Fault Tolerant Consensus and Co-ordination for Big Data Systems

Jeevesh Juneja

November 2021

1 RAFT

1.1 Introduction

RAFT([5]) is a consensus algorithm, for establishing consensus among the servers of a distributed service(possibly a data warehousing service), regarding the state of each of the systems in the service. The state, and updates thereof are stored in a log, which is to be replicated and maintained by Raft. It is a successor of Paxos([4], [3]); designed for better understandability. It separates key elements of consensus like leader election, log replication and safety.

1.2 Leader Election

During each *term*, a single *leader* is elected from all the servers in the cluster. To be elected as a leader, a server must secure a majority of votes. Each server can vote for only one server during a term. A server converts itself to a candidate if an *election timeout* period elapses without receiving any *AppendEntry RPC* from the leader, or granting any vote to some candidate. The election timeout is reset to a random number[1], each time a server grants someone vote, or receives an AppendEntry RPC from the leader.

If timeout happens, a server converts to a *candidate*, upon which, it increments its current term, votes for itself, resets an *election timer*. If within the time specified by the election timer, it receives votes from a majority of servers, it becomes the new leader(or if it receives AppendEntry RPC from some other leader, it becomes a follower, withdrawing from election). If election timeout elapses, without any of the previous events it starts a new election.

Upon election, the leader sends initial empty AppendEntry RPCs(equivalent to heartbeats in other distributed systems). This heartbeat informs the other servers who is the leader and that a new term has begun.

[1]The gap between two random numbers must be enough for an election to complete. Moreover, a server should only ask for if time elapsed without any contact from leader, is greater than some small integer multiple time b/w heartbeat messages.

1.3 The State

Each server maintains the last term it has seen, in a variable *currentTerm*, on its persistent storage. Additionally it maintains a *votedFor*(the server the current server voted for in the currentTerm), and the *log* of operations paired with the term during which the leader received the operation.

The leader sequences all operations, and enforces the same sequence on all the other servers. After ensuring that an operation has been logged at the correct location in a majority of servers, it commits that operation and also asks lets the other servers that the operation has been committed. A committed operation's receipt is returned to the client which trigerred the operation.

In its volatile memory, each server also maintains a *commitIndex*(the index of last log entry known to be committed), and *lastApplied*(the index of last log entry operation that was carried out by the server). The leaders additionally maintain the *nextIndex*(where the next log entry is to be appended) for each server, and the *matchIndex*(index of highest log entry known to replicated) for each server.

1.4 AppendEntries RPC

The AppendEntry RPC sent by the leader, contains leader's term, along with a list of log entries to store, and other supplementary information like leader's commitIndex, log index(*prevLogIndex*) and term(*prevLogTerm*) for the log preceeding the new ones. Heartbeat messages specify an empty list for log entries and pass *leaderId*, so that the receiving server can re-direct client requests to the leader for the current term.

The receiver, replies false, for a failed AppendEntry, if the term specified in the RPC is less than its currentTerm, or if its log doesn't contain an entry with prevLogTerm at prevLogIndex. Both of this situations mean that the server which sent the AppendEntry crashed while it was master and is delusional that it is still the master.

There can be other conflicts, for example, that log position may already be filled, or there may be some gap between the position from where the new log entry starts and uptil which the follower server has its log filled. To resolve the follower replies with:

- *xTerm*, i.e., term of conflicting entry
- *xIndex*, i.e., the index of first entry with term *xTerm*
- *xLen*, i.e., length of follower's log

Then, the leader will see these three, and figure out from what position does the log of the follower needs to be re-written and send new logs, accordingly. Note here that an important invariant is maintained: **The next operation of a term won't be appended to a log, till all other previous operations**

of the same term have already been appended. There may be some uncommitted entries from previous leader in some server, or missing entries. It is the new leader's job to bring that server up-to-date.

Upon successful acceptance of new log entries in the follower's log, the nextIndex and matchIndex variables for the follower are updated. When a new leader is elected, it tries sending log entries beginning at nextIndex to all the servers whose nextIndex<last log index of the new leader. Conflicts are resolved in same way, and nextIndex, matchIndex updated same way. After doing this, if there exists some log entry of currentTerm, N such that, the log entry is present in a majority of servers(matchIndex[i]$\geq N$ for a majority of servers) we increment commitIndex to this N.

1.5 RequestVote RPC

This RPC is used by candidates to request other servers for votes. The candidate sends term, Id, lastLogIndex and lastLogTerm, to all the servers it can ask for vote. The receiver, replies false, if the term it gets via RPC, is less than its currentTerm. If it hasn't already voted in the current term, and candidate's log is at least as up-to-date as its own, it votes for the candidate and returns *voteGranted* as true.

1.6 Guarantees

The RAFT system guarantees that:

- Only one leader can be elected in each term.
- A leader never overwrites or deletes entries in its log. It only appends.
- If two logs contain an entry with the same index and term, then the logs are identical in all entries upto the given index.
- If a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher terms.
- Only committed entries are applied to state machines of the servers, this means, if a server has applied a log entry at a given index, to its state machine, no other server will ever apply a different log entry for the same index.

Additionally, RAFT implements log compaction by taking snapshots of the server states, to keep the log size manageable.

2 ZooKeeper

2.1 Introduction

ZooKeeper([1]) is a wait-free coordination system for internet-scale systems, built on top of a RAFT like system, Zab([2]). This system is widely used as a

fault tolerant coordination manager, resistant to split brain problem, for various distributed systems. For example, it can be used as the test-and-set service for Fault Tolerant Virtual Machines of [6]).

2.2 Goals and Guarantees

While RAFT ensures full linearizability of all read and write operations, ZooKeeper is more focused on performance. It tries to yield N times performance, with N replicas. As a result, it distributes the read workload over all the replicas(though all the write workload must still pass through the leader). As a result it may sometimes return stale data. But, still it ensures:

- All writes are linearizable.
- FIFO client order is maintained, i.e., all requests by a single client are executed in the same order in which they are sent.

The various replicas are kept in sync with mechanisms similar to RAFT(leader election, logs etc.). It also maintains additional *watch tables*.

2.3 The API

Clients can request to watch some piece of data, and ZooKeeper server, the client is talking to, will log it into its watch table. The client will be notified as soon as the data which is set to be watch is updated by the server it is talking to. Additional version numbers help implement conditional reads and writes which work only if the version number matches the one specified by the client.

Data is stored in an hierarchical name-space, consisting of *z-nodes*. Regular z-nodes are created and deleted manually, while ephemeral z-nodes are created by clients, but they are deleted automatically when the *session* that creates them, terminates, either due to a fault or on completion. A sequential flag ensures that the z-node created has a number appended to its name, and that the number is higher than the number of any other sibling of the z-node. We can use the API to create various co-ordination mechanisms like the Read-Write Lock in Figure 1.

3 CRAQ

3.1 Introduction

The paper([7]), tries to device a replication system that allows higher throughput, without sacrificing consistency. Earlier, we saw how ZooKeeper sacrifices consistency for larger throughput, and raft does vice-versa. Their solution, CRAQ, basically involves chaining together replicas, instead of arranging them in random network topologies, and transmitting operations along chains.

Write Lock

```
1  n = create(l + "/write-", EPHEMERAL|SEQUENTIAL)
2  C = getChildren(l, false)
3  if n is lowest znode in C, exit
4  p = znode in C ordered just before n
5  if exists(p, true) wait for event
6  goto 2
```

Read Lock

```
1  n = create(l + "/read-", EPHEMERAL|SEQUENTIAL)
2  C = getChildren(l, false)
3  if no write znodes lower than n in C, exit
4  p = write znode in C ordered just before n
5  if exists(p, true) wait for event
6  goto 3
```

Figure 1: A Fault Tolerant Read-Write Lock without Herd effect implemented using the basic data storing and retrieval primitives provided by ZooKeeper. The true & false in second arguments of various functions, corresponds to setting a watch on the requested content.

3.2 Working

Figure 2 shows how the machines are chained and how reads and writes happen. All write requests are first sent to the head, which logs it, and propagates it to the next replica, all the way to the tail. The tail logs and applies the write and returns success, which travels upwards to the chain. As each replica gets success it applies the write. Only when the head has applied the write, it will send success to the client that requested write request.

Read requests are handled at the tail server only. Note that other replicas may contain information that is not yet confirmed to reach all the replicas, and hence may be dropped. This is better than RAFT as read/write workload is split between two nodes(HEAD and TAIL) instead of just one leader in RAFT.

3.3 Failure and Recovery

In case, the head fails, we can make the next machine in the system as the head. In case the tail fails, we can make the node just before it as the new tail and continue without any problem.

But we will have a problem if connection to the head or tail or any other machine in between, or any connection among them breaks. We will have a split brain problem. To avoid this, CRAQ uses ZooKeeper to maintain its configuration, which consists of various data, like which of the machines are live, which of the machines can talk to others and so on. Watches are maintained for events corresponding to failures of any connections or machines, and upon

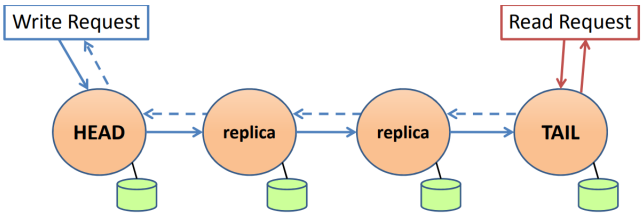

Figure 2: Writes are propagated along the chain, and reads are handled by the tail.

failure, as each machine can view the same ZooKeeper state, they can all find the optimal way to re-make the chain(which will be same for all) and act accordingly.

References

[1] Patrick Hunt et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *USENIX annual technical conference*. Vol. 8. 9. 2010.

[2] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. "Zab: High-Performance Broadcast for Primary-Backup Systems". In: *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks*. DSN '11. IEEE Computer Society, 2011, pp. 245–256.

[3] Leslie Lamport et al. "Paxos made simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[4] Leslie Lamport. "The part-time parliament". In: *Concurrency: the Works of Leslie Lamport*. 2019, pp. 277–317.

[5] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm". In: *2014 {USENIX} Annual Technical Conference*. 2014, pp. 305–319.

[6] Daniel J Scales, Mike Nelson, and Ganesh Venkitachalam. "The design of a practical system for fault-tolerant virtual machines". In: *ACM SIGOPS Operating Systems Review* 44.4 (2010), pp. 30–39.

[7] Jeff Terrace and Michael J Freedman. "Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads." In: *USENIX Annual Technical Conference*. June. San Diego, CA. 2009, pp. 1–16.