Lecture notes: Replication for fault tolerance in the Cloud

M2 MOSIG: Cloud Computing, from infrastructure to applications

Thomas Ropars

2023

This lecture is about techniques based on replication to avoid service failures in the Cloud. More specifically, it presents techniques that can be applied to deal with benign faults[1].

1 Introduction

1.1 Fault model

To be able to reason about fault tolerance techniques, we need to define an abstract model that captures the main characteristics of the faults we want be able to tackle.

In the following, we will assume **benign** faults. For processes, benign faults correspond to crashes: a process either executes according to the specification, or stops executing. For channels, benign faults correspond to the loss of messages.

Another model that could have been considered is **Random faults (Byzantine)**. In this model, a system provides *random* outputs to the same inputs. It covers the cases where different users might observe different behaviors and the cases of malicious behaviors. Due to the complexity of the solutions that deal with byzantine faults, we do not consider them hereafter.

1.2 Introduction to replication

The basic way of interacting between services/processes in Clouds and data centers is through a client/server model. A client sends a request to another node and waits for an answer.

Replication is a technique that allows us to increase the availability of a system or service. With replication, instead of having only one instance of a system/service, there are several copies. So, if one copy crashes, the system/service will still be available, thanks to the other copies (Fig. 1).

Several approaches to replication exists. We can distinguish 3 main categories:

Data replication : With this approach, the client can only issue read and write operations on the data. On the other hand, it allows a high degree of parallelism.

Passive replication : This approach is also called *active/standby* replication. A single leader executes all requests and sends updates to the other replicas.

[1] Acknowledgments: Parts of these notes are strongly inspired by the lectures notes of Andre Schiper on *Distributed Algorithms*.

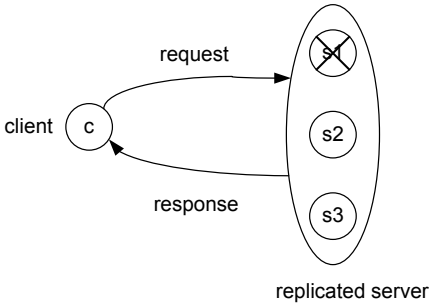

Figure 1: Replication (client-server model)

Active replication : This approach is also called *active/active* replication, or multi-leader replication. In this approach, all replicas can process requests.

Note that availability might not be the only concern when using replication. Replication may also be used to improve performance:

- To reduce the latency by keeping data geographically close to the users
- To improve throughput by allowing parallel reads

These concerns should also be taken into account when choosing a replication technique.

2 Data replication (Quorum systems)

In a first step, we consider a simple problem where data need to be replicated. Only two operations can be applied on a data item, *read* and *write*:

- `write` overwrites the previous value of the data item,
- `read` returns the most recent value written.

Defining the most recent value written might not be that trivial when the data is replicated on multiple servers. Figure 2 illustrates a basic scenario. We need a consistency criteria to define what is an acceptable result for a read operation.

2.1 Linearizability

The strongest consistency criteria for defining what is an acceptable result for a read request is called *linearizability*.

Definition Let x be a data item, and denote the *read* operation of x by $read(x)$, and the *write* operation by $write(x, val)$, where val is the value written. Consider an execution σ consisting of the concurrent execution of read and write operations by a set of processes. The execution σ is *linearizable* if there exists a *sequential* execution τ in which:

(1) All read operations in τ return the same value as in σ

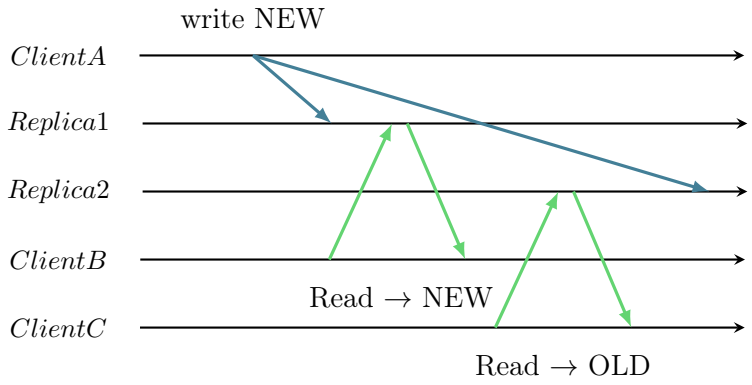

Figure 2: A problematic scenario

and

(2a) The start and the end of each read or write operation *op* in τ occur at the *same* time, denoted by t^{op}, and

(2b) The time t^{op} is in the interval $[t_s^{op}, t_e^{op}]$, where t_s^{op} is the time at which the operation *op* started in σ and t_e^{op} is the time at which the operation *op* ended in σ.

Example 1: Figure 3 shows an execution σ that is linearizable:

- Process p executes a $write(x, 0)$ operation that starts at time t_1 and ends at time t_3.
- Process q executes a $write(x, 1)$ operation that starts at time t_2 and ends at time t_5.
- Process p executes a $read(x)$ operation that starts at time t_4 and ends at time t_7, returning the value 0.
- Process q executes a $read(x)$ operation that starts at time t_6 and ends at time t_8, returning the value 1.

The bottom time-line in Figure 3 shows a sequential execution τ that satisfies the three conditions *(1)*, *(2a)* and *(2b)* above (t_a is in $[t_1, t_3]$, t_b is in $[t_4, t_7]$, etc.). Thus σ is linearizable.

Note that to define *linearizability*, we consider the operations for the client point of view. Figure 3 does not describe how clients p and q interacts with the (replicated) server.

Example 2: Figure 4 shows an execution σ that is not linearizable. The execution is not linearizable, since in any sequential execution τ, the $write(x, 1)$ operation of q must precede the $read(x)$ operation of p. Thus in any sequential execution τ, the $read(x)$ operation of p returns 1 (and not 0 as in σ).

Informal definition of linearizability Informally, we can say that linearizability aims at making the system appear as if there was a single copy of the data. To achieve this goal, we should enforce the following behavior:

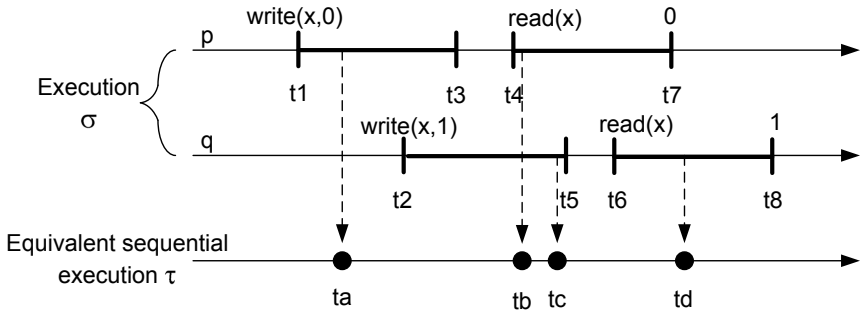

Figure 3: A linearizable execution

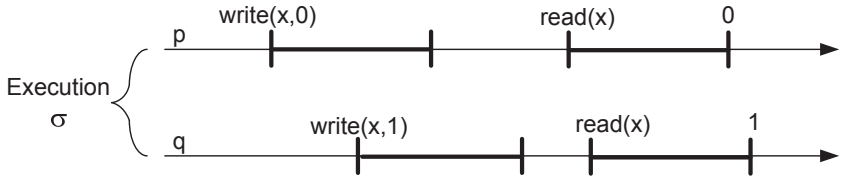

Figure 4: A non linearizable execution

- If a read request terminates before the beginning of a write request, it must return the old value.
- If a read request starts after the end of a write request, it must return the new value.
- If a read request is concurrent with a write request, it may return either the old or the new value.

In addition to these three basic rules, we should introduce the following one:

- After a read request that returned the new value, all subsequent read requests, from the same or from a different client, must return the new value.

This last rule describes the fact that in a linearizable system, a write operation should appear to occur atomically at some point between the invocation of the write method and its termination.

The definition of linearizability assumes the existence of a global clock to be able to define if a request occurred before/after another one, based on the physical time. Building such a clock may be difficult in practice but it is not necessary to have one to build a linearizable system.

Linearizability in practice Linearizability is a very desirable property because it simplifies the work of the application developers. Indeed, linearizability is a composable property. It means that if an application uses two linearizable services, the resulting execution is still linearizable. It is not the case for weaker consistency models (such as sequential consistency).

Unfortunately, insuring linearizability can be costly from performance point of view. Hence, many services make the choice to implement a weaker consistency model. We will resume this discussion later.

2.2 Non-replicated implementation of a linearizable data server

A non replicated server is easy to implement. The server consists of one process that manages the data. Clients send requests and wait to receive a reply from the server (this is called *synchronous invocation*). Requests are received and handled by the server process sequentially. A $write(x, val)$ request by client p leads the server to update x to val, and to send ok to p. A $read(x)$ request by client p leads the server to send val to p. This implementation trivially ensures linearizability.

2.3 Replicated implementation of a data server

In this section, we are going to present a replication technique that is sometimes called *leaderless replication*. In such an approach, a client directly contacts one/several replicas to execute read or write operations. This name is given by opposition to *leader-based* approaches (that we will study later), where the client sends its requests to a single node that is in charge of synchronizing with the other replicas.

2.3.1 Quorum systems

To build a linearizable replicated data server, we are going to rely on the concept of *quorum system*. Consider a set S of n servers $S_1, \dots S_n$, i.e., $S = \{S_1, \dots, S_n\}$:

- A *quorum system* of S is a set of "subsets of S", such that any two "subsets of S" have a non empty intersection.

For example, if $S = \{S_1, S_2, S_3\}$, then

$$Q = \{ \{S_1\}, \{S_1, S_2, S_3\} \}$$

is a quorum system of S. The following set Q' is also a quorum system of S:

$$Q' = \{ \{S_1, S_2\}, \{S_1, S_3\}, \{S_2, S_3\} \}$$

Each element of Q, and each element of Q', is called a *quorum*. Note that each quorum of Q' contains a majority of servers.

With quorum systems, the basic idea can be expressed as follows. Consider a quorum system of S:

- Each write operation must update a quorum of servers.
- Each read operation must access a quorum of servers.

As such, it is guaranteed that each read request will retrieve the latest value that has been written.

Another way of presenting the idea of quorum systems is to say that if there are n replicas, each write must update w replicas and each read must access r replicas, with $w + r > n$. In a system with 3 replicas ($n = 3$), it is very common to set $w = r = 2$, which corresponds to the quorum

system Q'. However a system may be configured differently depending on the needs. For instance, a system could be configured with $w = 3$ and $r = 1$ to have very fast reads at the cost of slower writes. Another limitation of this configuration is that if one server fails, then all write operations will fail.

2.3.2 Fault tolerance with a quorum system

To ensure fault tolerance, we should rely on a majority rule for quorums for both read and write operations. A majority rule states that $r > n/2$ and $w > n/2$. With such a rule, one is able to tolerate up to f server crashes, with $f < n/2$.

Here are a few additional comments about quorums and fault tolerance:

- If $w < n$, we can still write if a node crashes
- If $r < n$, we can still read if a node crashes
- A configuration with $n = 3$, $w = r = 2$, can tolerate on crash
- A configuration with $n = 5$, $w = r = 3$, can tolerate two crashes

It should also be mentioned that even if w and r are configured to be less than n, read and write requests are usually sent to all replicas by the client. The value of w and r defines the number of answers to wait for before considering an operation as terminated.

Based on the majority rule, if we consider the 2 quorum systems Q and Q' defined earlier, we can conclude that Q' is an appropriate quorum system for fault tolerance. Q is not.

2.3.3 Linearizable data replication based on quorum systems

We present a solution to implement linearizability for data items in a system with at most f faults ($f < n/2$).

Server code: The code executed by one replica S_i is presented in Figure 5. As it can be noticed, in addition to the value of the data item, two additional information are maintained by the server: a version number and the identifier of the client that most recently updated the item.

```
1     value = 0 # data item value
2     version = 0 # version number
3     clientId = 0 # id of the most recent client that has written the item

5     upon readReplica() by client c:
6         send (value,version,clientId) to c

8     upon writeReplica(val, v, id): #v is a version number; id is the client id
9         if (v > version) or ((v==version) and (id > clientId)):
10            value = val
11            version = v
12            clientId = id
```

Figure 5: Read and write operation: code of a replica S_i

The version number should be incremented every time a client updates the data item. The information about the `clientId` is needed to deal with concurrent writes. As shown in line 9, the `writeReplica` operation overwrites the current value only if: i) the version number of the operation is higher than the current version number or, ii) the version numbers are equal and the identifier of the client is larger than the one of the last client that updated the item.

Client code (first try): Figure 6 presents a first version of the client algorithm. The `write` operation first requires to read from a quorum (line 14). Only the `version` number is a relevant information in this case: the read operation is needed to obtain a valid version number. The client issues the write to a quorum with a version number incremented by one (line 16).

The `read` operation issues a read from a quorum and returns the value associated with the highest {`version`, `clientId`} observed.

```
13    def write(val, id):                  # id is the identifier of the client
14        readReplica(--, version, --) from a quorum
15        v = highest version number read
16        writeReplica(val, v+1, id) to a quorum        # synchronous invocation

18    def read():
19        readReplica(val, version , id) from a quorum
20        let (v, id) be the highest (version number, client Id) read
21        let value be the value with version/clientId (v, id)
22        return value
```

Figure 6: Code of a client C (Incorrect version)

The simple algorithm presented in Figure 6 is unfortunately not correct.

It does not ensure a linearizable execution as demonstrated by the scenario presented in Figure 7. In this scenario, linearizability would require the read operation of `Reader2` to return the new value (or `Reader1` to return the old value).

Client code (final version): To solve this problem, we should implement a technique called *read repair*. During a read operation, if a client observes stale values, it should synchronously issue a write to a quorum with the most recent value observed before terminating the read operation.

The new version of the client code is provided in Figure 8. The only modification compared to the algorithm presented in Figure 6 is the call to `writeReplica()` during the read operation (line 32).

The result is that, instead of the execution depicted in Figure 9 that corresponds to the scenario presented in Figure 7 and which is obviously a non-linearizable execution, we can get the execution depicted in Figure 10, which is linearizable.

2.3.4 Replication based on quorum systems in practice

Replication based on quorum systems is used by several NoSQL databases. Dynamo, the scalable KV-store used internally at Amazon, is one of the systems that made this approach popular. Apache

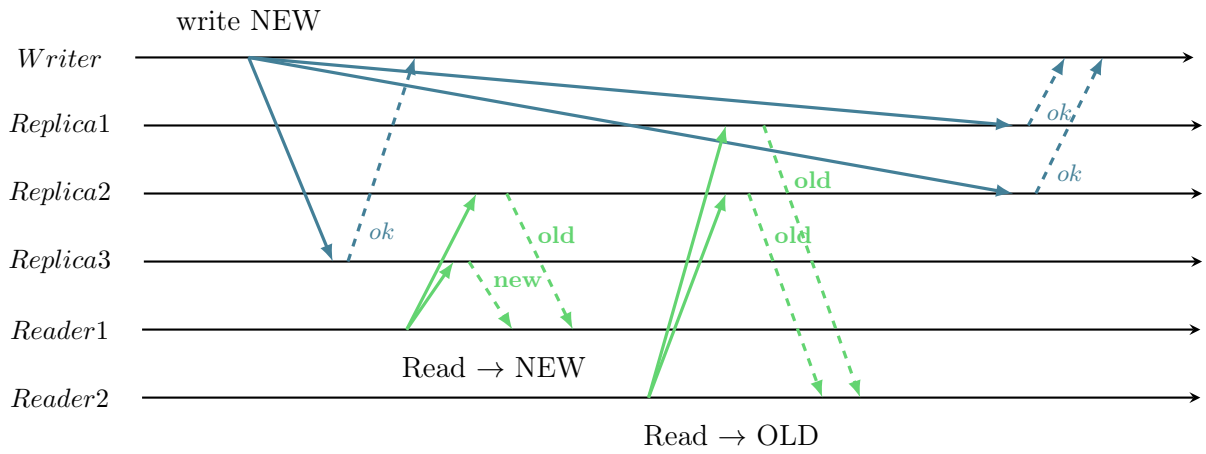

Figure 7: A non-linearizable execution with quorums

```
23      def write(val, id):                  # id is the identifier of the client
24          readReplica(--, version, --) from a quorum
25          v = highest version number read
26          writeReplica(val, v+1, id) to a quorum       # synchronous invocation

28      def read():
29          readReplica(val, version , id) from a quorum
30          let (v, id) be the highest (version number, client Id) read
31          let value be the value with version/clientId (v, id)
32          writeReplica(value, v, id) to a quorum        # synchronous invocation
33          return value
```

Figure 8: Code of a client C (Correct version)

Cassandra[2] is a famous open-source database that also uses this approach. When applying quorum-based approaches in practice, additional problems arise. We discuss some of them below.

Read repair and anti-entropy The *read repair* technique introduced in Algorithm 8 is not only used to ensure linearizability. It is also used to update a replica that has a stale version of the data because it experienced a failure. However, this mechanism slows down read operations. Furthermore, if a data is not accessed very often, it may take time before a stale replica is updated. As long as the replica is not updated, fault tolerance is reduced as the number of valid copies of the replica is decreased.

To deal with both issues, some databases introduce an *anti-entropy* mechanism. Anti-entropy is a background task that checks the health of data, and applies updates to stale replicas if need be.

Dealing with concurrent writes If two clients decide to write the same data item *at the same time*, a quorum-based algorithm needs to decide what should be the final value to store. Imagine that 2 clients, `A` and `B`, send a write request concurrently. Because of network transmission delays

[2] `http://cassandra.apache.org/`

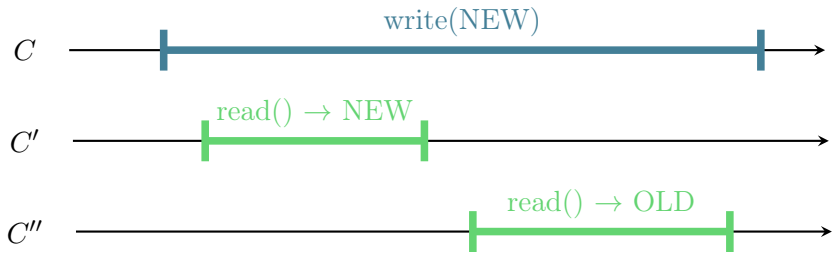

Figure 9: A possible execution with algorithm 6

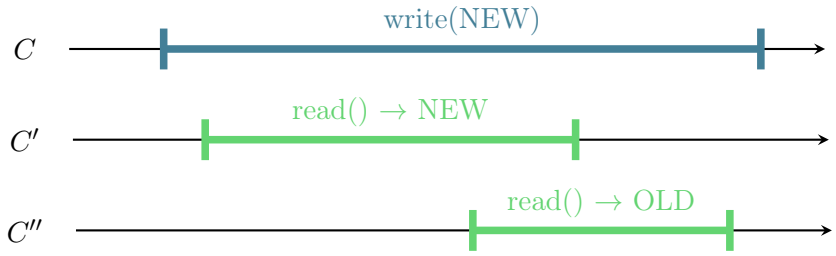

Figure 10: A possible execution with algorithm 8

(among other things), it may be the case that not all replicas receive the requests in the same order. In a system with 3 replicas, 2 may receive the request of `A` first while the last one will receive the one of `B`. In this situation, we need a solution to ensure that all replicas will select the same value as final value.

In Algorithm 5, the solution relies on the identifier of the clients to take a decision: the write operation issued by the client having the largest identifier wins. It solves the problem but we can ask ourselves whether it makes sense that client `B` has the priority over client `A`.

An alternative approach, adopted for instance by Cassandra, is to assign a timestamp to each request, and to decide than the most recent request will win. This approach is called *last-write-wins* (LWW). Such a solution can be better for fairness. As long as the clock of the different clients remain *well* synchronized, such a solution can work. However, it might lead to non-linearizable executions if clock skews are large. Discussing such a scenario in more details is beyond the scope of this lecture.

About multi-datacenter configurations To increase the availability of data and services and/or to reduce the latency of data accesses, some cloud applications are distributed over geographically distant data centers. Although very desirable, linearizability might be difficult to achieve in such a context for two main reasons.

The first reason is the limited availability that can be achieved. Imagine a scenario where replicas of a data item are distributed over two data centers DC_a and DC_b. Due to a network failure, communication between the two data centers is temporarily not possible: clients trying to access the data are only able to contact the replicas in one data center (can be DC_a or DC_b depending on the client). The majority rule that we apply to ensure linearizability despite failures implies that only a subset of the clients will be able to continue accessing the data. Indeed, either

the majority of replicas is in DC_a or in DC_b. If we assume that it is in DC_a, it means that the clients only having access to DC_b cannot read or write the data anymore, and so, the availability is reduced.

To deal with this issue, a technique called *sloppy quorums* may be used. The idea is to create more replicas inside a data center when w replicas are not accessible anymore from a client. This avoids that write operations will fail. Once the connection between the datacenters is restored, the additional replicas are destroyed, and the modifications in the different partitions are merged. Even if such a solution can ensure the durability of write operations, it leads to non-linearizable executions as read and write quorums may not overlap anymore.

The second reason is performance. As implied by the previous discussion, ensuring linearizability in a multi-datacenter setup requires that at least some clients access a remote replica, *i.e.*, a replica that is not in the closest datacenter, to follow the majority-quorum rules. In this case, it means that read and write operations may become slow as they involve potentially high latency communication. Once again, the solution in this case is to lower the level of consistency to avoid having to communicate with remote datacenters synchronously.