Programming Distributed Systems

10 Total-order broadcast with Raft

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Summer Term 2019

Classical Consensus Problem

- Each process p_i has an initial value v_i $(propose(v_i))$.
- All processors have to agree on common value v that is the initial value of some p_i $(decide(v))$.

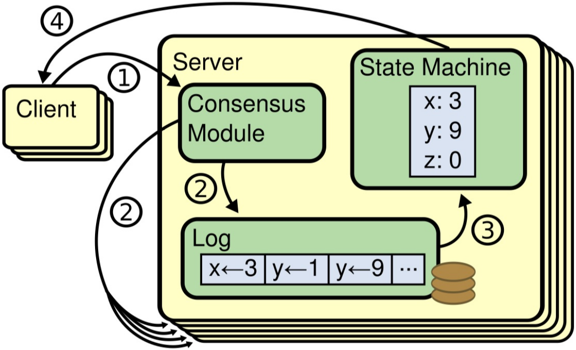
Properties of Consensus:

- *Uniform Agreement*: Every correct process must decide on the same value.
- *Integrity*: Every correct process decides at most one value, and if it decides some value, then it must have been proposed by some process.
- *Termination*: All processes eventually reach a decision.
- *Validity*: If all correct processes propose the same value v, then all correct processes decide v.

Challenges

- Fault-tolerance rules out “dictator” solution (i.e. one node makes the decision).
- Any consensus algorithm requires at least a majority of nodes to not crash to ensure termination. ⇒ Quorum!
- Typically, nodes decide on a *sequence of values*. ⇒ Total-order broadcast!

Motivation: Replicated state-machine via Replicated Log



All figures in these slides are taken from [4].

- Replicated log $\Rightarrow$ State-machine replication
  - Each server stores a log containing a sequence of state-machine commands.
  - All servers execute the same commands in the same order.
  - Once one of the state machines finishes execution, the result is returned to the client.
- Consensus module ensures correct log replication
  - Receives commands from clients and adds them to the log
  - Communicates with consensus modules on other servers such that every log eventually contains same commands in same order
- *Failure model:* Fail-stop (i.e. nodes may recover and rejoin), delayed/lost messages

Practical aspects

- **Safety**: Never return in incorrect result despite network delays, partitions, duplication, loss, reordering of messages
- **Availability**: Majority of servers is sufficient
  - Typical setup: 5 servers where 2 servers can fail
- **Performance**: (Minority of) slow servers should not impact the overall system performance

Approaches to consensus

- *Leader-less (symmetric)*
    - All servers are operating equally
    - Clients can contact any server
- *Leader-based (asymmetric)*
    - One server (called leader) is in charge
    - Other server follow the leader's decisions
    - Clients interact with the leader, i.e. all requests are forwarded to the leader
    - If leader crashes, a new leader needs to be (s)elected
    - Quorum for choosing leader in next epoch (i.e. until the leader is suspected to have crashed)
    - Then, overlapping quorum decides on proposed value $\Rightarrow$ Only accepted if no node has knowledge about higher epoch number

Classic approaches I

- Paxos[2]
    - The original consensus algorithm for reaching agreement on a **single value**
    - Leader-based
    - Two-phase process: Promise and Commit
        - Clients have to wait 2 RTTs
    - Majority agreement: The system works as long as a majority of nodes are up
    - Monotonically increasing version numbers
    - Guarantees safety, but not liveness

Classic approaches II

- Multi-Paxos
    - Extends Paxos for a stream of a agreement problems (i.e. total-order broadcast)
    - The promise (Phase 1) is not specific to the request and can be done before the request arrives and can be reused
    - Client only has to wait 1 RTT
- View-stamped replication (revisited)[3]
    - Variant of SMR + Multi-Paxos
    - Round-robin leader election
    - Dynamic membership

The Problem with Paxos

> *[...] I got tired of everyone saying how difficult it was to understand the Paxos algorithm.[...] The current version is 13 pages long, and contains no formula more complicated than n1 $>$ n2.* [1]

Still significant gaps between the description of the Paxos algorithm and the needs or a real-world system

- Disk failure and corruption
- Limited storage capacity
- Effective handling of read-only requests
- Dynamic membership and reconfiguration

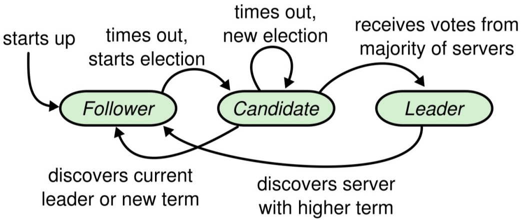
In Search of an Understandable Consensus Algorithm: Raft[4]

- Yet another variant of SMR with Multi-Paxos
- Became very popular because of its understandable description

In essence

- Strong leadership with all other nodes being passive
- Dynamic membership and log compaction

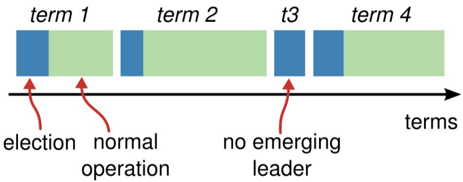
Server Roles



At any time, a server is either

- **Leader**: Handles client interactions and log replication
- **Follower**: Passively follows the orders of the leader
- **Candidate**: Aspirant in leader election
- During normal operation: 1 leader, N-1 followers

Terms = Epoch



- Time is divided into **terms**
- Each terms begins with an election
- After a successful election, a single leader operates till the end of the term
- Transitions between terms are observed on servers at different times

Leader election

- Servers start as followers
  - Followers expect to receive messages from leaders or candidates
  - Leaders must send **heartbeats** to maintain authority
- If *electionTimeout* elapses with no message, follower assumes that leader has crashed
- Follower starts new election
  - Increment current term (locally)
  - Change to candidate state
  - Vote for self
  - Send *RequestVote* message to all other servers
- Possible outcomes
  1. Receive votes from majority of servers ⇒ Become new leader
  2. Receive message from valid leader ⇒ Step down and become follower
  3. No majority (*electionTimeout* elapses) ⇒ Increment term and start new election

Properties of Leader Election

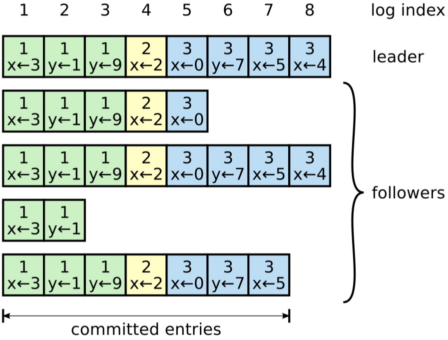
Safety: At most one leader per term

- Each server gives only one vote per term, namely to the first *RequestVote* message it receives (persist on disk)
- At most one server can accumulate majorities in same term

Liveness: Some candidate must eventually win

- Choose election timeouts randomly at every server
- One server usually times out and wins election before others consider elections
- Works well if time out is (much) larger than broadcast time

Log replication



log index	1	2	3	4	5	6	7	8
leader	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0	3 y←7	3 x←5	3 x←4
followers	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0			
	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0	3 y←7	3 x←5	3 x←4
	1 x←3	1 y←1						
	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0	3 y←7	3 x←5	

- Log entry: index + term + command
- Stored durably on disk to survive crashes
- Entry is **committed** if it is known to be stored on majority of servers

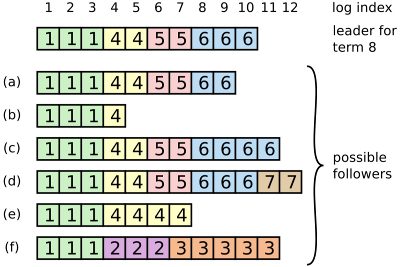
Operation (when no faults occur)

1. Client sends command to leader
2. Leader appends command to its own log
3. Leader sends *AppendEntry* to followers
4. Once new entry is committed, i.e. majority of servers acknowledge storing

   - Leader executes command and returns result to client
   - Leader notifies followers about committed entries in subsequent *AppendEntries*
   - Followers pass committed commands to their state machines

⇒ 1 RTT to any majority of servers

Log consistency



At beginning of new leader's term:

- Followers might miss entries
- Followers may have additional, uncommitted entries
- Both

Goal

Make follower's log identical to leader's log – without changing the leader log!

Safety Requirement

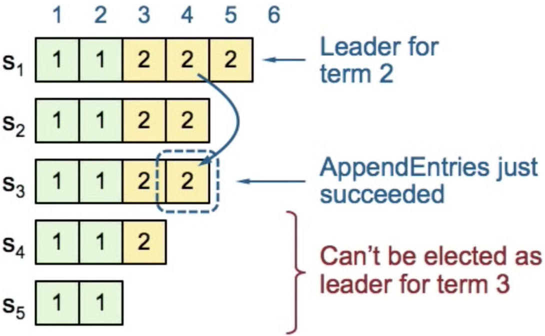
Once a log entry has been applied to a state machine, no other state machine must apply a different value for this log entry.

- If a leader has decided that a log entry is committed, this entry will be present in the logs of all future leaders.
    - Restriction on commit
    - Restriction on leader election

Restriction on leader election

- Candidates can't tell which entries are committed
- Choose candidate whose log is most likely to contain all committed entries
  - Candidates include log info in *RequestVote*, i.e. index + term of last log entry
  - Server denies a candidate its vote if the server's log contains more information; i.e. last term in server is larger than last term in candidate, or, if they are equal, server's log contains more entries than candidate's log

Example: Leader decides entry in current term is committed



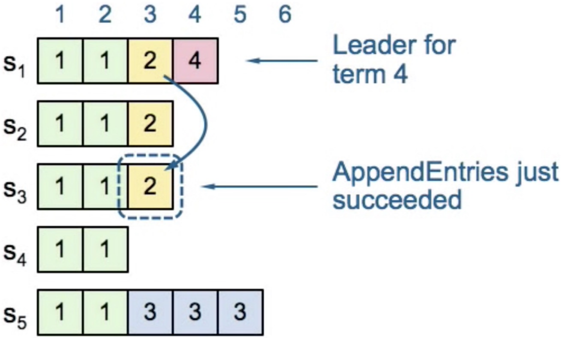
Leader for term 2

AppendEntries just succeeded

Can't be elected as leader for term 3

Leader for term 3 must contain entry 4!

Example: Leader is trying fo finish committing entry from an earlier term

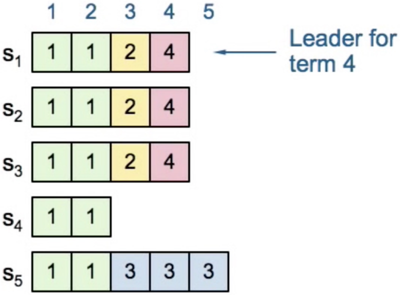


Entry 3 not safely committed!

- If elected, s_5 will overwrite entry 3 on s_1, s_2, s_3

Requirement for commitment

- Entry must be stored on a majority of servers
- At least one new entry from leader's term must also be stored on majority of servers.

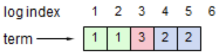


- Once entry 4 is committed, s_5 cannot be elected leader for term 5

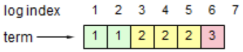
Question 1

Considering each of these logs in isolation, could such a log configuration occur in a proper implementation of Raft?

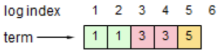
a.



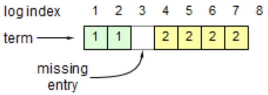
b.



c.

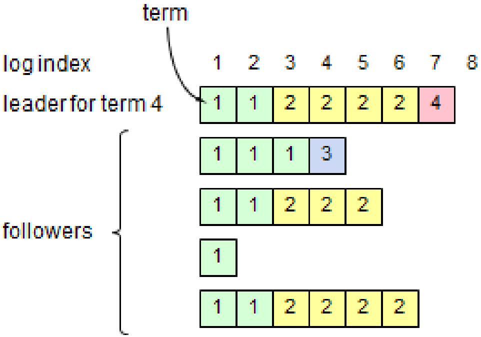


d.



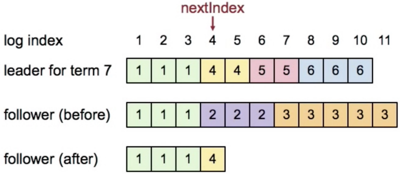
Question 2

Which log entries may safely be applied to state machines?



Repairing Follower Logs

- When appending new entry, send index+term of entry preceding the new one
- Follower must contain matching entry; otherwise, it rejects request
- Leader keeps *nextIndex* for each follower
    - Index of next log entry to send to that follower
    - Initialized to 1 + leader's last index
    - When *AppendEntry* consistency check fails, decrement *nextIndex* and retry
- When follower overwrites inconsistent entry, it deletes all subsequent entries



log index	1	2	3	4	5	6	7	8	9	10	11
leader for term 7	1	1	1	4	4	5	5	6	6	6	
follower (before)	1	1	1	2	2	2	3	3	3	3	3
follower (after)	1	1	1	4							

When old leaders recover

- E.g. temporarily disconnected from network
- How does a leader realize that it has been replaced?
  - Every request contains term of sender
  - If sender's term is older, request is rejected; sender reverts to follower and updates its term
  - If receiver's term is older, it reverts to follower, updates its term und process then the message
- Why does it work?
  - Election updates terms of majority of servers
  - Old leader cannot commit new log entries

Guarantees

Election Safety: At most one leader can be elected in a given term.

Leader Append-Only: A leader never overwrites or deletes entries in its log; it only appends new entries.

Log Matching: If two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index.

Leader Completeness: 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-numbered terms.

State-Machine Safety: If a server has applied a log entry at a given index to its state machine, then no other server will every apply a different log entry for the same index.

Beyond the Basics

In the paper, there is more information regarding

- Client interaction
- Cluster membership changes
- Log compaction
- Performance evaluation

Question: Why does Raft not circumvent the FLP theorem?

Consensus Algorithms in Real-World Systems

- Paxos made live - or: How Google uses Paxos
  - Chubby: Distributed coordination service built using Multi-Paxos and MSR
- Spanner: Paxos-based replication for hundreds of data centers; uses hardware-assisted clock synchronization for timeouts
- Apache Zookeeper: Distributed coordination service using Paxos
  - Typically used as naming service, configuration management, synchronization, priority queue, etc.
- etcd: Distributed KV store using Raft
  - Used by many companies / products (e.g. Kubernetes, Huawei)
- RethinkDB: JSON Database for realtime apps
  - Storing of cluster metadata such as information about primary

Summary

- Consensus algorithms are an important building block in many applications
- Replicated log via total-order broadcast
- Raft as alternative to classical Paxos
    - Leader election
    - Log consistency
    - Commit

Further reading I

[1] Leslie Lamport. “Paxos Made Simple”. In: *SIGACT News* 32.4 (Dez. 2001), S. 51–58. ISSN: 0163-5700. DOI: 10.1145/568425.568433. URL: http://research.microsoft.com/users/lamport/pubs/paxos-simple.pdf.

[2] Leslie Lamport. “The Part-Time Parliament”. In: *ACM Trans. Comput. Syst.* 16.2 (1998), S. 133–169. DOI: 10.1145/279227.279229. URL: http://doi.acm.org/10.1145/279227.279229.

[3] Barbara Liskov und James Cowling. *Viewstamped Replication Revisited (Technical Report)*. MIT-CSAIL-TR-2012-021. MIT, Juli 2012.

Further reading II

[4] Diego Ongaro und John K. Ousterhout. “In Search of an Understandable Consensus Algorithm”. In: *2014 USENIX Annual Technical Conference, USENIX ATC ’14, Philadelphia, PA, USA, June 19-20, 2014.* Hrsg. von Garth Gibson und Nickolai Zeldovich. USENIX Association, 2014, S. 305–319. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.