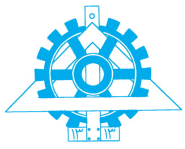
مبانی رایانش توزیع یافته

علی حمزه پور، مینا شیرازی، امیرعلی رحیمی



تمرین
کامپیوتری
شماره ۳

شماره دانشجویی:
۸۱۰۱۰۰۱۲۹
۸۱۰۱۰۰۲۵۰
۸۱۰۱۰۰۱۴۶

بخش اول

مقدمه

۱ پروتکل Raft

این پروژه بر روی پیاده‌سازی الگوریتم اجماع Raft متمرکز است. Raft پروتکلی برای مدیریت لاگ در سیستم‌های توزیع‌شده است که تحمل‌پذیری در برابر خرابی (Fault tolerance) را از طریق نگهداری نسخه‌های کامل وضعیت داده‌ها در چندین سرور فراهم می‌کند. چالش اصلی در سیستم‌های تکرار شده، حفظ سازگاری (Consistency) بین نسخه‌ها هنگام وقوع از کارافتادگی است. Raft با سازمان‌دهی درخواست‌های مشتری به صورت یک توالی مشخص به نام «لاگ» (Log)، اطمینان می‌دهد که همه سرورها نسخه یکسانی از لاگ را مشاهده می‌کنند. هر سرور دستورات (Commands) را به ترتیب در لاگ خود اعمال می‌کند و به این ترتیب، همگی وضعیت یکسانی را حفظ می‌کنند. در صورت از کار افتادن یک سرور و سپس بهبود یافتن آن، Raft مسئول هماهنگ‌سازی و به‌روزرسانی لاگ آن خواهد بود. سیستم تا زمانی که اکثریت سرورها فعال باشند و بتوانند با یکدیگر ارتباط برقرار کنند، به کار خود ادامه می‌دهد. در این پروژه، ما الگوریتم Raft را به صورت یک شی (Object) در زبان Go پیاده‌سازی می‌کنیم تا قابل استفاده در یک ماژول بزرگتر باشد. نمونه‌های Raft از طریق RPC با یکدیگر ارتباط برقرار می‌کنند و لاگ‌ها را هماهنگ نگه می‌دارند. رابط (Interface) ماژول Raft باید از دنباله‌ای نامحدود از دستورات شماره‌گذاری شده (ورودی‌ها) پشتیبانی کند. این ورودی‌ها به صورت اعدادی با شماره ایندکس (Index) ذخیره می‌شوند و هر ورودی با ایندکس معتبر، باید وارد لاگ شده و سپس به سیستم بزرگتر تحویل داده شود. طراحی پیاده‌سازی ما به خصوص بر اساس "شکل دوم" از مقاله توسعه‌یافته Raft (Extended Version) خواهد بود که اجزای کلیدی طراحی، از جمله ذخیره‌سازی پایدار، بازیابی پس از خرابی، و اعمال تغییرات پیکربندی را توضیح می‌دهد.

۲ مراحل پروژه

پروژه به چندین بخش تقسیم می‌شود:

- **انتخاب رهبر (Leader Election – قسمت 3A)**: پیاده‌سازی مکانیزم انتخاب رهبر و اطمینان از انتخاب رهبر جدید در صورت از کار افتادن رهبر فعلی یا قطع ارتباط آن. همچنین شامل پیاده‌سازی پیام‌های AppendEntries بدون محتوای لاگ برای ارسال ضربان قلب (heartbeat) است.

- **ثبت لاگ (Log Replication – قسمت 3B)**: پیاده‌سازی کد رهبر و پیرو (follower) برای افزودن ورودی‌های جدید به لاگ، ارسال آنها به همه پیروها، و اعمال آنها از طریق کانال applyCh.

- **پایداری (Persistence – قسمت 3C)**: اطمینان از اینکه سرور Raft پس از راه‌اندازی مجدد می‌تواند ادامه کار را از جایی که متوقف شده، از سر بگیرد. این بخش شامل ذخیره و بازیابی وضعیت پایدار Raft با استفاده از شی Persister است.

- **فشرده‌سازی لاگ (Log Compaction – قسمت 3D)**: تغییر پیاده‌سازی برای استفاده از وضعیت لحظه‌ای سیستم (snapshot) و حذف بخش‌های قدیمی لاگ جهت کاهش داده‌های ذخیره شده و افزایش سرعت راه‌اندازی. این بخش همچنین شامل پیاده‌سازی InstallSnapshot برای زمانی است که یک پیرو بسیار عقب‌تر از رهبر باشد.

بخش دوم

قسمت 3A: انتخاب رهبر

در این بخش، مکانیزم انتخاب رهبر (Leader Election) در الگوریتم Raft پیاده‌سازی شده است. هدف اصلی این قسمت، اطمینان از انتخاب یک رهبر در شرایط عادی و همچنین جایگزینی رهبر جدید در صورت از کار افتادن رهبر فعلی یا قطع ارتباط آن است. علاوه بر این، پیام‌های AppendEntries بدون محتوای لاگ نیز برای ارسال ضربان قلب (Heartbeat) پیاده‌سازی شده‌اند تا رهبر فعال بودن خود را به پیروان اطلاع دهد. این بخش تضمین می‌کند که کلاستر Raft همواره یک رهبر فعال داشته باشد و در صورت عدم دریافت پیام از رهبر، پیروها بتوانند فرآیند انتخاب رهبر جدید را آغاز کنند.

۱ ساختارهای داده اصلی (Raft Struct)

ساختار Raft وضعیت یک سرور Raft را نگهداری می‌کند و شامل فیلدهای زیر است:

```
1 type Raft struct {
```

```
    mu          sync.Mutex          // Lock to protect shared access to this
        peer's state
    peers       []*labrpc.ClientEnd // RPC end points of all peers
    persister   *tester.Persister   // Object to hold this peer's persisted
       state
    me          int                 // this peer's index into peers[]
    dead        int32               // set by Kill()
    gotPulse    bool
    state       State
    voteCount   int

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

    // volatile state on all servers
    commitIndex int
    lastApplied int

    // volatile state on leaders
    nextIndex   []int
    matchIndex  []int
}
```

- **mu sync.Mutex**: یک Mutex برای حفاظت از دسترسی مشترک به وضعیت سرور. استفاده از Mutex برای جلوگیری از Race Condition ها در دسترسی به متغیرهای مشترک بین Goroutineها ضروری است.
- **peers []*labrpc.ClientEnd**: آرایه‌ای از نقاط پایانی (ClientEnd) RPC برای همه سرورهای همتا در کلاستر. این آرایه برای ارسال RPC به سایر سرورها استفاده می‌شود.

- **persister *tester.Persister**: شیئی برای ذخیره‌سازی وضعیت پایدار سرور، مانند log، currentTerm، و votedFor. این اطلاعات در صورت راه‌اندازی مجدد سرور قابل بازیابی هستند.
- **me int**: ایندکس سرور فعلی در آرایه peers.
- **dead int32**: یک پرچم اتمیک که توسط تابع ()Kill تنظیم می‌شود تا نشان دهد سرور خاموش شده است. این پرچم به Goroutineهای در حال اجرا کمک می‌کند تا به graceful exit برسند و منابع CPU را مصرف نکنند.
- **gotPulse bool**: یک پرچم که نشان می‌دهد آیا سرور در دوره جاری (Election Timeout) یک Heartbeat از رهبر دریافت کرده است یا خیر. این پرچم برای تشخیص نیاز به شروع انتخابات جدید استفاده می‌شود.
- **state State**: وضعیت فعلی سرور، که می‌تواند Follower، Candidate یا Leader باشد.
- **voteCount int**: تعداد رأی‌هایی که یک کاندیدا در طول انتخابات دریافت کرده است.
- **currentTerm int**: آخرین ترم شناخته شده سرور (افزایش می‌یابد به صورت یکنواخت).
- **votedFor int**: شناسه کاندیدایی که سرور در ترم فعلی به آن رأی داده است (یا NotVoted اگر رأی نداده باشد).
- **log []LogEntry**: لاگ‌های ورودی سرور که شامل دستورات و ترم مربوطه هستند.

۲ تغییر وضعیت‌های Raft

در Raft، هر سرور می‌تواند در یکی از سه حالت زیر قرار داشته باشد:

```
const (
    Follower State = iota
    Candidate
    Leader
)
```

- **Follower (پیرو)**: حالت اولیه برای همه سرورها. پیروها به درخواست‌های رهبر (AppendEntries) و کاندیداها (RequestVote) پاسخ می‌دهند. اگر پیرو برای مدت زمان Election Timeout از رهبر پیامی دریافت نکند، به حالت کاندیدا تغییر وضعیت می‌دهد.
- **Candidate (کاندیدا)**: سرور زمانی به این حالت می‌رود که تصمیم به شروع انتخابات بگیرد. کاندیدا ترم خود را افزایش می‌دهد، به خود رأی می‌دهد و درخواست رأی (RequestVote RPC) را به سایر سرورها ارسال می‌کند. اگر کاندیدا از اکثریت رأی‌ها را به دست آورد، به رهبر تبدیل می‌شود.
- **Leader (رهبر)**: سروری که اکثریت رأی‌ها را کسب کرده و مسئول مدیریت لاگ و ارسال پیام‌های Heartbeat به پیروها است. رهبر به صورت دوره‌ای پیام‌های AppendEntries را به همه پیروها ارسال می‌کند.

۳ توابع کلیدی و جریان کار

۱.۳ Make(peers, me, persister, applyCh)

این تابع یک نمونه جدید از سرور Raft را ایجاد و مقداردهی اولیه می‌کند.

```
1  func Make(peers []*labrpc.ClientEnd, me int,
2      persister *tester.Persister, applyCh chan raftapi.ApplyMsg) raftapi.Raft
           {
3      rf := &Raft{}
4      rf.peers = peers
5      rf.persister = persister
6      rf.me = me
7      rf.state = Follower // initial state
8      rf.gotPulse = true  // to avoid starting an election immediately
9      rf.currentTerm = InitialTerm
10     rf.log = make([]LogEntry, 0)
11     rf.log = append(rf.log, LogEntry{Term: InitialTerm, Command: nil}) //
           initial empty log entry
12
13
```

```
    // initialize from state persisted before a crash
    rf.readPersist(persister.ReadRaftState())

    // start ticker goroutine to start elections
    go rf.ticker()

    return rf
}
```

- در اینجا **rf.state** به Follower اولیه می‌شود و **rf.gotPulse** روی true تنظیم می‌شود تا از شروع فوری انتخابات جلوگیری شود.
- **rf.currentTerm** به InitialTerm تنظیم می‌شود.
- یک ورودی لاگ اولیه و خالی به **rf.log** اضافه می‌شود.
- تابع **readPersist** برای بازیابی وضعیت پایدار قبلی (اگر وجود داشته باشد) فراخوانی می‌شود.
- یک **Goroutine** جدید برای اجرای ticker() راه‌اندازی می‌شود. این Goroutine مسئول بررسی نیاز به انتخابات و شروع آن است.

۲.۳ ticker()

این Goroutine اصلی در هر سرور Raft است که به صورت نامحدود اجرا می‌شود تا زمانی که سرور killed() شود.

```
func (rf *Raft) ticker() {
    for !rf.killed() {
        // Check if a leader election should be started.
        rf.mu.Lock()
        needElection := (rf.state != Leader) && !rf.gotPulse
        rf.gotPulse = false
        rf.mu.Unlock()

        if needElection {
```

```
            rf.startElection()
        }

        // pause for a random amount of time between 50 and 350
        // milliseconds.
        ms := 50 + (rand.Int63() % 300)
        time.Sleep(time.Duration(ms) * time.Millisecond)
    }
}
```

- در هر چرخه، **ticker** بررسی می‌کند که آیا سرور در وضعیت Leader نیست و آیا پیام Heartbeat از رهبر دریافت نکرده است (!rf.gotPulse).
- اگر این شرایط برقرار باشند، تابع **startElection()** فراخوانی می‌شود.
- سپس، **ticker** برای یک بازه زمانی تصادفی (بین ۵۰ تا ۳۵۰ میلی‌ثانیه) time.Sleep می‌کند تا از همگامی تایمرهای انتخابات در سرورهای مختلف جلوگیری شود و احتمال برخورد (Collision) کاهش یابد.

۳.۳ startElection()

این تابع توسط یک سرور فراخوانی می‌شود که تصمیم گرفته است انتخابات را آغاز کند:

```
func (rf *Raft) startElection() {
    rf.mu.Lock()
    rf.currentTerm++;
    rf.votedFor = rf.me;
    rf.voteCount = 1;
    rf.state = Candidate;
    rf.mu.Unlock();

    for i := range rf.peers {
        if i == rf.me {
            continue
```

```
        }

        go rf.prepareAndSendRequestVote(i)
    }
}
```

- ابتدا **rf.currentTerm** افزایش می‌یابد.
- سرور به خود رأی می‌دهد (**rf.votedFor = rf.me**) و **rf.voteCount** را به ۱ تنظیم می‌کند.
- وضعیت سرور به **Candidate** تغییر می‌کند.
- سپس، برای هر یک از همتاهای دیگر، یک **Goroutine** جدید برای فرستادن درخواست راه‌اندازی می‌شود.

۴.۳ ساختار RequestVoteArgs و RequestVoteReply

این ساختارها برای ارسال درخواست رأی (RequestVote RPC) و دریافت پاسخ آن استفاده می‌شوند.

```
type RequestVoteArgs struct {
    Term         int // candidate's term
    CandidateId  int // candidate requesting vote
    LastLogIndex int // index of candidate's last log entry
    LastLogTerm  int // term of candidate's last log entry
}

type RequestVoteReply struct {
    Term        int  // current term, for candidate to update itself
    VoteGranted bool // true means candidate received vote
}
```

۵.۳ prepareAndSendRequestVote(server int)

این تابع مسئول آماده‌سازی و ارسال درخواست رأی (RequestVote RPC) به یک سرور خاص است.

```go
func (rf *Raft) prepareAndSendRequestVote(server int) {
    rf.mu.Lock()

    lastLogIndex, lastLogTerm := rf.getLastLogIndexAndTerm()
    args := &RequestVoteArgs{
        Term:         rf.currentTerm,
        CandidateId:  rf.me,
        LastLogIndex: lastLogIndex,
        LastLogTerm:  lastLogTerm,
    }

    rf.mu.Unlock()

    reply := &RequestVoteReply{}

    if !rf.sendRequestVote(server, args, reply) {
        return // RPC failed, don't do anything
    }

    if rf.killed() {
        return
    }

    rf.mu.Lock()
    defer rf.mu.Unlock()

    rf.updateTerm(reply.Term)

    if rf.state != Candidate {
        return
    }
```

```
32
33      if reply.VoteGranted && reply.Term == rf.currentTerm {
34          rf.voteCount++
35
36          if rf.voteCount > len(rf.peers)/2 {
37              rf.state = Leader
38              go rf.heartbeatTicker()
39          }
40      }
41  }
```

- ابتدا **RequestVoteArgs** با currentTerm کاندیدا، CandidateId (خودش)، و اطلاعات آخرین ورودی لاگ (LastLogIndex، LastLogTerm) پر می‌شود.
- سپس **rf.sendRequestVote** برای ارسال RPC فراخوانی می‌شود.
- پس از دریافت پاسخ، سرور بررسی می‌کند:
  - اگر **reply.Term** از rf.currentTerm خودش بزرگتر باشد، سرور ترم خود را به روز می‌کند و به حالت Follower بازمی‌گردد (rf.updateTerm(reply.Term)).
  - اگر رأی اعطا شده باشد (**reply.VoteGranted**) و ترم پاسخ با ترم فعلی کاندیدا برابر باشد، rf.voteCount افزایش می‌یابد.
  - اگر **rf.voteCount** از نصف تعداد کل همتاها بیشتر شود (اکثریت)، سرور به Leader تبدیل می‌شود.
  - در صورت تبدیل شدن به Leader، یک **Goroutine** جدید برای اجرای heartbeatTicker() راه‌اندازی می‌شود.

۶.۳ تابع RPC RequestVote

این تابع، Handler RPC برای درخواست‌های رأی دریافتی از سایر سرورها است.

```
1  func (rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply)
       {
2      if rf.killed() {
```

```
        return
    }

    rf.mu.Lock()
    defer rf.mu.Unlock()

    rf.updateTerm(args.Term)
    reply.Term = rf.currentTerm

    if rf.votedFor != NotVoted || rf.votedFor == args.CandidateId {
        reply.VoteGranted = false
        return
    }

    isUpToDate := rf.isCandidateLogsUpToDate(args.LastLogIndex, args.
       LastLogTerm)

    if !isUpToDate || args.Term < rf.currentTerm {
        reply.VoteGranted = false
        return
    }

    reply.VoteGranted = true
    rf.votedFor = args.CandidateId
}
```

- ابتدا، ترم سرور دریافت‌کننده (پیرو) با **args.Term** کاندیدا به‌روزرسانی می‌شود (rf.updateTerm(args.Term)).
- پیرو به کاندیدا رأی نمی‌دهد اگر:
  - قبلاً در ترم فعلی به شخص دیگری رأی داده باشد (**rf.votedFor != NotVoted**).
  - **args.Term** کاندیدا کمتر از rf.currentTerm پیرو باشد.

- لاگ کاندیدا به اندازه پیرو به‌روز نباشد (**!rf.isCandidateLogsUpToDate**).

- در غیر این صورت، پیرو به کاندیدا رأی می‌دهد و **reply.VoteGranted** را true تنظیم می‌کند و rf.votedFor را به args.CandidateId تغییر می‌دهد.

۷.۳ ساختار AppendEntriesArgs و AppendEntriesReply

این ساختارها برای ارسال پیام‌های AppendEntries و دریافت پاسخ آنها استفاده می‌شوند. در بخش 3A، Entries در AppendEntriesArgs خالی است.

```
type AppendEntriesArgs struct {
    Term int // 'leaders term
    // TODO: add more fields
}

type AppendEntriesReply struct {
    Term int // currentTerm, for leader to update itself

    // TODO: uncomment
    // Success bool //true if follower contained entry matching prevLogIndex
        and prevLogTerm
}
```

۸.۳ تابع RPC AppendEntries

این تابع، Handler RPC برای پیام‌های AppendEntries دریافتی از رهبر است. در بخش 3A، این پیام‌ها فقط به عنوان Heartbeat استفاده می‌شوند (یعنی Entries خالی هستند).

```
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *
   AppendEntriesReply) {
    if rf.killed() {
        return
    }

```

```
    rf.mu.Lock()
    defer rf.mu.Unlock()

    rf.gotPulse = true
    rf.updateTerm(args.Term)
    reply.Term = rf.currentTerm
}
```

- هنگام دریافت این RPC، **rf.gotPulse** به true تنظیم می‌شود تا نشان دهد سرور از رهبر پیامی دریافت کرده و نیازی به شروع انتخابات نیست.
- **rf.currentTerm** سرور با args.Term رهبر مقایسه و به‌روزرسانی می‌شود.

۹.۳ heartbeatTicker()

این Goroutine فقط در سرور Leader اجرا می‌شود.

```
func (rf *Raft) heartbeatTicker() {
    for !rf.killed() {
        rf.mu.Lock()

        if rf.state != Leader {
            rf.mu.Unlock()
            return
        }
        rf.sendHeartbeat()

        rf.mu.Unlock()

        time.Sleep(time.Duration(HeartBeatDelay) * time.Millisecond)
    }
}
```

- به صورت دوره‌ای (با تأخیر **HeartBeatDelay** میلی‌ثانیه)، رهبر sendHeartbeat() را فراخوانی می‌کند.
- اگر در حین اجرای این Goroutine، وضعیت سرور از Leader تغییر کند، Goroutine متوقف می‌شود.

۱۰.۳ sendHeartbeat()

این تابع توسط رهبر برای ارسال پیام‌های Heartbeat به همه پیروها استفاده می‌شود.

```
func (rf *Raft) sendHeartbeat() {
    for i := range rf.peers {
        if i == rf.me {
            continue
        }

        args := &AppendEntriesArgs{
            Term: rf.currentTerm,
        }
        reply := &AppendEntriesReply{}
        go rf.sendAppendEntries(i, args, reply)
    }
}
```

- برای هر همتا (به جز خودش)، یک **AppendEntriesArgs** با currentTerm رهبر ایجاد می‌شود.
- سپس یک **Goroutine** جدید برای فراخوانی rf.sendAppendEntries راه‌اندازی می‌شود.

۱۱.۳ updateTerm(term int)

این تابع یک تابع کمکی است که وضعیت سرور را بر اساس یک term جدید به‌روزرسانی می‌کند.

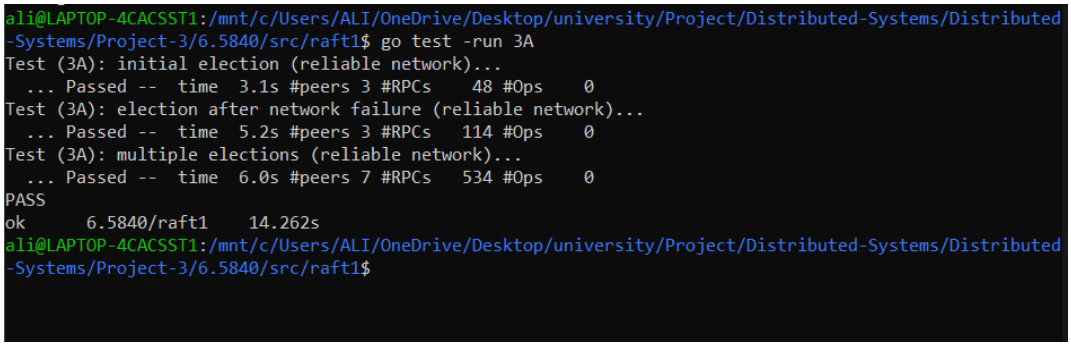
```
func (rf *Raft) updateTerm(term int) {
    if term > rf.currentTerm {
        rf.currentTerm = term
        rf.votedFor = NotVoted
        rf.state = Follower
```

```
    }
}
```

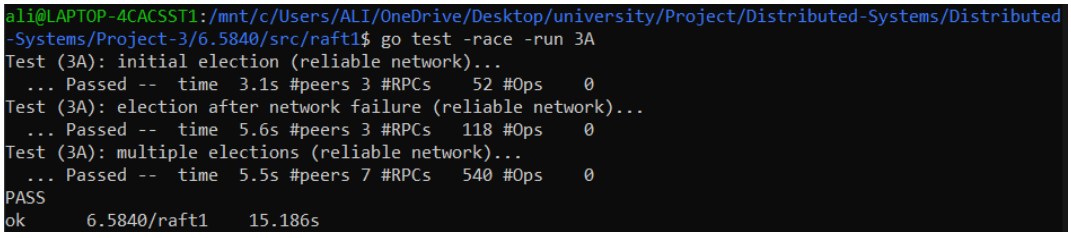
- اگر **term** ورودی بزرگتر از rf.currentTerm فعلی باشد، rf.currentTerm به‌روزرسانی می‌شود، rf.votedFor به NotVoted تنظیم می‌شود و وضعیت سرور به Follower تغییر می‌کند. این تضمین می‌کند که سرور همیشه به آخرین ترم شناخته شده خود به‌روز باشد.

۴ تست قدم 3A

تست‌های این قسمت را یک‌بار به صورت عادی و یک‌بار با race- اجرا کردیم و در هر دو حالت تست‌ها قبول شدند.



شکل ۱: نتیجه اجرای تست‌های 3A



شکل ۲: نتیجه اجرای تست‌های 3A با race-

بخش سوم

قسمت 3B: ثبت لاگ

در این بخش، تمرکز بر پیاده‌سازی مکانیزم ثبت و تکرار لاگ (Log Replication) در الگوریتم Raft است. پس از انتخاب رهبر، وظیفه اصلی آن اطمینان از سازگاری لاگ‌ها بین خود و پیروانش است. این فرآیند شامل افزودن ورودی‌های جدید به لاگ، ارسال آن‌ها به پیروها و تضمین اعمال صحیح آن‌ها بر روی ماشین‌های حالت (State Machines) همه سرورهاست. رهبر مسئول دریافت دستورات جدید از کلاینت‌ها، اضافه کردن آن‌ها به لاگ محلی خود، و سپس ارسال این ورودی‌ها به پیروان از طریق پیام‌های AppendEntries است. پیروها نیز مسئول بررسی سازگاری لاگ و پذیرش یا رد ورودی‌های دریافتی هستند. این بخش، قلب عملکرد Raft در حفظ سازگاری داده‌ها در یک سیستم توزیع‌شده است.

۱ تغییرات اعمال شده در کد

۱.۱ تغییرات در ساختار Raft

برای پشتیبانی از ثبت و تکرار لاگ، فیلدهای جدیدی به ساختار Raft اضافه شده‌اند:

```
type Raft struct {
    mu          sync.Mutex
    peers       []*labrpc.ClientEnd
    persister   *tester.Persister
    me          int
    dead        int32
    applyCh    chan raftapi.ApplyMsg
    applierCond *sync.Cond

    gotPulse  bool
    state     State
    voteCount int

    currentTerm int
```

```
    votedFor    int
    log         []LogEntry

    commitIndex int
    lastApplied int

    nextIndex  []int
    matchIndex []int
}
```

- **applyCh chan raftapi.ApplyMsg**: کانالی که Raft پیام‌های اعمال شده (ApplyMsg) را به سرویس بالاسری خود (مانند سرور key/value) ارسال می‌کند. این کانال برای ارسال دستوراتی که به صورت قطعی (committed) در لاگ ثبت شده‌اند، استفاده می‌شود.
- **applierCond *sync.Cond**: یک sync.Cond که برای هماهنگی بین Goroutine اصلی Raft و Goroutine applier استفاده می‌شود. این شرط به applier اجازه می‌دهد تا زمانی که ورودی‌های جدیدی برای اعمال وجود دارند، بیدار شود.
- **commitIndex int**: ایندکس بالاترین ورودی در لاگ که می‌دانیم به صورت قطعی تکرار شده است (به ماشین حالت اعمال شده است).
- **lastApplied int**: ایندکس بالاترین ورودی در لاگ که به ماشین حالت محلی اعمال شده است.
- **nextIndex []int**: برای هر سرور همتا، ایندکس بعدی‌ترین ورودی لاگ که رهبر باید آن را به آن سرور ارسال کند. این مقدار هنگام تبدیل شدن به رهبر، با طول لاگ رهبر + ۱ مقداردهی اولیه می‌شود.
- **matchIndex []int**: برای هر سرور همتا، ایندکس بالاترین ورودی لاگ که رهبر می‌داند با لاگ آن سرور مطابقت دارد.

۲.۱ ساختارهای AppendEntriesArgs و AppendEntriesReply

ساختارهای مربوط به فراخوانی RPC AppendEntries و پاسخ آن نیز گسترش یافته‌اند تا اطلاعات لازم برای تکرار لاگ را منتقل کنند:

```go
type AppendEntriesArgs struct {
    Term         int
    LeaderId     int
    PrevLogIndex int
    PrevLogTerm  int
    Entries      []LogEntry
    LeaderCommit int
}

type AppendEntriesReply struct {
    Term          int
    Success       bool
    ConflictTerm  int
    ConflictIndex int
    XLen          int
}
```

- **AppendEntriesArgs**:
  - **Term**: ترم فعلی رهبر.
  - **LeaderId**: شناسه رهبر، برای اینکه پیروها بتوانند درخواست‌ها را به او هدایت کنند.
  - **PrevLogIndex**: ایندکس ورودی قبل از اولین ورودی جدیدی که قرار است ارسال شود.
  - **PrevLogTerm**: ترم ورودی با ایندکس PrevLogIndex. این دو فیلد برای بررسی سازگاری لاگ بین رهبر و پیرو حیاتی هستند.
  - **Entries**: آرایه‌ای از ورودی‌های لاگ جدید برای ذخیره. این آرایه می‌تواند خالی باشد (برای پیام‌های Heartbeat).
  - **LeaderCommit**: commitIndex رهبر.
- **AppendEntriesReply**:
  - **Term**: ترم فعلی پیرو، برای اینکه رهبر ترم خود را به‌روز کند.

- **Success**: true اگر AppendEntries با موفقیت انجام شود.
- **ConflictTerm، ConflictIndex، XLen**: این فیلدها در صورت عدم موفقیت (= Success false) توسط پیرو پر می‌شوند تا به رهبر کمک کنند نقطه عدم تطابق لاگ را سریع‌تر پیدا کند.

۳.۱ تابع Start(command interface)

این تابع توسط سرویس بالاسری برای شروع توافق بر روی یک دستور جدید استفاده می‌شود. اگر سرور فعلی رهبر باشد، دستور به لاگ خود اضافه شده و سپس به پیروها ارسال می‌شود.

```
1  func (rf *Raft) Start(command interface{}) (int, int, bool) {
2      rf.mu.Lock()
3      defer rf.mu.Unlock()
4
5      term, isLeader := rf.currentTerm, rf.state == Leader
6      if !isLeader {
7          return -1, -1, false
8      }
9      rf.log = append(rf.log, LogEntry{Command: command, Term: term})
10
11     index := len(rf.log) - 1
12
13     for i := range rf.peers {
14         if i == rf.me {
15             continue
16         }
17         // Prepare and send AppendEntries RPC to followers
18         prevLogIndex, prevLogTerm := rf.nextIndex[i] - 1, rf.log[rf.
               nextIndex[i] - 1].Term
19         sendEntries := make([]LogEntry, len(rf.log[rf.nextIndex[i]:]))
20         copy(sendEntries, rf.log[rf.nextIndex[i]:])
21         args := &AppendEntriesArgs{
22             Term: rf.currentTerm,
```

```
            LeaderId: rf.me,
            Entries:  sendEntries,
            PrevLogIndex: prevLogIndex,
            PrevLogTerm:  prevLogTerm,
            LeaderCommit: rf.commitIndex,
        }

        go rf.sendAppendEntries(i, args)
    }
    return index, term, isLeader
}
```

- اگر سرور رهبر نباشد، بلافاصله false برگردانده می‌شود.
- در غیر این صورت، دستور جدید به عنوان LogEntry به لاگ محلی رهبر اضافه می‌شود.
- سپس، رهبر یک پیام AppendEntries برای هر پیرو آماده می‌کند و آن را در یک Goroutine جداگانه ارسال می‌کند. این پیام شامل ورودی‌های لاگ جدید (از nextIndex[i] به بعد)، ترم فعلی رهبر و commitIndex رهبر است.

۴.۱ تابع AppendEntries RPC

این تابع، Handler RPC برای پیام‌های AppendEntries دریافتی از رهبر است و منطق اصلی تکرار لاگ در سمت پیرو را پیاده‌سازی می‌کند.

```
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *
   AppendEntriesReply) {
    if rf.killed() {
        return
    }

    rf.mu.Lock()
    defer rf.mu.Unlock()
```

```
8
9       // Reply false if term < currentTerm (§5.1)
10      if rf.currentTerm > args.Term {
11          reply.Success = false
12          reply.Term = rf.currentTerm
13          return
14      }
15
16      //If AppendEntries RPC received from new leader: convert to follower
17      rf.state = Follower
18      rf.gotPulse = true
19      rf.updateTerm(args.Term) // This will also persist the state if term
            changed
20      reply.Term = rf.currentTerm
21      reply.Success = true
22
23      // Reply false if log 'doesnt contain an entry
24      // at prevLogIndex whose term matches prevLogTerm (§5.3)
25      if !rf.isLogMatching(args.PrevLogIndex, args.PrevLogTerm) {
26          reply.Success = false
27          reply.XLen = len(rf.log)
28          reply.ConflictIndex, reply.ConflictTerm = rf.findConflictData(args.
                PrevLogIndex)
29          return
30      }
31
32      // If an existing entry conflicts with a new one
33      // (same index but different terms), delete the
34      // existing entry and all that follow it (§5.3)
35      // Append any new entries not already in the log
36      startIndex := args.PrevLogIndex + 1
```

```
    i := 0
    for ; i < len(args.Entries); i++ {
        if startIndex+i >= len(rf.log) {
            break
        }
        if rf.log[startIndex+i].Term != args.Entries[i].Term {
            rf.log = rf.log[:startIndex+i]
            break
        }
    }
    rf.log = append(rf.log, args.Entries[i:]...)

    //If leaderCommit > commitIndex, set commitIndex =
    // min(leaderCommit, index of last new entry)
    if args.LeaderCommit > rf.commitIndex {
        lastNewEntryIndex := args.PrevLogIndex + len(args.Entries)
        rf.commitIndex = min(args.LeaderCommit, lastNewEntryIndex)
        rf.applierCond.Signal() // Signal the applier goroutine to apply new
            entries
    }
}
```

- ابتدا، پیرو ترم رهبر را با ترم خود مقایسه می‌کند. اگر ترم رهبر کمتر باشد، پیرو درخواست را رد می‌کند.
- اگر ترم رهبر بزرگتر یا مساوی باشد، پیرو به وضعیت Follower تغییر می‌کند و gotPulse را true تنظیم می‌کند. updateTerm نیز فراخوانی می‌شود تا ترم محلی به‌روز شود.
- سپس، پیرو بررسی می‌کند که آیا ورودی PrevLogIndex و PrevLogTerm در لاگ خود موجود و مطابق با لاگ رهبر است. اگر نباشد، Success را false تنظیم کرده و اطلاعات مربوط به نقطه تضاد را برای کمک به رهبر در یافتن سریع‌تر نقطه همگام‌سازی (ConflictTerm، ConflictIndex، XLen) در reply قرار می‌دهد.

- اگر لاگ‌ها مطابقت داشته باشند، پیرو ورودی‌های لاگ جدید را از args.Entries به لاگ خود اضافه می‌کند. اگر ورودی‌های موجود با ورودی‌های جدید تضاد داشته باشند (همان ایندکس اما ترم متفاوت)، ورودی‌های تضاد و همه ورودی‌های بعدی حذف شده و ورودی‌های جدید اضافه می‌شوند.
- در نهایت، commitIndex پیرو به‌روزرسانی می‌شود (بر اساس LeaderCommit و ایندکس آخرین ورودی جدید) و applierCond.Signal() فراخوانی می‌شود تا Goroutine applier از وجود ورودی‌های جدید برای اعمال آگاه شود.

۵.۱ sendAppendEntries

این تابع در سمت رهبر اجرا می‌شود و مسئول ارسال RPC AppendEntries به یک پیرو خاص و پردازش پاسخ آن است.

```
1  func (rf *Raft) sendAppendEntries(server int, args *AppendEntriesArgs) {
2      reply := &AppendEntriesReply{}
3      ok := rf.peers[server].Call("Raft.AppendEntries", args, reply)
4
5      if !ok{
6          return
7      }
8
9      rf.mu.Lock()
10     defer rf.mu.Unlock()
11
12     if reply.Term > rf.currentTerm{
13         rf.updateTerm(reply.Term)
14         return
15     }
16
17     if rf.state != Leader || rf.killed(){
18         return
19     }
20
```

```
    if reply.Success{
        // Update nextIndex and matchIndex for the follower
        if len(args.Entries) > 0{
            rf.nextIndex[server] = len(args.Entries) + args.PrevLogIndex + 1

        }
        rf.matchIndex[server] = rf.nextIndex[server] - 1

        // Attempt to commit new entries
        for i := rf.commitIndex + 1; i < len(rf.log); i++{
            count := 1 // Count itself
            for peer := range rf.peers{
                if peer != rf.me && rf.matchIndex[peer] >= i{
                    count ++
                }
            }
            if count > len(rf.peers) / 2 && rf.log[i].Term == rf.currentTerm
                {
                rf.commitIndex = i
            }
        }
        rf.applierCond.Signal()
    } else if args.Term >= reply.Term{
        // Handle log inconsistency
        if reply.XLen <= args.PrevLogIndex{
            rf.nextIndex[server] = reply.XLen
        }else if rf.hasTerm(reply.ConflictTerm){
            rf.nextIndex[server] = rf.findLastEntryIndex(reply.ConflictTerm)
                + 1
        }else{
            rf.nextIndex[server] = reply.ConflictIndex
```

```
49          }
50
51          // Retry sending AppendEntries with adjusted nextIndex
52          prevLogIndex, prevLogTerm := rf.nextIndex[server] - 1, rf.log[rf.
                nextIndex[server] - 1].Term
53          sendEntries := make([]LogEntry, len(rf.log[rf.nextIndex[server]:]))
54          copy(sendEntries, rf.log[rf.nextIndex[server]:])
55          newArgs := &AppendEntriesArgs{
56              Term: rf.currentTerm,
57              LeaderId: rf.me,
58              Entries:  sendEntries,
59              PrevLogIndex: prevLogIndex,
60              PrevLogTerm:  prevLogTerm,
61              LeaderCommit: rf.commitIndex,
62          }
63          go rf.sendAppendEntries(server, newArgs) // Recursive call (in a new
                goroutine)
64      }
65  }
```

- پس از دریافت پاسخ از پیرو، رهبر ابتدا ترم خود را با reply.Term به‌روز می‌کند (اگر reply.Term بزرگتر باشد).
- اگر **reply.Success** true باشد:
  - **nextIndex[server]** و **matchIndex[server]** برای آن پیرو به‌روزرسانی می‌شوند.
  - رهبر تلاش می‌کند تا ورودی‌های لاگ جدید را commit کند. این کار با بررسی ورودی‌هایی در لاگ رهبر (از commitIndex + 1 به بعد) که در لاگ اکثریت سرورها (matchIndex) مطابقت دارند و ترم آن‌ها با ترم رهبر فعلی یکسان است، انجام می‌شود. اگر چنین ورودی‌ای یافت شود، commitIndex رهبر به‌روزرسانی شده و applierCond.Signal() فراخوانی می‌شود.
- اگر **reply.Success** false باشد و args.Term از reply.Term کمتر نباشد (یعنی تضاد لاگ):

- رهبر nextIndex[server] را کاهش می‌دهد تا نقطه تضاد لاگ را در پیرو بیابد. این شامل منطق برای استفاده از ConflictTerm، ConflictIndex، و XLen از پاسخ پیرو برای بهینه‌سازی فرآیند کاهش nextIndex است (با استفاده از XLen اگر PrevLogIndex نامعتبر باشد، یا ConflictIndex اگر ترمی که تضاد دارد در لاگ رهبر نباشد، یا findLastEntryIndex برای یافتن آخرین ورودی با ConflictTerm).
- سپس، رهبر یک RPC AppendEntries جدید با nextIndex کاهش یافته به همان پیرو ارسال می‌کند (این کار در یک Goroutine جدید برای جلوگیری از مسدود شدن انجام می‌شود).

۶.۱ heartbeatTicker() و sendHeartbeat()

این توابع که در بخش 3A برای Heartbeat استفاده می‌شدند، اکنون با منطق ارسال ورودی‌های لاگ واقعی به روز شده‌اند:

```
1  func (rf *Raft) heartbeatTicker() {
2      for !rf.killed() {
3          rf.mu.Lock()
4
5          if rf.state != Leader {
6              rf.mu.Unlock()
7              return
8          }
9          rf.sendHeartbeat()
10
11         rf.mu.Unlock()
12
13         time.Sleep(time.Duration(HeartBeatDelay) * time.Millisecond)
14     }
15 }
16
17 func (rf *Raft) sendHeartbeat() {
18     for i := range rf.peers {
19         if i == rf.me {
```

```
            continue
        }
        // Calculate prevLogIndex and prevLogTerm based on nextIndex for
           this follower
        prevLogIndex, prevLogTerm := rf.nextIndex[i] - 1, rf.log[rf.
           nextIndex[i] - 1].Term

        // Send log entries starting from nextIndex[i]
        sendEntries := make([]LogEntry, len(rf.log[rf.nextIndex[i]:]))
        copy(sendEntries, rf.log[rf.nextIndex[i]:])

        args := &AppendEntriesArgs{
            Term:         rf.currentTerm,
            LeaderId:     rf.me,
            Entries:      sendEntries,
            PrevLogIndex: prevLogIndex,
            PrevLogTerm:  prevLogTerm,
            LeaderCommit: rf.commitIndex,
        }
        go rf.sendAppendEntries(i, args)
    }
}
```

- **sendHeartbeat** اکنون به جای ارسال پیام‌های AppendEntries خالی، ورودی‌های لاگ را از rf.nextIndex[i] به بعد ارسال می‌کند. این تضمین می‌کند که پیروها به محض دریافت Heartbeat، لاگ‌های خود را با رهبر همگام‌سازی می‌کنند.

۷.۱ Goroutine applier()

این Goroutine به صورت جداگانه در پس‌زمینه اجرا می‌شود و مسئول اعمال ورودی‌های لاگ commit شده به ماشین حالت محلی است.

```go
func (rf *Raft) applier(){
    for !rf.killed(){
        rf.mu.Lock()
        // Wait for new entries to be committed
        for rf.commitIndex <= rf.lastApplied {
            rf.applierCond.Wait()
        }

        // Copy entries to apply and update lastApplied
        commitIndex, lastApplied := rf.commitIndex, rf.lastApplied
        entries := make([]LogEntry, commitIndex-lastApplied)
        copy(entries, rf.log[lastApplied+1:commitIndex+1])
        startIndex := lastApplied + 1
        rf.lastApplied = commitIndex
        rf.mu.Unlock()

        // Apply entries to the state machine
        for i, entry := range entries{
            rf.applyCh <- raftapi.ApplyMsg{
                CommandValid: true,
                Command:      entry.Command,
                CommandIndex: startIndex + i,
            }
        }
    }
}
```

- **applier** منتظر می‌ماند تا commitIndex از lastApplied بزرگتر شود (یعنی ورودی‌های جدیدی برای اعمال وجود داشته باشند).
- هنگامی که ورودی‌های جدیدی برای اعمال وجود دارند، آن‌ها را از لاگ کپی کرده و lastApplied را

به‌روزرسانی می‌کند.

- سپس، ورودی‌های کپی شده از طریق applyCh به سرویس بالاسری ارسال می‌شوند تا به ماشین حالت اعمال شوند. این فرآیند خارج از قفل mu انجام می‌شود تا از مسدود شدن عملیات Raft جلوگیری شود.

۸.۱ توابع کمکی جدید/تغییر یافته

- **isLogMatching(index int, term int)**: بررسی می‌کند که آیا یک ورودی لاگ در index مشخص شده با term داده شده مطابقت دارد یا خیر.

```
func (rf *Raft) isLogMatching(index int, term int) bool {
    if index < 0 || index >= len(rf.log) {
        return false
    }
    return rf.log[index].Term == term
}
```

- **findConflictData(prevLogIndex int)**: این تابع در سمت پیرو هنگام رد AppendEntries استفاده می‌شود تا به رهبر اطلاعاتی درباره نقطه تضاد لاگ ارائه دهد.

```
func (rf *Raft) findConflictData(prevLogIndex int) (int, int){
    if prevLogIndex >= len(rf.log){
        return -1, -1 // No conflict, or prevLogIndex is beyond our
            log length
    }
    conflictTerm := rf.log[prevLogIndex].Term
    for i := 0; i < len(rf.log); i++{
        if rf.log[i].Term == conflictTerm{
            return i, conflictTerm // Return first index with this
               term and the term itself
        }
    }
    return -1, -1 // Should not happen if prevLogIndex is within
       bounds and has a term
```

```
    }
```

- **hasTerm(term int)**: بررسی می‌کند که آیا ترم مشخص شده در لاگ سرور وجود دارد یا خیر.

```
    func (rf *Raft) hasTerm(term int) bool{
        for _, entry := range rf.log{
            if entry.Term == term{
                return true
            }
        }
        return false
    }
```

- **findLastEntryIndex(term int)**: آخرین ایندکس ورودی لاگ را که دارای ترم مشخص شده است، پیدا می‌کند.

```
    func (rf *Raft) findLastEntryIndex(term int) int{
        lastEntryIndex := -1
        for i, entry := range rf.log{
            if entry.Term == term{
                lastEntryIndex = i
            }
        }
        return lastEntryIndex
    }
```

- **becomeLeader()**: این تابع هنگام تبدیل شدن سرور به رهبر فراخوانی می‌شود و nextIndex و matchIndex را مقداردهی اولیه می‌کند.

```
    func (rf *Raft) becomeLeader(){
        rf.state = Leader
        lastLogIndex, _ := rf.getLastLogIndexAndTerm()
        rf.nextIndex = make([]int, len(rf.peers))
```

```
        rf.matchIndex = make([]int, len(rf.peers))
        for i := range rf.peers {
            rf.nextIndex[i] = lastLogIndex + 1
            rf.matchIndex[i] = 0 // Initially 0, meaning no entries
                matched yet
        }
        go rf.heartbeatTicker()
    }
```

۲ تست قدم 3B

تست‌های این قسمت را یک‌بار به صورت عادی و یک‌بار با race- اجرا کردیم و در هر دو حالت تست‌ها قبول شدند.

```
ali@LAPTOP-4CACSST1:/mnt/c/Users/ALI/OneDrive/Desktop/university/Project/Distributed-Systems/Distributed
-Systems/Project-3/6.5840/src/raft1$ go test -run 3B
Test (3B): basic agreement (reliable network)...
  ... Passed --  time  0.8s #peers 3 #RPCs    16 #Ops    0
Test (3B): RPC byte count (reliable network)...
  ... Passed --  time  1.7s #peers 3 #RPCs    48 #Ops    0
Test (3B): test progressive failure of followers (reliable network)...
  ... Passed --  time  4.8s #peers 3 #RPCs   100 #Ops    0
Test (3B): test failure of leaders (reliable network)...
  ... Passed --  time  5.4s #peers 3 #RPCs   166 #Ops    0
Test (3B): agreement after follower reconnects (reliable network)...
  ... Passed --  time  3.7s #peers 3 #RPCs    79 #Ops    0
Test (3B): no agreement if too many followers disconnect (reliable network)...
  ... Passed --  time  3.6s #peers 5 #RPCs   172 #Ops    0
Test (3B): concurrent Start()s (reliable network)...
  ... Passed --  time  0.6s #peers 3 #RPCs    18 #Ops    0
Test (3B): rejoin of partitioned leader (reliable network)...
  ... Passed --  time  4.4s #peers 3 #RPCs   131 #Ops    0
Test (3B): leader backs up quickly over incorrect follower logs (reliable network)...
  ... Passed --  time 18.9s #peers 5 #RPCs  2054 #Ops    0
Test (3B): RPC counts aren't too high (reliable network)...
  ... Passed --  time  2.1s #peers 3 #RPCs    56 #Ops    0
PASS
ok      6.5840/raft1    46.112s
```

شکل ۳: نتیجه اجرای تست‌های 3B

```
ali@LAPTOP-4CACSST1:/mnt/c/Users/ALI/OneDrive/Desktop/university/Project/Distributed-Systems/Distributed
-Systems/Project-3/6.5840/src/raft1$ go test -race -run 3B
Test (3B): basic agreement (reliable network)...
  ... Passed --  time  0.8s #peers 3 #RPCs    16 #Ops    0
Test (3B): RPC byte count (reliable network)...
  ... Passed --  time  1.7s #peers 3 #RPCs    48 #Ops    0
Test (3B): test progressive failure of followers (reliable network)...
  ... Passed --  time  4.8s #peers 3 #RPCs   102 #Ops    0
Test (3B): test failure of leaders (reliable network)...
  ... Passed --  time  5.4s #peers 3 #RPCs   168 #Ops    0
Test (3B): agreement after follower reconnects (reliable network)...
  ... Passed --  time  5.6s #peers 3 #RPCs   163 #Ops    0
Test (3B): no agreement if too many followers disconnect (reliable network)...
  ... Passed --  time  3.8s #peers 5 #RPCs   180 #Ops    0
Test (3B): concurrent Start()s (reliable network)...
  ... Passed --  time  0.6s #peers 3 #RPCs    20 #Ops    0
Test (3B): rejoin of partitioned leader (reliable network)...
  ... Passed --  time  6.4s #peers 3 #RPCs   165 #Ops    0
Test (3B): leader backs up quickly over incorrect follower logs (reliable network)...
  ... Passed --  time 17.3s #peers 5 #RPCs  2026 #Ops    0
Test (3B): RPC counts aren't too high (reliable network)...
  ... Passed --  time  2.0s #peers 3 #RPCs    52 #Ops    0
PASS
ok      6.5840/raft1    49.515s
```

شکل ۴: نتیجه اجرای تست‌های 3B با race-

بخش چهارم

بخش 3C: پایداری (Persistence)

در این بخش، قابلیت پایداری (Persistence) به الگوریتم Raft اضافه شده است. هدف اصلی این قسمت این است که سرور Raft بتواند پس از خرابی (Crash) و راه‌اندازی مجدد، وضعیت خود را از جایی که متوقف شده بود، بازیابی کرده و به کار خود ادامه دهد. Raft برای حفظ سازگاری در سیستم‌های توزیع‌شده، نیاز دارد تا برخی از وضعیت‌های خود را به صورت پایدار ذخیره کند تا در صورت از کار افتادن موقت، داده‌ها از بین نروند. این وضعیت‌های پایدار شامل currentTerm، votedFor، و کل log سرور هستند که مطابق با Figure 2 در مقاله Raft باید در حافظه پایدار نگهداری شوند. با پیاده‌سازی این بخش، سرورهای Raft در برابر از کار افتادن مقطعی مقاوم می‌شوند.

۱ توابع کلیدی و تغییرات

برای پیاده‌سازی پایداری، دو تابع اصلی اضافه شده و فراخوانی‌های persist() در نقاط مهمی از کد قرار داده شده‌اند.

۱.۱ تابع persist()

این تابع مسئول ذخیره وضعیت پایدار Raft در فضای ذخیره‌سازی پایدار است. در اینجا، از پکیج labgob برای کدگذاری (encoding) و bytes.Buffer برای نوشتن داده‌ها به یک بافر بایت استفاده می‌شود. سپس، این بایت‌ها از طریق شی rf.persister ذخیره می‌شوند.

```
func (rf *Raft) persist() {
    w := new(bytes.Buffer)
    e := labgob.NewEncoder(w)
    e.Encode(rf.currentTerm)
    e.Encode(rf.votedFor)
    e.Encode(rf.log)
    rf.persister.Save(w.Bytes(), nil)
}
```

- **currentTerm**: آخرین ترم شناخته شده توسط سرور.
- **votedFor**: کاندیدایی که سرور در ترم فعلی به آن رأی داده است.
- **log**: تمامی ورودی‌های لاگ سرور.

این سه مقدار، وضعیت‌های پایداری هستند که برای عملکرد صحیح Raft پس از راه‌اندازی مجدد، حیاتی می‌باشند.

۲.۱ تابع readPersist(data []byte)

این تابع مسئول بازیابی وضعیت پایدار ذخیره شده قبلی است. این تابع در هنگام راه‌اندازی یک نمونه جدید Raft (Make()) فراخوانی می‌شود.

```
func (rf *Raft) readPersist(data []byte) {
    if data == nil || len(data) < 1 {
        return
    }
    r := bytes.NewBuffer(data)
    d := labgob.NewDecoder(r)

```

```
    var term int
    var vote int
    var log []LogEntry

    if d.Decode(&term) != nil ||
        d.Decode(&vote) != nil ||
        d.Decode(&log) != nil {
        return // error reading persist data
    }

    rf.currentTerm = term
    rf.votedFor = vote
    rf.log = log
}
```

این تابع داده‌های بایت دریافت شده را از حالت کدگذاری شده خارج (decode) می‌کند و مقادیر currentTerm، votedFor، و log را به وضعیت داخلی سرور Raft اختصاص می‌دهد.

۲ فراخوانی‌های persist() در کد

برای اطمینان از پایداری وضعیت Raft، تابع persist() باید در هر زمانی که یکی از وضعیت‌های پایدار (currentTerm، votedFor، log) تغییر می‌کند، فراخوانی شود. نقاط اصلی فراخوانی persist() عبارتند از:

۱.۲ در RequestVote Handler:

هنگامی که یک سرور به یک کاندیدا رأی می‌دهد، votedFor آن تغییر می‌کند و این تغییر باید پایدار شود.

```
func (rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply)
   {
    // ... (code omitted for brevity)
    reply.VoteGranted = true
    rf.votedFor = args.CandidateId
    rf.persist() // Persist after voting
```

```
}
```

۲.۲ در AppendEntries Handler:

هنگامی که لاگ سرور (پیرو) در پاسخ به پیام AppendEntries رهبر تغییر می‌کند، باید لاگ جدید پایدار شود. همچنین، اگر ترم پیرو به‌روز شود، currentTerm نیز باید پایدار شود. (توجه: در پیاده‌سازی ارائه شده، defer rf.persist() در ابتدای تابع AppendEntries وجود دارد که هرگونه تغییر در log یا currentTerm را پایدار می‌کند.)

```
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *
   AppendEntriesReply) {
    // ... (code omitted for brevity)
    defer rf.persist() // Persist at the end of the handler

    // ... (log modification logic)
}
```

۳.۲ در startElection():

وقتی یک سرور انتخابات را آغاز می‌کند، currentTerm و votedFor خود را افزایش/تغییر می‌دهد که این تغییرات نیاز به پایداری دارند.

```
func (rf *Raft) startElection() {
    rf.mu.Lock()
    rf.currentTerm++
    rf.votedFor = rf.me
    rf.voteCount = 1
    rf.state = Candidate
    rf.persist() // Persist after changing term and voting
    rf.mu.Unlock()
    // ... (code omitted for brevity)
}
```

۴.۲ در updateTerm(term int):

این تابع کمکی هر زمان که ترم یک سرور به‌روز می‌شود، فراخوانی می‌شود. از آنجایی که currentTerm یک وضعیت پایدار است، این تغییر باید بلافاصله پایدار شود.

```
func (rf *Raft) updateTerm(term int) {
    if term > rf.currentTerm {
        rf.currentTerm = term
        rf.votedFor = NotVoted
        rf.state = Follower
        rf.persist() // Persist after updating term
    }
}
```

۵.۲ در Start(command interface):

وقتی یک رهبر دستور جدیدی را به لاگ خود اضافه می‌کند، این تغییر در لاگ باید پایدار شود.

```
func (rf *Raft) Start(command interface{}) (int, int, bool) {
    rf.mu.Lock()
    defer rf.mu.Unlock()
    defer rf.persist() // Persist after modifying the log

    term, isLeader := rf.currentTerm, rf.state == Leader
    if !isLeader {
        return -1, -1, false
    }
    rf.log = append(rf.log, LogEntry{Command: command, Term: term})

    index := len(rf.log) - 1
    // ... (code omitted for brevity)
}
```

این فراخوانی‌های استراتژیک persist() تضمین می‌کنند که وضعیت حیاتی Raft حتی در صورت خرابی سیستم، حفظ شده و قابلیت بازیابی کامل را فراهم می‌کند.

۳ تست قدم 3C

تست‌های این قسمت را یک‌بار به صورت عادی و یک‌بار با race- اجرا کردیم و در هر دو حالت تست‌ها قبول شدند. این تست‌ها هر کدام ۱۰ بار اجرا شدند تا از درست بودنشان اطمینان حاصل کنیم.

```
ali@LAPTOP-4CACSST1:/mnt/c/Users/ALI/OneDrive/Desktop/university/Project/Distributed-Systems/Distributed
-Systems/Project-3/6.5840/src/raft1$ go test -run 3C
Test (3C): basic persistence (reliable network)...
  ... Passed --  time  4.6s #peers 3 #RPCs    66 #Ops    0
Test (3C): more persistence (reliable network)...
  ... Passed --  time 12.9s #peers 5 #RPCs   374 #Ops    0
Test (3C): partitioned leader and one follower crash, leader restarts (reliable network)...
  ... Passed --  time  1.8s #peers 3 #RPCs    33 #Ops    0
Test (3C): Figure 8 (reliable network)...
  ... Passed --  time 30.3s #peers 5 #RPCs   729 #Ops    0
Test (3C): unreliable agreement (unreliable network)...
  ... Passed --  time  1.7s #peers 5 #RPCs  1032 #Ops    0
Test (3C): Figure 8 (unreliable) (unreliable network)...
  ... Passed --  time 33.9s #peers 5 #RPCs  8462 #Ops    0
Test (3C): churn (reliable network)...
  ... Passed --  time 16.1s #peers 5 #RPCs 11124 #Ops    0
Test (3C): unreliable churn (unreliable network)...
  ... Passed --  time 16.1s #peers 5 #RPCs  5406 #Ops    0
PASS
ok      6.5840/raft1    117.292s
```

شکل ۵: نتیجه اجرای تست‌های 3C

```
ali@LAPTOP-4CACSST1:/mnt/c/Users/ALI/OneDrive/Desktop/university/Project/Distributed-Systems/Distributed
-Systems/Project-3/6.5840/src/raft1$ go test -race -run 3C
Test (3C): basic persistence (reliable network)...
  ... Passed --  time  4.4s #peers 3 #RPCs    66 #Ops    0
Test (3C): more persistence (reliable network)...
  ... Passed --  time 12.1s #peers 5 #RPCs   357 #Ops    0
Test (3C): partitioned leader and one follower crash, leader restarts (reliable network)...
  ... Passed --  time  2.0s #peers 3 #RPCs    35 #Ops    0
Test (3C): Figure 8 (reliable network)...
  ... Passed --  time 35.7s #peers 5 #RPCs   869 #Ops    0
Test (3C): unreliable agreement (unreliable network)...
  ... Passed --  time  2.7s #peers 5 #RPCs  1068 #Ops    0
Test (3C): Figure 8 (unreliable) (unreliable network)...
  ... Passed --  time 54.6s #peers 5 #RPCs 12695 #Ops    0
Test (3C): churn (reliable network)...
  ... Passed --  time 16.2s #peers 5 #RPCs  2877 #Ops    0
Test (3C): unreliable churn (unreliable network)...
  ... Passed --  time 16.3s #peers 5 #RPCs  2978 #Ops    0
PASS
ok      6.5840/raft1    145.161s
```

شکل ۶: نتیجه اجرای تست‌های 3C با race-

بخش پنجم

بخش 3D: فشرده‌سازی لاگ (Snapshotting)

در این بخش بر پیاده‌سازی قابلیت Snapshotting تمرکز دارد. هدف اصلی این بخش حل مشکل رشد بی‌رویه لاگ Raft است. با گذشت زمان، لاگ Raft می‌تواند بسیار بزرگ شود که منجر به افزایش مصرف حافظه، کندی در بازسازی وضعیت سرورهای جدید یا سرورهای از کار افتاده، و طولانی شدن زمان همگام‌سازی لاگ‌ها می‌شود. Snapshotting این امکان را فراهم می‌کند که بخش‌های قدیمی و اعمال شده لاگ، فشرده‌سازی شده و به صورت یک "عکس لحظه‌ای" از وضعیت سیستم در یک نقطه زمانی خاص ذخیره شوند. سپس، بخش‌های فشرده‌سازی شده از لاگ حذف می‌شوند و فضای حافظه آزاد می‌شود. این کار باعث بهبود کارایی و مدیریت منابع در Raft می‌شود.

۱ توابع و تغییرات کلیدی اعمال شده

برای پیاده‌سازی Snapshotting، تغییرات و توابع جدیدی به کد Raft اضافه شده‌اند که در ادامه به تفکیک توضیح داده می‌شوند:

۱.۱ فیلدهای جدید در Raft Struct

فیلدهای زیر برای مدیریت Snapshot به ساختار Raft اضافه شده‌اند:

- **lastIncludedIndex int**: اندیس آخرین ورودی لاگ که در Snapshot فعلی گنجانده شده است.
- **lastIncludedTerm int**: ترم مربوط به lastIncludedIndex.
- **snapshot []byte**: بایت‌های خام Snapshot (وضعیت فشرده شده سرویس). این فیلد برای نگهداری Snapshot در حافظه و ارسال آن به پیروان استفاده می‌شود.

```
type Raft struct {
    // ... (existing fields)
    // for snapshotting
    lastIncludedIndex int
    lastIncludedTerm  int
    snapshot          []byte // for caching the snapshot
```

```
}
```

۲.۱ تغییرات در persist()

تابع persist() که مسئول ذخیره وضعیت پایدار Raft بود، اکنون علاوه بر currentTerm، votedFor، و log، دو فیلد جدید lastIncludedIndex و lastIncludedTerm را نیز ذخیره می‌کند. همچنین، خود بایت‌های Snapshot (rf.snapshot) نیز به persister ارسال می‌شوند تا به صورت پایدار ذخیره شوند.

```
func (rf *Raft) persist() {
    w := new(bytes.Buffer)
    e := labgob.NewEncoder(w)
    e.Encode(rf.currentTerm)
    e.Encode(rf.votedFor)
    e.Encode(rf.log)
    e.Encode(rf.lastIncludedIndex) // New
    e.Encode(rf.lastIncludedTerm)  // New
    rf.persister.Save(w.Bytes(), rf.snapshot) // snapshot added
}
```

۳.۱ تغییرات در readPersist()

تابع readPersist() که وضعیت پایدار را بازیابی می‌کند، اکنون فیلدهای جدید lastIncludedIndex و lastIncludedTerm را نیز از داده‌های پایدارسازی شده می‌خواند. پس از بازیابی، commitIndex و lastApplied سرور به lastIncludedIndex تنظیم می‌شوند، زیرا این نقاط نمایانگر آغاز لاگ فعال پس از اعمال Snapshot هستند.

```
func (rf *Raft) readPersist(data []byte) {
    if len(data) < 1 {
        return
    }
    r := bytes.NewBuffer(data)
    d := labgob.NewDecoder(r)

    var term int
```

```
    var vote int
    var log []LogEntry
    var lastIncludedIndex int // New
    var lastIncludedTerm int  // New

    if d.Decode(&term) != nil ||
        d.Decode(&vote) != nil ||
        d.Decode(&log) != nil ||
        d.Decode(&lastIncludedIndex) != nil || // New
        d.Decode(&lastIncludedTerm) != nil {   // New
        return // error reading persist data
    }

    rf.currentTerm = term
    rf.votedFor = vote
    rf.log = log
    rf.lastIncludedIndex = lastIncludedIndex // New
    rf.lastIncludedTerm = lastIncludedTerm   // New

    // **reset volatile state to snapshot point** (New)
    rf.commitIndex = lastIncludedIndex
    rf.lastApplied = lastIncludedIndex
}
```

۴.۱ تابع جدید applySnapshot()

این تابع یک پیام ApplyMsg از نوع Snapshot را از طریق کانال applyCh به سرویس بالاسری ارسال می‌کند. این پیام حاوی بایت‌های Snapshot و اطلاعات lastIncludedIndex/lastIncludedTerm است تا سرویس بتواند وضعیت خود را از Snapshot بازیابی کند.

```
func (rf *Raft) applySnapshot() {
    rf.applyCh <- raftapi.ApplyMsg{
```

```
        SnapshotValid: true,
        Snapshot:      rf.snapshot,
        SnapshotTerm:  rf.lastIncludedTerm,
        SnapshotIndex: rf.lastIncludedIndex,
    }
}
```

۵.۱ تابع جدید Snapshot()

این تابع توسط سرویس Raft (مثلاً سرور Key-Value) فراخوانی می‌شود تا به Raft اطلاع دهد که یک Snapshot جدید تا اندیس index ایجاد شده است. وظایف این تابع:

- اعتبار سنجی index (باید بزرگتر از lastIncludedIndex قبلی و کوچکتر یا مساوی commitIndex و lastApplied باشد).
- کوتاه کردن (trim) لاگ rf.log تا فقط ورودی‌های جدیدتر از index را شامل شود. اولین ورودی لاگ (rf.log[0]) پس از کوتاه شدن، تبدیل به یک ورودی ساختگی با ترم lastIncludedTerm و nil به عنوان دستور می‌شود که به lastIncludedIndex نگاشت می‌شود.
- به‌روزرسانی lastIncludedIndex، lastIncludedTerm، commitIndex، lastApplied و snapshot.
- فراخوانی persist() برای ذخیره وضعیت جدید و Snapshot.

```
func (rf *Raft) Snapshot(index int, snapshot []byte) {
    rf.mu.Lock()
    defer rf.mu.Unlock()

    if index < rf.lastIncludedIndex || index > rf.commitIndex || index > rf.
       lastApplied {
        // This panic indicates an unexpected call pattern from the service.
        panic("Raft would never be asked to snapshot an index that is less
           than lastIncludedIndex or greater than commitIndex or lastApplied
           ")
```

```
    }

    // Calculate array index relative to rf.log (which starts from rf.
       lastIncludedIndex)
    arrayIndex := index - rf.lastIncludedIndex
    term := rf.log[arrayIndex].Term

    // Create a new log slice, effectively trimming the old log
    newLog := make([]LogEntry, len(rf.log)-(arrayIndex))
    newLog[0] = LogEntry{Term: term, Command: nil} // Initial empty log
       entry after snapshot
    if arrayIndex+1 < len(rf.log) {
        copy(newLog[1:], rf.log[arrayIndex+1:])
    }

    rf.log = newLog

    rf.lastIncludedIndex = index
    rf.lastIncludedTerm = term
    rf.commitIndex = max(rf.commitIndex, index) // Ensure commitIndex is at
       least snapshot index
    rf.lastApplied = max(rf.lastApplied, index) // Ensure lastApplied is at
       least snapshot index

    rf.snapshot = snapshot
    rf.persist() // Persist the updated state including the new snapshot
}
```

۶.۱ ساختارهای InstallSnapshotArgs و InstallSnapshotReply

این ساختارها برای پیام‌های RPC مربوط به Snapshot استفاده می‌شوند:

```
type InstallSnapshotArgs struct {
    Term           int    // leader's term
    LeaderId       int    // so follower can redirect clients
    LastIncludedIndex int // index of the last included entry in the
       snapshot
    LastIncludedTerm  int // term of the last included entry in the snapshot
    Snapshot          []byte // raw bytes of the snapshot
}type InstallSnapshotReply struct {
    Term int // currentTerm, for leader to update itself
}
```

۷.۱ تابع جدید InstallSnapshot()

این هندلر RPC در سرورهای پیرو اجرا می‌شود تا Snapshot ارسال شده از رهبر را دریافت و اعمال کند. وظایف این تابع:

- اعتبار سنجی ترم (اگر ترم رهبر کمتر از ترم فعلی پیرو باشد، رد می‌شود).
- اگر args.LastIncludedIndex کمتر یا مساوی commitIndex پیرو باشد، به معنی این است که پیرو قبلاً این Snapshot یا لاگ جلوتر را اعمال کرده است، پس رد می‌شود.
- بروزرسانی ترم پیرو، تغییر وضعیت به Follower و تنظیم gotPulse.
- کوتاه کردن لاگ پیرو:
  - اگر Snapshot تمام لاگ موجود پیرو را پوشش می‌دهد یا از آن جلوتر است، لاگ کاملاً حذف شده و با یک ورودی ساختگی بر اساس Snapshot جدید بازسازی می‌شود.
  - اگر Snapshot بخشی از لاگ پیرو را پوشش می‌دهد، بخش‌های قدیمی حذف شده و بخش‌های جدیدتر حفظ می‌شوند.
- به‌روزرسانی lastIncludedIndex، lastIncludedTerm و snapshot پیرو.
- فراخوانی persist() برای ذخیره تغییرات.

- به‌روزرسانی commitIndex و lastApplied (که باید حداقل args.LastIncludedIndex باشند).
- فراخوانی go rf.applySnapshot() برای اعمال Snapshot به سرویس بالاسری.

```
func (rf *Raft) InstallSnapshot(args *InstallSnapshotArgs, reply *
    InstallSnapshotReply) {
    rf.mu.Lock()
    defer rf.mu.Unlock()

    reply.Term = rf.currentTerm
    if args.Term < rf.currentTerm {
        return
    }

    // If the snapshot is older than what we've already committed, ignore it
    if args.LastIncludedIndex <= rf.commitIndex {
        return
    }

    rf.updateTerm(args.Term) // Potentially update term and become follower
    rf.state = Follower
    rf.gotPulse = true

    // Calculate the point in our current log that is covered by the
        snapshot
    removeUntil := args.LastIncludedIndex - rf.lastIncludedIndex
    if removeUntil >= len(rf.log) {
        // Snapshot covers entire log or beyond; discard log and start fresh
            .
        rf.log = []LogEntry{{Term: args.LastIncludedTerm, Command: nil}}
    } else if removeUntil > 0 {
        // Snapshot covers part of log; keep entries after LastIncludedIndex
```

```
            .
        // The first entry (index 0) of newLog corresponds to
           lastIncludedIndex.
        newLog := make([]LogEntry, len(rf.log)-removeUntil)
        newLog[0] = LogEntry{Term: args.LastIncludedTerm, Command: nil}
        copy(newLog[1:], rf.log[removeUntil+1:])
        rf.log = newLog
    } // else: snapshot aligns with current log start, no change needed

    rf.lastIncludedIndex = args.LastIncludedIndex
    rf.lastIncludedTerm = args.LastIncludedTerm
    rf.snapshot = make([]byte, len(args.Snapshot))
    copy(rf.snapshot, args.Snapshot)
    rf.persist() // Persist the new state including the snapshot

    // Update commitIndex and lastApplied to reflect the new state after
       snapshot
    rf.commitIndex = max(rf.commitIndex, args.LastIncludedIndex)
    rf.lastApplied = max(rf.lastApplied, args.LastIncludedIndex)

    go rf.applySnapshot() // Apply the snapshot to the state machine
}
```

۸.۱ تابع جدید sendInstallSnapshot

این تابع مسئول ارسال RPC InstallSnapshot به یک سرور خاص است.

```
func (rf *Raft) sendInstallSnapshot(server int, args *InstallSnapshotArgs) {
    reply := &InstallSnapshotReply{}
    ok := rf.peers[server].Call("Raft.InstallSnapshot", args, reply)
    if !ok || rf.killed() {
        return
```

```
    }

    rf.mu.Lock()
    defer rf.mu.Unlock()
    rf.updateTerm(reply.Term)
    rf.nextIndex[server] = args.LastIncludedIndex + 1
}
```

۹. تغییرات در AppendEntries() هندلر

در این هندلر، منطق پاسخ به رهبر در مورد عدم تطابق لاگ بهبود یافته است. اگر PrevLogIndex ارسالی از رهبر قبل از lastIncludedIndex پیرو باشد (یعنی لاگ پیرو از Snapshot رهبر جلوتر است و نیاز به همگام‌سازی از طریق Snapshot دارد)، پیرو reply.XLen را به 1- تنظیم می‌کند تا به رهبر اطلاع دهد که Snapshot لازم است.

```
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *
   AppendEntriesReply) {
    // ... (existing code)

    // our snapshot is further than lastIncludedIndex
    // so we send -1 so the leader send its snapshot to make us match
    if args.PrevLogIndex - rf.lastIncludedIndex < 0 {
        reply.XLen = -1 // Signal to leader that follower is too far behind,
            needs snapshot
        return
    }
    // ... (existing log matching and appending logic)
}
```

۹.۱ تغییرات در sendAppendEntries (منطق رهبر)

هنگامی که یک رهبر پاسخ AppendEntriesReply را دریافت می‌کند و Success برابر false است، رهبر تلاش می‌کند nextIndex برای آن پیرو را اصلاح کند. اگر reply.XLen برابر 1- باشد یا nextIndex محاسبه شده

کمتر یا مساوی lastIncludedIndex رهبر باشد، رهبر تصمیم می‌گیرد به جای AppendEntries، یک RPC InstallSnapshot به آن پیرو ارسال کند. این تضمین می‌کند که پیروان عقب‌مانده با استفاده از Snapshot به سرعت به روز شوند.

```
func (rf *Raft) sendAppendEntries(server int, args *AppendEntriesArgs) {
    // ... (existing code)

    if reply.Success {
        // ... (existing success logic)
    } else if args.Term >= reply.Term {
        // Log mismatch handling (existing logic modified for snapshot awareness)
        if reply.XLen <= args.PrevLogIndex { // This condition implies the follower's log is shorter or has a gap
            rf.nextIndex[server] = reply.XLen
        } else if rf.hasTerm(reply.ConflictTerm) { // Find the last index for the conflicting term
            rf.nextIndex[server] = rf.findLastEntryIndex(reply.ConflictTerm) + 1
        } else { // If the conflicting term is not in leader's log, set nextIndex to ConflictIndex
            rf.nextIndex[server] = reply.ConflictIndex
        }

        // If nextIndex falls at or before the last included index, send snapshot
        if rf.nextIndex[server] <= rf.lastIncludedIndex {
            snapshotArgs := &InstallSnapshotArgs{
                Term:          rf.currentTerm,
                LeaderId:      rf.me,
                LastIncludedIndex: rf.lastIncludedIndex,
                LastIncludedTerm:  rf.lastIncludedTerm,
```

```
                Snapshot:          rf.snapshot,
            }
            go rf.sendInstallSnapshot(server, snapshotArgs) // Send snapshot
                instead of AppendEntries
            return
        }

        // ... (re-send AppendEntries for the corrected nextIndex)
        // Corrected prevLogIndex and prevLogTerm
        prevLogIndex, prevLogTerm := rf.nextIndex[server]-1, rf.log[rf.
            nextIndex[server]-rf.lastIncludedIndex-1].Term
        sendEntries := make([]LogEntry, len(rf.log[rf.nextIndex[server]-rf.
            lastIncludedIndex:]))
        copy(sendEntries, rf.log[rf.nextIndex[server]-rf.lastIncludedIndex
            :])

        newArgs := &AppendEntriesArgs{
            Term:         rf.currentTerm,
            LeaderId:     rf.me,
            Entries:      sendEntries,
            PrevLogIndex: prevLogIndex,
            PrevLogTerm:  prevLogTerm,
            LeaderCommit: rf.commitIndex,
        }
        go rf.sendAppendEntries(server, newArgs)
    }
}
```

۱۰.۱ تغییرات در Start(command interface)

در تابع Start که توسط سرویس بالاسری فراخوانی می‌شود تا یک دستور جدید را به لاگ اضافه کند، منطق ارسال InstallSnapshot به پیروان، مشابه sendAppendEntries، اضافه شده است. این تضمین می‌کند که حتی در زمان شروع یک دستور جدید، پیروان بسیار عقب‌مانده به درستی همگام‌سازی شوند.

```
func (rf *Raft) Start(command interface{}) (int, int, bool) {
    rf.mu.Lock()
    defer rf.mu.Unlock()
    defer rf.persist() // Persist after adding command

    term, isLeader := rf.currentTerm, rf.state == Leader
    if !isLeader {
        return -1, -1, false
    }
    rf.log = append(rf.log, LogEntry{Command: command, Term: term})

    index := len(rf.log) + rf.lastIncludedIndex - 1 // Correct index
       calculation for new entry

    for i := range rf.peers {
        if i == rf.me {
            continue
        }

        // If follower is too far behind, send snapshot
        if rf.nextIndex[i] <= rf.lastIncludedIndex {
            args := &InstallSnapshotArgs{
                Term:          rf.currentTerm,
                LeaderId:      rf.me,
                LastIncludedIndex: rf.lastIncludedIndex,
                LastIncludedTerm:  rf.lastIncludedTerm,
                Snapshot:          rf.snapshot,
```

```
            }
            go rf.sendInstallSnapshot(i, args)
            continue // Skip AppendEntries for this peer in this round
        }
        // ... (existing AppendEntries preparation and send logic)
    }
    return index, term, isLeader
}
```

۱۱.۱ تغییرات در Make()

تابع Make() مسئول مقداردهی اولیه یک سرور Raft است. تغییرات برای Snapshotting شامل:

- مقداردهی اولیه lastIncludedIndex و lastIncludedTerm به ۰.
- مقداردهی اولیه rf.log با یک LogEntry ساختگی برای اندیس ۰، که نشان دهنده وضعیت اولیه لاگ قبل از هر Snapshot است.
- پس از readPersist()، Snapshot ذخیره شده قبلی با persister.ReadSnapshot() خوانده می‌شود. اگر Snapshot موجود باشد، آن را به rf.snapshot کپی کرده، دوباره persist() را فراخوانی می‌کند تا وضعیت کامل (شامل Snapshot) ذخیره شود، و سپس go rf.applySnapshot() را برای اعمال Snapshot به سرویس بالاسری اجرا می‌کند. این امر به سرویس اجازه می‌دهد تا وضعیت خود را از Snapshot بارگذاری کند.

```
func Make(peers []*labrpc.ClientEnd, me int,
    persister *tester.Persister, applyCh chan raftapi.ApplyMsg) raftapi.Raft
        {
    rf := &Raft{}
    // ... (existing initializations)

    rf.currentTerm = InitialTerm
    rf.lastIncludedIndex = 0 // Initialized for snapshotting
```

```
    rf.lastIncludedTerm = 0  // Initialized for snapshotting

    rf.log = make([]LogEntry, 0)
    rf.log = append(rf.log, LogEntry{Term: 0, Command: nil}) // Initial
        empty log entry at index 0 (log[0] corresponds to lastIncludedIndex)

    // initialize from state persisted before a crash
    rf.readPersist(persister.ReadRaftState())

    // If there's a stored snapshot, load it and apply it
    snapshot := persister.ReadSnapshot()
    if len(snapshot) != 0 {
        rf.snapshot = make([]byte, len(snapshot))
        copy(rf.snapshot, snapshot)
        rf.persist() // Re-persist to ensure the snapshot is fully part of
            the persisted state
        go rf.applySnapshot()
    }

    // ... (goroutine starts)
    return rf
}
```

۱۲.۱ تغییرات در applier()

تابع applier که مسئول اعمال دستورات به ماشین حالت سرویس است، اکنون محاسبه اندیس‌های لاگ را با در نظر گرفتن lastIncludedIndex انجام می‌دهد. این تضمین می‌کند که حتی پس از کوتاه شدن لاگ، اندیس‌های صحیح به سرویس ارسال شوند.

```
func (rf *Raft) applier() {
    for !rf.killed() {
        rf.mu.Lock()
```

```
        rf.applierCond.Wait()
        commitIndex, lastApplied := rf.commitIndex, rf.lastApplied
        // Adjust log slice indices based on lastIncludedIndex
        entries := make([]LogEntry, commitIndex-lastApplied)
        copy(entries, rf.log[lastApplied-rf.lastIncludedIndex+1:commitIndex-
           rf.lastIncludedIndex+1])
        startIndex := lastApplied + 1
        rf.lastApplied = commitIndex
        rf.mu.Unlock()

        for i, entry := range entries {
            rf.applyCh <- raftapi.ApplyMsg{
                CommandValid: true,
                Command:      entry.Command,
                CommandIndex: startIndex + i,
            }
        }
    }
}
```

۱۳.۱ تغییرات در getLastLogIndexAndTerm()

این تابع کمکی اکنون lastLogIndex را با جمع کردن lastIncludedIndex و طول لاگ (منهای ۱) محاسبه می‌کند، که نمایانگر اندیس جهانی (نه اندیس در آرایه rf.log) آخرین ورودی لاگ است.

```
func (rf *Raft) getLastLogIndexAndTerm() (int, int) {
    lastLogIndex := rf.lastIncludedIndex + len(rf.log) - 1 // Adjusted for
       snapshot offset
    lastLogTerm := rf.log[len(rf.log)-1].Term
    return lastLogIndex, lastLogTerm
}
```

۱۴.۱ تغییرات در sendHeartbeat() (رهبر)

مانند sendAppendEntries و Start، اگر nextIndex برای یک پیرو خاص، کمتر یا مساوی lastIncludedIndex رهبر باشد، رهبر به جای ارسال AppendEntries خالی (Heartbeat)، یک RPC InstallSnapshot را ارسال می‌کند تا آن پیرو را به روز کند.

```
func (rf *Raft) sendHeartbeat() {
    for i := range rf.peers {
        if i == rf.me {
            continue
        }

        // If follower is too far behind, send snapshot
        if rf.nextIndex[i] <= rf.lastIncludedIndex {
            args := &InstallSnapshotArgs{
                Term:          rf.currentTerm,
                LeaderId:      rf.me,
                LastIncludedIndex: rf.lastIncludedIndex,
                LastIncludedTerm:  rf.lastIncludedTerm,
                Snapshot:          rf.snapshot,
            }
            go rf.sendInstallSnapshot(i, args)
            continue
        }
        // ... (existing AppendEntries (heartbeat) send logic)
    }
}
```

این تغییرات جامع، Raft را قادر می‌سازد تا Snapshotها را ایجاد، ذخیره، بازیابی و به پیروان ارسال کند، که برای حفظ کارایی و پایداری در سیستم‌های طولانی‌مدت حیاتی است.

۲ تست قدم 3D

تست‌های این قسمت را یک‌بار به صورت عادی و یک‌بار با race- اجرا کردیم و در هر دو حالت تست‌ها قبول شدند.

```
ali@LAPTOP-4CACSST1:/mnt/c/Users/ALI/OneDrive/Desktop/university/Project/Distributed-Systems/Distributed
-Systems/Project-3/6.5840/src/raft1$ go test -run 3D
Test (3D): snapshots basic (reliable network)...
  ... Passed --  time  4.7s #peers 3 #RPCs   536 #Ops    0
Test (3D): install snapshots (disconnect) (reliable network)...
  ... Passed --  time 42.2s #peers 3 #RPCs  1563 #Ops    0
Test (3D): install snapshots (disconnect) (unreliable network)...
  ... Passed --  time 63.3s #peers 3 #RPCs  2191 #Ops    0
Test (3D): install snapshots (crash) (reliable network)...
  ... Passed --  time 30.5s #peers 3 #RPCs  1252 #Ops    0
Test (3D): install snapshots (crash) (unreliable network)...
  ... Passed --  time 35.1s #peers 3 #RPCs  1244 #Ops    0
Test (3D): crash and restart all servers (unreliable network)...
  ... Passed --  time  8.5s #peers 3 #RPCs   248 #Ops    0
Test (3D): snapshot initialization after crash (unreliable network)...
  ... Passed --  time  4.8s #peers 3 #RPCs   120 #Ops    0
PASS
ok      6.5840/raft1    189.043s
```

شکل ۷: نتیجه اجرای تست‌های 3D

```
ali@LAPTOP-4CACSST1:/mnt/c/Users/ALI/OneDrive/Desktop/university/Project/Distributed-Systems/Distributed
-Systems/Project-3/6.5840/src/raft1$ go test -race -run 3D
Test (3D): snapshots basic (reliable network)...
  ... Passed --  time  4.8s #peers 3 #RPCs   479 #Ops    0
Test (3D): install snapshots (disconnect) (reliable network)...
  ... Passed --  time 40.6s #peers 3 #RPCs  1415 #Ops    0
Test (3D): install snapshots (disconnect) (unreliable network)...
  ... Passed --  time 63.2s #peers 3 #RPCs  2370 #Ops    0
Test (3D): install snapshots (crash) (reliable network)...
  ... Passed --  time 28.8s #peers 3 #RPCs  1138 #Ops    0
Test (3D): install snapshots (crash) (unreliable network)...
  ... Passed --  time 38.0s #peers 3 #RPCs  1417 #Ops    0
Test (3D): crash and restart all servers (unreliable network)...
  ... Passed --  time 12.6s #peers 3 #RPCs   360 #Ops    0
Test (3D): snapshot initialization after crash (unreliable network)...
  ... Passed --  time  2.6s #peers 3 #RPCs    74 #Ops    0
PASS
ok      6.5840/raft1    191.589s
```

شکل ۸: نتیجه اجرای تست‌های 3D با race-

مراجع