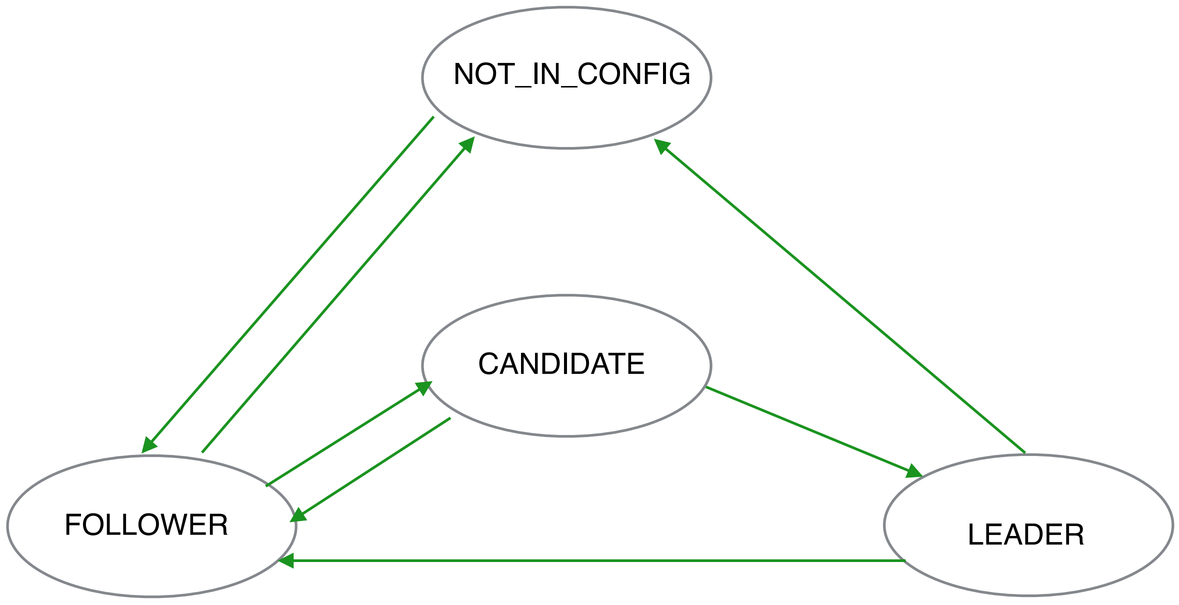
RAFT : A Distributed Consensus Algorithm

This implementation has an additional state NOT_IN_CONFIG to simplify the configuration change and it also helps to at the start-up time. Following diagram explains the state transition of this modified RAFT implementation.

It always starts with NOT_IN_CONFIG state and transition to other states based on inputs from user.



When user sets the raft configuration, all the nodes that are in configuration will transition from NOT_IN_CONFIG to FOLLOWER and at any time user changes the configuration, all nodes (FOLLOWER, LEADER) that are not part of configuration will transition to NOT_IN_CONFIG. A node in NOT_IN_COFIG state will never timeout and remain in that state until it receives a configuration change message.

FOLLOWER will transition to CANDIDATE state when it receives an election timeout.

CANDIDATE node will start the leader election process, it will become LEADER if it receives enough (QUORUM) votes. If it does not receive enough votes and it receives another election timeout, it will continue to remain in CANDIDATE state and start another leader election process. While a node is in CANDIDATE state, it may receive a ping from another node that has been elected as leader then it will transition to FOLLOWER state.

LEADER can transition to FOLLOWER state if there is another LEADER with higher term.

This has been implemented a 3 GEN_SERVERs.

ER_RAFT_SERVER is the main server that implements all the RAFT logic. LEADER uses ER_PEER_SERVER to communicate will all the peer FOLLWERs. LEADER and FOLLOWER both use RL_SERVER to maintain replicated logs.

RAFT configuration (sys.config)

The following configuration parameter are used to configure the RAFT operation:

All the timeout timings are given in milliseconds.

election_timeout_min and **election_timeout_max** are used to generate a random election timeout.

heartbeat_timeout is used by leader to send a ping (noop) message to all the followers if there is no user activity to maintain its authority.

log_request_timeout is used by ER_PEER_SERVER to timeout a multi_call request to FOLLOWERs.

log_entry_operation_api, log_entry_trailer_magic is used to customize the log entry format when it is saved into replicated logs. **log_entry_operation_api** implements a behavior **rl_entry_operation**, so if you wants to change the log entry format, you can write a new module implementing the same behavior and change the sys.config to reflect the new module name.

log_file_header_version is a number that gets written to every log file as file header, this can be disabled by returning 0 for function **file_header_byte_size** in **rl_entry_operation.**

state_machine_api has the name of module that is our API for our state machine. The module must implement **er_state_machine** behavior to integrate a state machine to RAFT. Additionally, it is required that state machine has idempotent property.

data_dir is the path name where we save replicated log files and metadata (vote).

file_replicated_log is used to create replicated log file name.

file_metadata is used to create metadata file name.

file_config is not being used currently, it is meant for file where we will save configuration data but with modified RAFT implementation we do not need to save configuration data.

optimistic_mode is used change the way RAFT will operate. If this parameter is set to true (optimistic) then any FOLLOWER node that is not current will be made current only when LEADER can not meet quorum requirement. This allows RAFT to work a little faster if at least majority of the nodes are current. In non-optimistic mode, every active FOLLOWER node will always current.

debug_mode is used to enable/disable outputs on the console. if **debug_mode** is set to true then we add an event_handler to event manager that enables all the debugging outputs.

sup_restart_intensity, sup_restart_period and **sup_child_shutdown** supervisor related parameters, please refer to Erlang - supervisor documents for more detail.

log_retention_size_min and **log_retention_size_max** are used to compacting the replicated log file.

Replicated logs are maintained automatically with correctly formed log entries in case of data corruption. Replicated logs are truncated at the if any log entry data is corrupted at the right location and all the uncorrupted log entries are retained. This repair happens when data is read from replicated log at the start of the system. So all the raft servers will have correct status of log entries. If a raft server has a fewer less entries because of data corruption as compared to other raft servers, it will not get elected as leader but its log will be made current eventually by the leader.

If replicated log size exceeds the max log retention size, it is being automatically compacted and saved. It retains all the log entries that have not been applied to state machine and a few more as given in min log retention size. When a log file is compacted it is written in a new temp file. After it has been written successfully in a temp file, the original log file is deleted and temp file is renamed to original log file name. This is not an atomic operation so if there is any failure during this operation, system recovers for such failure and brings the replicated logs to stable state.

erlang_raft.erl is the user API to use this system.