Quorums Consensus Replication

Content

How is works

In this protocol clients can communicate directly with the servers/replicas.

Consider a system with N replicas/servers. The client needs to **assemble** N_r as reading replicas and N_w as write replicas.

Read operation

The client requests a read to a set of quorums: the read set and performs the following operations in order:

- The client asks what's the most recent version of the file.
- Requests the file from the given version

Write operation

The client must proceed with a similar operation from the previous point:

- Asks the client what is the most recent version x of the file.
- Writes the new file with version $x + 1$.

However, we have here some constraints about N_r and N_w.

Properties

- Notice that $\{\exists\ n_i \parallel n_i \in N_r \wedge n_i \in N_w\} \rightarrow N_r + N_w > N$. If the two quorum don't overlap, we will not able to determine what is the most recent version o a file.

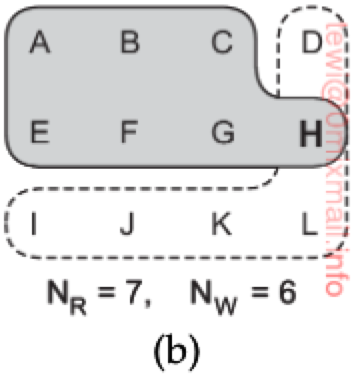


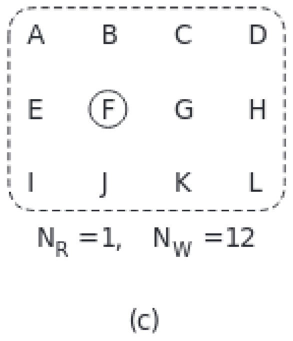
Image 1 - Tanebaum, page 403

Imagine that the client c_1 has $N_r = \{a, b, c, e, f, g\} \wedge N_w = \{d, h, l, i, j, k\}$. When the client tries to write, he will determine the most recent version only based on its write quorum.

- We still have the properties that $N_w \leq N/2$. Still having image 1 as reference, imagine that one client c_1 has $N_{w1} = \{a, b, c, e, f, g\}$ and a client c_2 has $N_{w2} = \{d, h, l, i, j, k\}$. If the the write quorums don't overlap, two updates might be accepted, without detecting that these both have a conflict.

Read-one/Write-all

The case on the right is interesting. Here we have just one reading copy and many write copies. The operation to read is really fast, but this is a trade off, since there's low availability. On the other hand the write system is slower (the velocity is dictated by slowest server in the network), but we have high availability.



Weighted voting

Still some replicas might have a higher weight in a vote. This provides extra flexibility.

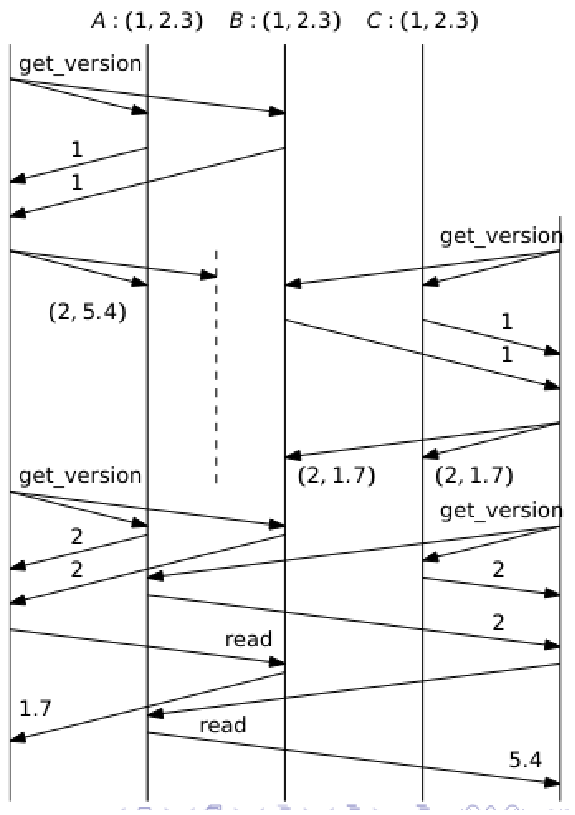
Ensuring consistency with transactions

Problem 1 (faults)

Imagine that two clients are writing in the system. If there's a fault in one peer, we will reach in a inconsistent state.

In the image on the right, it will not be possible to write in replica **B,** due to a network partition (e.g a problem in the switch). At the same time another client will write in peer **B** and **C.** The problem here is that all the peers will have in the end the same version of the file, but the content is different.

In the end we don't actually know what is the correct state.

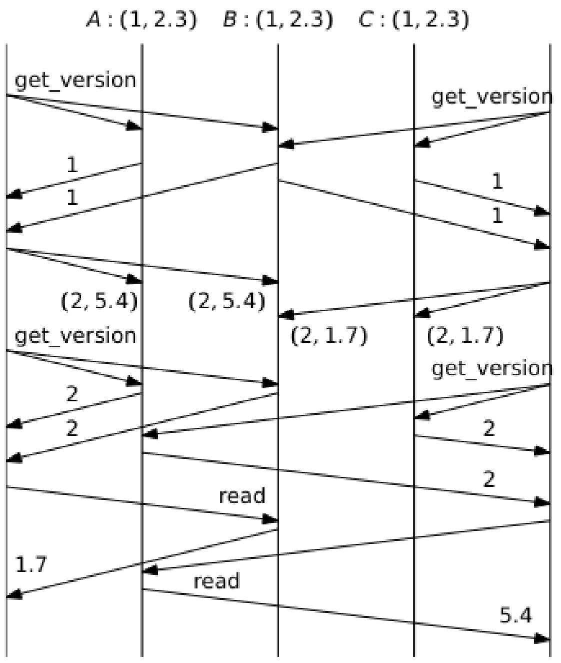


Problem 2 (concurrent writes)

We still have another problem where two peers tries to write at the same time.

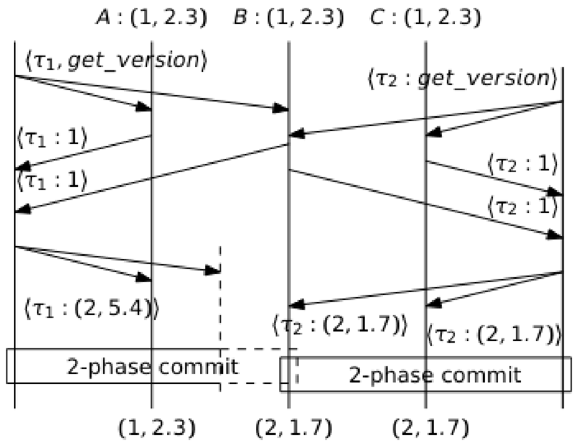
They might get the current version of the file, which is 1, and then one client writes it. When the other client writes, he might overwrite the content of the other.

Once again we reach to an inconsistent state.



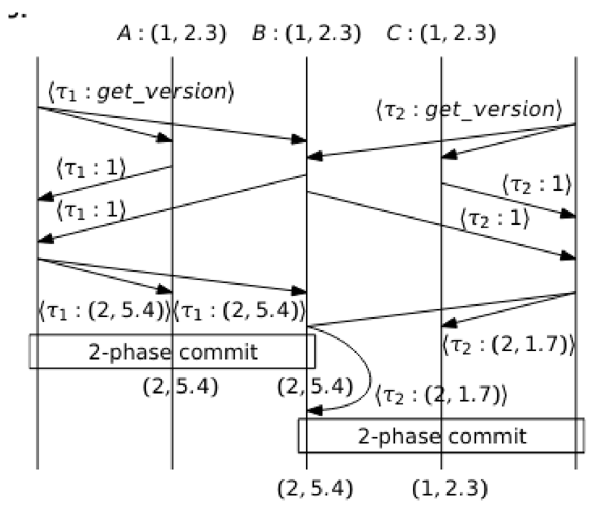
The solution (transactions)

The problem 1 can be easily solved with transactions, since the operation will not take place if **at least one quorum didn't accpted the operation.**



The problem 2 is also solved by using transactions. Transaction ensure isolation, so if one transaction is taking place another can't performed (e.g use of locks).

In the image on the write, when the left client writes a request in server B and then B locks the resources on behalf of t1. Since the resources are locked the operation t2 is cancelled.


$A : (1, 2.3)$
$B : (1, 2.3)$
$C : (1, 2.3)$
$\langle \tau_1 : get\_version \rangle$
$\langle \tau_2 : get\_version \rangle$
$\langle \tau_1 : 1 \rangle$
$\langle \tau_1 : 1 \rangle$
$\langle \tau_2 : 1 \rangle$
$\langle \tau_2 : 1 \rangle$
$\langle \tau_1 : (2, 5.4) \rangle$
$\langle \tau_1 : (2, 5.4) \rangle$
$\langle \tau_2 : (2, 1.7) \rangle$
2-phase commit
$(2, 5.4)$
$(2, 5.4)$
$\langle \tau_2 : (2, 1.7) \rangle$
2-phase commit
$(2, 5.4)$
$(1, 2.3)$

Transaction problems

Transaction deadlocks

Design your applications to avoid transaction deadlocks. A transaction needs exclusive control of resources while executing file control commands.

https://www.ibm.com/docs/en/cics-ts/5.6?topic=control-transaction-deadlocks

- It's always possible that transactions leads to **deadlocks**.
- And **blocking** may happen, if the coordinator fails in the wrong time. When this happens the other server must wait for the coordinator to recover. While this process happens other transactions will be aborted.

XA-based quorum consensus implementation ??

Dynamo's quorums

More than just a pair of (key, value), each replica of a (key, value) has a version vector.

Here we have N preferential servers and the remaining servers are just for backup.

Put(key, value, context)

In this case the coordinator acts:

- The new version vector is determined by the coordinator from the **context,** a set of version vectors.
- It send the (key,value) and its vesino vector to the N first servers in the key's preference list. The put is successful if at least W-1 replicas respond (it's -1, since the coordinator itself is one).

Get(key)

The coordinator:

- Requests all versions (key, value) pair, inclusive the respective version vector.
- Once received the response from at least R-1 replicas (the coordinator is one of R), it returns all the (key,value) pair whose version vectors are maximal (the most recent). If there're multiple pair returned, it is supposed to reconcile the different versions and write-back the reconciled pair using put().

Sloppy quorums

In case of failure the coordinator will not be able to get response from all the N first replicas.

To ensure **availability** the coordinator will try to get a sloppy quorum, by setting one of the backup replica in the preference list. The backup replicas will make the copies necessary in the name of the server **s.**

The backup server knows who he is substituting and checks periodically if the server is back active. If yes, it will transfer the information it stored to it.

However, this is a temporary solution. The longer duration uses an anti-entropy approach for replica synchronization. Also, sloppy quorums do not ensure that every quorum of a get() overlaps every quorum of a put().

Questions

1) What is the advantage of a replica with 0 votes?

▼ Answer

Not sure, but i think that replicas with zero vote can be used later as a backup, aren’t part of the group.

2) Let f be the maximum number of replicas that may crash simultaneously. (Assume 1 replica, 1 vote).
a) What is the minimum number of replicas that we need?

b) Do we need to change the quorum constraints?