Replication for Fault Tolerance

Quorum Consensus Protocols

- Clients communicate directly to the servers/replicas
- Each (replicated) operation (e.g. read/write) requires a **quorum** (set of replicas)

Fundamental Property:

- If the result of one operation depends on the result of another, then their quorums must overlap, i.e. have common replicas

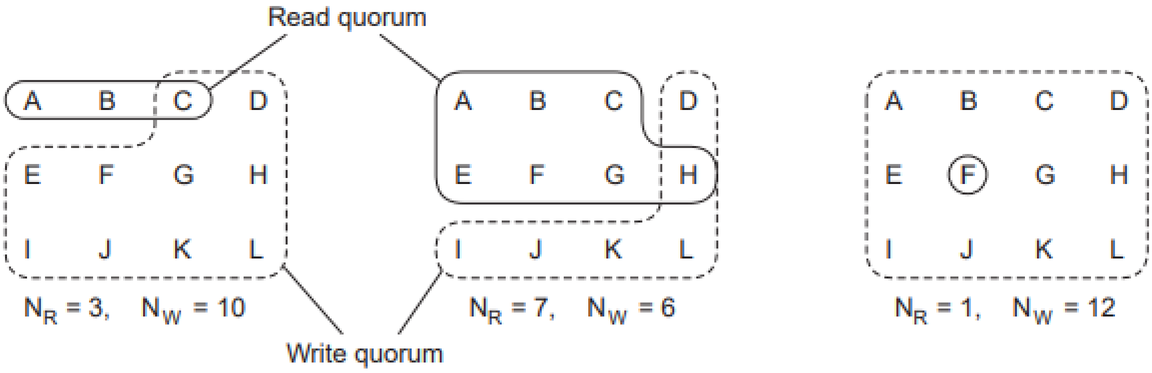
Defining Quorums

- Consider all replicas as peers
  - quorums are determined by their size, i.e. the number of replicas in the quorum
  - This is equivalent to assign 1 vote to each replica

Read/Write Quorums Must Overlap

- The replicas provide **only read** and **write** operations
- Because the output of a read operation depends on previous write operations, the read quorum must overlap the write quorum:

NR + NW > N, where
NR → size of the read quorum
NW → size of the write quorum
N → number of replicas



Implementation

Each object's replica has a version number

- **Read**

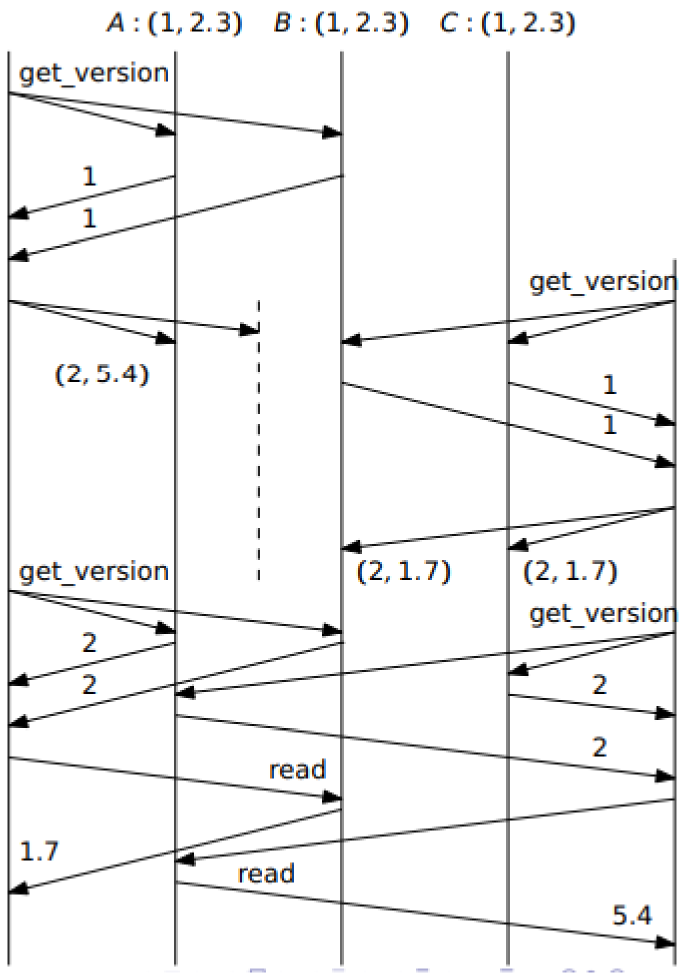
1. Poll a read quorum, to find out the current version
    - A server replies with the current version
2. Read the object value from an up-to-date replica
    - If the size of the object is small, it can be read as the read quorum is assembled

- **Write**
    1. Poll a write quorum, to find out the current version
        - A server replies with the current version
    2. Write the new value with the new version to a write quorum
        - We assume that writes modify the entire object, not parts of it

Note: A write operation depends on previous write operations (via the version) and therefore write quorums must overlap: NW + NW > N

Naive Implementation with Faults

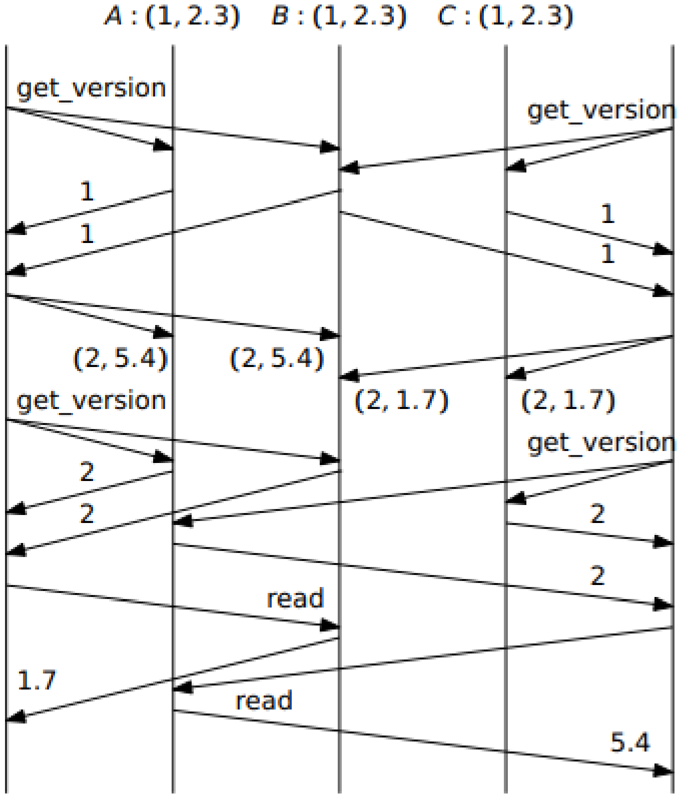
- N = 3, NR = 2, NW = 2



- First/left client attempts to write, but because of a partition it updates only one replica (A)
- Second/right client, in different partition, attempts to write and it succeeds.
- Variable has different values for the same version.
- The partition heals and each client does a read
- Each client gets a value different from the one it wrote.

Naive Implementation with Concurrent Writes

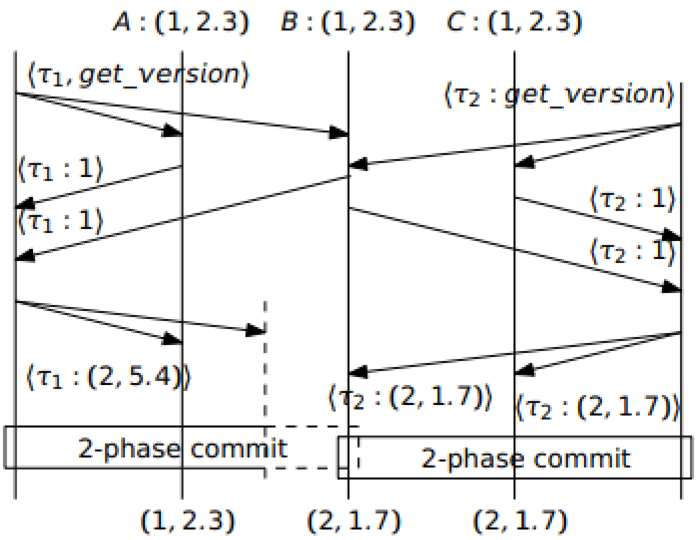
- N = 3, NR = 2, NW = 2



- Two clients attempt to write the replicas at more or less the same time
- The two write quorums are not equal, even though they overlap
- Again, replicas end up in an inconsistent state.
- Soon after, each client does a read
- Each client gets a value different from the one it wrote.

Ensuring Consistency with Transactions

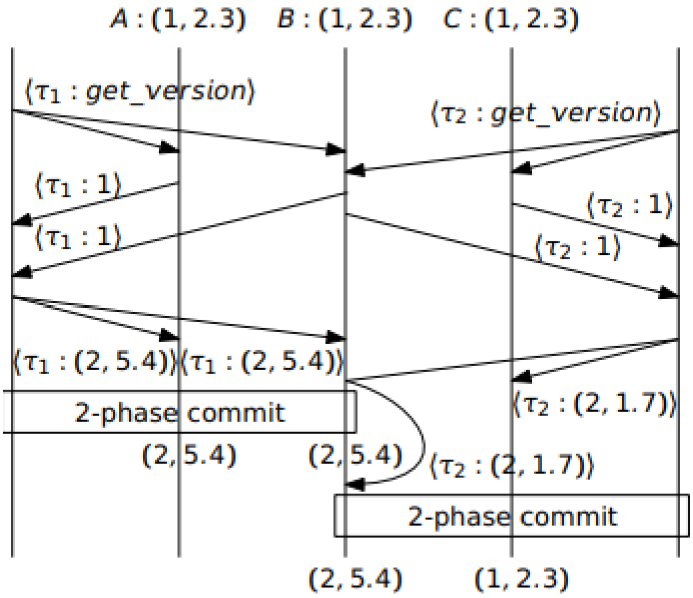
- The write (or read) of each replica is an operation of a distributed transaction
- If the write is not accepted by at least a write quorum, the transaction aborts



- The left client will not get the vote from replica B and therefore it will abort transaction τ1
- On the other hand, transaction τ2 commits, and its write will be effective.

Transactions also prevent consistencies in the case of concurrent writes

- Transactions ensure isolation, by using concurrency control
- Lets assume the use of locks.



- Server B processes the left client write request first, and acquires a write lock on behalf of τ1
- When server B processes the right client write request, it tries to acquire a write lock on behalf of τ2, but it is forced to wait for the termination ot τ1
- The commit of τ1 in server B invalidates the version number of τ2's write and therefore τ2 aborts.

XA-based Quorum Consensus Implementation

Each object's access is performed in the context of a transaction

Read

1. Poll a read quorum, to find out the current version
   - There is no need to read the object's state
   - Only the first time the transaction reads the object
2. Read the object state from an up-to-date replica
   - Only the first time the transaction reads the project

Write

1. Poll a write quorum, to find out the current version and which replicas are up-to-date

- On the first time the transaction writes the object:
    - Object state may have to be read from an up-to-date replica
    - Replicas may have to be updated

2. Write the new value with the new version
    - Replica rejects write if version is **not valid**
    - All writes by a transaction are applied to the same replicas
        - Because these will be the only ones with an up-to-date version

Transaction-based Quorum Consensus Replication

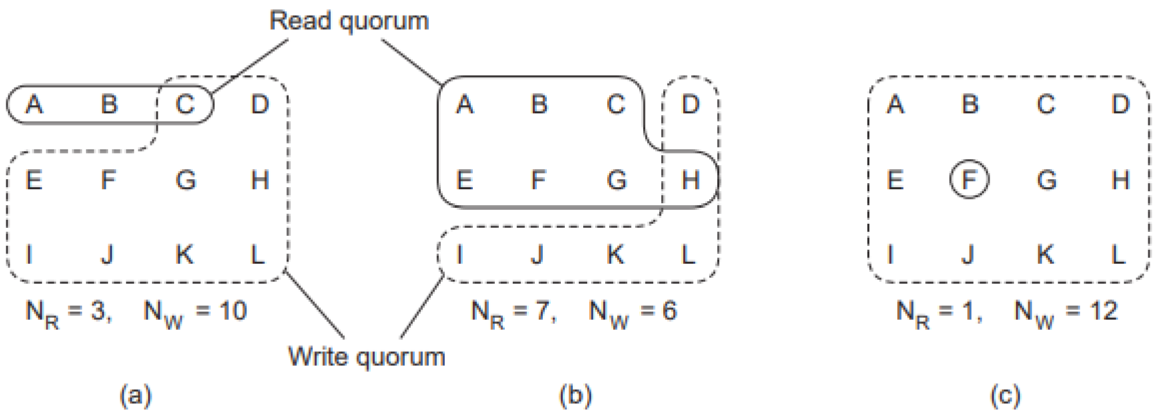
Pros

- Transactions solve both the problem of failures and concurrency
- Transactions can also support a more complex computations

Cons

- Deadlocks are possible, if transactions use locks
- Blocking if transactions use two-phase commit
- Availability problems

Playing with Quorums



(c) → **Read-one/Write-all protocol**

- By assigning each replica its own number of votes, which may be different from one, **weighted-voting** provides extra flexibility.

Quorum Consensus Fault Tolerance

- tolerates unavailability of replicas
- The availability analysis by Gifford relies on the probability of crashing of a replica/server

Dynamo Quorums

Dynamo

- replicated key-value storage system developed at Amazon
- uses quorums to provide high-availability
- enhances high-availability, by using multi-version objects

Dynamo's Quorums

- Each key is associated with a set of servers, the **preference list**
  - The first N servers in this list are the main replicas
  - The remaining servers are backup replicas and are used only in the case of failures
- I Each operation (get()/put()) has a coordinator, which is one of the first N servers in the preference list.
  - The coordinator is the process that executes the actions typically executed by the client in Gifford’s quorums

put(.) → requires a quorum of W replicas
get(.) → requires a quorum of R replicas
such that:
R + W > N

put(key, value, context) coordinator

1. Generates the version vector for the new version and writes the new value locally
   - The new version vector is determined by the coordinator from the **context**, a set of version vectors
2. Send the (key, value) and its version vector to the N first servers in the key's preference list
   - The put() is deemed successful if at least W–1 replicas respond

get(key) coordinator

1. Requests all versions of the (key, value) pair, including the respective version vectors, from the remaining first N servers in the preference list
2. On receiving the response from at least R–1 replicas, it returns all the (key,value) pairs whose version-vector are **maximal**
   - If there are multiple pairs, the application that executed the get() is supposed reconcile the different versions and write-back the reconciled pair using put().

Dynamo’s "Sloppy" Quorums and Hinted Handoff

- **Without failures** Dynamo provides strong consistency
- **In the case of failures** the coordinator may not be able to get a quorum from the N first replicas in the preference list

To **ensure availability**:

- the coordinator will try to get a **sloppy quorum** by enlisting the backup replicas in the preference list
    - The copy of the (key, value) sent to the backup server has a **hint** in its metadata identifying the server that was supposed to keep that copy
    - The backup server scans periodically the servers it is substituting

At the cost of consistency sloppy quorums do not ensure that every quorum of a get() overlaps every quorum of a put()

- **Sloppy quorums** are intended as a solution to temporary failures
    - To handle failures with a longer duration, Dynamo uses a anti-entropy approach for replica synchronization