Unit 5 – Failure Management

Student Guide

Introduction

Transparency is one of the objectives of every distributed system. There are different types of transparency to be provided in a distributed system. Failure transparency is one of these types. With failure transparency, users should not be aware of failures in system resources or application components. In order to obtain such behaviour, we need to replicate all components or resources that might fail. Thus, when one of those replicas fails, another one will replace it, ensuring that the functionality being expected by users is still provided.

This unit describes what a failure is, explaining which failure models may be assumed when a distributed algorithm is being designed. It also presents dependability and the mechanisms that are needed for providing it. Replication is one of those mechanisms; it provides the basis for building reliable, available and maintainable components. There are two classical replication models. They are also described in this unit, discussing which model is appropriate for each kind of component, depending on the computing interval and the amount of updated state generated by each of its operations.

When there are multiple replicas of a given element, there might be divergences among the state of those replicas due to the interval needed for propagating and applying state updates in each replica. Consistency models characterise the degree of divergence that may arise, depending on the update propagation protocols being used in a replicated service. This unit presents some consistency models, analysing the effort needed to comply with each of these models. When scalability is the main objective, the consistency model to be selected should minimise replica coordination. This usually leads to a relaxed consistency.

1. Failures: Concept and Models

Informally, a failure happens when a system component is unable to behave according to its specification. This happens, for instance, when a component cannot provide any answer to the incoming requests, when those answers are always incorrect or when the answer is provided too late.

Let us revise this informal definition, presenting later the existing failure models.

1.1. Failure Concept

In order to understand what a failure is and why failures happen, three different stages of "unexpected behaviour" should be distinguished. These three stages are [1]:

- *Fault*: There is a fault when some unexpected event occurs; i.e., something that was not considered when the faulty component was designed. A fault is an anomalous condition. As a result, the affected component is unable to react correctly against that event. Examples: unexpected input, interferences, power outage, design errors...
- *Error*: An error is the manifestation of a fault in a system. With errors, the state of some system component differs from what is expected considering the received input and the execution that has been completed up to now. Examples: to provide an incorrect result, to be unable to answer a request...
- *Failure*: There is a failure when an element is unable to execute its intended functions due to errors in the element or in its environment, caused by some faults.

In some cases, the observed errors are transient, originated by a fault that is quickly fixed. In those cases there is no failure.

If a failure happens, this definition says that such failure was produced by a first fault that the system was unable to manage, thus generating an error that corrupted some component. With replicated components, this chain of events will not be observable by the user, since the inoperative component replica is replaced by a correct one. In this case, the fault in a component had generated an error in that component but it did not produce a failure in the entire set of component replicas.

As a result, a failure is the consequence of two previous transitions: from fault to error, and from error to failure. We should design mechanisms that avoid that chain of transitions. This is shown in Figure 1.

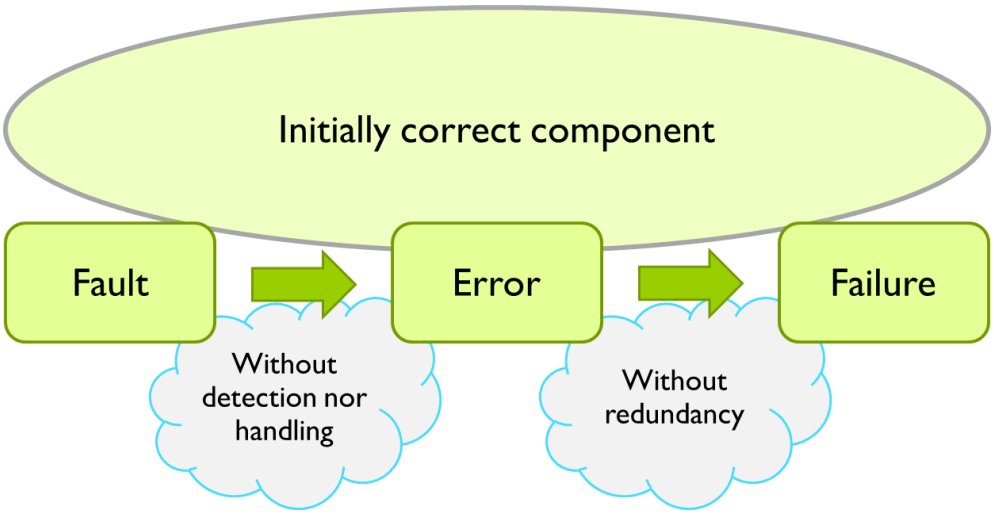

Figure 1: Transitions when a failure is generated.

Although replication is able to mask errors, avoiding that users perceive the failures in a component replica, there are some tasks that are needed for managing these events. The first of these tasks is an adequate detection and diagnosis. This implies that the system should notice that there has been an error in a replica (detection), find out its cause (diagnosis) and take the appropriate actions to avoid the faults that caused such error. Usually, this leads to stopping the replica, repairing its faulty elements, reconfiguring the replica and restarting or recovering it. Recovery means that once a replica has been repaired and reconfigured (eliminating the problem that caused its fault), the system needs to transfer all the state updates that it lost (i.e., such updates had been applied in all other replicas but not in that one) in such inactivity interval. Once recovery is terminated, this replica will maintain a state coherent with that of other replicas. At this time, it is able to serve again the incoming requests in a regular way.

We have centred our discussion in failures. In some cases, the literature on this field talks about fault-tolerance. At a glance, it might seem a different subject, but it is not. The aim is the same: faults may arise, but they should not generate any failure. As a result, fault-tolerance and failure transparency are two complementary views of the same property.

It is worth noting that a service A could not be fault-tolerant if it depends on an external non-fault-tolerant service B. Note that a fault on B might generate a failure on B which would cause also failures on A. For instance, A might request some operation to B. As B fails, it is unable to answer that request. As a result A remains blocked waiting for the answer from B and might be also unable to answer its own requests.

1.2. Failure Models

Multiple kinds of failure may arise while a distributed application is executed. These types depend on the faults that arise and on the existing fault management mechanisms. Scalable and highly available distributed systems should manage multiple replicas of their components. So, they are deployed on multiple computers. Some of these nodes may fail when we consider a long time interval. As a result, we could find multiple types of failures and those failures may affect a variable number of system nodes.

In the field of distributed computing theory, there have been multiple proposals of failure classifications, identifying multiple models [1] [2] [3]. We talk about "models" [4] because a model does not consider any concrete details. Instead, models are defined considering only the main aspects (i.e., those that are general and applicable to all scenarios) and using a high level of abstraction. The aim is an adequate characterisation of each failure variant.

Schneider [4] proposes a classification based on the following failure models (considering only a single processor or a single communication link):

- *Failstop*: A processor fails by halting. Once it is halted, it remains in that state. This fact may be detected by other processors.
- *Crash*: A processor fails by halting. Once it is halted, it remains in that state. This fact may not be detectable by other processors.
- *Crash and link*: A processor fails by halting. Once it is halted, it remains in that state. A communication link fails by losing some messages, but it does not delay, duplicate or corrupt the messages it delivers.
- *Receive omission*: A processor fails by receiving only a subset of the messages that have been sent to it or by halting and remaining halted.
- *Send omission*: A processor fails by sending only a subset of the messages it should send or by halting and remaining halted.
- *General omission*: This model combines receive omission and send omission.
- *Arbitrary (or Byzantine)*: A processor fails by exhibiting an arbitrary behaviour.

This list is also ordering the models. The first listed model (failstop) is the one that offers the easiest environment for designing distributed algorithms. It provides the most favourable scope. On the other hand, failstop is also the failure model with the most difficult support. A system that tries to ensure this failure model should guarantee that faulty components should be halted before other processes interact with them.

On the other hand, an arbitrary failure model complicates algorithm design but it does not require any effort from the underlying software (i.e., OS or middleware) in order to support it.

None of these models discusses recovery. There is no problem with this. Dependable systems ensure high availability replicating their agents. When a replica fails, sooner or later it will be recovered. The failstop model seems to forbid recovery ("a processor fails by halting; once it is halted, it remains for ever in that state") but this is not true. In order to recover a replica we will need to start a new process providing such functionality. Such new process will have its own identity. As a result, the failed process remains halted for ever. A different (and new) one will replace it.

Regarding replicated components, some care should be taken about the nodes to be selected for deploying their set of replicas. Such nodes should be independent regarding failures. This means that when there is a failure, the number of nodes and replicas affected by such failure should be minimal. For instance, those computers should be attached to different segments of the electric grid in order to avoid a power outage in all of them.

If all the replication mechanisms are correctly chosen, problems in a given replica will not cause that the replicated component fails, and such replica failure will remain unnoticed by all other system components. To this end, the interaction between replicas of a service A with replicas of another service B should be dynamic. If a replica of B crashes, the replicas of A that were using it should transparently forward their requests to another replica of B. This ensures failure contention.

At the moment we have only considered failures in a single process or a single communication link, but this discussion should be extended to groups of nodes. In some cases a given distributed application is deployed onto multiple datacentres. It might happen that a link connecting two of those datacentres fails, being impossible the communication between replicas located in different centres. This is the scenario of a "network partition". In case of network partition a given subset of system nodes remains isolated from the other nodes. As a result, the system being considered is partitioned in two (or more) subgroups of nodes. Communication is possible among nodes located in the same subgroup but not between nodes located in different subgroups.

When a network partition occurs, there are two different strategies to manage it [5]:

- *Partitionable systems*. Each isolated subgroup may go on with its work. If there is at least a replica of a given component in each one of the resulting subgroups, those replicas will be unable to exchange messages. If each subgroup generates updates on the component state, each subgroup obtains a different state in that replicated component. This means that replica consistency is lost using this strategy.
  Partitionable systems need a reconciliation protocol for joining all the state updates applied in each subgroup when connectivity is restored. Reconciliation is trivial when all the operations executed in the partitioned system are commutative.
- *Primary subgroup* (also known as *primary component* or *primary partition*). In this model, only the subgroup with a majority of the system computers is allowed to continue. All the other subgroups should be halted. This strategy ensures that all users being served receive answers that are consistent.
  However, when a network partition arises, it may happen that there is no subgroup with a majority of nodes. For instance, in a system with 40 nodes a network partition

could create three subgroups with 18, 12 and 10 nodes each one. None of these subgroups is a valid “primary”. As a result, the activity in this system would be forbidden.

2. Replication

As it has already been suggested in the previous sections when failures and recovery were described, replication is the basic mechanism for ensuring failure transparency and for guaranteeing the service availability in a distributed system.

Multiple aspects should be considered in order to replicate a service:

- Its replicas should be located in computers that do not depend on the same potential source of failures. For instance, electrical power is critical for maintaining an element in execution. The computers being used should not be located in the same segment of the electrical grid.
- The dependencies among the distributed system components should be revised with care. When a component replica has failed, such failure should be isolated, hiding it from the rest of the system components. The requests being processed in the faulty replica must be sent again to any other correct replica, resending them automatically; i.e., without needing the intervention of any system administrator.

Besides its contribution to failure management, replication also provides benefits when performance or scalability is considered. Read-only operations may be executed in a single replica. As a result, requests using these operations may obtain a linear scalability in a replicated service.

Updating operations are more complex. Requests using them should regularly reach all service replicas. This demands a higher effort for request transmission and request service. In these cases, scalability is not improved due to replication. Hopefully, in many services most of their operations are read-only.

Although there are other options (e.g., dynamic quorums) that will not be described in this document, the strategy commonly used in current distributed systems [6] consists in using a single replica for read-only operations and all replicas for updating operations.

The protocol being used for managing updating requests depends on the replication model and the type of instructions being used for implementing those operations.

Brief operations that update a high percentage of the state should be propagated to all service replicas and directly executed in each replica. This avoids any state propagation. Since request messages are generally smaller than update propagation messages, this strategy reduces bandwidth usage and service time.

On the other hand, when operations require a long computing interval but they only update a small part of the component state, a different strategy is needed. In this case the request is only forwarded to a single replica. Such replica locally executes all the operation sentences, transmitting later the small updates to the remaining replicas. Thus the other replicas do not

need to invest a lot of time directly processing these requests, but only a short one applying the received updates.

We should carefully consider the sequence of operations followed in each replica for receiving and serving the incoming requests. If that sequence is not identical in all replicas, there will be state divergences at the end. The divergence level being tolerated defines the consistency model. These consistency models are explained in Section 4.

Let us see now the replication models that may be used for managing requests in a replicated service. The two classical models are passive [7] and active [8].

2.1. Passive Replication Model

In the passive replication model, clients send their requests to a single special replica: the primary replica. A passively replicated service only has a primary replica. This primary replica serves the request locally. Once finished, it propagates the updates to the other replicas (known as secondary or back-up replicas) and sends an answer to the client.

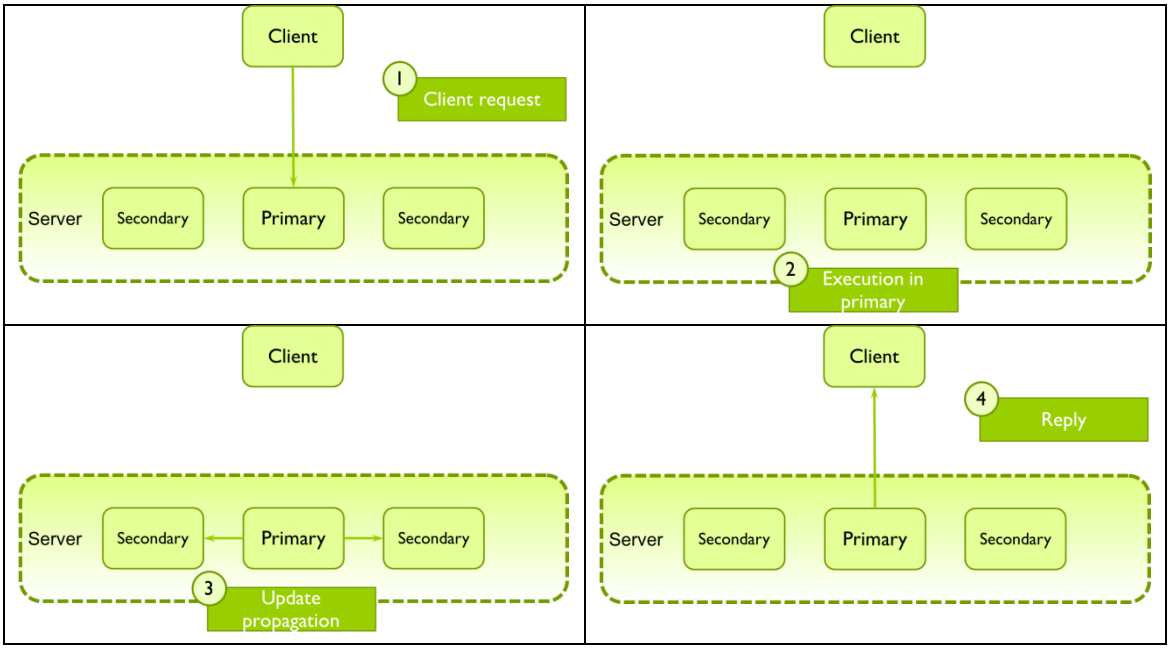

Figure 2. Operation steps in the passive model.

For read-only operations, the specification of the passive model [7] requires their forwarding to the primary replica. However, multiple systems admit that secondary replicas execute those operations, since this does not introduce inconsistencies and improves scalability.

Figure 2 summarizes the steps needed for executing an operation in this replication model. The passive model is appropriate for operations with a long computing interval that only update a small fragment of the component state. Thus, the primary replica is the single one responsible of the direct (and lengthy) execution of the requested operation, while secondary replicas do only need to apply the small updates. This does not require a high network bandwidth and allows a fast management in secondary replicas.

Advantages

The passive replication model provides the following advantages:

- Minimal workload. The heavy part in the management of each operation is assigned to the primary replica. Secondary replicas minimise their workload. This allows that in a system with N nodes, N different services may be passively replicated choosing a different primary replica for each of them.
- Trivial operation ordering. The primary replica labels each update message with a sequential number. Secondary replicas use that sequential labelling for respecting the same update application order in all of them. This ensures a strong consistency among all replicas of a given service.
- Local concurrency control. Multiple operations may be concurrently executed. To this end, the primary replica may start the execution of a new request before terminating the previous one. Only local concurrency control mechanisms (e.g., locks, semaphores, monitors...) are required for managing concurrency, since such control is only needed in the primary replica.
- Non-deterministic operations are admitted. An operation is qualified as "non-deterministic" when it may generate different results in multiple requests of such operation that have used the same argument values. This is generally caused by using selection instructions (i.e., conditionals, switches, etc.) that do not base their decision exclusively in the input arguments that have been received.

  Since in the passive model only the primary replica is executing the updating operations, their results will be received and applied in a blind way by all other replicas. No divergence may arise even when non-deterministic operations are used.

Inconveniences

On the other hand, the passive model has the following drawbacks:

- The failure of the primary replica requires a complex recovery. A new primary replica should be chosen deterministically. Moreover, client processes should be reconfigured updating their pointer to the (new) primary replica. This is achieved receiving some kind of exception when they try to use the address of the failed primary replica. Clients react to this event with a request to a name service for obtaining the address of the newly selected primary replica. This mechanism could be embedded in the code of the client stubs or proxies, being transparent to the client programs.
- The Byzantine failure model is not tolerated. There is no means for detecting when the answers of a primary replica are correct.
- In some failure models, a passive strategy requires a high number of replicas for ensuring service continuity. For instance, in the general omission failure model, when "f" simultaneous failures may arise[1], we will need at least 2f+1 replicas in order to ensure a correct behaviour [7]. Note that a correct passively replicated service should always have a single primary replica.

  Let us see what happens with 2f replicas. Those replicas could be structured in two different disjoint sets A and B, each one with f replicas.
  - If in a first execution the f replicas in A failed, then a primary is selected in B.
  - If in a second execution f replicas in B failed, then a primary is selected in A.

[1] When "f" general omission failures happen, the "f" faulty processors are unable to communicate with "f" other processors.

- If in a third execution the f replicas in A had a general omission failure in the channels that communicate them with the nodes in B, replicas in B would behave as in the first execution. On the other hand, replicas in A would behave as in the second execution. As a result, we would find two different primary replicas: one in A and another in B, violating the requirements of the passive replication model.

In order to avoid this situation we would need at least another replica that intercommunicates the nodes in A with the nodes in B. This would prevent the selection of two primaries from happening in the third execution.

2.2. Active Replication Model

In the active replication model [8], clients directly forward their requests to all service replicas. Each server replica executes the requested operation. When it terminates, sends a response to the client. The steps being followed in this replication model are shown in Figure 3.

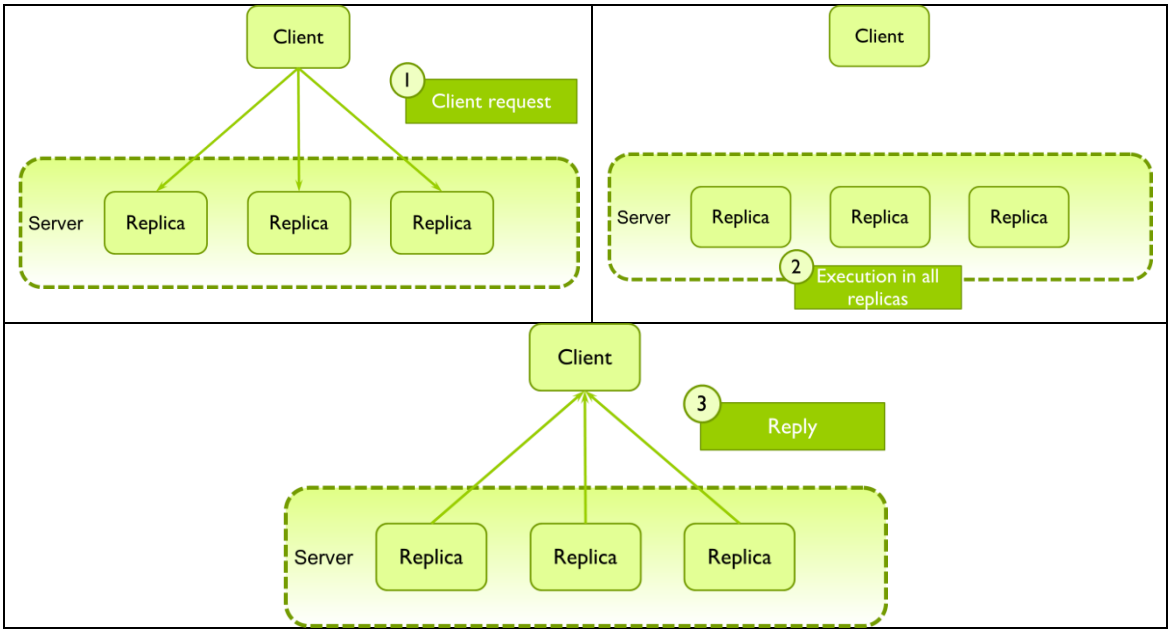

Figure 3. Operation steps in the active replication model.

In this strategy only three steps are needed, while in the passive model there were four steps. However, the first step in this active model requires a total-order multicast for ensuring the same state in all replicas. The protocol that ensures this total order is complex, requiring a large amount of messages. It includes a solution for the distributed consensus problem. In Unit 4 we already saw that such problem has no solution in an asynchronous distributed system where processes may fail [9].

In the third step, the client obtains an answer from each server replica. Despite the potential large amount of messages, this is not a serious drawback as we will see on the sequel.

Advantages

When it is compared with the passive model, the active replication model provides two important advantages:

- Reconfiguration of a replicated service in case of failure is immediate. While there are enough live replicas, there is no task to be done. Since all replicas are identical, there is no need for promoting or activating replicas. All surviving replicas will behave in the regular way. As a result, there will not be any unavailability intervals caused by failures in this model.
- This replication model tolerates Byzantine failures. In a favourable environment (a partially synchronous system where message senders may certify their identity) "f" simultaneous arbitrary failures can be tolerated using 2f+1 replicas. When a client receives the request answers, assuming no failures, all replies will be identical. In case of arbitrary failures, faulty replicas would return an arbitrary result. Assuming a maximum of "f" simultaneous failures, 2f+1 replicas are used and each client waits for receiving f+1 identical (i.e., correct) answers.
  Note that with other more relaxed failure models, a lower amount of replicas is needed. For instance, assuming failstop failures, f+1 replicas suffice and the client is only waiting for the first answer.

Inconveniences

In spite of the advantages described above, the active replication model has some drawbacks:

- If we need a strong consistency among service replicas, the requests being propagated by clients should be received in total order in all replicas; i.e., all replicas should see the same request sequence.
- Non-deterministic operations generate inconsistencies in this replication model. Since all replicas execute all the operations in a direct way, non-deterministic operations could generate different results in each replica. This provokes state divergences among the states of the service replicas.
- If an actively replicated service A (with N replicas) invokes an operation of another actively replicated service B, we should filter in B the N replicas of the same request, accepting only one of those identical requests. Otherwise each operation would be executed N times in each replica of B.

2.3. Comparison

Let us analyse which replication model manages in the best way each aspect to be considered when we deploy a replicated service. Later, we extend what was previously mentioned about which model provides best response time depending on the request service time, the size of the updated state and the state transfer time.

Aspect	Passive model	Active model
Processing replicas	**1**	All
Avoidance of distributed request ordering	**Yes**	No
Avoidance of update transfer	No	**Yes**
Admits indeterminism	**Yes**	No
Tolerates arbitrary failures	No	**Yes** (It needs that each reply process also the readings)
Consistency	At least, sequential	At least, sequential
Recovery in case of failure	Election and reconfiguration when the primary fails	**Immediate**

Table 1.Comparison of replication models

General aspects

The main advantages and problems of each model can be shown in Table 1, where the best alternative is highlighted in bold.

In addition to this comparison among the aspects to be considered, it is also convenient to present some use cases in order to set a general recommendation on which replication model provides the best performance.

Case 1. Scenario 1

We assume a service S1 with these characteristics:

- 5 replicas
- Network bandwidth: 1 Gbps
- Propagation time in the communication channels: 2 ms. Valid as transfer time for small messages.
- The average operation response time is 200 ms, but it does not assume concurrence. When N requests are processed concurrently, then the response time becomes 200*N.
- The average size of the modifications made in each operation is 10 KB; i.e., 80 Kb = 0.08 Mb = 0.00008 Gb. With this, the update transfer time becomes (considering that it includes both propagation and transmission of messages) 0.002 + 0.00008 s = 2.08 ms
- The update application time in backup replicas is 3 ms.

With these data, which replication model will have a minimal response time? Let us see...

Aspect	Passive model		Active model
	Primary replica	Secondary replica	
Request delivery	2 ms	--	4 – 6 ms (assuming a sequencer-based total order multicast algorithm)
Request processing	200 ms	--	200 ms
Update transfer	2.08 ms		0 ms
Update application	--	3 ms	0 ms
Update acknowledgement	2 ms		0 ms
Reply to client	2 ms	--	2ms
TOTAL:	211.08 ms		206 – 208 ms

In this example, the passive model needs an average of 211.08 ms to manage each request. Initially, it needs 2 ms to transfer the request messag to the primary replica. We assume that the client is in the same network than the servers. This initial interval may be longer in other cases. Later, the primary replica takes 200 ms in order to process this request. Then, it transfers the generated updates to the other replicas. Let us assume that there are independent channels for each replica and those transfers may be made in parallel. Backup replicas use 3 ms to apply these updates on their local state. Later, they use 2 ms to transfer an acknowledgement to the primary. Once those acknowledgements have been received, the

primary replica replies to the client, needing 2 ms to this end. Therefore, we have used 211.08 ms to complete all these steps.

In the active model, the request message may be sent to a single delegate replica, who will broadcast it to the other replicas. This type of broadcast is known as a sequencer-based broadcast, and it is the fastest one from all the existing alternatives. The sequencer behaves as a central point of ordering for all messages that should be spread. There are other variants that use three rounds of message propagation instead of two. Thus, this initial broadcast needs from 4 to 6 ms. Later, each replica uses 200 ms in the request processing stage and 2 ms to reply to the client. With this, the time invested in request management ranges from 206 to 208 ms (assuming a minimum load, as in the passive model).

Case 1. Scenario 2

At a glance, the results of the previous scenario are similar in both models. However, we have to consider that each replica has been deployed in a different computer. If we deployed five services of type S1 in these computers, each service would have its primary replica in a different computer in the passive replication model, but the active model would endure a load five times greater in all its replicas.

With this, the new results would be now:

Aspect	Passive model		Active model
	Primary replica	Secondary replica	
Request delivery	2 ms	--	4 – 6
Request processing	200 +4*3 = 212 ms	--	200*5=1000 ms
Update transfer	2.08 ms		0 ms
Update application	--	3 ms	0 ms
Update acknowledgement	2 ms		0 ms
Reply to client	2 ms	--	2ms
TOTAL:	223.08 ms		1006 – 1008 ms

Thus, the passive model is clearly more efficient in this second scenario.

Case 2. Scenario 2

We assume a service S1 with these characteristics:

- 5 replicas
- Network bandwidth: 1 Gbps
- Propagation time in the communication channels: 2 ms. Valid as transfer time for small messages.
- The average operation response time is 5 ms, but it does not assume concurrence. When N requests are processed concurrently, then the response time becomes 5*N.
- The average size of the modifications made in each operation is 10 MB; i.e., 80 Mb = 0.08 Gb. With this, the update transfer time becomes (considering that it includes both propagation and transmission of messages) 0.002 + 0.08 s = 82 ms
- The update application time in backup replicas is 3 ms.

With these data, which replication model will have a minimal response time? Let us compute it assuming that we deploy five different services of this class S2 in a same set of five machines...

Aspect	Passive model		Active model
	Primary replica	Secondary replica	
Request delivery	2 ms	--	4 – 6 ms
Request processing	5 + 4*3=17 ms	--	5*5=25 ms
Update transfer	82 ms		0 ms
Update application	--	3 ms	0 ms
Update acknowledgement	2 ms		0 ms
Reply to client	2 ms	--	2ms
TOTAL:	108 ms		31 – 33 ms

In this second case, the active model is the clear winner.

Analysis

Considering their response times shown above, the replication model to be chosen depends mainly on two factors:

- The average request service time. The passive model is appropriate when the processing interval is long, as only it affects to its primary replica, whereas in the active model affects to all its replicas.
- The size of the modifications. The active model is especially adequate for those services that modify a large state, as these modifications do not need to be packed in messages, transferred and applied in other replicas. On the other hand, the passive model is only acceptable in services that update a very small state, as in this model it is necessary to pack these modifications, transfer and apply them in backup replicas.

3. Consistency

When data is replicated in multiple processes, a consistency model specifies which data divergences are admitted among the values of the replicas of a given data element.

When a consistency model is being specified it is common to assume that:

- One of the processes writes on the data element.
- Such write operation is propagated to the other replicas.
- Each one of the other processes is able to read that new value afterwards. To this end, they will read their local copies.

The differences among consistency models consist in how the propagation delays are being managed in each model and also in the consistency conditions required in each of them.

Although there are multiple proposals for classifying consistency models, in this course we will only study two alternatives:

- Data-centred consistency models [10] [11] [12], limiting their study to those models that do not use any synchronisation tools.

- The concept of "*eventual consistency*" [13].

3.1. Data-centred Consistency Models

If we assume that no synchronisation tools are used, the main consistency models to be considered are: strict, sequential, causal, processor, FIFO (also known as pRAM) and cache. Let us describe each one of them in the following paragraphs.

Strict Consistency

Strict consistency [14] assumes that write propagation is immediate and while a process is executing a write no other process could be executing any other write. This is the model that will be obtained in a single computer with a single processor.

Its practical implementation in a distributed system requires excessive blocking steps, generating an inefficient system.

In order to guide the reader on the implications of a strict consistency model, we may use time intervals. Each time some process executes a write (for instance, to write value V on a variable "x"), such action is closing the current interval (where variable "x" still had another value, e.g., W) and is opening a new interval. All read operations executed in the new interval should return value V.

An execution example for the strict consistency model is shown in Figure 4.

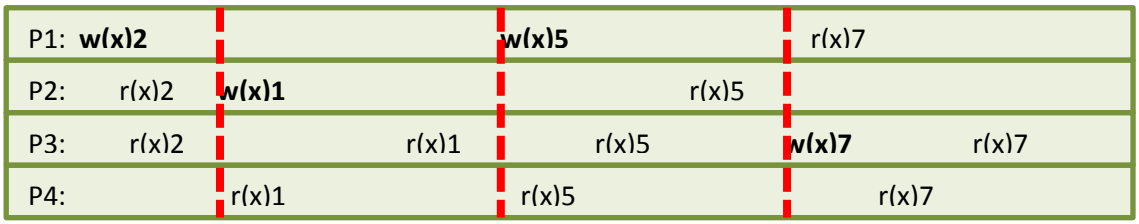

Figure 4. Execution following the strict consistency model.

In this example, the global writing sequence has been P1:w(x)2, P2:w(x)1, P1:w(x)5 and P3:w(x)7. Read values always match the last value written in all the system. This is possible when we assume that update propagation time is zero.

Sequential Consistency

In a sequentially consistent system [15], the result of an execution is identical to one where all operations of all processes were executed in a sequential order and the operations of each process were in that sequence in the order stated by their respective program.

This implies that all processes see all writes in the same order and such order is consistent with what each process has written in its own node. As a difference with the strict model, this does not imply that all processes advance at the same pace. Each process might read the values of such shared sequence sooner or later. Moreover, such agreed shared order does not need to match the real physical writing order.

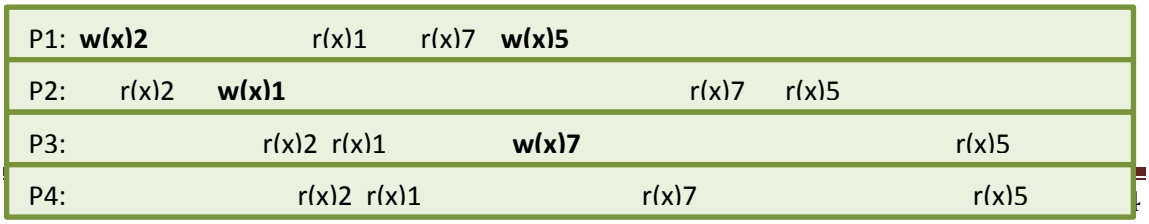

Figure 5. Execution following the sequential consistency model.

For instance, although the writes in Figure 5 have physically happened in the order P1:w(x)2, P2:w(x)1, P1:w(x)5, P3:w(x)7; the agreed order has been P1:w(x)2, P2:w(x)1, P3:w(x)7, P1:w(x)5. Besides, P3 and P4 have still read value 2 when value 1 had already been written. That alteration of the global order (between values 5 and 7) and the read inversion (to read old value 2 when a new value had already been written) would not be admitted in the strict consistency model.

Causal Consistency

Causal consistency assumes that write operations (that need to be propagated to other processes) could be seen as **send()** message actions and that read operations could be seen as **receive()** message actions. Based on this, causal consistency follows the "happens before" relationship defined by Leslie Lamport in [16]. Such relation was already explained in CSD in order to define logical clocks.

Thus, two write events "a" and "b" will be ordered "a<b" when both had happened in the same process in that order or when write "b" has happened in P2 (that was not the writer in "a") once P2 has read the value written in "a". This second case is generated by a sequence similar to: P1:w(x)a, P2:r(x)a, P2:w(x)b. Additionally, this order relation complies with the transitive property.

In the execution shown in Figure 6, the write of value 2 causally precedes the write of value 1 and the latter precedes the writes of values 5 and 7. However, values 5 and 7 are concurrent (i.e., no causally precedence occurs between them). As a result, all system processes have freedom for ordering values 5 and 7 as they prefer. Thus, P1 and P2 see value 5 before value 7, while P3 and P4 see value 7 before value 5. Since the read sequence is not identical in all processes, this concrete execution does not respect the conditions of a sequential consistency model.

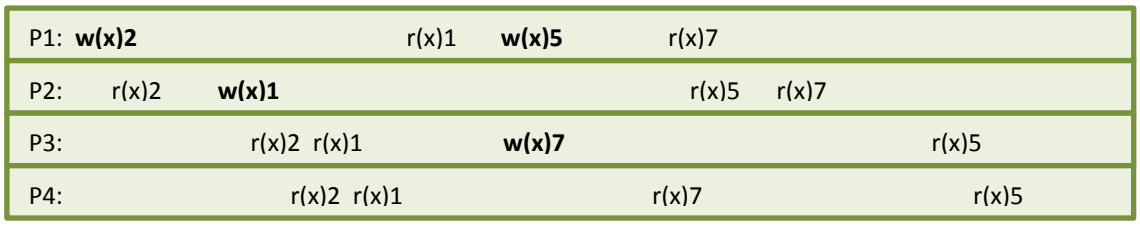

Figure 6. Execution following the causal consistency model.

FIFO Consistency

This consistency (also known as pRAM) requires that the writes made by a given process were read by other processes in their writing order. On the other hand, it does not impose any constraint on how writes executed by different processes could be read.

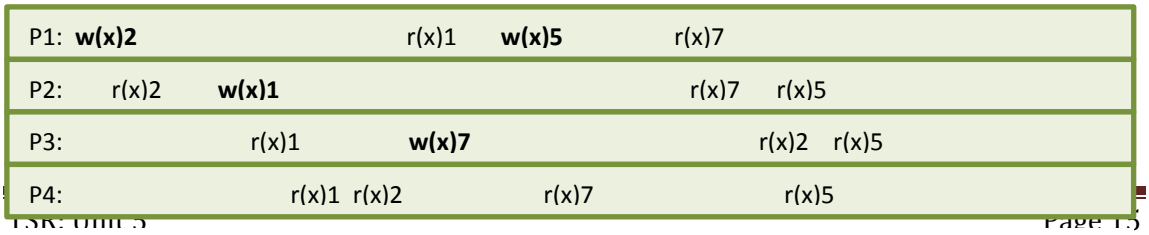

Figure 7. Execution following the FIFO consistency model.

For instance, in the execution shown in Figure 7 there are four writes, but only two of them have been written by the same process (values 2 and 5, written by P1). As a result, a single constraint exists in that execution: value 2 should be read before value 5. Values 1 and 7 could be freely interleaved in the sequence read by each process. Indeed, in this example the sequence observed by each process is different to all the others.

The reader may check that this execution does not comply with any of the previous consistency models (causal, sequential and strict). It is not causal since all readers should get value 2 before reading value 1, and this does not happen in P3 or P4. It is not sequential since not all readers follow an agreed reading sequence. It is not strict because some processes (e.g., P4) have read value 2 when other processes had already read value 1 that was written physically afterwards.

FIFO order also considers the writes applied onto different variables. Figure 8 shows that the write setting value 5 onto variable 'z' by P1 will be always obtained after receiving value 2 on 'x'.

P1:	**w(x)2**							r(x)1		**w(z)5**				r(x)7									
P2:		r(x)2		**w(x)1**												r(x)7		r(z)5					
P3:					r(x)1				**w(x)7**										r(x)2		r(z)5		
P4:						r(x)1	r(x)2					r(x)7					r(z)5						

Figure 8. Another example using FIFO consistency.

Cache Consistency

Cache consistency requires that all writes applied on the same variable were obtained in the same order by all reading processes. Additionally, when multiple writes onto the same variable were made by the same process, such process order will be globally respected. There is no constraint when writes on different variables are considered.

For instance, let us assume a system where processes P1 and P2 modify variables 'x' and 'z'. The program needs that this constraint is respected 'z>=x+2'. Cache consistency will not be enough for ensuring such constraint in the distributed system. Figure 9 shows that although P1 and P2 had executed instructions that respected the constraint when they were locally run, they later receive updates from other processes that violate that constraint. In the end, all processes end their executions with values that do not comply with the constraint required by that program.

P1:		r(x)1	**w(x)2**					**w(z)5**	r(z)3														
P2:							r(z)5	**w(x)1**				**w(z)3**	r(x)2										
P3:							r(z)5				r(z)3							r(x)1	r(x)2				
P4:										r(z)5			r(x)1	r(x)2	r(z)3								

Figure 9. Execution following the cache consistency model.

Indeed, they finish this execution fragment with value 2 in ‘x’ and value 3 in ‘z’.

In order to ensure that program constraint we would need sequential consistency, as shown in Figure 10. Using sequential consistency the constraint is respected, since all processes agree on the global order for the read operations. Besides, such global order should comply with the local writes of P1 and P2 (onto every variable; i.e., FIFO consistency) and with the dependences between P1 and P2 (causal consistency).

P1:	**w(x)2** **w(z)5** r(x)1 r(z)3
P2:	r(x)2 r(z)5 **w(x)1** **w(z)3**
P3:	r(x)2 r(z)5 r(x)1 r(z)3
P4:	r(x)2 r(z)5 r(x)1 r(z)3

Figure 10. Sequentially-consistent execution of the example shown in Figure 9.

Processor Consistency[2]

This model requires that processes comply simultaneously with the conditions of the FIFO and cache models. As a result:

- An agreement is needed on the write order onto each variable (as required by the cache model).
- The writing order of each processor on all variables should be respected (as required by the FIFO model).

P1:	**w(x)2** **w(z)5** r(x)1 r(z)3
P2:	r(x)2 r(z)5 **w(x)1** **w(z)3**
P3:	r(x)2 r(x)1 r(z)5 r(z)3
P4:	r(x)2 r(z)5 r(x)1 r(z)3

Figure 11. Execution in the processor model of the example shown in Figure 9.

Considering the example shown in Figure 9 and executing it using processor consistency as shown in Figure 11, we may see that processor consistency avoids the inconsistencies generated by all processes in Figure 9. Indeed, with processor consistency we could reach the executions of P3 and P4 shown in Figure 10, that were correct regarding the constraint stated in that example.

However, processor consistency is more relaxed than sequential consistency. Note that P3 and P4 do not follow the same read sequences in Figure 11.

Hierarchy of Models

In some cases, the consistency models already presented in this document can be ordered depending on their degree of exigency. The strict model is the strongest one, while the FIFO

[2] Processor consistency will not be considered in the exams of the current academical year. It is included in this guide only for completeness.

and cache models can only ensure weak consistencies. The result is a partial order that is shown in Figure 12.

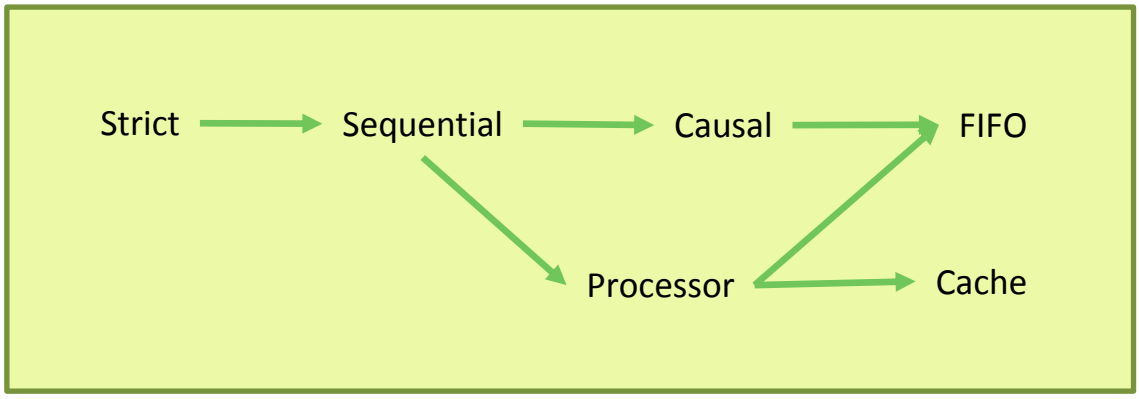

Figure 12. Hierarchy of consistency models.

This means that when an execution is complying with the strict model, it is also sequential. Being sequential, it is also causal and processor. Being causal, it is also FIFO. Being processor, it is also FIFO and cache.

On the other hand, there are some pairs of models that are incomparable. Thus, the causal and processor models cannot be compared. This means that causal is not stronger than processor, or processor is not stronger than causal. There are executions that comply with one of these models, but not with the other. Something similar happens between causal and cache and between cache and FIFO.

That comparability relationship among models may become clear if we represent graphically each model as the set of executions that comply with its requirements. Such a representation is shown in Figure 13. As we can see, the arrows used in Figure 12 for expressing that a model A is stronger than another model B (A --> B) are now represented as A being a subset of B. As an example, all strict executions (i.e., those that respect the strict model and are contained in its set) are also contained in all the other sets, since the strict model is the strongest one. On the other hand, part of the sequential executions are not strict, but all sequential executions are also causal, FIFO and cache; i.e., sequential executions define a proper subset of the causal, FIFO and cache sets. Those inclusion relationships also arise between causal and FIFO, between processor and cache, and between processor and FIFO.

The processor consistency model has not been identified in that figure. In spite of this, it is defined by the intersection between the cache and FIFO sets. As a result, that processor set is also a superset of the sequential one.

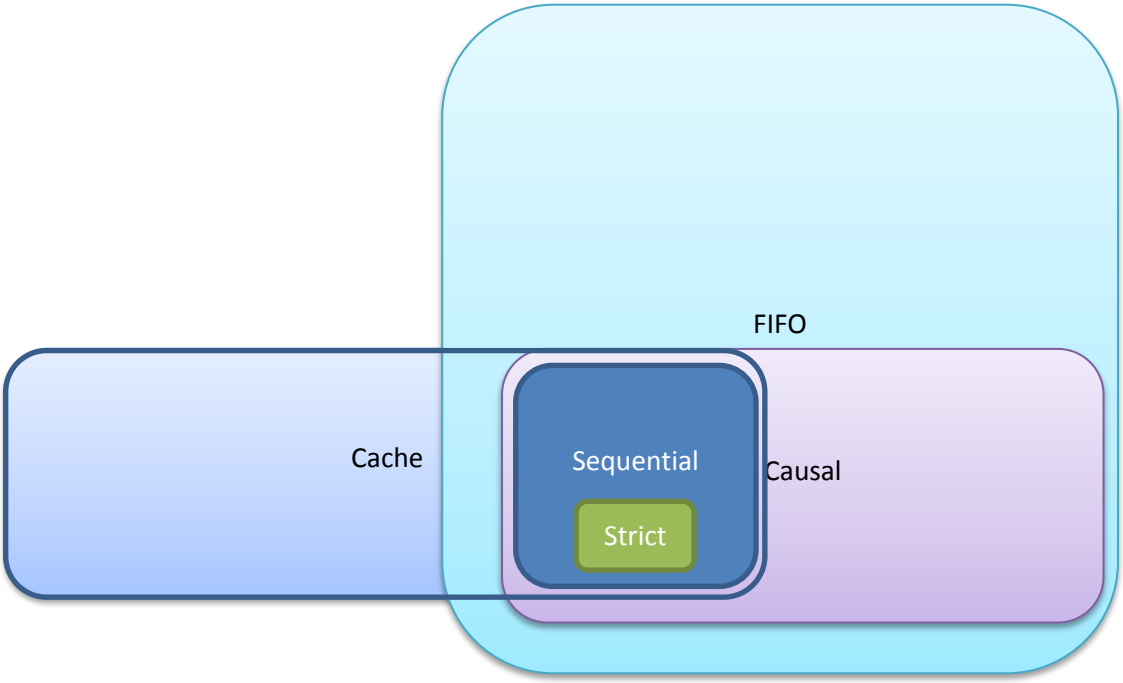

Figure 13. Another view of the hierarchy of consistency models

Let us identify now some examples of executions in concrete parts of the diagram shown in Figure 13. They may help us to understand the differences among consistency models. To begin with, the execution E1 shown in Figure 14 belongs to the cache subset that is not part of the FIFO set. It is a cache execution, since there is only one value written in each variable. Then, it trivially respects cache consistency (since all processes have observed the same sequence on each variable: a sequence that only consists of a single value). However, it does not respect FIFO consistency since P1 wrote value 2 on 'x' before writing value 5 on 'z' but P2 has received those two values in the opposite order. If E1 is not FIFO, it cannot either be processor, causal, sequential or strict.

P1: **w(x)2**		**w(z)5**	
P2:	r(z)5	r(x)2	

Figure 14. Cache execution that is not FIFO.

Figure 15 shows another execution E2 that is FIFO but not causal nor cache. In this case, P1 had written an initial value (2) on 'x' that was propagated to P2 and read before P2 has written value 1 on the same variable. However, both values are known in the opposite order in P3. That fact breaks causal consistency, since value 1 was caused by value 2 (i.e., there is a communication path that is started by w(x)2 and ends with w(x)1). FIFO consistency is still (trivially) preserved since each writer has only generated one value and the write order of each of those processes is respected by the sequence of reads observed in the remaining processes.

P1: **w(x)2**			r(x)1	
P2:	r(x)2	**w(x)1**		
P3:		r(x)1		r(x)2

Figure 15. FIFO execution that is not causal nor cache.

As we can see, since E2 is not causal it cannot either be sequential nor strict. It is not sequential because P3 has not observed the same sequence of values than P1 and P2 observe. P3 sees value 1 before value 2, while P1 and P2 see value 2 before value 1. E2 is not strict because P3 has seen value 2 once the interval for value 1 had already been started.

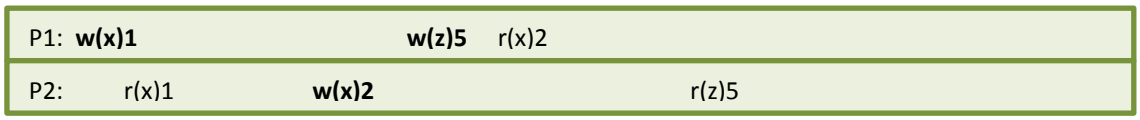

Figure 16. Causal execution that is also cache but not sequential.

Figure 16 shows an execution E3 that is causal but not sequential. It is causal because the two conditions of the causal model are respected. P1 writes two values (1 on 'x' and 5 on 'z', in that order) and those values are observed in the same order in the remaining processes (i.e., in P2 in this case). Besides, there is a communication path between the value 1 written by P1 and the value 2 written by P2. Value 1 precedes causally value 2. That dependency is respected in both processes, since P1 has also seen value 2 after writing value 1. Values 2 and 5 have no causal dependency. They are concurrent. Because of this, it does not matter (for the causal consistency model) in which order those writes are read by the processes. Indeed, P1 has seen value 5 before value 2 and P2 has seen those values in the opposite order. This breaks sequential consistency, since the latter requires that all processes see the same sequence of values. Since E3 is not sequential, it cannot be either strict.

Execution E3 is also cache. There have been two writes on variable 'x' and they have been seen in the same order in all the processes (1 before 2). There is a single write on variable 'z'. So, E3 respects the cache consistency model. Since E3 is causal and all causal executions are also FIFO, this also means that E3 respects processor consistency.

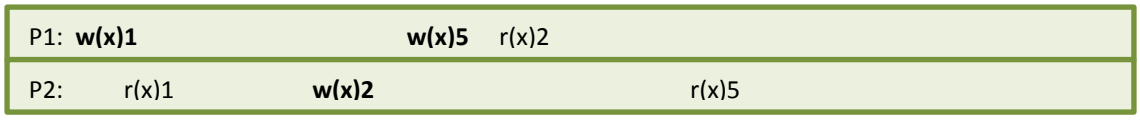

Figure 17. Causal execution that is not cache nor sequential.

Finally, execution E4 (shown in Figure 17) is a variation of execution E3 that still respects the causal (and FIFO) models, but not cache or sequential consistencies. The third write operation in E3 (on variable 'z') has been replaced by a write of the same value on 'x'. This breaks cache consistency, since P1 has seen the sequence 1, 5 and 2 of values for that variable, while P2 sees a different sequence (1, 2, and 5).

These four executions (E1, E2, E3 and E4) have presented concrete examples of the relationships between all consistency models described in this document. They also provide a guide for discussing which consistency models are respected by a given execution. Figure 18 depicts their location in the collection of sets initially shown in Figure 13.

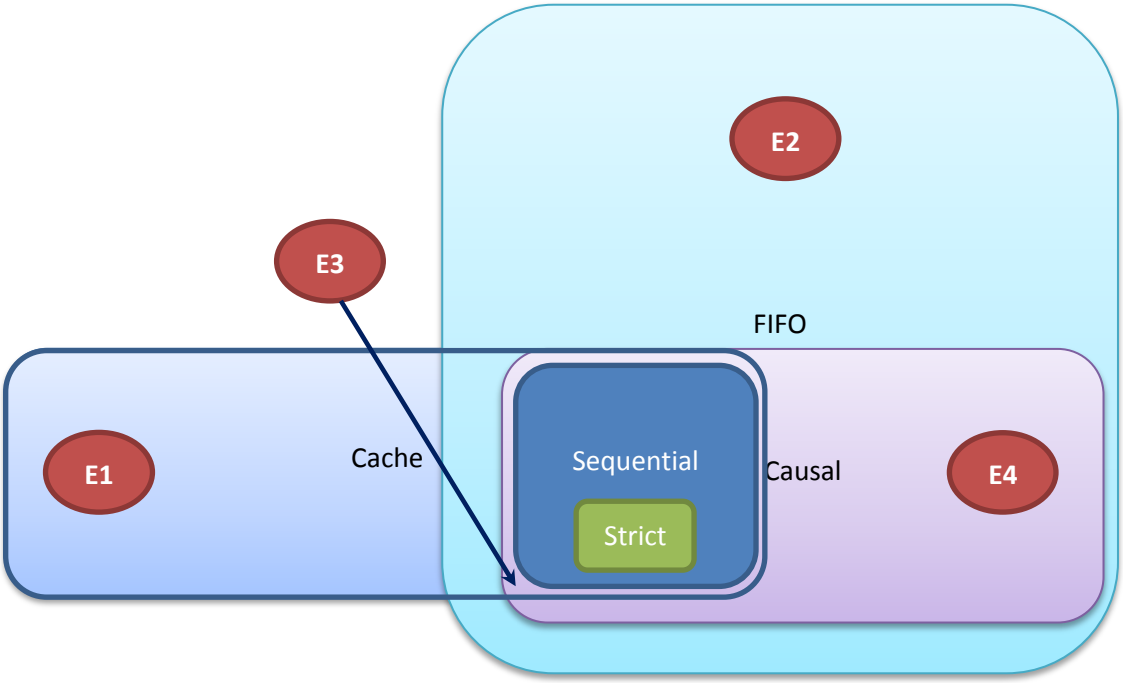

Figure 18. Graphical representation of executions E1, E2, E3 and E4 in the hierarchy of models.

Fast Models

As it has been said in previous sections, consistency models are needed for identifying the degree of divergence among replicas of a given service or data element. Those replicas use a replication protocol that manages update propagations among them. That replication protocol implements one of the classical replication models or, alternatively, any intermediate variant.

When a replication protocol allows that both read and write operations on a local data element return control to the invoker before any message is exchanged with the other replicas, then such protocol implements a "fast" consistency model [17]. Fast models forward those updates in a deferred way and they never block read operations. A read operation blocks its invocations when the replication protocol is compelled to suspend that reader process until some pending write propagation messages are received from any remote replicas.

The strict, sequential, processor and cache[3] models are not fast since they demand agreement on the order of update receptions among all replicas. Such agreement demands message exchange among the participating processes for managing write operations. While a write agreement is being managed, read attempts on the variable being written are also blocked. The FIFO and causal models are fast since there are protocols for those models that admit a fast implementation of their read and write operations.

[3] There is an algorithm [20] that implements cache consistency with fast operations that do not block the process (in both reads and writes), but it does not tolerate process failures or network partitions. Besides, it propagates its broadcasts in a synchronous way, avoiding new broadcasts until the current one is delivered. However, in a general sense, in systems where process failures or network partitions may arise, cache consistency should not be considered a fast model.

3.2. Eventual Consistency

Eventual consistency was defined in [13] as a condition that is respected when there is a time interval without writes. Such condition can be stated as follows: “component replicas will reach the same value when there is a sufficiently long interval without new writes”.

This means that in an eventually consistent system there is freedom for writing in any replica, propagating the updates with the order and delay that each writer considers appropriate. When the entire set of writes is received, each process will know how to apply those writes for obtaining the same final value. A possible way for achieving this consists in converting assignment operations into commutative operations.

For instance, let us assume that the initial value for a variable ‘x’ was 530. Different processes have applied some operations that would generate the following sequence of values assuming a strict consistency model: 520, 600, 650, 640, and 700. Instead of applying those assignments, each process should read the local value of ‘x’ at the time of writing, converting those (assignment) operations into the following sequence: x:=x-10, x:=x+80, x:=x+50, x:=x-10, x:=x+60. Once converted, each writer propagates its operation to all other processes. In that case, it is irrelevant the order in which those operations are received by all the system processes. In the end, those processes will reach the same value in variable ‘x’ (assuming that there is an interval of inactivity afterwards). In this example, all processes obtain 700.

However, if each replica received the operations in a different order, each one may have seen a different sequence of intermediate values. This implies that in that interval of propagation and application of operations, replica consistency may have been very relaxed.

Let us analyse this execution in detail, as follows:

- Initial value of variable ‘x’: 530.
- Operations to be run:
  - op1: x:=x-10;
  - op2: x:=x+80;
  - op3: x:=x+50;
  - op4: x:=x-10;
  - op5: x:=x+60;
- Existing replicas: P1, P2, P3, P4, P5.
- Possible execution orders:
  - P1: op1, op2, op3, op4, op5.
  - P2: op2, op4, op3, op5, op1.
  - P3: op3, op5, op2, op4, op1.
  - P4: op4, op1, op3, op2, op5.
  - P5: op5, op4, op3, op2, op1.
- Sequence of values being observed in each replica:
  - P1: 530, 520, 600, 650, 640, 700.
  - P2: 530, 610, 600, 650, 710, 700.
  - P3: 530, 580, 640, 720, 710, 700.
  - P4: 530, 520, 510, 560, 640, 700.

- P5: 530, 590, 580, 630, 710, 700.

Additionally, these partial values might be obtained at different times. Processes do not advance synchronously. Despite this, once each operation sequence is terminated, all processes obtain the same final value: 700. As a result, they are consistent again.

Eventual consistency is one of the keys for improving performance and scalability in distributed systems [18].

References

[1] V. P. Nelson, "Fault-Tolerant Computing: Fundamental Concepts," *IEEE Computer,* vol. 23, no. 7, pp. 19-25, 1990.

[2] F. Cristian, "Understanding Fault-Tolerant Distributed Systems," *Commun. ACM,* vol. 34, no. 2, pp. 57-78, 1991.

[3] V. Hadzilacos and S. Toueg, "Fault-Tolerant Broadcasts and Related problems," in *Distributed Systems*, Addison-Wesley, 1993, pp. 97-145.

[4] F. B. Schneider, "What Good Are Models and What Models Are Good?," in *Distributed Systems*, Addison-Wesley, 1993, pp. 17-26.

[5] G. Chockler, I. Keidar and R. Vitenberg, "Group Communication Specifications: A Comprehensive Study," *ACM Comput. Surv.,* vol. 33, no. 4, pp. 427-469, 2001.

[6] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso and B. Kemme, "Are Quorums an Alternative for Data Replication?," *ACM Trans. Database Syst.,* vol. 28, no. 3, pp. 257-294, 2003.

[7] N. Budhiraja, K. Marzullo, F. B. Schneider and S. Toueg, "Optimal Primary-Backup Protocols," in *6th International Workshop on Distributed Algorithms and Graphs (WDAG)*, Haifa, Israel, Springer, 1992, pp. 362-378.

[8] F. B. Schneider, "Implementing Faul-Tolerant Services using the State-Machine Approach: A Tutorial," *ACM Comput. Surv.,* vol. 22, no. 4, pp. 299-319, 1990.

[9] M. J. Fischer, N. A. Lynch and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM,* vol. 32, no. 2, pp. 374-382, 1985.

[10] D. Mosberger, "Memory Consistency Models," *Operating Systems Review,* vol. 27, no. 1, pp. 18-26, 1993.

[11] R. C. Steinke and G. J. Nutt, "A Unified Theory of Shared Memory Consistency," *J. ACM,* vol. 51, no. 5, pp. 800-849, 2004.

[12] V. Cholvi and J. M. Bernabéu-Aubán, “Relationships between Memory Models,” *Inf. Process. Lett.,* vol. 90, no. 2, pp. 53-58, 2004.

[13] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer and B. B. Welch, “Session Guarantees for Weakly Consistent Replicated Data,” in *3rd International Conference on Parallel and Distributed Information Systems (PDIS)*, Austin, Texas, EE.UU., IEEE-CS Press, 1994, pp. 140-149.

[14] L. Lamport, “On Interprocess Communication,” *Distributed Computing,* vol. 1, no. 2, pp. 77-101, 1986.

[15] L. Lamport, “How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs,” *IEEE Trans. Computers,* vol. 28, no. 9, pp. 690-691, 1979.

[16] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” *Commun. ACM,* vol. 21, no. 7, pp. 558-565, 1978.

[17] H. Attiya and R. Friedman, “Limitations of Fast Consistency Conditions for Distributed Shared Memories,” *Inf. Process. Lett.,* vol. 57, no. 5, pp. 243-248, 1996.

[18] W. Vogels, “Eventually Consistent,” *Commun. ACM,* vol. 52, no. 1, pp. 40-44, 2009.

[19] H. Kopetz and P. Veríssimo, “Real Time and Dependability Concepts,” in *Distributed Systems*, 2ª ed., S. J. Mullender, Ed., Addison-Wesley, 1993, pp. 411-446.

[20] E. Jiménez, A. Fernández and V. Cholvi, “A parametrized algorithm that implements sequential, causal and cache memory consistencies,” *Journal of Systems and Software,* vol. 81, pp. 120-131, 2008.