Replication for Fault Tolerance

Quorums-Consensus Replicated ADT

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

January 6, 2022

Roadmap

Initial and Final Quorums

(Herlihy's) Replicated ADTs

Replicated Queue: an example of a replicated ADT

Critical Evaluation

Quorum-Consensus and Replicated Abstract Data T.

- Herlihy proposed a generalization of quorum consensus to replicated abstract data types such as queues.

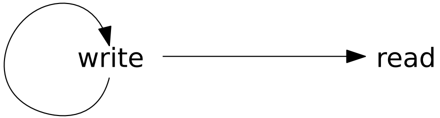
Quorum for an operation is any set of replicas whose cooperation is sufficient to execute the operation.

- When executing an operation, a client:
  - reads from an **initial quorum**
  - writes to a **final quorum**
- For example, in the read operation, a client must read from some set of replicas, but its final quorum is empty.
- A quorum for an operation is any set of replicas that includes both an initial and a final quorum.
- Assuming that all replicas are considered equals, a quorum may be represented by a pair, (m, n), whose elements are the sizes of its initial, m, and its final, n, quorums

- Quorum intersection constraints are defined between the final quorum of one operation and the final quorum of another

Example: Gifford's Read/Write Quorums

- Object (e.g. file) read/write operations are subject to two constraints:
  1. Each final quorum for write must intersect each initial quorum for read
  2. Each final quorum for write must intersect each initial quorum for write
     - To ensure that versions are updated properly
- These constraints can be represented by the following quorum intersection graph:



- Choices of minimal (size) quorums for an object with 5 replicas:

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

Gifford's-based Replicated Queue (1/2)

- Gifford's read/write quorums can be used to implement arbitrary data types
  - Any data type can be built on top of memory read/write operations
- A queue has two basic operations:

  Enq adds an item to the queue

  Deq removes least recent the item from the queue, raising an exception if the queue is empty

  1. Read an initial **read quorum** to determine the current version of the queue
  2. Read the state from an updated replica
  3. If the queue is not empty, **normal deq**:
     - 3.1 Remove the item at the head of the queue
     - 3.2 Write the new queue state to a final **write quorum**
     - 3.3 Return the item removed in 3.1
  4. If the queue is empty, **abnormal deq**, raise an exception

Gifford's-based Replicated Queue (2/2)

- From the minimal quorum choices for the read/write operations:

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

  we can derive the following minimal quorum choices for the operations on a replicated queue using read/write quorums:

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)

- Only the quorum choice in the last column makes sense
  - The other choices would favor Abnormal Deq over both Normal Deq and Enq

Roadmap

Initial and Final Quorums

(Herlihy's) Replicated ADTs

Replicated Queue: an example of a replicated ADT

Critical Evaluation

Herlihy's Replication Method

Timestamps instead of version numbers

- Reduce quorum intersection constraints
- Reduce messages

Logs instead of (state) versions

- These changes allow for more flexible replication quorums

Assumption Clients are able to generate timestamps that can be totally ordered

- This order is consistent to that seen by an omniscient observer, i.e. consistent with **linearizability**

Replicated Read/Write Objects with Timestamps

Read similar to the version-based, except that a client uses the timestamp instead of the version to identify a replica that is up-to-date

Write there is no need to read the versions from an initial quorum:

- Timestamp generation guarantees total order consistent with the order seen by an omniscient observer
- No need for initial message round
- Client needs only write the new state to a final quorum
  - Suitable only for whole state changes

Quorum intersection graph

$$\text{write} \longrightarrow \text{read}$$

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)

Replicated Event Logs vs Replicate State

Event State change, represented as a pair of:

- **Operation** with respective arguments, e.g. Read() or Write(x)
- **Outcome** a termination condition and returned results, e.g. Ok(x) or Ok()

E.g. [Read(), Ok(x)] and [Write(x), Ok()]

Event log a sequence of log entries

Log entry is a timestamped event:

$$t_0 : [op(args); term(results)]$$

E.g., an Enq event in a queue might be:

$$t_0 : [Enq(x); Ok()]$$

- Entries in a log are ordered by their timestamps

Idea rather than replicate state, replicate event logs

- An event log subsumes the state

Roadmap

Initial and Final Quorums

(Herlihy's) Replicated ADTs

Replicated Queue: an example of a replicated ADT

Critical Evaluation

Herlihy's Replicated Queue

Deq implementation – Client:

1. reads the logs from an inital Deq quorum and creates a view

   View is a log obtained by:

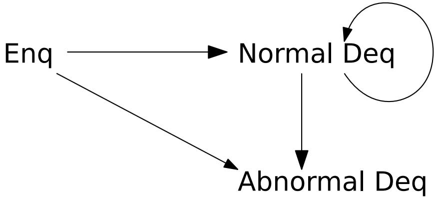
   1.1 merging in timestamp order the entries of a set of logs
   1.2 discarding duplicates
2. reconstructs the queue state from the view, and finds the item to return
3. if the queue is not-empty:

   3.1 records the Deq event, by appending a new entry to the view
   3.2 sending the modified view to a final Deq quorum of replicas

   - Replicas merge this view with their local logs
4. returns the response (the dequeued item or an exception) to Deq's caller.

Note: This is just conceptual implementation. There are many possible optimizations to improve performance.

Herlihy's Replicated Queue: Constraints



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)

1. Every initial Deq quorum must intersect every final Enq quorum
   - So that the reconstructed queue reflects all previous Enq events
2. Every initial Deq quorum must intersect every final Deq quorum
   - So that the reconstructed queue reflects all previous Deq events

Note 1 The views for Enq operations need not include any prior events, because Enq returns no information about the queue's state

- An initial Enq quorum may be empty.

Note 2 As before, an abnormal Deq has an empty final quorum.

Herlihy's Replicated Queue: Example Execution Trace

Op.	Rep. 1	Rep. 2	Rep. 3
Enq(x): R1, R2			

Herlihy's Replicated Queue: Example Execution Trace

Op.	Rep. 1	Rep. 2	Rep. 3
Enq(x): R1, R2	**t1:[Enq(x);Ok()]**	**t1:[Enq(x);Ok()]**	
Deq(): R2, R3			

- A missing entry is represented as blank

Herlihy's Replicated Queue: Example Execution Trace

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)]**
Enq(y): R1, R2			

- A missing entry is represented as blank

Herlihy's Replicated Queue: Example Execution Trace

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)]
Enq(y): R1, R2	**t3:[Enq(y);Ok()]**	**t3:[Enq(y);Ok()]**	
Enq(z): R1, R3			

- A missing entry is represented as blank
- No single replica contains all the entries that define the queue's state

Herlihy's Replicated Queue: Example Execution Trace

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)]
Enq(y): R1, R2	t3:[Enq(y);Ok()]	t3:[Enq(y);Ok()]	
Enq(z): R1, R3	**t4:[Enq(z);Ok()]**		**t4:[Enq(z);Ok()]**
Deq(): R1, R3			

- A missing entry is represented as blank

Herlihy's Replicated Queue: Example Execution Trace

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

Herlihy's Replicated Queue: Example Execution Trace

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)

Herlihy's Replicated Queue: Example Execution Trace

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 there is only one quorum choice
  - Herlihy's approach allows Enq to be more available, at the expense of Deq's availability

Herlihy's Replicate Queue: Optimizations (1/2)

Disadvantages logs and **messages** grow indefinitely

Fixes

Garbage collect logs take advantage of observation

- If an item has been dequeued, all items with earlier timestamps must have been dequeued
- However, we cannot just remove all the entries with earlier timestamps
  - Otherwise, some of these might be added again upon merging logs
- But it is enough to keep the **horizon timestamp**, i.e. the timestamp of the most recently dequeued item
  - In addition to the horizon timestamp, we need a log **with only Enq entries**, whose timestamps are later than the horizon timestamp

Cache logs at clients

Herlihy's Replicate Queue: Optimizations (2/2)

Deq implementation

1. Client reads from an initial Deq quorum
   1.1 the horizon timestamp
   1.2 the local logs (which include only Enq log entries)
2. The client:
   2.1 creates a view as before
   2.2 discards all entries earlier than the latest observed horizon time
3. The oldest remaining Enq entry indicates
   - the item to dequeue
   - the new horizon time
4. The client writes the new horizon time to a final Deq quorum

Example trace Enq(x)R1R2

Rep. 1	Rep. 2	Rep. 3
horizon: 0	horizon: 0	horizon: 0

Herlihy's Replicate Queue: Optimizations (2/2)

Deq implementation

1. Client reads from an initial Deq quorum
   1.1 the horizon timestamp
   1.2 the local logs (which include only Enq log entries)
2. The client:
   2.1 creates a view as before
   2.2 discards all entries earlier than the latest observed horizon time
3. The oldest remaining Enq entry indicates
   - the item to dequeue
   - the new horizon time
4. The client writes the new horizon time to a final Deq quorum

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()]	

Herlihy's Replicate Queue: Optimizations (2/2)

Deq implementation

1. Client reads from an initial Deq quorum
   1.1 the horizon timestamp
   1.2 the local logs (which include only Enq log entries)
2. The client:
   2.1 creates a view as before
   2.2 discards all entries earlier than the latest observed horizon time
3. The oldest remaining Enq entry indicates
   - the item to dequeue
   - the new horizon time
4. The client writes the new horizon time to a final Deq quorum

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()]		

Herlihy's Replicate Queue: Optimizations (2/2)

Deq implementation

1. Client reads from an initial Deq quorum
   1.1 the horizon timestamp
   1.2 the local logs (which include only Enq log entries)
2. The client:
   2.1 creates a view as before
   2.2 discards all entries earlier than the latest observed horizon time
3. The oldest remaining Enq entry indicates
   - the item to dequeue
   - the new horizon time
4. The client writes the new horizon time to a final Deq quorum

Example trace

Enq(x):R1R2 Deq()R2R3 **Enq(y)R1R2** Enq(z)R1R2

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

Herlihy's Replicate Queue: Optimizations (2/2)

Deq implementation

1. Client reads from an initial Deq quorum
   1.1 the horizon timestamp
   1.2 the local logs (which include only Enq log entries)
2. The client:
   2.1 creates a view as before
   2.2 discards all entries earlier than the latest observed horizon time
3. The oldest remaining Enq entry indicates
   - the item to dequeue
   - the new horizon time
4. The client writes the new horizon time to a final Deq quorum

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()

Herlihy's Replicate Queue: Optimizations (2/2)

Deq implementation

1. Client reads from an initial Deq quorum
   1.1 the horizon timestamp
   1.2 the local logs (which include only Enq log entries)
2. The client:
   2.1 creates a view as before
   2.2 discards all entries earlier than the latest observed horizon time
3. The oldest remaining Enq entry indicates
   - the item to dequeue
   - the new horizon time
4. The client writes the new horizon time to a final Deq quorum

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()

Roadmap

Initial and Final Quorums

(Herlihy's) Replicated ADTs

Replicated Queue: an example of a replicated ADT

Critical Evaluation

Issues with Replicated ADTs

Timestamps generated by **clients** and **consistent with linearizability**

- Herlihy's relies on transactions, and hierarchical timestamps
  - So the problem reduces to that of ordering transactions
  - However, if we use locking the serial order ensured by transactions is usually determined at commit time
- If replicated ADTs do not use transactions this is challenging
  - How do clients generate a timestamp consistent with linearizability, if the initial quorum is empty (e.g. in the case of Enq)?

Logs must be garbage collected to bound the size of messages

- Garbage collecting log entries is ADT-dependent
  - Herlihy's solution for replicated queues may be very effective
  - But it may be harder or less effective for other replicated ADTs

(Herlihy's) Replicated ADTs vs CvRDTs

- There are clear similarities between replicated ADTs and CvRDTs, i.e. state-based CRDTs
  - They both support replicated data types
  - Herlihy's logs appear to be a monotonic semi-lattice object.
    - Log entries merging is similar to CvRDTs' merge operation
  - They both require design ingenuity
    - Unless, you can solve your problem with a cataloged solution
    - Actually, it is not clear whether you are able to implement some data types. Is there a conflict-free replicated queue?
- However there are also important differences:
  - CRDTs do not ensure strong consistency, but strong eventual consistency
  - However, CRDTs will likely be more available, an operation can be executed as long as there is one accessible replica
    - Replicated ADTs require a quorum to perform one operation
- Can we easily convert an implementation of a replicated ADT to an implementation of a CvRDT?
  - E.g. just by dropping the quorum intersection constraints?

Quorum Consensus: Final Thoughts

- Quorum-based systems are usually restricted to "simple" data storage systems
  - Herlihy generalized this approach to other ADT, including **dictionaries**, i.e. mappings from keys to values
- SMR with Paxos, uses majority voting – which is a special kind of quorum
  - What is the **real** difference between these two approaches?
  - The concept of quorum appears to be fundamental
- Quorums need not be restricted to assigning/counting votes
  - Consider replicas laid out in a square of l by l replicas
    - Let a read quorum be the set of (l) replicas in one column.
    - Try to find out write and read quorums that are **both minorities** (with less than $l^2/2 + 1$ replicas)
    - What would be the advantage of such quorums?
- Quorums may be dynamic, e.g. by changing the replicas and/or the vote assignments
  - Changing quorum configurations can be tricky

Further Reading

- Maurice Herlihy, *A quorum-consensus replication method for abstract data types*, in ACM Transactions on Computer Systems (TOCS), (4)1:32-53 (February 1986)
- Maurice Herlihy, *Replication methods for abstract data types*, Tech. Rep. MIT/LCS/TR-319 (May 1984)
  - Chapter 2, Sections 2.1 to 2.3, i.e. about 15 pages, are all you need