INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Big Data Storage and Processing

Lab 1: Strong and Eventual Consistency

The goal of this TP is to understand the concept of **consistency** (and its different levels) in the context of Big Data. In particular, we will focus on consistency for data **replication** in distributed systems.

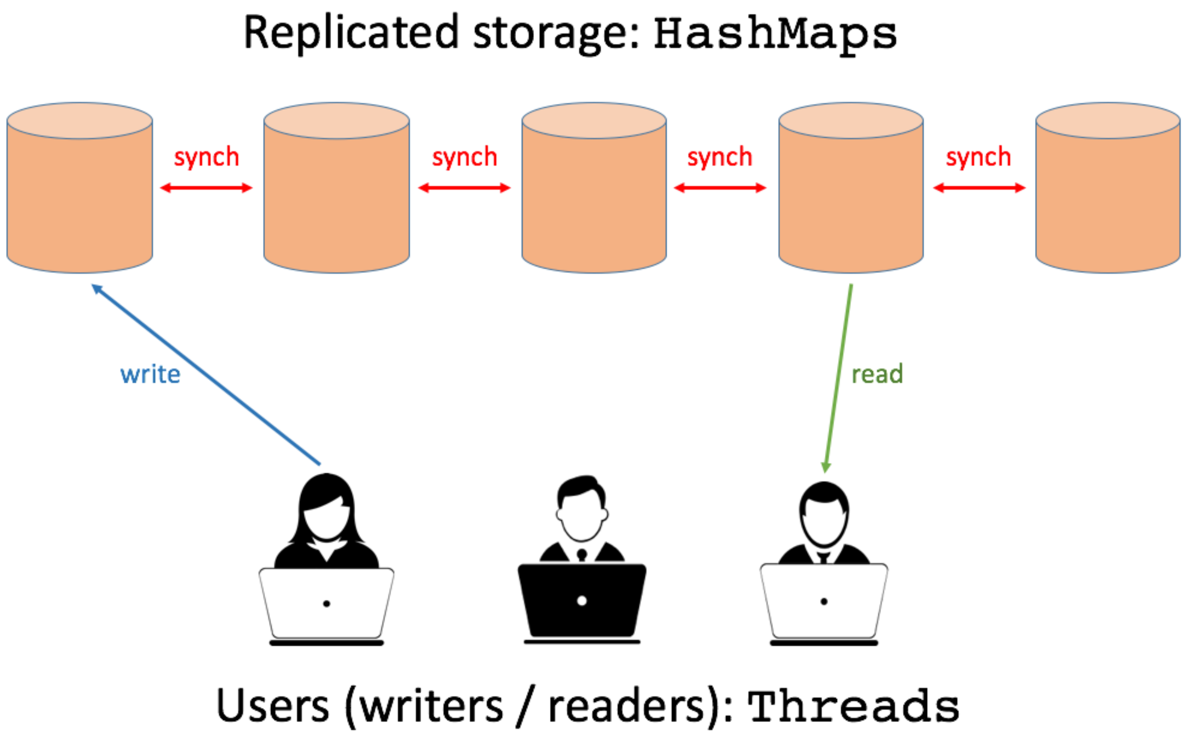
In an ideal world there would only be one consistency model. Consider a storage system made up of multiple replicas and (many) clients accessing it concurrently. Ideally, when an update is made on one replica, all clients will see that update (independent of the replica they query). This is the **strong consistency**.

However, this is very hard to achieve: it comes at the cost of increased latency (needed to synchronize all replicas before acknowledging the update to the client). In many cases (e.g., Facebook updates), it is preferable to have a very low latency of the updates, although maybe not all replicas are immediately synchronized. To enable this, the degrees of consistency can vary, according to the applications needs.

One such relaxed model of consistency is the **eventual consistency**, useful when the “fast access” requirement dominates. Clients update some replica (e.g., the closest or some designated replica) and the updated replica sends update messages to all other replicas. This means that if clients query different replicas during the inconsistency window (the time needed to synchronize all replicas) they will get outdated / different values from different replicas. However, the storage system guarantees that if no new updates are made to the data, **eventually** (after the inconsistency window closes) all accesses will return the last updated value.

The most popular system that implements eventual consistency is DNS, the domain name system. Updates to a name are distributed according to a configured pattern and in combination with time controlled caches; eventually clients will see the update.

We will implement in Java these two levels of consistency. We will use the environment described in the picture bellow:


Replicated storage: HashMaps
synch
synch
synch
synch
write
read
Users (writers / readers): Threads

- **The Replicated storage**. You should assume that it is something big and geographically distributed. We can use a set of `HashMaps` to emulate each replica. Remember, all replicas should (eventually) store the same data.
- **The Users**. We can emulate them using `Threads` that write to and read from (different replicas of) the Replicated storage. At the user side, consistency has to do with *how* and *when* a user sees updates made by another user to the storage system.

Exercise 1. Setting up the environment (done)

The archive *"Squelette du programme"* contains:

- `Replica.java` class that implements the replicas of the storage.
- `User.java` extends JavaThread to describe user behaviour.
- `Main.java` class that launches replicas and users.

Of course, you can implement your own Replica / User / Main classes, if you wish so.

Exercise 2. Basic strong consistency

A first approach to implementing strong consistency relies on transactions.

```
WRITE
start transaction (lock all replicas)
        make the same update to all replicas
end transaction (unlock all replicas: either commit – all changes are
made, are visible and persist or abort – no changes are made to any
replica).

READ
read from any replica
```

Implement user write and read methods according to this protocol. For transaction support you can use the `synchronized` construct available in Java or specific synchronization data structures from the `java.util.concurrent` package (e.g. `Lock`, `Semaphore` etc.). Test the methods with users reading and writing concurrently from different replicas. What do you notice? What would happen if the storage system had much more replicas?

Exercise 3. Optimized strong consistency (optional)

There are several problems with locking **all** replicas to make an update: some replicas may be at the end of slow communication lines; some replicas may fail, or be slow or overloaded. All these translate into high delays in responding to queries. A better option is to try a majority voting scheme – **Quorum Assembly** – and lock only a subset of the replicas, instead of all. Let's define:

N – the number of replicas
W – the write quorum: the number of replicas that need to be locked for write (i.e., acknowledge the receipt of the update before the update completes)
R – the read quorum: the number of replicas that are contacted for a read operation

The conditions that need to be satisfied when choosing W and R are:

W > N/2 (only one write quorum can successfully be assembled at any time)

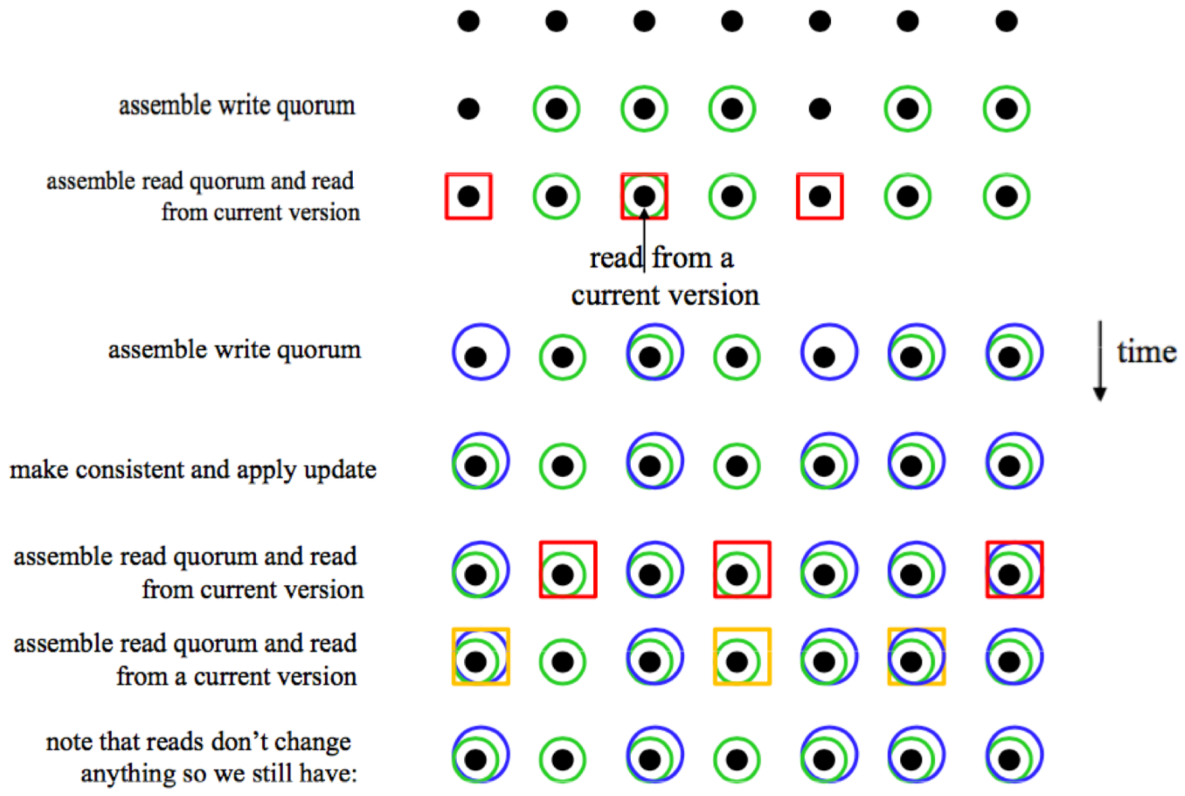
W + R > N (every W and R contain at least one up-to-date replica: the write set and the read set always overlap and one can guarantee strong consistency)

```
WRITE
start transaction (lock W replicas)
        bring all W replicas up-to-date (i.e., synch their data)
        make the same update to all W replicas
end transaction

READ
contact R replicas and read from the most recently updated
```

e.g. W = N, R = 1 is lock all copies for writing, read from any – as before.

Another example: N = 7, W = 5, R = 3. Circles are writes, rectangles are reads. Each color represents a new action (write or read).


assemble write quorum
assemble read quorum and read from current version
read from a current version
assemble write quorum
time
make consistent and apply update
assemble read quorum and read from current version
assemble read quorum and read from a current version
note that reads don't change anything so we still have:

Change the previous read and write methods to implement this protocol. Choose the appropriate W and R values. For the read operation, you will need to know which replica is the most recent updated (for instance, you can store a timestamp of the latest update for each replica, or a total number of updates).

Comment. In distributed storage systems that need to address high-performance and high-availability the number N of replicas is in general higher than 2. Systems that focus solely on fault-tolerance often use N=3 (with W=2 and R=2 configurations). Systems that need to serve very high read loads often replicate their data beyond what is required for fault-tolerance, where N can be tens or even hundreds of nodes and with R configured to 1 such that a single read will return a result. For systems that are concerned about consistency they set W=N for updates, but which may decrease the probability of the write succeeding. How to configure N, W and R depends on what the common case is and which performance path needs to be optimized. In R=1 and N=W we optimize for the read case and in the W=1 and R=N we would optimize for a very fast write.

Summary

Inconsistency can be tolerated for two reasons: for improving read and write performance under highly concurrent conditions and for handling partition cases where a majority model would render part of the system unavailable even though the nodes are up and running.

Whether or not inconsistencies are acceptable depends on the client application. A specific popular case is a website scenario in which we can have the notion of user-perceived consistency; the inconsistency window needs to be smaller than the time expected for the customer to return for the next page load. This allows for updates to propagate through the system, before the next read is expected.