Chapter Six
Replication, Consistency and Fault Tolerance

6.1 Replication

Replication involves creating and maintaining copies of services and data provided by a distributed system. Unlike communication, without which it is impossible to build a distributed system, replication is not a fundamental principle. This means that it is possible to build a distributed system that does not make use of replication.

When considering the replication of services, there are two types of replication possible: data replication and control replication. In the first case, only a service's data is replicated. Processing and manipulation of the data is performed either by a non-replicated server, or by clients accessing the data. Data are generally replicated to enhance reliability or improve performance. One of the major problems is keeping replicas consistent. Informally, this means that when one copy is updated we need to ensure that the other copies are updated as well; otherwise the replicas will no longer be the same.

6.1.1 Reasons for Replication

In the very beginning chapters, particularly in Chapter 1, we are introduced to some important concepts of distributed systems, such as the major characteristic and the challenges rose together with while achieving the goals. For example the primary advantages of replicating data are *performance, availability, fault tolerance* and *reliability.*

Increasing Availability, Reliability and Fault Tolerance

First, data are replicated to increase the *reliability* of a system. If a file system has been replicated it may be possible to continue working after one replica crashes by simply switching to one of the other replicas. Also, by maintaining multiple copies, it becomes possible to provide better protection *against corrupted data*. For example, imagine there are three copies of a file and every read and write operation is performed on each copy. We can safeguard ourselves against a single, failing write operation, by considering the value that is returned by at least two copies as being the correct one.

Enhancing Performance

The other reason for replicating data is *performance*. Replication for performance is important when the distributed system needs to scale in numbers and geographical area. Scaling in numbers occurs, for example, when an increasing number of processes needs to access data that are managed by a single server. In that case, performance can be improved by replicating the server and subsequently dividing the work.

Scaling with respect to the size of a geographical area may also require replication. The basic idea is that by placing a copy of data in the proximity of the process using them, the time to access the data decreases. As a consequence, the performance as perceived by that process increases. This example also illustrates that the benefits of replication for performance may be hard to evaluate. Although a client process may perceive *better performance*, it may also be the case that more network bandwidth is now consumed keeping all replicas up to date.

6.1.2 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.

A possible trade-off that needs to be made is that keeping copies up to date may require more network bandwidth. If the copies are refreshed more often than used (low access-to-update ratio), the cost (bandwidth) is more expensive than the benefits; not all updates have been used.

Replication itself is a subject to serious scalability problems intuitively, a read operation made on any copy should return the same value (the copies are always the same). Thus, when an update operation is performed on one copy, it should be propagated to all copies before a subsequent operation takes places: this is sometimes called tight consistency (a write is performed at all copies in a single atomic operation or transaction). But it's difficult to implement since that all replicas first need to reach agreement on when exactly an update is to be performed locally (for e.g., by deciding a global ordering of operations and this takes a lot of communication time).

Keeping copies consistent requires also global synchronization which is generally costly in terms of performance. So the solution is to loosen the consistency constraints, for instance:-

- Updates do not need to be executed as atomic operations (no more instantaneous global synchronization); but copies may not be always the same everywhere.
- To what extent the consistency can be loosened depends on the specific application (the purpose of data as well as access and update patterns).

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. First the data could be stale, that is, the data at some replicas has not been updated, while others have. Staleness is typically measured using time or versions. The other type of inconsistency occurs when operations are performed in different orders at different replicas. This can cause more problems than staleness because applying operations in different orders can lead to different results that cannot be consolidated by simply ensuring that all replicas have received all updates.

Exactly when and how those modifications need to be carried out determines the price of replication.

To understand the problem, consider improving access times to Web pages. If no special measures are taken, fetching a page from a remote Web server may sometimes even take seconds to complete. To improve performance, Web browsers often locally store a copy of a previously fetched Web page (i.e., they cache a Web page). If a user requires that page again, the browser automatically returns the local copy. The access time as perceived by the user is

excellent. However, if the user always wants to have the latest version of a page, s/he may be in for bad luck. The problem is that if the page has been modified in the meantime, modification will not have been propagated to cached copies, making those copies out-of-date.

One solution to the problem of returning a stale copy to the user is to forbid the browser to keep local copies in the first place, effectively letting the server to be fully in charge of replication. However, this solution may still lead to poor access times if no replica is placed near the user. Another solution is to let the Web server invalidate or update each cached copy, but this requires that the server keeps track of all caches and sending those messages. This, in turn, may degrade the overall performance of the server.

In the following we concentrate on consistency with regards to the ordering of operations at different replicas.

6.2.1 Consistency Models

In a non-distributed replica/data store the program order of operations is always maintained (i.e., the order of writes as performed by a single client must be maintained). Likewise, data coherence is always respected. This means that if a value is written to a particular data item, subsequent reads will return that value until the data item is modified again. Ideally, a distributed replica/data store would also exhibit these properties in its total ordering. However, implementing such a distributed data store is expensive, and so weaker models of consistency that are less expensive to implement) have been developed. 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

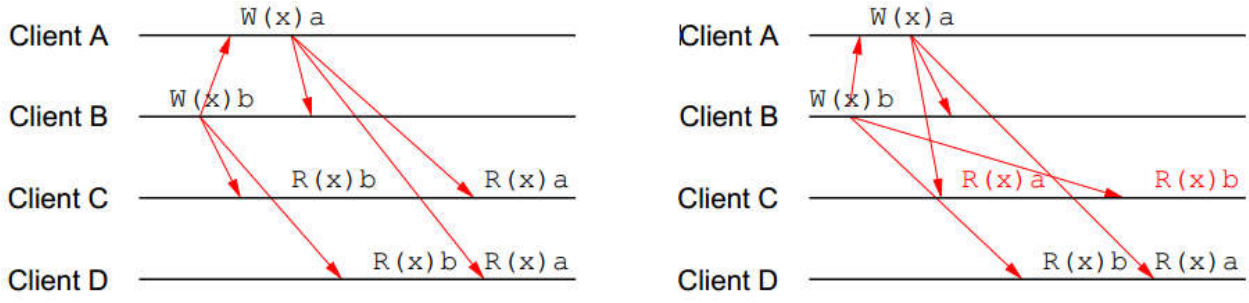
The first, and most widely used, class of consistency models is the class of data-centric consistency models. These are consistency models that apply to the whole data store. This means that any client accessing the replicas/data store will see operations ordered according to the model.

Strict Consistency: The strict consistency model requires that any read on a data item returns a value corresponding to the most recent write on that data item. This is what one would expect from a single program running on a uniprocessor. A problem with strict consistency is that the interpretation of ‘most recent’ is not clear in a distributed data store.

Linearisable consistency model: In this model the requirement of absolute global time is dropped. Instead all operations are ordered according to a timestamp taken from the invoking client’s loosely synchronized local clock. Linearisable consistency requires that all operations be ordered according to their timestamp.

Sequential Consistency: The sequential consistency model drops the time ordering requirement. In a data store that provides sequential consistency, all clients see all (write) operations performed in the same order. However, unlike in the linearisable consistency model where

there is exactly one valid total ordering, in sequential consistency there are many valid total orderings. The only requirement is that all clients see the same total ordering.



Sequential **not sequential**

Figure 6.1: An example of a valid and an invalid ordering of operations for the sequential consistency model.

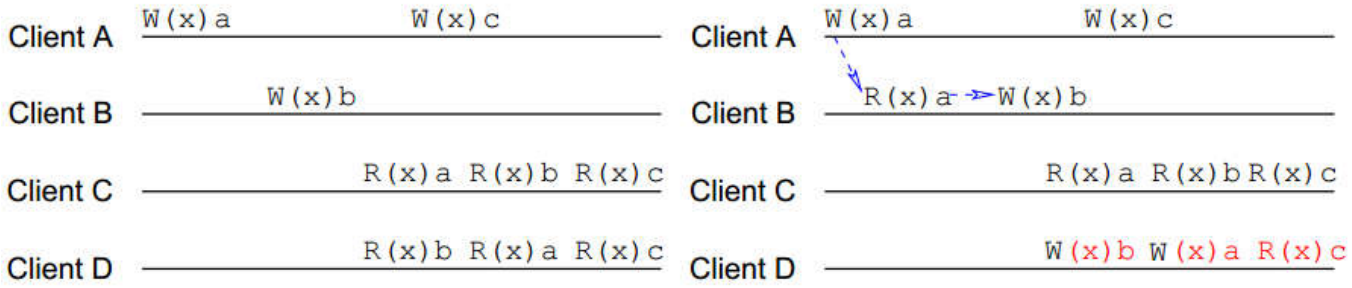
In the example of invalid ordering the two write operations are executed in a different order on the replicas associated with client C and client D. This is not admissible with the sequential consistency model.

Causal Consistency: The causal consistency model weakens sequential consistency by requiring that only causally related write operations are executed in the same order on all replicas.

Specifically 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.



Causally consistent **not causally consistent**

Figure 6.2: An example of a valid and an invalid ordering of operations for the causal consistency model.

In the example of an invalid ordering we see that the write performed by client B (W(x)b) is causally related to the previous write performed by client A (W(x)a). As such, these writes must appear in the same (causal) order at all replicas. This is not the case for client D where we see that W(x)a is executed after W(x)b.

FIFO Consistency: The FIFO consistency model weakens causal consistency in that it removes limitations about the order of any concurrent operations. FIFO consistency requires only that any total ordering respect the partial orderings of operations (i.e., program order).

FIFO consistent **not FIFO consistent**

Figure 6.3: An example of a valid and an invalid ordering of operations for the FIFO consistency model.

In the invalid ordering example client D does not observe the writes coming from client A in the correct order (i.e., W(x)c is executed before W(x)a).

Reading assignment: Weak Consistency, Release consistency and Entry consistency model

2. Client-Centric Consistency Models

The data-centric consistency models had an underlying assumption that the number of reads is approximately equal to the number of writes, and that concurrent writes occur often. Client centric consistency models, on the other hand, assume that 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. Client-centric consistency models 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. For the discussion of client-centric consistency models we extend the data store model and notation somewhat. The change to the data store model is that the client can change which replica it communicates with (i.e., the client is mobile). We also introduce the concept of a write set (WS). A write set 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.

1. **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.

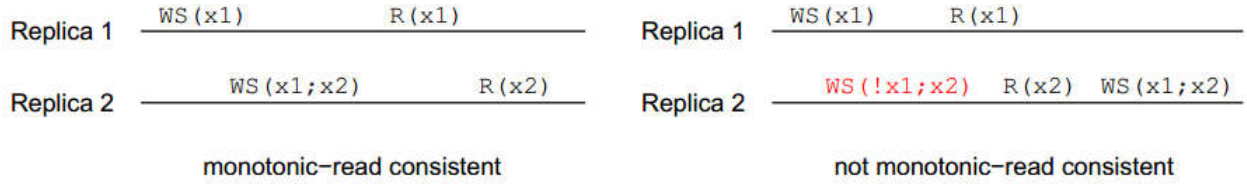


Figure 6.4: An example of a valid and invalid ordering for the monotonic reads consistency model.

As it is observed from the figure, there is an invalid ordering for monotonic reads. This ordering is invalid because the write set at the second replica does not yet contain that from the first.

2. **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.

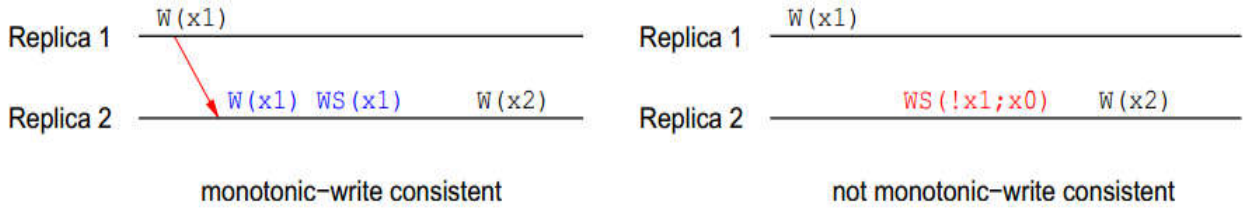


Figure 6.5: An example of a valid and invalid ordering for the monotonic writes consistency model.

The example of the invalid ordering shows that the write performed at replica 1 has not yet been executed at replica 2 when the second write is performed at that replica.

3. **Read your writes model:** In this model, a client is guaranteed to always see its most recent writes.

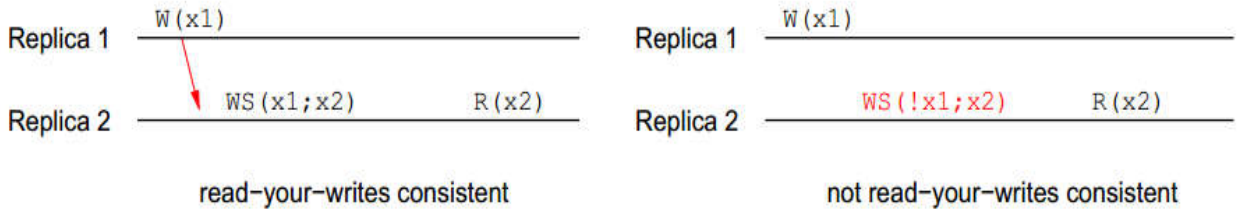


Figure 6.6: An example of a valid and invalid ordering for the read your writes consistency model.

As it is shown on the above figure, the client does not see its most recent write at another replica. In this case, the write set at replica 2 does not contain the most recent write operation performed on replica 1.

4. **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.

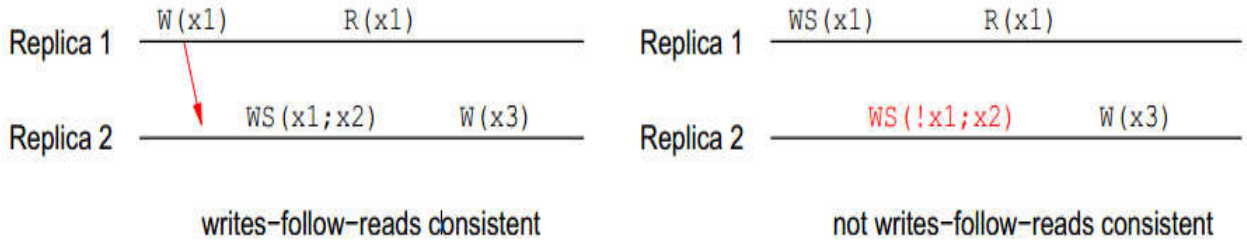


Figure 6.7: An example of a valid and invalid ordering for the write follows reads consistency model

In the example of the not write follows reads ordering, the two replicas do not have the same write set (and the one on replica 2 is also not newer than the one on replica 1). This means that the read and the write operations are not performed on the same state.

6.2.2 Consistency Protocols

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. In contrast, the 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. In remote-write protocols writes are possibly executed on a remote replica. In local-write writes are always executed on the local replica.

Single Server: The first of the remote-write protocols is the single server 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. Note that due to the lack of replication, this protocol does not provide a distributed system with reliability.

Primary-Backup: The primary-backup protocol 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. Furthermore, while it is possible to make the write non-blocking, greatly improving performance, such a system would no longer guarantee sequential consistency.

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. However, when a data item is accessed it is moved from its original location to the replica of the client accessing it. The benefit of this approach is that data is always consistent and repeated reads and writes occur quickly, with no delay. The drawback is that 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. There are many techniques for doing this including broadcast, forwarding pointers, name services, etc.

Migrating Primary (multiple reader/single writer): An improvement on the migration protocol is to 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. Before disconnecting from the network the client becomes the primary allowing it to perform updates locally. When the client reconnects to the network it updates all the backups.

Active Replication: The active replication protocol is a replicated write protocol. 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. The benefit of this approach is that all replicas receive all operations at the same time (and in the same order), and it is not necessary to track a primary, or send all operations to a single server.

Quorum-Base: Protocols with quorum based protocols 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.

This protocol 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 characteristic feature of distributed systems that distinguishes them from single-machine systems is the notion of partial failure. A partial failure may happen when one component in a distributed system fails. This failure may affect the proper operation of other components, while at the same time leaving yet other components totally unaffected. In contrast, a failure in non-distributed systems is often total in the sense that it affects all components, and may easily bring down the entire system.

An important goal in distributed systems design is to construct the system in such a way that it can automatically recover from partial failures without seriously affecting the overall performance. In particular, whenever a failure occurs, the distributed system should continue to operate in an acceptable way while repairs are being made, that is, it should tolerate faults and continue to operate to some extent even in their presence.

6.3.1 Fault Tolerance Basic Concepts

Fault tolerance is strongly related to ***dependable systems*;** and this dependability in turn covers the following four basic concepts:-

- ✓ *Availability*: - Refers to the probability that the system is operating correctly at any given time; it is defined in terms of an instant in time.
- ✓ *Reliability*: - A property that a system can run continuously without failure; it is defined in terms of a time interval.
- ✓ *Safety*: - Refers to the situation that even if a system temporarily fails to operate correctly, nothing catastrophic happens.
- ✓ *Maintainability*: - Refers how easily a failed system can be repaired.

Often, dependable systems are also required to provide a high degree of *security*, especially when it comes to issues such as *integrity*.

A system is said to *fail* when it cannot meet its promises, for instance failing to provide it users one or more of the services it promises. An error is a part of a system's state that may lead to a failure, for example packets damaged when transmitting across a network before they arrive at the receiver.

The cause of an error is called a ***fault*** and finding out what caused an error is important. Fault can be due to wrong or bad transmission medium which is relatively easy to remove the fault or transmission errors caused by bad weather conditions such as in wireless networks. So building dependable systems closely relates to controlling faults (i.e.; preventing, removing, and forecasting faults).

Generally, a fault tolerant system is a system that can provide its services even in the presence of faults. Faults are classified into three:-

(a) *Transient*: occurs once and then disappears. If the operation is repeated, the fault goes away. For e.g., a bird flying through a beam of a microwave transmitter may cause some lost bits.

(b) *Intermittent:* it occurs, then vanishes on its own accord, then reappears, and so on (e.g., a loose connection). It is difficult to diagnose like taking yourself to the nearest clinic, but does not show any sickness by the time you reach there.

(c) *Permanent:* one that continues to exist until the faulty component is repaired for e.g., disk head crash, software bug, etc.

6.3.2 Failure Models

A system that fails is not adequately providing the services it was designed for. If we consider a distributed system as a collection of servers that communicate with one another and with their clients, not adequately providing services means that servers, communication channels, or possibly both, are not doing what they are supposed to do. However, a malfunctioning server itself may not always be the fault we are looking for. If such a server depends on other servers to adequately provide its services, the cause of an error may need to be searched for somewhere else. Such dependency relations appear in abundance in distributed systems.

There are several classification schemes of failure, five of them are:

1) *Crash failure*: a server halts, but was working correctly until it stopped.
2) *Omission failure*: a server fails to respond to incoming requests.
   - Receive omission: a server fails to receive incoming messages; e.g., may be no thread is listening.
   - Send omission: a server fails to send messages.
3) *Timing failure*: a server's response lies outside the specified time interval; e.g., may be it is too fast over flooding the receiver or too slow.
4) *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.
5) *Arbitrary failure* (or *Byzantine failure*): a server may produce arbitrary responses at arbitrary times (most serious).

6.3.3 Failure Masking by Redundancy

To be fault tolerant, the system tries to hide the occurrence of failures from other processes: this is referred as *failure masking*. The key technique for masking faults is to use redundancy; and the three possible kinds are listed below:-

- *Information redundancy* - add extra bits to allow recovery from garbled bits (error correction).
- *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).

6.3.4 Failure Recovery

The term 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)).

While restarting processes or computers is a relatively straightforward exercise in a centralized system, things are (as usual) significantly more complicated in a distributed system. The main challenges are:

Reclamation of resources: a process may hold resources, such as locks or buffers, on a remote node. 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. In order to achieve consistency it is, in general, necessary to undo partially completed operations on other nodes prior to restarting.

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). Forward recovery implies the ability to completely assess the nature of all errors and damages resulting from the faults that lead to failure. 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.

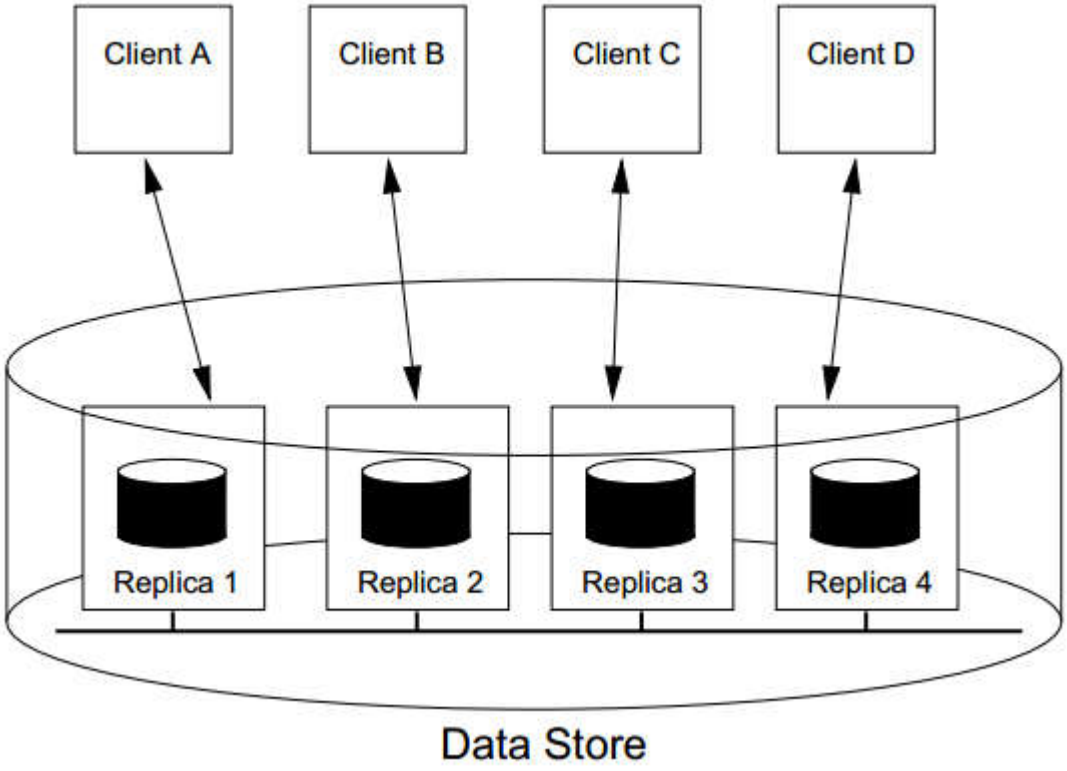
The alternative is backward error recovery. This restores the process or system state to a previous state known to be free from errors, from which the system can proceed (and initially retrace its previous steps). Obviously this incurs overheads due to the lost computation and the work required to restore the state. Also, there is in general no guarantee that the same error will not reoccur. Furthermore, there may be irrecoverable components, such as external input (from humans) or irrevocable outputs (e.g. cash dispensed from an ATM). 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.

Let's look at backward recovery in detail.

Backward recovery: Backward error recovery works by restoring processes to a recovery point, which represents a pre-failure state of the process. A system can be recovered by restoring all its active processes to their recovery points. 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**.

Reading assignment

1. Kinds of replicas (Permanent, server-initiated and client-initiated)
2. How updates are propagated to other replicas?
3. Process resilience
4. Reliable client-server and group communication (Covered on chapter 4)
5. Distributed commit



The Data store model