Replication and Consistency Models

Pedro F. Souto (pfs@fe.up.pt)

October 1, 2024

Data Replication

Replicate data at many nodes

Performance local reads

Reliability no data-loss unless data is lost in all replicas

Availability data available unless all replicas fail or become unreachable

Scalability balance load across nodes for reads

Data Replication: Challenge

Upon an update

Issue The values of the different replicas will be different

Solution Run some protocol that

- Pushes data to all replicas

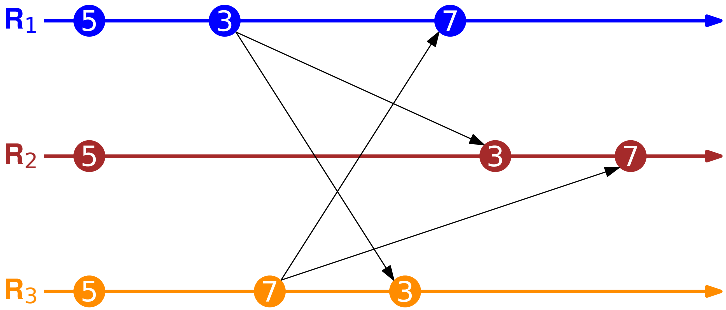
Challenge ensure **data consistency**

Note Right now, we will focus on **concurrency**

- Failures make this even more challenging

Conflicts

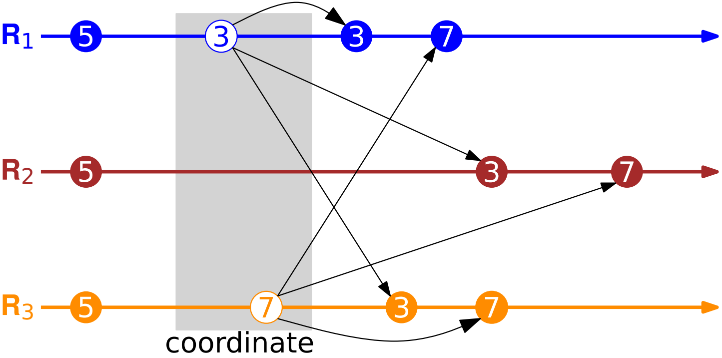
Observation Updating at different replicas may lead to different results, i.e. **inconsistent** data



Strong Consistency

All replicas execute/apply updates in the same order

- Deterministic updates: **same initial state** leads to **same result**



Actually, total order is not enough: it must be sensible

Strong Consistency Models

Sequential Consistency

Serializability

Linearizability

Sequential Consistency Model (Lamport 79)

Definition An execution is **sequential consistent iff** it is identical to a sequential execution of all the operations in that execution such that

- all operations executed by any thread, appear in the order in which they were executed by the corresponding thread

Observation This is the model provided by a multi-threaded system on a uniprocessor

Counter-example Consider the following operations executed on two replicas of variables x and y, whose initial values are 2 and 3, respectively

```
Répl. 1      Répl. 2
(2,3)        (2,3)      /* Initial values */
x = y + 2;   y = x + 3;
(5,5)        (5,5)      /* Final values */
```

If the two operations are executed sequentially, the final result **cannot** be $(5, 5)$

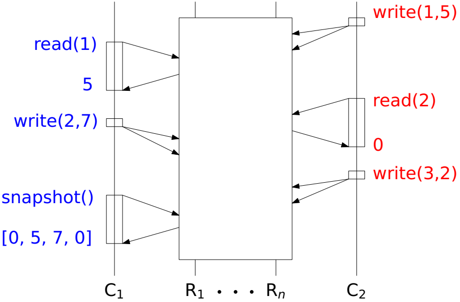
Sequentially Consistent Execution

Array Data type

Read(a) read value of array's element/index *a*
Write(a, v) write value *v* to array's element/index *a*
Snapshot() read all values stored in the array

Execution for an array with 4 elements (initial value $[0, 0, 0, 0]$)

- reads/snapshots access only one replica
- writes access more than one replica, but return immediately



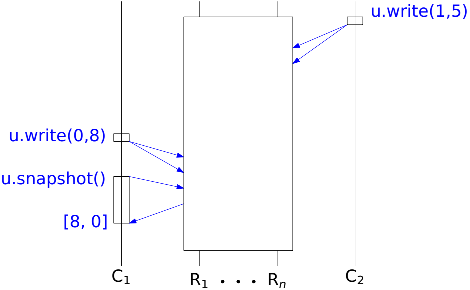
src: Fekete and Ramamritham 09

1. *write*$(1, 5)$
2. *read*(1)
3. *read*(2)
4. *write*$(2, 7)$
5. *snapshot*$()$
6. *write*$(3, 2)$

- What other ordering would be possible?

Sequential Consistency Is Not Composable

- Consider two arrays, *u* and *v*, each with 2 elements;
- Assume that we use a sequential consistent protocol to replicate each of the arrays, so that each array is sequential consistent



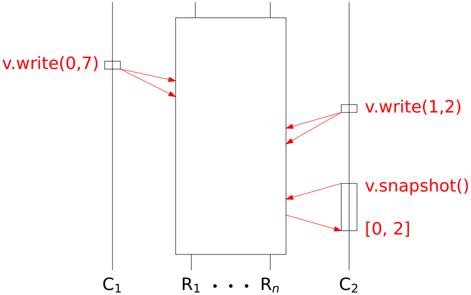
src: Fekete and Ramamritham 09

Array *u*

1. $u.write(0, 8)$
2. $u.snapshot()$
3. $u.write(1, 5)$

Sequential Consistency Is Not Composable

- Consider two arrays, *u* and *v*, each with 2 elements;
- Assume that we use a sequential consistent protocol to replicate each of the arrays, so that each array is sequential consistent



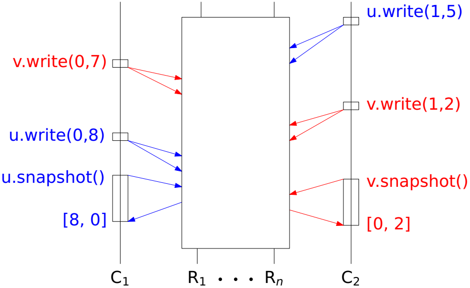
src: Fekete and Ramamritham 09

Array *v*

1. $v.write(1,2)$
2. $v.snapshot()$
3. $v.write(0,7)$

Sequential Consistency Is Not Composable

- Consider two arrays, *u* and *v*, each with 2 elements;
- Assume that we use a sequential consistent protocol to replicate each of the arrays, so that each array is sequential consistent



src: Fekete and Ramamritham 09

Array *u*

1. $u.write(0, 8)$
2. $u.snapshot()$
3. $u.write(1, 5)$

Array *v*

1. $v.write(1, 2)$
2. $v.snapshot()$
3. $v.write(0, 7)$

- The combined execution may not be sequential consistent
  - We can show that it is not possible to merge these two sequences into a single one that is sequential consistent.

Linearizability (Herlihy&Wing90)

Def. An execution is **linearizable** iff it is **sequential consistent** and

- if op_1 **occurs before** op_2, according to one **omniscient observer**, then op_1 appears before op_2

Assumption Operations have:

start time
finish time

measured on some clock accessible to the omniscient observer

- *op*1 **occurs before** *op*2, if op_1's finish time is smaller than that op_2's start time
- If op_1 and op_2 overlap in time, their relative order may be any

Important Even though the specification uses the concept of omniscient observer, which does not exist in a distributed system

- Communication delay is different from 0 and depends on distance (think speed of ligh and relativity)

this does not mean that we cannot implement linearizability

- Usually, we add synchronization

Linearizable Execution of an Array

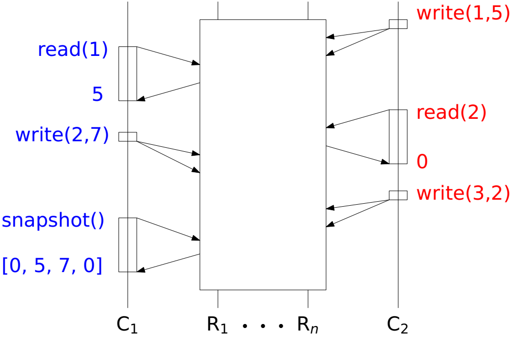
Array Data type

Read(a) read value of array's element/index *a*

Write(a, v) write value *v* to array's element/index *a*

Snapshot() read all values stored in the array

Execution for an array with 4 elements (initial value $[0, 0, 0, 0]$)



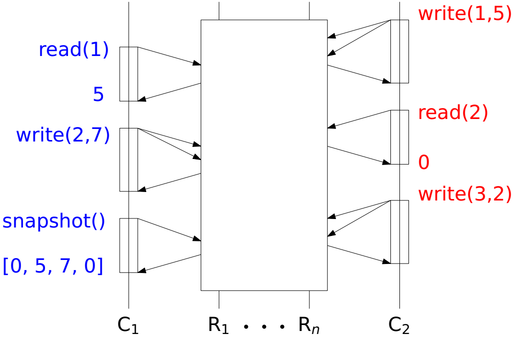
src: Fekete and Ramamritham 09

1. *write*$(1, 5)$
2. *read*(1)
3. *read*(2)
4. *write*$(2, 7)$
5. *snapshot*$()$
6. *write*$(3, 2)$

- Is not linearlizable
  - `write(3,2)` on C_2 occurs before `snapshot` on C_1

Linearizable Execution

- To ensure linearizability we need additional synchronization
  - e.g. add an "ack" to the write operation



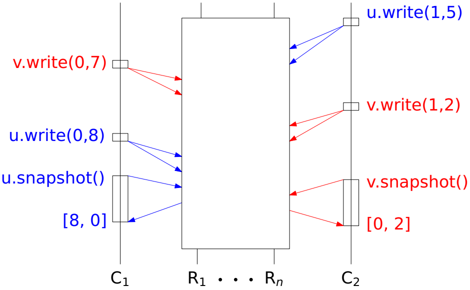
src: Fekete and Ramamritham 09

1. *write*(1, 5)
2. *read*(1)
3. *read*(2)
4. *write*(2, 7)
5. *snapshot*()
6. *write*(3, 2)

- write(3,2) on C_2 is now concurrent with snapshot on $C_!$
  - Even though its finish time is earlier than that of snapshot

Linearizability is composable

- We will not demonstrate it here, you will have to accept it
- Why doesn't the execution we have presented earlier show that linearizability is not composable?
  - Even if the writes tookß longer supposedly because of additional synchronization



src: Fekete and Ramamritham 09

Array *u*

1. $u.write(0, 8)$
2. $u.snapshot()$
3. $u.write(1, 5)$

Array *v*

1. $v.write(1, 2)$
2. $v.snapshot()$
3. $v.write(0, 7)$

One-copy Serializability (Transaction-based Systems)

Definition The execution of a set of transactions is **one-copy serializable** **iff** its outcome is similar to the execution of those transactions in a **single** copy

Observation 1 Serializability used to be the most common consistency model used in transaction-based systems

- DB systems nowadays provide weaker consistency models to achieve higher performance

Observation 2 This is essentially the sequential consistency model, when the operations executed by all processors are transactions

- The isolation property ensures that the outcome of the concurrent execution of a set of transactions is equal to some sequential execution of those transactions

Observation 3 (Herlihy . . . sort of) Whereas

Serializability Was proposed for databases, where there is a need to preserve complex application-specific invariants

Sequential consistency Was proposed for multiprocessing, where programmers are expected to reason about concurrency

Weak Consistency

- Strong consistency models usually make it easy to reason about replicated systems
  - Essentially, they strive to give the illusion of a non-replicated system
- However, their implementation usually requires tight synchronization among replicas, this adversely affects:
  - **Scalability (performance)**
  - **Availability**
- Weakly consistency models strive to:
  - Improve scalability and availability
  - While providing a set of guarantees that is useful for their users
    - Weak consistency models are usually application domain dependent
- We (Prof. Carlos Baquero) will discuss some weak consistency models later in the course

Further Reading

- Fekete A.D., Ramamritham K. (2010) *Consistency Models for Replicated Data*. In: Charron-Bost B., Pedone F., Schiper A. (eds) Replication. Lecture Notes in Computer Science, vol 5959. pp. 1-17
- van Steen and Tanenbaum, *Distributed Systems, 3rd Ed.*
  - Section 7.2 *Data-centric consistency models*
  - Section 7.3 *Client-centric consistency models*