Quorum Consensus Replicated ADT

Content

Introduction

When executing an operation a client may do two operations:

- Read from an **initial quorum**
- Write into a **final quorum**

One of these two sets might be empty.

Then a quorum for an operation is any set of replicas that includes both an initial set of quorums and a final set. The pair **initial** and **final quorums** can be represented as **(m,n),** where **m** is the number of **inital quorums** and **n** is the number of final quorums.

Example: Gifford’s Read/Write quorums

As we have seem in 🫂 Quorums Consensus Replication to we perform an operation of write in the *Gifford's quorums model* we need to determine the most recent version of a file and then write. The Read operation determines the correct version on the act of read. Thus, the representation of these two operations are:

- Write: (m_1, n_1)
- Read: $(m_2, empty)$

However, to avoid errors on the version we must the following constraints:

- **The final quorum of write must overlap the next initial quorum of write.** If this is not done, it will not be possible to determine the correct version in the next write. For this reason $W_w > n/2$, where n is the total number of quorums.
- **The write operation and the read must overlap.** Thus $W_w + R_r > n$.

These constraints leads us to the following graph:

Where the beggining of the arrow mean the final state and the end of the arrow means the initial quorum state. Thus the graph can be translated as:

- The final quorum of write must intersect the initial quorum.
- The final quorum of write must intersect the initial quorum of read.

Thus, we just have the following options of systems:

Operation	quorum choices		
read	(1,0)	(2,0)	(3,0)
write	(1,5)	(2,4)	(3,3)

The problem with the first column is that, since quorums operations are transactions, by having 5 replicas in the final quorum of write, we are decreasing the availability of the write operation, but increasing the availability and probably speed in the read operation. Because if one of the 5 replicas fails, then the operation will not take place.

Gifford’s-based Replicated Queue

This type of implementation is built on top of writing and reading files.

The queue has two types of operations:

- **Enq**: Adds an item to the queue.
- **Deq:** Removes the oldest item of the queue and raises an exception if the queue is empty.

Then a simple **Deq** using this approach, follows as:

- First we select a read quorum to read the current state of the file.
- Then we read the state of an updated replica
- If the queue is not empty, **normal deq:**
  - Remove the item at the head of the queue
  - Write the new state to a final write quorum
  - Return the item removed
- If the queue is empty, we have an **abnormal deq.** This leads us to an exception.

Only the last option here makes sense, since the other choices would favor Abnormal Deq over both Normal Deq and Enq.

Operation	quorum choices		
Enq	(1,5)	(2,4)	(3,3)
Normal Deq	(1,5)	(2,4)	(3,3)
Abnormal Deq	(1,0)	(2, 0)	(3,0)

Herlihy’s Replication Method

Instead of version numbers, **Herlihy’s** uses **timestamps** and **logs** to track the changes.

The advantages of the use of **timestamp** is:

- **The number of messages is reduced.** Since it’s not necessary to read in order to find the most recent version.
- **Reduce the contraints from the Gifford’s quorums.** Since we don’t need to find the most recent version of the file, the final quorum of write doesn’t need to overlap the changes in another initial quorum of write.

The *assumption* is that client is able to generate timestamps, so that we have a **linearizability.**

Thus the **Minimal Quorum Choices** became:

Operation	Minimal Quorum Choices				
Read	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)
Write	(0,5)	(0,4)	(0,3)	(0,2)	(0,1)

> This changes are well written in [1] pages 35 and 36.

Replicated Event Logs vs Replicate State

Terminology

The *operation* are the means to change an object. Examples of operations are: *Read(), Write(), Deq(), Enq().*

An *event* is the pair [operation, response]. A response can be an Ok(), for example. An example of operation is: [Read(), Ok(x)] or [Write(x), Ok()].

A log is an *event* with a timestamp in the format t0:[op(args); term(results)]. E.g t0: [Enq(x); Ok()].

A *history* is a set of logs.

Replicating with timestamps

Now the read is similar to thethe previous version.

But the Write operation has no need o read the most recent version.

Minimal quorum choices for 5 replicas (treated as equals)

Operation	Minimal Quorum Choices				
Read	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)
Write	(0,5)	(0,4)	(0,3)	(0,2)	(0,1)

Dequeue

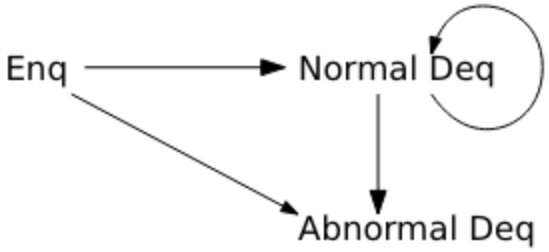
The operation of Deq is done by the following way in the perspective of the frontend.

- First the client reads the log from the **initial dequeue** quorum and recreates the structure: the *view.* It's done by merging the history by the timestamping. Then duplicates are discarded.
- If the queue is not empty it **records the Deq event in the local history** and send the modified view o a final Deq quorum of replicas.
- If the queue is empty it raises an exception and do not write.

In the end we gonna have the following contraints:

- Every initial **Deq** quorum must intersect every final **Enq** operation. So the **Deq** operation will reflect over every Enq operation.
- Every initial Deq quorum must intersect every final Deq quorum. So that the reconstructed queue reflcts all previous Deq events.

In the end our graph is represented like this:



Operation	Quorum Choices		
Enq	(0,1)	(0,2)	(0,3)
Normal Deq	(3,1)	(2,2)	(1,3)
Abnormal Deq	(3,0)	(2,0)	(1,0)

Op.	Rep. 1	Rep. 2	Rep. 3
Enq(x): R1, R2	t1:[Enq(x);Ok()]	t1:[Enq(x);Ok()]	t1:[Enq(x);Ok()]
Deq(): R2, R3	t2:[Deq():Ok(x)]	t2:[Deq():Ok(x)]	t2:[Deq():Ok(x)]
Enq(y): R1, R2	t3:[Enq(y);Ok()]	t3:[Enq(y);Ok()]	t3:[Enq(y);Ok()]
Enq(z): R1, R3	t4:[Enq(z);Ok()]		t4:[Enq(z);Ok()]
Deq(): R1, R3	t5:[Deq():Ok(y)]		t5:[Deq():Ok(y)]

- A missing entry is represented as blank

Minimal quorum choices for 5 replicas (treated as equals)

Operation	Quorums		
Enq	(0,1)	(0,2)	(0,3)
Normal Deq	(5,1)	(4,2)	(3,3)
Abnormal Deq	(5,0)	(4,0)	(3,0)

Operation	Quorum
Enq	(3,3)
Normal Deq	(3,3)
Abnormal Deq	(3,0)

With Gifford's method (the one in the right) we just have one option, since the others would prioritize the Abnormal Enq over two other operations. In the Herliye's method we have an harmony.

Optimizations

The disadvantage of **Herlihy's** method is that the *history* might grow indefinitely. A solution for this is to implement a garbage collector:

- If an item has been dequeued, all items with earlier timestamps msut have been dequeued.
- We cannot just remove all the entries with earlier timestamps in a single quorum. Upon a merge these entries might be added again.
- But we can keep a variable called **horizon timestamps,** where we keep the timestamp of the most recently dequeued item.

For example:

Example trace **Enq(x)R1R2**Deq()R2R3

Rep. 1	Rep. 2	Rep. 3
horizon: 0	horizon: 0	horizon: 0
t1:[Enq(x);Ok()]	t1:[Enq(x);Ok()]	

Example trace Enq(x):R1R2 **Deq()R2R3**Enq(y)R1R2

Rep. 1	Rep. 2	Rep. 3
horizon: 0	horizon: t1	horizon : t1
t1:[Enq(x);Ok()]		

By performing **Deq** in will set the horizon as t1 and delete the logs with timestamp previous to t1.

Ex. trace Enq(x):R1R2 Deq()R2R3 Enq(y)R1R2 **Enq(z)R1R2** Deq()R1R3

Rep. 1	Rep. 2	Rep. 3
horizon: 0	horizon: t1	horizon : t1
t1:[Enq(x);Ok()]		
t2:[Enq(y);Ok()]	t2:[Enq(y);Ok()]	
t3:[Enq(z);Ok()]		t3:[Enq(z);Ok()

Ex. trace Enq(x):R1R2 Deq()R2R3 Enq(y)R1R2 Enq(z)R1R2 **Deq()R1R3**

Rep. 1	Rep. 2	Rep. 3
horizon: t2	horizon: t1	horizon : t2
	t2:[Enq(y);Ok()]	
t3:[Enq(z);Ok()]		t3:[Enq(z);Ok()

Now if we perform a dequeue in R1 and R3 we can get the horizon in R3 in increment it. The Rep1 will follow this horizon.

References

[1] M. Herlihy, “A quorum-consensus replication method for abstract data types,” *ACM Trans. Comput. Syst.*, vol. 4, no. 1, pp. 32–53, Feb. 1986, doi: 10.1145/6306.6308.

http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-319.pdf