchronized { field }
```

- All reads and updates are **serialized** (in a order).
- Guarantees **updated reads**, at the cost of **high latency**

Strong Consistency Examples

ACID

- Transactions: a unit of work performed against a database.
- Atomic: entire transaction takes place at once or doesn’t happen at all.
- Consistent: database is consistent before and after the transaction
- Isolation: concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially.
- Durability: ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist.

Weak Consistency

- After an update, conditions need to met until reads return the update value.
- Inconsistency window: period of time between a update and when all observers are guaranteed to see that observed value.

```
private @volatile var field = 0
def update(f: Int => Int): Future[Int] = Future {
  synchronized {
    field = f(field)
    field
  }
}
def read(): Int = field
```

- **Low latency** reads, at the cost of **old data** returned from reads

DEMO: Strong and Weak Consistency

Eventual Consistency

- A type of weak consistency
- Once no more updates, are made to an object there is a time after which all reads return the last written value.

BASE

- (B)asically (A)vailable: basic reading and writing operations are available as much as possible, but without any kind of consistency guarantees (May not get the latest write)
- (S)oft state: no consistency guarantees, after some amount of time, we only have some probability of knowing the state
- (E)ventually consistent: If the system is functioning and we wait long enough after any given set of inputs, we will eventually be able to know what the state of the database is.

Eventual Consistency and Actors

- Actors need to have consistency regarding the data that is shared
- Messaging between actors is not guaranteed and does not happen synchronously
- Eventual consistency needs to be implement in actors.
- Eventual consistency requires that all parties are notified of all updates
- Eventual consistency and actors need suitable data structures.
    - Known as convergent or commutative replicated data types

CRDTs

Properties:

- Lock-free
- Wait-free
- Do not require consensus

$x = 0$

$x = 0$

$x = 0$

x = 0

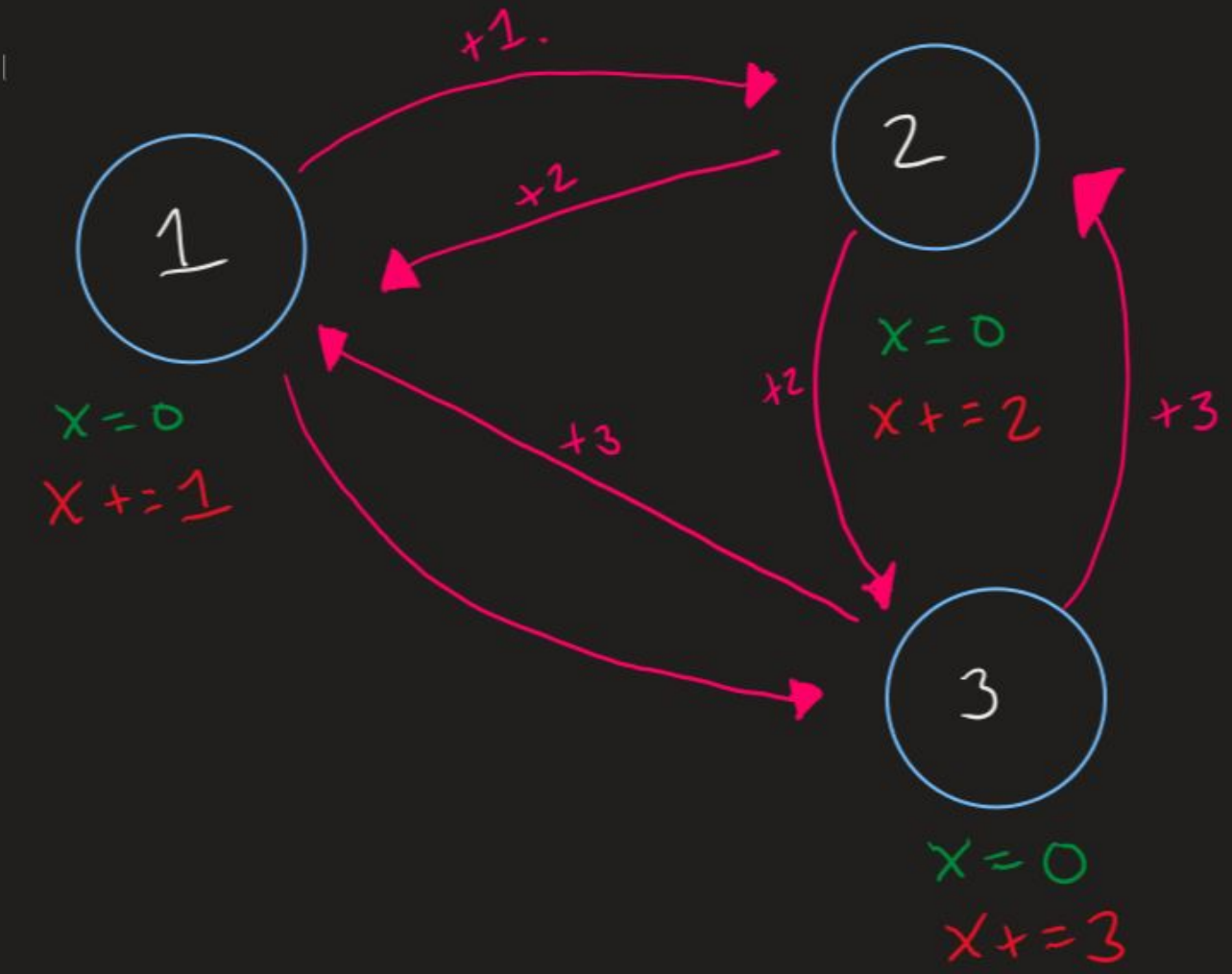
x+ = 1

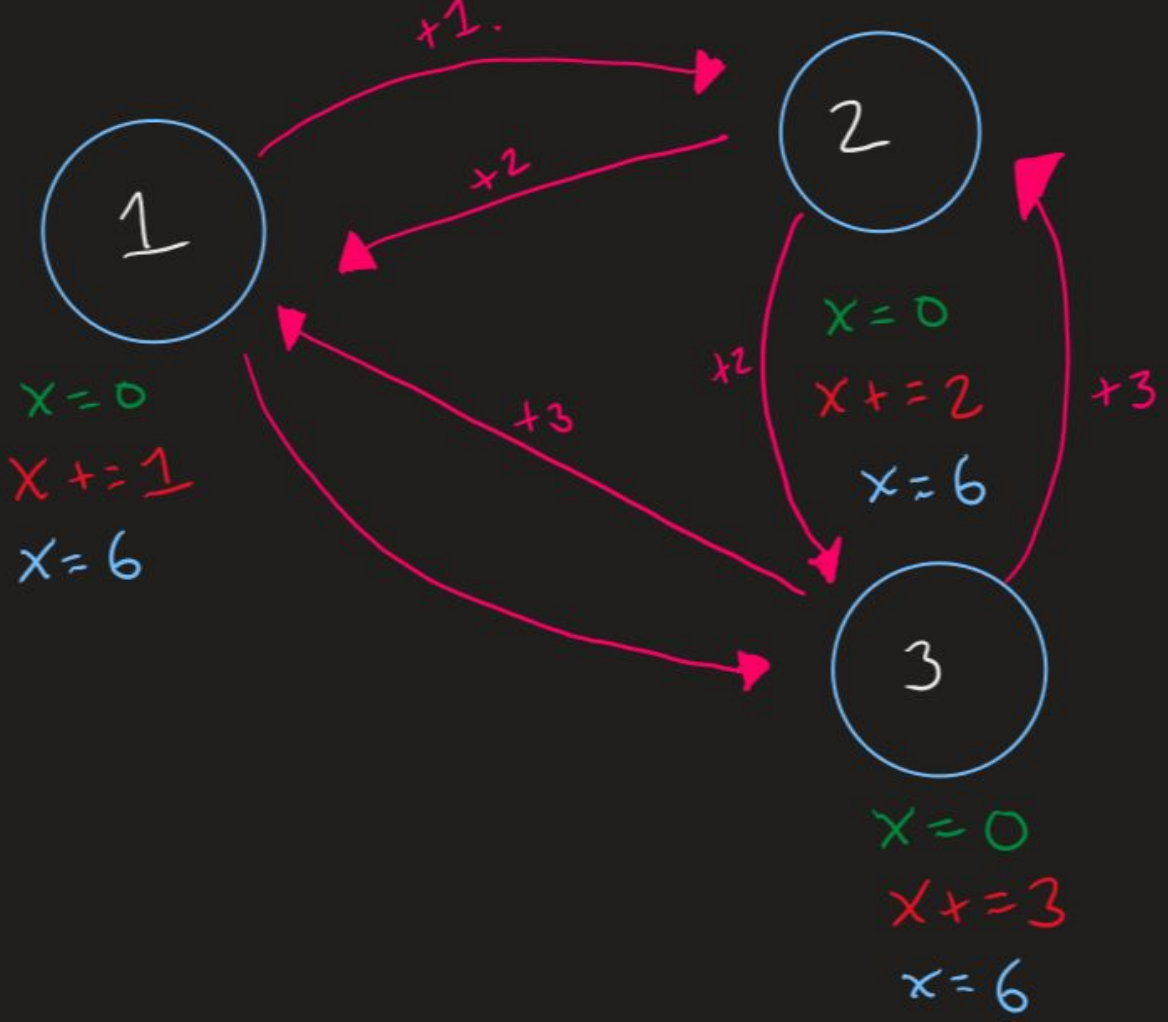
x = 0

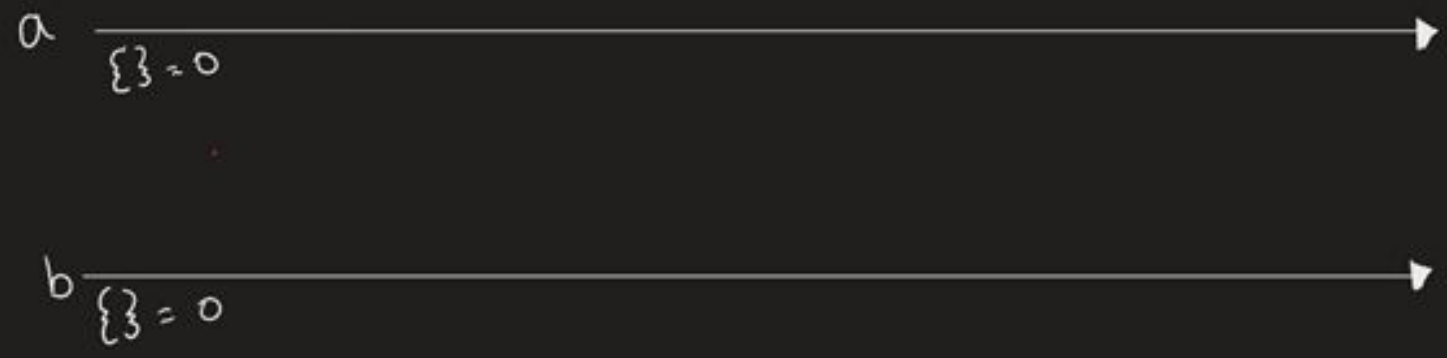
x+ = 2

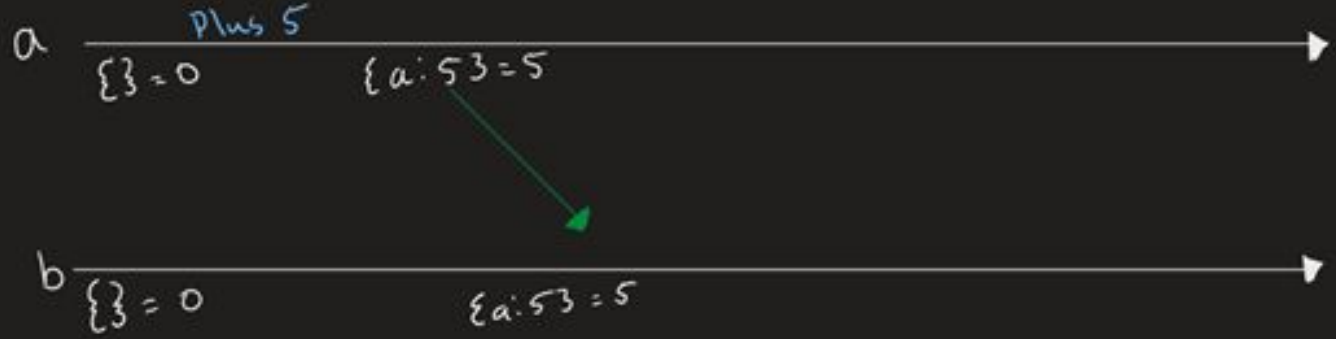
x = 0

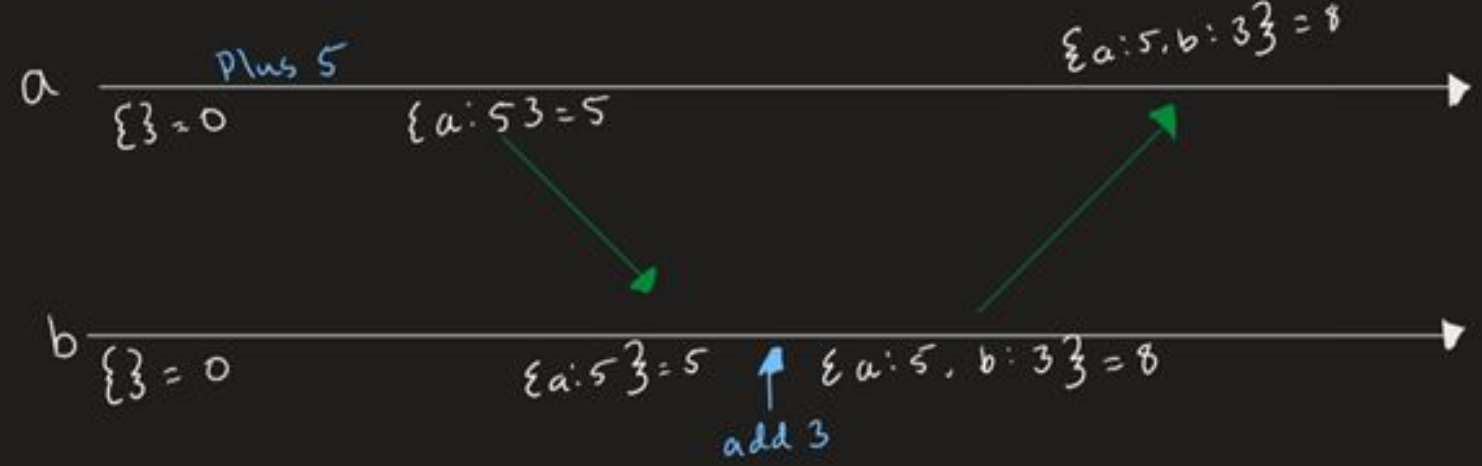
x+ = 3

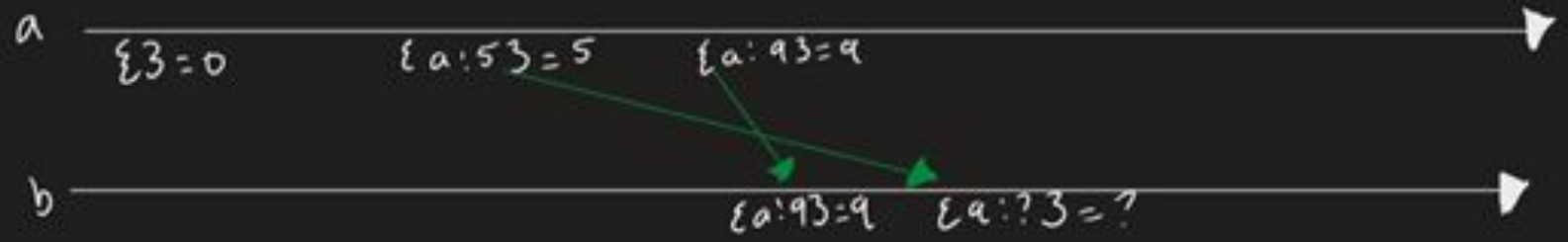

+1.
+2
+3
+2
+3
1
2
3
X = 0
X += 1
X = 0
X += 2
X = 0
X += 3

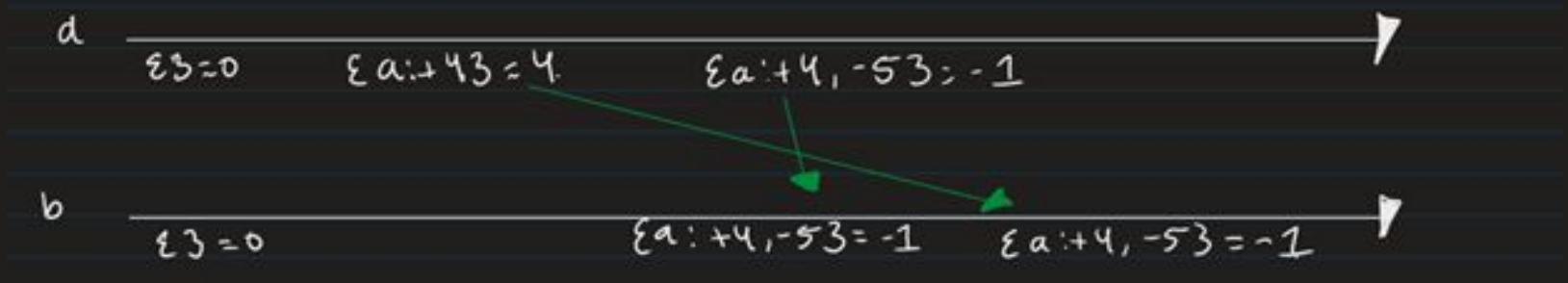

+1.
+2
+3
1
2
3
x = 0
x += 1
x = 6
x = 0
x += 2
x = 6
+2
+3
x = 0
x += 3
x = 6


a
{} = 0
b
{} = 0


a
Plus 5
{} = 0
{a: 53 = 5
b
{} = 0
{a: 53 = 5

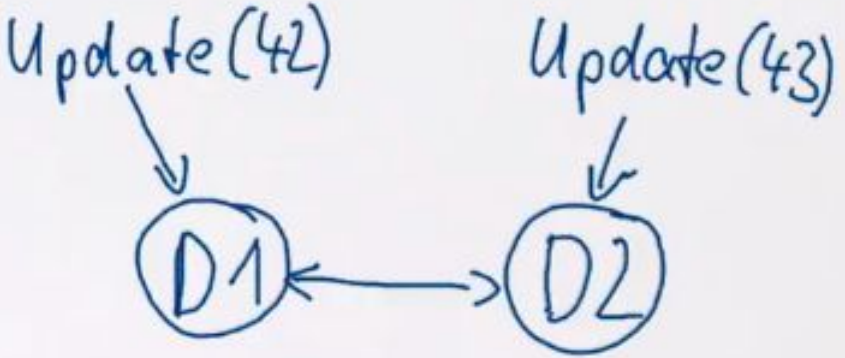

a
Plus 5
{} = 0
{a: 5} = 5
{a: 5, b: 3} = 8
b
{} = 0
{a: 5} = 5
{a: 5, b: 3} = 8
add 3


a
{3=0
{a:5}3=5
{a:9}3=9
b
{a:9}3=9
{a:?}3=?


a
{3=0
{a:+4}=4
{a:+4,-5}=-1
b
{3=0
{a:+4,-5}=-1
{a:+4,-5}=-1

Example

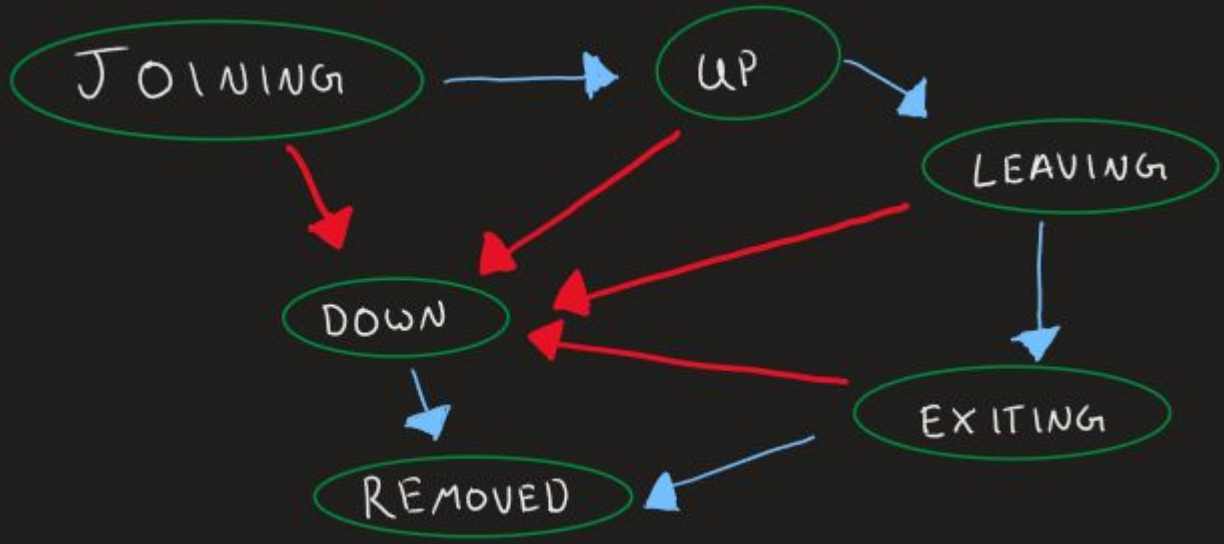
```
case class Update(x: Int)
case object Get
case class Result(x: Int)
case class Sync(x: Int, timestamp: Long)
case object Hello

class DistributedStore extends Actor {
  var peers: List[ActorRef] = Nil
  var field = 0
  var lastUpdate = System.currentTimeMillis()

  def receive = ...
}
```

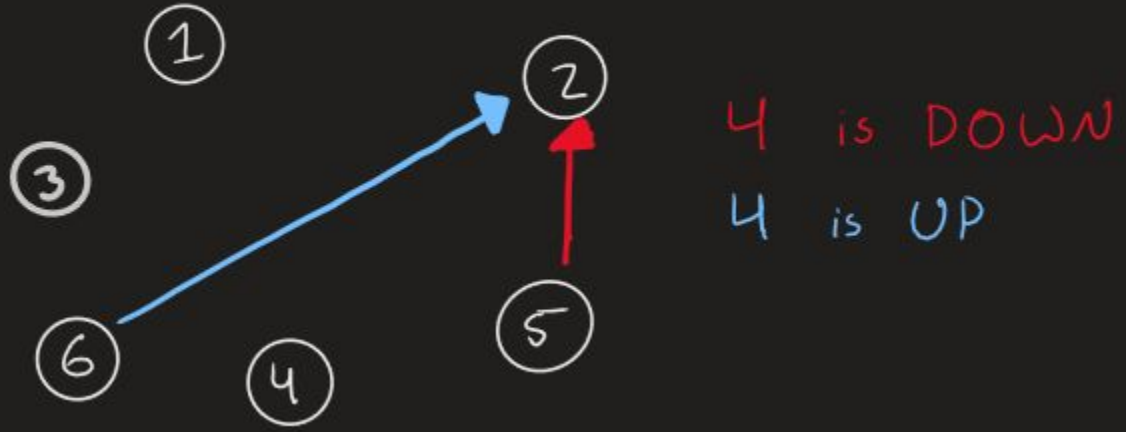


Example Cont.

```
def receive = {
  case Update(x) =>
    field = x
    lastUpdate = System.currentTimeMillis()
    peers foreach (_ ! Sync(field, lastUpdate))
  case Get => sender ! Result(field)
  case Sync(x, timestamp) if timestamp > lastUpdate =>
    field = x
    lastUpdate = timestamp
  case Hello =>
    peers ::= sender
    sender ! Sync(field, lastUpdate)
}
```

DEMO: Eventual Consistency


JOINING
UP
LEAVING
DOWN
EXITING
REMOVED


1
2
3
4
5
6
4 is DOWN
4 is UP

Resources

Book: https://cuny-hc.primo.exlibrisgroup.com/permalink/01CUNY_HC/1veum9a/alma991027548145006121

Other resources: https://drive.google.com/drive/folders/1pJDBxVcDxiYfizTua2hDmEi03g9jHs5n?usp=sharing

https://www.allthingsdistributed.com/2008/12/eventually_consistent.html

CRDTs for NonAcademics

Practical Data Synchronization with CRDTs

CRDTs Illustrated

Building a collaborative text editor with WebRTC and CRDTs

Notes - Delete Later

- If you imagine people who are living on different continents, that it takes some effort on a common truth or common decision, same thing is True for actors. This called eventual consistency.

Actor communication is asynchronous, one-way and not guaranteed.

Actor encapsulation makes them look the same, regardless where they live.

- Everything takes time, it takes time for a node to join the cluster and pass infomation to everyone else. And it takes time for the welcome message to come. They are taken in some consistent fashion but not immediately.
- Cluster is one example that is eventually consistent