Eventual Consistency

Principles of Reactive Programming

Roland Kuhn

Eventual Consistency (1)

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 }
```

Eventual Consistency (2)

Strong Consistency: after an update completes all reads will return the updated value

Weak Consistency: after an update conditions need to be met until reads return the update value; this is the *inconsistency window*

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

Eventual Consistency (3)

Strong Consistency: after an update completes all reads will return the updated value

Weak Consistency: after an update conditions need to be met until reads return the update value; this is the *inconsistency window*

Eventual Consistency: once no more updates are made to an object there is a time after which all reads return the last written value

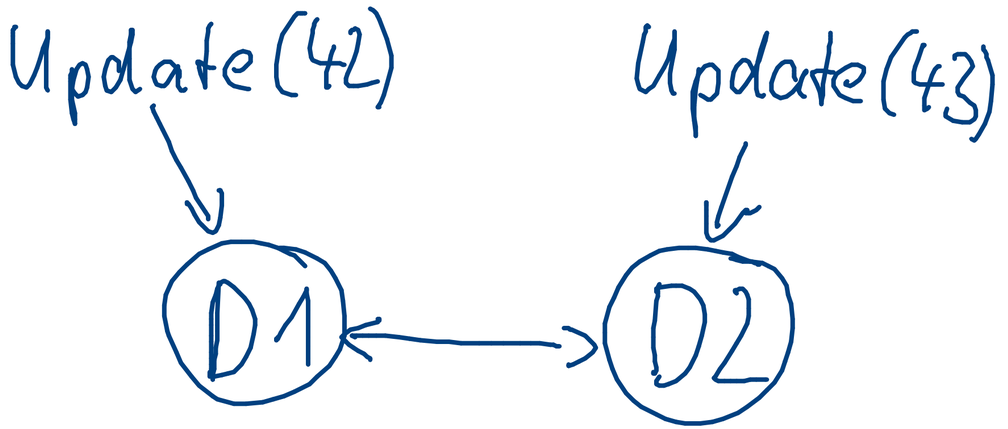
http://www.allthingsdistributed.com/2008/12/eventually_consistent.html
http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

Eventually Consistent Store (1)

```
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 = ...
}
```



Eventually Consistent Store (2)

```
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)
}
```

Actors and Eventual Consistency

- an actor forms an island of consistency
- collaborating actors can at most be eventually consistent
- actors are not automatically eventually consistent
- event consistency requires eventual dissemination of all updates
- need to employ suitable data structures, for example CRDTs[1]

[1] Shapiro, Preguiça, Baquero, Zawirski (2011): *A comprehensive study of Convergent and Commutative Replicated Data Types*, inria-00555588

An Example Data Structure

The cluster membership state is a convergent data type:

- directed acyclic graph of states
- conflicts can always be resolved locally
- conflict resolution is commutative