Sistemas Distribuídos

Consistency and Replication

Eurico Pedrosa

António Rui Borges

Universidade de Aveiro - DETI

2025-05-15

Consistency and Replication

Regions of Distributed Storage

A distributed storage region can be viewed as a form of memory, database, or file system that is **replicated across multiple processing nodes**.

It is assumed that:

- Each processing node has direct access to its **local replica of the entire region**.
- Two types of operations are relevant:
  - **Write-like operations**, which modify the contents of some data item.
  - **Read-like operations**, which include all remaining operations.

Regions of Distributed Storage

- **Write-like operations must be propagated** to all replicas to ensure each one is updated.
- **Read-like operations may or may not be propagated**, depending on the consistency model.
- Access to local replicas is **performed in parallel**, i.e., concurrently by the respective processes.

Consistency Models

A **consistency model**- defines a set of rules — or a **contract** — that, if followed by the participating processes, ensures the **correct behavior** of a distributed storage system.

- The key challenge is to **precisely define what is meant by "correct behavior"**, and to apply this definition **systematically**.
- Each consistency model is characterized by specifying the **range of values that a read operation is allowed to return**, based on the **write operations that have previously occurred**.
- When **concurrent access** to data is possible, these models are referred to as **data-centric consistency models**.

Criterion for 'Correct Operation'

Let $\text{op}_{i,0}, \text{op}_{i,1}, \text{op}_{i,2}, \ldots$ with $i = 0, 1, \ldots, N-1$, denote the sequence of **write-like**- and **read-like** operations executed by process i on the distributed storage region.

- Each operation is defined by:
  - its **type** (read or write),
  - its **arguments**,
  - and its **return value**.
- All operations are **synchronous**: a process may only issue a new operation after the previous one has completed.
- The N processes perform their sequences **in parallel**, potentially interleaving accesses to the shared (distributed) storage region.

Criterion for ‘Correct Operation’

If the storage were centralized, the result of these interleaved accesses would be a **single global sequence** of operations such as:

$$\mathrm{op}_{2,0}\,,\mathrm{op}_{2,1}\,,\mathrm{op}_{0,0}\,,\mathrm{op}_{1,0}\,,\mathrm{op}_{1,1}\,,\mathrm{op}_{2,2}\,,\mathrm{op}_{2,3}\,,\mathrm{op}_{0,1}\,,\mathrm{op}_{1,2}\,,\ldots$$

Criterion for ‘Correct Operation’

Defining ‘Correct Operation’

- The **correctness** of a distributed storage region is established by comparing the observed behavior to one or more **global canonical sequences**.
- These canonical sequences are:
  - **virtual** (they need not actually occur),
  - formed by interleaving the local sequences of operations from all processes,
  - and must **conform to the consistency model** being applied.
- If **no such canonical sequence exists** that satisfies the constraints of the consistency model for a given execution, the system is considered to have **violated the model**.

Strict Consistency

Strict Consistency

Any read-like operation performed on the register x returns the value produced by the most recent write-like operation on x.

Strict Consistency

The **ideal consistency model**, where all operations appear to occur in a single, globally ordered sequence, can only be realized in **monoprocessor systems**.

- This ideal model **cannot be implemented in parallel or distributed systems**, because the concept of a "**most recent**" **event** becomes **ambiguous**.

Strict Consistency

Why ambiguity arises:

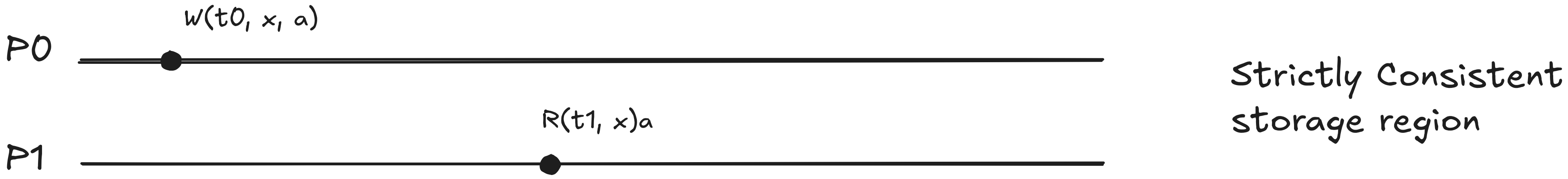
- The **propagation speed of physical signals is finite**, meaning:
  - Local clocks across processing nodes **cannot be perfectly synchronized**.
  - It is **impossible to establish a global clock** that provides an absolute ordering of events.
- Additionally:
  - **Write-like and read-like operations take non-zero time** to execute.
  - **Delays and asynchrony** further obscure a single, unified timeline.

Strict Consistency

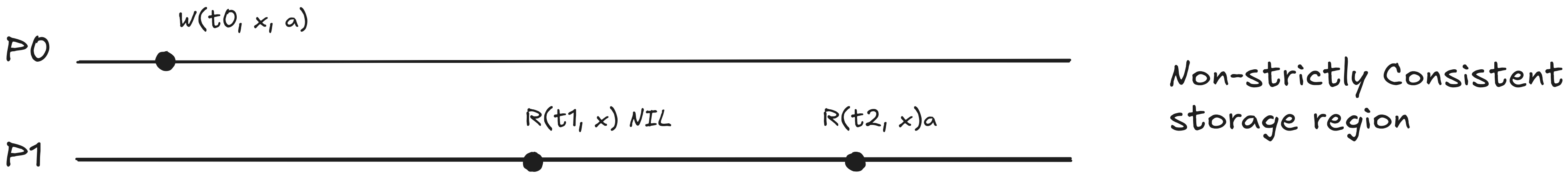
Conclusion:

Events in a distributed system **cannot be globally ordered using a unique time standard**. Instead, **logical clocks** or **causal ordering** must be used to reason about the sequence of events.

Strict Consistency



canonical sequence: $t_0 < t_1 \Rightarrow W_0(t_0, x, a,) - R_1(t_1, a)a$



there is no canonical sequence which mimics the results

Linearizability

Linearizability

> Parallel access to a register x is seen by all involved processes as if the performed operations were ordered in an unique and well-defined sequence by keeping the chronological order that is locally perceived.
>
> — Lamport / Herlihy & Wing

Linearizability

Linearizability is a strong consistency model that assumes the existence of a synchronization mechanism that provides an approximation to a **global clock**, allowing events to be **totally ordered**.

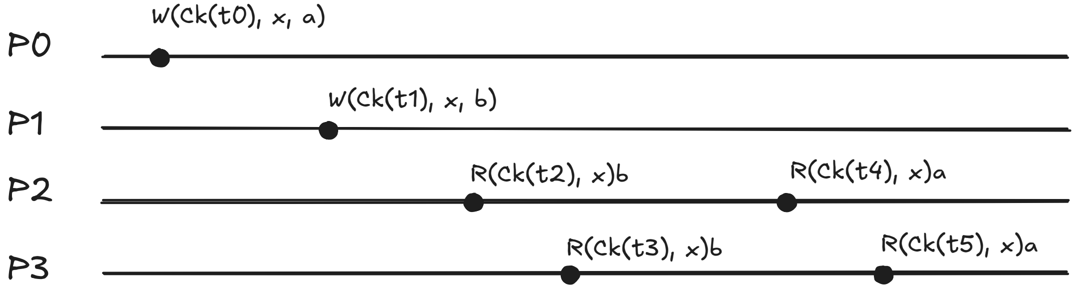
- The objective is to ensure that **every process always observes the most up-to-date value**.
- Conceptually, there exists a **canonical global sequence** of all operations — as if they were executed **atomically** and **instantaneously** on a centralized storage.

Linearizability

Key properties:

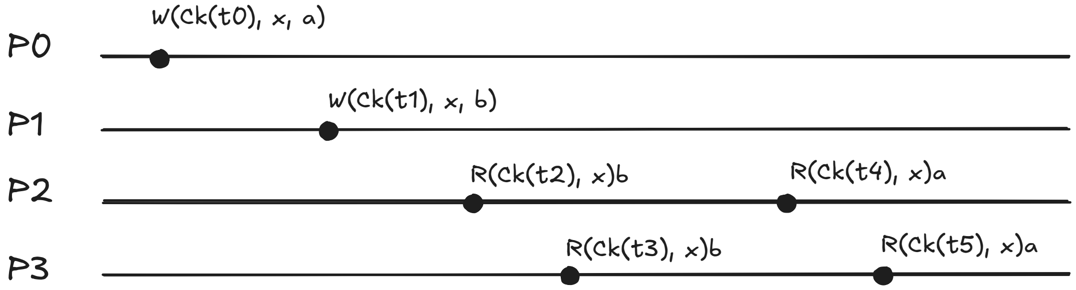
- Each operation appears to take effect **at a single, indivisible point in time** between its invocation and completion.
- The global sequence of operations must be **consistent with the real-time order** in which operations are perceived locally.
- The system provides the **illusion of a single copy** of the data being accessed by all processes — even though in reality, operations are performed **concurrently** on **replicated** or **distributed** data.

Linearizability



Linearizable
storage region

canonical sequence: $Ck_1(t_1) < Ck_2(t_2), Ck_3(t_3) < Ck_0(t_0) < Ck_2(t_4), Ck_3(t_5) \Rightarrow$
$\Rightarrow W_1(Ck_1(t_1), x, b) - R_{2,3}(-, x)b - W_0(Ck_0(t_0), x, a) - R_{2,3}(-, x)a$



Non linearizable
storage region

$Ck_0(t_0) < Ck_1(t_1) \Rightarrow$ there is no canonical sequence which mimics the results

Linearizability

Implementation

To implement **linearizability**, the system must ensure that **all read-like and write-like operations are propagated and acknowledged** by all processes managing the local replicas of the distributed storage region **before the operation takes effect**.

- This requires a mechanism to **totally order all operations**.
- A common approach is to use **Lamport scalar clocks**:
  - Each process maintains a local logical clock.
  - Clocks are updated according to **Lamport's rules** to maintain a **consistent event ordering** across processes.

Linearizability

The use of logical clocks ensures that all operations are **serialized** in a way that preserves **causal and real-time constraints**, approximating a global ordering.

Linearizability

Possible Application Areas

Linearizability is essential in systems where **strong consistency and strict fairness** are required, such as:

- Government support services
- Financial transaction systems
- Distributed databases requiring immediate visibility of updates
- Critical infrastructure control systems

Sequencial Consistency

Sequencial Consistency

> Parallel access to a register x is seen by all involved processes as if the performed operations were ordered in an unique and well-defined sequence where the operations of each individual process keep the order of local execution.
>
> — Lamport / Herlihy & Wing

Sequencial Consistency

Sequential consistency is a model where **time (global or local) plays no explicit role** in defining the correctness of execution.

- As a result, it is **not necessary to totally order all events** based on timestamps or clocks.

Key Concept:

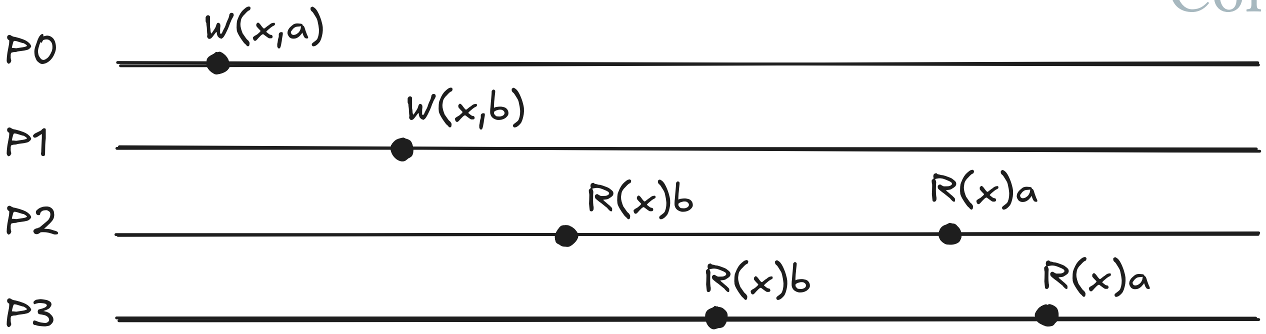
There exists a **canonical global sequence** of operations — over a centralized virtual region — that:

- **Preserves the program order** of operations from each individual process,
- **Provides a consistent view of execution**, as if the operations were performed one at a time in some sequential order.

Sequencial Consistency

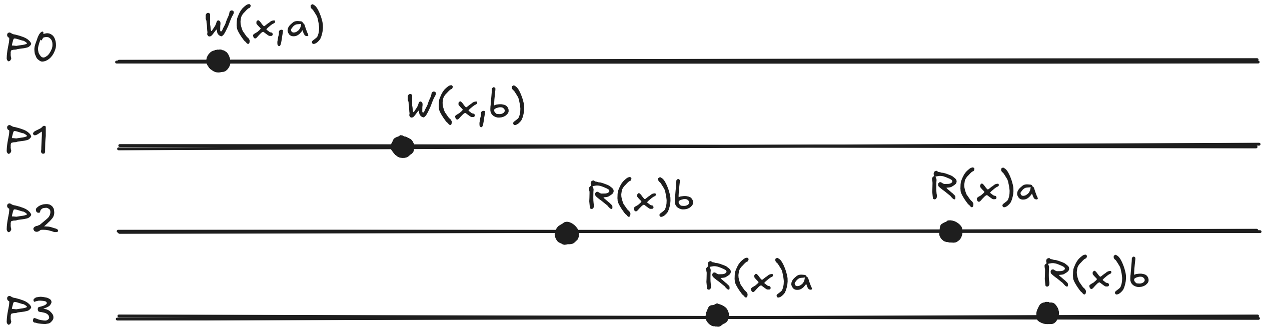
Even though the actual operations are executed concurrently, the outcome must reflect an execution where each process's operations appear in order, and all processes see a single, consistent history.

Sequencial Consistency



Sequentially Consistent storage region

canonical sequence: $W_1(x, b) - R_{2,3}(x)b - W_0(x, a) - R_{2,3}(x)a$



No Sequentially Consistent storage region

there is no canonical sequence which mimics the results

Sequencial Consistency

Implementation

To implement **sequential consistency**, only **write-like operations** need to be **propagated and acknowledged** by all processes managing local replicas **before they take effect**.

- **Read-like operations** may access local copies without coordination, as long as consistency is preserved.
- A **total order of write operations** is established using **Lamport's logical scalar clocks**:
  - Each process maintains a local logical clock.
  - Clocks are updated and messages are timestamped according to **Lamport's rules**, ensuring a consistent ordering of writes across the system.

Sequencial Consistency

This approach guarantees that all processes observe write operations in the same order, preserving **program order** without requiring global time.

Possible Application Areas

Sequential consistency is suitable for systems where **general fairness** and **logical operation order** are important, including:

- Booking systems
- E-commerce platforms
- Online inventory management
- Collaborative editing tools

Causal Consistency

Causal Consistency

> Parallel access to a register x is seen by all involved processes as if the performed operations, which keep between them a causal relation, were ordered in an unique and well-defined sequence. The remaining operations, said to be concurrent, may be perceived in any order by the different processes
>
> — Hutto / Ahamad

Causal Consistency

To understand **causal consistency**, we must first define **causality** in this context. An operation is said to be **causally related** to another if:

- The operations are executed **in sequence by the same process** (program order).
- A **read-like operation** observes a value produced by a **write-like operation** — the two are causally related, regardless of which process performed them.

Causal Consistency

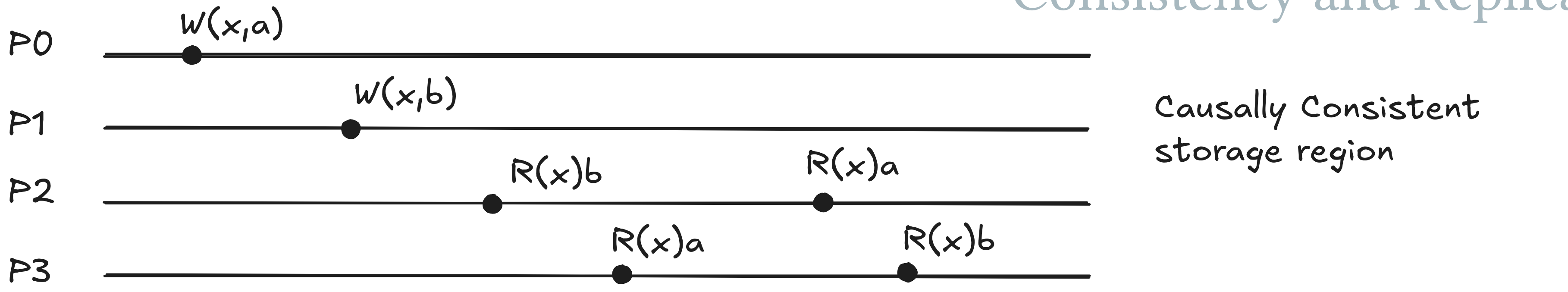
Key Concepts:

There exists a **canonical global execution** over a virtual centralized memory that reflects a **consistent, unique view**, but **only for operations that are causally related**.

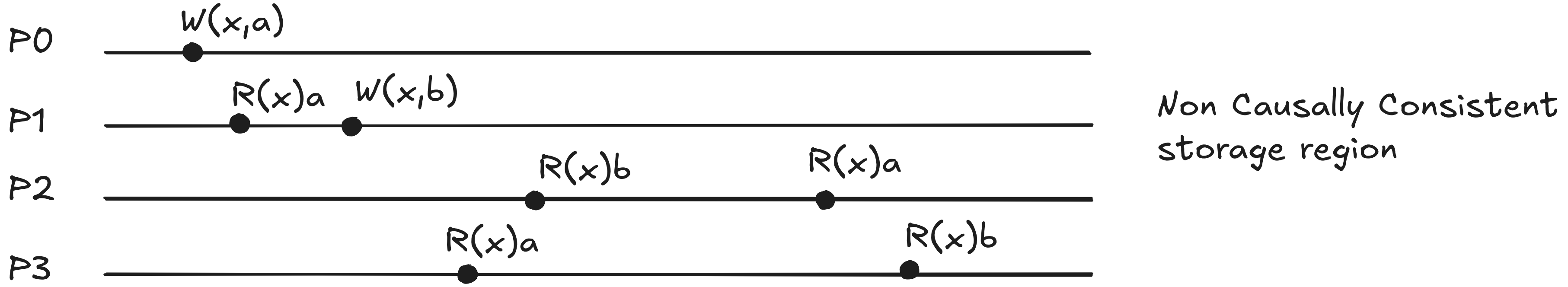
- For **causally related operations**, the system must preserve their **sequential order** across all processes.
- For **concurrent (i.e., non-causally related) operations**, processes are **allowed to perceive different orders**.

In short: **Sequential consistency is enforced only within causal chains**. All other operations may be reordered without violating causal consistency.

Causual Consistency



as the write like operations are concurrent, the canonical sequence is in this case empty



as the write like operations are causally related, there is no canonical sequence

Causal Consistency

Implementation

To implement **causal consistency**, only **write-like operations** need to be **propagated and acknowledged** by all processes managing local replicas.

- The system maintains a **partial order** of write-like operations by using **logical vector clocks** at each process.
- Vector clocks are updated and compared according to the standard **causality-preserving rules**:
  - Each process maintains a vector of counters, one per process.
  - Operations are timestamped and propagated with these vectors to **preserve the causal relationships** between events.

Causal Consistency

This ensures that **causally related operations are observed in the same order** by all processes, while allowing **independent operations to be reordered freely**.

Possible Application Areas

Causal consistency is ideal for applications where **preserving the cause-effect relationship between operations** is important, such as:

- Chat applications (e.g., ensuring replies are seen after original messages)
- Collaborative editing tools
- Social media feeds (e.g., ensuring comments appear after the posts they respond to)

Fifo Consistency

Fifo Consistency

> Parallel access to a register x is seen by all involved processes as if the performed operations by any given process keep the order of local execution.
>
> — Lipton / Sandberg

Fifo Consistency

In **FIFO consistency**, the notion of a **global canonical sequence of operations** is **not required**.

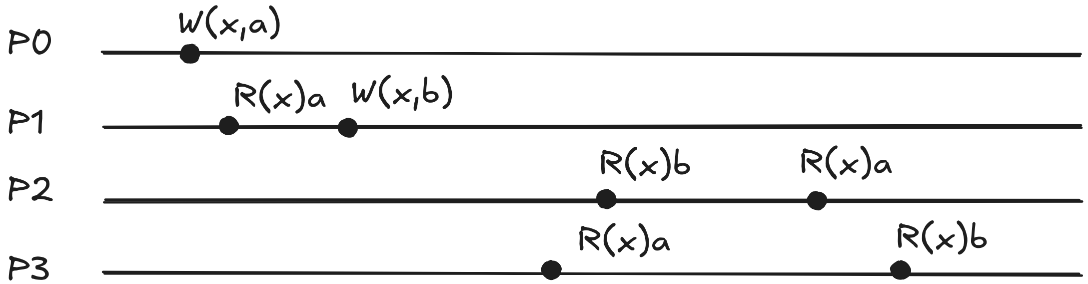
Key Concepts:

- The only guarantee is that **write operations issued by the same process are observed by all other processes in the same order.**
- **Reads and writes from different processes may be interleaved differently** across nodes — no global ordering of writes across processes is enforced.
- **Read-like operations are not constrained** in this model, as they **do not affect the shared state.**

Fifo Consistency

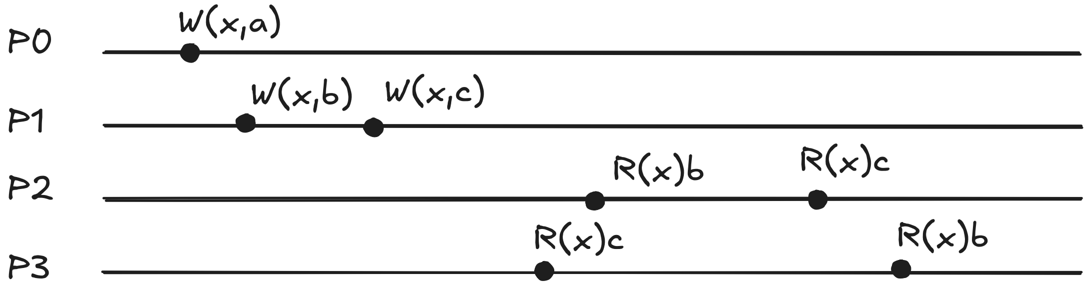
Each process sees the **writes from any given other process in the order they were issued**, but the interleaving of writes from different sources may differ between observers.

Fifo Consistency



FIFO Consistent storage region

as the write like operations are concurrent, the local order of the [write like] operations is trivial



Non FIFO Consistent storage region

the local order of the [write like] operations executed by process P_1 is not perceived as such by process P_3

Fifo Consistency

Implementation

To implement **FIFO consistency**, only **write-like operations** need to be **propagated** by all processes managing the local copies of the distributed storage region.

- The system ensures that **write operations from each process are delivered in the same order** to all other processes.
- This can be achieved by assigning a **per-process message counter**:
  - Each process tags its outgoing writes with a **monotonically increasing sequence number**.
  - Receiving processes **buffer and deliver writes in order**, based on the sender's counter.

Fifo Consistency

This guarantees **per-sender ordering**, but does not impose any order between writes from different sources.

Possible Application Areas

FIFO consistency is suitable for systems where a **weaker form of consistency is acceptable**, including:

- News distribution platforms
- Weather forecasting systems
- Multimedia streaming
- Sensor data aggregation

Suggested Reading

- M. van Steen and A.S. Tanenbaum, Distributed Systems, 4th ed., distributed-systems.net, 2023.
- Distributed Systems: Concepts and Design, 4th Edition, Coulouris, Dollimore, Kindberg, Addison-Wesley