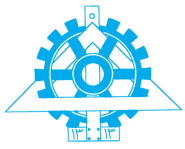
سیستم های توزیع شده

بهار ۲۰۲۵



پروژه سوم

تاریخ تحویل ۳۰ خرداد

تعریف پروژه: پیاده‌سازی الگوریتم اجماع Raft

در این پروژه، شما باید الگوریتم Raft را پیاده‌سازی کنید؛ این الگوریتم یک پروتکل تکرارشده (Replicated) برای مدیریت لاگ در ماشین‌های حالت (State) است.

یک سرویس تکرارشده از طریق نگهداری نسخه‌های کامل از وضعیت (داده‌ها) در چندین سرور کپی، تحمل‌پذیری در برابر خرابی (Fault tolerance) را فراهم می‌کند. فرآیند تکرار باعث می‌شود سرویس حتی در صورت خرابی برخی از سرورها به‌دلیل توقف در عملکرد Crash یا ناپایداری شبک) همچنان فعال باقی بماند. با این حال، چالش اصلی این است که در هنگام وقوع ازکارافتادگی، سازگاری (Consistency) بین نسخه‌ها حفظ گردد.

الگوریتم Raft با سازمان دهی درخواست‌های مشتری به صورت یک توالی مشخص به‌نام «لاگ» (Log) اطمینان می‌دهد که همه سرورها نسخه‌ی یکسانی از لاگ را می‌بینند. هر سرور،دستورات (Commands) را به ترتیب در لاگ خود اعمال می‌کند و بنابراین، همگی وضعیت یکسانی را نگه می‌دارند.

اگر یک سرور از کار بیفتد و سپس بهبود یابد، Raft مسئول هماهنگ‌سازی و به‌روزرسانی لاگ آن خواهد بود. تا زمانی که اکثریت سرورها فعال باشند و بتوانند با هم ارتباط برقرار کنند، سیستم به کار خود ادامه می‌دهد. اگر اکثریتی وجود نداشته باشد، سیستم متوقف می‌شود اما پس از برقراری ارتباط مجدد، ادامه‌ی فرآیند از آخرین نقطه ممکن پیگیری خواهد شد.

در این پروژه، شما باید الگوریتم Raft را به‌صورت یک شی Object در زبان Go و با تابع‌های مرتبط پیاده‌سازی کنید، به‌گونه‌ای که قابل استفاده در یک ماژول بزرگ‌تر باشد.

مجموعه‌ای از نمونه‌های Raft باید از طریق RPC با یکدیگر ارتباط برقرار کرده و لاگ‌ها را هماهنگ نگه دارند. رابط Interface ماژول Raft شما باید از دنباله‌ای بی‌نهایت از دستورات شماره‌گذاری‌شده (ورودی‌ها) پشتیبانی کند. این ورودی‌ها به‌صورت اعدادی با شماره‌ی ایندکس Index ذخیره می‌شوند. هر ورودی که دارای ایندکس معتبر باشد، باید وارد لاگ شده و سپس به سیستم بزرگ‌تر تحویل داده شود.

لازم است تا طراحی خود را به خصوص با تمرکز ویژه بر شکل دوم از مقاله‌ی Raft (نسخه توسعه‌یافته-Extended Version) انجام دهید که در به همراه این دستورالعمل در فایل فشرده موجود است. این مقاله به‌طور دقیق اجزای

کلیدی طراحی از جمله ذخیره‌سازی پایدار، بازیابی پس از خرابی، و اعمال تغییرات پیکربندی در سیستم را توضیح داده است.

شروع کار با پروژه

کد اولیه به‌صورت فایل raft.go به همراه مجموعه‌ای از تست‌ها در فایل raft_test.go برای ارزیابی صحت پیاده‌سازی در اختیار شما قرار داده شده است. شما باید از این تست‌ها برای هدایت روند پیاده‌سازی خود و همچنین صحت عملکرد در سناریوهای مختلف استفاده نمایید. موفقیت در گذراندن این تست‌ها، معیار اصلی ارزیابی نهایی پروژه شما خواهد بود.

در زمان ارزیابی، تست‌ها بدون استفاده از پرچم race- اجرا خواهند شد. با این حال، اکیدا توصیه می‌شود که خودتان کد را با استفاده از پرچم race- بررسی نمایید تا مطمئن شوید پیاده‌سازی‌تان دارای شرایط رقابتی (race conditions) نیست.

برای شروع کار لازم است تا ابتدا پوشه کدها در پروژه‌های پیشین (6.5840) بروید و سپس با استفاده از دستور

```
$ cd src/raft1
```

به کدهای موردنیاز این پروژه دسترسی پیدا کنید. سپس با اجرای دستور go test باید خروجی زیر را بگیرید:

```
$ go test
Test (3A): initial election (reliable network)...
Fatal: expected one leader, got none
--- FAIL: TestInitialElection3A (4.90s)
Test (3A): election after network failure (reliable network)...
Fatal: expected one leader, got none
--- FAIL: TestReElection3A (5.05s)
...
$
```

شکل ۱

برای پیاده‌سازی الگوریتم Raft، لازم است تکه کد مربوطه را به فایل raft.go اضافه نمایید. در این فایل، کد اولیه و همچنین مثال‌هایی از نحوه ارسال و دریافت RPC وجود دارد.

پیاده‌سازی شما باید از interface مشخص‌شده در زیر پشتیبانی کند، چرا که این interface توسط برنامه تست مورد استفاده قرار می‌گیرد. جزئیات بیشتر در کامنت‌های فایل raft.go ارائه شده است.

```
// create a new Raft server instance:
rf := Make(peers, me, persister, applyCh)

// start agreement on a new log entry:
rf.Start(command interface{}) (index, term, isleader)

// ask a Raft for its current term, and whether it thinks it is leader
rf.GetState() (term, isLeader)

// each time a new entry is committed to the log, each Raft peer
// should send an ApplyMsg to the service (or tester).
type ApplyMsg
```

شکل ۲

سرویسی که می‌سازید باید از تابع Make برای ایجاد نمونه‌ای از Raft استفاده کند.

```
Make(peers []*labrpc.ClientEnd, me int, persister *Persister,
    applyCh chan ApplyMsg)
```

- peers آرایه‌ای از کلاینت‌های RPC به صورت "peers []*labrpc" است که نمایانگر تمام سرورهای Raft در کلاستر (شامل سرور فعلی) هستند. ماژول Raft شما از این آرایه برای ارسال RPC به سایر سرورها استفاده خواهد کرد.
- me ایندکس سرور فعلی در آرایه peers است.
- persister یک شی برای ذخیره‌سازی وضعیت پایدار Raft مانند votedFor، currentTerm و log است تا در صورت ری‌استارت شدن سرور، این اطلاعات بازیابی شوند.
- applyCh یک کانال Channel است که ماژول Raft شما باید از آن برای ارسال پیام‌های ApplyMsg شامل آن entry از طریق این کانال ارسال شود.

تابع Start(command) از Raft می‌خواهد که فرآیند افزودن این فرمان به لاگ تکرارشونده را آغاز کند. این تابع باید بلافاصله خروجی دهد و منتظر کامل شدن فرآیند ثبت لاگ نماند. سرویس مورد انتظاری که شما پیاده می‌نمایید باید برای هر لاگ جدید که ثبت می‌شود، یک پیام از نوع ApplyMsg به کانال applyCh ارسال کند.

```
Start(command interface{}) (index int, term int, isLeader bool*)
```

توجه داشته باشید که حتی اگر این سرور رهبر باشد، هیچ تضمینی وجود ندارد که این دستور در نهایت تثبیت (Commit) شود (مثلاً اگر رهبر قبل از تثبیت، ارتباط خود را با اکثریت از دست بدهد).

```
GetState() (term int, isLeader bool*)
```

این متد، ترم فعلی سرور و اینکه آیا سرور خود را رهبر می‌داند یا خیر، بازمی‌گرداند. تستر از این تابع برای بررسی وضعیت سرورها استفاده می‌کند.

```
ApplyMsg
```

این ساختار (Struct) برای ارسال دستورات تثبیت شده از ماژول Raft به ماشین حالت استفاده می‌شود. جزئیات فیلدهای آن مانند CommandIndex، Command و CommandValid در فایل raft.go تعریف شده است.
فایل raft.go شامل مثال‌هایی از ارسال و دریافت پیام‌های RPC مانند RequestVoteReply، RequestVoteArgs و sendRequestVote است.
نمونه‌های Raft شما باید از طریق پکیج labrpc موجود در مسیر src/labrpc و پیاده‌شده در پروژه اول با هم ارتباط برقرار کنند.
برنامه تست از نسخه اصلی labrpc توسعه داده شده در پروژه اول استفاده می‌کند. بنابراین حتی اگر شما برای شبیه‌سازی خطاهای شبکه آن را موقتاً تغییر دهید، در نهایت باید کد نهایی شما با نسخه اصلی labrpc سازگار باشد، چرا که ارزیابی شما بر همین اساس صورت خواهد گرفت.
نمونه‌های Raft شما فقط باید از RPC برای ارتباط استفاده کنند. اجازه استفاده از متغیرهای اشتراکی بین نمونه‌ها را ندارید.

قسمت 3A: انتخاب رهبر

در این بخش، باید مکانیزم انتخاب رهبر در الگوریتم Raft را پیاده سازی کنید. هدف اصلی این بخش، انتخاب یک رهبر در شرایط بدون ازکارافتادگی و سپس جایگزینی رهبر جدید در صورت از کار افتادن رهبر فعلی یا قطع ارتباط آن است. همچنین باید پیام‌های AppendEntries بدون محتوای لاگ را نیز برای ارسال ضربان قلب (heartbeat) پیاده سازی کنید.
نکته راهنما:

- اجرای مستقیم پیاده سازی شما از Raft ممکن نیست؛ باید آن را با دستور زیر از طریق برنامه تست اجرا کنید:

```
$ go test -run 3A
```

- تصویر ۲ در مقاله به‌ویژه بخش ارسال و دریافت RequestVote RPCs و همچنین قوانین سرورها برای رای‌گیری و وضعیت (State) مربوط به انتخاب رهبر را با دقت دنبال نمایید.

- وضعیت انتخاب رهبر را که در تصویر ۲ آمده است برای ساختار Raft در فایل raft.go اضافه نمایید. همچنین نیاز است تا ساختاری برای نگهداری هر مدخل (Entry) لاگ تعریف نمایید.
- تابع‌های RequestVoteArgs و RequestVoteReply را کامل کرده و با تابع ()Make یک حلقه پس‌زمینه ایجاد کنید که به‌صورت دوره‌ای رهبر بودن را بررسی کند.
- مطمئن شوید که سرور در صورت عدم دریافت پیام AppendEntries از رهبر، به وضعیت کاندیدا برمی‌گردد و درخواست رأی ارسال می‌کند.
- برنامه تست از شما می‌خواهد که ظرف کمتر از پنج ثانیه، رهبر جدید انتخاب شود.
- بازه زمانی بین ۱۵۰ تا ۳۰۰ میلی ثانیه برای تایمرهای انتخاب مناسب است. اگر همه سرورها در بازه‌های زمانی مشابه‌ای اقدام کنند، احتمال برخورد زیاد می‌شود.
- پیام‌های RPC باید هم برای دریافت رأی و هم برای ارسال heartbeat پیاده سازی شود.
- برای ساخت تایمر از go از time.Sleep() استفاده نکنید. به جای آن از time.Timer یا time.Ticker استفاده نمایید.
- برای عیب یابی می‌توانید با فراخوانی مجدد go test -run 3A نتایج را بررسی کنید.
- از دستور fmt.Println() برای چاپ وضعیت‌ها و کمک به اشکال زدایی استفاده کنید.
- اگر نمونه‌ای از Raft خاموش شود، باید با ()kill متوقف شود. پیام‌های ناخواسته در این حالت می‌توانند باگ ایجاد کنند.
- ساختار پیام‌ها در فایل‌های تست نباید تغییر کند. اگر نیاز به متغیر جدید دارید، در ساختار خودتان اضافه نمایید.
- همه ارتباطات فقط باید از طریق RPC انجام شوند. نباید از متغیرهای مشترک بین نمونه‌ها استفاده کنید.

برای تست نهایی:

```
$ go test -run 3A
Test (3A): initial election (reliable network)...
  ... Passed --   3.6  3   106    0
Test (3A): election after network failure (reliable network)...
  ... Passed --   7.6  3   304    0
Test (3A): multiple elections (reliable network)...
  ... Passed --   8.4  7   954    0
PASS
ok      6.5840/raft1    19.834sak
$
```

شکل ۳

هر خط Passed شامل پنج عدد است:

- مدت زمان اجرای تست (بر حسب ثانیه)
- تعداد کل سرورهای Raft
- تعداد پیام‌های RPC
- حجم کل پیام‌های RPC (بر حسب بایت)
- تعداد ورودی‌های لاگ که ثبت شده‌اند

این اعداد ممکن است با سیستم شما تفاوت داشته باشند، اما برای بررسی صحت تقریبی اجرای تست‌ها مفید هستند. اگر مجموع زمان اجرای همه تست‌ها (در قسمت‌های ۳، ۴ یا ۵) از ۶۰۰ ثانیه بیشتر شود یا هر تست بیش از ۱۲۰ ثانیه طول بکشد، برنامه نمره دهی با شکست مواجه خواهد شد.
در نهایت، تست‌ها بدون پرچم race- اجرا می‌شوند، اما شما باید همیشه مطمئن شوید که برنامه با پرچم race- نیز بدون خطا اجرا می‌شود:

```
$ go test -race -run 3A
```

قسمت3B: ثبت لاگ

در این مرحله باید کد رهبر و پیرو (follower) را برای افزودن ورودی‌های جدید به لاگ پیاده سازی کنید، به‌گونه‌ای که اجرای دستور زیر با موفقیت انجام شود:

```
$ go test -run 3B
```

نکات راهنما:

- لاگ Raft از شماره ۱ شروع می‌شود، ولی برای ساده سازی می‌توانید آن را به صورت صفر مبنا در نظر بگیرید. در این صورت اولین ورودی (در ایندکس ۰) مقدار term 0 خواهد داشت که به شما اجازه می‌دهد اولین پیام AppendEntries شامل $\text{PrevLogIndex} = 0$ باشد.
- اولین هدف شما باید عبور از تست TestBasicAgree3B() باشد. با تابع Start() شروع کنید و کدی برای ارسال و دریافت ورودی‌های لاگ جدید از طریق پیام‌های AppendEntries بنویسید. هر ورودی ثبت شده باید برای همه پیروها ارسال شود و از طریق کانال applyCh در هر پیرو اعمال شود.
- باید محدودیت‌های رای‌گیری را نیز پیاده سازی کنید. برای این بخش به قسمت ۵.۴.۱ مقاله مراجعه کنید.
- ممکن است در کد خود حلقه‌هایی داشته باشید که منتظر رویداد خاصی هستند. این حلقه‌ها را بدون توقف اجرا نکنید، زیرا باعث کندی اجرای تست‌ها می‌شود. برای کنترل بهتر از condition variables استفاده کنید یا در هر دور از حلقه یک مکث کوتاه (مثلا time.Sleep(10 * time.Millisecond) اضافه کنید.
- اگر تستی شکست خورد، فایل raft_test.go را بررسی کرده و ببینید دقیقاً چه چیزی تست می‌شود.

مثال خروجی موفق از تست 3B:

```
$ time go test -run 3B
Test (3B): basic agreement (reliable network)...
  ... Passed --   1.3  3    18    0
Test (3B): RPC byte count (reliable network)...
  ... Passed --   2.8  3    56    0
Test (3B): test progressive failure of followers (reliable network)...
  ... Passed --   5.3  3   188    0
Test (3B): test failure of leaders (reliable network)...
  ... Passed --   6.4  3   378    0
Test (3B): agreement after follower reconnects (reliable network)...
  ... Passed --   5.9  3   176    0
Test (3B): no agreement if too many followers disconnect (reliable network)...
  ... Passed --   4.3  5   288    0
Test (3B): concurrent Start()s (reliable network)...
  ... Passed --   1.5  3    32    0
Test (3B): rejoin of partitioned leader (reliable network)...
  ... Passed --   5.3  3   216    0
Test (3B): leader backs up quickly over incorrect follower logs (reliable network)...
  ... Passed --  12.1  5  1528    0
Test (3B): RPC counts aren't too high (reliable network)...
  ... Passed --   3.1  3   106    0
PASS
ok      6.5840/raft1    48.353s
go test -run 3B  1.37s user 0.74s system 4% cpu 48.865 total
$
```

شکل ۴

در خط "ok 6.5840/raft1 35.557s"، مدت زمان کل اجرای تست‌ها (زمان واقعی یا real) نشان داده می‌شود. عبارت user 0m2.556s نشان می‌دهد که زمان مصرف شده توسط پردازنده (نه در حالت انتظار یا خواب) فقط ۵.۲ ثانیه بوده است.

اگر زمان اجرای تست‌های 3B از ۶۰ ثانیه بیشتر شود یا زمان استفاده از پردازنده بیش از ۵ ثانیه شود، احتمال دارد در تست‌های بعدی با خطا مواجه شوید. بنابراین:

- از حلقه‌هایی که بدون توقف اجرا می‌شوند پرهیز کنید.
- از توقف‌های طولانی یا ارسال پیام‌های زیاد RPC اجتناب کنید.

قسمت3C: پایداری

اگر سروری که بر پایه Raft ساخته شده، پس از راه اندازی مجدد بتواند ادامه کار را از جایی که متوقف شده از سر بگیرد، باید وضعیت پایدار خود را حفظ کرده باشد. شکل ۲ مقاله نشان می‌دهد که کدام وضعیت‌ها باید پایدار باشند. در پیاده سازی واقعی، وضعیت پایدار Raft باید در زمان تغییر، روی دیسک ذخیره شود و هنگام راه اندازی مجدد از دیسک بازیابی شود. در این تمرین، دیسک واقعی استفاده نمی‌شود، بلکه از شی‌ای به نام Persister برای شبیه سازی دیسک استفاده می‌کنید.

هرکس که تابع Raft.Make() را صدا بزند، باید یک شی Persister را که شامل آخرین وضعیت پایدار Raft است فراهم کند. Raft باید وضعیت خود را از Persister بخواند و در زمان تغییر آن را ذخیره کند. برای این کار باید از توابع ReadRaftState() و Save() استفاده شود.

وظیفه شما در این بخش این است که تابع‌های persist() و readPersist() را در فایل raft_go تکمیل نمایید تا امکان دخیره و بازیابی وضعیت پایدار فراهم باشد. باید وضعیت را به آرایه‌ای از بایت‌ها تبدیل کنید (serialize) و سپس به شی Persister بدهید. برای این کار از انکودر labgob استفاده کنید (مشابه gob در زبان Go). توجه کنید که فیلدهایی با حروف کوچک را نمی‌توان انکود کرد.

در حال حاضر، می‌توانید مقدار nil را به عنوان آرگومان دوم Save() ارسال نمایید. در بخش‌هایی از کد که وضعیت پایدار تغییر می‌کند، تابع persist() را فراخوانی کنید. پس از انجام این مراحل، باید همه تست‌های بخش 3C را با موفقیت بگذرانید.

نکته مهم: احتمالاً باید الگوریتم بازگرداندن nextIndex را به گونه‌ای پیاده سازی کنید که بیش از یک ورودی را عقب بزند. برای این کار به صفحه ۷ و ۸ مقاله Raft مراجعه کنید.

یک پیام رد ممکن است شامل موارد زیر باشد:

```
XTerm:  term in the conflicting entry (if any)
XIndex: index of first entry with that term (if any)
XLen:   log length
```

شکل ۵

سپس منطق رهبر می‌تواند به شکل زیر باشد:

```
Case 1: leader doesn't have XTerm:
  nextIndex = XIndex
Case 2: leader has XTerm:
  nextIndex = (index of leader's last entry for XTerm) + 1
Case 3: follower's log is too short:
  nextIndex = XLen
```

شکل ۶

نکته:

- تست‌های بخش 3C سخت‌تر از 3A و 3B هستند. خطاهایی که در این بخش می‌افتد ممکن است به‌دلیل مشکلاتی در کد بخش‌های قبلی باشد.

نمونه‌ای از اجرای موفق تست‌های 3C:

```
$ go test -run 3C
Test (3C): basic persistence (reliable network)...
  ... Passed --   6.6  3   110    0
Test (3C): more persistence (reliable network)...
  ... Passed --  15.6  5   428    0
Test (3C): partitioned leader and one follower crash, leader restarts (reliable network)...
  ... Passed --   3.1  3    50    0
Test (3C): Figure 8 (reliable network)...
  ... Passed --  33.7  5   654    0
Test (3C): unreliable agreement (unreliable network)...
  ... Passed --   2.1  5  1076    0
Test (3C): Figure 8 (unreliable) (unreliable network)...
  ... Passed --  31.9  5  4400    0
Test (3C): churn (reliable network)...
  ... Passed --  16.8  5  4896    0
Test (3C): unreliable churn (unreliable network)...
  ... Passed --  16.1  5  7204    0
PASS
ok      6.5840/raft1    126.054s
$
```

شکل ۷

پیشنهاد: تست‌ها را چندین بار اجرا کنید تا مطمئن شوید که اجرای شما پایدار است و در همه دفعات نتیجه PASS چاپ می‌شود.

مثال:

```
$ for i in {0..10}; do go test; done
```

شکل ۸

قسمت3D: فشرده سازی لاگ

در حال حاضر، وقتی یک سرور ریبوت می‌شود، تمام لاگ Raft را مجدد اجرا می‌کند تا وضعیت را بازیابی کند. اما برای سیستم‌هایی که مدت طولانی کار می‌کنند، این روش به‌صرفه نیست. در این بخش، باید پیاده سازی خود را طوری تغییر دهید که از وضعیت لحظه‌ای سیستم (snapshot) استفاده کند و بخش قدیمی لاگ را حذف کند. این کار باعث کاهش داده‌های ذخیره شده و افزایش سرعت راه اندازی می‌شود.

اما اگر یک پیرو (follower) بسیار عقب‌تر از رهبر باشد و لاگ‌های مورد نیاز حذف شده باشند، رهبر باید یک snapshot کامل به همراه لاگ فعلی ارسال کند.

برای این کار باید تابع زیر را پیاده سازی کنید که توسط سرویس فراخوانی می‌شود:

```
Snapshot(index int, snapshot []byte)
```

توضیحات:

- در آزمایش 3D، برنامه تست به‌صورت دوره‌ای Snapshot() را صدا می‌زند.
- پارامتر index نشان دهنده آخرین ورودی لاگی است که در snapshot آمده. باید لاگ‌هایی که قبل از این اندیس هستند را حذف کنید.
- باید InstallSnapshot را نیز پیاده سازی کنید تا رهبر بتواند به یک پیرو عقب‌مانده، وضعیت جدید را با snapshot ارسال کند.

وقتی سروری InstallSnapshot دریافت کرد، باید snapshot را در قالب یک ApplyMsg از طریق applyCh به سرویس ارسال کند.

اگر سروری کرش کند، باید وضعیت را از داده‌های پایدار (persisted) و snapshot بازیابی کند. برای این کار، هنگام ذخیره وضعیت با persister.Save()، snapshot را نیز به عنوان آرگومان دوم ارسال کنید. اگر snapshot وجود ندارد، مقدار nil ارسال کنید.

نکات مهم:

- کد خود را طوری تغییر دهید که فقط بخش از لاگ را ذخیره کند. مثلاً لاگ را از اندیس X به بعد نگه دارید و بقیه را با Snapshot(index) حذف کنید.
- رایج‌ترین دلیل شکست تست اول 3D این است که پیروها برای به‌روز شدن بیش از حد زمان می‌گیرند.
- اگر رهبر لاگ کافی برای پیرو نداشته باشد، باید پیام InstallSnapshot ارسال کند.
- کل snapshot را در یک پیام InstallSnapshot بفرستید (از مکانیزم offset استفاده نکنید).
- لاگ‌های قدیمی را طوری حذف کنید که اشاره‌گرها (pointers) از بین بروند و garbage collector بتواند حافظه را آزاد کند.
- زمان مناسب برای اجرای همه تست‌های بخش سوم (3A+3B+3C+3D) بدون race- حدود ۶ دقیقه و با race- حدود ۱۰ دقیقه است.

نمونه اجرای موفق تست‌های3D:

```
$ go test -run 3D
Test (3D): snapshots basic (reliable network)...
  ... Passed --   3.3  3   522    0
Test (3D): install snapshots (disconnect) (reliable network)...
  ... Passed --  48.4  3  2710    0
Test (3D): install snapshots (disconnect) (unreliable network)...
  ... Passed --  56.1  3  3025    0
Test (3D): install snapshots (crash) (reliable network)...
  ... Passed --  33.3  3  1559    0
Test (3D): install snapshots (crash) (unreliable network)...
  ... Passed --  38.1  3  1723    0
Test (3D): crash and restart all servers (unreliable network)...
  ... Passed --  11.2  3   296    0
Test (3D): snapshot initialization after crash (unreliable network)...
  ... Passed --   4.3  3    84    0
PASS
ok      6.5840/raft1    195.006s
```

شکل ۹

نکات لازم جهت تحویل:

- مهلت ارسال در سربرگ تمرین همچنین در ایلرن درج شده است.
- قالب تمرینات به صورت LaTeX تنها در Template تمرینات مورد پذیرش است. (قالب Template در سامانه ایلرن در دسترس است.)
- فایل تمرین ارسالی باید شامل فایل های مورد نیاز به جهت اجرای فایل LaTeX به همراه PDF و همچنین کدهای‌پیاده سازی شده باشد. نام این فایل را به صورت زیر انتخاب کنید:

PR3_Student#_Name

- ارسال با تاخیر پروژه، تنها طبق قوانین درس امکان پذیر است.
- در صورت وجود هرگونه سوال یا ابهام می توانید با خانم الهام رحیمی و آقای هادی قوامی نژاد از طریق ایمیل های e.rahimii@ut.ac.ir و hadighavaminejad@ut.ac.ir در ارتباط باشید.