Programming Distributed Systems

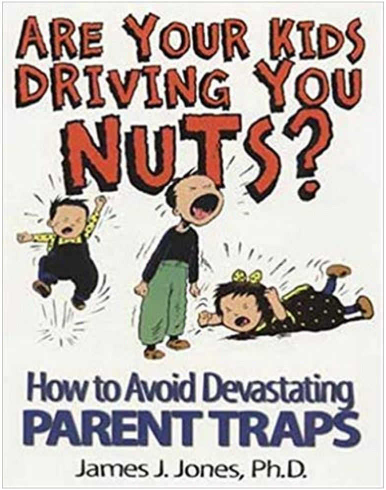
05 Consistency

Annette Bieniusa

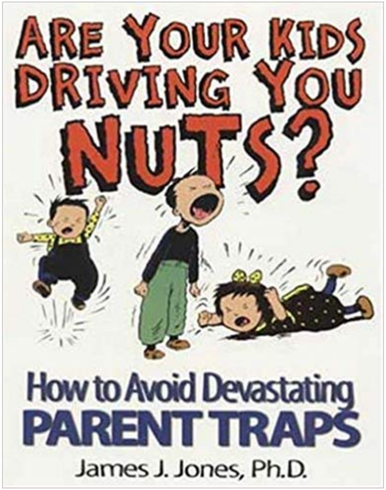
AG Softech
FB Informatik
TU Kaiserslautern

Summer Term 2019

KIDS OUT OF CONTROL?

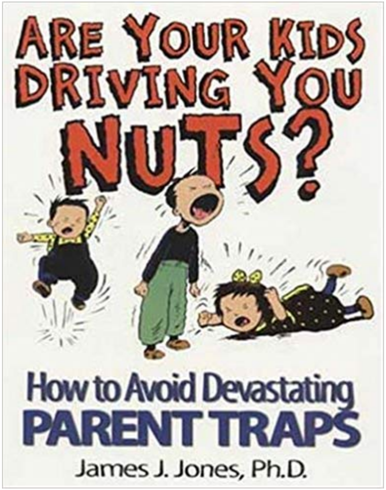

ARE YOUR KIDS DRIVING YOU NUTS?
How to Avoid Devastating PARENT TRAPS
James J. Jones, Ph.D.

KIDS OUT OF CONTROL?


ARE YOUR KIDS DRIVING YOU NUTS?
How to Avoid Devastating PARENT TRAPS
James J. Jones, Ph.D.

Inconsistency might be the problem!

KIDS OUT OF CONTROL?


ARE YOUR KIDS DRIVING YOU NUTS?
How to Avoid Devastating PARENT TRAPS
James J. Jones, Ph.D.

Inconsistency might be the problem!

Roadmap

- What is consistency?
- How can we define and distinguish between different notions of consistency?
- What implications does a consistency model have for an application?

All material and graphics in this section are based on material by Sebastian Burkhardt (Microsoft Research)[1] and the survey by Paolo Viotti and Marko Vukolic [3].

Consistency

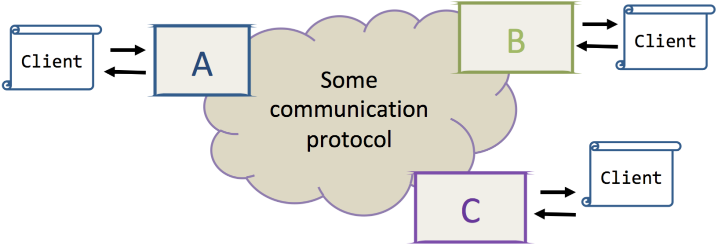
- Distributed systems: “Consistency” refers to the observable behaviour of a system (e.g. a data store).
- Consistency model defines the correct behavior when interacting with the system.

. . . #### Remark: Consistency in Database systems - The

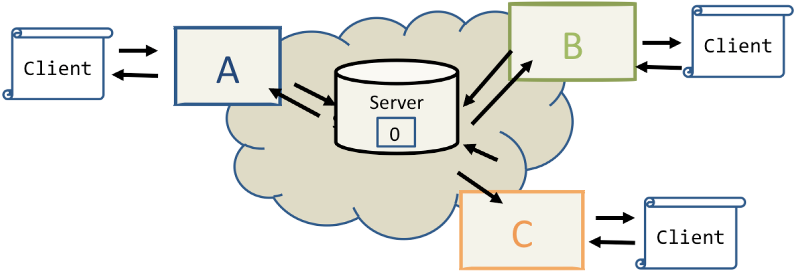
distributed systems and database communities use the same word, consistency, with different meanings. - Roughly the same concept is called “isolation”, whereas the term “consistency” refers to the property that application code is sequentially safe (the C in ACID).

Example: “Single-Value Register”

- Operations $rd() \rightarrow v$ and $wr(v) \rightarrow ok$
- System architecture:

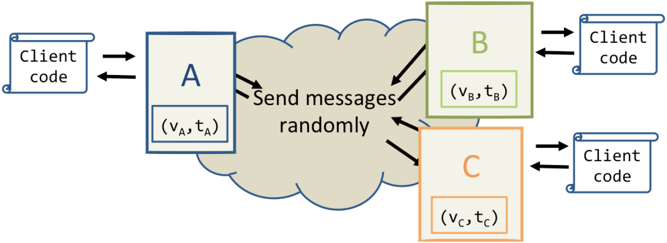


Implementation 1: Single-copy Register



- 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

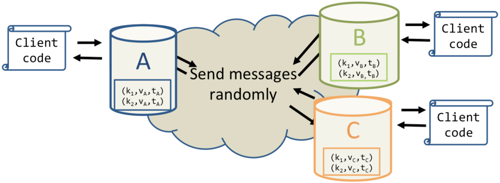
Can clients observe a difference between the two implementations (single-copy vs. epidemic)?

Assumptions:

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

Notions of consistency

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



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) or **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.

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

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

- Sequence of all prior operations ("current state")
- 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)

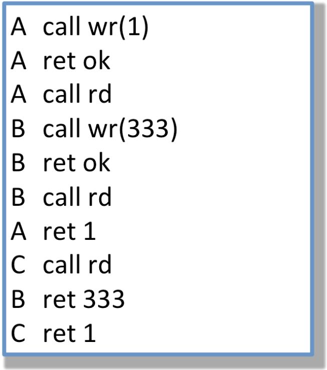
$S(rd \cdot wr(2) \cdot wr(8), wr(3)) = ok$ (write always returns ok)

Histories

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

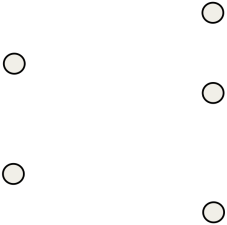
- Operations performed
- Indication whether operation successfully completed and corresponding return value
- Relative order of concurrent operations
- Session of an operation (corresponds to client / connection)

Classically, histories are represented as sequences of calls and returns[2].



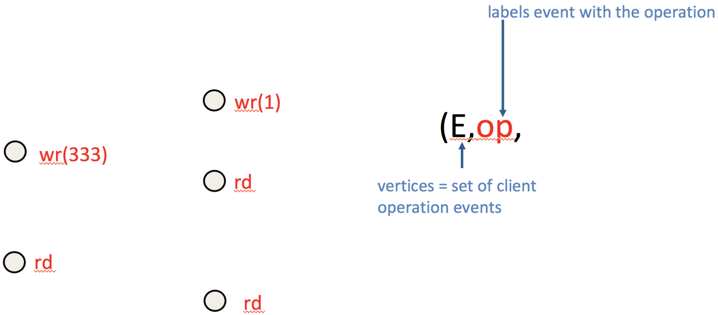
A	call wr(1)
A	ret ok
A	call rd
B	call wr(333)
B	ret ok
B	call rd
A	ret 1
C	call rd
B	ret 333
C	ret 1

⇒ Generalization: **event graphs**



(E,

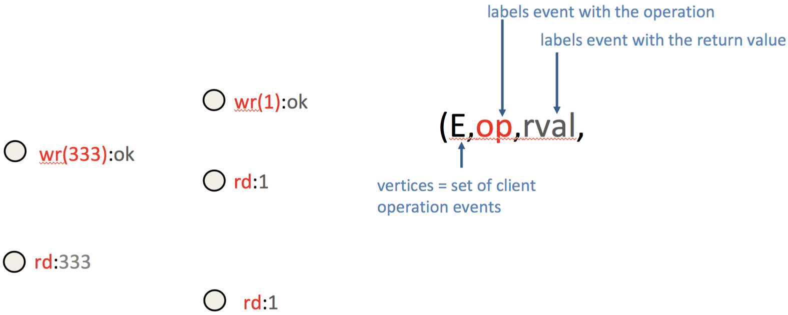
vertices = set of client operation events



labels event with the operation

(E,op,

vertices = set of client operation events



labels event with the operation

labels event with the return value

(E,op,rval,

vertices = set of client operation events

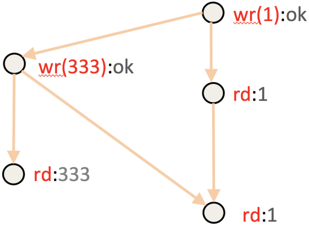
wr(1):ok

wr(333):ok

rd:1

rd:333

rd:1



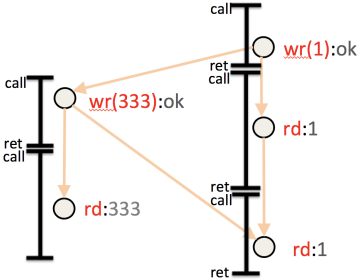
labels event with the operation

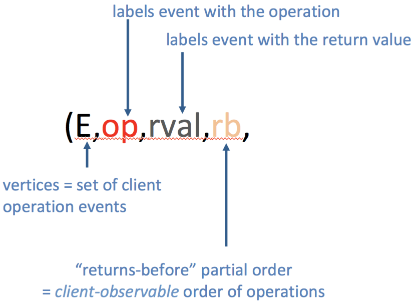
labels event with the return value

(E,op,rval,rb,

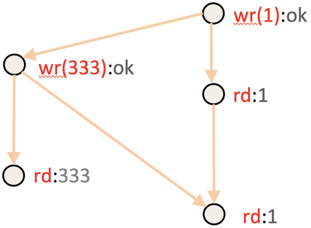
vertices = set of client operation events

“returns-before” partial order
= *client-observable* order of operations
a.k.a “real-time” order




labels event with the operation
labels event with the return value
(E,op,rval,rb,
vertices = set of client
operation events
“returns-before” partial order
= *client-observable* order of operations

a.k.a. real-time order
orders the “non-overlapping intervals”



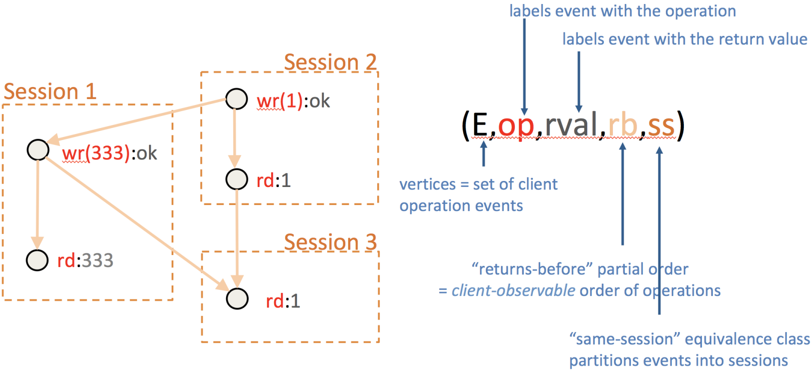
labels event with the operation

labels event with the return value

(E,op,rval,rb,

vertices = set of client operation events

“returns-before” partial order = *client-observable* order of operations



(E,op,rval,rb,ss)

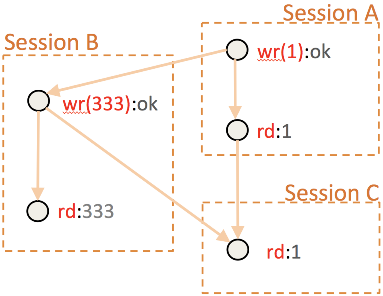
labels event with the operation

labels event with the return value

vertices = set of client operation events

"returns-before" partial order = *client-observable* order of operations

"same-session" equivalence class partitions events into sessions



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

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

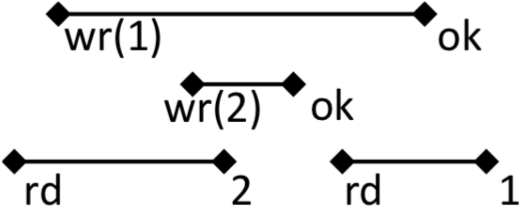
An event graph G is a tuple $(E, d_1, \ldots, d_n)$ where $E \subseteq Events$ is a finite or countably infinite set of events, and each d_i is an attribute or relation over E.

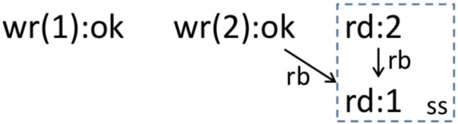
Histories as event graphs

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





$$\left[\begin{array}{rl}
\multicolumn{2}{l}{\text{graph is } (V, \mathsf{op}, \mathsf{rval}, \mathsf{rb}) \text{ where}} \\
V &= \{a, b, c, d\} \\
\mathsf{op} &= \{(a, \mathsf{wr}(1), (b, \mathsf{wr}(2)), \\
&\qquad (c, \mathsf{rd}), (d, \mathsf{rd})\} \\
\mathsf{rval} &= \{(a, ok), (b, ok), \\
&\qquad (c, 2), (d, 1)\} \\
\mathsf{rb} &= \{(b, d), (c, d)\} \\
\mathsf{ss} &= \{(a, a), (b, b), (c, c), \\
&\qquad (c, d), (d, d), (d, c)\}
\end{array}\right]$$

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

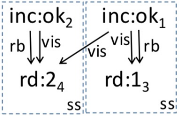
An **abstract** execution is an event graph $(E, op, rval, rb, ss, vis, ar)$ such that

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

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



$$\left[\begin{array}{rl} \text{graph is } (\{a_1, a_2, b_1, b_2\}, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss}, \mathsf{vis}, \mathsf{ar}) \text{ where} & \\ \mathsf{op} = \{(a_1, \mathsf{inc}), (a_2, \mathsf{rd}), (b_1, \mathsf{inc}), (b_2, \mathsf{rd})\} & \\ \mathsf{rval} = \{(a_1, ok), (a_2, 2), (b_1, ok), (b_2, 1)\} & \\ \mathsf{rb} = \{(a_1, a_2), (b_1, b_2)\} & \\ \mathsf{ss} = \{(a_1, a_2), (a_2, a_1), (b_1, b_2), (b_2, b_1)\} & \\ \mathsf{vis} = \{(a_1, a_2), (b_1, b_2), (b_1, a_2)\} & \\ \mathsf{ar} = b_1 < a_1 < b_2 < a_2 & \end{array}\right]$$

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 ReadValueConsistency

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 ⇒ Difficult to program against

Eventual visibility

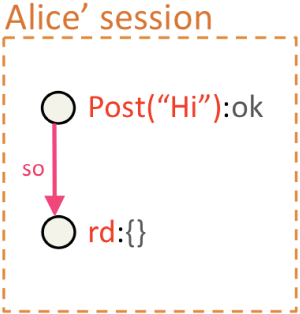
An abstract execution satisfies EventualVisibility if all events become eventually visible.

$$\forall e \in E : |\{e' \in E | (e \xrightarrow{rb} e') \wedge (e \centernot{\xrightarrow{vis}} e')\}| < \infty$$

Session guarantees

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

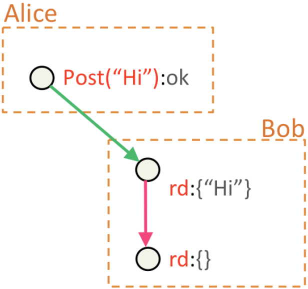
Read My Writes



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

$$so \subseteq vis$$

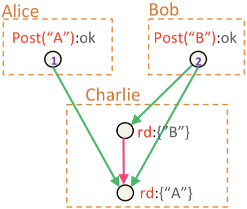
Monotonic Reads



- 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.

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

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

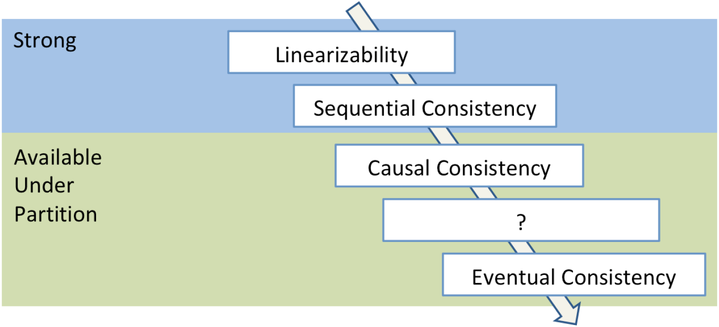
- Linearizability requires RealTime:

$$rb \subseteq ar$$

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

Further reading I

[1] 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.

[2] 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.

[3] 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.