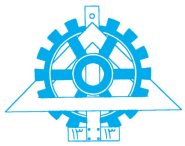
مبانی رایانش توزیع‌شده
Spring 2025



پروژه دوم

۱۴۰۴/۰۳/۰۹

ساخت Key/Value Server

مقدمه

در این پروژه شما قرار است قدم‌به‌قدم یک KV server ساده اما قدرتمند برای یک single machine بسازید. فرض کنید در یک لابراتوار شلوغ دانشگاه، بین قفسه‌های سرور و کابل‌های شبکه، شما و هم‌کلاسی‌هایتان مشغول طراحی سیستمی هستید که حتی وقتی ارتباط شبکه ناپایدار بوده و قطع و وصل می‌شود، هیچ داده‌ای را دو بار ننویسد و همه عملیات‌ها دقیقاً به ترتیب درخواستی کاربران اجرا شوند.

اولین چیزی که با آن سر و کار داریم، «کلاینت» است؛ یک برنامه کوچک که می‌خواهد با ارسال درخواست‌های Put و Get، کلیدها و مقادیر را ذخیره و بازیابی کند. اما بین کلاینت و سرور شبکه‌ای دارای خطاهای احتمالی مانند latency، packet loss و قطع و وصل‌های ناگهانی وجود دارد. شما در نقش «مهندس Reliability» باید تضمین کنید که هر درخواست Put حداکثر یک‌بار روی سرور اجرا شود. یعنی حتی اگر کلاینت بعد از timeout دوباره همان درخواست را ارسال کند، نباید داده را دو بار بنویسید یا وضعیت ناهماهنگی به‌وجود بیاید.

برای رسیدن به این هدف، داستان از این قرار است:

۱. هر زمان کلاینت می‌خواهد یک مقدار را با متد Put ذخیره کند، یک ID تولید می‌کند. سرور هر بار که یک Put می‌گیرد، ابتدا بررسی می‌کند آیا این ID را قبلاً پردازش کرده یا نه. اگر نه، مقدار جدید را در حافظه (یا دیتابیس ساده داخلی) ذخیره می‌کند و نتیجه موفق را برمی‌گرداند. اگر قبلاً همین ID را ذخیره کرده باشد، مقدار قدیم آن را با مقدار جدید دریافتی به‌روزرسانی می‌کند. این مکانیزم باعث می‌شود در شرایط network failures و retryهای متعدد، هیچ Put ای دو بار اثر نگذارد.

۲. *Linearizable* یعنی اگر دو کاربر همزمان روی یک key کار می‌کنند، نتیجه دقیقاً مثل این است که یکی‌شان زودتر کل عملیاتش را تمام کرده و بعد نفر دوم شروع به کار کند. برای این کار باید:

(آ) روی داده‌ها یک قفل درونی (mutex) یا ساختار lock-free ساده استفاده کنید تا هر دفعه فقط یک thread یا goroutine بتواند به‌روزرسانی‌ها را انجام دهد.

(ب) ترتیب درخواست‌ها را بر اساس زمان رسیدن به سرور نهایی کنید به کمک timestamp یا ترتیب صف.

(ج) مطمئن شوید که بعد از هر Put، نتیجه‌ی موفق تایید و سپس به کلاینت اطلاع‌رسانی می‌شود؛ اگر سرور وسط کار crash کند، بعد از ری‌استارت بتواند از روی log یا snapshot درستی وضعیت را بازسازی کند. (این قسمت برای پروژه های بعدی است.)

۳. وقتی سرور KV شما آماده باشد، یک قدم جلو می‌رویم و یک lock توزیع‌شده روی یک ماشین پیاده می‌کنیم. این طراحی به شما اجازه می‌دهد چندین کلاینت فرضی روی یک ماشین رقابت کنند تا lock را بگیرند و linearizability تضمین می‌کند که هیچ دو کلاینت همزمان قفل را نداشته باشند.

۴. هدف این پروژه تنها نوشتن کد نیست؛ باید مطمئن شویم سیستم در شرایط واقعی هم پایدار است.

این پروژه فرصتی است تا مفاهیم پایه‌ای distributed systems مثل *at-most-once semantics*، *linearizability* و *lock-service* را از نزدیک لمس کنید. در خلال طراحی و پیاده‌سازی، یاد می‌گیرید چطور با مشکلات دنیای واقعی نظیر network failures و crashهای احتمالی سرور مقابله کنید و سیستم خود را قابل اتکا نگه دارید.

Key/Value Server

هر کلاینت از طریق یک «Clerk» با سرور کلید/مقدار (KV server) تعامل می‌کند و RPC ارسال می‌نماید. کلاینت‌ها می‌توانند دو نوع RPC مختلف به سرور بفرستند:

۱. Put(key, value, version)

۲. Get(key)

سرور یک ساختار نگاشت (map) در حافظه نگه می‌دارد که برای هر key، یک دوتایی مرتب (tuple) (value, version) را ثبت می‌کند. کلیدها و مقدارها از نوع رشته (string) هستند. شماره نسخه (version) نشان‌دهنده تعداد دفعاتی است که آن key نوشته شده است.

۱. Put(key, value, version)

این فراخوانی مقدار key را تنها در صورتی می‌نویسد یا جایگزین می‌کند که شماره version ارسال‌شده با شماره نسخه فعلی آن key در سرور برابر باشد. در این صورت، سرور نیز version مربوط به آن key را یک واحد افزایش می‌دهد.

۱. اگر نسخه‌ها با هم مطابقت نداشته باشند، سرور باید خطای rpc.ErrVersion را برگرداند.

۲. اگر کلاینت بخواهد key جدیدی بسازد، باید به وسیله Put با مقدار $\text{version} = 0$ این کار را انجام دهد؛ در این حالت سرور پس از موفقیت، مقدار version مربوط به آن key را ۱ قرار می‌دهد.

۳. اگر version ارسالی بزرگ‌تر از ۰ باشد اما key قبلاً وجود نداشته باشد، سرور باید خطای rpc.ErrNoKey را برگرداند.

۲. Get(key)

این فراخوانی مقدار value و version مربوط به آن key را بازمی‌گرداند. اگر key وجود نداشته باشد، سرور باید rpc.ErrNoKey را برگرداند.
در نظر گرفتن یک شماره version برای هر key برای پیاده‌سازی قفل (lock) با استفاده از Put و نیز تضمین «حداکثر یک‌بار» بودن (at-most-once semantics) فراخوانی‌های Put در شبکه‌های ناپایدار که امکان ارسال مجدد درخواست‌ها وجود دارد، بسیار مفید است.

پس از اتمام این آزمایش و گذراندن تمام تست‌ها، از دید کلاینت‌هایی که Clerk.Get و Clerk.Put را فراخوانی می‌کنند، یک سرویس کلید/مقدار خطی‌سازی‌پذیر (linearizable) خواهید داشت. به این معنی که:

۱. اگر عملیات کلاینت‌ها متوالی (non-concurrent) باشند، هر فراخوانی Clerk.Get و Clerk.Put تغییرات وضعیت را دقیقاً مطابق با توالی عملیات قبلی خواهد دید.

۲. اگر عملیات همزمان (concurrent) باشند، مقادیر بازگشتی و حالت نهایی آن‌چنان خواهد بود که گویی عملیات، یکی‌یکی و در یک ترتیب معین اجرا شده‌اند.

عملیاتی را همزمان می‌نامیم اگر بازه زمانی اجرای آن‌ها با هم هم‌پوشانی داشته باشد. برای مثال، اگر کلاینت X صدا بزند Clerk.Put()، سپس کلاینت Y همزمان Clerk.Put() را صدا بزند و بعد فراخوانی X تمام شود، این دو فراخوانی همزمان‌اند. هر عملیات موظف است تأثیر تمام عملیات‌های خاتمه‌یافته قبل از شروع خود را مشاهده کند.
خطی‌سازی‌پذیری (Linearizability) برای توسعه برنامه‌ها بسیار راحت است، چرا که رفتاری مشابه یک سرور تک‌نخی (single thread) که درخواست‌ها را یکی‌یکی پردازش می‌کند ارائه می‌دهد. به عنوان نمونه، اگر یک کلاینت پاسخی موفق از سرور بابت یک عملیات به‌روزرسانی دریافت کند، خوانش‌های بعدی از سوی سایر کلاینت‌ها حتماً تغییر یادشده را خواهند دید. فراهم‌کردن خطی‌سازی‌پذیری برای یک سرور نسبتاً ساده است.

دید کلی نسبت به پروژه

برای شروع کار پروژه، یک اسکلت از کد و مجموعه‌ای از تست‌های آماده در پوشه‌ی src/kvsrv1 در اختیار شماست. این پوشه ساختار پایه را طوری فراهم کرده که بدون نیاز به ایجاد فایل‌ها یا پوشه‌های جدید، بتوانید به سرعت روی منطق اصلی سیستم ذخیره‌سازی کلید/مقدار (Key/Value) متمرکز شوید. هدف این است که شما دو فایل مهم client.go و server.go را بر اساس نیازمندی‌ها تغییر دهید و طوری طراحی کنید که رفتار آن‌ها با مجموعه تست‌ها کاملاً منطبق شود.

در فایل client.go، نوع داده‌ای به نام Clerk تعریف شده است که مسئول مدیریت کامل تعامل با سرور از طریق RPC است. شما باید دو متد اصلی آن یعنی Put(key, value, version) و Get(key) را تکمیل کنید. جزئیاتی که باید در نظر بگیرید شامل این موارد است:

۱. نگهداری و ارسال مجدد (retry) درخواست‌ها در صورت بروز خطاهای موقتی شبکه

۲. ثبت و نگهداری version برای هر عملیات Put و مقایسه‌ی آن با نسخه‌ی فعلی سمت سرور

۳. کنترل همزمانی (concurrency) زمانی که چند Goroutine به‌طور هم‌زمان از یک Clerk استفاده می‌کنند

۴. پیاده‌سازی backoff مناسب برای جلوگیری از ارسال بی‌وقفه درخواست‌ها در زمان قطع ارتباط

تمام این موارد باید به گونه‌ای نوشته شوند که وقتی کلاینت شما ناخواسته خطایی مانند rpc.ErrVersion یا خطاهای مربوط به اتصال می‌گیرد، با رویکردی مشخص (مثلاً سه تلاش متوالی با فاصله‌ی افزایشی) دوباره تلاش کند و در نهایت یا داده را بنویسد یا خطای نهایی را به فراخواننده بازگرداند.

در فایل server.go، کد سمت سرور قرار دارد که از روی یک ساختار ساده‌ی درختی در حافظه برای ذخیره‌سازی زوج‌های (value, version) استفاده می‌کند. نکته‌ی کلیدی این است که هر بار که درخواست Put دریافت می‌شود، نسخه‌ی ارسالی باید با نسخه‌ی فعلی مطابقت داشته باشد؛ در غیر این صورت سرور باید rpc.ErrVersion برگرداند. اگر version بزرگ‌تر از صفر باشد ولی آن کلید در سرور وجود نداشته باشد، باید rpc.ErrNoKey ارسال شود. علاوه بر این، برای کار در محیط چند-نخی(Multi thread) ضروری است از مکانیزم قفل‌گذاری مثلاً sync.Mutex به‌درستی استفاده کنید تا به‌روزرسانی‌های موازی روی یک key باعث اختلال در داده‌ها نشود. توجه کنید که پس از موفقیت‌آمیز بودن Put، نسخه‌ی کلید باید یک واحد افزایش یابد تا در دفعات بعدی قابل شناسایی باشد.

بخش kvsrv1/rpc/rpc.go شامل تعاریف انواع داده‌ها و ارورهای استاندارد برای ارتباط RPC است. ساختارهایی مانند:

۱. PutArgs{Key string; Value string; Version int}

۲. PutReply{Err error}

۳. GetArgs{Key string}

۴. GetReply{Value string; Version int; Err error}

و ثابت‌هایی مانند ErrVersion و ErrNoKey در این فایل تعریف شده‌اند. بد نیست یک بار این فایل را با دقت بخوانید تا از نام دقیق فیلدها و امضا (signature) متدها آگاهی پیدا کنید، چون هرگونه اختلاف در نام یا نوع داده ممکن است باعث شکست تست‌ها شود. هر چند لزوما نیاز به تغییر این فایل نخواهید داشت.

قبل از هر چیز، فایل‌های پروژه که پیوست شده اند را دانلود و ذخیره کنید. در قدم‌های مختلف شبکه تمامی تست‌ها را اجرا کرده و از Pass شدن آن‌ها اطمینان حاصل نمایید.

در ادامه، می‌توانید لاگ (log)های مفصل در داخل سرور و کلاینت قرار دهید تا در زمان خطا یا زمان‌بندی عملیات، اطلاعات کافی برای عیب‌یابی در اختیار داشته باشید. برای مثال، قبل و بعد از هر فراخوانی RPC یک پیام در لاگ بنویسید که شامل key، version و نتیجه‌ی عملیات باشد. همچنین پیشنهاد می‌شود پس از هر تغییر کوچک، دوباره تست‌ها را اجرا کنید تا از تثبیت رفتار صحیح مطمئن شوید و در صورت بروز مشکل سریعاً به نقطه‌ی تغییر اخیر بازگردید.(اختیاری)

با این توضیحات، حالا می‌توانید فایل‌های client.go ، server.go و lock.go را قدم‌به‌قدم توسعه دهید، منطق نسخه‌گذاری و کنترل همزمانی را پیاده کنید، و در نهایت سیستمی مقاوم بسازید که با سناریوهای پیچیده‌ی شبکه و دسترسی‌های هم‌زمان کاملاً سازگار باشد.

قدم اول (پیاده‌سازی سرور key/value در شبکه قابل اتکا)

در این پروژه شما قرار است یک سرویس ساده کلید/مقدار (Key/Value Server) را پیاده‌سازی کنید که در شرایطی کار کند که هیچ پیامی در طول انتقال بین کلاینت و سرور از دست نمی‌رود (شبکه‌ی مطمئن).

اولین گام شما این است که در فایل client.go، متدهای Put و Get از ساختار Clerk را طوری تغییر دهید که بتوانند با سرور از طریق فراخوانی‌های RPC ارتباط برقرار کنند. به عبارت دیگر، هر بار که کلاینت بخواهد یک مقدار را بنویسد یا بخواند، باید یک درخواست RPC مناسب بسازد و آن را ارسال کند. در ادامه، در فایل server.go باید دو هندلر (Handler) مجزا برای RPC با نام‌های Put و Get بنویسید تا سرور بتواند به درخواست‌های ارسال‌شده توسط کلاینت پاسخ دهد و عملیات مورد نظر را انجام بدهد.

نکته‌ی کلیدی این است که در این مرحله شما نیازی به رسیدگی به هیچ‌گونه قطعی یا از دست رفتن پیام ندارید؛ فرض بر این است که همه‌ی پیام‌های ارسالی از سوی کلاینت حتماً و بدون هیچ خطایی به سرور می‌رسند و سرور نیز

بدون هیچ مشکلی پاسخ می‌دهد. پس نیازی به مکانیسم‌های پیچیده‌ی retry یا backoff نیست و فقط کافی است کد ارسال و دریافت RPC را پیاده‌سازی کنید.

پس از نوشتن این کد، وارد پوشه‌ی پروژه شوید و دستور زیر را اجرا کنید:

```
$ cd src/kvsrv1
$ go test -v -run Reliable
```

این دستور فقط تست‌های مربوط به حالت «شبکه‌ی مطمئن» را اجرا می‌کند. خروجی‌ای شبیه به مثال زیر خواهید دید:

```
=== RUN   TestReliablePut
One client and reliable Put (reliable network)...
  ... Passed --   0.0  1     5    0
--- PASS: TestReliablePut (0.00s)
=== RUN   TestPutConcurrentReliable
Test: many clients racing to put values to the same key (reliable network)...
info: linearizability check timed out, assuming history is ok
  ... Passed --   3.1  1 90171 90171
--- PASS: TestPutConcurrentReliable (3.07s)
=== RUN   TestMemPutManyClientsReliable
Test: memory use many put clients (reliable network)...
  ... Passed --   9.2  1 100000    0
--- PASS: TestMemPutManyClientsReliable (16.59s)
PASS
ok   6.5840/kvsrv1 19.681s
```

در این خروجی، هر «Passed» چهار عدد را نمایش می‌دهد:

۱. زمان واقعی اجرا (Real time) به ثانیه

۲. عدد ثابت ۱ که نشان‌دهنده‌ی تعداد کلاینت فعال یا مقدار پیش‌فرضی است که در تست استفاده می‌شود

۳. تعداد کل RPCهای ارسال‌شده، شامل همه‌ی فراخوانی‌های RPC که کلاینت‌ها به سرور می‌فرستند

۴. تعداد کل عملیات کلید/مقدار که شامل فراخوانی‌های Clerk.Put و Clerk.Get می‌شود

برای نمونه، در خط مربوط به TestReliablePut می‌بینید که یک کلاینت با ارسال یک RPC به سرور موفق به نوشتن شده و در مجموع ۵ عملیات کلید/مقدار انجام شده است. در تست دوم هم که چند کلاینت به‌طور همزمان روی یک کلید واحد کار می‌کنند، تعداد بسیار زیادی RPC ارسال و تقریباً همان تعداد عملیات کلید/مقدار ثبت می‌شود.

در نهایت، در تست سوم، عملکرد حافظه (Memory) با تعداد زیاد کلاینت‌ها و عملیات PUT سنجیده می‌شود.

به این ترتیب، زمانی این بخش از کار شما به پایان می‌رسد که موفق شوید هر سه تست Reliable را بدون هیچ خطا یا شکست (FAIL) پشت سر بگذارید. در این نقطه، پیاده‌سازی اولیه‌ی شما برای شرایطی که شبکه کاملاً مطمئن است (بدون هیچ پیام گمشده یا خطایی) کامل و صحیح خواهد بود.

نکته: بررسی race-free

برای اطمینان از عدم وجود شرایط رقابت در دسترسی‌های همزمان با کمک دستور زیر می‌توانید از race-free بودن کد خود اطمینان پیدا کنید:

```
$ go test -race
```

قدم دوم (پیاده‌سازی lock برای key/value clerk)

در بسیاری از سیستم‌های توزیع‌شده (distributed systems)، کلاینت‌های مختلف که روی ماشین‌های جداگانه اجرا می‌شوند، برای هماهنگی و همگام‌سازی رفتار خود ناگزیرند از یک سرویس مشترک استفاده کنند. یکی از روش‌های متداول برای ایجاد این هماهنگی، استفاده از یک سرور کلید/مقدار (key/value server) است که امکان خواندن و نوشتن داده‌ها را به‌صورت اتمیک(Atomic) و با قابلیت نسخه‌بندی فراهم می‌کند.
در حقیقت محصولاتی مثل ZooKeeper و Etcd دقیقاً همین کار را می‌کنند: با فراهم‌کردن متدهای شرطیِ نوشتن (conditional put) و خواندن (get)، امکان پیاده‌سازی قفل توزیع‌شده (distributed lock) را برای کلاینت‌ها مهیا می‌سازند؛ قفلی که شبیه قفل‌های درون‌برنامه‌ای Go همان sync.Mutex عمل کرده و تضمین می‌کند در هر لحظه تنها یک مالک (owner) وجود دارد.
در این تمرین، شما باید یک لایه‌ی ساده اما کاربردی از قفل را روی متدهای پایه‌ای Clerk یعنی Clerk.Put و Clerk.Get بسازید. این قفل که در فایلی با نام lock.go در پوشه lock قرار دارد، دارای دو متد اصلی میباشد:

۱. **Acquire**

(آ) وقتی کلاینتی قصد دارد وارد بخش بحرانی (critical section) شود، باید ابتدا این متد را فراخوانی کند.

(ب) اگر قفل آزاد باشد، فراخوانی موفق شده و کلاینت «مالک» قفل می‌شود.

(ج) اگر قفل در اختیار کلاینت دیگری باشد، لازم است فراخوانی شما منتظر بماند تا قفل آزاد شود.

۲. **Release**

(آ) وقتی کلاینت owner کارش با بخش بحرانی تمام شد، باید این متد را فراخوانی کند.

(ب) این عمل قفل را برای سایر کلاینت‌ها آزاد می‌کند.

چالش اصلی این است که تضمین کنید در هیچ شرایطی بیشتر از یک کلاینت نتواند قفل را بگیرد. اگر دو کلاینت هم‌زمان درخواست Acquire بدهند، تنها یکی باید موفق شود و دیگری باید منتظر بماند تا قفل آزاد شود. در سناریوهای دنیای واقعی ممکن است یک کلاینت در زمان نگهداری قفل کرش کند و قفل هیچگاه آزاد نشود که برای حل این مشکل در دنیای واقعی نیاز به مکانیزم‌های پیچیده‌تر مثل الحاق «اجاره» یا Lease به قفل و منقضی‌شدن خودکار آن داریم. در این پروژه فرض می‌کنیم که کلاینت‌ها هیچگاه کرش نمی‌کنند و بنابراین شما می‌توانید این چالش را نادیده بگیرید.

گام‌های پیشنهادی برای پیاده‌سازی

۱. ایجاد شناسه‌ی یکتا برای هر کلاینت

(آ) در ابتدای ساخت قفل درون MakeLock یک مقدار تصادفی ۸ کاراکتری به عنوان شناسه تولید کنید. برای این منظور می‌توانید از تابع زیر استفاده کنید.

```
kvtest.RandValue(8)
```

(ب) این شناسه به شما کمک می‌کند زمانی که چند کلاینت روی یک کلید مشترک قفل می‌زنند، تشخیص دهید کدام‌یک درخواستِ آزادسازی (Release) را فرستاده است.

۲. متد Acquire

ایده‌ی اصلی:

(آ) حلقه بزنید تا زمانی که موفق شوید وضعیت قفل را بنویسید.

(ب) اگر موفق شدید، حلقه را بشکنید و owner قفل شوید.

(ج) اگر ناموفق بودید، کمی صبر کنید (مثلاً با time.Sleep) و دوباره امتحان کنید.

۳. متد Release

وقتی نوبت آزادسازی قفل می‌رسد:

(آ) دوباره نسخه‌ی فعلی کلید را با Get بخوانید،

(ب) اگر مقدار خوانده‌شده (value) با lk.id شما برابر بود، یعنی شما مالک قفل هستید.

(ج) در این حالت یک Put شرطی دیگر بفرستید تا نسخه را افزایش داده و مقدار را به یک رشته‌ی خالی (یا هر مقدار پیش‌فرض) برگردانید.

(د) اگر مقدارِ فعلی با شناسه‌ی شما مطابقت نداشت، یعنی قفل در اختیار شخص دیگری است و لازم نیست چیزی انجام دهید یا می‌توانید ارور بدهید.

۴. تست

با اجرای دستور زیر، تست Reliable را انجام دهید:

```
$ cd lock
$ go test -v -run Reliable
```

در صورت شکست تست ManyClientsReliable، احتمالاً حلقه‌ی Acquire یا چک کردن نسخه در Release به درستی پیاده نشده است.

```
=== RUN   TestOneClientReliable
Test: 1 lock clients (reliable network)...
  ... Passed --   2.0  1   974    0
--- PASS: TestOneClientReliable (2.01s)
=== RUN   TestManyClientsReliable
Test: 10 lock clients (reliable network)...
  ... Passed --   2.1  1 83194    0
--- PASS: TestManyClientsReliable (2.11s)
PASS
ok   6.5840/kvsrv1/lock 4.120s
```

توصیه‌هایی برای موفقیت

۱. **حلقه با backoff**: در حلقه‌ی Acquire هر بار که در به دست آوردن قفل ناموفق بودید، به جای تلاش مکرر و بی‌وقفه، چند میلی‌ثانیه صبر کنید.

۲. **مقایسه‌ی مقدار**: در Release قبل از هر عملی مطمئن شوید که Get برگردانده lk.id خودتان باشد تا از آزادسازیِ قفل توسط کلاینت‌های دیگر جلوگیری شود.

قدم سوم (سرور key/value با قابلیت مقابله با حذف یا از دست رفتن پیام‌ها)

در این تمرین، چالش اصلی این است که شبکه ارتباطی بین کلاینت و سرور می‌تواند تحت تأثیر ناپایداری‌های مختلفی قرار بگیرد: پیام‌های RPC ممکن است به ترتیب اشتباه برسند، در مسیر دچار تأخیر شوند یا اصلاً هرگز به مقصد نرسند و حذف شوند. هدف شما در سمت کلاینت، یعنی در پیاده‌سازی ساختار «Clerk» در فایل kvsrv1/client.go است که بتواند در مقابل همه این خطاهای شبکه مقاومت کند و در نهایت نتیجهٔ قابل اطمینانی به لایه بالاتر (برنامه) برگرداند.

مرور چند سناریوی ممکن:

۱. **از دست رفتن پیام درخواست (Request)**: اگر پیغام Put یا Get که از کلاینت ارسال می‌شود، در میانه راه گم شود، کافی است کلاینت همان درخواست را دوباره ارسال کند. سرور، هنگام دریافت مجدد همان درخواست، آن را اجرا می‌کند (در مورد Get که تغییری ایجاد نمی‌کند مشکلی پیش نمی‌آید، و در مورد Put هم اجرای مشروط بر version، از دو بار نوشتنِ یکسان جلوگیری می‌کند).

۲. **از دست رفتن پیام پاسخ (Reply)**: اگر سرور درخواست را اجرا کرده اما پاسخ آن (شامل مقدار موفق یا خطا) در برگشت گم شود، کلاینت پس از یک timeout مشخص (مثلاً ۱۰۰ میلی‌ثانیه) پیغام دریافت‌نشده را تشخیص می‌دهد و درخواست را مجدداً ارسال می‌کند. این همان مکانیزم retry است که باید پیاده‌سازی کنید.

در این حالت، یک پیچیدگی وجود دارد:
فرض کنید کلاینت، یک درخواست Put(key, value, version) را ارسال می‌کند و سرور برای اولین اجرا آن را می‌پذیرد و version را ++ می‌کند ولی پاسخ موفق در میانه راه گم می‌شود. کلاینت هنوز پاسخی دریافت نکرده و مجدداً همان RPC را ارسال می‌کند. این بار سرور متوجه می‌شود که شماره نسخه ارسالی با نسخه فعلی سرور مطابقت ندارد و به‌جای نوشتن مجدد، rpc.ErrVersion را برمی‌گرداند. حالت دیگری وجود دارد که هم درخواست اول و هم درخواست دوم با خطا مواجه شده باشند که نتیجه خطای درخواست اول به دست کلاینت نرسیده باشد.
حال از نگاه کلاینت این سؤال پیش می‌آید: آیا درخواست اول اجرا شده یا نه؟ آیا پیغام خطای دریافتی به دلیل تداخل با به‌روزرسانی دیگری بوده یا صرفاً پاسخ اول گم شده؟

قانون کلی برای حل این ابهام:

۱. اگر اولین ارسالِ Put با rpc.ErrVersion پاسخ داده شود، یعنی نسخه ارسالی شما با نسخه فعلی متفاوت بوده و سرور هیچ عملی انجام نداده. در این حالت باید همان rpc.ErrVersion را به برنامه بالادست برگردانید.

۲. اما اگر در تلاش مجدد (retry) پاسخ rpc.ErrVersion دریافت کنید، احتمال دارد نسخه اولیه روی سرور اعمال شده اما صرفا پاسخ اول گم شده باشد یا اینکه هر دو درخواست با خطای rpc.ErrVersion مواجه شده باشند. در این حالت باید مقدار ویژه‌ای به نام rpc.ErrMaybe را برگردانید.

نحوه پیاده‌سازی در client.go

حالا باید فایل kvsrv1/client.go خود را طوری تغییر دهید که در مواجهه با از دست رفتن درخواست‌ها و پاسخ‌های RPC هم بتواند ادامه دهد. مقدار بازگشتی true از فراخوانی ck.clnt.Call() در کلاینت نشان می‌دهد که کلاینت پاسخ RPC را از سرور دریافت کرده است؛ مقدار بازگشتی false یعنی پاسخی دریافت نشده است (به‌طور دقیق‌تر، متد Call() به مدت مشخصی منتظر دریافت پاسخ می‌ماند و اگر در آن بازه زمانی پاسخی نرسد، false برمی‌گرداند). Clerk شما باید تا وقتی پاسخی دریافت نکرده است، همان RPC را دوباره ارسال کند. بحث مربوط به rpc.ErrMaybe را هم مدنظر قرار دهید.

نکات مهم:

۱. بعد از هر بار تلاش برای ارتباط با سرور باید حتماً time.Sleep(100 * time.Millisecond) فراخوانی شود تا با ایجاد وقفه زمانی، از مصرف زیاد CPU و فشار به سرور جلوگیری شود.

۲. برای متد Get نیز مشابه عمل می‌کنید، ولی چون تغییری ایجاد نمی‌کند، تنها تا دریافت پاسخ مناسب retry می‌کنید و نیازی به بازگرداندن ErrMaybe نیست.

۳. در این مرحله نیازی به تغییر هیچ چیزی در سمت سرور (kvsrv1/server.go) ندارید؛ تمام کار سمت کلاینت و درون client.go انجام می‌شود.

۴. تابع clnt.Call به‌صورت blocking منتظر پاسخ می‌ماند و مقدار bool بازگشتی نشان می‌دهد که آیا پاسخی دریافت شده یا نه.

تست نهایی

زمانی که تغییرات لازم را انجام دادید، با اجرای دستور زیر اطمینان حاصل کنید که همه تست‌ها موفق هستند:

```
$ go test -v
```

نمونه خروجی موفق:

```
=== RUN   TestReliablePut
  ... Passed --   0.0  1     5    0
=== RUN   TestPutConcurrentReliable
  ... Passed --   3.1  1 106647 106647
=== RUN   TestMemPutManyClientsReliable
  ... Passed --   8.0  1 100000    0
=== RUN   TestUnreliableNet
  ... Passed --   7.6  1   251  208
PASS
ok   6.5840/kvsrv1 25.319s
```

با رعایت این نکات، ساختار Clerk شما نسبت به از دست رفتن پیام‌های درخواست یا پاسخ مقاوم می‌شود و بدون دستکاری سرور، قابلیت retry و بازگرداندن خطاهای مناسب ErrVersion و ErrMaybe را فراهم می‌کند.

قدم چهارم (پیاده‌سازی lock برای key/value clerk در یک شبکه غیرقابل اتکا)

در این تمرین قرار است پیاده‌سازی قفل توزیع‌شده (distributed lock) را بر اساس همان «Clerk» که برای ارسال و دریافت فراخوانی‌های RPC به سرور key/value استفاده می‌شود، به گونه‌ای گسترش دهیم که حتی در شبکه‌های غیرقابل اتکا (unreliable network) نیز بتواند به درستی کار کند.
نکته کلیدی در این‌جا آن است که در چنین شبکه‌ای ممکن است پیام‌های درخواست (RPC request) و یا پیام‌های پاسخ (RPC reply) به کلی گم شوند، دوباره تکرار شوند (duplicate)، یا تأخیر طولانی داشته باشند؛ بنابراین پیاده‌سازی شما باید خود را برای این نوع رفتارهای غیرقابل‌پیش‌بینی آماده کند و بتواند در عمل نیز از پس آن‌ها بربیاید.

درک سازوکار فعلی Clerk

۱. هر کلاینت برای انجام عملیات قفل دو متد اصلی دارد: Acquire(lockName) که تلاشی برای گرفتن قفل با شناسه مشخص می‌کند و Release(lockName) که قفل را آزاد می‌کند.

۲. درون این متدها، فراخوانی‌های Put و Get به سرور key/value ارسال می‌شوند تا وضعیت قفل (مثلاً نگهداری یک کلید با مقدار هدر یا شناسه کلاینتِ صاحب قفل را ثبت یا بخوانند).

۳. در شرایط شبکه مطمئن (reliable)، با یک یا چند تلاش ساده برای ارسال پیام، کارها به درستی پیش می‌رود؛ اما وقتی امکان گم شدن یا تأخیر پیام‌ها وجود دارد، باید مکانیزم‌هایی اضافه شوند تا بدون ایجاد خطا یا بن‌بست (deadlock) بتوان عملیات را ادامه داد.

مراحل تست

در گام بعدی، باید با دستورات زیر به پوشه lock رفته و تست مورد نظر را اجرا کرده و مطمئن شوید که تست‌های زیر پاس می‌شوند:

```
$ cd lock
$ go test -v
```

خروجی نهایی باید شامل این چهار سناریو باشد:

۱. TestOneClientReliable: تک‌کلاینت در شبکه قابل اتکا

۲. TestManyClientsReliable: چند کلاینت در شبکه قابل‌اتکا

۳. TestOneClientUnreliable: تک‌کلاینت در شبکه غیرقابل‌اتکا

۴. TestManyClientsUnreliable: چند کلاینت در شبکه غیرقابل‌اتکا

نمونه خروجی موفق:

```
=== RUN   TestOneClientReliable
Test: 1 lock clients (reliable network)...
  ... Passed --   2.0  1   968    0
--- PASS: TestOneClientReliable (2.01s)

=== RUN   TestManyClientsReliable
Test: 10 lock clients (reliable network)...
  ... Passed --   2.1  1 10789    0
--- PASS: TestManyClientsReliable (2.12s)

=== RUN   TestOneClientUnreliable
Test: 1 lock clients (unreliable network)...
  ... Passed --   2.3  1    70    0
--- PASS: TestOneClientUnreliable (2.27s)

=== RUN   TestManyClientsUnreliable
Test: 10 lock clients (unreliable network)...
```

```
  ... Passed --   3.6  1   908    0
--- PASS: TestManyClientsUnreliable (3.62s)

PASS
ok   6.5840/kvsrv1/lock 10.033s
```

تحلیل نتایج:

۱. تعداد ارسال‌ها (sends) و دریافت‌ها (receives) نشان می‌دهد که در حالت غیرقابل‌اتکا، برخی پیام‌ها گم شده‌اند اما مکانیزم Retry شما توانسته بدون خطا پروتکل را کامل کند.

۲. مدت زمان اجرا افزایش یافته که امری طبیعی است وقتی تعداد تلاش‌ها بیشتر می‌شود.

با رعایت موارد فوق و آزمودن دقیق سناریوهای مختلف شبکه از تأخیرهای بسیار کوتاه تا گم شدن متناوب پیام‌ها می‌توانید مطمئن شوید که پیاده‌سازی قفل شما در هر شرایطی، چه شبکه مطمئن چه غیرمطمئن، به صورت صحیح و پایدار عمل خواهد کرد.

نکاتی که باید توجه داشته باشید:

الف) مهلت ارسال در سربرگ تمرین همچنین در ایلرن درج شده است.

ب) قالب تمرینات به صورت LaTeX و تنها در Template تمرینات مورد پذیرش است. (Template در ایلرن در دسترس است.)

ج) فایل تمرین ارسالی باید شامل فایل های مورد نیاز به جهت اجرای فایل LaTeX به همراه PDF و همچنین کدهای پیاده سازی شده باشد. نام این فایل را به صورت زیر انتخاب کنید:

PR2_Student#_Name

د) ارسال با تاخیر پروژه، تنها طبق قوانین درس امکان پذیر است.

ه) در صورت وجود هرگونه سوال یا ابهام می توانید با آقای صفری از طریق ایمیل me.safari@ut.ac.ir و آقای قربانی از طریق ایمیل alighorbani1380@ut.ac.ir در ارتباط باشید.