Distributed ES with Akka Persistence

Duncan K. DeVore
SWE @ Typesafe
@ironfish

Typesafe

Outline

1. Introduction
2. What is Event Sourcing?
3. What are Commands?
4. What are Events?
5. What is Akka Persistence?
6. What is Akka Cluster Sharding?
7. Consistency
8. Conclusion

Outline

1. Introduction
2. What is Event Sourcing?
3. What are Commands?
4. What are Events?
5. What is Akka Persistence?
6. What is Akka Cluster Sharding?
7. Consistency
8. Conclusion

It’s a different world

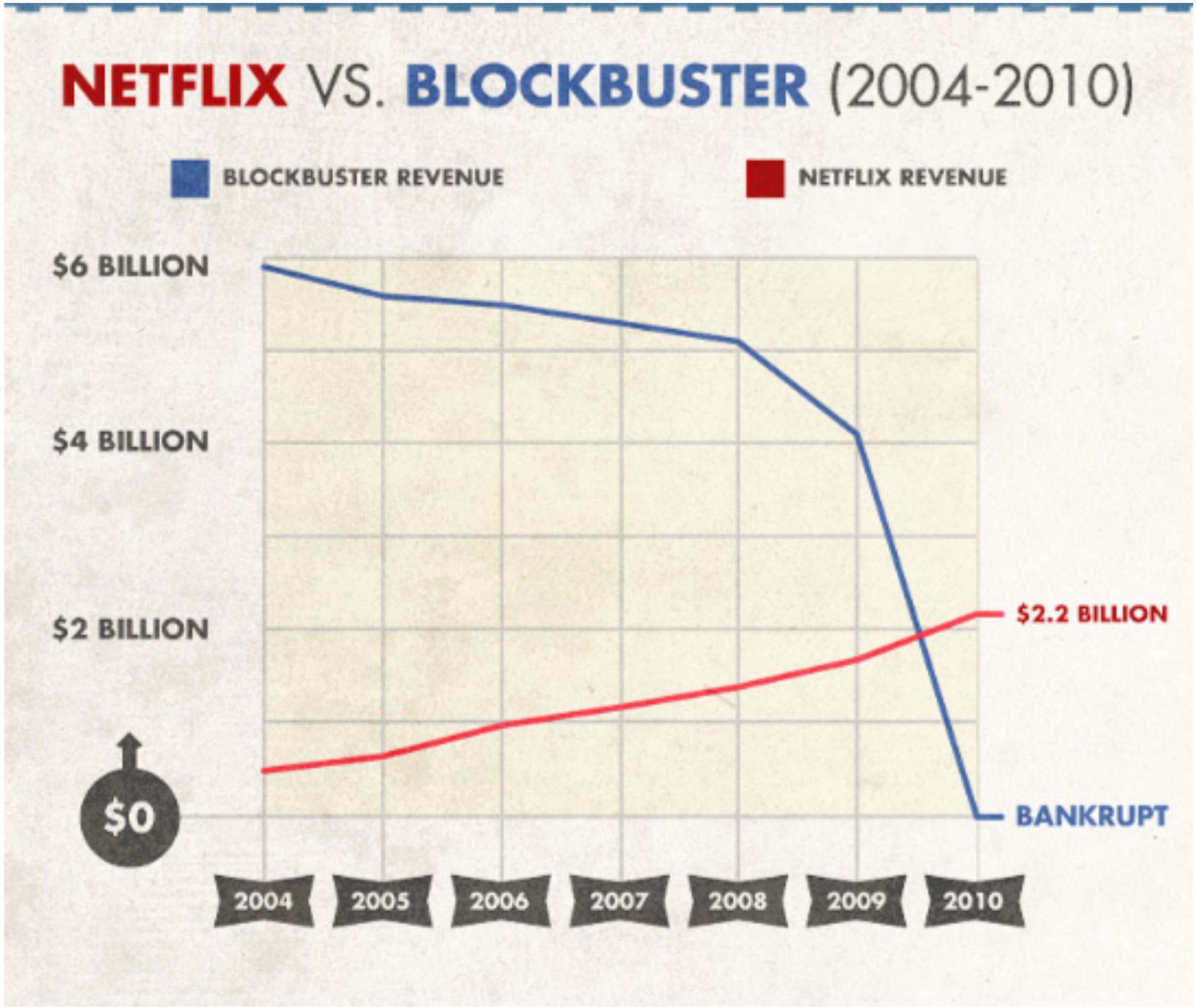
out there

Yesterday	Today
Single machines	Clusters of machines
Single core processors	Multicore processors
Expensive RAM	Cheap RAM
Expensive disk	Cheap disk
Slow networks	Fast networks
Few concurrent users	Lots of concurrent users
Small data sets	Large data sets
Latency in seconds	Latency in milliseconds

*A study by MIT Sloan Management Review and Capgemini Consulting finds that companies now face a **digital imperative**: adopt new technologies effectively or face competitive **obsolescence**.*

- October 2013

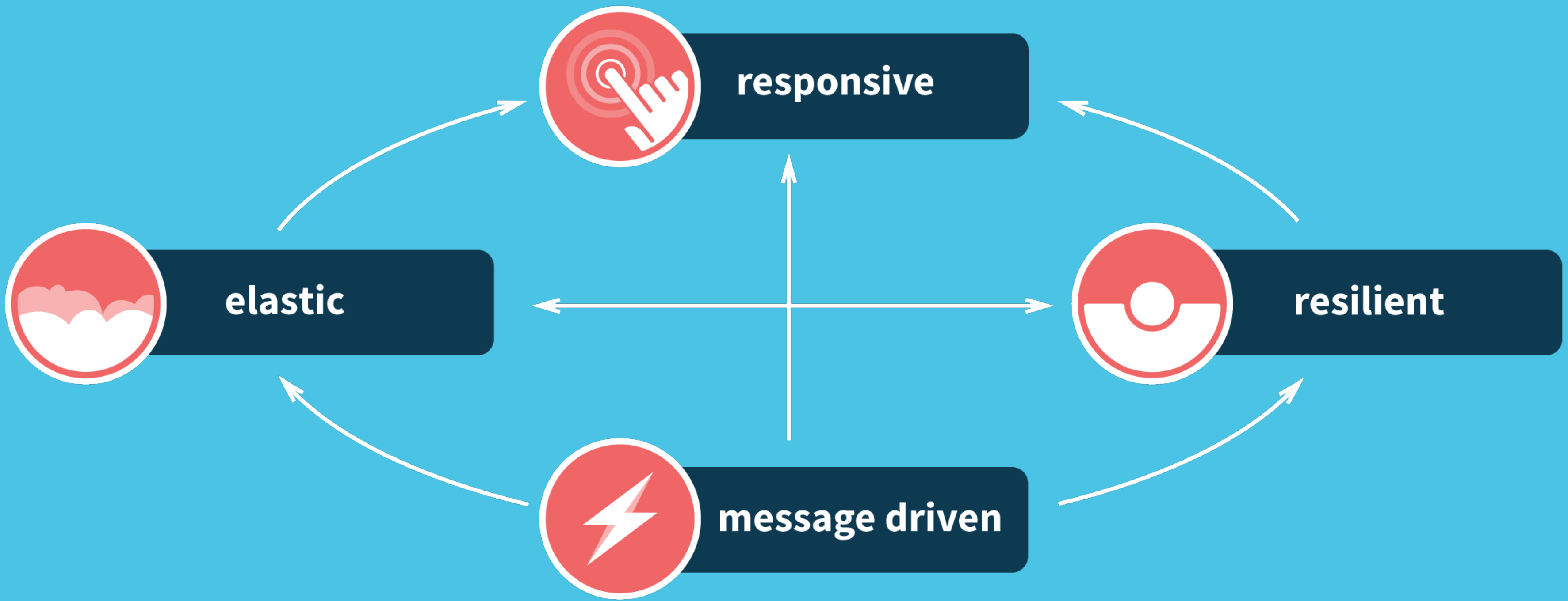
Case and Point



*"In today's world, the demand for distributed systems has exploded. As customer expectations such as an **immediate response**, **no failure**, and **access anywhere** increase, companies have come to realize that distributed computing is the **only** viable solution."*

- Reactive Application Development (Manning)

Reactive Systems



*"Modern applications must **embrace** these changes by incorporating this **behavior** into their **DNA**".*

- Reactive Application Development (Manning)

Outline

1. Introduction
2. What is Event Sourcing?
3. What are Commands?
4. What are Events?
5. What is Akka Persistence?
6. What is Akka Cluster Sharding?
7. Consistency
8. Conclusion

what is
event sourcing?

*"The **majority** of business applications today rely on storing **current state** in order to process transactions. As a result in order to track history or implement audit capabilities **additional** coding or frameworks are required."*

- Greg Young

Current State Only??

This was **not always** the case:

- Side-effect of the adoption of **RDBMS** systems
- High performance, systems **do not** do this
- Mission critical systems **do not** do this
- RDBMS's do not do this **internally**!
- SCADA (System Control and Data Acquisition) Historian
- File control **audit** systems

*"Event sourcing provides a means by which we can capture the **real behavior (intent)** of our users"*

- Reactive Application Development (Manning)

Event Sourcing

Historical **behavior** is captured

- **Behavioral** by nature
- Convert valid **commands** into **one or more** events
- Current state **is not** persisted
- Current state is **derived**
- **Append only** store with **simple** key structure
- Designed for **distribution**

*"This pattern can **simplify** tasks in **complex** domains by avoiding the requirement to **synchronize** the data model and the business domain"*

- Reactive Application Development (Manning)

what are
commands?

command | kə`mand |

- [reporting verb] give an authoritative order: [with obj. and infinitive]

Commands

Commands are about **behavior** rather than data centricity.

Commands are a **request** of the system to perform a **task** or **action**. They follow a **VerbNoun** format, for example:

```
case class RegisterClient(id: String, . . .)

case class ChangeClientLocale(id: String, expVer: Long, . . .)
```

Commands

- Commands are **imperative**
- They are requests to **mutate state**
- An action one **would like** to take
- Transfer as **messages** not DTO's
- Implies **task-based** UX

Commands

- Conceptually, performing **task**
- **Not** data edits, rather behavioral **request**
- Can be **rejected**
- They do not **expose** internal state
- Greatly **simplified** repository layer
- Single command can = **multiple** events

what are
events?

event | i`vent |

noun

- a thing that happens, especially one of importance

Events

Events are **Indicative** in nature. They serve as a sign or **indication** that something has **happened**.

As such, they are **immutable** and cannot be **rejected**. They follow a **NounVerb** format, for example:

```
case class ClientRegistered(id: String, ver: Long, . . .)

case class ClientLocaleChanged(id: String, ver: Long, . . .)
```

Events

- **Atomic** by nature
- Record of state **change**
- **VerbNoun** implies behavior
- **Immutable**
- Natural, **verifiable** audit log
- Cannot be **rejected**

Canonical Example

One of the best ways to understand event sourcing is to look at the **canonical** example, a bank account register.

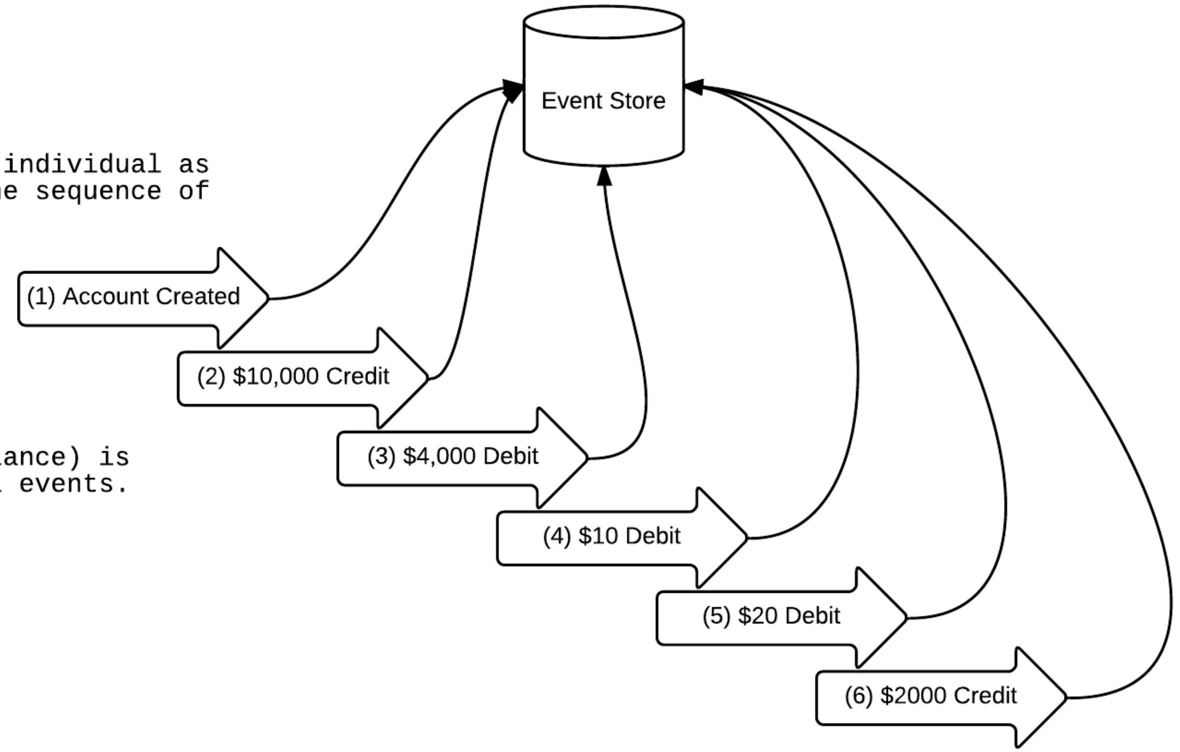
Date	Comment	Change	Balance
7/1/2014	Deposit from 3300	+ 10,000.00	10,000.00
7/3/2014	Check 001	- 4,000.00	6,000.00
7/4/2014	ATM Withdrawal	- 3.00	5,997.00
7/11/2014	Check 002	- 5.00	5,992.00
7/12/2014	Deposit from 3301	+ 2,000.00	7,992.00

Canonical Example

Each event is persisted individual as a delta. Following is the sequence of events:

1. Account Created
2. $10,000 Credit
3. $4,000 Debit
4. $10 Debit
5. $20 Debit
6. $2,000 Credit

NOTE: Current state (balance) is derived by replaying all events.



Canonical Example

- We persist each transaction as an **independent** event
- To calculate the balance, the **delta** of the current transaction is **applied** to the last known value
- We have a **verifiable** audit log that can be **reconciled** to ensure **validity**
- The current balance at **any** point can be **derived** by **replaying** all the **transactions** up to that point
- We have captured the **real** intent of **how** the account holder manages their finances

Outline

1. Introduction
2. What is Event Sourcing?
3. What are Commands?
4. What are Events?
5. What is Akka Persistence?
6. What is Akka Cluster Sharding?
7. Consistency
8. Conclusion

what is
akka persistence?

Akka Persistence

- **PersistentActor** model can **persist** internal state
- Underlying semantics use **event sourcing**
- **Append-only** store w/ **community** plugins
- **State** recovered by **replaying** events/snapshots
- **Point-to-point** communication with **at-least-once** message delivery
- **Identity** or **Identity** and **State**!

PersistentActor

- Persistent, **stateful** actor model
- Reacts in a **thread-safe** manner
- Supports **command** sourcing
- Supports **event** sourcing
- Implicit **replay** optimization
- **Recovers** the state from journal/snapshots

Journal

- Event sourced **append-only** storage model
- Application **controls** message journaling
- Journal implementation is **pluggable**
- The **default** journals to the filesystem
- **Replicated** journals @ Community Plugins

Snapshots

- A snapshot stores a “**moment-in-time**”
- Snapshots **internal state** of the actor
- Used for **optimizing** recovery times
- Snapshot implementation is **pluggable**.
- The **default** snapshots to the filesystem.
- **Replicated** snapshots @ Community Plugins

receiveCommand

```
class Client extends PersistentActor {

  . . .

  val receiveCommand: Receive = { //<- process commands

    case cmd: RegisterClient => validateRegistration(cmd) fold (

      f => sender ! f,

      s => persist(Event) { e =>

        state = state.update(e)

        // side effects go here

        . . .

}
```

Identity and State

```
object Client {
  . . .
  private def empty: Client = Client()
  private case class State(c: Client) {
    def update(e: Event): State = e match {
    . . .
  }
}
class Client extends PersistentActor {
  var state = State(empty) //<- mutable state OK!
    . . .
}
```

Typesafe

receiveRecover (resilience)

```scala
class Client extends PersistentActor {
  . . .
  val receiveRecover: Receive = {
    case e: Event => e match {
      case evt: ClientRegistered =>
        state = state.update(evt)
       // there should be no side effects here
        . . .
    }
    case SnapshotOffer(_, snapshot: Client) => state = snapshot
}
```

Typesafe

what is
akka cluster sharding?

Akka Cluster Sharding

- **Identity** distribution across several nodes
- One machine is **not enough**, cluster required
- Naturally **elastic** & **resilient**
- Actor **activation** & **passivation**
- Messages sent to **shard** (think proxy)
- Physical location of **actor** is managed

Shard Region

- ShardRegion actor on **each** cluster **node**
- Entry point for **entity** identification
- **Extracts** identifier from **incoming** messages
- If actor location **unknown**, asks ShardCoordinator

Shard Coordinator

- ShardCoordinator **decides** (1st message):
  - which **ShardRegion** that owns the shard
- **Subsequent** messages:
  - delivered to the **target** destination
  - **without** involving the ShardCoordinator

Shard Region

```
  . . .
val clientRegion: ActorRef = ClusterSharding(system).start(
  typeName = Client.shardName,
  entryProps = Some(Client.props),
  idExtractor = Fellow.idExtractor,
  shardResolver = Client.shardResolver)
. . .
val cmd = ChangeClientName("123", "Jason", expVer=4)
clientRegion ! cmd
```

Outline

what is

consistency?

consistency | kən’sistənsē |

noun

- conformity in the application of something, typically necessary for the sake of logic; accuracy or fairness

*“Consistency is often taken for **granted** when designing traditional **monolithic** systems as you have **tightly** coupled services connected to a **centralized** database”*

- Reactive Application Development (Manning)

Strong Consistency

Traditional monolithic systems default to Strong Consistency as there is only **one path** to the data store for a given service and that path is **synchronous** in nature.

- All **accesses** are available to all **processes**
- All **accesses** are seen in the same **sequential** order

Strong Consistency

In distributed computing, however, this is **not the case**. By design, distributed systems are **asynchronous** and **loosely coupled** and rely on patterns such as atomic shared memory systems and distributed data stores achieve Availability and Partition Tolerance

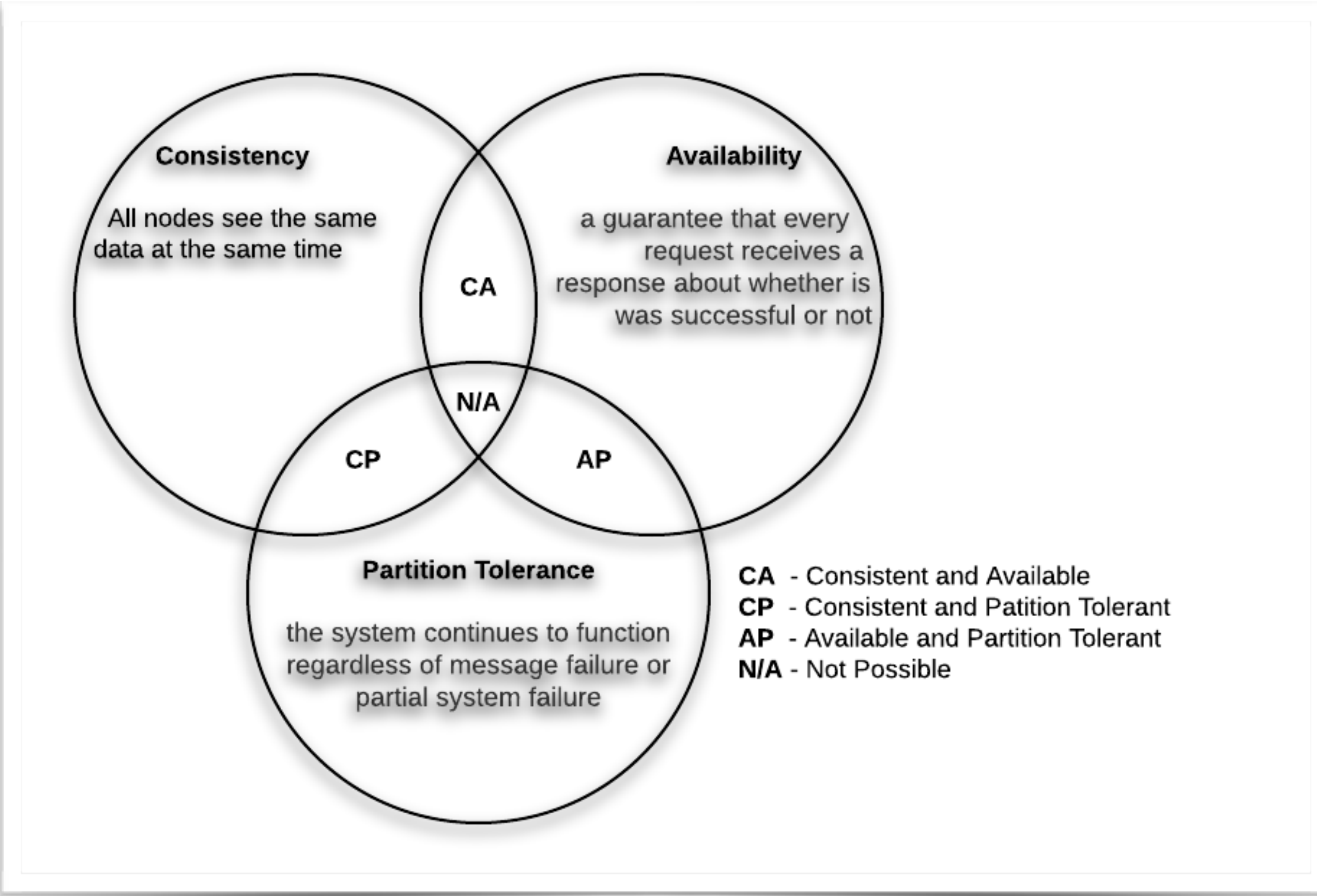
Therefore, strongly consistent systems are not distributable **as a whole contiguous system** as identified by the CAP theorem.

CAP Theorem

In Theoretical Computer Science, CAP Theorem, also known as Brewer's Theorem, states that its **impossible** in Distributed Systems to **simultaneously** provide **all three** of the following **guarantees**:

- Consistency - all **nodes** see the **same** data at the **same** time
- Availability - a guarantee that **every** request receives a **response** about whether successful or not
- Partition Tolerance - the system **continues** to **function** regardless of **message** failure or **partial system** failure

CAP Theorem


Consistency
All nodes see the same data at the same time
Availability
a guarantee that every request receives a response about whether is was successful or not
CA
N/A
CP
AP
Partition Tolerance
the system continues to function regardless of message failure or partial system failure
CA - Consistent and Available
CP - Consistent and Patition Tolerant
AP - Available and Partition Tolerant
N/A - Not Possible

"In distributed computing, a system supports a given ***consistency*** *model if operations follow* ***specific*** *rules as* ***identified*** *by the model. The model specifies a* ***contractual*** *agreement between the* ***programmer*** *and the* ***system****, wherein the system* ***guarantees*** *that if the rules are followed, memory will be* ***consistent*** *and the results will be* ***predictable****."*

- Wikipedia

Eventual Consistency

Eventual consistency is a consistency model used in distributed computing that **informally guarantees** that, if no new updates are made to a given data item, **eventually** all accesses to that item will return the last **known** value.

- **Pillar** of distributed systems
- Often under the moniker of **optimistic replication**
- Matured in the **early** days of **mobile** computing

Eventual Consistency

A system that has **achieved** eventual consistency is often said to have **converged**, or achieved replica convergence.

- While stronger models, like **linearizability** (Strong Consistency) are **trivially** eventually consistent, the converse does not hold.
- **Eventually Consistent** services are often classified as as **B**asically **A**vailable **S**oft state **E**ventual consistency semantics as opposed to a more traditional **ACID** (Atomicity, Consistency, Isolation, Durability) guarantees.

Causal Consistency

Causal consistency is a **stronger** consistency model that **ensures** that the operations processes in the order expected.

More precisely, partial order over operations is **enforced** through **metadata**.

- If operation A occurs before operation B, then any data center that sees operation B must see operation A first.

There are three rules that define potential causality.

Causal Consistency (3 Rules)

- **Thread of Execution**: If A and B are two operations in a single thread of execution, then A -> B if operation A happens before B.
- **Reads-From**: If A is a write operation and B is a read operation that returns the value written by A, then A -> B.
- **Transitivity**: For operations A, B, and C, if A -> B and B -> C, then A -> C. Thus the casual relationship between operations is the **transitive closure** of the first two rules.

what is
conflict resolution?

resolution | rezə’lōōSHən |

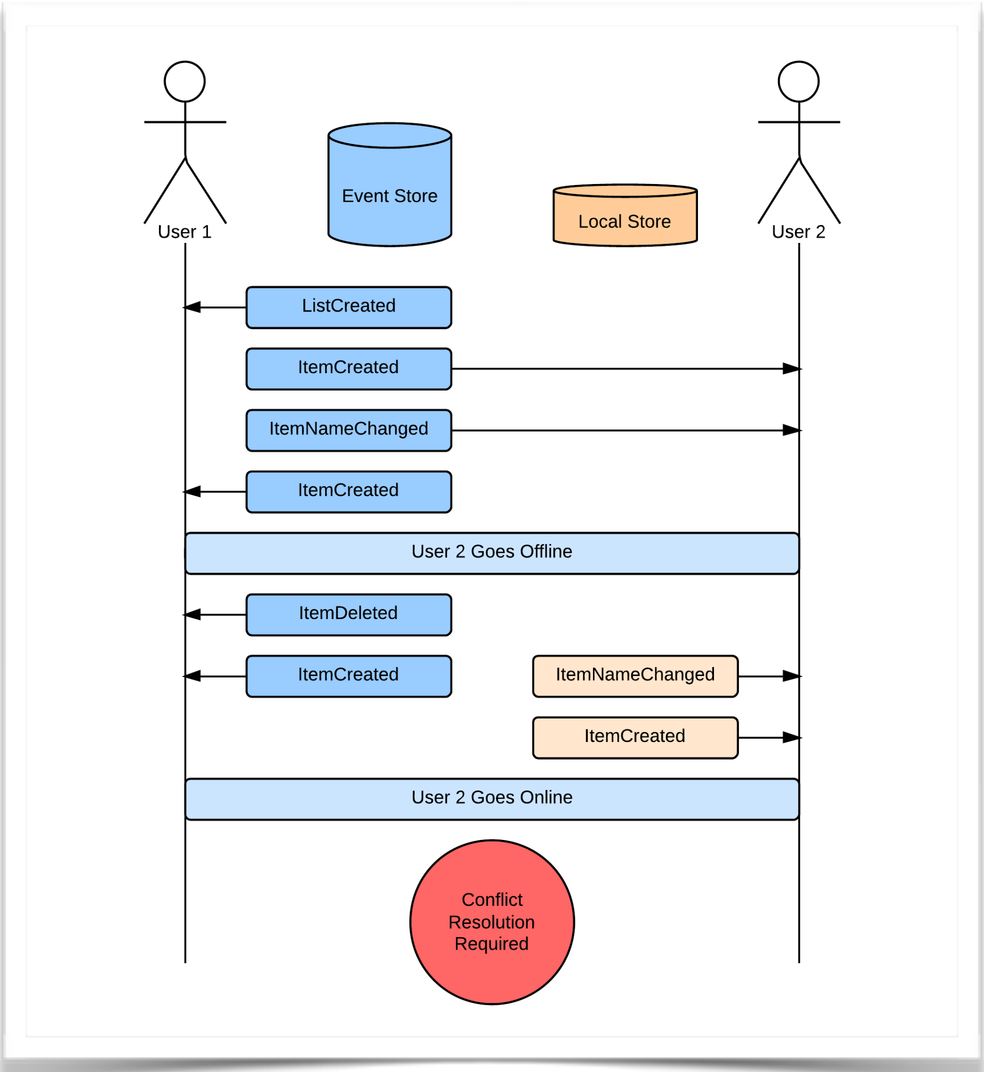
noun

- a firm decision to do or not to do something

*"In order to ensure the **convergence** of replicated data, a **reconciliation** between the distributed copies is **required**. This process, often known as [anti-entropy], requires versioning **semantics** to as part of the data"*

- Wikipedia

Conflict Resolution



Conflict Resolution

The recommended way to solve is the problem for the command side of CQRS, is by **embedding** into the data structure a simple **metadata** attribute, **version number**.

- Known as **current state** versioning
- Does **command** version = **current state** version?
- If they are not equal, the command is **rejected**
- **First** writer **wins**.

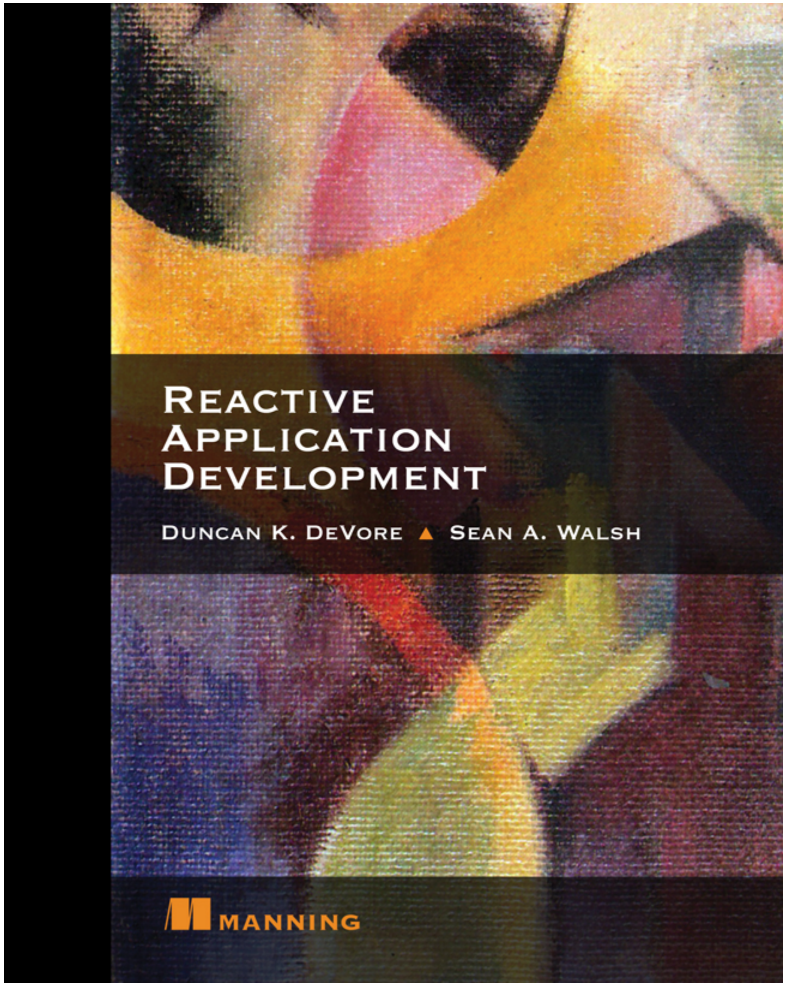
Conflict Resolution

```
object Client {
  def requireVersion[C <: EventableCommand]
      (c: Client, cmd: C): Either[ErrorMsg, C] =
    if(cmd.expVer == c.ver)
      Right(cmd)
    else
      Left(ErrorMsg(List("Expected version mismatch")))
  . . .
}
```

Outline

1. Introduction
2. What is Event Sourcing?
3. What are Commands?
4. What are Events?
5. What is Akka Persistence?
6. What is Akka Cluster Sharding?
7. Consistency
8. Conclusion

Reactive Application Development



Questions?

Distributed ES with Akka Persistence

Duncan K. DeVore
Typesafe
@ironfish