مبانی رایانش توزیع یافته
علی حمزه پور، مینا شیرازی، امیرعلی رحیمی

تمرین
کامپیوتری
شماره ۲

شماره دانشجویی:
۸۱۰۱۰۰۱۲۹
۸۱۰۱۰۰۲۵۰
۸۱۰۱۰۰۱۴۶

# بخش اول

# مقدمه

## ۱ هدف پروژه

در این پروژه قصد داریم یک سیستم ساده ولی قابل اتکا برای ذخیره‌سازی داده به صورت کلید/مقدار (Key/Value Server) طراحی و پیاده‌سازی کنیم که بتواند در شرایط مختلف شبکه‌ای، از جمله قطعی، تأخیر، و تکرار پیام‌ها، عملکرد درستی داشته باشد. هدف این است که با رعایت مفاهیمی مانند linearizability و at-most-once semantics، یک سیستم توزیع‌شده کوچک ولی مقاوم و قابل اعتماد بسازیم که پایه‌ای برای درک بهتر سیستم‌های واقعی مثل ZooKeeper یا Etcd باشد.

## ۲ نقش سرور و کلرک

در این سیستم، «سرور» نقش هسته‌ی اصلی را ایفا می‌کند و وظیفه‌ی نگهداری داده‌ها را با حفظ نسخه‌گذاری (versioning) بر عهده دارد. از طرف دیگر، «کلرک» (Clerk) نماینده‌ی کلاینت است که با استفاده از RPC درخواست‌های Put و Get را به سرور ارسال می‌کند. کلرک باید بتواند با در نظر گرفتن خطاهای موقت شبکه، درخواست‌ها را به‌درستی و بدون ایجاد اختلال یا تکرار، مدیریت کند. ارتباط بین کلرک و سرور اساس پیاده‌سازی تمامی مراحل پروژه است.

## ۳ مراحل پروژه

پروژه در چهار مرحله انجام می‌شود:
- **مرحله اول:** یک سرور و کلرک ساده طراحی می‌کنیم که در شرایط شبکه‌ی پایدار کار می‌کنند.
- **مرحله دوم:** از همین سرور برای پیاده‌سازی یک قفل توزیع‌شده استفاده می‌کنیم تا بتوانیم رقابت میان چند کلاینت را کنترل کنیم.
- **مرحله سوم و چهارم:** همین دو مرحله را برای شرایط شبکه‌ی غیرقابل اتکا توسعه می‌دهیم؛ به این صورت که باید سیستم را در برابر پیام‌های گم‌شده، تکراری یا با تأخیر مقاوم کنیم.

# بخش دوم

# پیاده‌سازی سرور key/value در شبکه قابل اتکا

در قدم اول پروژه، هدف ما پیاده‌سازی یک سرور کلید/مقدار ساده (KVServer) و کلرکی (Clerk) بود که بتوانند از طریق RPC با هم ارتباط برقرار کرده و عملیات‌های پایه‌ای Get و Put را انجام دهند. در این مرحله فرض بر این بود که شبکه کاملاً قابل اتکاست؛ یعنی هیچ پیام گم نمی‌شود یا دوباره ارسال نمی‌گردد. بنابراین نیازی به پیاده‌سازی مکانیزم‌هایی مانند retry یا backoff وجود نداشت و تمرکز اصلی بر روی منطق عملکرد سرور و کلرک بود.

## ۱ ساختار سرور

در سمت سرور، ساختار KVServer شامل یک نگاشت از کلیدها به زوج مقدار/نسخه (ValueVersionPair) و یک قفل برای کنترل هم‌زمانی است:

```
type ValueVersionPair struct {
    Value   string
    Version rpc.Tversion
}
type KVServer struct {
    mu       sync.Mutex
    mappings map[string]ValueVersionPair
}
```

## ۲ متد Get

متد Get بررسی می‌کند که آیا کلید مورد نظر در سرور وجود دارد یا نه. اگر وجود داشته باشد، مقدار و نسخه را برمی‌گرداند، در غیر این صورت خطای ErrNoKey:

```
func (kv *KVServer) Get(args *rpc.GetArgs, reply *rpc.GetReply) {
    kv.mu.Lock()
    defer kv.mu.Unlock()
    if pair, exists := kv.mappings[args.Key]; exists {
        reply.Value = pair.Value
        reply.Version = pair.Version
        reply.Err = rpc.OK
    } else {
        reply.Err = rpc.ErrNoKey
    }
}
```

## ۳ متد Put

در متد Put، اگر کلید وجود داشته باشد و نسخه‌های ارسالی و فعلی با هم مطابقت داشته باشند، مقدار جدید با نسخه جدید ذخیره می‌شود. اگر نسخه‌ها برابر نباشند، ErrVersion بازگردانده می‌شود. اگر کلید وجود نداشته باشد، تنها در صورتی ذخیره انجام می‌شود که نسخه برابر با صفر باشد:

```
func (kv *KVServer) Put(args *rpc.PutArgs, reply *rpc.PutReply) {
    kv.mu.Lock()
    defer kv.mu.Unlock()
    if pair, exists := kv.mappings[args.Key]; exists {
        if pair.Version == args.Version {
            kv.mappings[args.Key] = ValueVersionPair{Value: args.Value,
               Version: args.Version + 1}
            reply.Err = rpc.OK
        } else {
            reply.Err = rpc.ErrVersion
```

```
        }
    } else {
        if args.Version == 0 {
            kv.mappings[args.Key] = ValueVersionPair{Value: args.Value,
                Version: 1}
            reply.Err = rpc.OK
        } else {
            reply.Err = rpc.ErrNoKey
        }
    }
}
```

# ۴ ساختار کلاینت

در سمت کلاینت، ساختار Clerk شامل یک شیء tester.Clnt برای ارتباط RPC و نام سرور است. متد Get یک درخواست ساده برای دریافت مقدار و نسخه کلید ارسال می‌کند:

```
func (ck *Clerk) Get(key string) (string, rpc.Tversion, rpc.Err) {
    args := &rpc.GetArgs{Key: key}
    reply := &rpc.GetReply{}
    ck.clnt.Call(ck.server, "KVServer.Get", args, reply)
    if reply.Err == rpc.OK {
        return reply.Value, reply.Version, rpc.OK
    }
    return "", 0, rpc.ErrNoKey
}
```

متد Put نیز با ارسال کلید، مقدار و نسخه، درخواست به‌روزرسانی را ارسال می‌کند و خطای بازگشتی را مستقیماً به کاربر بازمی‌گرداند:

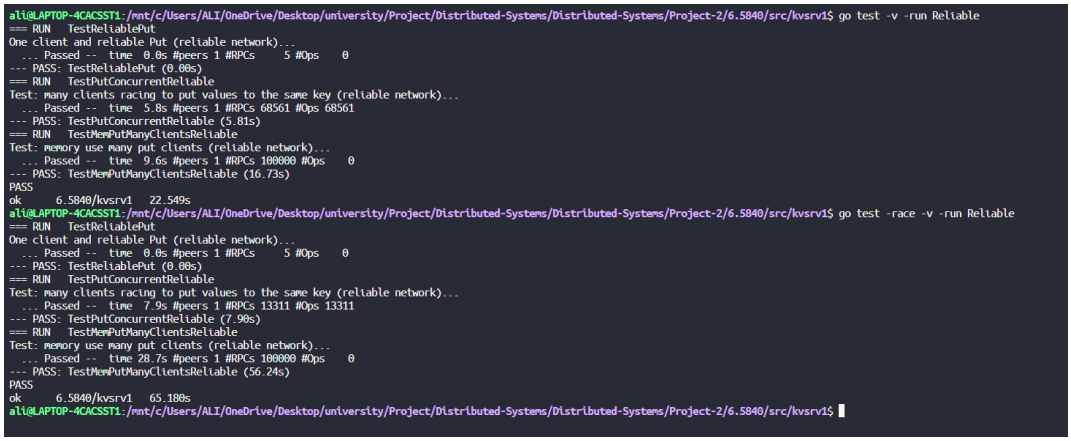
```
func (ck *Clerk) Put(key, value string, version rpc.Tversion) rpc.Err {
    args := &rpc.PutArgs{Key: key, Value: value, Version: version}
    reply := &rpc.PutReply{}
```

```
    ck.clnt.Call(ck.server, "KVServer.Put", args, reply)
    return reply.Err
}
```

در این مرحله، ساختار کلی ارتباط کلاینت و سرور و منطق پایه‌ای ذخیره و بازیابی اطلاعات به‌درستی پیاده‌سازی شده و آماده‌ی گسترش در مراحل بعدی پروژه است.

## ۵ تست قدم اول

در این قسمت تست‌های مخصوص قدم اول را هم در حالت تک کلاینته و هم برای حالت چند کلاینته( با فلگ race-) اجرا کردیم و تست‌ها با موفقیت pass شدند.




شکل ۱: نتیجه اجرای تست‌های قدم اول

# بخش سوم

# پیاده‌سازی lock برای key/value clerk

در این مرحله از پروژه، هدف اصلی ما پیاده‌سازی یک قفل توزیع‌شده (Distributed Lock) بود، به‌گونه‌ای که چند کلاینت بتوانند به‌صورت ایمن برای دسترسی به یک بخش بحرانی رقابت کنند، ولی فقط یکی از آن‌ها در هر زمان مجاز به ورود باشد. نکته کلیدی اینجاست که ما از همان سرور کلید/مقدار (Key/Value Server) مرحله اول استفاده می‌کنیم، و بدون هیچ سازوکار اضافی، تنها با استفاده از متدهای Put و Get این قفل را پیاده‌سازی می‌کنیم.

# ۱ ساختار Lock

ساختار اصلی برای مدیریت قفل، Lock نام دارد:

```
type Lock struct {
    ck   kvtest.IKVClerk
    name string
    id   string
}
```

- ck شیئی از نوع IKVClerk است که اجازه فراخوانی توابع Put و Get روی سرور KV را می‌دهد.
- name نام قفلی است که قرار است در سرور ذخیره شود. هر قفل در واقع یک کلید در سرور است.
- id یک رشته‌ی تصادفی است که هنگام ساخت قفل تولید می‌شود و شناسه مالک قفل در آن لحظه است. این شناسه در MakeLock با استفاده از kvtest.RandValue(8) تولید می‌شود و تضمین می‌کند که هر کلاینت ID منحصربه‌فردی داشته باشد.

# ۲ تابع Acquire: گرفتن قفل

در این تابع، کلاینت تلاش می‌کند قفل را به‌دست آورد. این کار به‌صورت حلقه‌ای انجام می‌شود تا زمانی که موفق شود:

```
func (lk *Lock) Acquire() {
    for {
        val, ver, err := lk.ck.Get(lk.name)
```

در اینجا ابتدا مقدار قفل را از سرور می‌خوانیم. حال چند حالت مختلف ممکن است پیش بیاید:

- **قفل هنوز روی سرور تعریف نشده است (ErrNoKey)**

این به معنی آن است که هیچ کلاینتی تا به حال تلاش نکرده این قفل را بگیرد. در این حالت، کلاینت سعی می‌کند مقدار id خود را با نسخه ۰ روی سرور Put کند:

```
    if err == rpc.ErrNoKey {
        if lk.ck.Put(lk.name, lk.id, 0) == rpc.OK {
            return
        }
```

```
    }
```

**- قفل وجود دارد و مقدار آن خالی است (val == "")**

یعنی قفل آزاد شده است ولی کلید در سرور وجود دارد. در این صورت نیز کلاینت با یک Put مشروط (یعنی با تطابق نسخه) سعی می‌کند مقدار را به id خود تغییر دهد:

```
    else if err == rpc.OK && val == "" {
        if lk.ck.Put(lk.name, lk.id, ver) == rpc.OK {
            return
        }
    }
```

**- در سایر موارد**، یعنی قفل در دست کلاینت دیگری است یا عملیات موفق نیست، کلاینت کمی صبر می‌کند و دوباره تلاش می‌کند:

```
    time.Sleep(50 * time.Millisecond)
```

این حلقه تا زمانی ادامه پیدا می‌کند که یکی از دو حالت بالا منجر به موفقیت در Put شود.

# ۳ تابع Release: آزاد کردن قفل

هنگامی که کلاینت کارش با بخش بحرانی تمام شد، باید قفل را آزاد کند تا دیگران بتوانند آن را بگیرند:

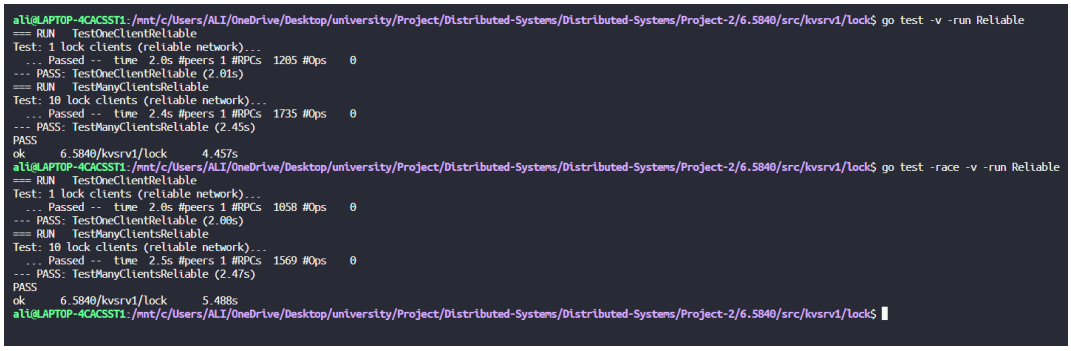
```
func (lk *Lock) Release() {
    val, ver, err := lk.ck.Get(lk.name)
```

ابتدا وضعیت قفل را می‌خوانیم. اگر خواندن موفق نبود (err != rpc.OK) قفل آزاد نمی‌شود. اگر موفق بود، بررسی می‌کنیم آیا کلاینت فعلی مالک قفل است یا نه:

```
if val == lk.id {
    lk.ck.Put(lk.name, "", ver)
}
```

اگر مقدار خوانده‌شده برابر با id ما بود، یعنی ما صاحب قفل هستیم و باید آن را با یک Put خالی (رشته خالی) آزاد کنیم. به این ترتیب، کلاینت‌های دیگر می‌توانند در Acquire خود مقدار خالی را تشخیص دهند و برای گرفتن قفل تلاش کنند.

## ۴ تست قدم دوم




شکل ۲: نتیجه اجرای تست‌های قدم دوم

# بخش چهارم

# سرور key/value با قابلیت مقابله با حذف یا از دست رفتن پیام‌ها

در قدم سوم پروژه، هدف ما مقاوم‌سازی کلاینت (Clerk) در برابر مشکلات رایج شبکه‌های غیرقابل اتکا بود. برخلاف مراحل قبل که فرض می‌شد تمام پیام‌های RPC بدون خطا و به موقع به مقصد می‌رسند، در این مرحله باید در نظر بگیریم که ممکن است پیام‌ها در مسیر گم شوند، تکرار شوند یا با تأخیر زیاد به مقصد برسند. بنابراین، باید منطق کلرک به‌گونه‌ای تغییر کند که این خطاها را مدیریت کرده و تا جای ممکن رفتار سیستم را صحیح نگه دارد.

## ۱ تغییرات در متد Get

مهم‌ترین تغییر اعمال‌شده در متد Get است. پیش از این، اگر سرور پاسخ نمی‌داد، کلرک درخواست را متوقف می‌کرد. حالا در صورت عدم دریافت پاسخ (یعنی gotResponse == false)، کلرک با استفاده از یک حلقه بی‌نهایت و تأخیر بین تلاش‌ها (WaitForRetransmit) درخواست را تا زمانی که سرور پاسخ دهد تکرار می‌کند:

```
for {
    gotResponse := ck.clnt.Call(ck.server, "KVServer.Get", args, reply)
    if gotResponse {
```

```
        break
    }
    WaitForRetransmit()
}
```

## ۲ تغییرات در متد Put

در مورد Put، شرایط پیچیده‌تری وجود دارد. اگر نخستین تلاش برای ارسال RPC موفق باشد، پاسخ را مستقیم بازمی‌گردانیم. اما اگر این ارسال موفق نباشد، فرض می‌کنیم که ممکن است درخواست در سرور اجرا شده ولی پاسخ گم شده باشد. بنابراین، با استفاده از تابع retransmit درخواست را تکرار می‌کنیم:

```
if !gotResponse {
    ck.retransmit(args, reply)
    if reply.Err == rpc.ErrVersion {
        return rpc.ErrMaybe
    }
}
```

نکته مهم اینجاست که اگر در تلاش‌های مجدد، خطای ErrVersion دریافت کنیم، به جای بازگرداندن همان خطا، ErrMaybe برمی‌گردانیم. این یعنی: "ممکن است این عملیات قبلاً در سرور انجام شده باشد، ولی ما مطمئن نیستیم." این تصمیم از منطق at-most-once و رعایت safety در اجرای Put ناشی می‌شود.

در نهایت، این تغییرات باعث شدند که کلرک بتواند با تکرار هوشمندانه‌ی درخواست‌ها، تا حد زیادی خطاهای شبکه را پوشش دهد و در عین حال از اعمال مجدد ناخواسته‌ی عملیات جلوگیری کند. به این ترتیب، یک لایه‌ی مقاوم در برابر خطا روی سیستم اولیه ساخته‌ایم که پایه‌ای برای مراحل پیشرفته‌تر پروژه است.

## ۳ تست قدم سوم

```
ali@LAPTOP-4CACSST1:/mnt/c/Users/ALI/OneDrive/Desktop/university/Project/Distributed-Systems/Distributed-Systems/Project-2/6.5840/src/kvsrv1$ go test -v
=== RUN   TestReliablePut
One client and reliable Put (reliable network)...
  ... Passed --  time  0.0s #peers 1 #RPCs     5 #Ops    0
--- PASS: TestReliablePut (0.00s)
=== RUN   TestPutConcurrentReliable
Test: many clients racing to put values to the same key (reliable network)...
  ... Passed --  time  3.9s #peers 1 #RPCs 68371 #Ops 68371
--- PASS: TestPutConcurrentReliable (3.87s)
=== RUN   TestMemPutManyClientsReliable
Test: memory use many put clients (reliable network)...
  ... Passed --  time  9.0s #peers 1 #RPCs 100000 #Ops    0
--- PASS: TestMemPutManyClientsReliable (15.73s)
=== RUN   TestUnreliableNet
One client (unreliable network)...
  ... Passed --  time  8.8s #peers 1 #RPCs   266 #Ops  212
--- PASS: TestUnreliableNet (8.81s)
PASS
ok      6.5840/kvsrv1   28.428s
ali@LAPTOP-4CACSST1:/mnt/c/Users/ALI/OneDrive/Desktop/university/Project/Distributed-Systems/Distributed-Systems/Project-2/6.5840/src/kvsrv1$ go test -race -v
=== RUN   TestReliablePut
One client and reliable Put (reliable network)...
  ... Passed --  time  0.0s #peers 1 #RPCs     5 #Ops    0
--- PASS: TestReliablePut (0.00s)
=== RUN   TestPutConcurrentReliable
Test: many clients racing to put values to the same key (reliable network)...
  ... Passed --  time  6.9s #peers 1 #RPCs 14037 #Ops 14037
--- PASS: TestPutConcurrentReliable (6.92s)
=== RUN   TestMemPutManyClientsReliable
Test: memory use many put clients (reliable network)...
  ... Passed --  time 26.9s #peers 1 #RPCs 100000 #Ops    0
--- PASS: TestMemPutManyClientsReliable (52.82s)
=== RUN   TestUnreliableNet
One client (unreliable network)...
  ... Passed --  time  7.7s #peers 1 #RPCs   251 #Ops  212
--- PASS: TestUnreliableNet (7.75s)
PASS
ok      6.5840/kvsrv1   68.524s
ali@LAPTOP-4CACSST1:/mnt/c/Users/ALI/OneDrive/Desktop/university/Project/Distributed-Systems/Distributed-Systems/Project-2/6.5840/src/kvsrv1$
```

شکل ۳: نتیجه اجرای تست‌های قدم سوم

# بخش پنجم

# پیاده‌سازی lock برای key/value clerk در یک شبکه غیر قابل اتکا

در این مرحله از پروژه، هدف ما مقاوم‌سازی قفل توزیع‌شده‌ای بود که در مرحله‌ی قبل طراحی کرده بودیم، به‌گونه‌ای که در شرایط شبکه‌ی غیرقابل اتکا نیز به‌درستی عمل کند. در چنین شبکه‌هایی، پیام‌های RPC ممکن است گم شوند، تأخیر داشته باشند یا دوباره ارسال شوند. بنابراین باید منطق Acquire و Release به گونه‌ای تنظیم شود که در صورت بروز چنین مشکلاتی، رفتار قفل نادرست نباشد.

# ۱ مشکل احتمالی در Acquire

در نسخه‌ی قبلی تابع Acquire، فرض می‌شد که اگر کلاینت پاسخ Put را دریافت نکرد، یعنی عملیات انجام نشده است. اما در شبکه‌ی غیرقابل اتکا ممکن است چنین وضعیتی رخ دهد:

- کلاینت درخواست Put را به سرور می‌فرستد.
- سرور درخواست را دریافت و پردازش می‌کند، و قفل را به این کلاینت می‌دهد.
- پاسخ سرور در مسیر گم می‌شود یا خیلی دیر به دست کلاینت می‌رسد.
- کلاینت چون پاسخی دریافت نکرده، تصور می‌کند که قفل را نگرفته و دوباره تلاش می‌کند.

در این حالت اگر تابع Acquire وضعیت فعلی را بررسی نکند، ممکن است کلاینت بی‌پایان تلاش کند قفلی را بگیرد که خودش هم‌اکنون صاحب آن است.

# ۲ راه‌حل: بررسی مالکیت پیش از اقدام

برای حل این مشکل، کافی است در ابتدای حلقه‌ی Acquire بررسی کنیم که آیا مقدار قفل (val) برابر با شناسه‌ی ما (id) است یا نه:

```
if val == lk.id {
    return
}
```

اگر مقدار قفل برابر با شناسه‌ی خود کلاینت باشد، یعنی ما پیش‌تر قفل را گرفته‌ایم، ولی پاسخش را دریافت نکرده‌ایم. بنابراین نیازی به تلاش مجدد نیست و تابع می‌تواند مستقیماً از حلقه خارج شود. این بررسی ساده باعث می‌شود که رفتار قفل در برابر پیام‌های گم‌شده یا تکراری پایدار باقی بماند و از تلاش‌های بی‌فایده جلوگیری شود.

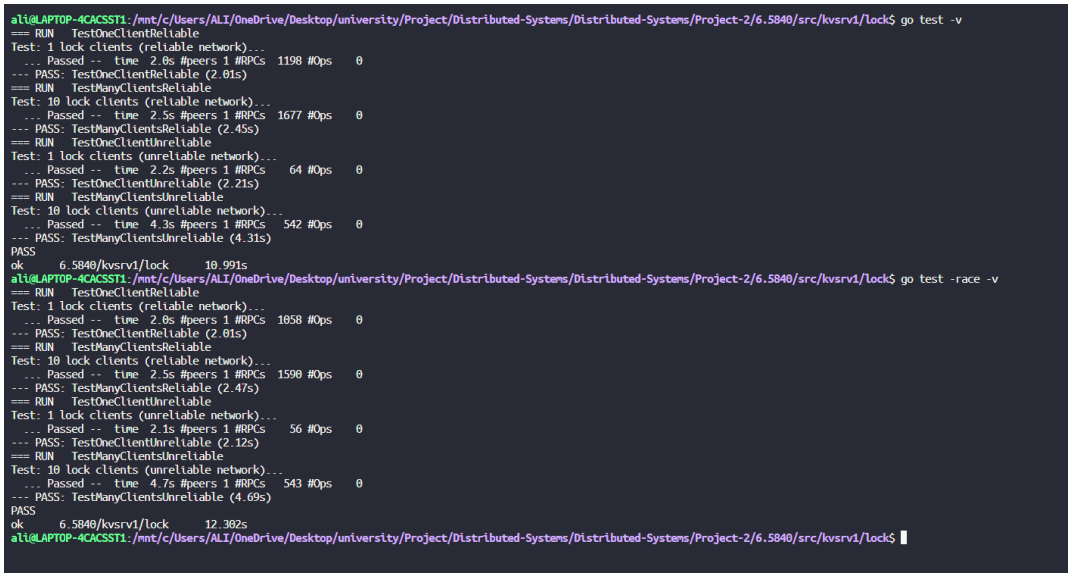
# ۳ چرا Release بدون تغییر باقی ماند؟

تابع Release با همان منطق مرحله قبل باقی مانده و نیازی به تغییر ندارد. دلیل این امر این است که این تابع تنها زمانی قفل را آزاد می‌کند که مطمئن شود کلاینت فعلی مالک قفل است:

```
if val == lk.id {
    lk.ck.Put(lk.name, "", ver)
}
```

نکته مهم اینجاست که در قدم سوم، متد Put به گونه‌ای پیاده‌سازی شده که اگر درخواست اولیه پاسخی دریافت نکند، با استفاده از تابع retransmit درخواست را به‌صورت مکرر ارسال می‌کند تا زمانی که بالاخره پاسخی دریافت شود. این طراحی تضمین می‌کند که عملیات Put در نهایت روی سرور اجرا خواهد شد. بنابراین در تابع Release، نیازی به بررسی یا کنترل اضافی وجود ندارد، چون دستور آزادسازی قفل حتماً به سرور خواهد رسید و اجرا خواهد شد.

# ۴ تست قدم چهارم




شکل ۴: نتیجه اجرای تست‌های قدم چهارم

# مراجع