CQRS / ES

Duncan K. DeVore
Typesafe
@ironfish

Typesafe

Outline

1. Introduction
2. The “C” in CQRS
3. Event Sourcing
4. The “Q” in CQRS
5. Consistency
6. Conclusion

Outline

1. Introduction
2. The “C” in CQRS
3. Event Sourcing
4. The “Q” in CQRS
5. Consistency
6. Conclusion

It’s a different world

out there

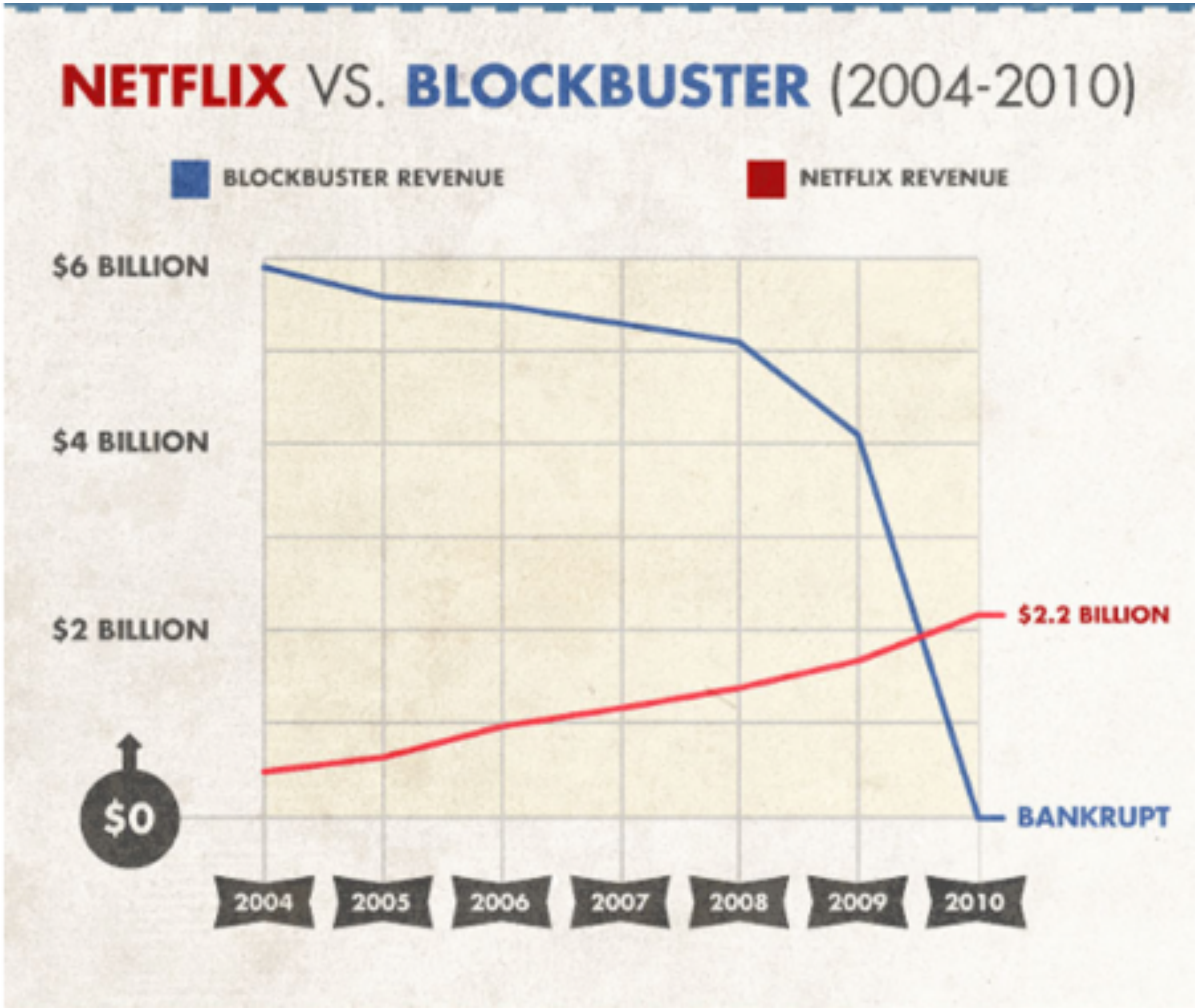
Yesterday	Today
Single machines	Clusters of machines
Single core processors	Multicore processors
Expensive RAM	Cheap RAM
Expensive disk	Cheap disk
Slow networks	Fast networks
Few concurrent users	Lots of concurrent users
Small data sets	Large data sets
Latency in seconds	Latency in milliseconds

Typesafe

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

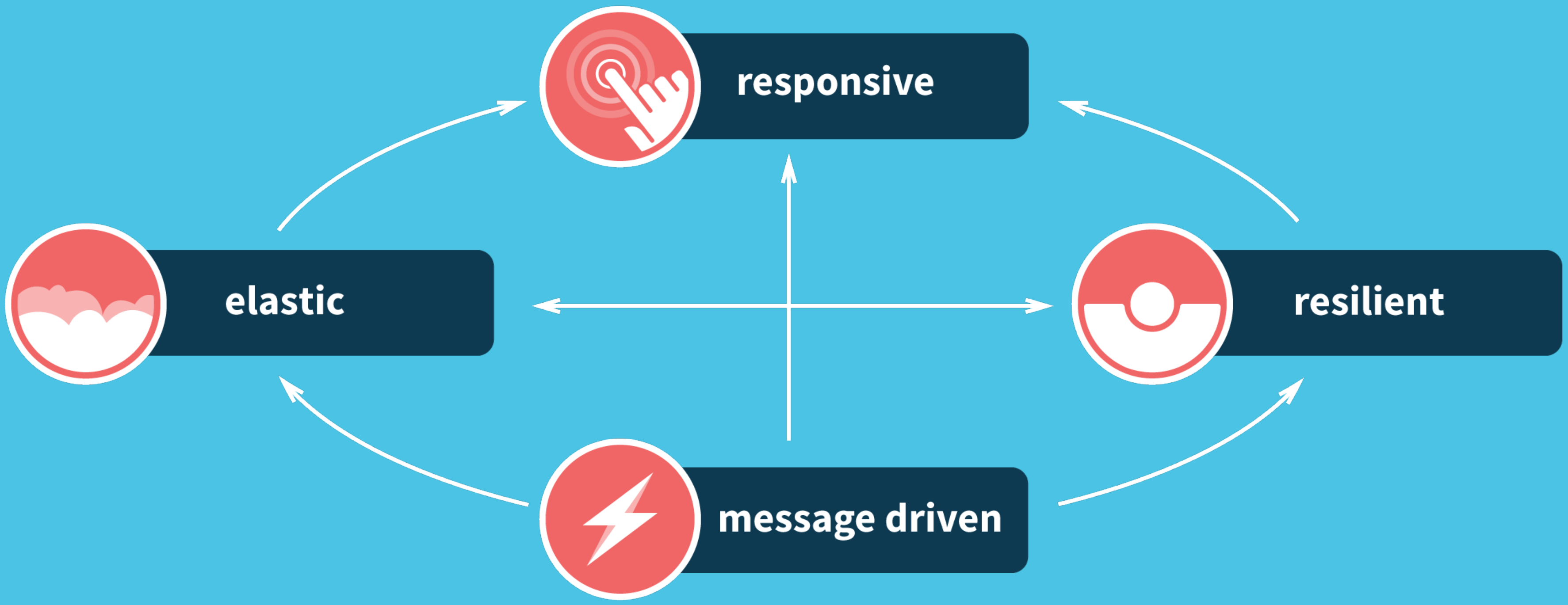
Case and Point



“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)

what is
cqrs?

Origins from CQS

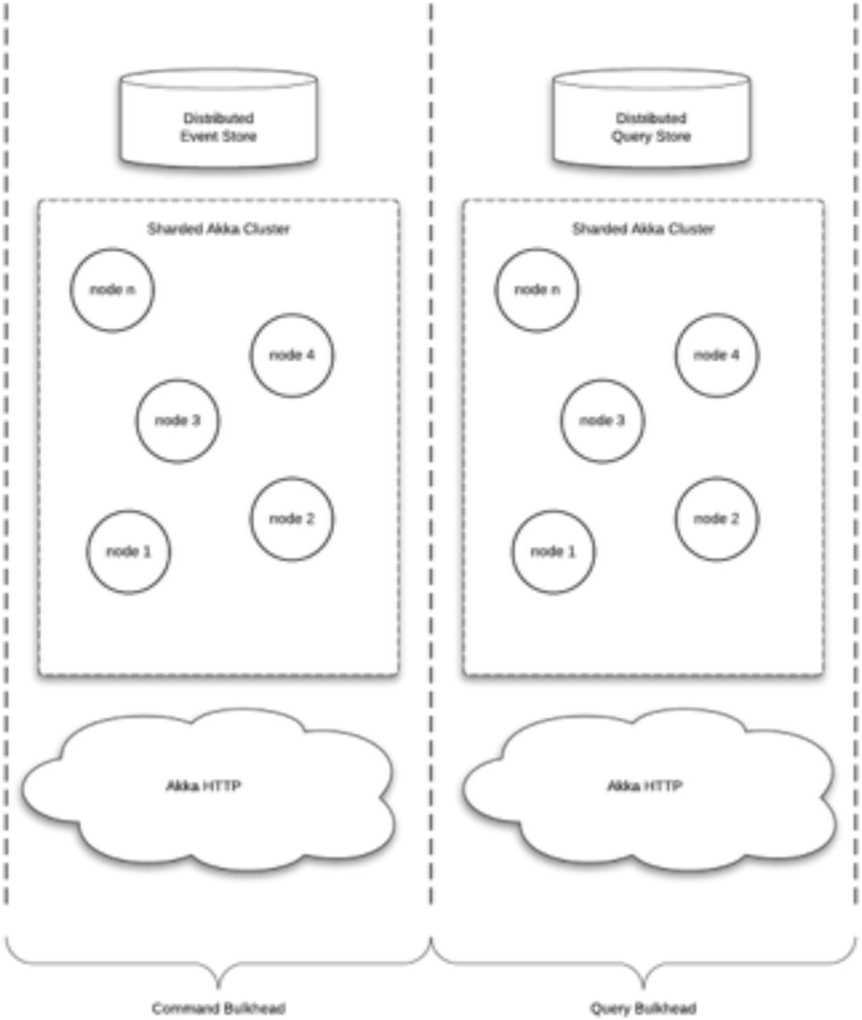
Command Query Separation

- *Object Oriented Software Construction* by Bertrand Meyer
- Methods should be either **commands** or **queries**
- A query **returns** data, does **not** alter the state
- A command **changes** the state, does **not** return data.
- Becomes clear what **does** and **does not** change state

*"CQRS is simply the creation of **two objects** where there was previously only one. The separation occurs based upon whether the methods are a command or a query (the same definition that is used by Meyer in Command and Query Separation: a command is any method **(object)** that mutates state and a query is any method **(object)** that returns a value)"*

- Greg Young

Two Distinct Paths



- One for writes (commands)
- One for reads (queries)
- Allows **separate** optimization
- **Simpler** reasoning about paths

Reason for Segregation

- Large imbalance between the number of reads and writes
- Single model encapsulating reads/writes **does neither** well
- Command side often involves **complex** business logic
- Read side **de-normalized** (redundant) for fast queries
- More atomic and easier to reason about
- Read side easily **re-creatable**

Outline

1. Introduction
2. The “C” in CQRS
3. Event Sourcing
4. The “Q” in CQRS
5. Consistency
6. Conclusion

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. This leads to a more true implementation of DDD.

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 behavior
- Can be **rejected**
- They do not **expose** internal state
- Greatly **simplified** repository layer
- Single command can = **multiple** events

Command Handler

In CQRS command handlers are objects that process commands

- Client sends command in form of a **message**
- Processed by a command handler
- Commands can be **rejected**
- If **valid**, become one or more events

Command Handlers

```
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

        . . .

}
```

Outline

1. Introduction
2. The “C” in CQRS
3. Event Sourcing
4. The “Q” in CQRS
5. Consistency
6. 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

Event Sourcing

This was **not always** the case

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

“Event sourcing provides a means by which we can capture the real 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

"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 is an
event?

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**
- Immutable
- 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.

In a **mature** business model, the notion of tracking behavior is quite **common**. Consider, for example, a bank accounting system.

- A customer can make deposits
- Write checks
- Make ATM withdrawals
- Transfer monies to other accounts
- Etc.

Canonical Example

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

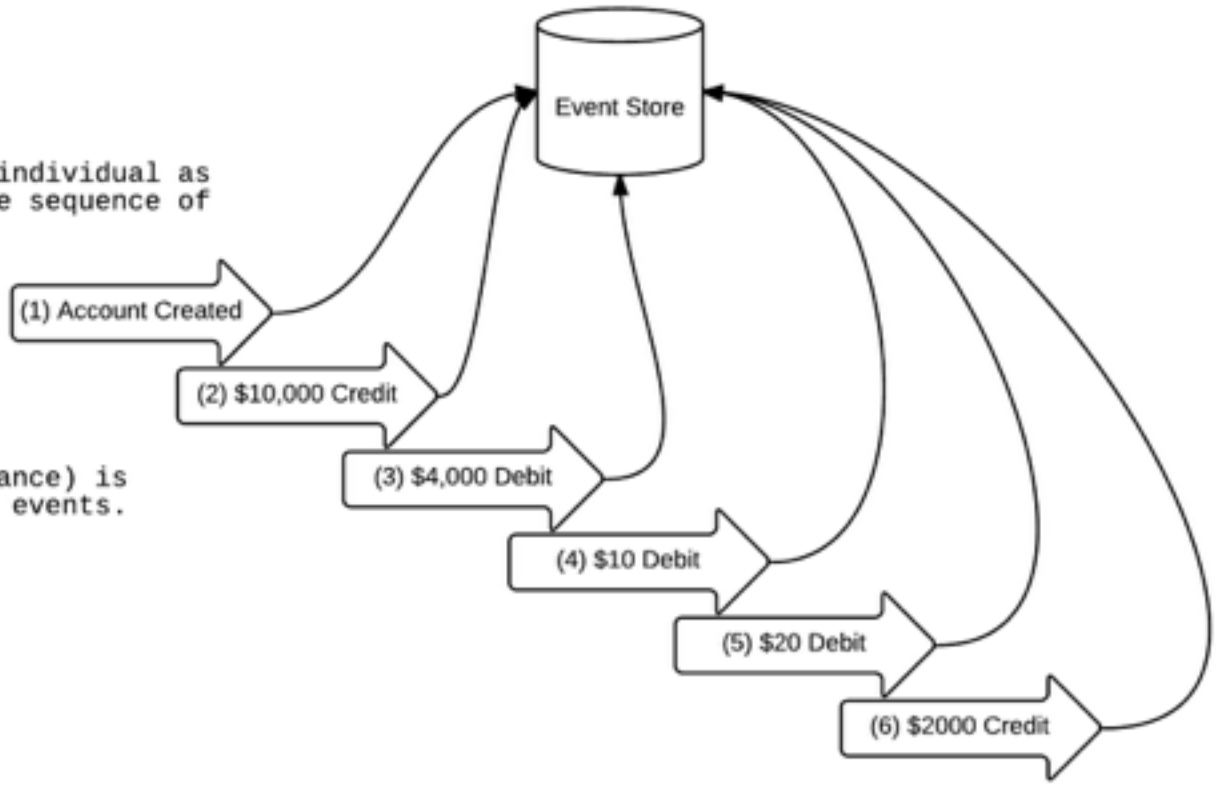
- We persist each transaction as an **independent** event
- To calculate the balance, the **delta** of the current transaction is **applied** to the last known value
- We have a **verifiable** audit log that can be **reconciled** to ensure **validity**
- The current balance at **any** point can be **derived**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

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.
```



PersistentActor

- Persistent, **stateful** actor that can persist events to a journal
- Reacts to them in a **thread-safe** manner
- Can be used to implement **both** command and event sourcing
- When restarted, journaled messages are **replayed**
- The actor **recovers** the internal state from these messages

Journal

- Stores the sequence of **messages** sent to a persistent actor
- Application **controls** which messages are journaled
- Application **controls** which messages are not journaled
- The storage backend of a journal is **pluggable**
- The default journal storage plugin writes to the local filesystem
- Replicated journals are **available** as Community Plugins

Snapshots

- A snapshot stores a **“moment-in-time”**
- It is **internal state** of the actor
- Used for **optimizing** recovery times
- The storage backend of a snapshot store is **pluggable**.
- The default snapshot plugin writes to the local filesystem.
- Replicated snapshots are **available** as Community Plugins

Event Handler (Internal 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!
    . . .
}
```

Event Handler (Persist)

```
class Client extends PersistentActor {

  . . .

  val receiveCommand: Receive = {

    case cmd: RegisterClient =>

      validateRegistration(cmd) fold (

        f => sender ! f,

        s => persist(s) { e =>  // <- partial function persist

          state = state.update(e)

          // side effects go here

      )

    . . .

}
```

Event Handler (Recover)

```
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
}
```

Outline

1. Introduction
2. The “C” in CQRS
3. Event Sourcing
4. The “Q” in CQRS
5. Consistency
6. Conclusion

what are

queries?

query | kwi(ə)rē |

noun

- a question, especially one addressed to an official or organization

CRUD & ORM’s = ???

CRUD & ORM's =

CRUD & ORM's = PAIN

- DTO's projected off domain
- Aggregate getters **expose** internal state
- DTO's **different** model than domain
- Usually require extensive mapping
- Large # of read method on repositories
- **Optimization** of queries becomes difficult
- Query objects **not equal** to data model
- Object model translated to data model
- Impedance mismatch

Thin Read Layer

- CQRS applies a **natural** boundary
- DTO’s **no longer** project of the domain
- Reads from the **projection friendly** data store
- Read side **projects** DTO’s
- No need for **complex ORM’s**
- Data stored/fetched in **projection** structure
- No more impedance mismatch
- Easier to **optimize**; much more **responsive**
- No more **looping** to construct view

Query Handler (Side Effects)

```
class ClientHistoryView extends Actor with ActorLogging {
  . . .
  val receive: Receive = {
    case cmd: ClientRegistered =>
      mergeHistory(cmd) fold ( // do other stuff
        f => sender ! f,
        s => persist(s)
    )
    . . .
}
```

Outline

1. Introduction
2. The “C” in CQRS
3. Event Sourcing
4. The “Q” in CQRS
5. Consistency
6. Conclusion

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

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

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

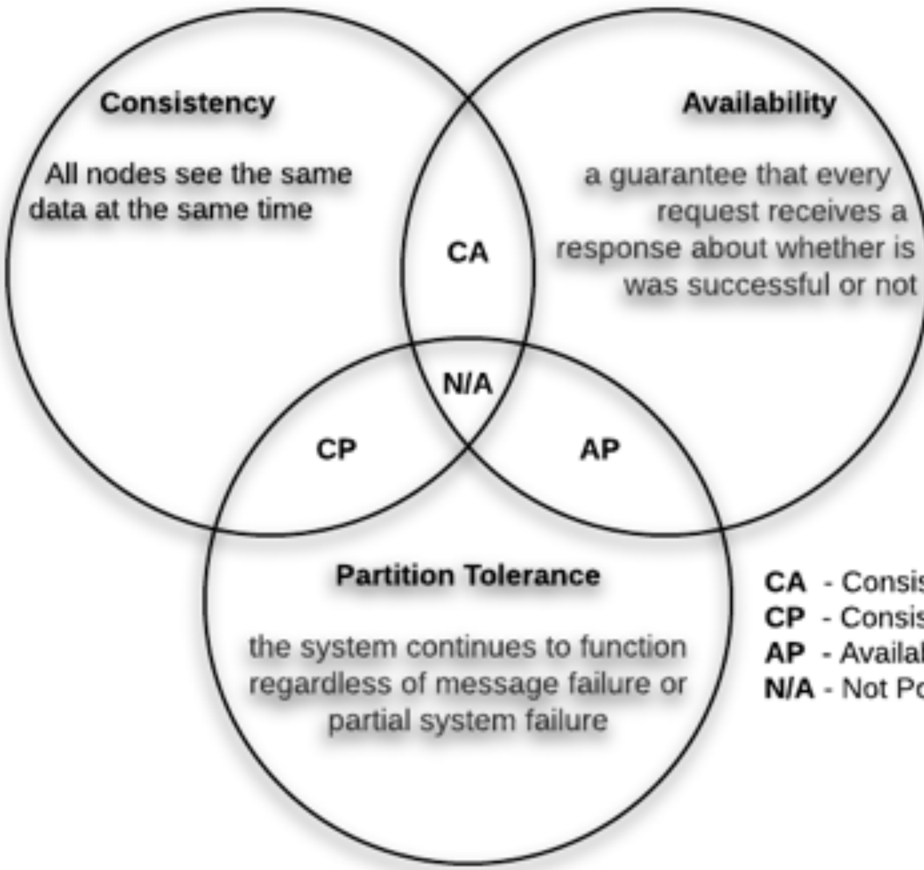
Therefore, strongly consistent systems are not distributable **as a whole contiguous system** as identified by the CAP theorem.

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



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 updated 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 | rezəˈlo͞oSHən |

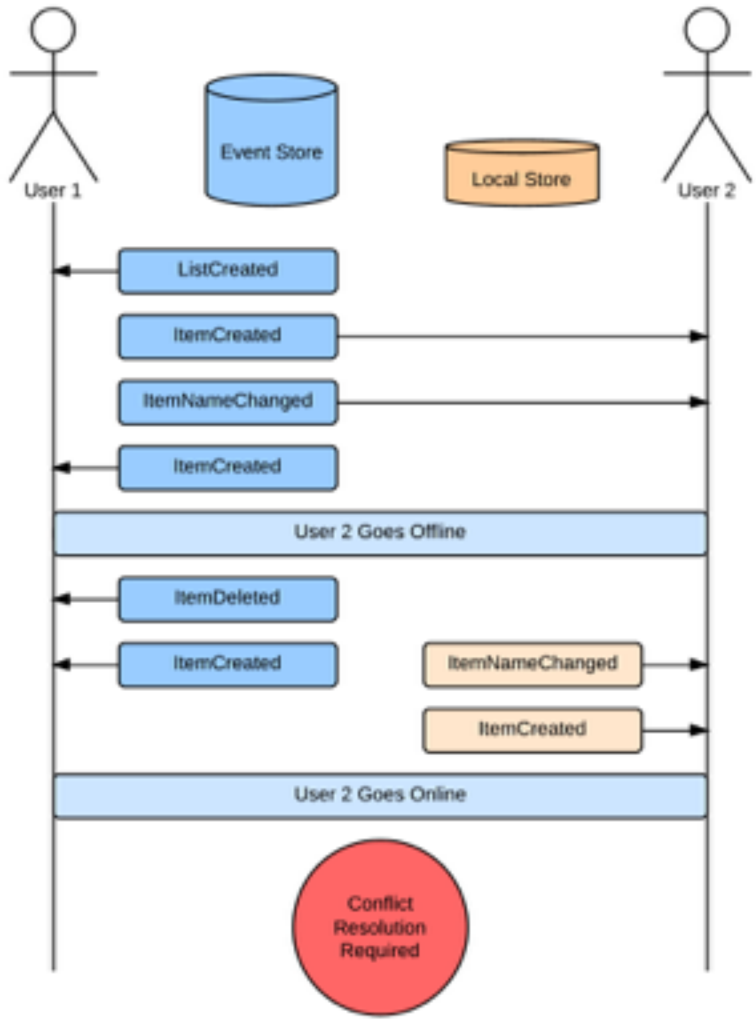
noun

- 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
- The system **compares** the **current state version** to the **version** on the incoming command
- If they are not equal, the command is **rejected**
- **First** writer **wins**.

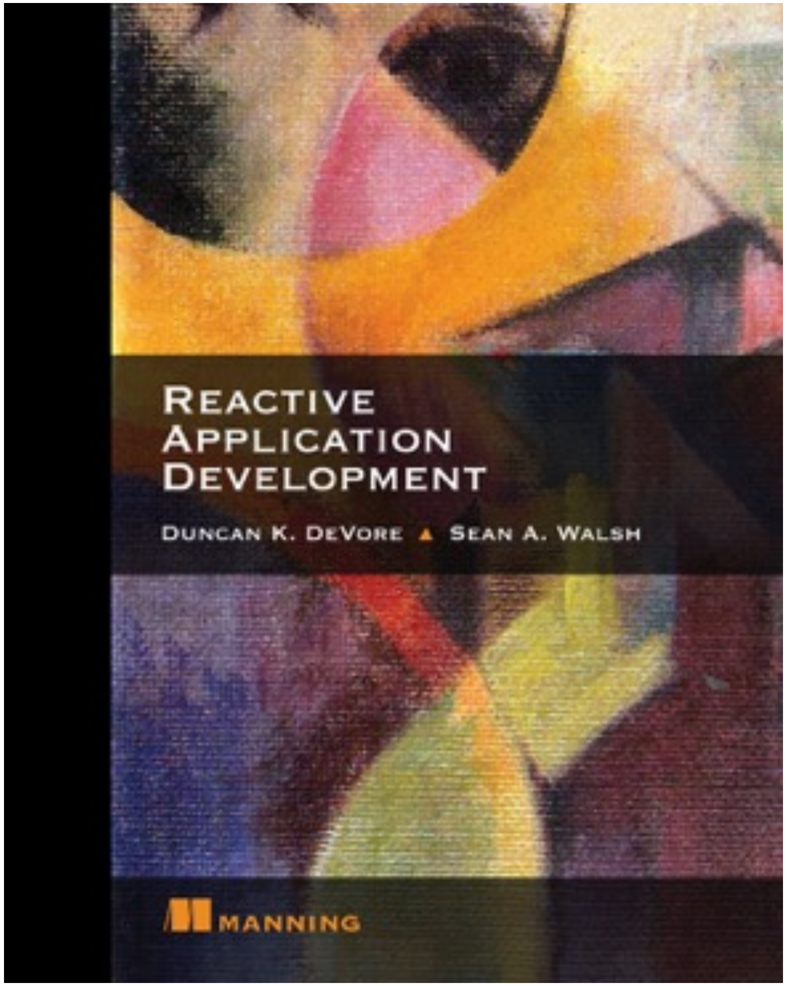
Conflict Resolution

```
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. The “C” in CQRS
3. Event Sourcing
4. The “Q” in CQRS
5. Consistency
6. Conclusion

Reactive Application Development



Questions?

CQRS / ES

Duncan K. DeVore
Typesafe
@ironfish

Typesafe