thuduyen07 / raft

Code Pull requests Actions Projects Wiki Security Insights

raft / part1 / server.go

eliben Backport server and TestNoCommitWithNoQuorum passes test fi... 3 years ago

200 lines (177 loc) · 4.79 KB

```go
// Server container for a Raft Consensus Module. Exposes Raft to the network
// and enables RPCs between Raft peers.
//
// Eli Bendersky [https://eli.thegreenplace.net]
// This code is in the public domain.
package raft

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

// Server wraps a raft.ConsensusModule along with a rpc.Server that exposes its
// methods as RPC endpoints. It also manages the peers of the Raft server. The
// main goal of this type is to simplify the code of raft.Server for
// presentation purposes. raft.ConsensusModule has a *Server to do its peer
// communication and doesn't have to worry about the specifics of running an
// RPC server.
type Server struct {
	mu sync.Mutex

	serverId int
	peerIds  []int

	cm       *ConsensusModule
	rpcProxy *RPCProxy

	rpcServer *rpc.Server
	listener  net.Listener

	peerClients map[int]*rpc.Client

	ready <-chan interface{}
	quit  chan interface{}
	wg    sync.WaitGroup
}

func NewServer(serverId int, peerIds []int, ready <-chan interface{}) *Server {
	s := new(Server)
	s.serverId = serverId
	s.peerIds = peerIds
	s.peerClients = make(map[int]*rpc.Client)
	s.ready = ready
	s.quit = make(chan interface{})
	return s
}

func (s *Server) Serve() {
	s.mu.Lock()
	s.cm = NewConsensusModule(s.serverId, s.peerIds, s, s.ready)

	// Create a new RPC server and register a RPCProxy that forwards all methods
	// to n.cm
	s.rpcServer = rpc.NewServer()
	s.rpcProxy = &RPCProxy{cm: s.cm}
	s.rpcServer.RegisterName("ConsensusModule", s.rpcProxy)

	var err error
	s.listener, err = net.Listen("tcp", ":0")
	if err != nil {
		log.Fatal(err)
	}
	log.Printf("[%v] listening at %s", s.serverId, s.listener.Addr())
	s.mu.Unlock()

	s.wg.Add(1)
	go func() {
		defer s.wg.Done()

		for {
			conn, err := s.listener.Accept()
			if err != nil {
				select {
				case <-s.quit:
					return
				default:
```

```
                                        log.Fatal("accept error:", err)
                                }
                        }
                        s.wg.Add(1)
                        go func() {
                                s.rpcServer.ServeConn(conn)
                                s.wg.Done()
                        }()
                }
        }()
}

// DisconnectAll closes all the client connections to peers for this server.
func (s *Server) DisconnectAll() {
        s.mu.Lock()
        defer s.mu.Unlock()
        for id := range s.peerClients {
                if s.peerClients[id] != nil {
                        s.peerClients[id].Close()
                        s.peerClients[id] = nil
                }
        }
}

// Shutdown closes the server and waits for it to shut down properly.
func (s *Server) Shutdown() {
        s.cm.Stop()
        close(s.quit)
        s.listener.Close()
        s.wg.Wait()
}

func (s *Server) GetListenAddr() net.Addr {
        s.mu.Lock()
        defer s.mu.Unlock()
        return s.listener.Addr()
}

func (s *Server) ConnectToPeer(peerId int, addr net.Addr) error {
        s.mu.Lock()
        defer s.mu.Unlock()
        if s.peerClients[peerId] == nil {
                client, err := rpc.Dial(addr.Network(), addr.String())
                if err != nil {
                        return err
                }
                s.peerClients[peerId] = client
        }
        return nil
```

```
}

// DisconnectPeer disconnects this server from the peer identified by peerId.
func (s *Server) DisconnectPeer(peerId int) error {
```

raft / part1 / server.go

Top

Code Blame

Raw

```
			return err
		}
		return nil
}

func (s *Server) Call(id int, serviceMethod string, args interface{}, reply interface{}) erro
		s.mu.Lock()
		peer := s.peerClients[id]
		s.mu.Unlock()

		// If this is called after shutdown (where client.Close is called), it will
		// return an error.
		if peer == nil {
			return fmt.Errorf("call client %d after it's closed", id)
		} else {
			return peer.Call(serviceMethod, args, reply)
		}
}

// RPCProxy is a trivial pass-thru proxy type for ConsensusModule's RPC methods.
// It's useful for:
// - Simulating a small delay in RPC transmission.
// - Avoiding running into https://github.com/golang/go/issues/19957
// - Simulating possible unreliable connections by delaying some messages
//   significantly and dropping others when RAFT_UNRELIABLE_RPC is set.
type RPCProxy struct {
		cm *ConsensusModule
}

func (rpp *RPCProxy) RequestVote(args RequestVoteArgs, reply *RequestVoteReply) error {
		if len(os.Getenv("RAFT_UNRELIABLE_RPC")) > 0 {
			dice := rand.Intn(10)
			if dice == 9 {
				rpp.cm.dlog("drop RequestVote")
				return fmt.Errorf("RPC failed")
			} else if dice == 8 {
				rpp.cm.dlog("delay RequestVote")
				time.Sleep(75 * time.Millisecond)
			}
		} else {
			time.Sleep(time.Duration(1+rand.Intn(5)) * time.Millisecond)
		}
		return rpp.cm.RequestVote(args, reply)
	}

	func (rpp *RPCProxy) AppendEntries(args AppendEntriesArgs, reply *AppendEntriesReply) error {
		if len(os.Getenv("RAFT_UNRELIABLE_RPC")) > 0 {
			dice := rand.Intn(10)
			if dice == 9 {
				rpp.cm.dlog("drop AppendEntries")
				return fmt.Errorf("RPC failed")
			} else if dice == 8 {
				rpp.cm.dlog("delay AppendEntries")
				time.Sleep(75 * time.Millisecond)
			}
		} else {
			time.Sleep(time.Duration(1+rand.Intn(5)) * time.Millisecond)
		}
		return rpp.cm.AppendEntries(args, reply)
	}
```