Replication for Fault Tolerance

Quorum Consensus

Pedro F. Souto (pfs@fe.up.pt)

January 3, 2022

Roadmap

Quorums and Quorum Consensus Replication

Ensuring Consistency with Transactions

Playing with Quorums

Dynamo Quorums

Further Reading

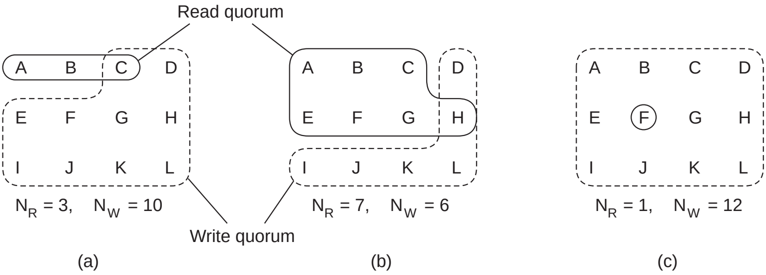
Quorum Consensus Protocols

- In these protocols, clients communicate directly to the servers/replicas
  - Unlike in Primary Backup or State Machine Replication with Paxos
- Each (replicated) operation (e.g. read/write) requires a **quorum**
  - This is a set of replicas
- The fundamental property of these quorums is that
  - If the result of one operation depends on the result of another, then their quorums must overlap, i.e. have common replicas
- A simple way to define quorums is to consider all replicas as peers.
  - In this case 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
    - In his work, Gifford proposed the use of weighted voting, i.e. the assignment of different votes to each replica, so as to obtain different trade-offs between performance and availability of the different operations

Read/Write Quorums Must Overlap

- The replicas provide **only read** and **write** operations
  - These operations apply to the whole object
- Because the output of a read operation depends on previous write operations, the read quorum must overlap the write quorum: $N_R + N_W > N$, where

N_R is the size of the read quorum
N_W is the size of the write quorum
N is the number of replicas



Quorum Consensus Implementation

IMP 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

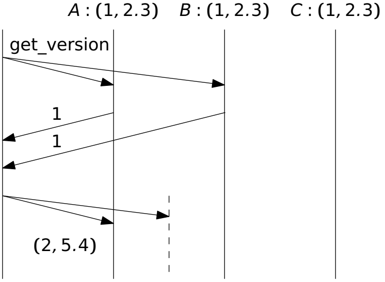
1. Poll a **write quorum**, to find out the current version
   - A server replies with the current version
2. Write the new value with the new version to a **write quorum**
   - We assume that writes modify the entire object, not parts of it

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

- Quorum b) above, ($N_R = 7, N_W = 6, N = 12$) violates this requirement

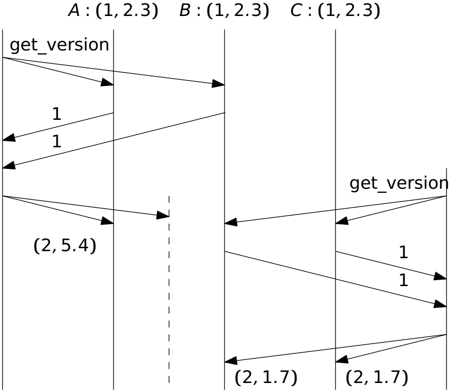
Naïve Implementation with Faults

- $N = 3, N_R = 2, N_W = 2$
- First/left client attempts to write, but because of a partition it updates only one replica (A)



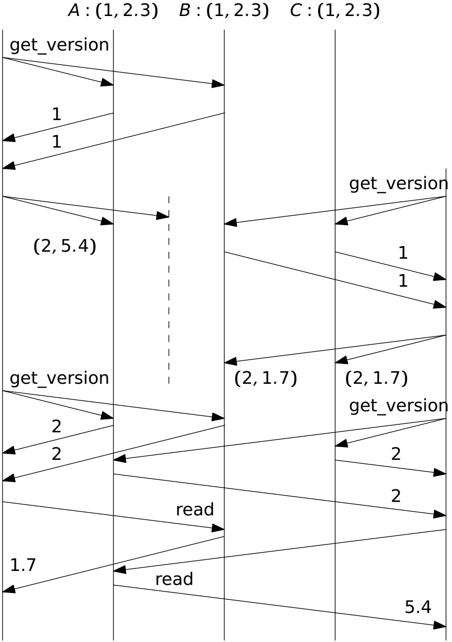
Naïve Implementation with Faults

- $N = 3, N_R = 2, N_W = 2$
- 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.



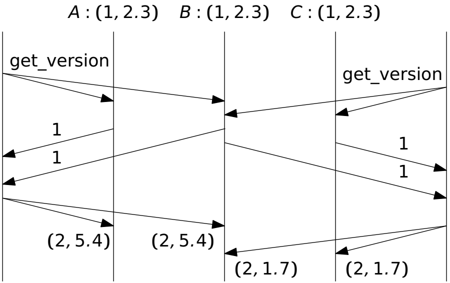
Naïve Implementation with Faults

- $N = 3, N_R = 2, N_W = 2$
- 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.
    - I.e. protocol does not ensure **read-your-writes**



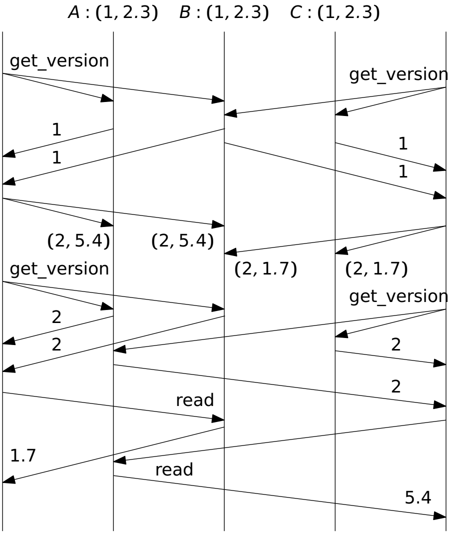
Naïve Implementation with Concurrent Writes

- $N = 3, N_R = 2, N_W = 2$
- 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.



Naïve Implementation with Concurrent Writes

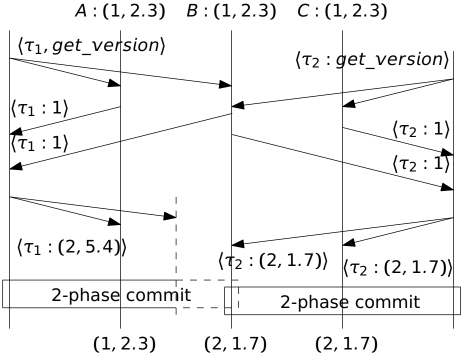
- $N = 3, N_R = 2, N_W = 2$
- Two clients attempt to write the replicas at more or less the same time
- The two write quorums are not equal, even though they overlap
- Again, replicas end up in an inconsistent state.
- Soon after, each client does a read
- Each client gets a value different from the one it wrote.



Roadmap

Ensuring Consistency with Transactions (1/2)

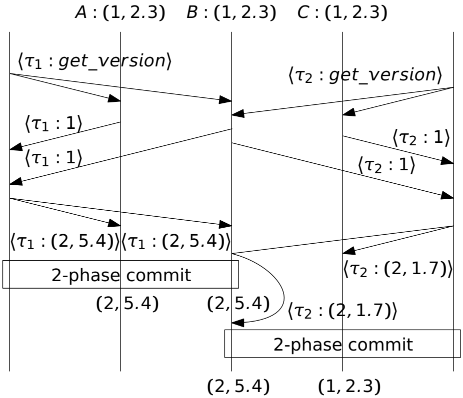
- Gifford assumes the use of transactions, which use two-phase commit, or some variant
  - The write (or read) of each replica is an operation of a distributed transaction
    - We can view the sequence of operations in a replica on behalf of a distributed transaction as a sub-transaction on that replica
  - 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 τ_1
  - The state of replica A will not be changed
- On the other hand, transaction τ_2 commits, and its write will be effective.

Ensuring Consistency with Transactions (2/2)

- 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 τ_1
- When server B processes the right client write request, it tries to acquire a write lock on behalf of τ_2, but it is forced to wait for the termination ot τ_1



- The commit of τ_1 in server B invalidates the version number of τ_2's write and therefore τ_2 aborts.

XA-based Quorum Consensus Implementation

IMP 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 object

Write (supporting **writes to an object's part**)

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

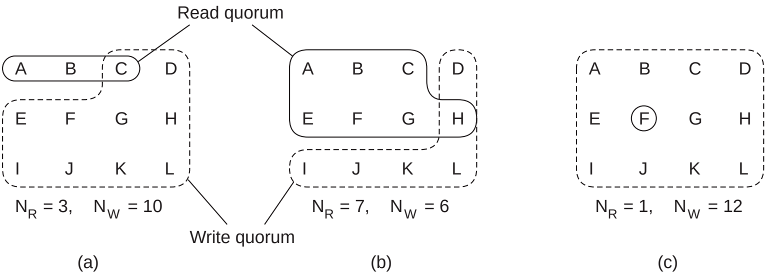
- Transactions solve both the problem of failures and concurrency.
- Transactions can also support a more complex computations:
  - E.g. with multiple operations and/or multiple replicated objects
- But, transactions also have problems on their own:

  Deadlocks are possible, if transactions use locks
    - Can deadlock also occur when a transaction comprises a single operation on one object?
    - Other concurrency control approaches, e.g. optimistic CC based on timestamps, may be used
      - These also have trade-offs

  Blocking if transactions use two-phase commit
    - If the coordinator fails at the wrong time, the participants, i.e. the servers, may have to wait for the coordinate to recover
      - Meanwhile, the objects accessed by such a transaction may become inaccessible, causing aborts of other transactions
    - It may be a good idea to use as coordinator proxy servers instead of clients, because the latter are failure-prone
- Not forgetting the availability problems induced by transactions

Roadmap

Playing with Quorums (1/2)



- The quorum in c) corresponds to a protocol known as **read-one/write-all**
  - By choosing N_R and N_W appropriately we can trade off performance and availability of the different operations

Playing with Quorums (2/2)

- By assigning each replica its own number of votes, which may be different from one, **weighted-voting** provides extra flexibility. E.g., assuming the crash probability of each replica to be 0.01:

	Example 1	Example 2	Example 3
Latency (msec)			
Representative 1	75	75	75
Representative 2	65	100	750
Representative 3	65	750	750
Voting Configuration	$\langle 1, 0, 0 \rangle$	$\langle 2, 1, 1 \rangle$	$\langle 1, 1, 1 \rangle$
r	1	2	1
w	1	3	3
Read			
Latency (msec)	65	75	75
Blocking Probability	1.0×10^{-2}	2.0×10^{-4}	1.0×10^{-6}
Write			
Latency (msec)	75	100	750
Blocking Probability	1.0×10^{-2}	1.0×10^{-2}	3.0×10^{-2}

source: Gifford79

Question What is the advantage of a replica with 0 votes?

Quorum Consensus Fault Tolerance

- Quorum-consensus tolerates unavailability of replicas
  - This includes unavailability caused by both process (replicas) failures and communication failures, including partitions
  - Actually, quorum consensus replication does not require distinguishing between the two types of failure
- The availability analysis by Gifford relies on the probability of crashing of a replica/server
  - But we can follow the standard approach to evaluate the resiliency of a fault-tolerant protocol in a distributed system

    Question Let f be the maximum number of replicas that may crash simultaneously.

    - What is the minimum number of replicas that we need?
    - Do we need to change the quorum constraints?

    (Assume 1 replica, 1 vote).

Roadmap

Dynamo

- Dynamo is a **replicated key-value storage system** developed at Amazon
- It uses quorums to provide high-availability
  - Whereas Gifford's quorums support a simple read/write memory abstraction, Dynamo supports an associative memory abstraction, essentially a put(key,value)/get(key) API
  - Rather than a simple version number, each replica of a (key,value) pair has a version vector
- Dynamo further enhances high-availability, by using multi-version objects
  - Thus sacrificing strong consistency under certain failure scenarios

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
- 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
    - As well as the actions required from a replica
- As in Gifford's quorums:

  put(.) requires a quorum of W replicas
  get(.) requires a quorum of R replicas

  such that:

$$R + W > N$$

Dynamo's Quorums

put(key,value,context) the 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. Sends 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) the coordinator

- Requests all versions of the (key, value) pair, including the respective version vectors, from the remaining first N servers in the preference list
- 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().

Without failures Dynamo provides strong consistency

Dynamo's "Sloppy" Quorums and Hinted Handoff

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
  - Upon detecting the recovery of a server, it will attempt to transfer the copy of the (key,value)
  - If it succeeds, the backup server will delete its local copy

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

Roadmap

Further Reading

- David K. Gifford, *Weighted Voting for Replicated Data*, SOSP'79: Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP'79), 1979, Pages 150-162
  - Section 4 describes several refinements of the basic idea (weighted voting) that allow to improve reliability or performance
- van Steen and Tanenbaum, *Distributed Systems*, 3rd Ed.
  - Section 7.5.3: Replicated-Write Protocols
- Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. *Dynamo: amazon's highly available key-value store.* In Proceedings of twenty-first ACM SIGOPS Symposium on Operating systems principles (SOSP '07), 2007. Pages 205–220.