thuduyen07 / raft

Code Pull requests Actions Projects Wiki Security Insights

raft / part1 / raft.go

thuduyen07 add doc for raft 2 minutes ago

405 lines (354 loc) · 10.2 KB

```go
// Core Raft implementation - Consensus Module.
//
// Eli Bendersky [https://eli.thegreenplace.net]
// This code is in the public domain.
package raft

import (
	"fmt"
	"log"
	"math/rand"
	"os"
	"sync"
	"time"
)

const DebugCM = 1

/*
	cấu trúc log gồm:
	- một mục Command (lệnh được gửi)
	- Term (nhiệm kỳ hiện hành)
*/
type LogEntry struct {
	Command interface{}
	Term    int
}

/*
	Trạng thái của một Consensus Module
*/
type CMState int

const (
	Follower CMState = iota
	Candidate
	Leader
	Dead
)

func (s CMState) String() string {
	switch s {
	case Follower:
		return "Follower"
	case Candidate:
		return "Candidate"
	case Leader:
		return "Leader"
	case Dead:
		return "Dead"
	default:
		panic("unreachable")
	}
}

// ConsensusModule (CM) implements a single node of Raft consensus.
type ConsensusModule struct {
	// mu protects concurrent access to a CM.
	mu sync.Mutex

	// id is the server ID of this CM.
	id int

	// peerIds lists the IDs of our peers in the cluster.
	peerIds []int

	// server is the server containing this CM. It's used to issue RPC calls
	// to peers.
	server *Server

	// Persistent Raft state on all servers
	currentTerm int
	votedFor    int
	log         []LogEntry

	// Volatile Raft state on all servers
	state              CMState
	electionResetEvent time.Time
}

// NewConsensusModule creates a new CM with the given ID, list of peer IDs and
// server. The ready channel signals the CM that all peers are connected and
// it's safe to start its state machine.
```

```go
func NewConsensusModule(id int, peerIds []int, server *Server, ready <-chan interface{}) *Con
	cm := new(ConsensusModule)
	cm.id = id
	cm.peerIds = peerIds
	cm.server = server
	cm.state = Follower
	cm.votedFor = -1

	go func() {
		// The CM is quiescent until ready is signaled; then, it starts a countdown
		// for leader election.
		<-ready
		cm.mu.Lock()
		cm.electionResetEvent = time.Now()
		cm.mu.Unlock()
		cm.runElectionTimer()
	}()

	return cm
}

// Report reports the state of this CM.
func (cm *ConsensusModule) Report() (id int, term int, isLeader bool) {
	cm.mu.Lock()
	defer cm.mu.Unlock()
	return cm.id, cm.currentTerm, cm.state == Leader
}

// Stop stops this CM, cleaning up its state. This method returns quickly, but
// it may take a bit of time (up to ~election timeout) for all goroutines to
// exit.
func (cm *ConsensusModule) Stop() {
	cm.mu.Lock()
	defer cm.mu.Unlock()
	cm.state = Dead
	cm.dlog("becomes Dead")
}

// dlog logs a debugging message if DebugCM > 0.
func (cm *ConsensusModule) dlog(format string, args ...interface{}) {
	if DebugCM > 0 {
		format = fmt.Sprintf("[%d] ", cm.id) + format
		log.Printf(format, args...)
	}
}

// See figure 2 in the paper.
type RequestVoteArgs struct {
	Term         int
	CandidateId  int
	LastLogIndex int
	LastLogTerm  int
}

type RequestVoteReply struct {
	Term        int
	VoteGranted bool
}

// RequestVote RPC.
func (cm *ConsensusModule) RequestVote(args RequestVoteArgs, reply *RequestVoteReply) error {
	cm.mu.Lock()
	defer cm.mu.Unlock()
	if cm.state == Dead {
		return nil
	}
	cm.dlog("RequestVote: %+v [currentTerm=%d, votedFor=%d]", args, cm.currentTerm, cm.vo

	if args.Term > cm.currentTerm {
		cm.dlog("... term out of date in RequestVote")
		cm.becomeFollower(args.Term)
	}

	if cm.currentTerm == args.Term &&
		(cm.votedFor == -1 || cm.votedFor == args.CandidateId) {
		reply.VoteGranted = true
		cm.votedFor = args.CandidateId
		cm.electionResetEvent = time.Now()
	} else {
		reply.VoteGranted = false
	}
	reply.Term = cm.currentTerm
	cm.dlog("... RequestVote reply: %+v", reply)
	return nil
}

// See figure 2 in the paper.
type AppendEntriesArgs struct {
	Term     int
	LeaderId int

	PrevLogIndex int
	PrevLogTerm  int
	Entries      []LogEntry
	LeaderCommit int
}

type AppendEntriesReply struct {
	Term    int
	Success bool
}

func (cm *ConsensusModule) AppendEntries(args AppendEntriesArgs, reply *AppendEntriesReply) e
	cm.mu.Lock()
	defer cm.mu.Unlock()
	if cm.state == Dead {
		return nil
	}
	cm.dlog("AppendEntries: %+v", args)

	if args.Term > cm.currentTerm {
		cm.dlog("... term out of date in AppendEntries")
		cm.becomeFollower(args.Term)
	}

	reply.Success = false
	if args.Term == cm.currentTerm {
		if cm.state != Follower {
			cm.becomeFollower(args.Term)
		}
		cm.electionResetEvent = time.Now()
		reply.Success = true
	}

	reply.Term = cm.currentTerm
	cm.dlog("AppendEntries reply: %+v", *reply)
	return nil
}

/*
	Phương thức của đối tượng ConsensusModule dùng để triển khai bộ đếm thời gian
	cho việc bắt đầu một cuộc bầu cử mới. Ở đây đang đặt là 300ms
*/
// electionTimeout generates a pseudo-random election timeout duration.
func (cm *ConsensusModule) electionTimeout() time.Duration {
	// If RAFT_FORCE_MORE_REELECTION is set, stress-test by deliberately
	// generating a hard-coded number very often. This will create collisions
	// between different servers and force more re-elections.
	if len(os.Getenv("RAFT_FORCE_MORE_REELECTION")) > 0 && rand.Intn(3) == 0 {
		return time.Duration(300) * time.Millisecond
	} else {
		return time.Duration(300+rand.Intn(300)) * time.Millisecond
	}
}

/*
	Hàm này tạo ra một election timer để đếm ngược đến thời gian bầu cử
```

```
		và chuyển trạng thái của CM thành Candidate nếu cần.
	*/
	// runElectionTimer implements an election timer. It should be launched whenever
	// we want to start a timer towards becoming a candidate in a new election.
	//
	// This function is blocking and should be launched in a separate goroutine;
	// it's designed to work for a single (one-shot) election timer, as it exits
	// whenever the CM state changes from follower/candidate or the term changes.
	func (cm *ConsensusModule) runElectionTimer() {
		timeoutDuration := cm.electionTimeout()
		cm.mu.Lock()
		termStarted := cm.currentTerm
```

raft / part1 / raft.go ↑ Top

Code Blame Raw

```
	func (cm *ConsensusModule) runElectionTimer() {
```

```
		*/
		// This loops until either:
		// - we discover the election timer is no longer needed, or
		// - the election timer expires and this CM becomes a candidate
		// In a follower, this typically keeps running in the background for the
		// duration of the CM's lifetime.
		ticker := time.NewTicker(100 * time.Millisecond)
		defer ticker.Stop()
		for {
			<-ticker.C

			cm.mu.Lock()
			if cm.state != Candidate && cm.state != Follower {
				cm.dlog("in election timer state=%s, bailing out", cm.state)
				cm.mu.Unlock()
				return
			}

			if termStarted != cm.currentTerm {
				cm.dlog("in election timer term changed from %d to %d, bailing out",
				cm.mu.Unlock()
				return
			}

			// Start an election if we haven't heard from a leader or haven't voted for
			// someone for the duration of the timeout.
			if elapsed := time.Since(cm.electionResetEvent); elapsed >= timeoutDuration {
				cm.startElection()
				cm.mu.Unlock()
				return
			}
			cm.mu.Unlock()
		}
	}

// startElection starts a new election with this CM as a candidate.
// Expects cm.mu to be locked.
func (cm *ConsensusModule) startElection() {
	cm.state = Candidate
	cm.currentTerm += 1
	savedCurrentTerm := cm.currentTerm
	cm.electionResetEvent = time.Now()
	cm.votedFor = cm.id
	cm.dlog("becomes Candidate (currentTerm=%d); log=%v", savedCurrentTerm, cm.log)

	votesReceived := 1

	// Send RequestVote RPCs to all other servers concurrently.
	for _, peerId := range cm.peerIds {
		go func(peerId int) {
			args := RequestVoteArgs{
				Term:        savedCurrentTerm,
				CandidateId: cm.id,
			}
			var reply RequestVoteReply

			cm.dlog("sending RequestVote to %d: %+v", peerId, args)
			if err := cm.server.Call(peerId, "ConsensusModule.RequestVote", args,
				cm.mu.Lock()
				defer cm.mu.Unlock()
				cm.dlog("received RequestVoteReply %+v", reply)

				if cm.state != Candidate {
					cm.dlog("while waiting for reply, state = %v", cm.sta
					return
				}

				if reply.Term > savedCurrentTerm {
					cm.dlog("term out of date in RequestVoteReply")
					cm.becomeFollower(reply.Term)
					return
				} else if reply.Term == savedCurrentTerm {
					if reply.VoteGranted {
						votesReceived += 1
						if votesReceived*2 > len(cm.peerIds)+1 {
							// Won the election!
							cm.dlog("wins election with %d votes"
							cm.startLeader()
							return
						}
```

```
                                        }
                                }
                        }
                }(peerId)
        }

        // Run another election timer, in case this election is not successful.
        go cm.runElectionTimer()
}

// becomeFollower makes cm a follower and resets its state.
// Expects cm.mu to be locked.
func (cm *ConsensusModule) becomeFollower(term int) {
        cm.dlog("becomes Follower with term=%d; log=%v", term, cm.log)
        cm.state = Follower
        cm.currentTerm = term
        cm.votedFor = -1
        cm.electionResetEvent = time.Now()

        go cm.runElectionTimer()
}

// startLeader switches cm into a leader state and begins process of heartbeats.
// Expects cm.mu to be locked.
func (cm *ConsensusModule) startLeader() {
        cm.state = Leader
        cm.dlog("becomes Leader; term=%d, log=%v", cm.currentTerm, cm.log)

        go func() {
                ticker := time.NewTicker(3000 * time.Millisecond)
                defer ticker.Stop()

                // Send periodic heartbeats, as long as still leader.
                for {
                        cm.leaderSendHeartbeats()
                        <-ticker.C

                        cm.mu.Lock()
                        if cm.state != Leader {
                                cm.mu.Unlock()
                                return
                        }
                        cm.mu.Unlock()
                }
        }()
}

// leaderSendHeartbeats sends a round of heartbeats to all peers, collects their
// replies and adjusts cm's state.
```

```go
func (cm *ConsensusModule) leaderSendHeartbeats() {
	cm.mu.Lock()
	if cm.state != Leader {
		cm.mu.Unlock()
		return
	}
	savedCurrentTerm := cm.currentTerm
	cm.mu.Unlock()

	for _, peerId := range cm.peerIds {
		args := AppendEntriesArgs{
			Term:     savedCurrentTerm,
			LeaderId: cm.id,
		}
		go func(peerId int) {
			cm.dlog("sending AppendEntries to %v: ni=%d, args=%+v", peerId, 0, ar
			var reply AppendEntriesReply
			if err := cm.server.Call(peerId, "ConsensusModule.AppendEntries", arg
				cm.mu.Lock()
				defer cm.mu.Unlock()
				if reply.Term > savedCurrentTerm {
					cm.dlog("term out of date in heartbeat reply")
					cm.becomeFollower(reply.Term)
					return
				}
			}
		}(peerId)
	}
}
```