TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

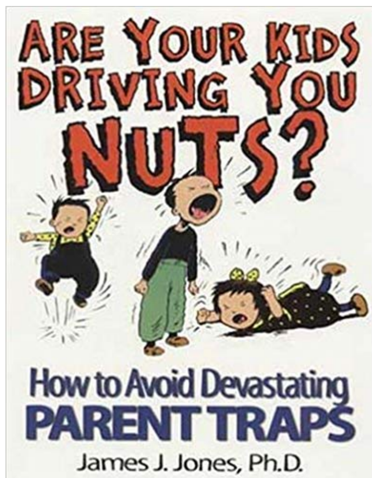
Programming Distributed Systems

Consistency and Conflict-free Replication

Annette Bieniusa

FB Informatik
TU Kaiserslautern

KIDS OUT OF CONTROL?



Inconsistency might be the problem!

Overview

- What is consistency?
- How can we define and distinguish between different notions of consistency?
- How can we keep replicated data consistent under concurrent updates?
- What implications does a consistency model have for an application?

Goals of this Learning Path

In this learning path, you will learn

- to compare formal declarative models for different types of consistency
- to relate sequential and concurrent semantics of register and set data types
- to translate space-time diagrams to event graphs
- to distinguish different conflict resolution strategies of replicated data types
- to explain the pros and cons of state- vs operation-based replication strategies for replicated data types

Consistency

Consistency

- Distributed systems: "Consistency" refers to the observable behaviour of a system (e.g. a data store).
- Consistency model defines the correct behavior when interacting with the system.

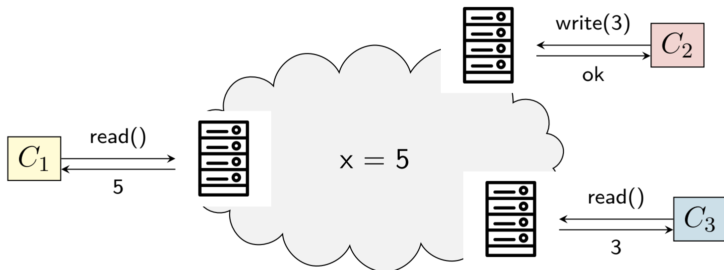
Remark: Consistency in Database systems

- The distributed systems and database communities also use the term "consistency", but with different meanings.
  - C in ACID
  - Refers to the property that application code is sequentially safe
- What we discuss here, is closer to "isolation"

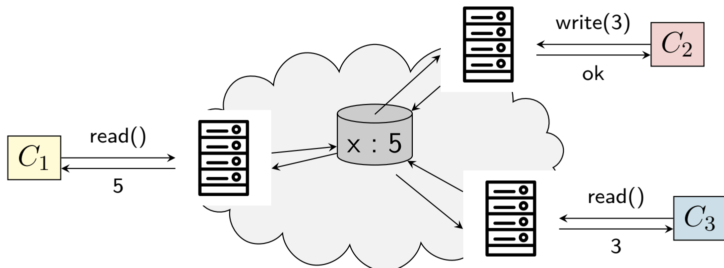
All material and graphics in this section are based on material by Sebastian Burkhardt (Microsoft Research)[2] and the survey by Paolo Viotti and Marko Vukolic [5].

Example: Shared Register

- Operations on registers
  - $rd() \to v$
  - $wr(v) \to ok$
- System architecture:

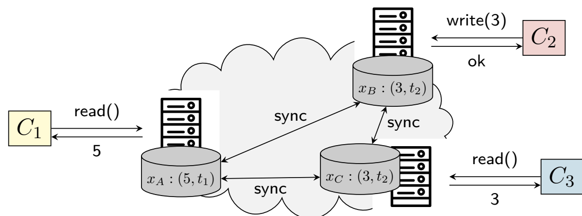


Implementation 1: Single-copy Register



- Single replica of shared register
- Forward all read and write requests

Implementation 2: Epidemic Register



- Each replica stores a timestamped value
- Reads return the currently stored value; writes update this value, stamped with current time (e.g. logical clock)
- At random times, replicas send stored timestamped value to arbitrary subset of replicas
- When receiving timestamped value, replica replaces locally stored value if incoming timestamp is later

Question

Can clients observe a difference between the two implementations (single-copy vs. epidemic)?

Assumptions:

- Asynchronous communication
- Fairness of transport
- “Randomly” generated values

Question

Can clients observe a difference between the two implementations (single-copy vs. epidemic)?

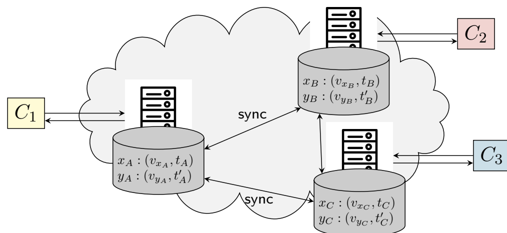
Assumptions:

- Asynchronous communication
- Fairness of transport
- "Randomly" generated values

Notions:

- Single-Copy Register: **Linearizability**
- Epidemic Register: **Sequential Consistency**

Consistency for key-value stores



When generalized to key-value stores (i.e. collection of registers), the epidemic variant guarantees

- **Eventual Consistency** (if sending a randomly selected tuple in each message)
- **Causal Consistency** (if sending all tuples in each message).

Consistency model

- Required for any type of storage (system) that processes operations concurrently.
- Unless the consistency model is linearizability (= single-copy semantics), applications may observe non-sequential behaviors (often called **anomalies**).
- The set of possible behaviors, and conversely of possible anomalies, constitutes the consistency model.

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Consistency specifications

What is a replicated shared object / service?

- Examples: REST Service, file system, key-value store, counters, registers, ...
- Formally specified by a set of operations Op and either
  - a sequential semantics S, or
  - a concurrent semantics F

Sequential semantics

$$S : Op^* \times Op \rightarrow Val$$

- Sequence of all prior operations represents current state (with default initial value)
- Operation to be performed
- Returned value

Example: Register

$S(\epsilon, rd()) = undef$ (read without prior write is undefined)

$S(wr(2) \cdot wr(8), rd()) = 8$ (read returns last value written)

$S(rd() \cdot wr(2) \cdot wr(8), wr(3)) = ok$ (write always returns ok)

Sequential semantics

$$S : Op^* \times Op \rightarrow Val$$

- Sequence of all prior operations represents current state (with default initial value)
- Operation to be performed
- Returned value

Example: Register

$S(\epsilon, rd()) = undef$ (read without prior write is undefined)

$S(wr(2) \cdot wr(8), rd()) = 8$ (read returns last value written)

$S(rd() \cdot wr(2) \cdot wr(8), wr(3)) = ok$ (write always returns ok)

But what about the semantics under concurrency?

Histories

A history records all the interactions between clients and the system:

- Operations performed
- Indication whether operation successfully completed and corresponding return value
- Relative order of concurrent operations
- Session of an operation (corresponds to client / connection)

Concurrent semantics

Classically, histories are represented as sequences of calls and returns[3].

A	call wr(1)
A	ret ok
A	call rd
B	call wr(333)
B	ret ok
B	call rd
A	ret 1
C	call rd
B	ret 333
C	ret 1

Event graphs

Event graphs

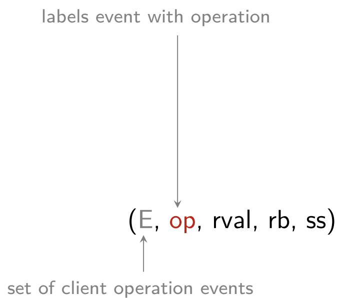
wr(1)

wr(3)

rd()

rd()

rd()



Event graphs

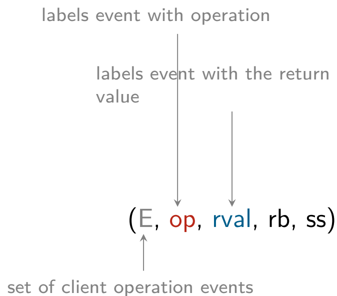
wr(3):ok

rd() :3

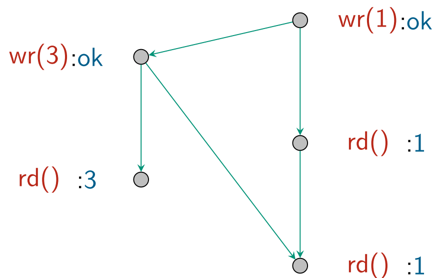
wr(1):ok

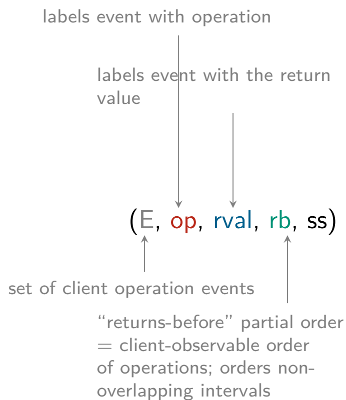
rd() :1

rd() :1

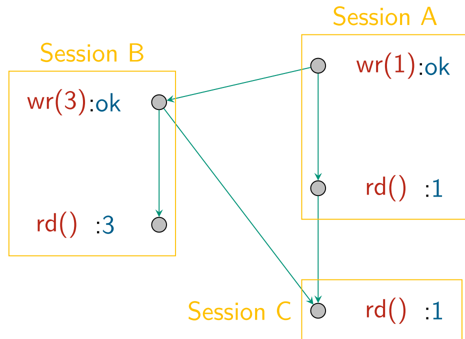


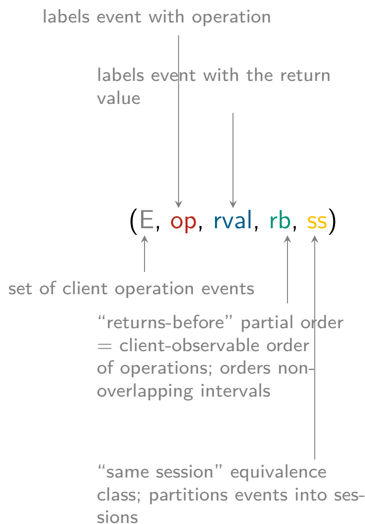
Event graphs





Event graphs





Event graphs

An event graph represents an execution of a system.

- **Vertices**: events
- **Attributes**: label for vertices with information on the corresponding event (e.g. which operation, parameters, return values)
- **Relations**: orderings or groupings of events

Definition

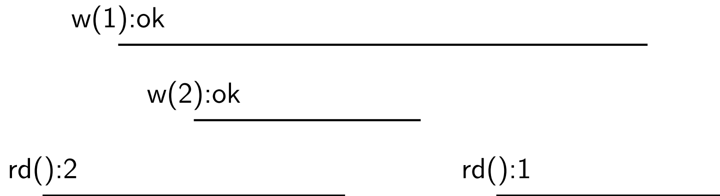
An event graph G is a tuple $(E, d_1, \ldots, d_n)$ where $E \subseteq Events$ is a finite or countably infinite set of events, and each d_i is an attribute or relation over E.

Histories as event graphs

A history is an event graph $(E, op, rval, rb, ss)$ where

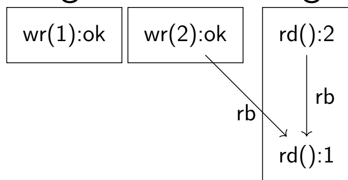
- $op : E \rightarrow Op$ associate operation with an event
- $rval : E \rightarrow Values \cup \{\nabla\}$ are return values (∇ denotes that operation never returns)
- rb is returns-before order
- ss is same-session relation

Hands-on: Timeline diagram vs. event graph



TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Solution: Timeline diagram vs. event graph



Event graph $G = (E, op, rval, rb)$ with

$$
\begin{aligned}
E &= \{a, b, c, d\} \\
op &= \{(a, wr(1)), (b, wr(2)), (c, rd()), (d, rd())\} \\
rval &= \{(a, ok), (b, ok), (c, 2), (d, 1)\} \\
rb &= \{(b, d), (c, d)\} \\
ss &= \{(a, a), (b, b), (c, c), (c, d), (d, d), (d, c)\}
\end{aligned}
$$

When is a history correct / valid?

- Common approach: Require linearizability
  - Insert linearization points between begin and end of operation
  - Semantics of operations must hold with respect to these linearization points
  - Linearization points serves as justification / witness for a history
- Here: Consistency semantics beyond linearizability!

Specifying the Consistency Semantics I

- An *execution* is an account of what happened when executing the implementation
- A *history* defines the observable client interaction
- A *specification* is a "test" on histories
  - But how do we specify such a "test" / predicate?

Operational consistency model

- Provides an abstract reference implementation whose behaviors provide the specifications
- Well-studied methodology for proving correctness (e.g. simulation relations or refinement)
- *Problem:* Typically close to specific concrete implementation technique

Specifying the Consistency Semantics II

- An *abstract execution* is an account of the “essence” of what happened
  - Applicable to many implementations
  - Correctness criterion: History is valid if consistent with an abstract execution satisfying some consistency guarantees
- A *concrete execution* is the account of what happened when executing an actual implementation

Axiomatic consistency model

- Uses logical conditions on histories to define valid behaviors
- Allows to combine different aspects (here: consistency guarantees)

Decomposing abstract executions

- Essence of what happened can be tracked down to two basic responsibilities of the underlying protocol:
  1. **Update Propagation:** All operations must eventually become visible everywhere
  2. **Conflict Resolution:** Conflicting operations must be arbitrated consistently

Visibility

- Relation that determines the subset of operations “visible” to (and potentially influencing) an operation
- Describes relative timing of update propagation and operations

$$a \xrightarrow{vis} b$$

- Effect of operation a is visible to the client performing b
- Updates are concurrent if they are not ordered by visibility (i.e. if they cannot observe each other’s effect)

Arbitration

- Used for resolution of update conflicts (i.e. concurrent updates that do not commute)

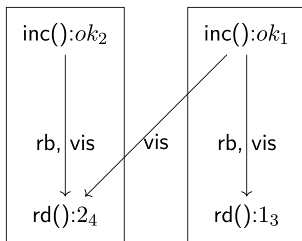
$$a \xrightarrow{ar} b$$

- Total order on operations
- Often solved in practice by using timestamps

Definition: Abstract Executions

An **abstract** execution is an event graph $(E, op, rval, rb, ss, vis, ar)$ such that

- $(E, op, rval, rb, ss)$ is a history
- vis is acyclic
- ar is a total order



Arbitration order: inc():ok_1 $\rightarrow$ inc():ok_2 $\rightarrow$ rd():1_3 $\rightarrow$ rd():2_4

Return Values in Abstract Executions

An abstract execution $(E, op, rval, rb, ss, vis, ar)$ satisfies a sequential semantics S if

$$rval(e) = S(op(e), vis^{-1}.sort(ar))$$

- Observed state = visible operations sorted by arbitration

Consistency guarantee

A consistency guarantee is a predicate or property of an abstract execution.

- Consistency model is collection of all the guarantees needed; histories must be justifiable by an abstraction execution that satisfies them all.
- *Ordering guarantees* ensure that the order of operations is preserved (under certain conditions).
- *Transactions* ensure that operation sequences do not become visible individually.
- *Synchronization operations* can enforce ordering selectively.

Important consistency models: Overview

$$\textbf{Linearizability} = \text{SingleOrder} \wedge \text{Realtime} \wedge \text{RVal}$$

$$\textbf{SequentialConsistency} = \text{SingleOrder} \wedge \text{ReadMyWrites} \wedge \text{RVal}$$

$$\textbf{CausalConsistency} = \text{EventualVisibility} \wedge \text{Causality} \wedge \text{RVal}$$

$$\textbf{BasicEventualConsistency} = \text{EventualVisibility} \wedge \text{NoCircularCausality} \wedge \text{RVal}$$

RVal refers to ReturnValueConsistency

Eventual Consistency (Quiescent Consistency)

In any execution where the updates stop at some point (i.e. where there are only finitely many updates), then eventually (i.e. after some unspecified amount of time) each session converges to the same state.

- Often used in replicated data stores
- In essence: Convergence
- It says nothing about
  - when the replicas will converge
  - what the state is that they will converge to
  - what is allowed in the meantime
  - when there is no phase of quiescence
- Very weak guarantee $\Rightarrow$ Difficult to program against

Eventual visibility

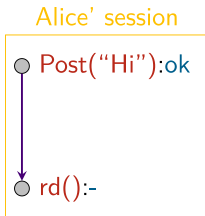
An abstract execution satisfies EventualVisibility if all events become eventually visible.

$$\forall e \in E : |\{e' \in E | (e \xrightarrow{rb} e') \wedge (e \not\xrightarrow{vis} e')\}| < \infty$$

Session guarantees

- When issuing multiple operations in sequence within a session, we usually expect additional properties (**session consistency**)
- Session Order: $so = rb \cap ss$

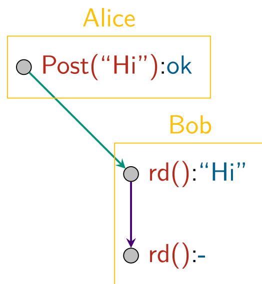
Read My Writes



- It would be confusing if Alice would not see her own message.
- **Fix:** Require that session order implies visibility

$$so \subseteq vis$$

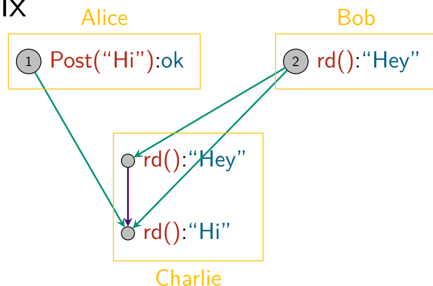
Monotonic Reads



- It would be confusing if Bob read Alice' message, but when he later read again, he would not see the message anymore
- **Fix:** Require that visibility is monotonic with respect to session order

$$vis \circ so \subseteq vis$$

Consistent Prefix



- Alice and Bob post concurrent different values, and the write of Bob is arbitrated after the update of Alice.
- Charlie reads and sees Bob's message; then later, in the same session, he only sees the "earlier" message of Alice.
- **Fix:** Require that remote operations become visible after all operations that precede them in arbitration order

$$ar \circ (vis \cap \neg ss) \subseteq vis$$

Causality Guarantees

- Axiomatic definition of happens-before relation:

$$hb = ((rb \cap ss) \cup vis)^+$$

  - Captures session order and transitive closure of session order and visibility
- NoCircularCausality: $acyclic(hb)$
- CausalVisibility: $hb \subseteq vis$
- CausalArbitration: $hb \subseteq ar$
- Causality: CausalVisibility $\wedge$ CausalArbitration

Causal Consistency

- Strongest model that can implemented in such a way as to be available even under (network) partitions
- Causal consistency implies all session guarantees with the exception of Consistent Prefix.

$\textbf{CausalConsistency} = \text{EventualVisibility} \wedge \text{Causality} \wedge \text{RVal}$

Strong Models

- Ensure a single global order of operations that determines both visibility and arbitration
- SingleOrder:

$$\exists E' \subseteq rval^{-1}(\nabla) : vis = ar \setminus (E' \times E)$$

- What this means: Arbitration and visibility are the same except for subset E' that represents incomplete operations that are not visible to any other operation.
- Assuming, arbitration order corresponds to (perfect global) timestamps, the SingleOrder implies that:
  1. An operation can only see operations with earlier timestamps.
  2. An operation must see all complete operations with earlier timestamps.

Linearizability vs. Sequential Consistency

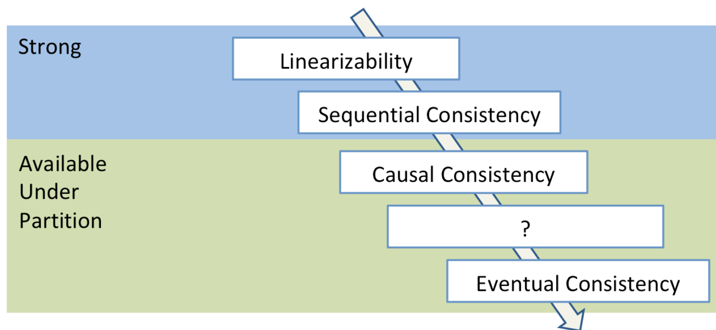
- Linearizability requires RealTime:

$$rb \subseteq ar$$

- Sequential consistency requires ReadMyWrites (restricted to sessions)

- To observe the difference between the two, clients must be able to communicate over some "side channel" that allows them to observe real time ordering.

Conclusion



- In this lecture: Consistency for single operations
- Other aspect: Consistency for groups of operations (**transactions**)
- Open problem: Can we safely mix and match different types of consistency?

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Conflict-free Replicatd Data Types

Motivation

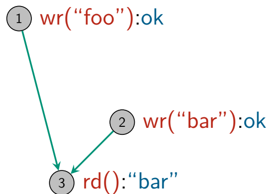
So far, we resolved conflicting updates (i.e. non-commutative updates) simply by sequencing operations using arbitration order (ar).

But sometimes, applications

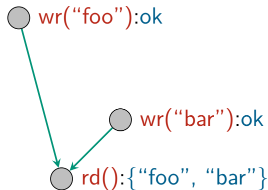
- do not want to depend on a global order such as ar
- want to be made aware of conflicts
- want to resolve conflicts in a specific way

Example: Multi-value register

Standard Register (Last-Writer-Wins)

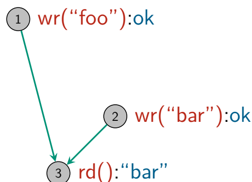


Multi-Value Register



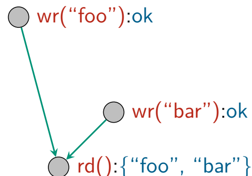
How can we determine the state?

Sequence-based conflict resolution



visible state = **sequence** of visible operations, sorted by arbitration order

General conflict resolution



visible state = **subgraph** of visible operations

Formal model

Before:

$$S : Op^* \times Op \rightarrow Val$$

- "Current state" $Op^* =$ Sequence of all prior operations

Now:

$$F : Op \times C \rightarrow Val$$

- Operation context $C =$ Event graph of visible operations

Revisited: Sequential semantics for registers

$$S : Op^* \times Op \rightarrow Val$$

$S(wr(2) \cdot wr(8), rd()) = 8$ (read returns last value written)

$S(\epsilon, rd()) = undef$

$S(rd() \cdot wr(2) \cdot wr(8), wr(3)) = ok$ (write always returns ok)

Operation Context

An operation context is a finite event graph $C = (E, op, vis, ar)$.

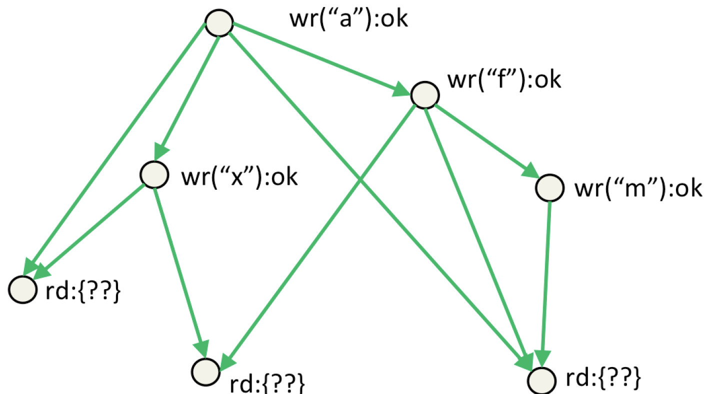
- Events in E capture what prior operations are visible to the operation that is to be performed.
- Models the situation at a single replica

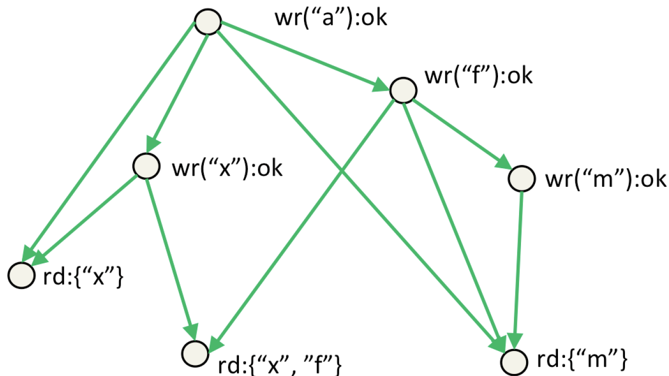
Concurrent semantics for Multi-Value Register

$$F : Op \times C \rightarrow Val$$

$$
\begin{aligned}
F_{mvr}(wr(x), C) &= ok \\
F_{mvr}(rd(), C) &= \{x \mid \text{exists e in C such that } op(e) = wr(x) \\
&\qquad \text{and e is vis-maximal in C}\}
\end{aligned}
$$

Quizz: What do the read ops return?




wr(“a”):ok
wr(“f”):ok
wr(“x”):ok
wr(“m”):ok
rd:{“x”}
rd:{“x”, ”f”}
rd:{“m”}

Return values in Abstract Executions revisited

- Previous lecture:

An abstract execution $(E, op, rval, rb, ss, vis, ar)$ satisfies a sequential semantics S if

$$rval(e) = S(op(e), vis^{-1}.sort(ar))$$

- Read-value consistency can also be defined wrt concurrency semantics

An abstract execution $A = (E, op, rval, rb, ss, vis, ar)$ satisfies a concurrent semantics F if

$$rval(e) = F(op(e), A\mid_{vis^{-1}(e), op, vis, ar})$$

Conflict-free Replicated Data Types (CRDTs) [4]

- Same API as sequential abstract data type, but with concurrency semantics
- Catalogue of CRDTs
  - Register (Laster-writer wins, Multi-value)
  - Set (Grow-Only, Add-Wins, Remove-Wins)
  - Flags
  - Counter (unlimited, restricted/bounded)
  - Graph (directed, monotone DAG)
  - Sequence / List
  - Map, JSON
- If operations are commutative, same semantics as in sequential execution
- Otherwise, need arbitration to resolve conflict

Specification: Replicated counter

- Operation inc commutes $\Rightarrow$ No conflict resolution policy is needed
- Value returned depends only on E and op, but not on vis and ar

$$F_{ctr}(rd(), (E, op, vis, ar)) = |\{e' \in E \mid op(e') = inc\}|$$

Semantics of a replicated Set or How to design a CRDT

- Sequential specification of abstract data type Set S:

$$\{\text{true}\} \quad \text{add(e)} \quad \{e \in S\}$$

$$\{\text{true}\} \quad \text{rmv(e)} \quad \{e \notin S\}$$

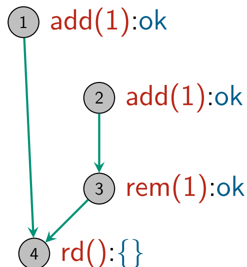
- The following pairs of operations are commutative (for two elements e, f and $e \neq f$):
  - {true} add(e); add(e) $\{e \in S\}$
  - {true} add(e); add(f) $\{e, f \in S\}$
  - {true} rmv(e); rmv(e) $\{e \notin S\}$
  - {true} rmv(e); rmv(f) $\{e, f \notin S\}$
  - {true} add(e); rmv(f) $\{e \in S, f \notin S\}$

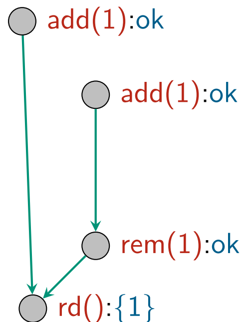
$\Rightarrow$ For these ops, the concurrent execution should yield the same result as executing the ops in any order.

What are the options regarding a concurrency semantics for add(e) and rmv(e)?

- The operations add(e) and rmv(e) are *not* commutative
  - $\{$true$\}$ add(e); rmv(e) $\{e \notin S\}$
  - $\{$true$\}$ rmv(e); add(e) $\{e \in S\}$
- Options for conflict-resolution strategy when *concurrently* executing add(e) and rmv(e)
  - add-wins: $e \in S$
  - remove-wins: $e \notin S$
  - erroneous state (i.e. escalate the conflict to the user)
  - last-writer wins (i.e. define arbitration order through total order, e.g., by adding totally- ordered timestamps)

Set Semantics





Formal Semantics for the Add-Wins Set

$$
\begin{aligned}
F_{aws}(add(x), C) &= ok \\
F_{aws}(rmv(x), C) &= ok \\
F_{aws}(rd(), C) &= \{x \mid \text{exists e in C such that } op(e) = add(x) \\
&\quad \text{and there exists no e' in C such that} \\
&\quad op(e') = rmv(x) \text{ and } e \xrightarrow{vis} e'\}
\end{aligned}
$$

Sets with “interesting” semantics

- Grow-only set
  - Convergence by union on element set
  - No remove operation
- 2P-Set (Wuu & Bernstein PODC 1984)
  - Set of added elements + set of tombstones (= removed elements)
  - Add/remove each element once
  - Problem: Violates sequential spec
- c-set (Sovran et al., SOSP 2011)
  - Count for each element how often it was added and removed
  - Problem: Violates sequential spec

Take a break!

A Mathematician, a Biologist and a Physicist are sitting in a street cafe watching people going in and coming out of the house on the other side of the street. First they see two people going into the house. Time passes. After a while they notice three persons coming out of the house.

The Physicist: “The measurement wasn’t accurate.”.

The Biologist: “They have reproduced”.

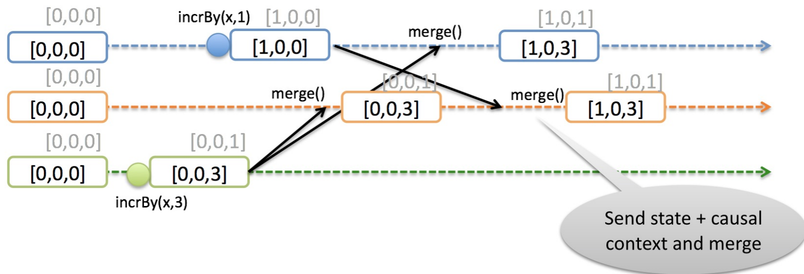
The Mathematician: “If now exactly one person enters the house then it will be empty again.”

CRDTs: Strong Eventual Consistency

- *Eventual delivery:* Every update is eventually applied at all correct replicas
- *Termination:* Update operation terminates
- *Strong convergence:* Correct replicas that have **applied the same update** have equivalent state

How to implement CRDTs

State-based CRDTs: Counter



- Synchronization by propagating replica state
- Updates must inflate the state
- State must form a join semi-lattice wrt merge

$\Rightarrow$ Merge must be idempotent, commutative, associative

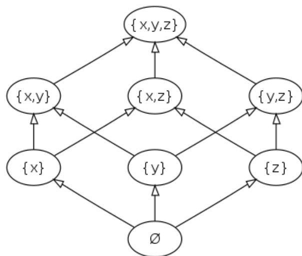
Join-semilattice

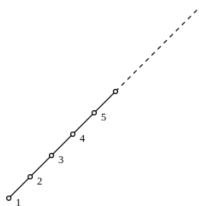
- A join-semilattice S is a set that has a join (i.e. a least upper bound) for any non-empty finite subset:

For all elements $x, y \in S$, the least upper bound (LUB) $x \sqcup y$ exists.

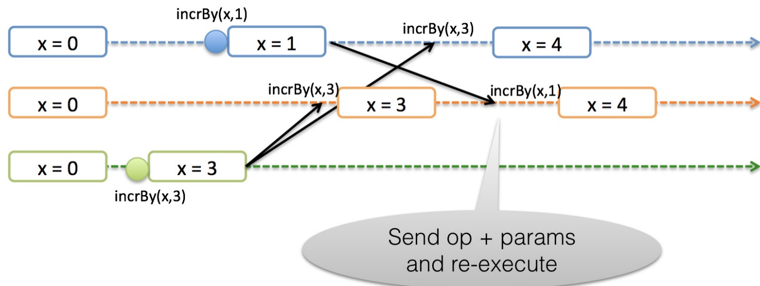
- A semilattice is commutative, idempotent and associative.
- A partial order on the elements of S is induced by setting $x \leq y$ iff $x \sqcup y = y$.

Examples



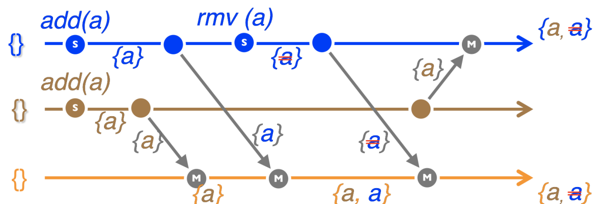


Operation-based CRDTs



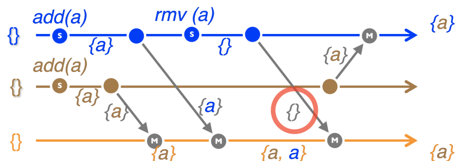
- *Concurrent* updates must commute
- Requires reliable causal delivery for CRDTs with non-commutative operations

Example: Add-wins Set (Observed-remove Set)



{a}.add(a) = {a, a}	*unique*
{a, b}.rmv(a) = {~~a~~, b}	*mark visible instances of a*
{~~a~~, a}.contains(a) | true	∃ *non-marked instance of a*
{a, b}.merge({~~a~~, c}) = {~~a~~, b, c}	*union + marker*

Optimized version of Add-wins Set



- Possible to garbage-collect the tombstone after remove
- Trick: Assuming causal delivery, a removed element will never be re-introduced (with the same id)[1]

Challenges with CRDTs

- Meta-data overhead for CRDTs that require causal contexts
  - Version vectors track concurrent modifications
  - Problematic under churn (i.e. when nodes come and go)
- Monotonically growing state with state-based approach
  - Infeasible for inherently growing data types such as sets, maps, lists with prevalent add
  - When removing elements, often tombstones are required for conflict resolution that relies on concurrency information
  - Requires garbage collection of tombstones when updates become causally stable
- Composability
  - CRDTs can be recursively nested (e.g. Maps, Sequences) or atomically updated in transactions
  - Which type of composability is preferable? What is the semantics of the composed entity?

Delta-based CRDTs

- State-based CRDTs suffer from monotonically growing state (lattice!)
- Op-based CRDTs require reliable causal delivery

Adoption of CRDTs in industry

Conclusion

- CRDTs provide Strong Eventual Consistency (sometimes even more)
- Properties of good conflict resolution
  - Don't loose updates/information!
  - Deterministic (independent of local update order)
  - Semantics close to sequential version
- Meta-data overhead can be substantial

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Further reading I

[1] Annette Bieniusa u. a. „An optimized conflict-free replicated set“. In: *CoRR* abs/1210.3368 (2012). arXiv: 1210.3368. URL: http://arxiv.org/abs/1210.3368.

[2] Sebastian Burckhardt. „Principles of Eventual Consistency“. In: *Foundations and Trends in Programming Languages* 1.1-2 (2014), S. 1–150. DOI: 10.1561/2500000011. URL: https://doi.org/10.1561/2500000011.

[3] Maurice Herlihy und Jeannette M. Wing. „Linearizability: A Correctness Condition for Concurrent Objects“. In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990), S. 463–492. DOI: 10.1145/78969.78972. URL: http://doi.acm.org/10.1145/78969.78972.

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Further reading II

[4] Nuno Preguiça, Carlos Baquero und Marc Shapiro. „Conflict-Free Replicated Data Types (CRDTs)“. In: *Encyclopedia of Big Data Technologies*. Hrsg. von Sherif Sakr und Albert Zomaya. Cham: Springer International Publishing, 2018, S. 1–10. ISBN: 978-3-319-63962-8. DOI: 10.1007/978-3-319-63962-8_185-1. URL: https://doi.org/10.1007/978-3-319-63962-8_185-1.

[5] Paolo Viotti und Marko Vukolic. „Consistency in Non-Transactional Distributed Storage Systems“. In: *CoRR* abs/1512.00168 (2015). arXiv: 1512.00168. URL: http://arxiv.org/abs/1512.00168.