The University of Texas at Austin

Distributed Systems

Software Engineering - Option III

Raft - An Understandable Distributed Consensus Protocol

Authors:
Howie Benefiel
Patrick Sigourney

Professor:
Dr. Vijay Garg

November 27, 2017

Abstract

Paxos has been the industry-standard protocol for implementing distributed consensus for the past 25 years. Though the basics of single-decree consensus are easily enough grokked, delving into why it works, forming a replicated log or implementing membership changes is fraught with unproven ambiguity. Raft was designed with a singular design goal, understandability. To achieve this goal of optimizing for understandability, the authors employed two techniques, problem decomposition and minimization of state space. For this paper, we examined and implemented their protocol. We found their protocol performed as well as they claimed and it was much easier to fundamentally grok than Paxos.

1 Introduction

The primary focus of distributed consensus is to take an unreliable machine and replicate it so that it becomes a cluster of individually unreliable but collectively reliable machines. The most studied model for this is the replicated state machine model. In this model, each machine has some internal state which can be changed by external stimulus. Perhaps obviously, the state of these state machines should be replicated by all the machines in the cluster. At a high level, the duty of a distributed consensus protocol is to ensure the state stays consistent across all the machines in the cluster.

The core data structure in a practical consensus algorithm, and indeed Raft, is the replicated log. The replicated log contains all the commands that have been run against the cluster. If a machine were to start at the beginning of the log and execute each command, in order, in its log, that machine should arrive at a state consistent with other machines in the cluster. At a more practical level, a distributed consensus protocol will keep these replicated logs consistent across the cluster.

FLP states that an asynchronous distributed system can not tolerate failure and satisfy agreement, validity and termination. [1] So, consensus algorithms generally satisfy:

1. *Safety* They always return a correct result. If a client randomly queries any server for the state of the system, the result should be the same regardless of which machine was queried.

2. *Availability* If a majority of the servers are available, the system, as a whole should be available.

3. *Time Invariance* The system should not depend on the physical timing of events.

4. *Quorum Rate-Limiting* The response to the client should be returned once quorum is reached, not when all machines have responded.

First, they on decomposed the primary problem, distributed consensus, into a couple independent sub-problems. Second, they minimized the state space of the algorithm.

2 Previous Work

3 Algorithm

3.1 Everything below this is just an example

3.2 Basic Operations on the Data Structure

There are two basic operations which can be used on a bag, `Add(item)` and `TryRemoveAny()`. `Add(item)` will add an item anywhere in the data structure, as expected. The `Add(item)` algorithm with detailed comments is shown in Algorithm 1. When adding or removing from the data structure, the threads keep track of the position in the block they can access and the first block in their linked list through the thread-local variables, `threadHead` and `threadBlock`, respectively. This is shown in Fig. 1 as the thread-numbered circles pointing to a position. One thing to note about `Add()` is that items will always only be added to the first block in the linked list and at most 1 thread will be adding to a block.

`TryRemoveAny()` is used to remove any random item from the data structure and return it. This is shown with detailed comments in Algorithm 2. Now the situation may arise where a thread's linked list only contains empty blocks. For this situation, the `TryRemoveAny` operation provides functionality via the `Steal()` method to steal from another thread's block. This is shown in Algorithm 2 at line 4.

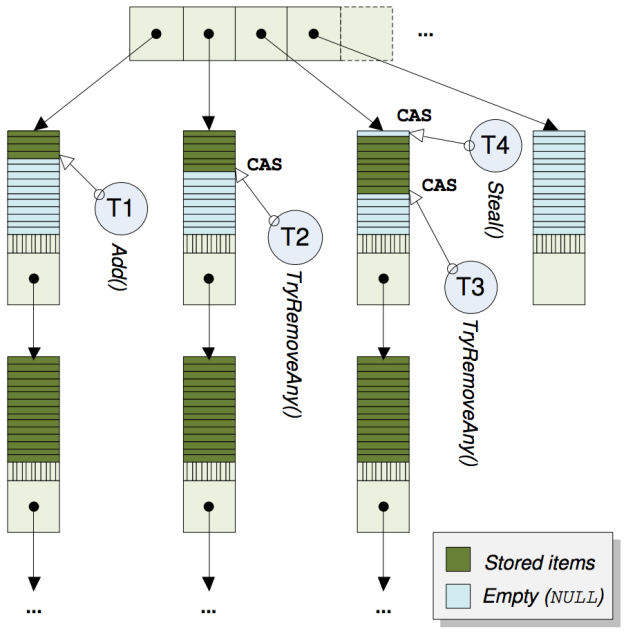

Figure 1: Above is the graphical representation of the basic data structure. The top array is globalHeadBlock. The linked list nodes below the array are the blocks. The green and blue cells are the objects being operated on.

Algorithm 1 `Add(item)` Operation

1: **if** $threadHead == BLOCK\_SIZE$ **then** ▷ the current block is full
2: Create new block and add it at the head of the linked list
3: $threadHead \leftarrow 0$
4: **end if**
5: $threadBlock[threadHead] \leftarrow item$ ▷ Insert value at current threadHead
6: $threadHead \leftarrow threadHead + 1$ ▷ Advance threadHead to next position

Algorithm 2 `TryRemoveAny()` Operation

```
1: loop
2:     if threadHead < 0 then
3:         if head < 0 and next block == null then      ▷ End of linked list
4:             return Steal()                                  ▷ Go to steal
5:         end if
6:         threadBlock ← threadBlock.next          ▷ advance position in list
7:         threadHead ← BLOCK_SIZE                   ▷ last position in block
8:     end if
9:     item ← threadBlock[threadHead]              ▷ Retrieve item from block
10:    if item ≠ null and CAS(threadBlock[threadHead],item,null) then
11:        return item                    ▷ If CAS successful, return the item
12:    else
13:        threadHead ← threadHead − 1          ▷ If not, try next and loop
14:    end if
15: end loop
```

4 Implementation

5 Conclusion

References

[1] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the Association for Computing Machinery*, 32(2):374–382, 1985.