Chapter Six

Replication, Consistency and Fault Tolerance

6.1 **Replication**

- Data are generally replicated to enhance reliability or improve performance
- Major Problem
  - keeping replicas consistent

Reason for Replication

- *performance enhancement*,
- *increasing availability*,
- *fault tolerance* and *reliability*.

Reliability

- keep working after one replica crashes by simply switching to one of the other replicas
- provide better protection against corrupted data

Performance

- performance can be improved by replicating the server and subsequently dividing the work
- By placing a copy of data in the proximity of the process using them, the time to access the data decreases as a result the performance as perceived by the processes increases

Replication as Scaling Technique

- Replication and caching for performance are widely applied as scaling techniques.
- Scalability issues generally appear in the form of performance problems.
- **Placing copies of data close to the processes** using them **can improve performance through reduction of access time** and **thus solve scalability problems**.
- **Possible Problem**: keeping copies up to date requires more bandwidth.
  - If the copies are refreshed more often than used (low access-to-update ratio), the cost (bandwidth) is more expensive than the benefits.
- **Replication itself is subject to serious scalability problems due to tight consistency** (a write is performed at all copies in a single atomic operation or transaction).
- **Caching Problems:**

  E.g. Web browsers often locally store a copy of a previously fetched Web page (i.e., they cache a Web page).
  - modification will not have been propagated to cached copies
  - **Solution**: forbid the browser to keep local copies and let the Web server invalidate or update each cached copy

6.2 Consistency

- When we replicate data we must ensure that when one copy of the data is updated all the other copies are updated too.
- Depending on how and when these updates are executed we can get inconsistencies in the replicas (i.e., all the copies of the data are not the same).
- There are two ways in which the replicas can be inconsistent.
  - Staleness and if operations are performed in different orders at different replicas.

Consistency model

- A consistency model defines which interleaving of operations (i.e., total orderings) are acceptable (admissible).
- A replica/data store that implements a particular consistency model must provide a total ordering of operations that is admissible.

1. Data-Centric Consistency Models

- most widely used class of consistency models
- Gives emphasis for write operation

a) Strict Consistency

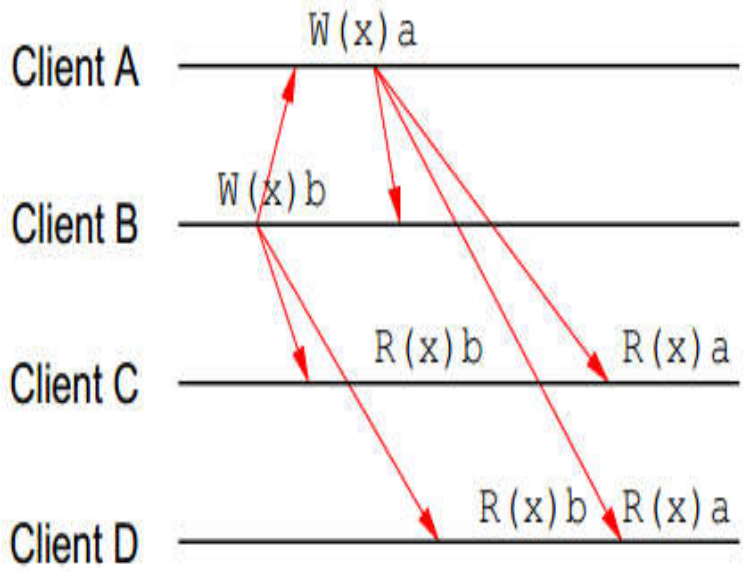
- requires that **any read on a data item returns** a value corresponding to the **most recent write** on that data item.

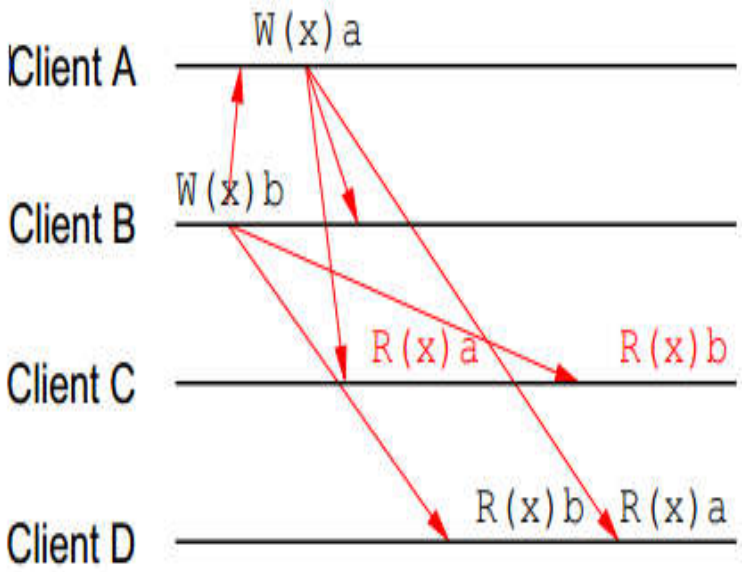
b) Linearisable consistency model

- requirement of absolute global time is dropped.
- requires that all operations be ordered according to their timestamp.

c) Sequential Consistency

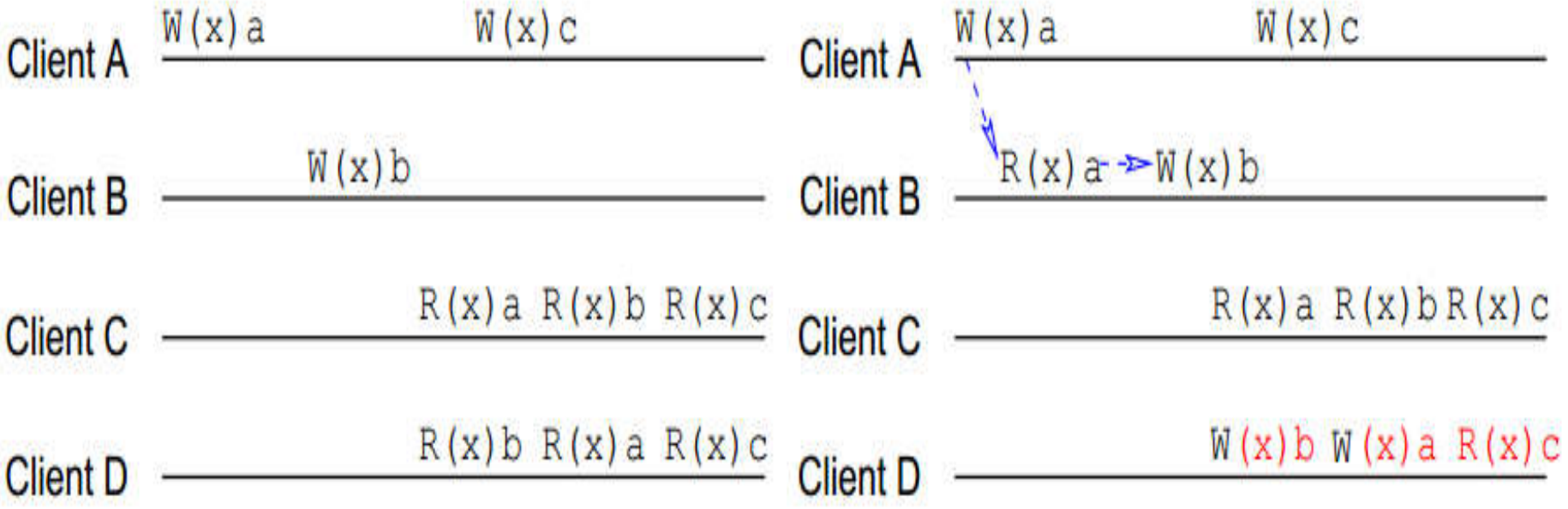
- drops the time ordering requirement.
- In a data store that provides sequential consistency, all clients see all (write) operations performed in the same order.
- In sequential consistency there are many valid total orderings.
- The only requirement is that all clients see the same total ordering.





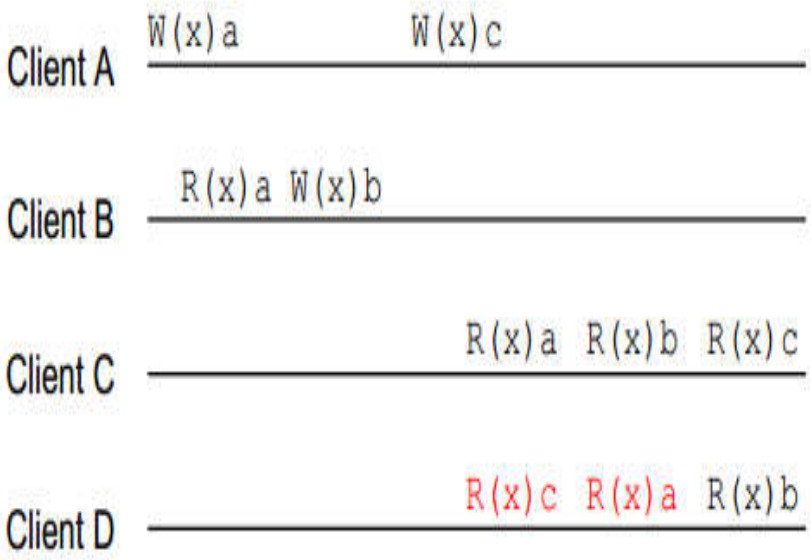
d) Causal Consistency

- Only causally related write operations are executed in the same order on all replicas.
- Two operations are causally related if:
  - A read is followed by a write in the same client
  - A write of a particular data item is followed by a read of that data item in any client.
- If operations are not causally related they are said to be concurrent.
- Concurrent writes can be executed in any order, as long as program order is respected.



e) FIFO Consistency

- removes limitations about the order of any concurrent operations.
- requires only that any total ordering respect the partial orderings of operations (i.e., program order).

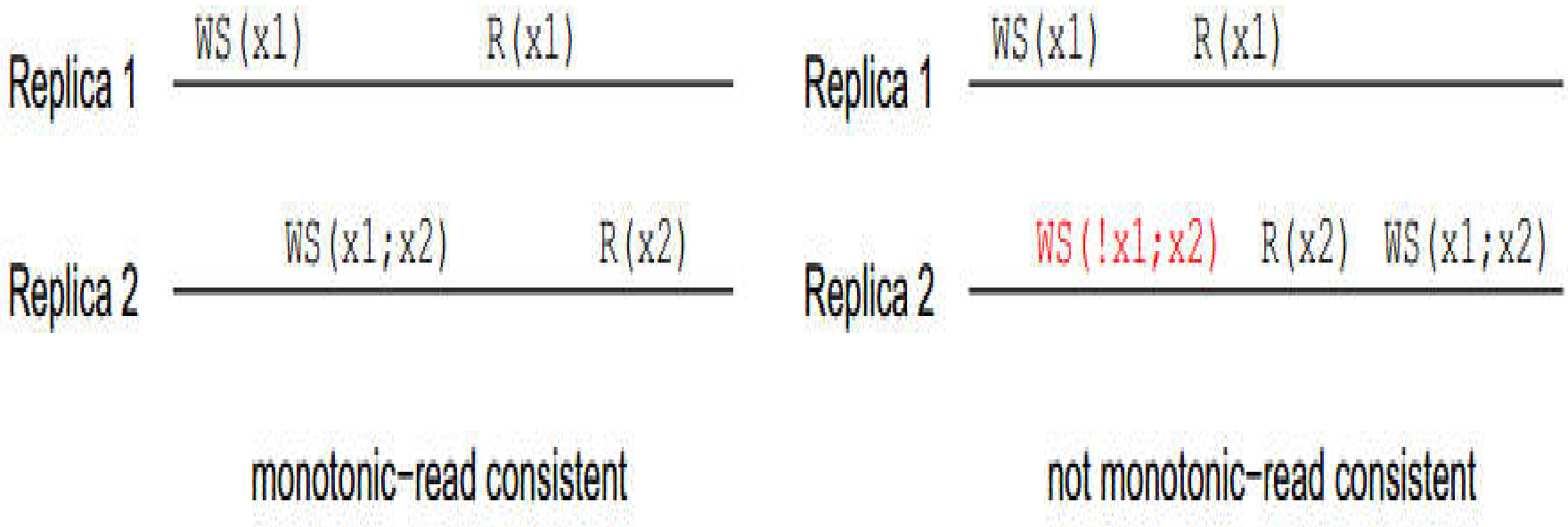


2. Client-Centric Consistency Models

- Underlying assumption
  - ***Data-Centric Consistency Models***
    - the number of reads is approximately equal to the number of writes, and that concurrent writes occur often.
  - **Client centric consistency models**
    - clients perform more reads than writes and that there are few concurrent writes
    - They also assume that clients can be mobile, that is, they will connect to different replicas during the course of their execution.
    - are **based on** the ***eventual consistency model*** but offer per client models that hide some of the inconsistencies of eventual consistency.
- Client-centric consistency models are useful because they are relatively cheap to implement.
- A **write set W(S) contains the history of writes** that led to a particular value of a particular data item at a particular replica.
- *When showing timelines for client-centric consistency models we are now concerned with only one client performing operations while connected to different replicas.*

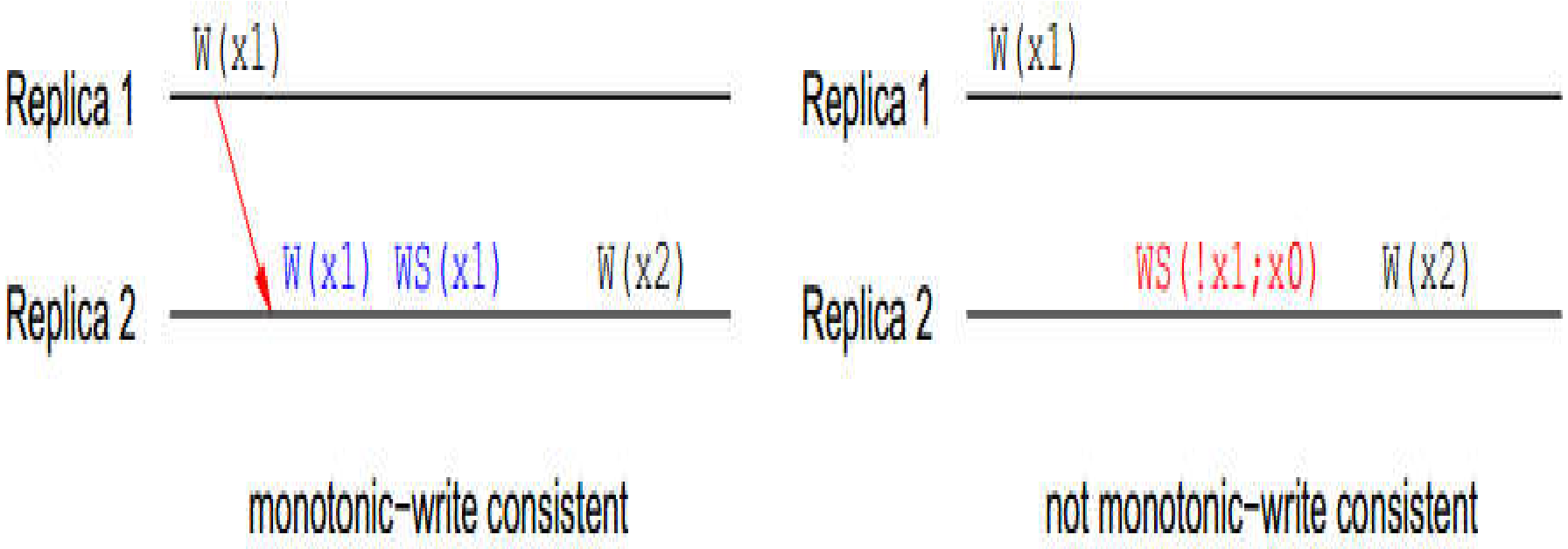
i. Monotonic reads model

- This model ensures that a **client will always see progressively newer data and never see data older than what it has seen before**.
- This means that **when a client performs a read on one replica and then a subsequent read on a different replica, the second replica will have at least the same write set as the first replica.**



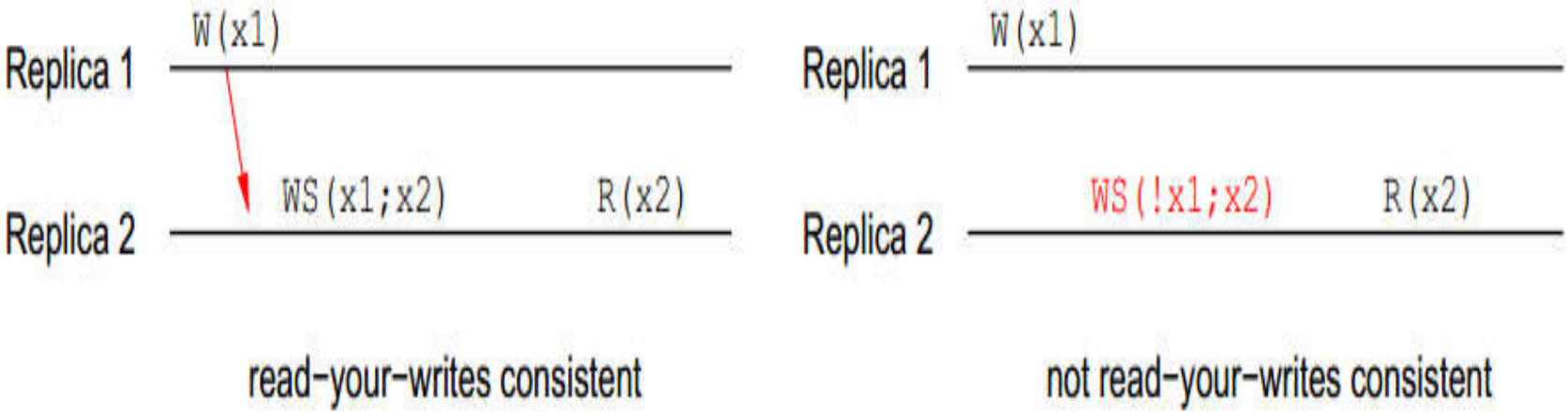
ii. Monotonic-writes model

- This model ensures that a write operation on a particular data item will be completed before any successive write on that data item by the same client.
- In other words, **all writes that a client performs on a particular data item will be sequentially ordered**.



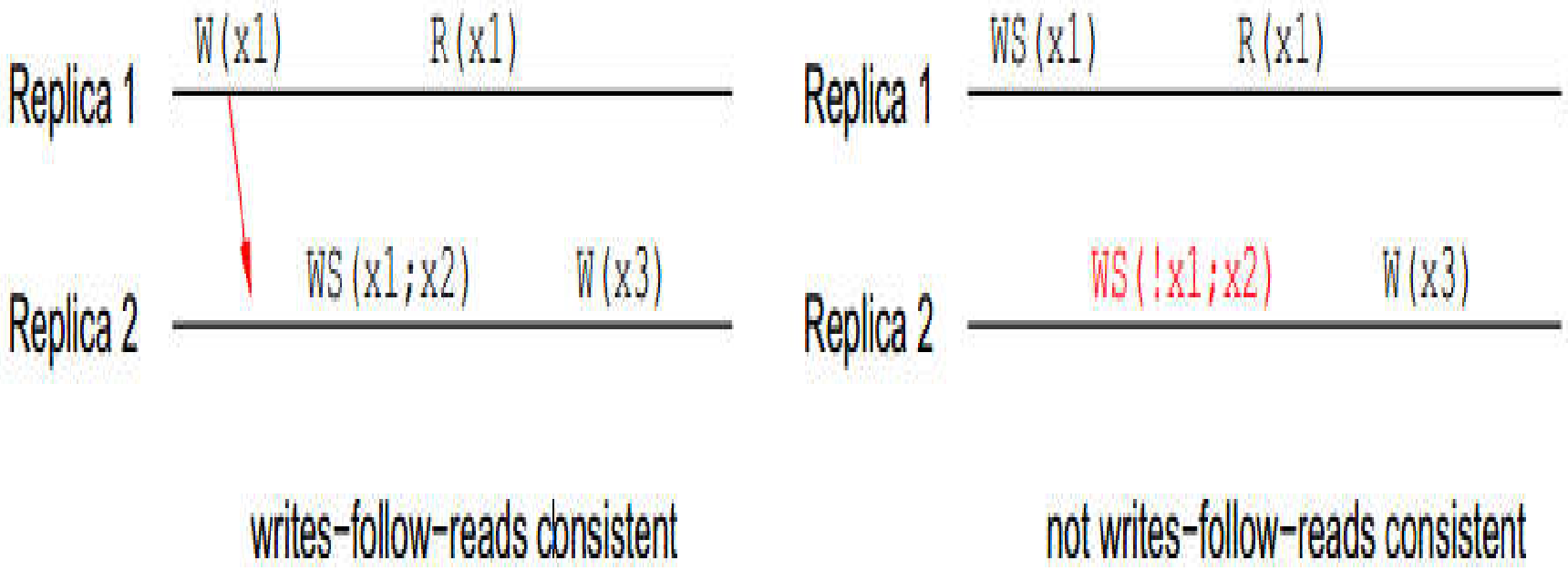
iii. Read your writes model

- In this model, **a client is guaranteed to always see its most recent writes**.



iv. Write Follows Reads model

- This model states the **opposite of read your writes**, and
- guarantees that **a client will always perform writes on a version of the data that is at least as new the last version it saw**.



Consistency protocol

- **A consistency protocol provides an implementation of a consistency model** in that it **manages the ordering of operations** according to its particular consistency model.
- **In this section we focus on** the various ways of implementing data-centric consistency models, with an emphasis on sequential consistency.
- There are two main classes of data-centric consistency protocols:
  - primary-based protocols and replicated-write based protocols.
    - **Primary-based protocols** require that each data item have a primary copy (or home) on which all writes are performed.
    - **Replicated-write protocols** require that writes are performed on multiple replicas simultaneously.
- The **primary-based approach** to consistency protocols can further be split into two classes: **remote-write** and **local-write**.

i. Single Server

- Is a remote-write protocol
- This protocol implements sequential consistency by effectively centralizing all data and foregoing data replication altogether (it does, however, allow data distribution).
- All write operations on a data item are forwarded to the server holding that item's primary copy.
- Reads are also forwarded to this server.
- Although this protocol is easy to implement, it does not scale well and has a negative impact on performance. (Lacks replication)

ii. Primary-Backup

- allows **reads to be executed at any replica**, however, **writes can still only be performed at a data item's primary copy.**
- The replicas (called backups) all hold copies of the data item, and **a write operation blocks until the write has been propagated to all of these replicas**.
- Because of the blocking write, this protocol can easily be used to implement sequential consistency. However, this has a negative impact on performance and scalability.
- It does, however, improve a system's reliability.

iii. Migration

- The migration protocol is the first of the **local-write protocols**.
- This protocol is **similar to single server in that the data is not replicated**.
- **When a data item is accessed, it is moved from its original location to the replica of the client accessing it**.
- **Benefit**
  - **data is always consistent** and **repeated reads and writes occur quickly, with no delay**.
- **Drawback**
  - **concurrent reads and writes can lead to thrashing behavior** where the data item is constantly being copied back and forth.
  - Furthermore **the system must keep track of every data item's current home**.

iv. Migrating Primary (multiple reader/single writer)

- **allow read operations to be performed on local replicas** and **to migrate the primary copy only on writes**.
- This **improves** on the **write performance of primary-backup** (only if non-blocking writes are used), and **avoids some of the thrashing of the migration approach**.
- **It is also good for (mobile) clients operating in disconnected mode.**

v. Active Replication

- In this protocol **write operations are propagated to all replicas, while reads are performed locally**.
- The writes can be propagated using either point-to-point communication or multicast.
- **Benefit**
  - **all replicas receive all operations at the same time (and in the same order)**.

vi. Quorum-Base

- write operations are executed at a subset of all replicas.
- When performing read operations, clients must also contact a subset of replicas to find out the newest version of the data.
- In this protocol all data items are associated with a version number.
- Every time a data item is modified its version number is increased.
- It **defines a write quorum** and **a read quorum**, which specify the number of replicas that must be contacted for writes and reads respectively.

6.3 Fault Tolerance

- A system is said to *fail* when it cannot meet its promises;
  - for instance failing to provide its user one or more of the services it promises.
- An error is a part of a system's state that may lead to a failure
- The cause of an error is called a fault:
  - Fault can be due to wrong or bad transmission medium or transmission errors caused by bad weather conditions
- Generally, a fault tolerant system is a system that can provide its services even in the presence of faults.
- Faults are classified into three
  - ***Transient***: *occurs once and then disappears*. If the operation is repeated, the fault goes away. E.g. Losing some bits due to interference
  - ***Intermittent***: it occurs, and then vanishes on its own accord, then reappears, and so on (e.g., a loose connection). It is difficult to diagnose.
  - ***Permanent***: one that continues to exist until the faulty component is repaired; for example, disk head crash, software bug, etc.

- **Partial failure is a characteristics of DS that distinguishes it from other systems**
- The failure of one component does not affect others in the system, the DS continues to work while the repair is being made (should tolerate faults)
- Fault tolerance is strongly related to *dependable systems*. **Dependability covers**
  - ***Availability***:- the probability that the system is operating correctly at any given time
  - ***Reliability***:- a system can run continuously without failure
  - ***Safety***:- *when a system temporarily fails to operate correctly, nothing catastrophic happens*
  - ***Maintainability***:- *how easily a failed system can be repaired*
  - Systems are also required to provide a high degree of ***security***

Failure Models

- In a DS, services may not be provided to clients due to failure of communication channels, servers or both.
- There are several classification schemes of failure, five of them are:
  - ***Crash failure***: a server halts, but was working correctly until it stopped
  - ***Omission failure***: a server fails to respond to incoming requests
    - ***Receive omission***: a server fails to receive incoming messages
    - ***Send omission***: a server fails to send messages
  - ***Timing failure***: a server's response lies outside the specified time interval; e.g., maybe it is too fast over flooding the receiver or too slow
  - ***Response failure***: the server's response is incorrect
    - **Value failure**: the value of the response is wrong; e.g., a search engine returning wrong Web pages as a result of a search
    - **State transition failure**: the server deviates from the correct flow of control; e.g., taking default actions when it fails to understand the request
  - ***Arbitrary failure*** (or Byzantine failure): a server may produce arbitrary responses at arbitrary times (most serious)

Failure Masking by Redundancy

- To be fault tolerant, the system tries to hide the occurrence of failures from other processes –this is called masking.
- The key technique for masking faults is to use redundancy.
- Three kinds are possible:
  - ***Information redundancy*** - add extra bits to allow recovery from garbled bits (error correction) e.g. Cyclic Redundancy Check
  - ***Time redundancy*** - an action is performed more than once if needed; e.g., redo an aborted transaction; useful for transient and intermittent faults
  - ***Physical redundancy*** - add (replicate) extra equipment (hardware) or processes (software)

Failure Recovery

- Refers to the process of restoring a (failed) system to a normal state of operation.
- Recovery can apply to the complete system (involving rebooting a failed computer) or to a particular application (involving restarting of failed process(es)).
- Restarting processes or computers is significantly more complicated in a distributed system.
- The main challenges are:

Reclamation of resources:

- Naively restarting the process or its host will lead to resource leaks and possibly deadlocks.

Consistency

- Naively restarting one part of a distributed computation will lead to a local state that is inconsistent with the rest of the computation.

Efficiency

- One way to avoid the above problems would be to restart the complete computation whenever one part fails.
- However, this is obviously very inefficient, as a significant amount of work may be discarded unnecessarily.

Forward vs. backward recovery

- Recovery can proceed either forward or backward.

- ***Forward error recovery***
  - requires removing (repairing) all errors in the system's state, thus enabling the processes or system to proceed.
  - In some case, actual computation may not be lost (although the repair process itself may be time consuming).
  - An example could be the replacement of a broken network cable with a functional one.
    - Here it is known that all communication has been lost, and if appropriate protocols are used (which, for example, buffer all outgoing messages) a forward recovery may be possible (e.g. by resending all buffered messages).
  - In most cases, however, forward recovery is impossible.

- ***Backward error recovery***
  - This restores the process or system state to a previous state known to be free from errors, from which the system can proceed (and initially retrace its previous steps).
  - Obviously this incurs overheads due to the lost computation and the work required to restore the state.
  - While the implementation of backward recovery faces substantial difficulties, it is in practice the only way to go, due to the impossibility of forward-recovery from most errors.

- Backward recovery can happen in one of two ways:

Operation-based recovery

  - keeps a log (or audit trail) of all state-changing operations.
  - The recovery point is reached from the present state by reversing these operations

State-based recovery

  - stores a complete prior process state (called a checkpoint).
  - The recovery point is reached by restoring the process state from the checkpoint (called roll-back).
  - State based recovery is also frequently called **rollback-recovery**.