hochschule mannheim

Understanding Eventual Consistency

MSI Presentation SS2014

Horst Schneider, Patrick Beedgen
Hochschule Mannheim

June 17th, 2014

Introduction

" ...the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value"
–W. Vogels (2009)

Interpretation 1

"When you read data[...], the response might not reflect the results of a recently completed write operation. The response might include some stale data. Consistency across all copies of the data is usually reached within a second; so if you repeat your read request after a short time, the response returns the latest data."

Interpretation 2

"This sort of system we term "single writer eventual consistency". So what are its properties?
(1) A client could read stale data.
(2) The client could see out-of-order write operations. [...] So this is our weakest form of consistency - eventually consistent with out of order reads in the short term."

DynamoDB Documentation

> *"When you read data[...], the response might not reflect the results of a recently completed write operation. The response might include* **some stale data**. *Consistency across all copies of the data is* **usually reached within a second**; *so if you repeat your read request after a short time, the response returns the latest data."*

MongoDB Documentation

> *"This sort of system we term "single writer eventual consistency". So what are its properties?*
> *(1)***A client could read stale data**.
> *(2)The client could see out-of-order write operations.[...]*
> *So this is our weakest form of consistency - eventually consistent with* **out of order reads** *in the short term."*

Problems

- disparate and low-level formalisms
  consistency model is tied to system implementation

Problems

- disparate and low-level formalisms
  consistency model is tied to system implementation
- weak guarantees
  in realistic scenarios updates **never** *stop*

Problems

- disparate and low-level formalisms
  consistency model is tied to system implementation
- weak guarantees
  in realistic scenarios updates **never** *stop*
- conflict resolution policies
  resolution of conflicts in multiple replicas

Problems

- disparate and low-level formalisms
  consistency model is tied to system implementation
- weak guarantees
  in realistic scenarios updates **never** *stop*
- conflict resolution policies
  resolution of conflicts in multiple replicas
- combinations of different consistency levels
  strong consistency may be needed at certain times

Problems

- disparate and low-level formalisms
  consistency model is tied to system implementation
- weak guarantees
  in realistic scenarios updates **never** *stop*
- conflict resolution policies
  resolution of conflicts in multiple replicas
- combinations of different consistency levels
  strong consistency may be needed at certain times

$\Rightarrow$ some sort of formalism is needed to define semantics of Eventual Consistency

Agenda

1. Replicated Data Types
2. Axiomatic Specification Framework
3. Consistency Strengthening Interfaces
4. Conclusion / Discussion

Replicated Data Types

- a replicated database stores **objects** $\mathrm{Obj} = \{x, y, \dots\}$

Replicated Data Types

- a replicated database stores **objects** $\mathrm{Obj} = \{x, y, \dots\}$
- every object $x \in \mathrm{Obj}$ has
  - a **value** $\in \mathrm{Val}$
  - a **type** type$(x) \leftrightarrow \tau$
  - **operations** $\mathrm{Op}_{\mathrm{type}(x)}$ that a client can perform on it

Replicated Data Types

- a replicated database stores **objects** $\mathrm{Obj} = \{x, y, \dots\}$
- every object $x \in \mathrm{Obj}$ has
  - a **value** $\in \mathrm{Val}$
  - a **type** type$(x) \leftrightarrow \tau$
  - **operations** $\mathrm{Op}_{\mathrm{type}(x)}$ that a client can perform on it
- two examples: Int Register **intreg**, Counter **ctr**

$$\mathrm{Op}_{\mathrm{ctr}} = \{\mathrm{rd}, \mathrm{inc}\}$$
$$\mathrm{Op}_{\mathrm{intreg}} = \{\mathrm{rd}, \mathrm{wr}(k) | k \in \mathbb{Z}\}$$

Replicated Data Types

Sequential Data Type Specification

in a *strongly consistent system*, the semantics of a data type can be specified by a function:

$$S_\tau : \mathrm{Op}_\tau^+ \to \mathrm{Val}$$

Replicated Data Types

Sequential Data Type Specification

in a *strongly consistent system*, the semantics of a data type can be specified by a function:

$$S_\tau : \mathrm{Op}_\tau^+ \to \mathrm{Val}$$

example:

$$S_{\mathrm{intreg}}(\sigma\ \mathrm{wr}(k)) = S_{\mathrm{ctr}}(\sigma\ \mathrm{inc}) = \bot;$$

(e.g. $\sigma = \{\mathrm{rd}\ \mathrm{rd}\ \mathrm{wr}(5)\ \mathrm{wr}(6)\ \mathrm{rd}\}$ or $\sigma = \{\mathrm{rd}\ \mathrm{rd}\ \mathrm{inc}\ \mathrm{inc}\ \mathrm{rd}\}$)

Replicated Data Types
Sequential Data Type Specification

in a *strongly consistent system*, the semantics of a data type can be specified by a function:

$$S_\tau : \mathrm{Op}_\tau^+ \rightarrow \mathrm{Val}$$

example:

$$\begin{aligned} S_{\mathrm{intreg}}(\sigma\ \mathrm{wr}(k)) = S_{\mathrm{ctr}}(\sigma\ \mathrm{inc}) = \bot; \\ S_{\mathrm{ctr}}(\sigma \mathrm{rd}) = (\text{number of inc operations in } \sigma); \end{aligned}$$

(e.g. $\sigma = \{\mathrm{rd\ rd\ wr}(5)\ \mathrm{wr}(6)\ \mathrm{rd}\}$ or $\sigma = \{\mathrm{rd\ rd\ inc\ inc\ rd}\}$)

in a *strongly consistent system*, the semantics of a data type can be specified by a function:

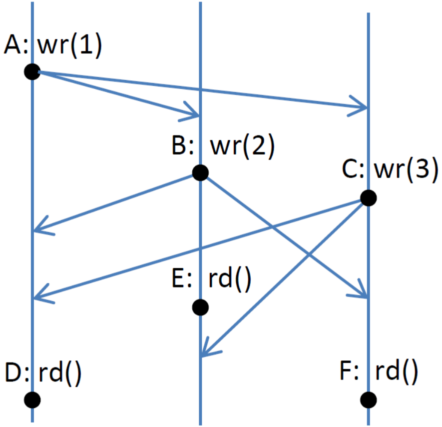
$$S_\tau : \mathrm{Op}_\tau^+ \rightarrow \mathrm{Val}$$

example:

$$\begin{aligned} S_{\mathrm{intreg}}(\sigma\ \mathrm{wr}(k)) = S_{\mathrm{ctr}}(\sigma\ \mathrm{inc}) &= \bot; \\ S_{\mathrm{ctr}}(\sigma \mathrm{rd}) &= (\text{number of inc operations in } \sigma); \\ S_{\mathrm{intreg}}(\sigma \mathrm{rd}) &= k;\ \text{if } \mathrm{wr}(0)\sigma = \sigma_1 \mathrm{wr}(k) \sigma_2 \text{ and} \\ &\quad\ \sigma_2 \text{ does not contain wr operations} \end{aligned}$$

(e.g. $\sigma = \{\mathrm{rd\ rd\ wr}(5)\ \mathrm{wr}(6)\ \mathrm{rd}\}$ or $\sigma = \{\mathrm{rd\ rd\ inc\ inc\ rd}\}$)

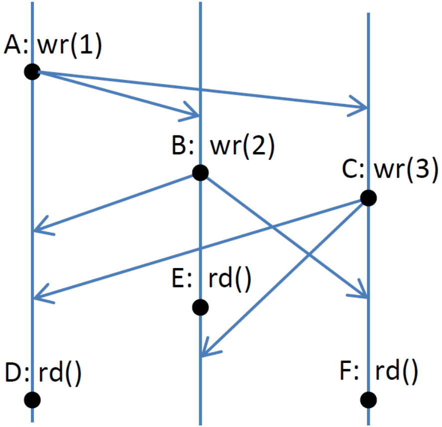
- semantics of eventually consistent systems are harder to formalize
- concurrent operations on the same object happen on multiple replicas
- each replica executes operations immediately, updating other replicas later → **conflicts**
- different conflict resolution strategies for replicated data types



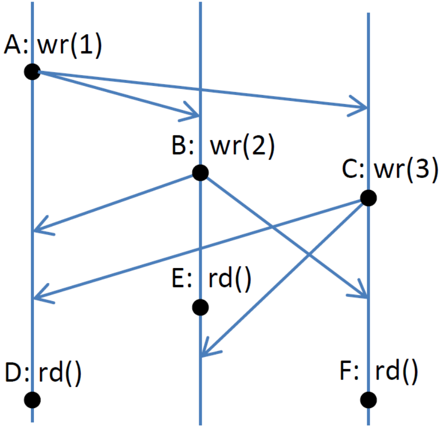
A: wr(1)
B: wr(2)
C: wr(3)
E: rd()
D: rd()
F: rd()

Replicated Data Types

Conflict Resolution Strategies



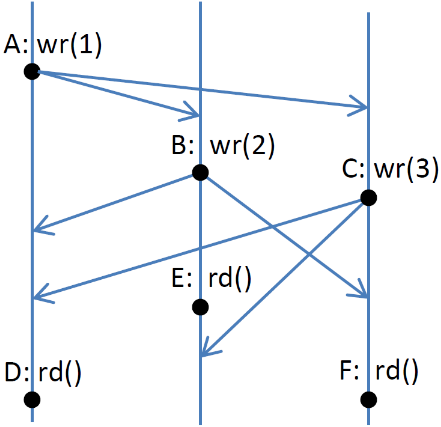
1. make concurrent operations commutative



1. make concurrent operations commutative
2. order concurrent operations

Replicated Data Types

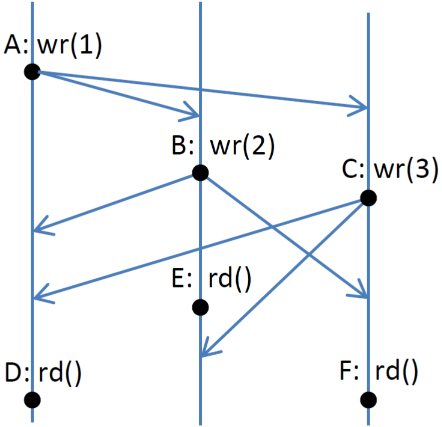
Conflict Resolution Strategies



1. make concurrent operations commutative
2. order concurrent operations
3. flag conflicts (let the user decide)

Replicated Data Types

Conflict Resolution Strategies



1. make concurrent operations commutative
2. order concurrent operations
3. flag conflicts (let the user decide)
4. resolve conflicts semantically

- S_τ is not strong enough to formalize these strategies
- **visibility** and **order** of preceding operations have to be included

- S_τ is not strong enough to formalize these strategies
- **visibility** and **order** of preceding operations have to be included
- F_τ: takes an **operation context** C and returns a **value**

$$F_\tau(C) \in \mathrm{Val}$$

- S_τ is not strong enough to formalize these strategies
- **visibility** and **order** of preceding operations have to be included
- F_τ: takes an **operation context** C and returns a **value**

$$F_\tau(C) \in \mathrm{Val}$$

- C adds **visibility** and **arbitration relations** to preceding operations:

$$C = (f, V, \mathrm{ar}, \mathrm{vis})$$

- S_τ is not strong enough to formalize these strategies
- **visibility** and **order** of preceding operations have to be included
- F_τ: takes an **operation context** C and returns a **value**

$$F_\tau(C) \in \mathrm{Val}$$

- C adds **visibility** and **arbitration relations** to preceding operations:

$$C = (f, V, \mathrm{ar}, \mathrm{vis})$$
$$u \xrightarrow{\mathrm{vis}} v, \mathrm{vis} \subseteq V \times V$$
$$u \xrightarrow{\mathrm{ar}} v, \mathrm{ar} \subseteq V \times V$$

example: Strategy **Make Concurrent Calls Commutative**

$$F_{\mathrm{ctr}}(\mathrm{inc}, V, \mathrm{vis}, \mathrm{ar}) = \perp;$$
$$F_{\mathrm{ctr}}(\mathrm{rd}, V, \mathrm{vis}, \mathrm{ar}) = (\text{the number of inc operations in } V);$$

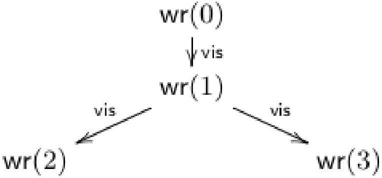
example: Strategy **Make Concurrent Calls Commutative**

$$F_{\mathrm{ctr}}(\mathrm{inc}, V, \mathrm{vis}, \mathrm{ar}) = \bot;$$
$$F_{\mathrm{ctr}}(\mathrm{rd}, V, \mathrm{vis}, \mathrm{ar}) = (\text{the number of inc operations in } V);$$

example: Strategy **Order Concurrent Operations**

$$F_{\mathrm{intreg}}(f, V, \mathrm{vis}, \mathrm{ar}) = S_{\mathrm{intreg}}(V^{\mathrm{ar}} f)$$
$$(S_\tau \ : \ \mathrm{Op}_\tau^+ \to \mathrm{Val})$$

example: Strategy **Flag Conflicts**



writes on a multi value register

Axiomatic Specification Framework

Session and Action

- a single client may do several changes to the same object
- **sessions** provide a way to track client identity for operations
- an **action** is a tuple $(e, s, [x.f : k])$
  - e: unique identifier
  - s: session id $\in \mathrm{SId}$
  - $[x.f : k]$: object, operation and return value

Axiomatic Specification Framework

Session and Action

- a single client may do several changes to the same object
- **sessions** provide a way to track client identity for operations
- an **action** is a tuple $(e, s, [x.f : k])$
  - e: unique identifier
  - s: session id $\in \mathrm{SId}$
  - $[x.f : k]$: object, operation and return value

example:

$$a = (1af3c, 17, [x.rd : k]);\ \mathrm{type}(x) = \mathrm{intreg}$$

Axiomatic Specification Framework

History and Execution

- the set of all actions that happen in a database is denoted as Act
- a **history** (A, so) is a set of actions $A \subseteq \mathrm{Act}$ and a **session order** relation $\mathrm{so} \subseteq A \times A$
- an **execution** $X = (A, \mathrm{so}, \mathrm{vis}, \mathrm{ar})$ enhances the history with visibility and arbitration relations
- we can now extract an operation context for any action in any session, providing a deterministic return value

Axiomatic Specification Framework

Levels of Eventual Consistency

- with replicated data types we can define multiple forms of Eventual Consistency
  - Basic Eventual Consistency
  - Session Guarantees
  - Causality
- every form contains multiple axioms
- more axioms mean stronger consistency

Axiomatic Specification Framework

Basic Eventual Consistency Axioms

- axioms a database implementation has to enforce to offer **basic eventual consistency**
- Well Formedness Axioms
  - SOwf: **so** *is the union of transitive, irreflexive and total orders on actions by each session*
  - VISwf: $\forall a, b.\ a \xrightarrow{\text{vis}} b \Rightarrow obj(a) = obj(b)$
  - ARwf: $\forall a, b.\ a \xrightarrow{\text{ar}} b \Rightarrow obj(a) = obj(b)$

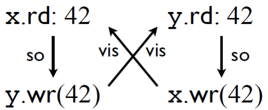
Axiomatic Specification Framework

Basic Eventual Consistency Axioms

- Basic Eventual Consistency axioms:
  - THINAIR: $\mathrm{so} \cup \mathrm{vis}$ is anticyclic

- Basic Eventual Consistency axioms:
    - THINAIR: $\mathrm{so} \cup \mathrm{vis}$ is anticyclic

not possible in THINAIR:

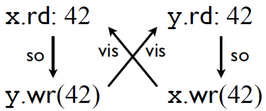


Axiomatic Specification Framework

Basic Eventual Consistency Axioms

- Basic Eventual Consistency axioms:
  - THINAIR: $\mathrm{so} \cup \mathrm{vis}$ is anticyclic

    not possible in THINAIR:



  - EVENTUAL:
    $\forall a \in A.\ \neg(\exists \text{infinitely many } b \in A.\ \mathrm{same}(a, b)) \wedge \neg(a \xrightarrow{\mathsf{vis}} b))$

TODO: Image explaining photo/noboss example from paper

Axiomatic Specification Framework

Session guarantees

- with basic eventual consistency we still might be reading values out of order
- axioms that formalise that all operation within a session keep the current context consistent:
  - Read Your Writes: *An operation sees all previous operations by the same session*
  - Writes Follow Reads in Visibility: *Arbitration orders an operation after other operations previously seen by the same session*
  - ... etc.

Axiomatic Specification Framework

Causality Axioms

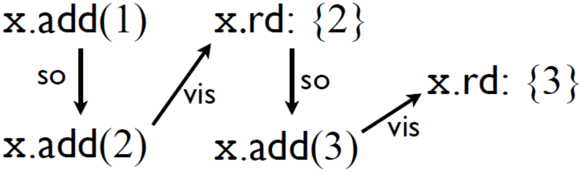
- Per-object-causal-visibility: *POCV guarantees that an operation sees all operations on the same object that causally affect it*

Axiomatic Specification Framework

Causality Axioms

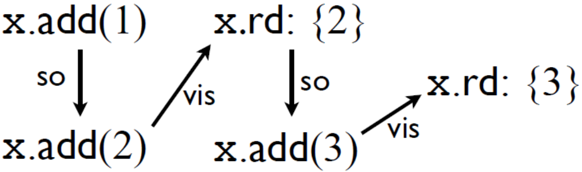
- Per-object-causal-visibility: *POCV guarantees that an operation sees all operations on the same object that causally affect it*

not allowed in POCV:



- Per-object-causal-visibility: *POCV guarantees that an operation sees all operations on the same object that causally affect it*

not allowed in POCV:



- Per-object-causal-arbitration: *POCA correspondingly restricts the arbitration relation*

Consistency Strengthening Interfaces

Online Shopping

- almost everybody does online shopping
  - we put items in our shopping cart
  - we pay them
  - we continue shopping.. or not

Consistency Strengthening Interfaces

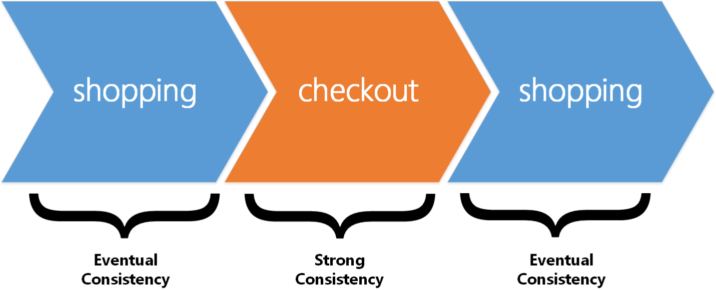
Online Shopping

- almost everybody does online shopping
  - we put items in our shopping cart
  - we pay them
  - we continue shopping.. or not

Consistency Strengthening Interfaces

Online Shopping

- almost everybody does online shopping
  - we put items in our shopping cart
  - we pay them
  - we continue shopping.. or not



Consistency Strengthening Interfaces

Consistency Annotations

- every **action** accepted by the database has to marked with a **consistency annotation**

$$(e, s, [x.f_{\mu} : k])\mu \in \{ORD, CSL\}$$

- either **ordinary** or **causal**
- ordinary actions behave like we defined previously
- causal actions make all operations performed before the annotations visible to all previous actions

Consistency Strengthening Interfaces

Fences

- instead of annotating every single action, a **fence** can be used
- a **fence** is an **action** where the executing replica forces all its updates on every other replica in the cluster

$$\text{action a} = (e, s, \mathrm{fence})$$

- the execution of other actions is halted until all replicas acknowledge the receipt

Consistency Strengthening Interfaces

Fences

- instead of annotating every single action, a **fence** can be used
- a **fence** is an **action** where the executing replica forces all its updates on every other replica in the cluster

$$\text{action a} = (e, s, \mathrm{fence})$$

- the execution of other actions is halted until all replicas acknowledge the receipt
- the database behaves like a **CP** database !

Conclusion

- the paper provides a formal way to **precisely specify eventually consistent systems**
- **Every aspect of a system is covered**, from data types to client interaction
- specifications are **independent of implementation details**
- still **very theoretical**, no tools available to map between specifications and implementation
- the framework is **not suitable for programmers**, as it is very abstract and not easily understandable and applicable
- the paper is still "work in progress"

Discussion