GroveKV is:

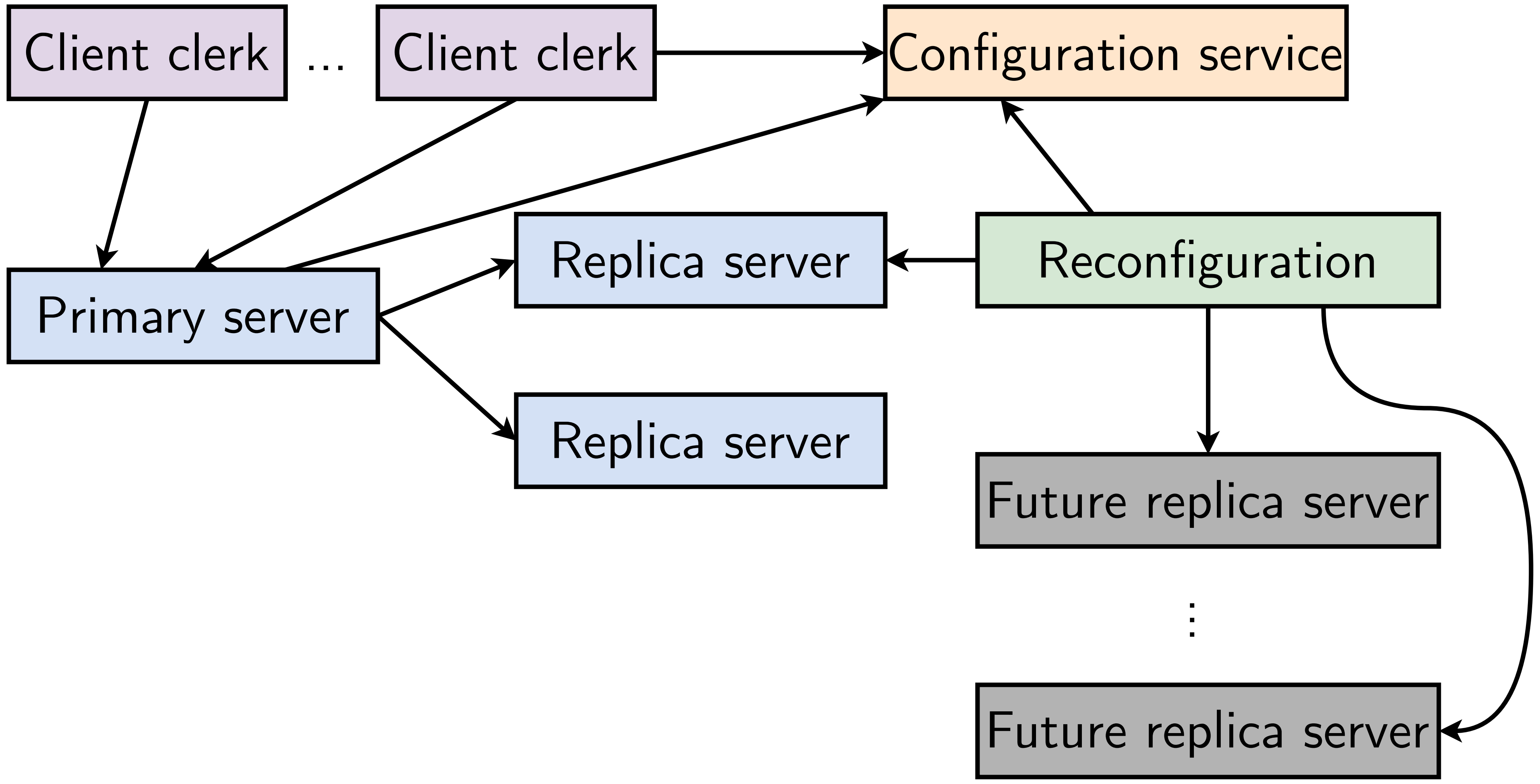
a **fault-tolerant**, **linearizable key-value service**

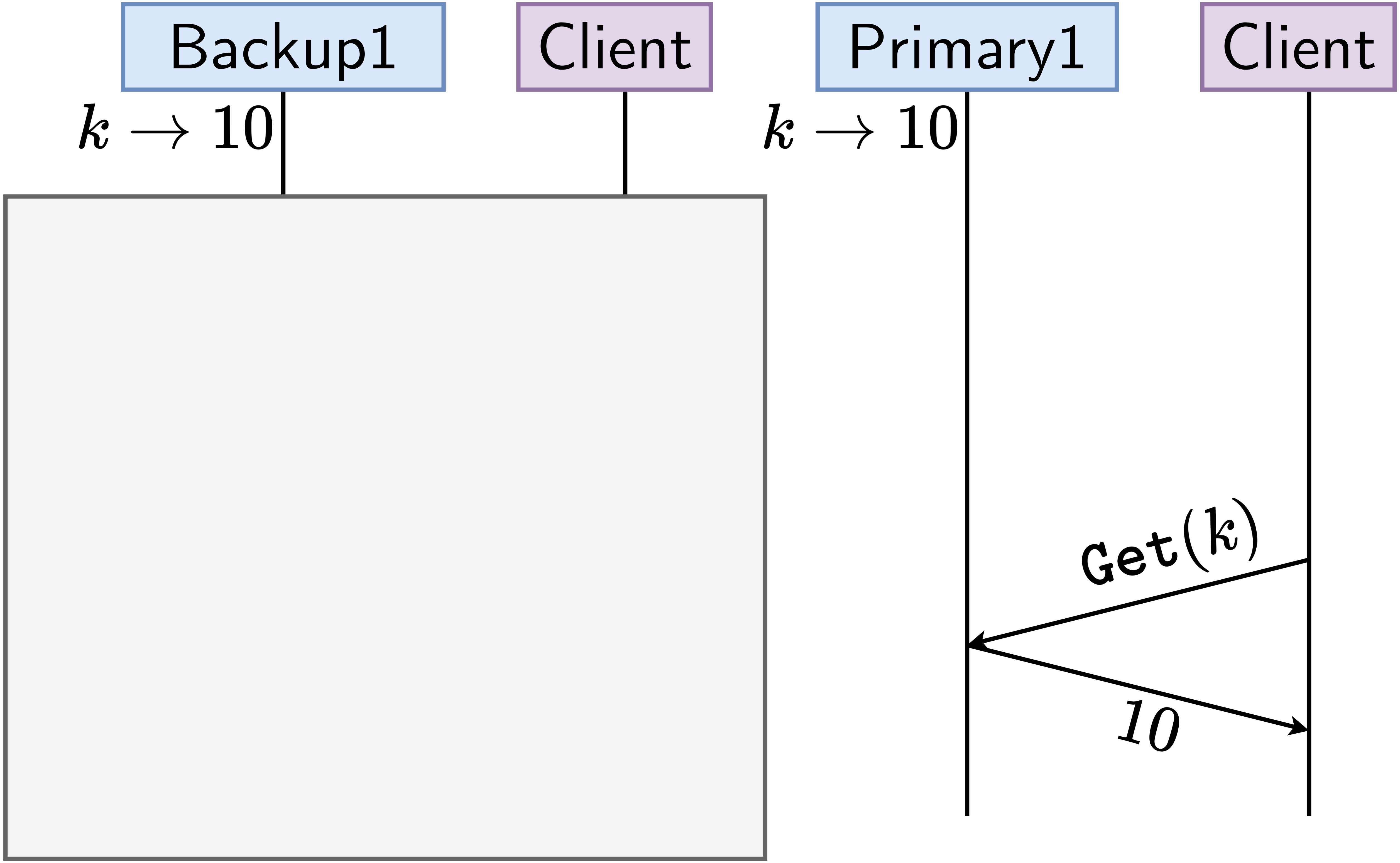
`Put(k,v)` and `Get(k)`

Crash-safe and reconfigurable

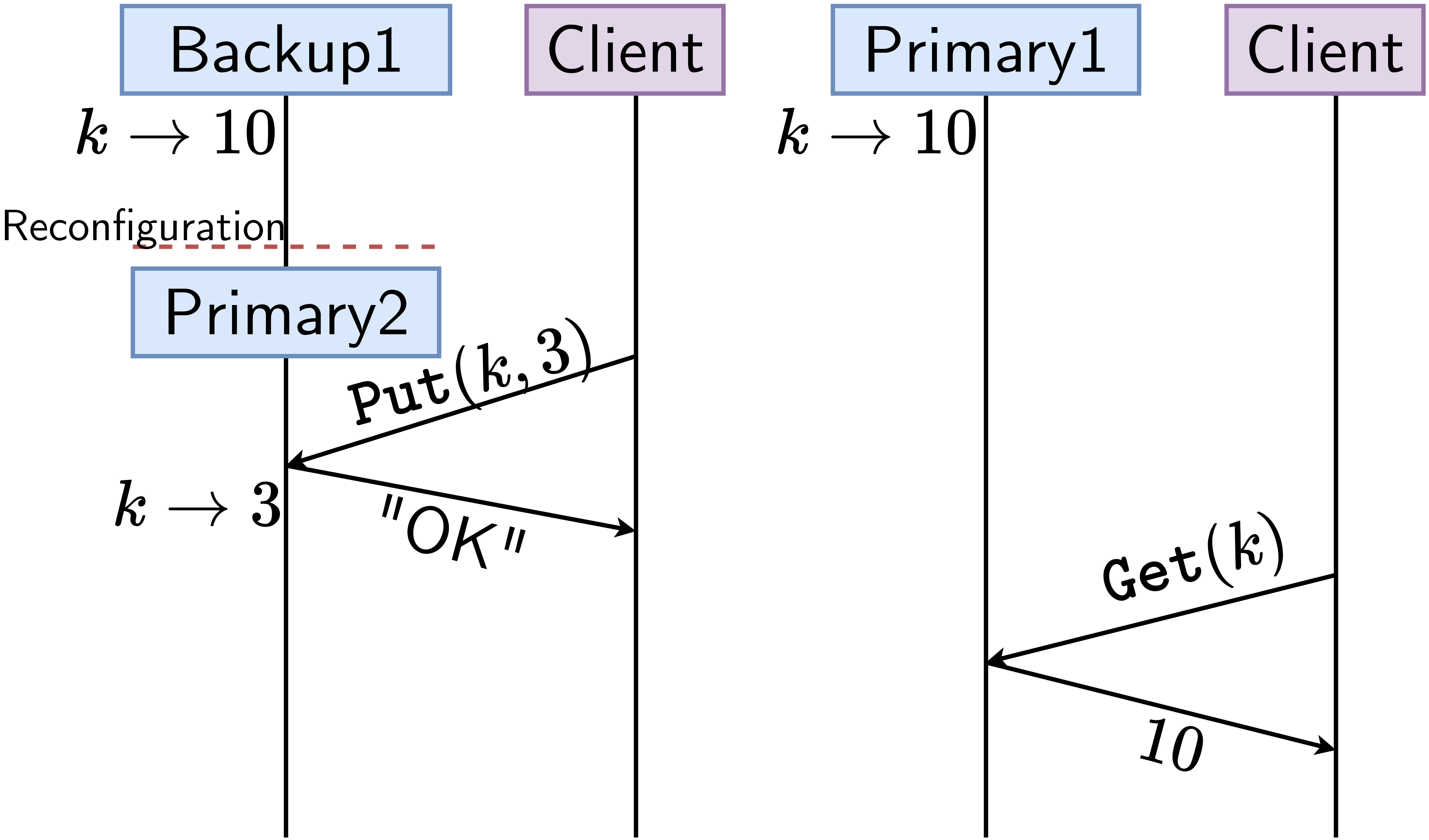
\+ Lease-based reads

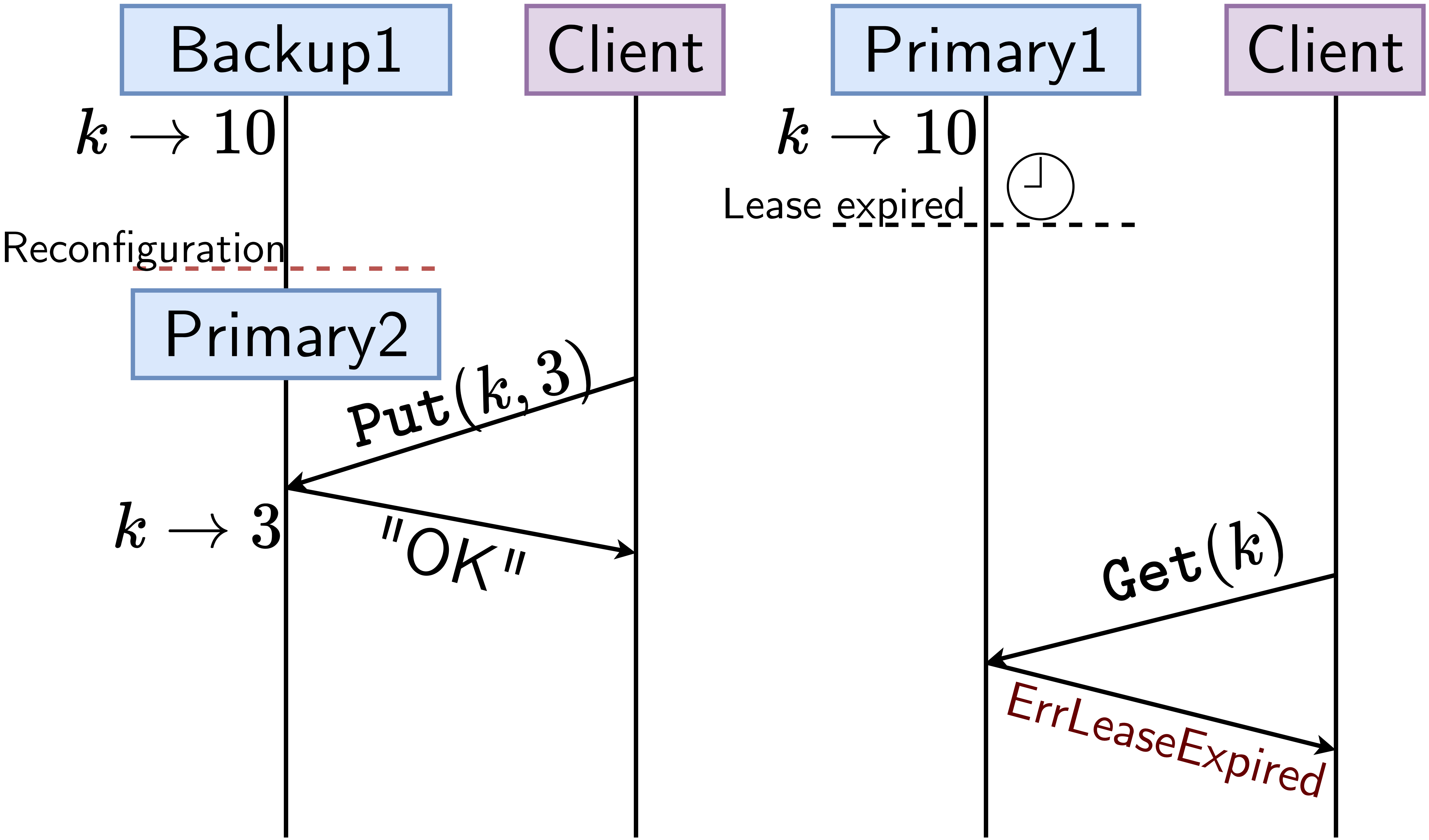
How does GroveKV work?





Backup1
Client
Primary1
Client
$k \to 10$
$k \to 10$
Get(k)
10


Backup1
Client
Primary1
Client
$k \to 10$
$k \to 10$
Reconfiguration
Primary2
Put$(k, 3)$
$k \to 3$
"OK"
Get(k)
10


Backup1
Client
Primary1
Client
k → 10
k → 10
Lease expired
Reconfiguration
Primary2
Put(k, 3)
k → 3
"OK"
Get(k)
ErrLeaseExpired

Want to do all this correctly

Common approach: tests

E.g.

```
...
clerk.Put("x", "10")
clerk.Put("x", "3")
assert(clerk.Get("x") == "3")
```

Want to do all this correctly

Challenge: lots of "what if" questions

- What if there is a `Get` on a server concurrently with `Put`?
- What if backup crashes and loses operations?
- What if lease expires in the middle of a `Get`?
- What if servers try operations (`Get`? `Put`?) during reconfiguration?
- What if ...?

Want to know:

For any thread interleaving and crashes etc., the code behaves correctly.

↓

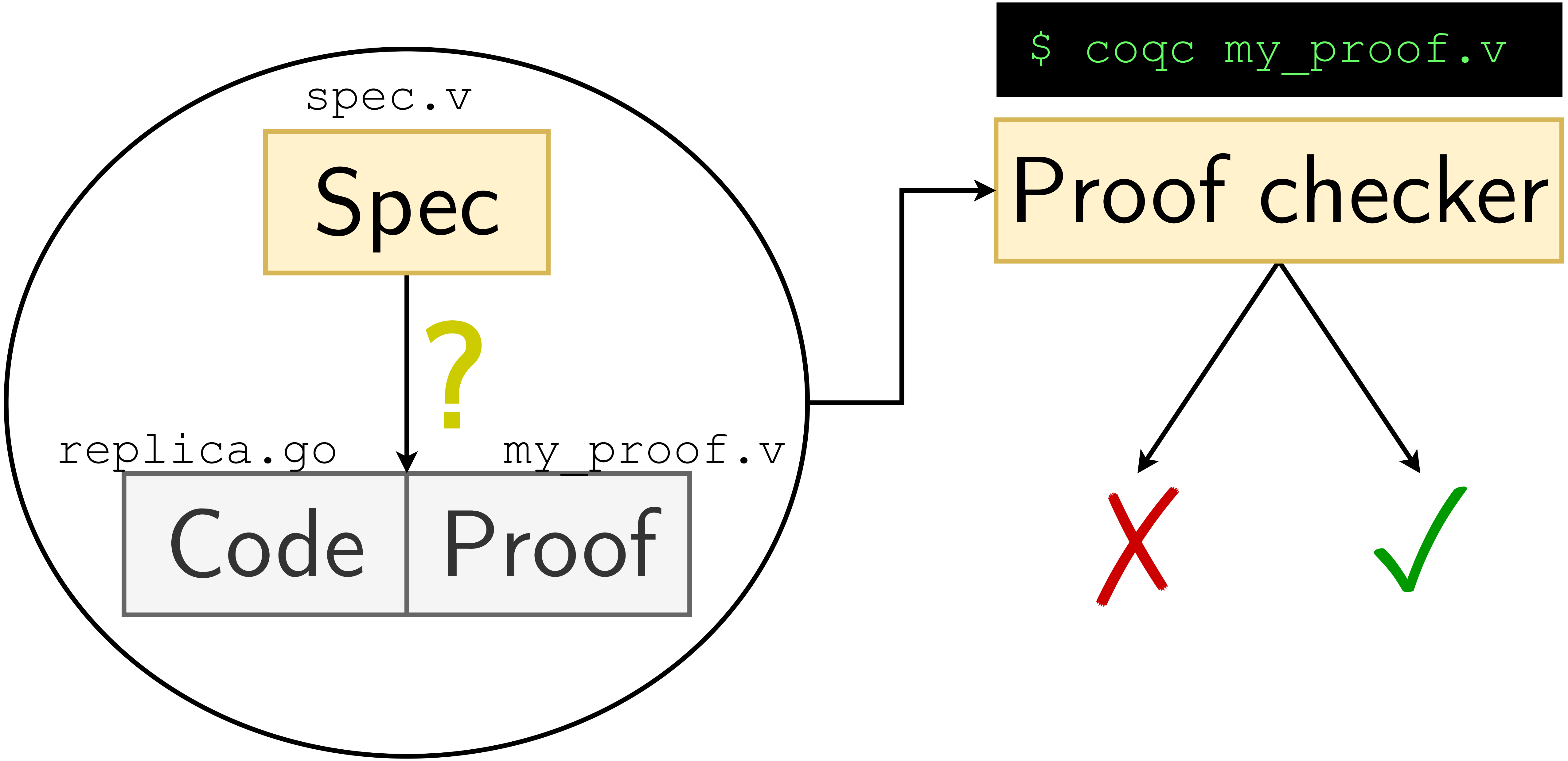
$\forall$ thread interleaving and crashes etc., the code behaves correctly.

Formal verification

Theorem. $\forall$ thread interleaving and crashes etc., the code behaves correctly.

Need a "mathematical model" for how code executes
Need a definition for "correctly"
Use mechanized proof checker

Formal verification



Grove

A framework for proving "∀ theorems", without explicitly writing proofs about interleavings, etc.

Verify system by going line-by-line 😎

Grove

Specs are pre+postconditions (Hoare logic)

E.g. $\{\top\}$ `sort(a)` $\{\texttt{RET}\ b,\ \mathrm{is\_sorted}(b)\}$

Precondition   Code   Postcondition

$\{\mathrm{is\_sorted}\ a\}$ `bsearch(a, x)` $\{\texttt{RET}\ i, a[i] = x\}$

How to specify GroveKV?

Maybe:

$\{\top\}$ **`clerk.Get(k)`** $\{\textbf{RET } v, \texttt{kvmap[k]} = v\}$?

$\{\top\}$ **`clerk.Put(k, v)`** $\{\texttt{kvmap[k]} = v\}$?

What exactly is **`kvmap`**? Does this deal with concurrent operations?

Concurrent separation logic: *ownership of resources*

Concurrent Separation Logic

Pre/post condition describe what is logically *owned* by the running code

"Heap points-to"

$\underbrace{x \mapsto v}$ denotes ownership of address x

$x \mapsto v * y \mapsto w$ denotes separate ownership

Similar to ownership in Rust

GroveKV top-level spec

$$\{k \mapsto_{\mathrm{kv}} w\}\ \texttt{clerk.Put(k, v)}\ \{k \mapsto_{\mathrm{kv}} v\}$$

$$\{k \mapsto_{\mathrm{kv}} v\}\ \texttt{clerk.Get(k)}\ \{\mathrm{ret}\, v, k \mapsto_{\mathrm{kv}} v\}$$

$$x \mapsto_{\mathrm{kv}} \texttt{"0"} * y \mapsto_{\mathrm{kv}} \texttt{"0"}$$

```
ck.Put(x, "1")

ck.Put(y, "2")
go f(...)
```

```
ck.Put(x, "5")                         |   ck.Put(y, "3")

assert(ck.Get(x) == "5")               |   assert(ck.Get(y) == "3")
```

$x \mapsto_{\text{kv}} \texttt{"0"} * y \mapsto_{\text{kv}} \texttt{"0"}$

```
ck.Put(x, "1")
```

$x \mapsto_{\text{kv}} \texttt{"1"} * y \mapsto_{\text{kv}} \texttt{"0"}$

```
ck.Put(y, "2")
go f(...)
```

`ck.Put(x, "5")`	`ck.Put(y, "3")`
`assert(ck.Get(x) == "5")`	`assert(ck.Get(y) == "3")`

$$x \mapsto_{\text{kv}} \texttt{"0"} * y \mapsto_{\text{kv}} \texttt{"0"}$$

```
ck.Put(x, "1")
```

$$x \mapsto_{\text{kv}} \texttt{"1"} * y \mapsto_{\text{kv}} \texttt{"0"}$$

```
ck.Put(y, "2")
```

$$x \mapsto_{\text{kv}} \texttt{"1"} * y \mapsto_{\text{kv}} \texttt{"2"}$$

```
go f(...)
```

```
ck.Put(x, "5")                        |   ck.Put(y, "3")

assert(ck.Get(x) == "5")              |   assert(ck.Get(y) == "3")
```

$x \mapsto_{\mathrm{kv}} \texttt{"0"} * y \mapsto_{\mathrm{kv}} \texttt{"0"}$

```
ck.Put(x, "1")
```

$x \mapsto_{\mathrm{kv}} \texttt{"1"} * y \mapsto_{\mathrm{kv}} \texttt{"0"}$

```
ck.Put(y, "2")
go f(...)
```

$x \mapsto_{\mathrm{kv}} \texttt{"1"} * y \mapsto_{\mathrm{kv}} \texttt{"2"}$

$x \mapsto_{\mathrm{kv}} \texttt{"1"}$

```
ck.Put(x, "5")
```

$x \mapsto_{\mathrm{kv}} \texttt{"5"}$

```
assert(ck.Get(x) == "5")
```

$x \mapsto_{\mathrm{kv}} \texttt{"5"}$

$y \mapsto_{\mathrm{kv}} \texttt{"2"}$

```
ck.Put(y, "3")
```

$y \mapsto_{\mathrm{kv}} \texttt{"3"}$

```
assert(ck.Get(y) == "3")
```

$y \mapsto_{\mathrm{kv}} \texttt{"3"}$

What about specs for RPCs?

$\{\ldots\}$ **`ApplyAsBackupRPC(e, index, op)`** $\{???\}$

Idea from CSL: *ghost resources*

Ghost state useful for specification & proof

Append-only list ghost resource

$a \overset{\text{list}}{\mapsto} \ell$: ownership of append-only list

$a \overset{\text{list}}{\sqsupseteq} \ell$: knowledge of list lower-bound

$a \overset{\text{list}}{\mapsto}_{\Box} \ell$: knowledge of frozen list

Primary/backup resources

Server j owns $\texttt{accepted}_j[e] \overset{\text{list}}{\mapsto} ops$

Replication invariant:

$$\exists \ell\ e, \texttt{committed} \overset{\text{list}}{\mapsto} \ell *$$
$$\texttt{accepted}_0[e] \overset{\text{list}}{\sqsupseteq} \ell * \cdots * \texttt{accepted}_n[e] \overset{\text{list}}{\sqsupseteq} \ell$$

ApplyAsBackupRPC spec

$\{\ldots\}$

```
  server[j].ApplyAsBackupRPC(e, index, op)
```

$$\left\{ \texttt{accepted}_j[e] \overset{\text{list}}{\sqsupseteq} ops + [op] \right\}$$

Captures the "promise" made by RPC

```
Proof code.
 mutex_lock.
 ....
 update (a:=ops+[op]).
 get_lower_bound a.
 mutex_unlock with a.
 applyBackup_spec.
 applyBackup_spec.
 ...
 applyBackup_spec.
 open I_rep.
 update (c:=ops+[op]).
 get_lower_bound c.
 close I_rep.
 done.
Qed.
```

```
s.mutex.Lock()
nextIndex := s.nextIndex
e := s.epoch
s.nextIndex += 1
res := s.stateLogger.LocalApply(op)
s.mutex.Unlock()

for j := 0; j < len(s.backupClerks); j++ {
    s.backupClerks[j].ApplyAsBackupRPC(e,
nextIndex, op)
}

return res
```

```
Proof code.
 mutex_lock.
 ....
 update (a:=ops+[op]).
 get_lower_bound a.
 mutex_unlock with a.
 applyBackup_spec.
 applyBackup_spec.
 ...
 applyBackup_spec.
 open I_rep.
 update (c:=ops+[op]).
 get_lower_bound c.
 close I_rep.
 done.
Qed.
```

$\mathbf{accepted}_0[e] \overset{\text{list}}{\mapsto} \mathit{ops}$

...

```
s.mutex.Lock()
nextIndex := s.nextIndex
e := s.epoch
s.nextIndex += 1
res := s.stateLogger.LocalApply(op)
s.mutex.Unlock()

for j := 0; j < len(s.backupClerks); j++ {
    s.backupClerks[j].ApplyAsBackupRPC(e,
nextIndex, op)
}

return res
```

```
Proof code.
 mutex_lock.
 ....
 update (a:=ops+[op]).
 get_lower_bound a.
 mutex_unlock with a.
 applyBackup_spec.
 applyBackup_spec.
 ...
 applyBackup_spec.
```
open I_{rep}.
```
 update (c:=ops+[op]).
 get_lower_bound c.
```
close I_{rep}.
```
 done.
Qed.
```

$\mathbf{accepted}_0[e] \overset{\text{list}}{\mapsto} ops$

...

```
s.mutex.Lock()
nextIndex := s.nextIndex
e := s.epoch
s.nextIndex += 1
res := s.stateLogger.LocalApply(op)
s.mutex.Unlock()

for j := 0; j < len(s.backupClerks); j++ {
    s.backupClerks[j].ApplyAsBackupRPC(e,
nextIndex, op)
}

return res
```

```
Proof code.
 mutex_lock.
 ....
 update (a:=ops+[op]).
 get_lower_bound a.
 mutex_unlock with a.
 applyBackup_spec.
 applyBackup_spec.
 ...
 applyBackup_spec.
 open I_rep.
 update (c:=ops+[op]).
 get_lower_bound c.
 close I_rep.
 done.
Qed.
```

$\mathbf{accepted}_0[e] \overset{\text{list}}{\mapsto} ops + [op]$

...

```
s.mutex.Lock()
nextIndex := s.nextIndex
e := s.epoch
s.nextIndex += 1
res := s.stateLogger.LocalApply(op)
s.mutex.Unlock()

for j := 0; j < len(s.backupClerks); j++ {
    s.backupClerks[j].ApplyAsBackupRPC(e,
nextIndex, op)
}

return res
```

```
Proof code.
 mutex_lock.
 ....
 update (a:=ops+[op]).
 get_lower_bound a.
 mutex_unlock with a.
 applyBackup_spec.
 applyBackup_spec.
 ...
 applyBackup_spec.
 open I_rep.
 update (c:=ops+[op]).
 get_lower_bound c.
 close I_rep.
 done.
Qed.
```

$$\mathbf{accepted}_0[e] \overset{\text{list}}{\mapsto} ops + [op]$$

$$\mathbf{accepted}_0[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$$

...

```
s.mutex.Lock()
nextIndex := s.nextIndex
e := s.epoch
s.nextIndex += 1
res := s.stateLogger.LocalApply(op)
s.mutex.Unlock()

for j := 0; j < len(s.backupClerks); j++ {
    s.backupClerks[j].ApplyAsBackupRPC(e,
nextIndex, op)
}

return res
```

```
Proof code.
 mutex_lock.
 ....
 update (a:=ops+[op]).
 get_lower_bound a.
 mutex_unlock with a.
```

$\mathbf{accepted}_0[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$

```
 applyBackup_spec.
 applyBackup_spec.
 ...
 applyBackup_spec.
 open I_rep.
 update (c:=ops+[op]).
 get_lower_bound c.
 close I_rep.
 done.
Qed.
```

```
s.mutex.Lock()
nextIndex := s.nextIndex
e := s.epoch
s.nextIndex += 1
res := s.stateLogger.LocalApply(op)
s.mutex.Unlock()

for j := 0; j < len(s.backupClerks); j++ {
    s.backupClerks[j].ApplyAsBackupRPC(e,
nextIndex, op)
}

return res
```

```
Proof code.
 mutex_lock.
 ....
 update (a:=ops+[op]).
 get_lower_bound a.
 mutex_unlock with a.
 applyBackup_spec.
 applyBackup_spec.
 ...
 applyBackup_spec.
 open I_rep.
 update (c:=ops+[op]).
 get_lower_bound c.
 close I_rep.
 done.
Qed.
```

$$\textbf{accepted}_0[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$$

$$\textbf{accepted}_1[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$$

...

```
s.mutex.Lock()
nextIndex := s.nextIndex
e := s.epoch
s.nextIndex += 1
res := s.stateLogger.LocalApply(op)
s.mutex.Unlock()

for j := 1; j < len(s.backupClerks); j++ {
    s.backupClerks[j].ApplyAsBackupRPC(e,
nextIndex, op)
}

return res
```

```
Proof code.
 mutex_lock.
 ....
 update (a:=ops+[op]).
 get_lower_bound a.
 mutex_unlock with a.
 applyBackup_spec.
 applyBackup_spec.
 ...
 applyBackup_spec.
 open I_rep.
 update (c:=ops+[op]).
 get_lower_bound c.
 close I_rep.
 done.
Qed.
```

$$\mathbf{accepted}_0[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$$

$$\mathbf{accepted}_1[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$$

$$\mathbf{accepted}_2[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$$

```
s.mutex.Lock()
nextIndex := s.nextIndex
e := s.epoch
s.nextIndex += 1
res := s.stateLogger.LocalApply(op)
s.mutex.Unlock()

for j := 2; j < len(s.backupClerks); j++ {
    s.backupClerks[j].ApplyAsBackupRPC(e,
nextIndex, op)
}

return res
```

```
Proof code.
 mutex_lock.
 ....
 update (a:=ops+[op]).
 get_lower_bound a.
 mutex_unlock with a.
 applyBackup_spec.
 applyBackup_spec.
 ...
 applyBackup_spec.
 open I_rep.
 update (c:=ops+[op]).
 get_lower_bound c.
 close I_rep.
 done.
Qed.
```

$$\mathbf{accepted}_0[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$$
$$\mathbf{accepted}_1[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$$
$$\ldots$$
$$\mathbf{accepted}_n[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$$

```
s.mutex.Lock()
nextIndex := s.nextIndex
e := s.epoch
s.nextIndex += 1
res := s.stateLogger.LocalApply(op)
s.mutex.Unlock()

for j := 0; j < len(s.backupClerks); j++ {
    s.backupClerks[j].ApplyAsBackupRPC(e,
nextIndex, op)
}

return res
```

```
Proof code.
 mutex_lock.
 ....
 update (a:=ops+[op]).
 get_lower_bound a.
 mutex_unlock with a.
 applyBackup_spec.
 applyBackup_spec.
 ...
 applyBackup_spec.
 open I_rep.
 update (c:=ops+[op]).
 get_lower_bound c.
 close I_rep.
 done.
Qed.
```

$\texttt{accepted}_0[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$ $\texttt{committed} \overset{\text{list}}{\mapsto} ops$

$\texttt{accepted}_1[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$

$\ldots$

$\texttt{accepted}_n[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$

```
s.mutex.Lock()
nextIndex := s.nextIndex
e := s.epoch
s.nextIndex += 1
res := s.stateLogger.LocalApply(op)
s.mutex.Unlock()

for j := 0; j < len(s.backupClerks); j++ {
    s.backupClerks[j].ApplyAsBackupRPC(e,
nextIndex, op)
}

return res
```

```
Proof code.
 mutex_lock.
 ....
 update (a:=ops+[op]).
 get_lower_bound a.
 mutex_unlock with a.
 applyBackup_spec.
 applyBackup_spec.
 ...
 applyBackup_spec.
 open I_rep.
 update (c:=ops+[op]).
 get_lower_bound c.
 close I_rep.
 done.
Qed.
```

$\texttt{accepted}_0[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$ $\texttt{committed} \overset{\text{list}}{\mapsto} ops + [op]$

$\texttt{accepted}_1[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$

...

$\texttt{accepted}_n[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$

```
s.mutex.Lock()
nextIndex := s.nextIndex
e := s.epoch
s.nextIndex += 1
res := s.stateLogger.LocalApply(op)
s.mutex.Unlock()

for j := 0; j < len(s.backupClerks); j++ {
    s.backupClerks[j].ApplyAsBackupRPC(e,
nextIndex, op)
}

return res
```

```
Proof code.
 mutex_lock.
 ....
 update (a:=ops+[op]).
 get_lower_bound a.
 mutex_unlock with a.
 applyBackup_spec.
 applyBackup_spec.
 ...
 applyBackup_spec.
 open I_rep.
 update (c:=ops+[op]).
 get_lower_bound c.
 close I_rep.
 done.
Qed.
```

$$\texttt{accepted}_0[e] \overset{\text{list}}{\sqsupseteq} ops + [op] \quad \texttt{committed} \overset{\text{list}}{\mapsto} ops + [op]$$

$$\texttt{accepted}_1[e] \overset{\text{list}}{\sqsupseteq} ops + [op] \quad \texttt{committed} \overset{\text{list}}{\sqsupseteq} ops + [op]$$

$$\ldots$$

$$\texttt{accepted}_n[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$$

```
s.mutex.Lock()
nextIndex := s.nextIndex
e := s.epoch
s.nextIndex += 1
res := s.stateLogger.LocalApply(op)
s.mutex.Unlock()

for j := 0; j < len(s.backupClerks); j++ {
    s.backupClerks[j].ApplyAsBackupRPC(e,
nextIndex, op)
}

return res
```

```
Proof code.
 mutex_lock.
 ....
 update (a:=ops+[op]).
 get_lower_bound a.
 mutex_unlock with a.
 applyBackup_spec.
 applyBackup_spec.
 ...
 applyBackup_spec.
 open I_rep.
 update (c:=ops+[op]).
 get_lower_bound c.
 close I_rep.
 done.
Qed.
```

$\texttt{accepted}_0[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$ $\texttt{committed} \overset{\text{list}}{\mapsto} ops + [op]$

$\texttt{accepted}_1[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$ $\texttt{committed} \overset{\text{list}}{\sqsupseteq} ops + [op]$

$\dots$

$\texttt{accepted}_n[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$

```
s.mutex.Lock()
nextIndex := s.nextIndex
e := s.epoch
s.nextIndex += 1
res := s.stateLogger.LocalApply(op)
s.mutex.Unlock()

for j := 0; j < len(s.backupClerks); j++ {
    s.backupClerks[j].ApplyAsBackupRPC(e,
nextIndex, op)
}

return res
```

```
Proof code.
 mutex_lock.
 ....
 update (a:=ops+[op]).
 get_lower_bound a.
 mutex_unlock with a.
 applyBackup_spec.
 applyBackup_spec.
 ...
 applyBackup_spec.
```
open I_{rep}.
update (c:=ops+[op]).
get_lower_bound c.
close I_{rep}.
```
 done.
Qed.
```

committed $\overset{\text{list}}{\sqsupseteq} ops + [op]$

```
s.mutex.Lock()
nextIndex := s.nextIndex
e := s.epoch
s.nextIndex += 1
res := s.stateLogger.LocalApply(op)
s.mutex.Unlock()

for j := 0; j < len(s.backupClerks); j++ {
    s.backupClerks[j].ApplyAsBackupRPC(e,
nextIndex, op)
}

return res
```

```
Proof code.
 mutex_lock.
 ....
 update (a:=ops+[op]).
 get_lower_bound a.
 mutex_unlock with a.
 applyBackup_spec.
 applyBackup_spec.
 ...
 applyBackup_spec.
 open I_rep.
 update (c:=ops+[op]).
 get_lower_bound c.
 close I_rep.
 done.
Qed.
```

$$\textbf{committed} \overset{\text{list}}{\sqsupseteq} ops + [op]$$

```
s.mutex.Lock()
nextIndex := s.nextIndex
e := s.epoch
s.nextIndex += 1
res := s.stateLogger.LocalApply(op)
s.mutex.Unlock()

for j := 0; j < len(s.backupClerks); j++ {
    s.backupClerks[j].ApplyAsBackupRPC(e,
nextIndex, op)
}

return res
```

Time-bounded invariants

$$\boxed{\texttt{CurrentEpoch} \mapsto e}_{L}$$

$$L \overset{\text{expires}}{\geq} t \qquad \Longrightarrow \texttt{CurrentEpoch} \mapsto e$$

$$\texttt{GetTimeRange().latest} < t$$

Time-bounded invariants

$$\texttt{CurrentEpoch} \mapsto e$$

$$\texttt{accepted}_0[e] \overset{\text{list}}{\mapsto} ops$$

$$I_{\text{rep}}$$

$$\Longrightarrow \quad \texttt{committed} \overset{\text{list}}{\mapsto} ops'$$

$$ops' \preceq ops$$

$$\exists \ell\ e, \texttt{committed} \overset{\text{list}}{\mapsto} \ell *$$
$$\texttt{accepted}_0[e] \overset{\text{list}}{\sqsupseteq} \ell * \cdots * \texttt{accepted}_n[e] \overset{\text{list}}{\sqsupseteq} \ell$$

What about "what if" questions?

Let's look at a few "what if" scenarios, and see how the proof handles them

What if backup loses operations?

Would be buggy:

`ApplyAsBackupRPC` promises $\texttt{\textbf{accepted}}_j[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$

Backup can't roll back $\texttt{\textbf{accepted}}_j[e] \overset{\text{list}}{\mapsto} ops + [op]$

What if old primary does a **`Put`**?

Would be buggy:

At least one replica promises $\texttt{accepted}_j[e] \overset{\text{list}}{\mapsto_\Box} ops$

New op requires $\texttt{accepted}_j[e] \overset{\text{list}}{\sqsupseteq} ops + [op]$

Replica can't modify $\texttt{accepted}_j[e] \overset{\text{list}}{\mapsto_\Box} ops$

What if lease expires during `Get`?

Not buggy:

Get access to **`CurrentEpoch`** $\mapsto e$ and **`committed`** $\overset{\text{list}}{\mapsto} ops'$ at the moment of **`GetTimeRange()`**

Benefits of verifying GroveKV

Eliminate the need for tests?

Not quite... GroveKV's spec rules out (some) safety bugs

Liveness + performance bugs still possible

(When) are proofs worth it?

Component	Lines of Code	Lines of Proof
Network library	120	Trusted
Filesystem library	50	Trusted
RPC library	161	1,311
Replica server	574	7,986
Reconfiguration	65	803
Configuration server	200	2,048
Clerk	156	878
Append-only file	90	1,345
State logger	134	1,732
Exactly-once operations	128	2,186
Key-value mappings	157	916
Time bounded invariants	–	142
Total	1,835	–
Total verified	1,665	19,347

Figure 5: Lines of Go code and Coq proof for GroveKV.