Formal verification of a concurrent file system

Tej Chajed

VMware Research / UW-Madison

joint work with Joseph Tassarotti, Frans Kaashoek, Nickolai Zeldovich, Ralf Jung, and Mark Theng

Iris workshop
May 24, 2023

Systems software is challenging to get right

Ext4

PostgreSQL

Linux

Systems software is challenging to get right

Applications exercise all corners of the system API

Ext4

PostgreSQL

Linux

Runs on raw hardware: crashes, concurrency, devices

Systems verification is becoming feasible

Microkernels (seL4, CertiKOS)

Cryptography libraries (Fiat Crypto, HACL*)

Distributed systems (IronFleet, Verdi)

File systems (FSCQ, BilbyFS)

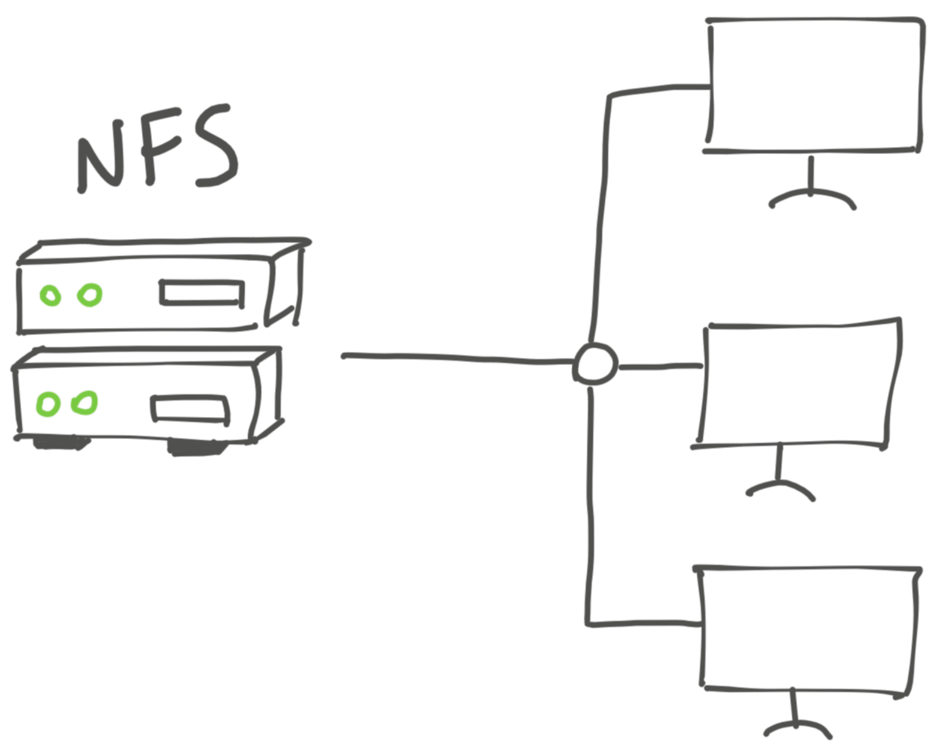
This talk: verifying DaisyNFS

DaisyNFS is a verified, concurrent file system

Built on the **Perennial** logic, based on Iris

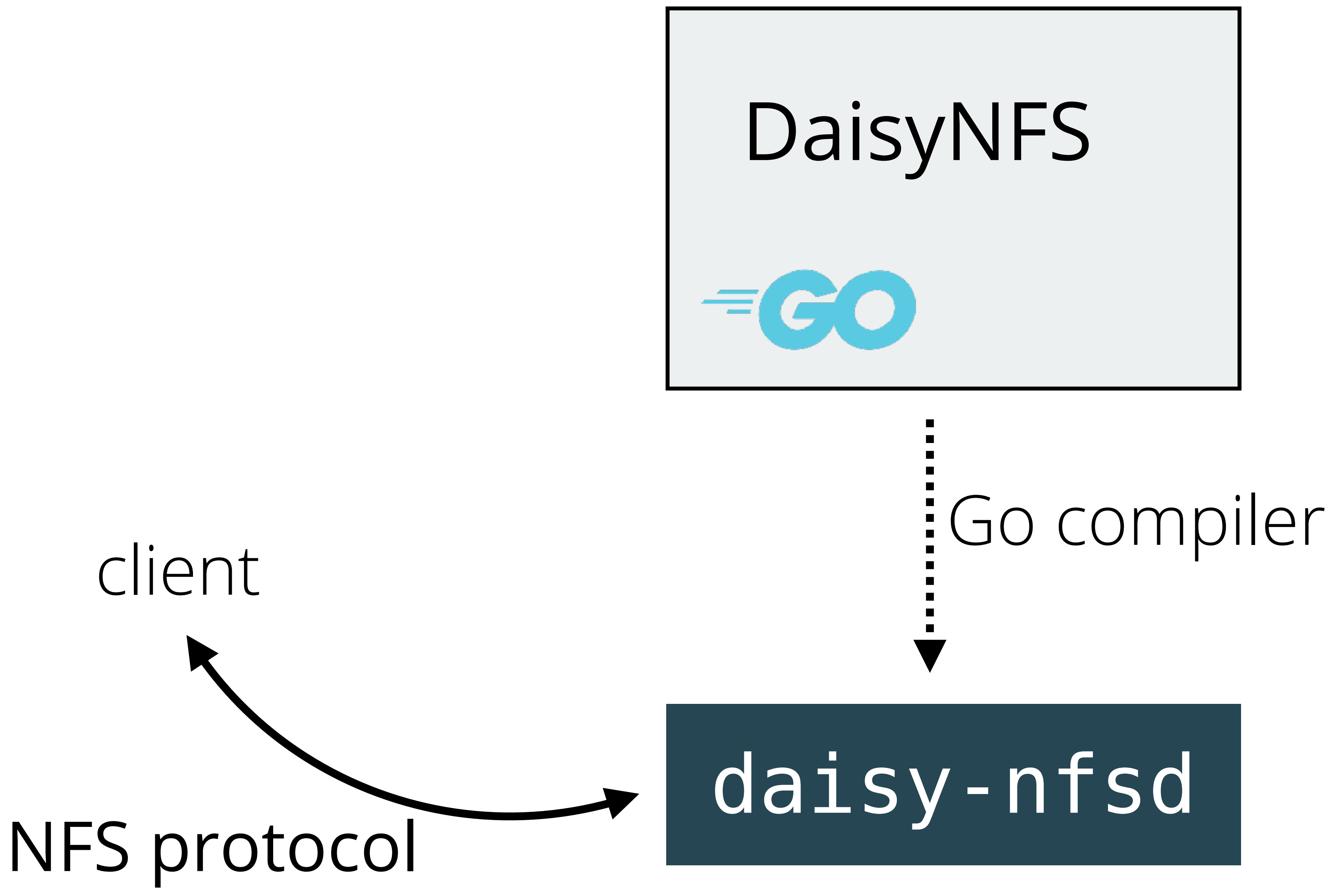
Combines PL and systems techniques

NFS is a good target for verification

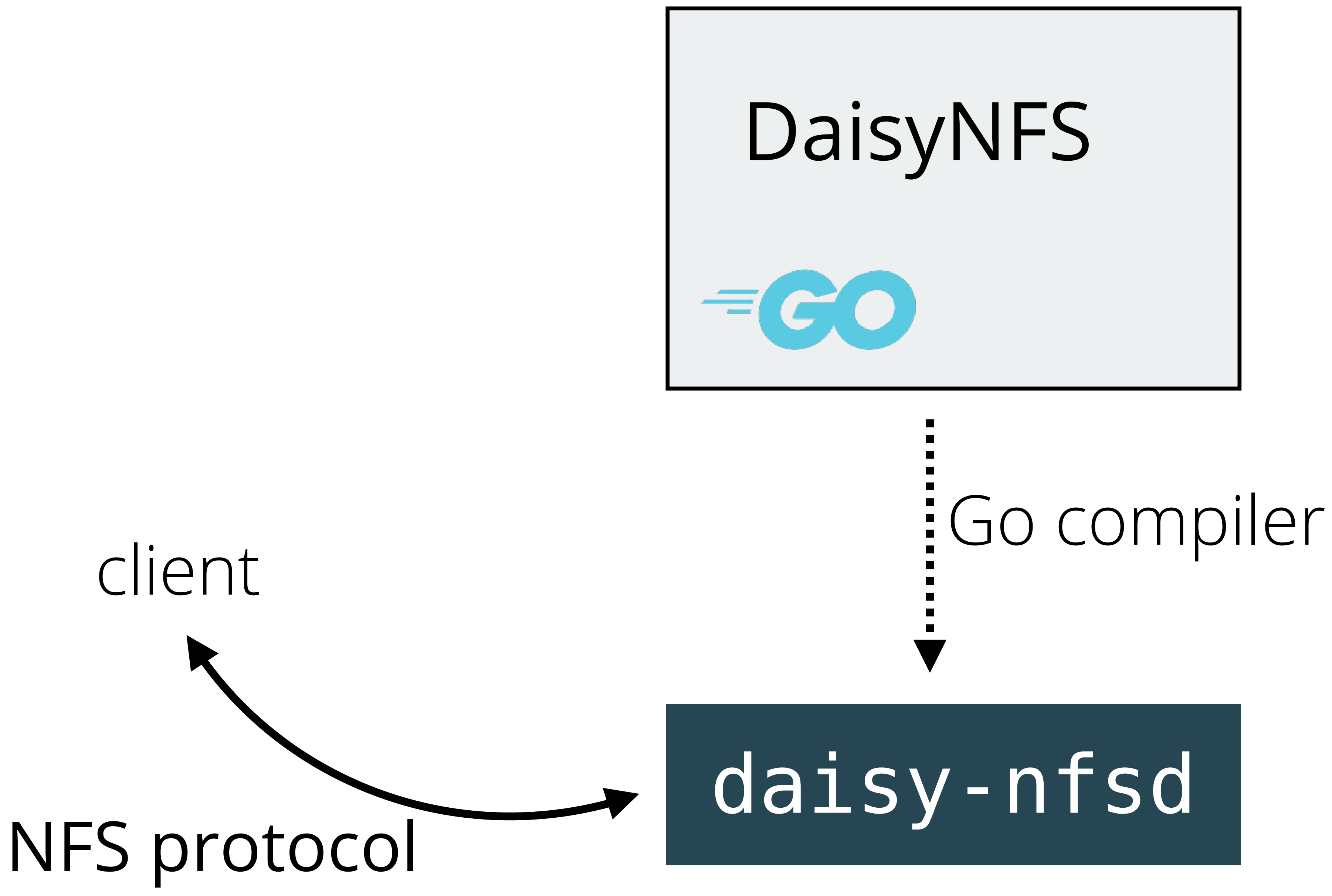


1. **Widely used**
2. **Sophisticated implementations** with concurrency & high performance
3. **Bugs are costly**, especially data loss

DaisyNFS implements an NFS server



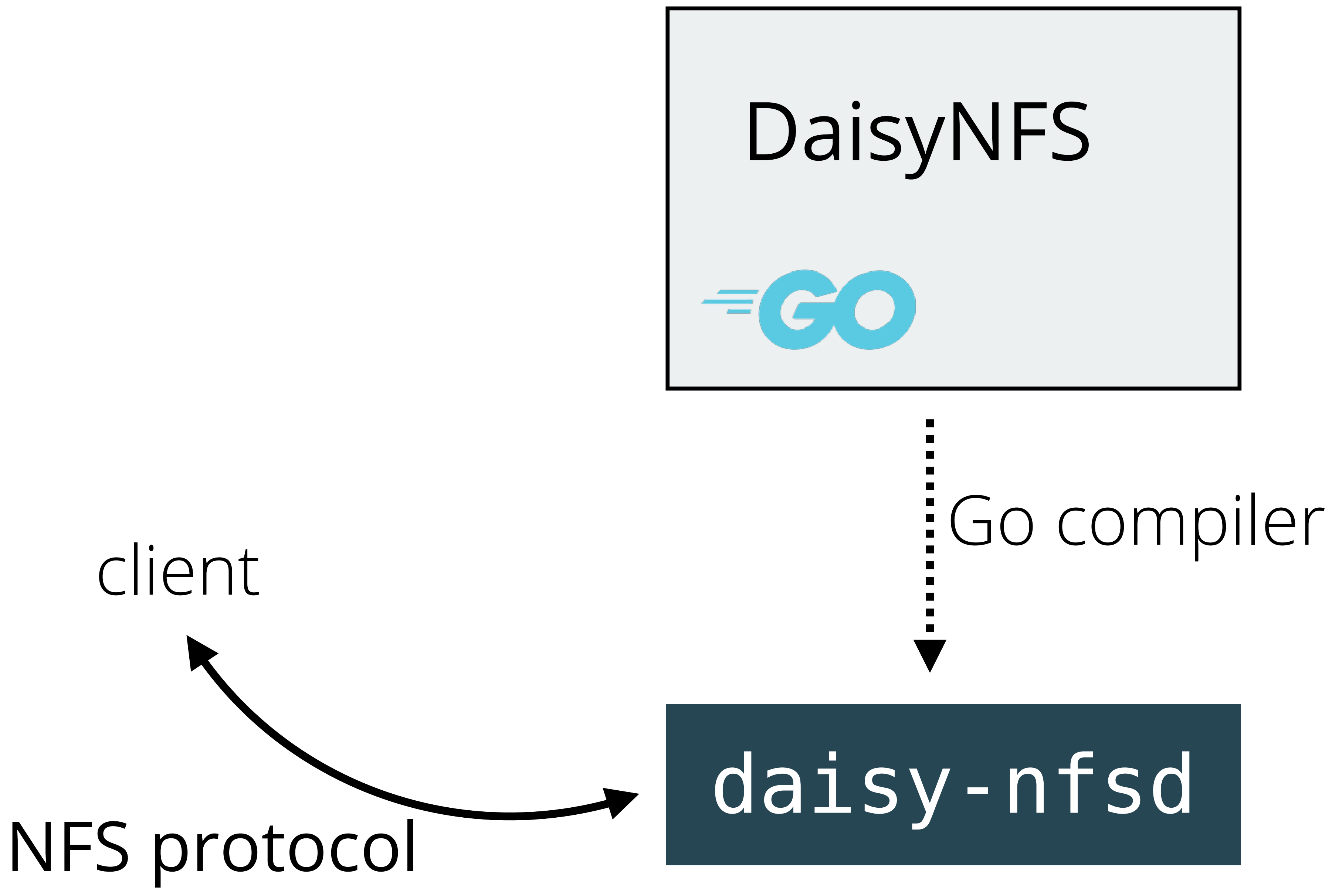
DaisyNFS implements an NFS server



What is Go?

- Popular new language, supported by Google
- Compiled, efficient, good concurrency

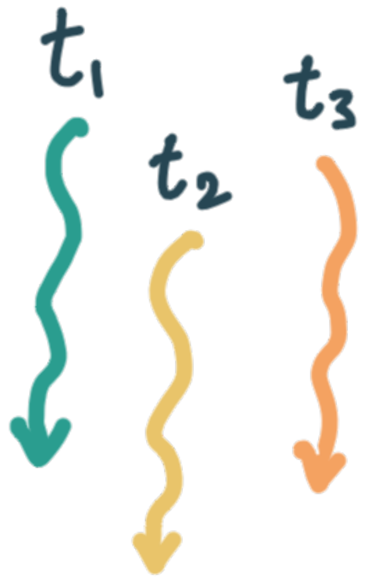
DaisyNFS is a verified NFS server



Theorem (informal): the server correctly implements the NFS protocol.

Challenges in verifying a file system

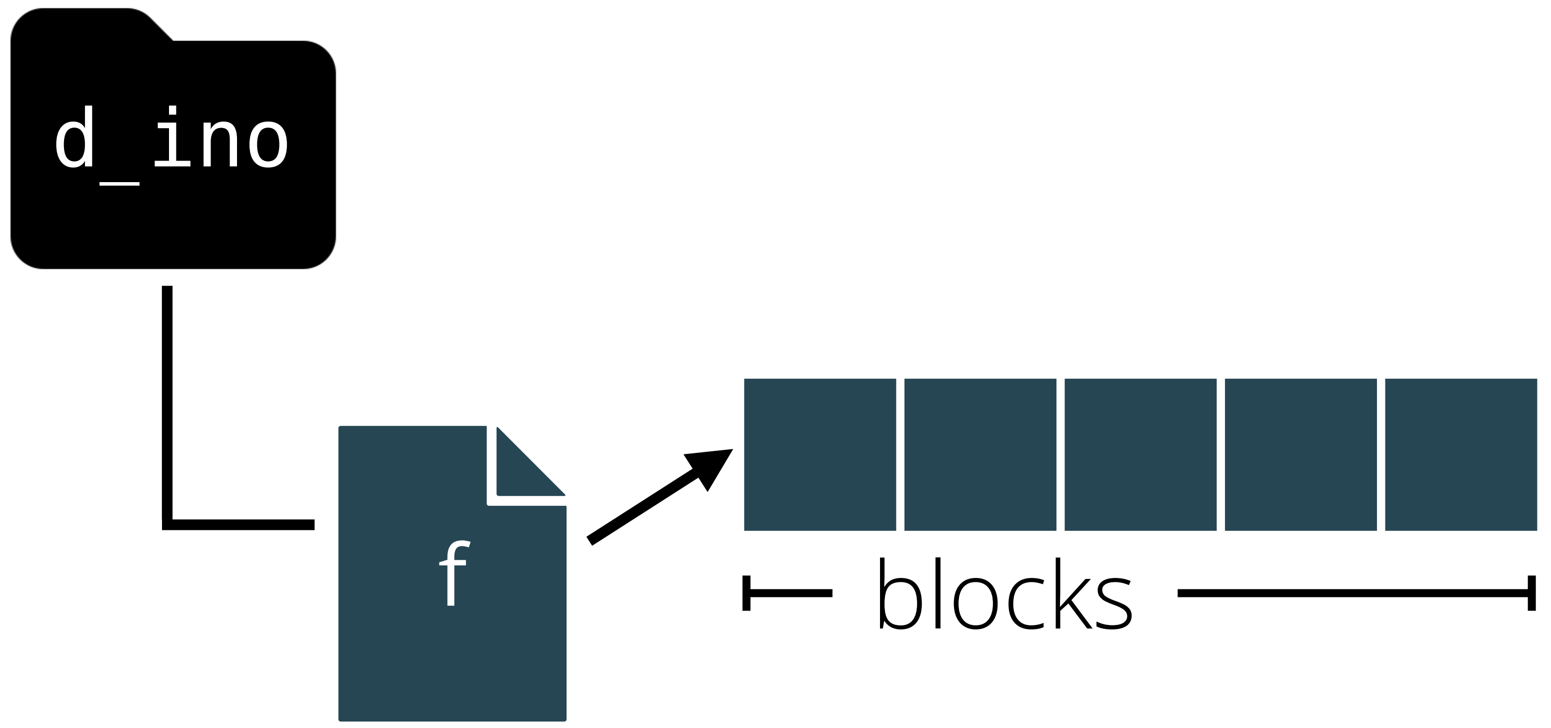
Crashes



Concurrency

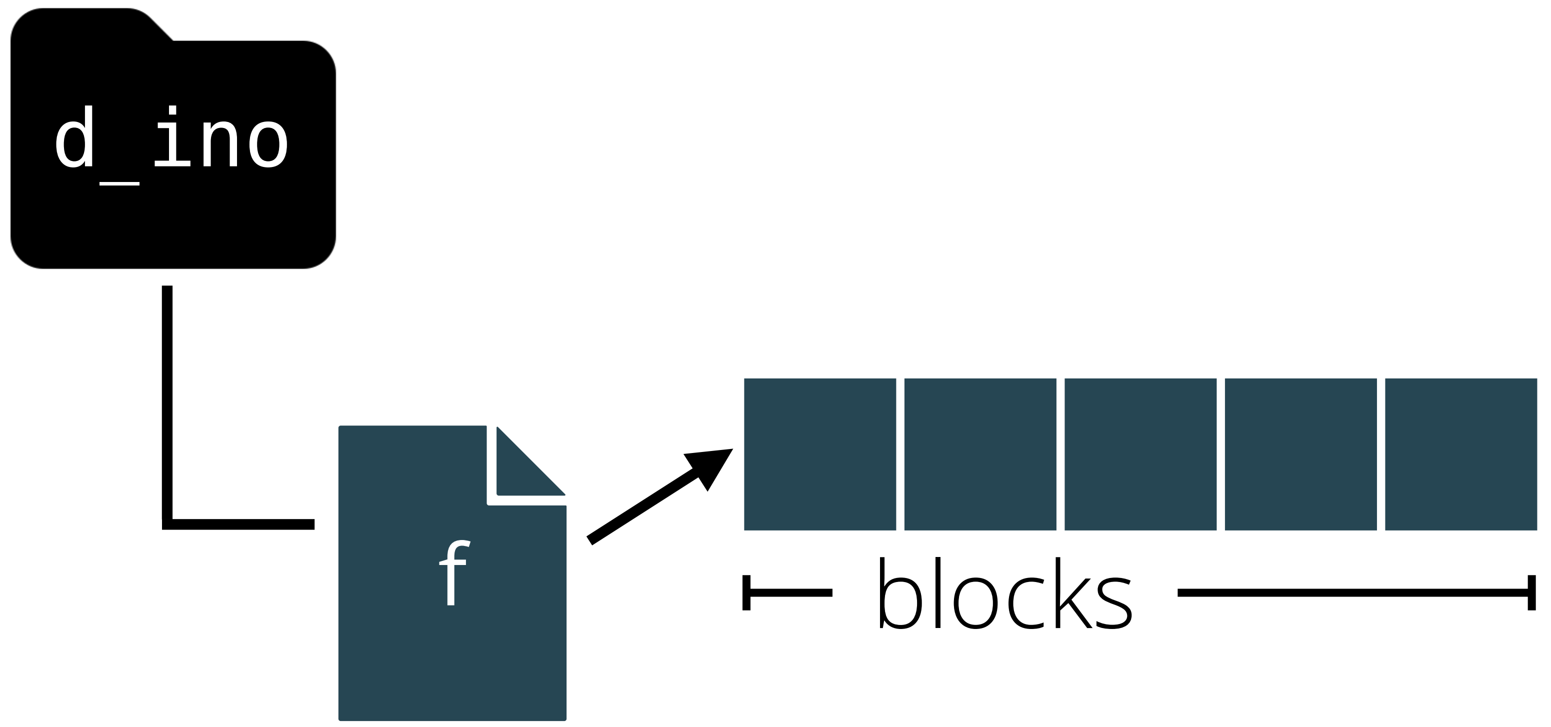
REMOVE has several steps

```
func REMOVE(d_ino: uint64,
            name: []byte) {

  f := unlink(d_ino, name)

  blocks := getBlocks(f)

  free(blocks)

}
```



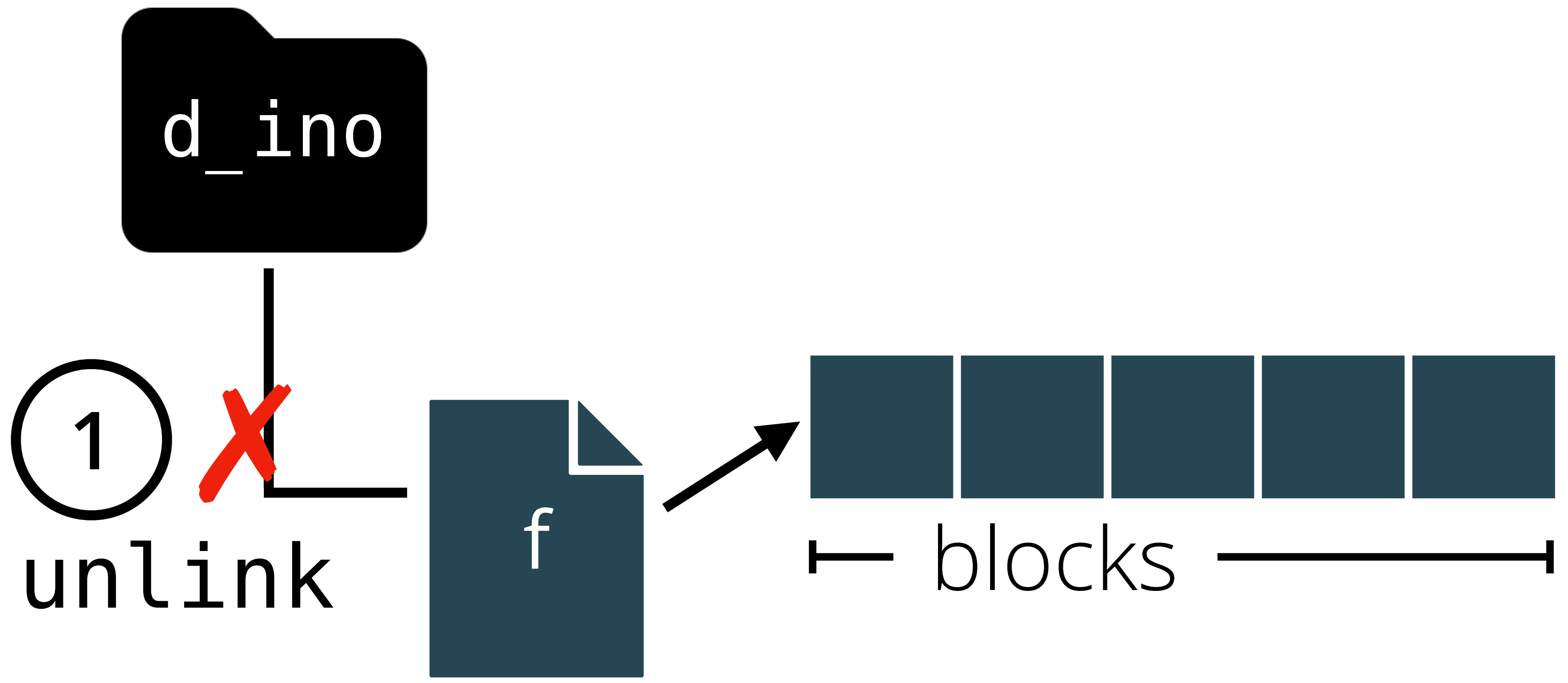
REMOVE has several steps

```
func REMOVE(d_ino: uint64,
            name: []byte) {
① f := unlink(d_ino, name)
② blocks := getBlocks(f)
③ free(blocks)
}
```



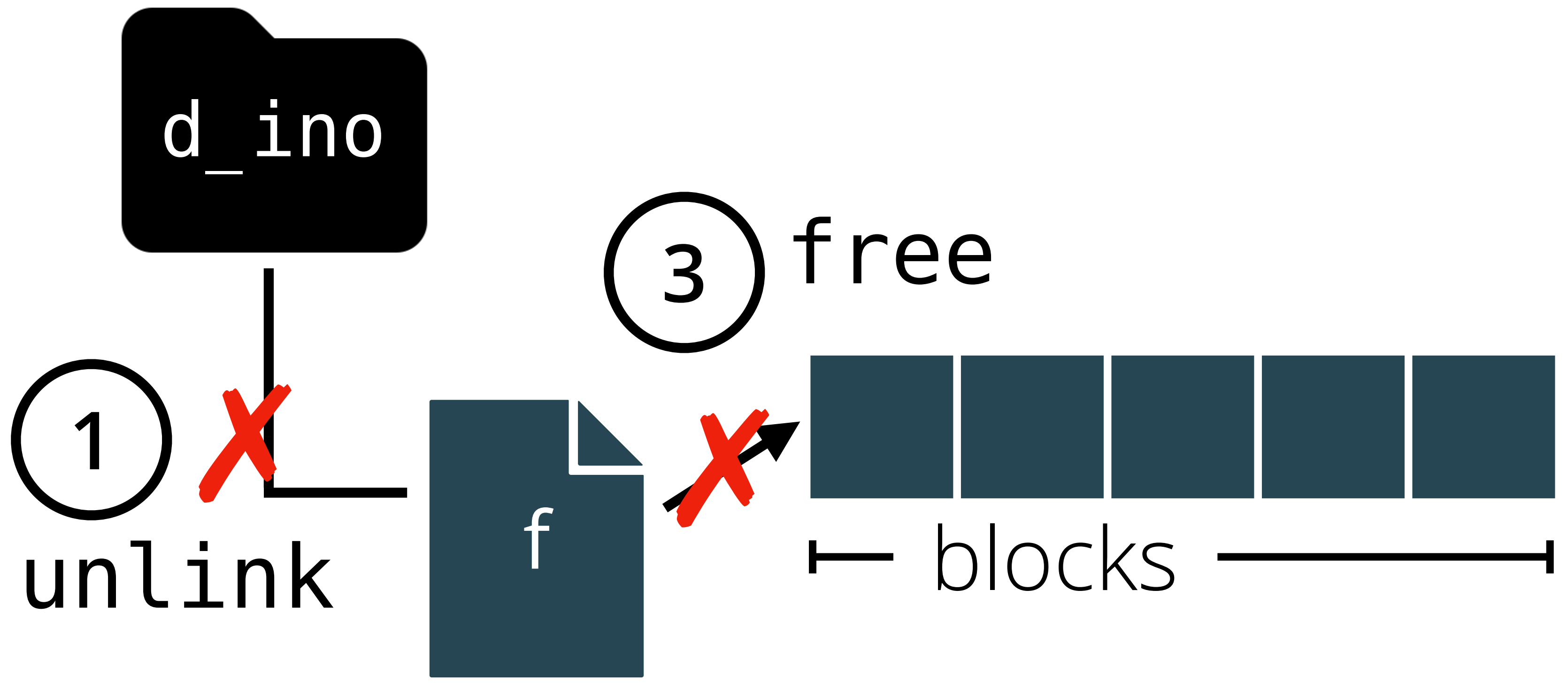
REMOVE has several steps

```
func REMOVE(d_ino: uint64,
            name: []byte) {
① f := unlink(d_ino, name)
② blocks := getBlocks(f)
③ free(blocks)
}
```



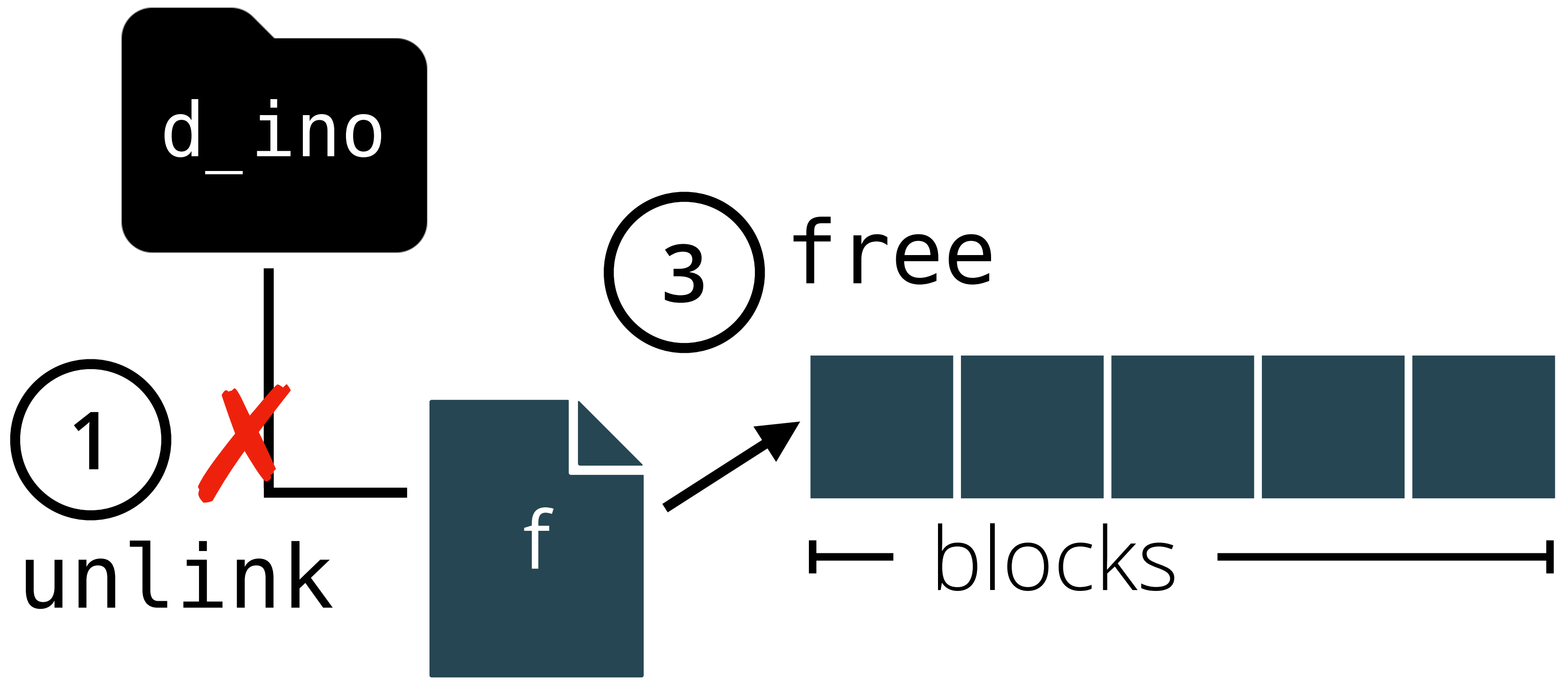
REMOVE has several steps

```
func REMOVE(d_ino: uint64,
            name: []byte) {

① f := unlink(d_ino, name)

② blocks := getBlocks(f)

③ free(blocks)

}
```



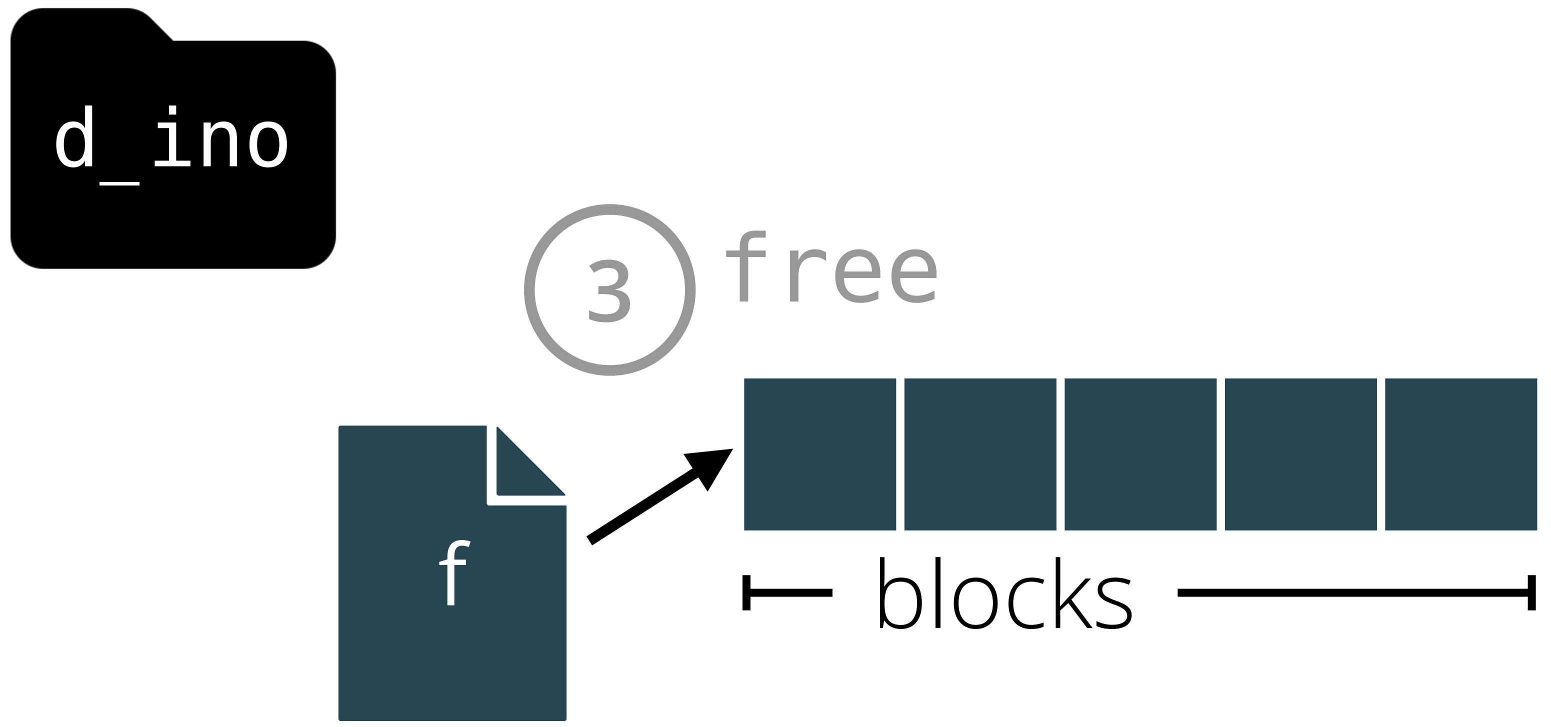
Crashes create subtle bugs

```
func REMOVE(d_ino: uint64,
            name: []byte) {

  ① f := unlink(d_ino, name)
  ───────────────────────────── crash
  ② blocks := getBlocks(f)

  ③ free(blocks)

}
```



Crashes create subtle bugs

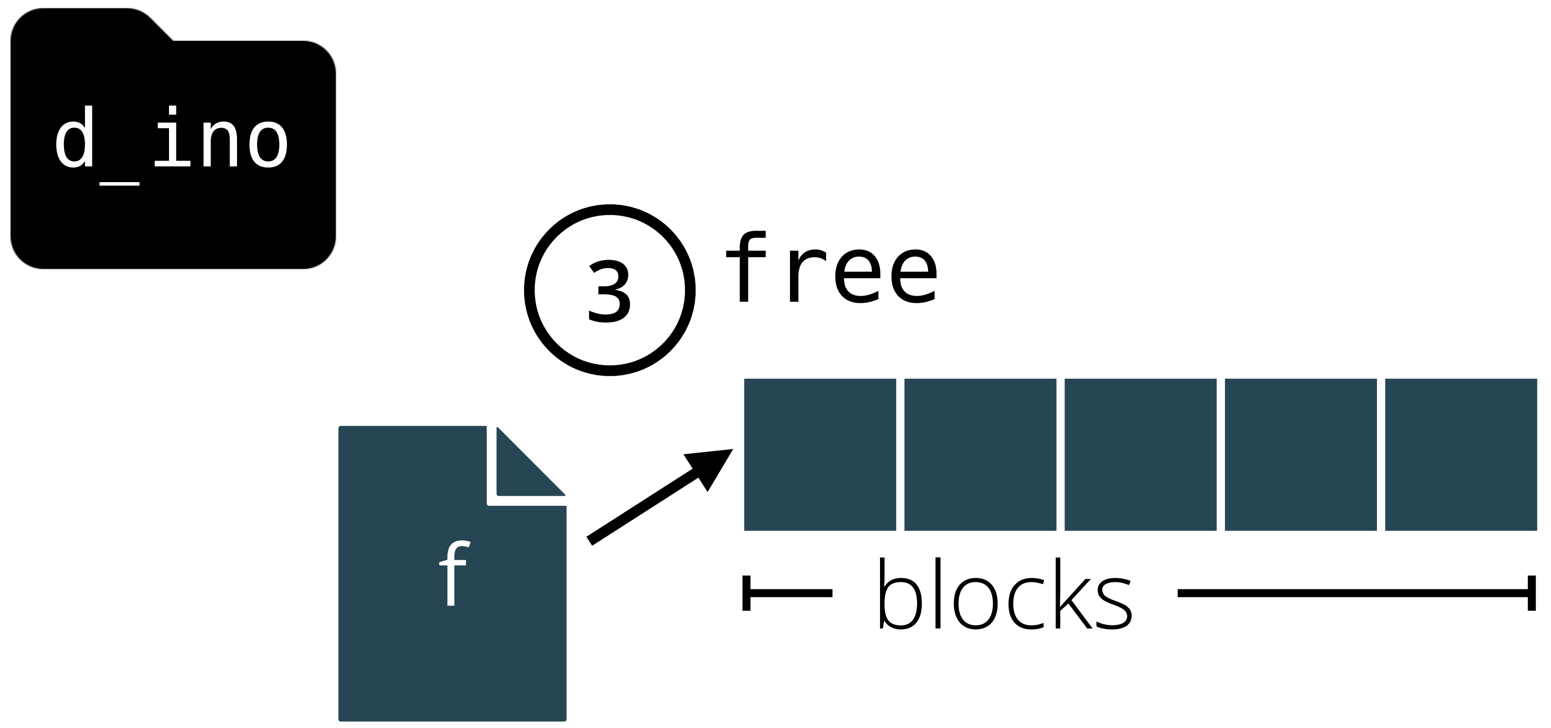
```
func REMOVE(d_ino: uint64,
            name: []byte) {

(1) f := unlink(d_ino, name)
        crash
(2) blocks := getBlocks(f)

(3) free(blocks)

}
```

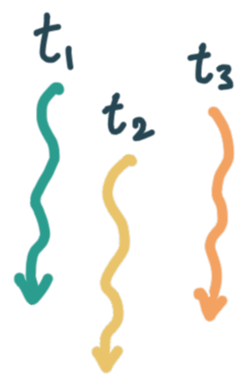


crash leaks f's blocks

Concurrency also creates subtle bugs

```
func REMOVE(d_ino: uint64,
            name: []byte) {

1  f := unlink(d_ino, name)

2  blocks := getBlocks(f)

3  free(blocks)

}
```

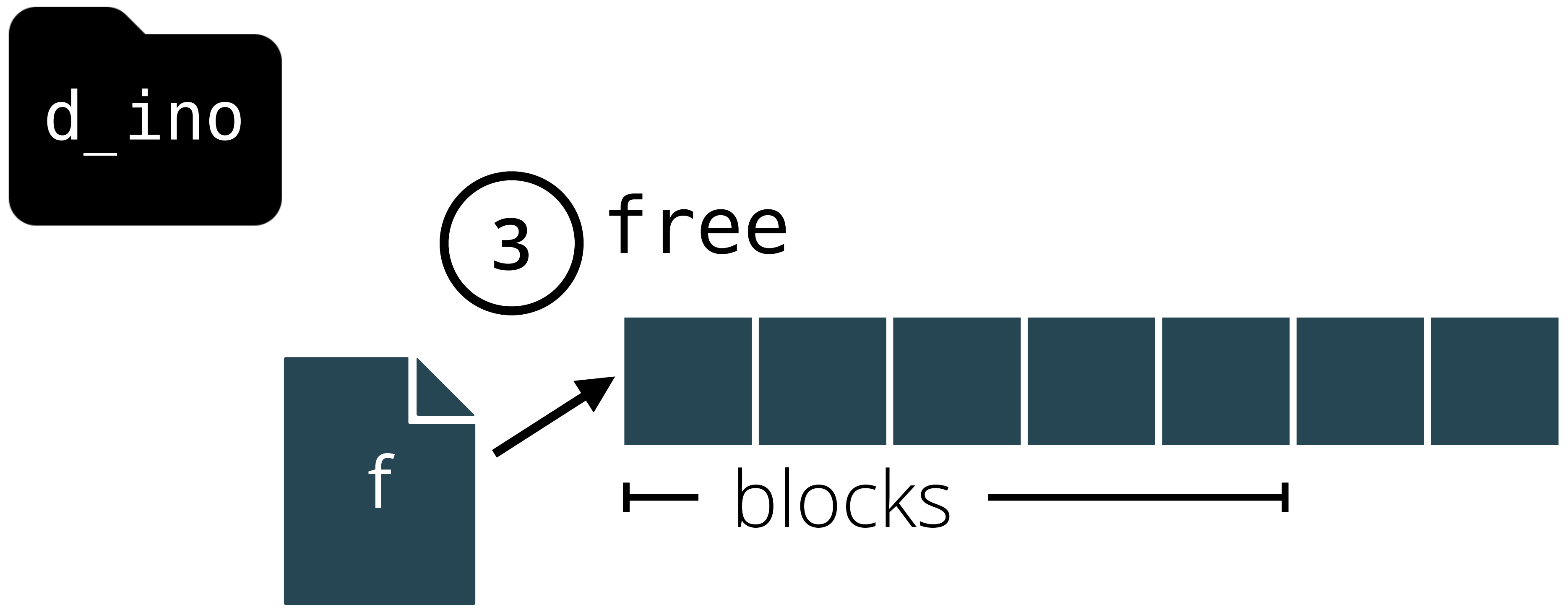




Concurrency also creates subtle bugs

```
func REMOVE(d_ino: uint64,
            name: []byte) {

① f := unlink(d_ino, name)

② blocks := getBlocks(f)
-------------------------------------
③ free(blocks)

}
```

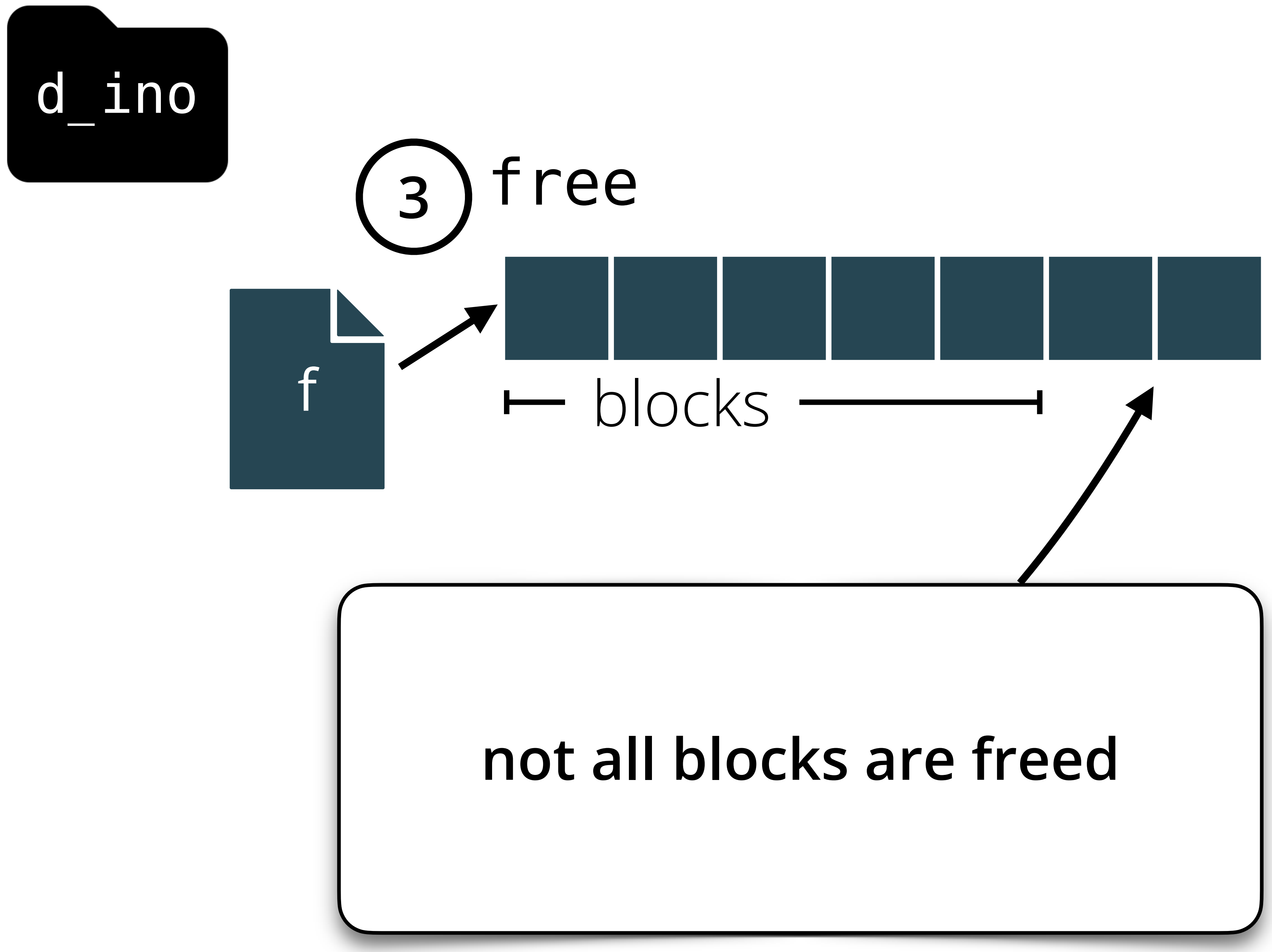
concurrent append



Concurrency also creates subtle bugs

```
func REMOVE(d_ino: uint64,
            name: []byte) {

①  f := unlink(d_ino, name)

②  blocks := getBlocks(f)
   ----------------------------
③  free(blocks)

}
```

concurrent append



Crashes and concurrency bugs can be severe

Might leak resources

Might return the wrong user's data

Might lose user data

Approach: verification-friendly system design

DaisyNFS

File-system code implemented with transactions

GoTxn

Transaction system gives atomicity

Approach: verification-friendly system design

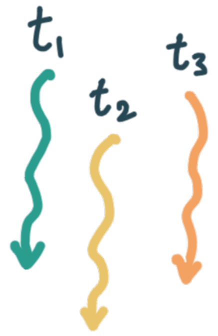
DaisyNFS

File-system code implemented with transactions

GoTxn

Transaction system gives atomicity

Crashes



Concurrency

Approach: verification-friendly system design

DaisyNFS

File-system code implemented with transactions

Sequential reasoning

GoTxn

Transaction system gives atomicity

Crashes

Concurrency

Approach: verification-friendly system design

DaisyNFS

File-system code implemented with transactions

Sequential reasoning

Specification for transactions bridges the two

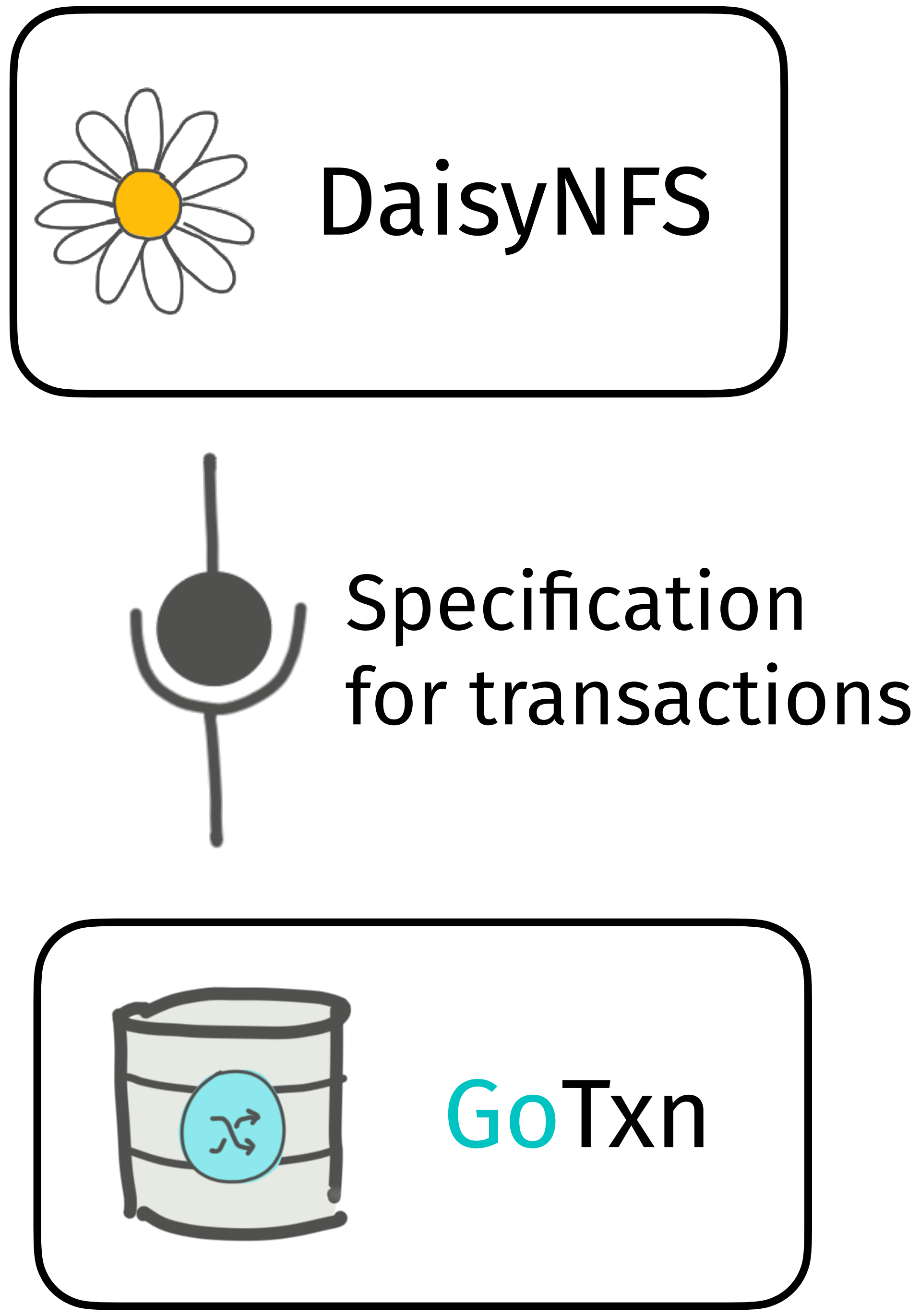
GoTxn

Transaction system gives atomicity

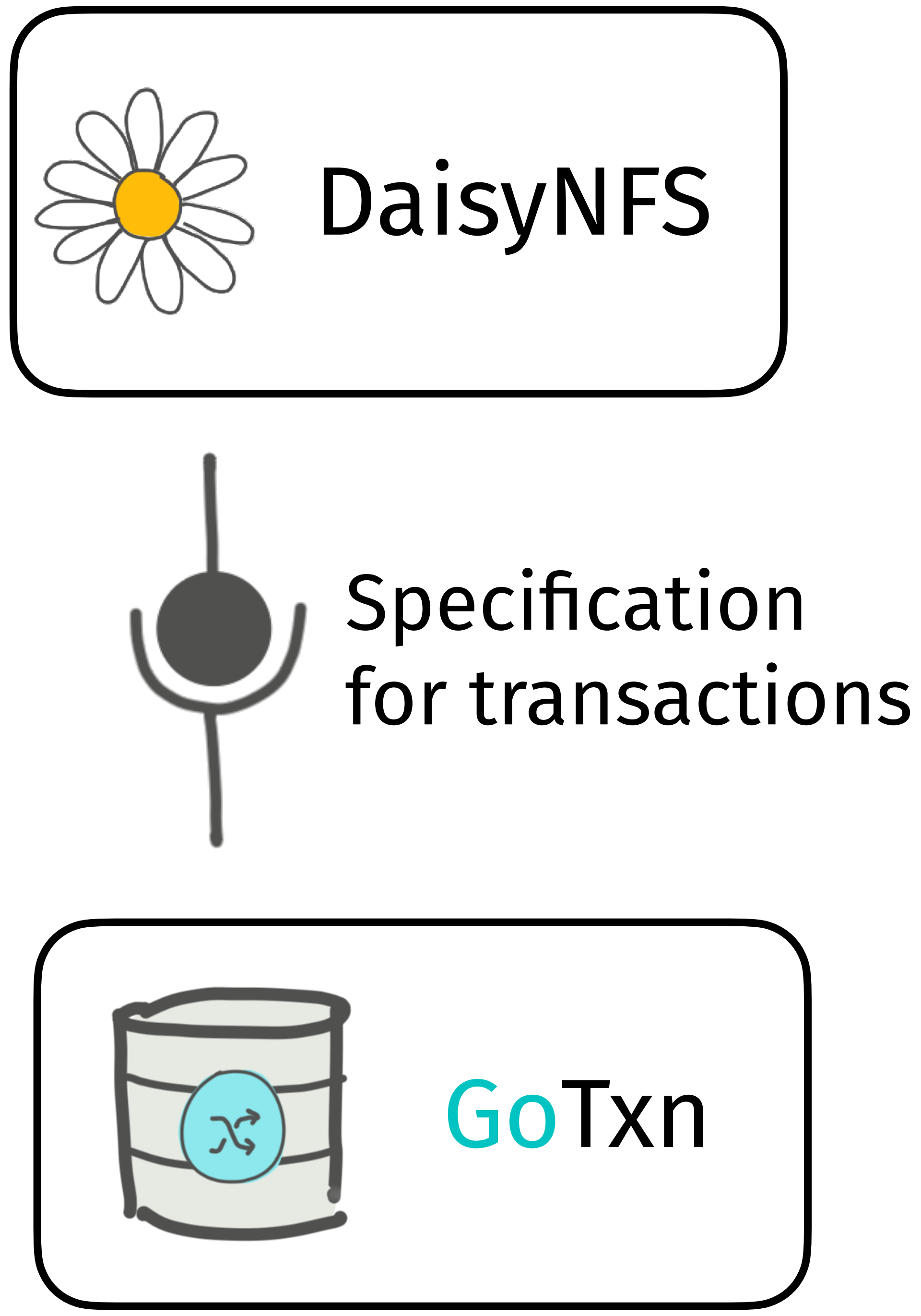
Crashes

Concurrency

Contributions

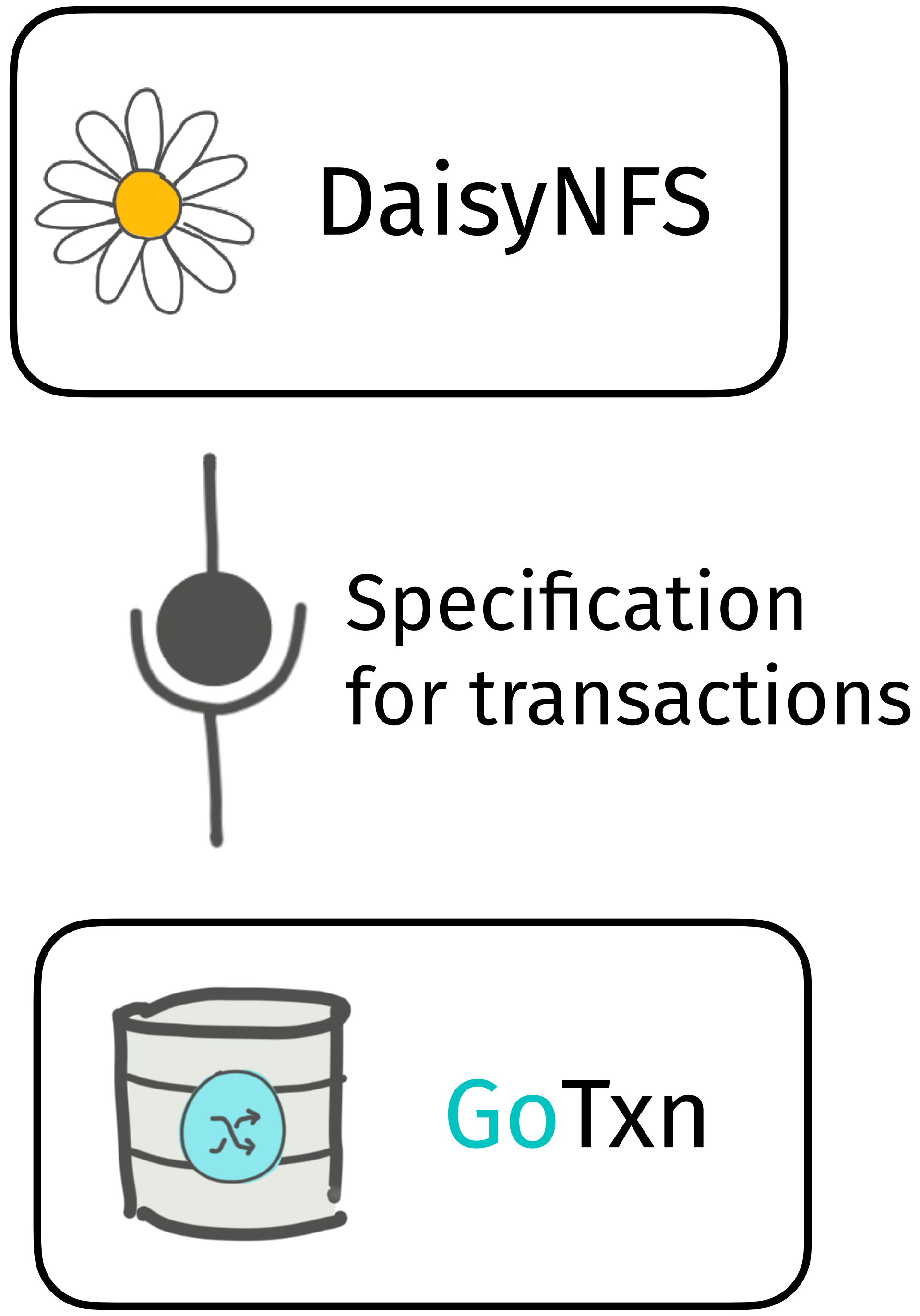


Contributions



Reduce proof effort with **sequential reasoning for a concurrent system**

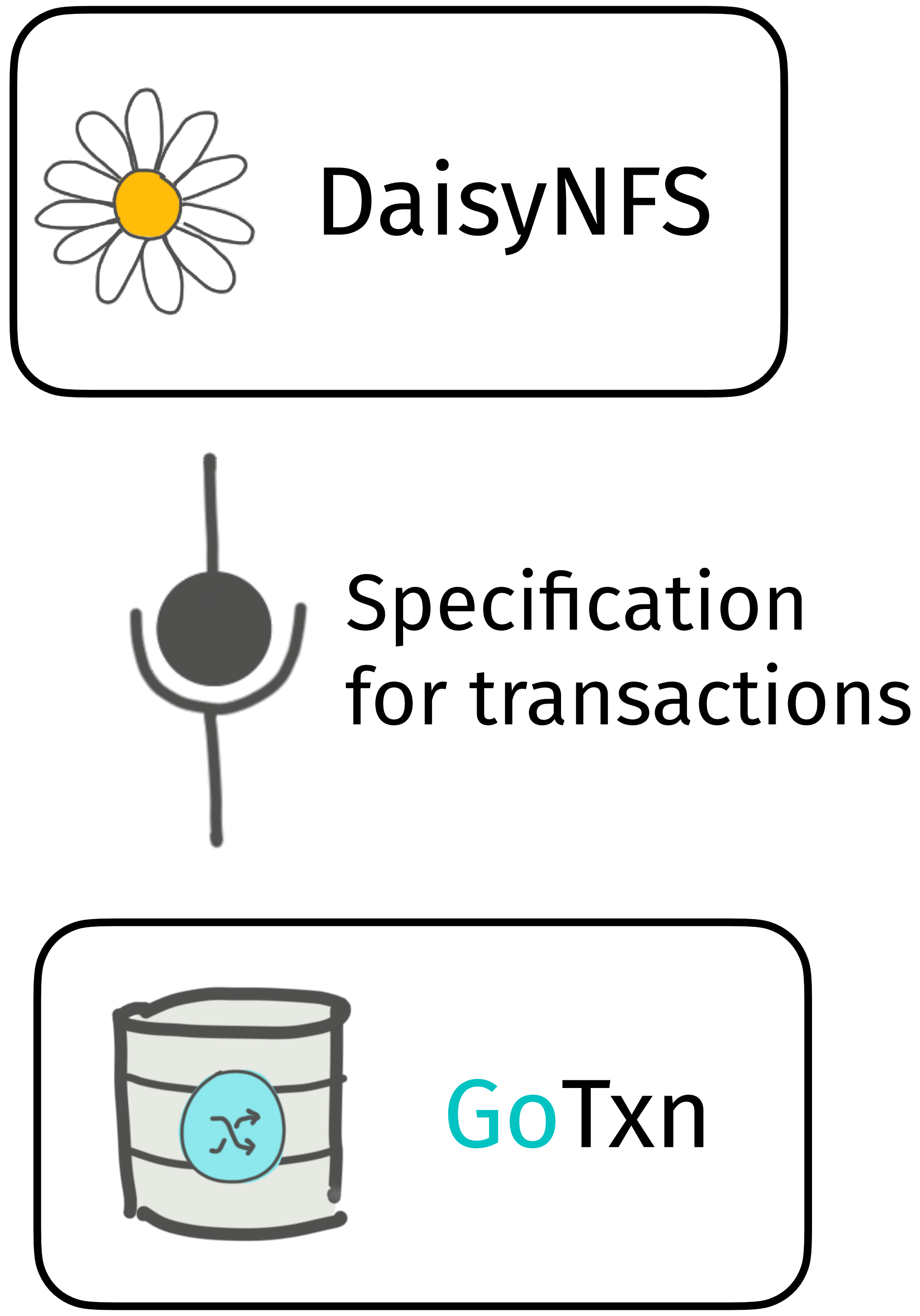
Contributions



Reduce proof effort with **sequential reasoning for a concurrent system**

Lifting specification for concurrent transactions

Contributions

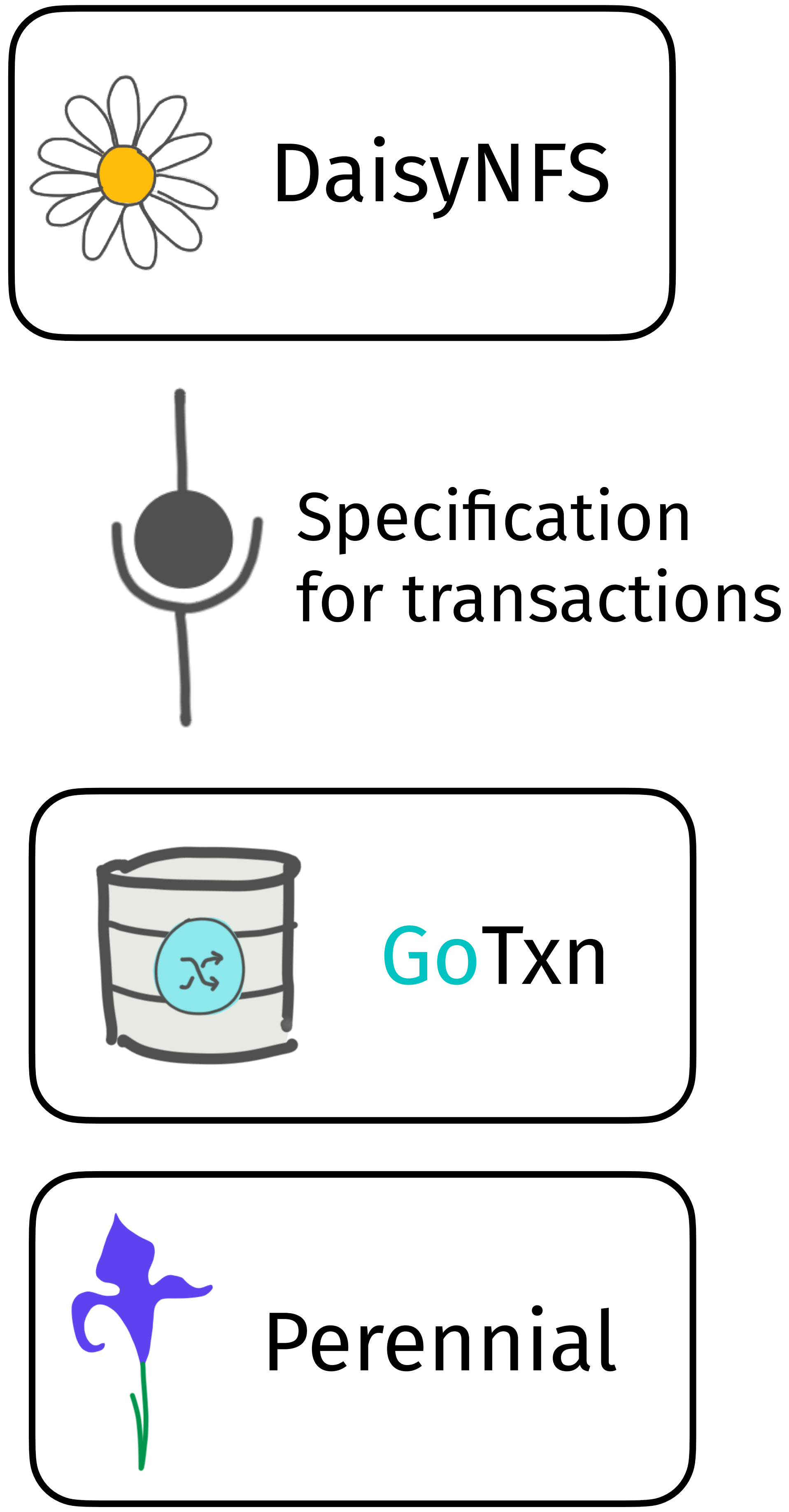


Reduce proof effort with **sequential reasoning for a concurrent system**

Lifting specification for concurrent transactions

Abstract state for write-ahead logging based on history of writes

Contributions



Reduce proof effort with **sequential reasoning for a concurrent system**

Lifting specification for concurrent transactions

Abstract state for write-ahead logging based on history of writes

Perennial logic for concurrency and crash reasoning

Specification
for transactions

Transactions automatically give atomicity

```
func               Begin() *Txn

func (tx *Txn) Read(…)
func (tx *Txn) Write(…)

func (tx *Txn) Commit()
```

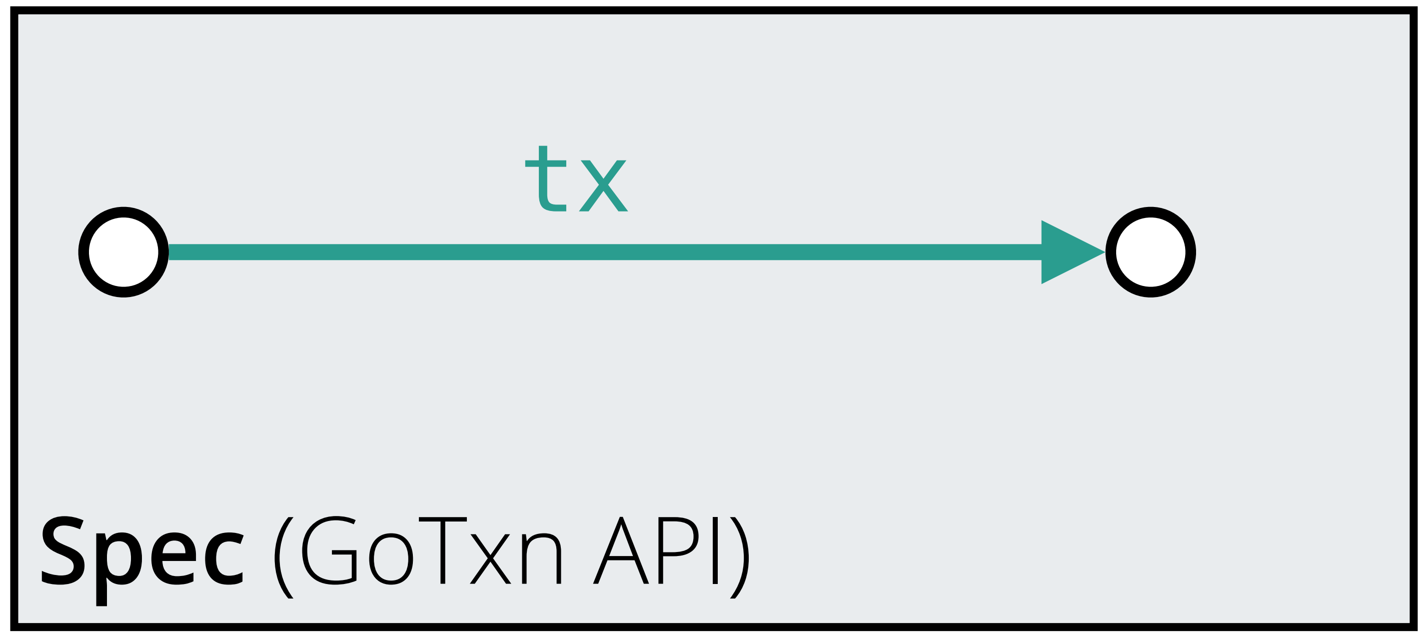
GoTxn

Code between `Begin()` and `Commit()` is **atomic both on crash and to other threads**

Specifying a transaction system

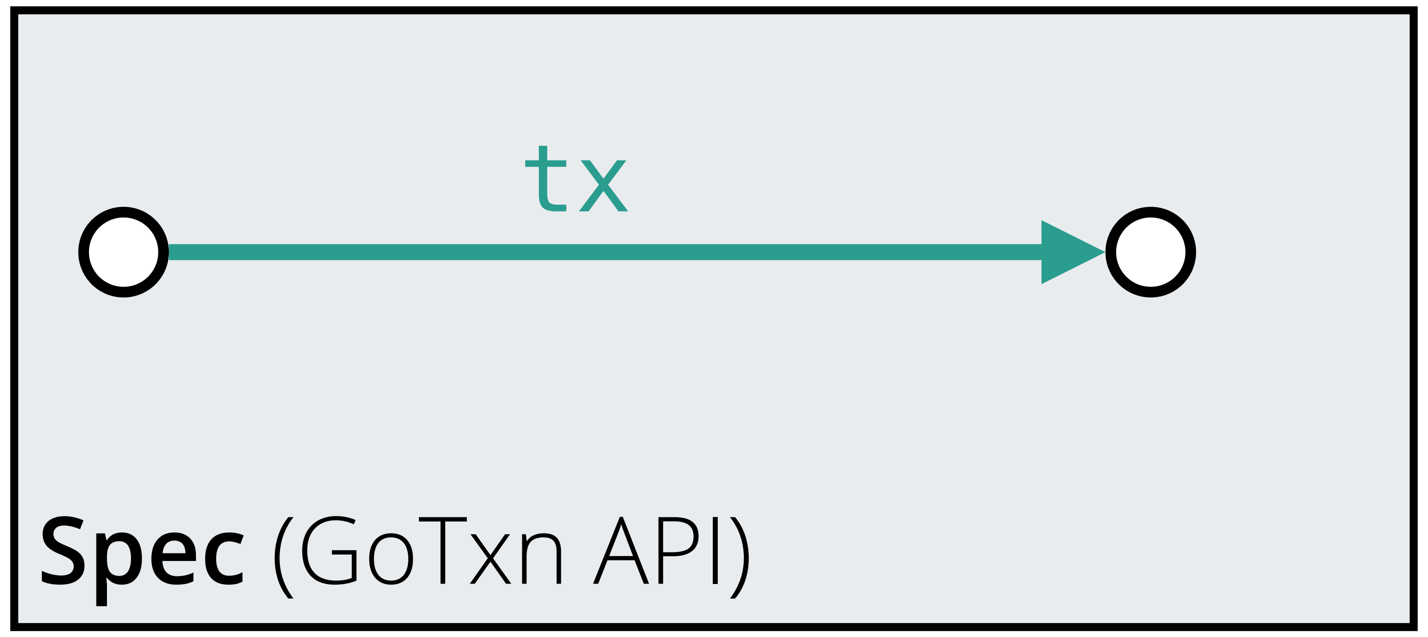
tx

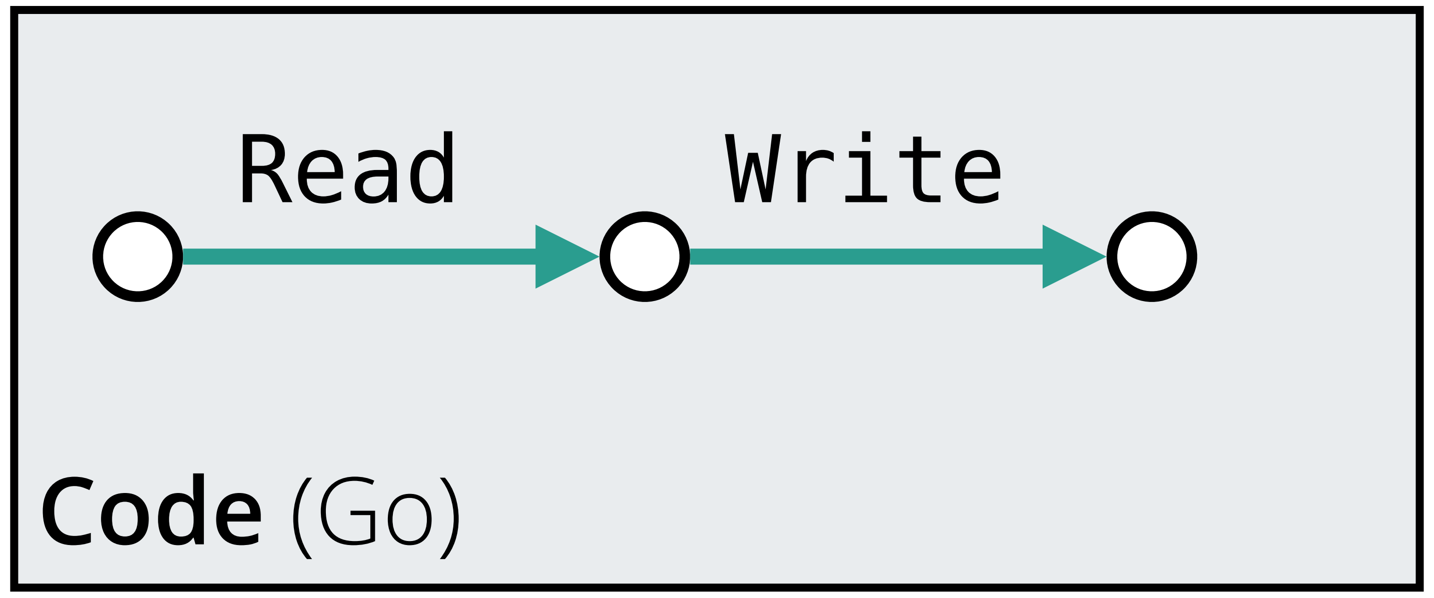
```
v := tx.Read(0)
tx.Write(1, v)
```



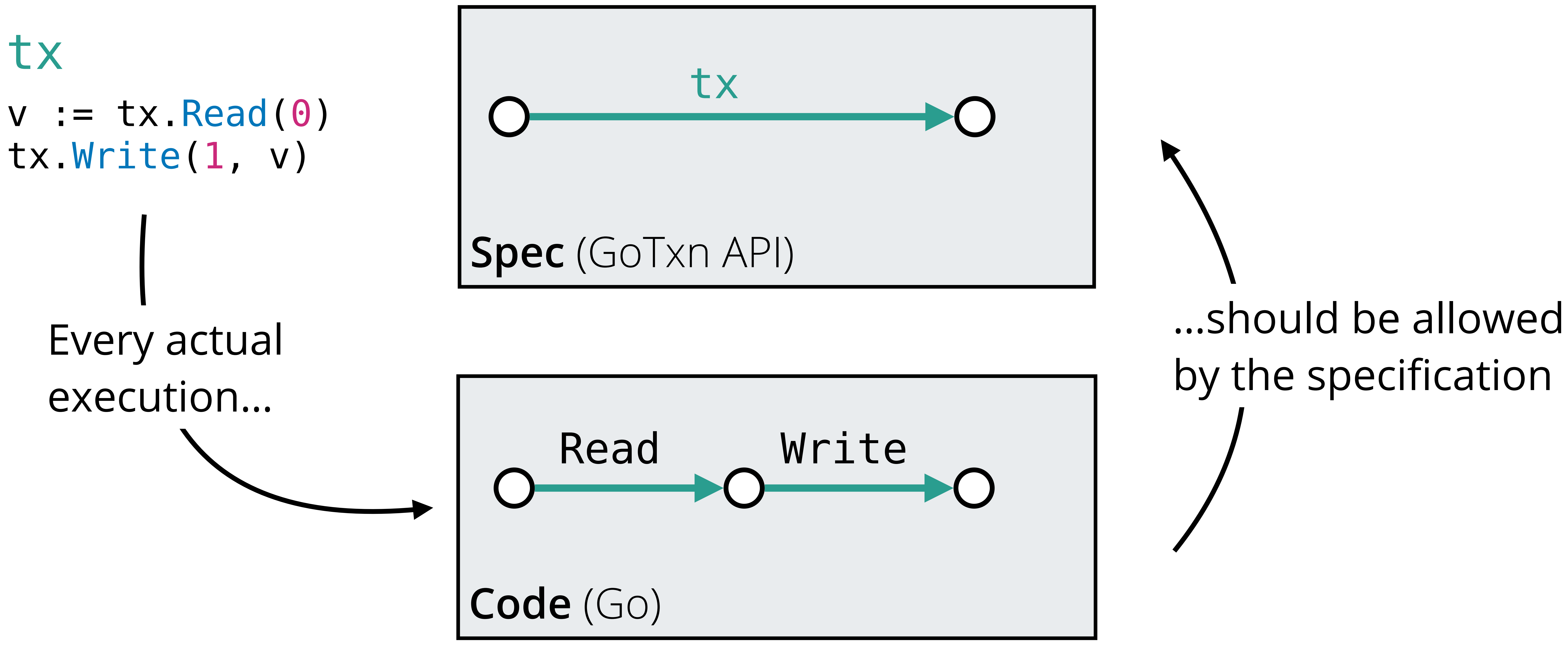
Specifying a transaction system

```
tx
v := tx.Read(0)
tx.Write(1, v)
```





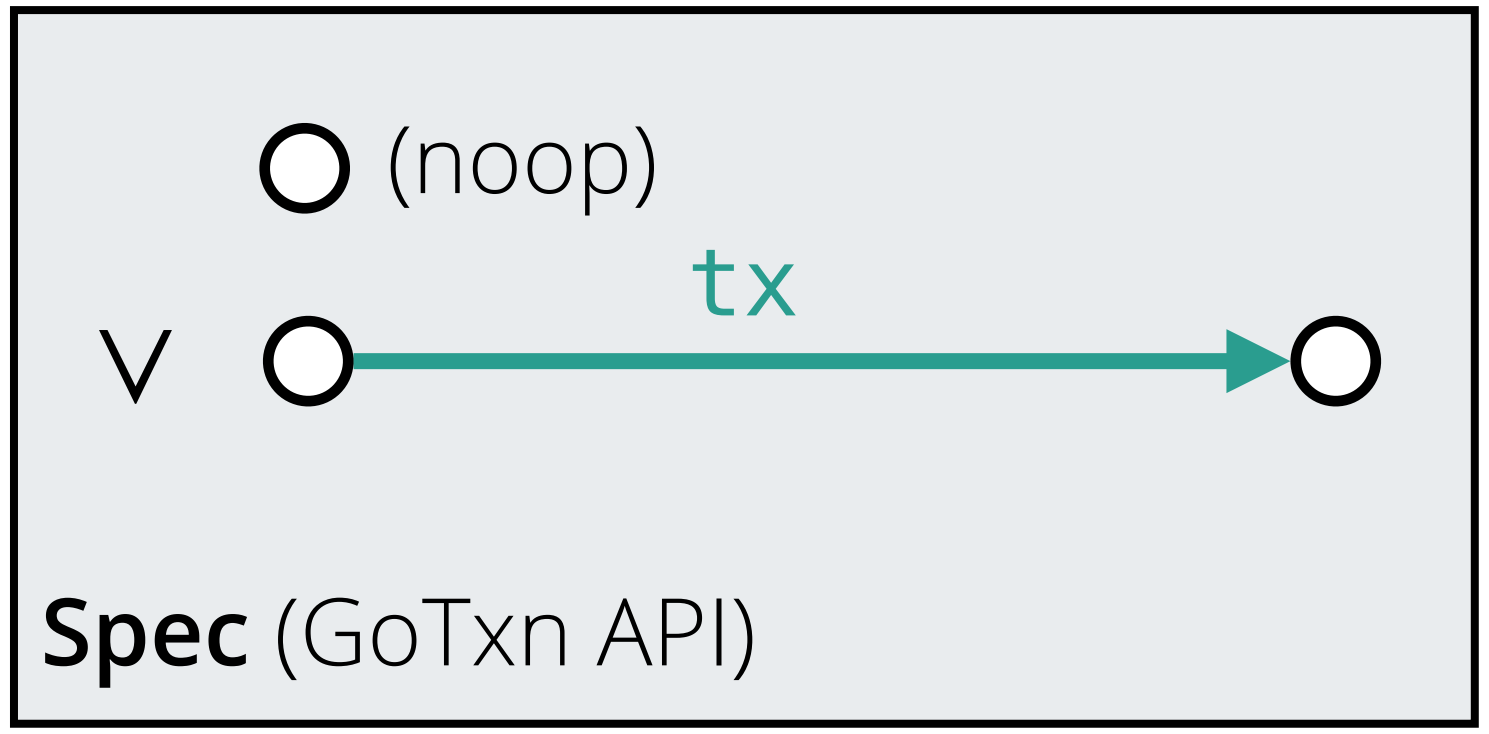
Specifying a transaction system



Specifying crash atomicity for transactions

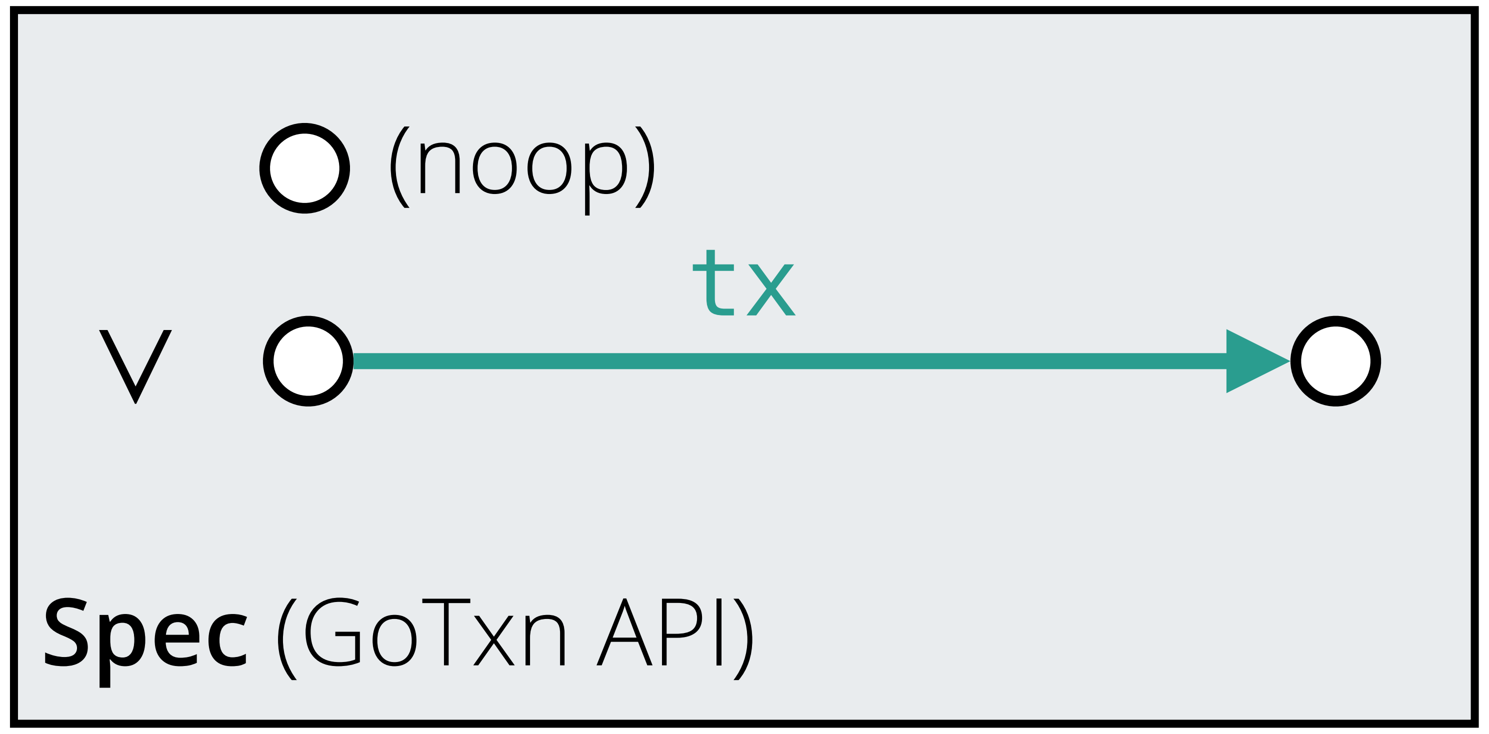
tx

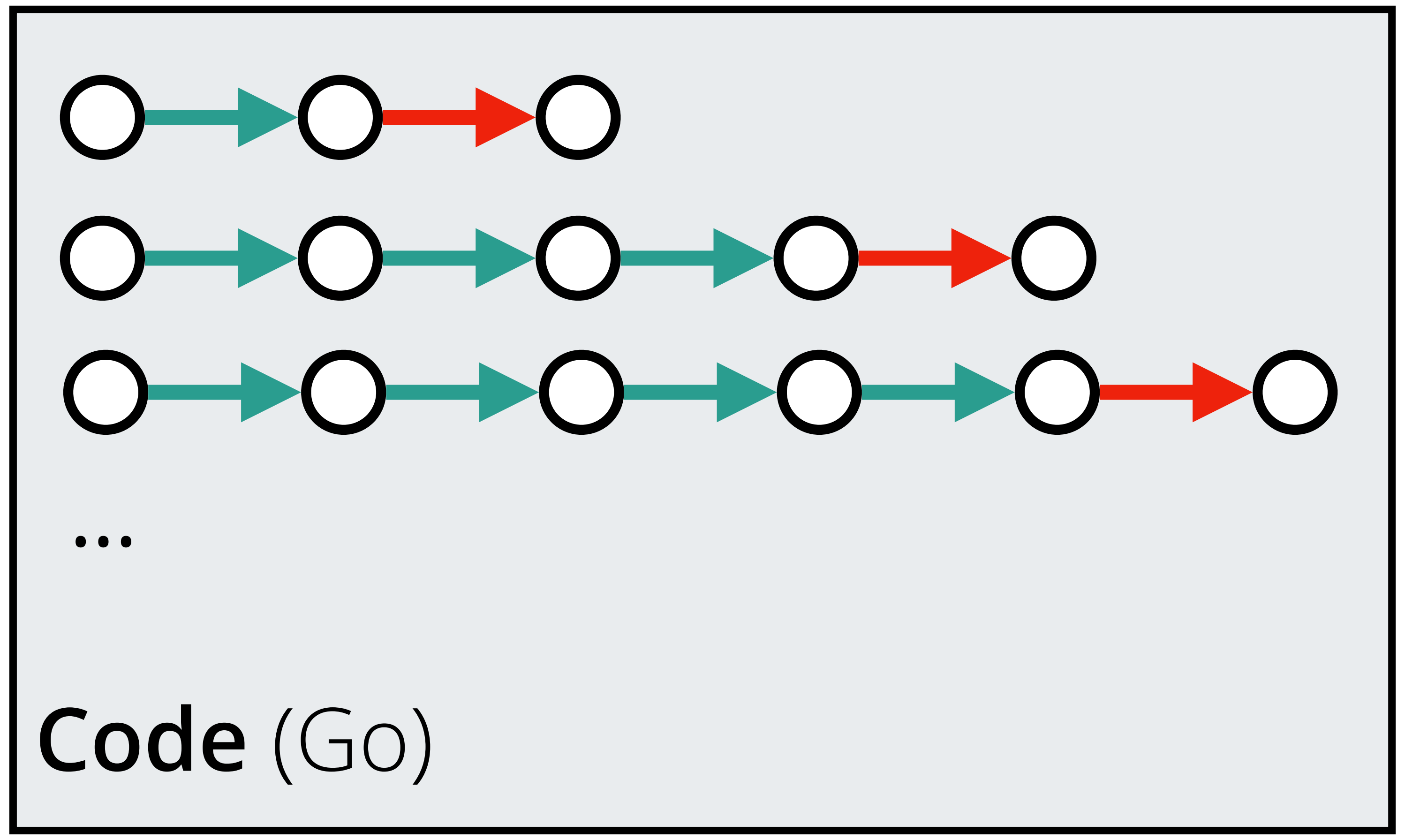
```
v := tx.Read(0)
tx.Write(1, v)
```



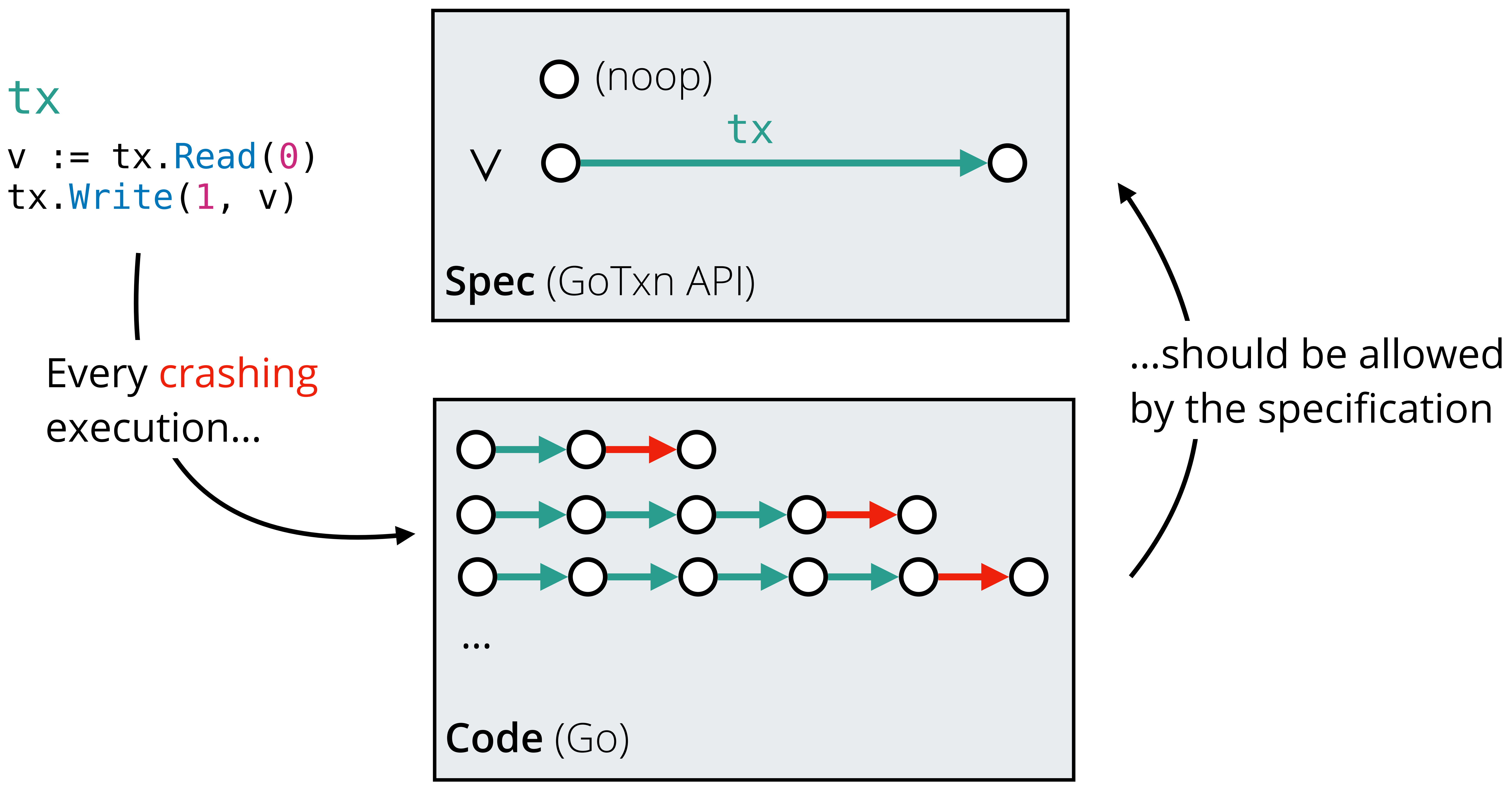
Specifying crash atomicity for transactions

```
tx
v := tx.Read(0)
tx.Write(1, v)
```





Specifying crash atomicity for transactions



Specifying sequential transactional API

```
tx := Begin()
v := tx.Read(0)
tx.Write(1, v)
tx.Write(2, v)
tx.Commit()
```

Specifying sequential transactional API


```
tx := Begin()
v := tx.Read(0)
tx.Write(1, v)
tx.Write(2, v)
tx.Commit()
```

Specifying sequential transactional API


```
tx := Begin()
v := tx.Read(0)
tx.Write(1, v)
tx.Write(2, v)
tx.Commit()
```

what is the state here?

Specifying sequential transactional API


```
tx := Begin()
v := tx.Read(0)
tx.Write(1, v)
tx.Write(2, v)
tx.Commit()
```

what is the state here?

how to capture that crash results in initial or final state?

Specifying sequential transactional API


```
tx := Begin()
v := tx.Read(0)
tx.Write(1, v)
```

```
tx.Write(2, v)
tx.Commit()
```

Specifying sequential transactional API


```
tx := Begin()
v := tx.Read(0)
tx.Write(1, v)
```

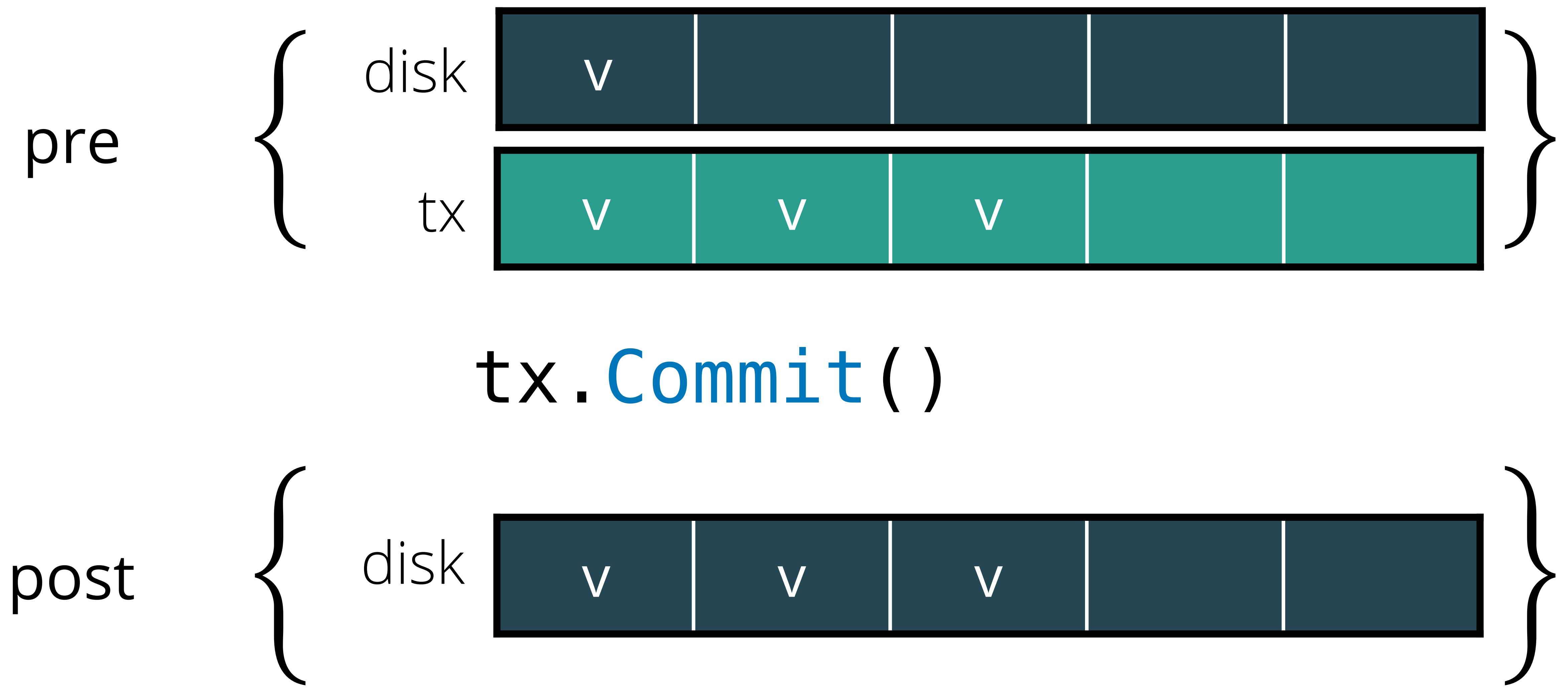

```
tx.Write(2, v)
tx.Commit()
```

Specifying sequential transactional API


```
tx := Begin()
v := tx.Read(0)
tx.Write(1, v)
tx.Write(2, v)
```


```
tx.Commit()
```

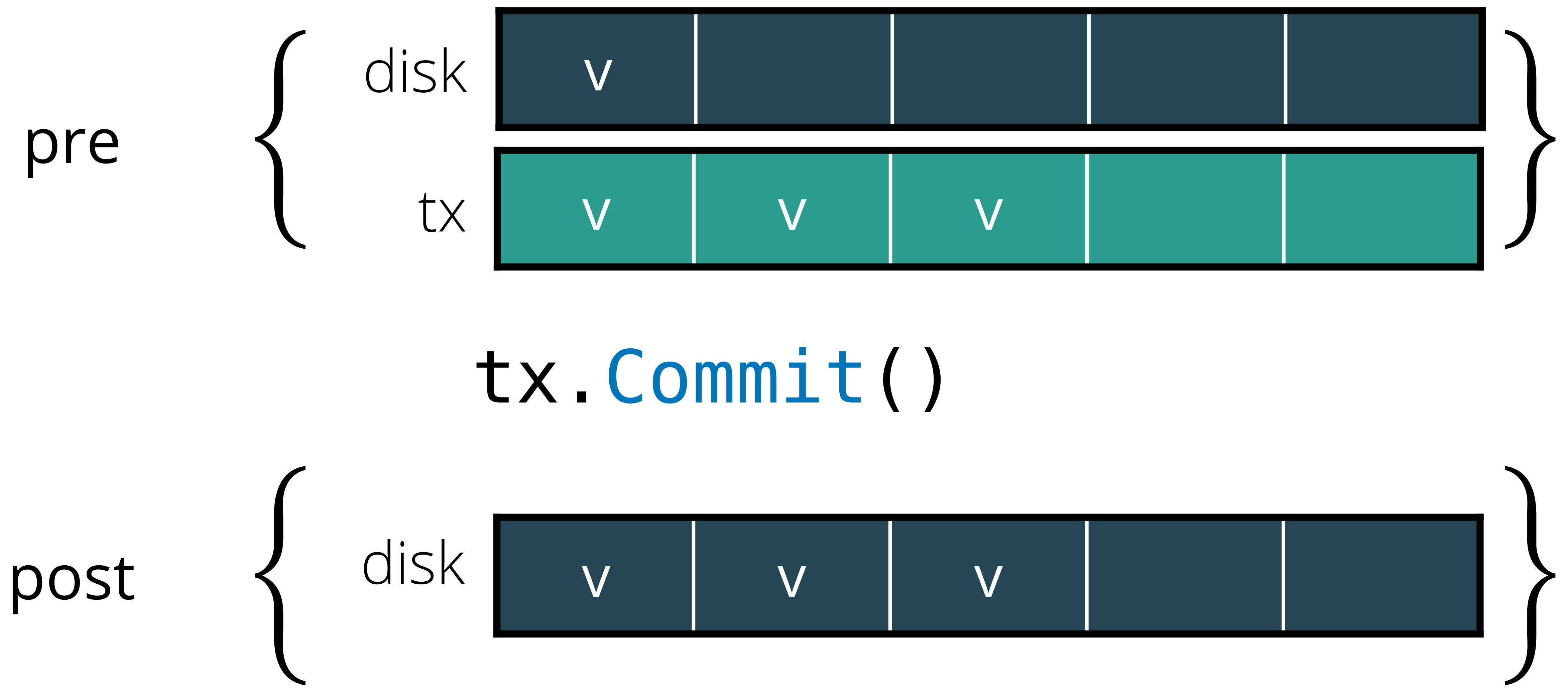
Separation logic to specify Commit without crashes



Perennial logic adds crash conditions

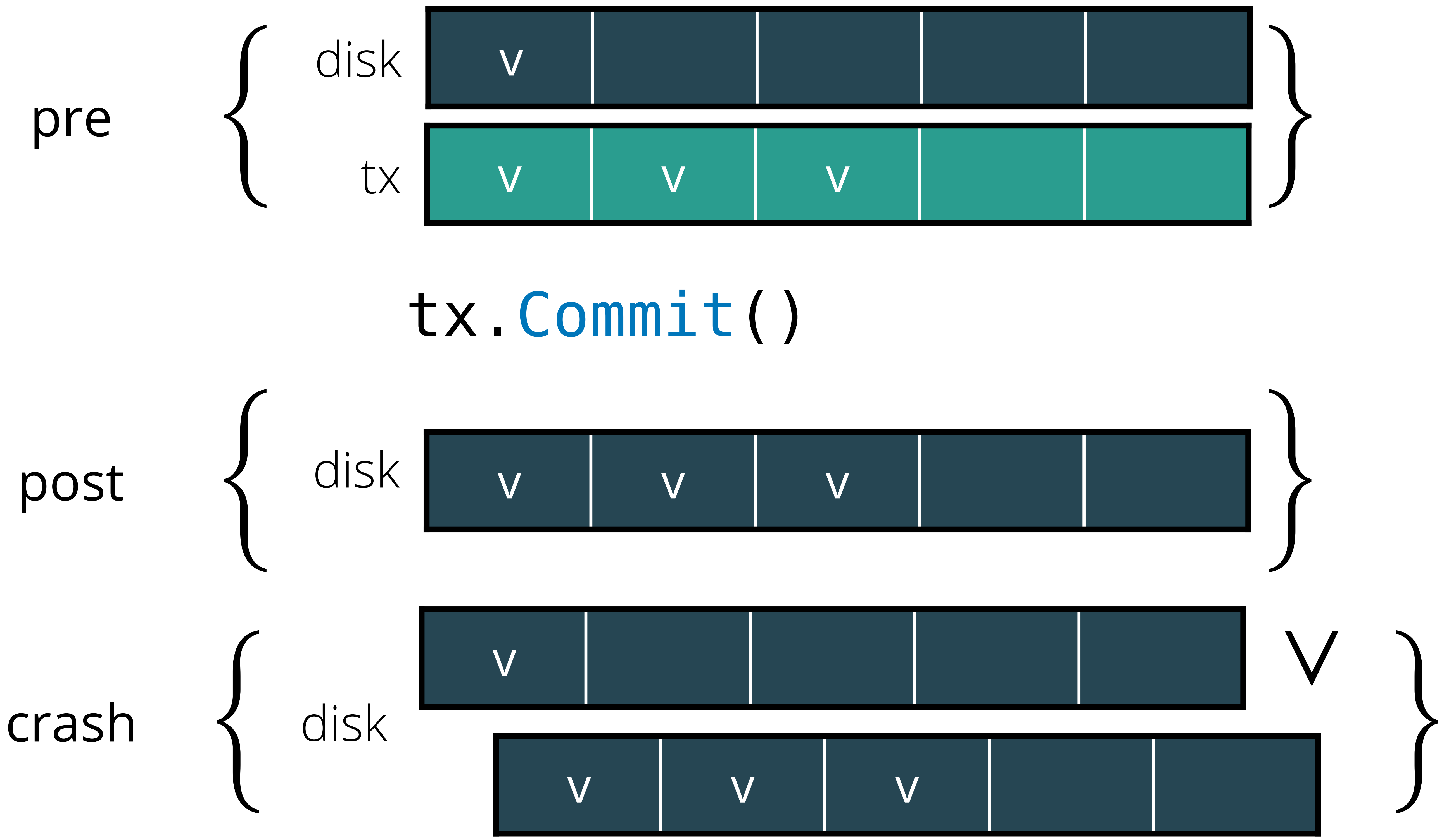
[CZ**C**CKZ, SOSP '15]

[**C**TTJKZ, OSDI '21]



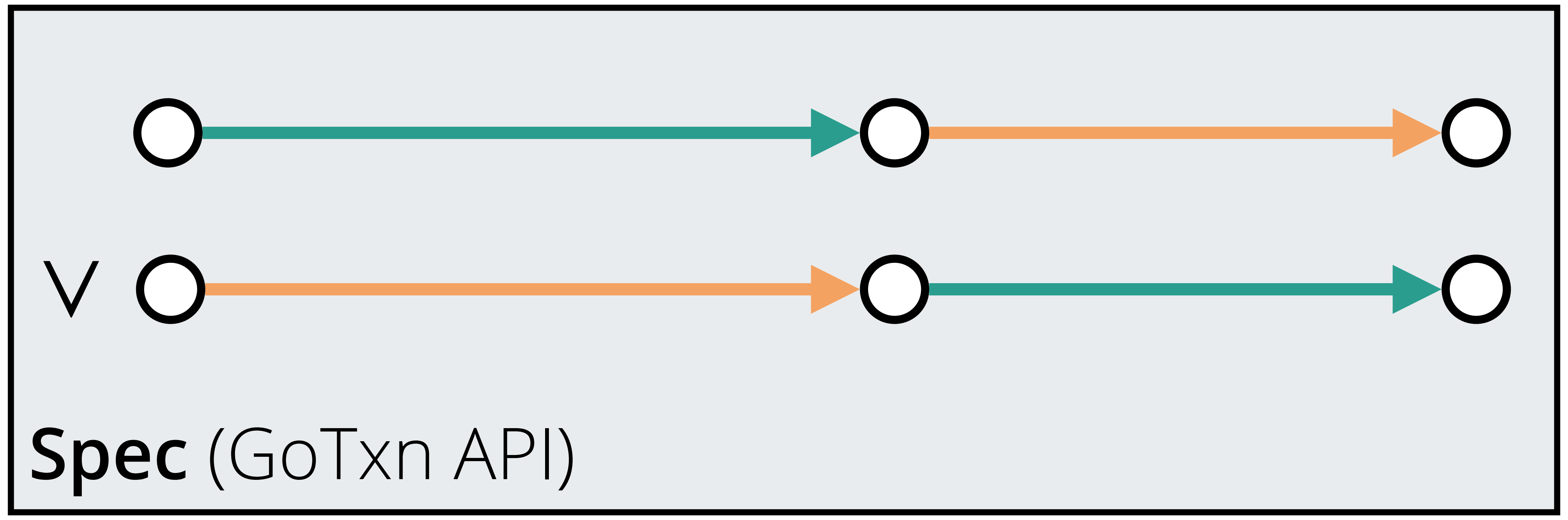
Perennial logic adds crash conditions

[CZ**C**CKZ, SOSP ’15]
[**C**TTJKZ, OSDI ’21]

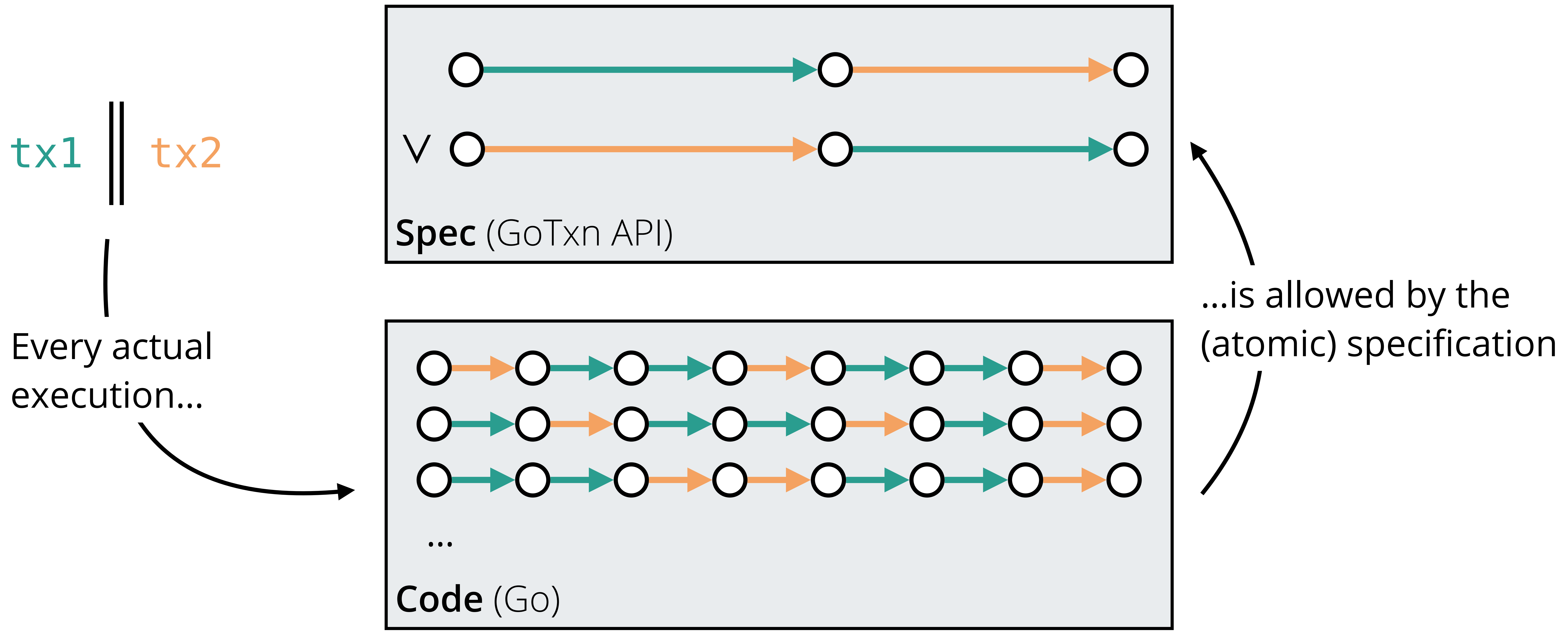


Generalizing to include concurrency

`tx1` ∥ `tx2`



Generalizing to include concurrency



Challenge: specifying concurrent transactions

```
tx1 := Begin()
v := tx1.Read(0)
tx1.Write(1, v)
tx1.Write(2, v)
tx1.Commit()
```

‖

```
tx2 := Begin()
tx2.Write(4, data)
tx2.Commit()
```

Challenge: specifying concurrent transactions

disk


```
tx1 := Begin()
v := tx1.Read(0)
tx1.Write(1, v)
          tx2 := Begin()
          tx2.Write(4, data)
          tx2.Commit()
```


```
tx1.Write(2, v)
tx1.Commit()
```

Challenge: specifying concurrent transactions

disk


```
tx1 := Begin()
v := tx1.Read(0)
tx1.Write(1, v)
```

```
tx2 := Begin()
tx2.Write(4, data)
tx2.Commit()
```


```
tx1.Write(2, v)
tx1.Commit()
```

How to reason about transactions separately?

Idea: lifting-based specification

[**C**TTJKZ, OSDI '21]


```
tx1 := Begin()
v := tx1.Read(0)
tx1.Write(1, v)
tx1.Write(2, v)
```

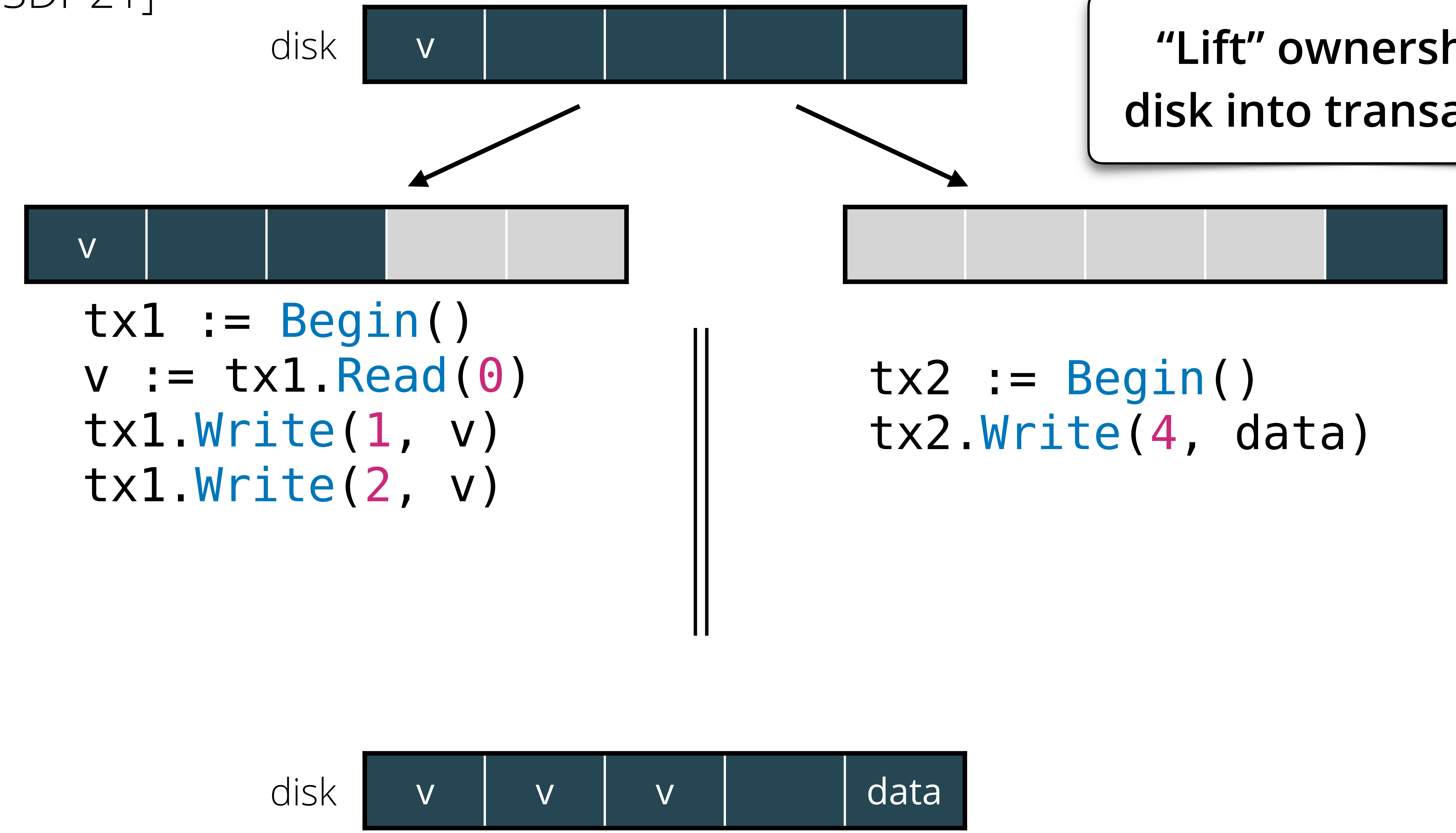
||

```
tx2 := Begin()
tx2.Write(4, data)
```

Idea: lifting-based specification

[CTTJKZ, OSDI '21]

"Lift" ownership of disk into transactions



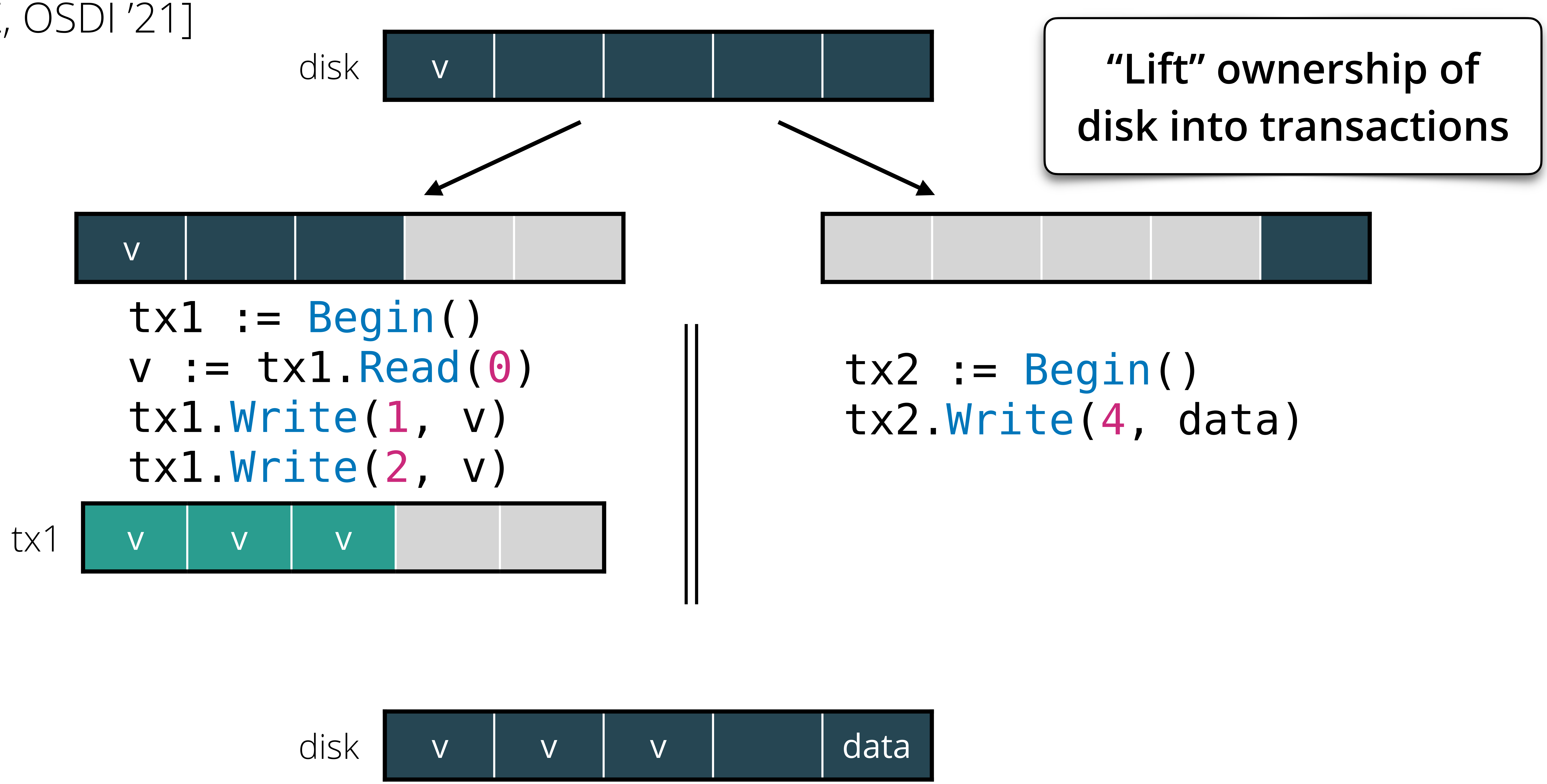

```
tx1 := Begin()
v := tx1.Read(0)
tx1.Write(1, v)
tx1.Write(2, v)
```

||

```
tx2 := Begin()
tx2.Write(4, data)
```

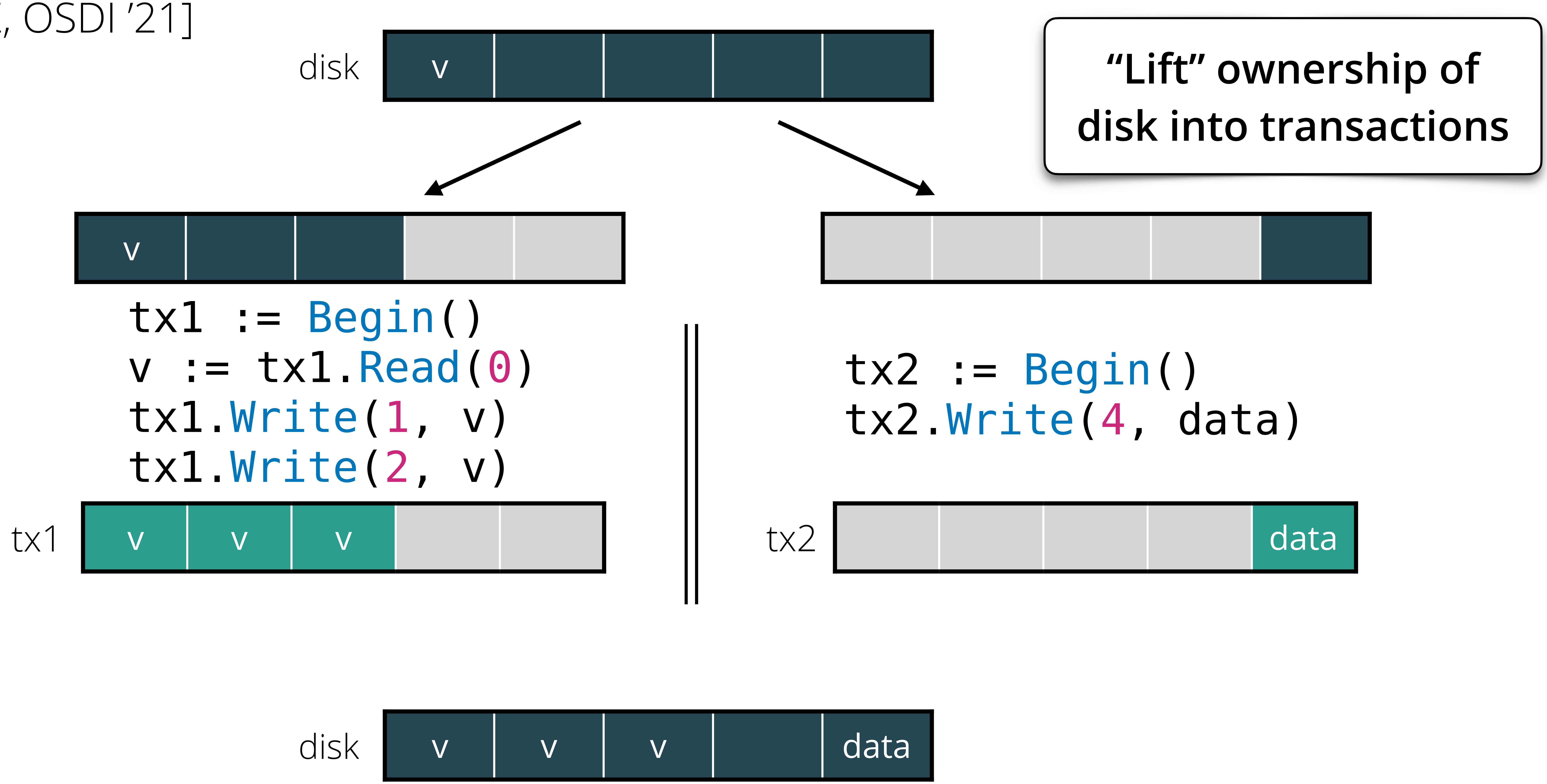
Idea: lifting-based specification

[CTTJKZ, OSDI ’21]



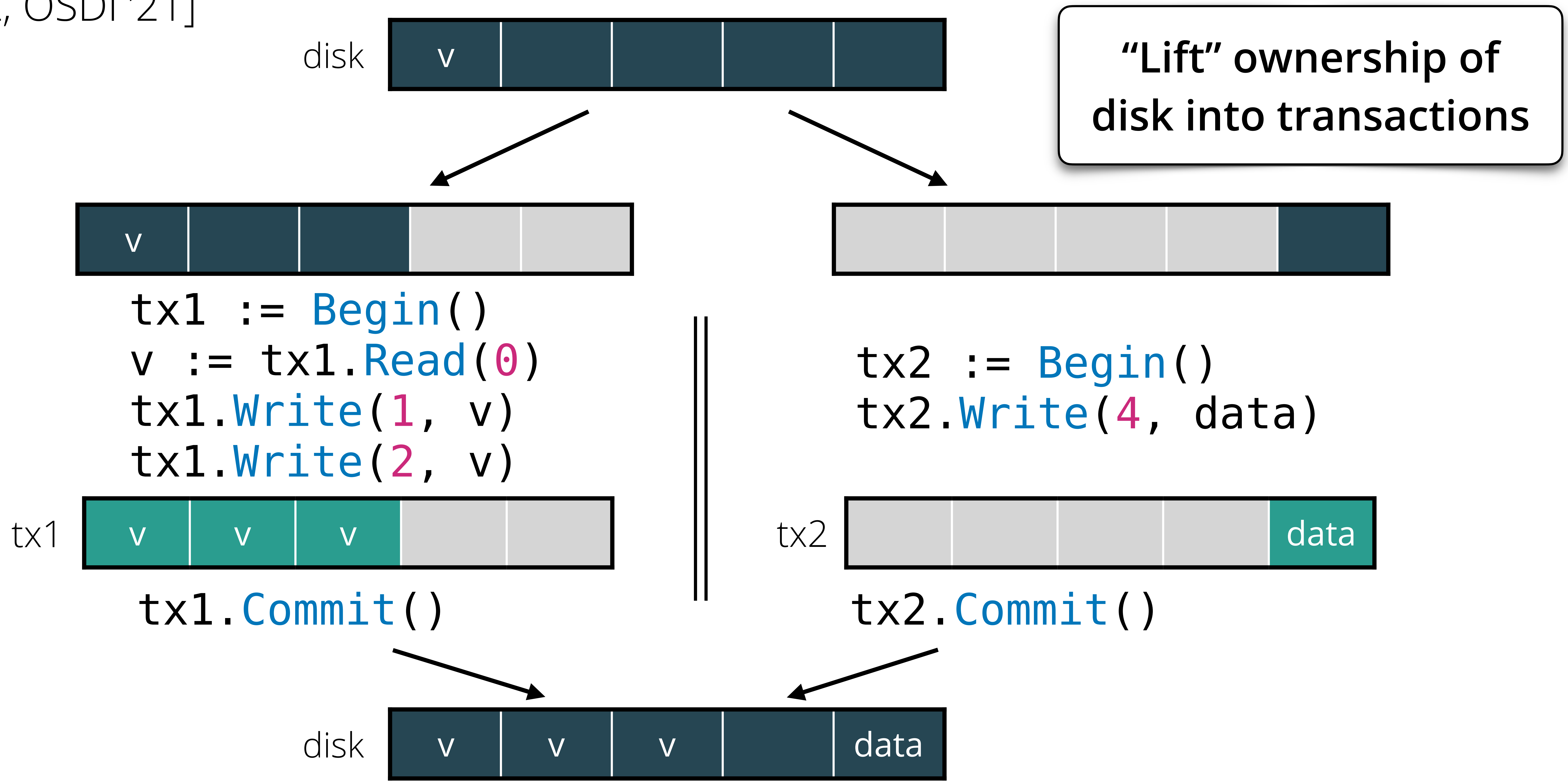
Idea: lifting-based specification

[CTTJKZ, OSDI ’21]

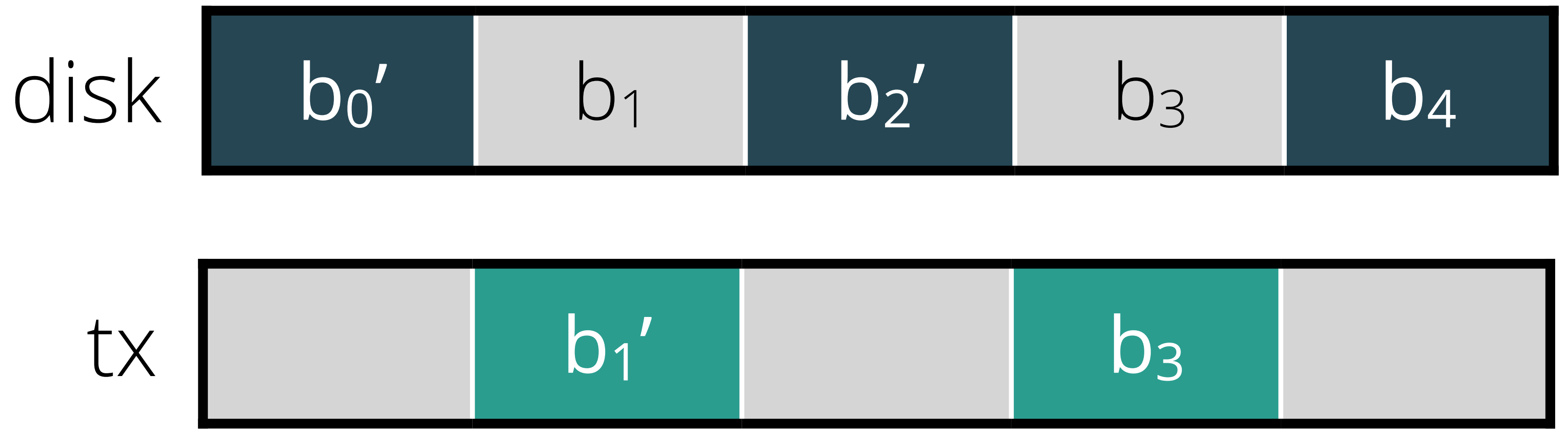


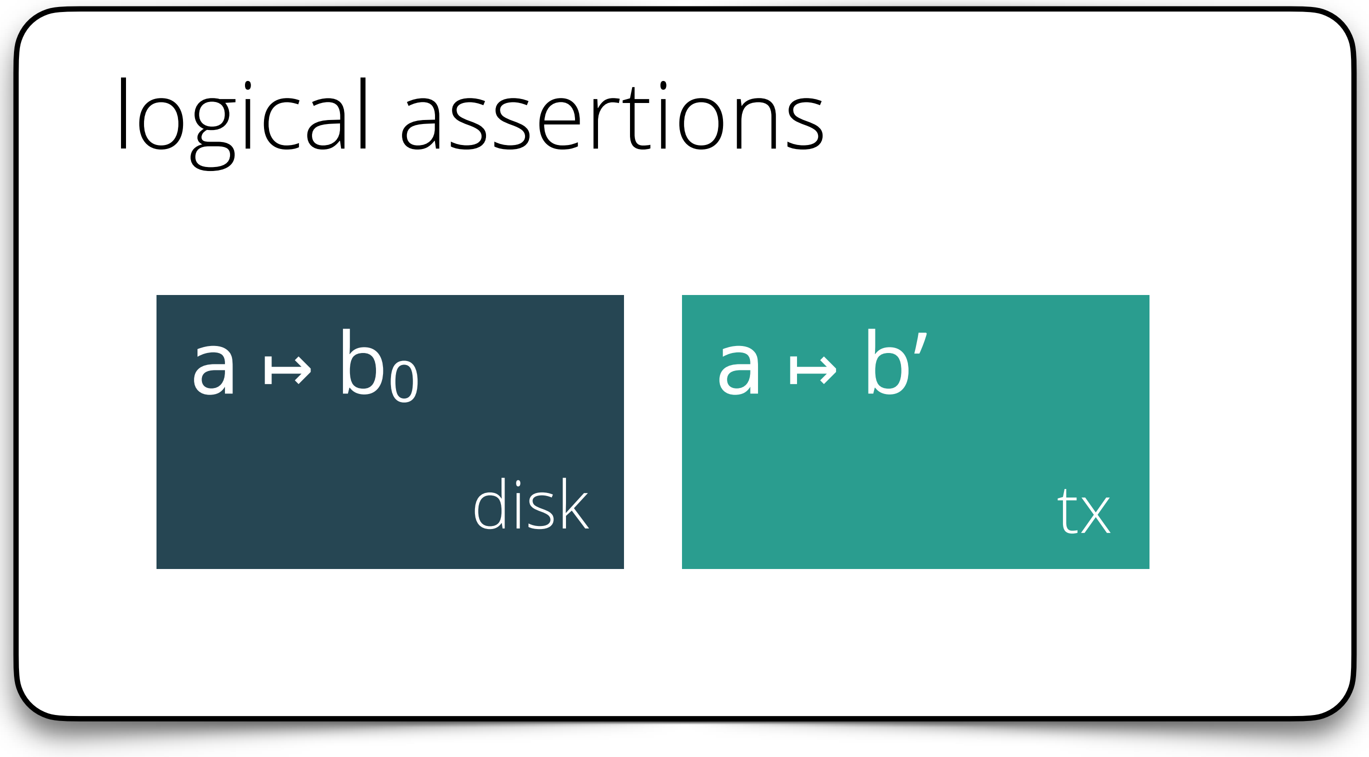
Idea: lifting-based specification

[**C**TTJKZ, OSDI '21]



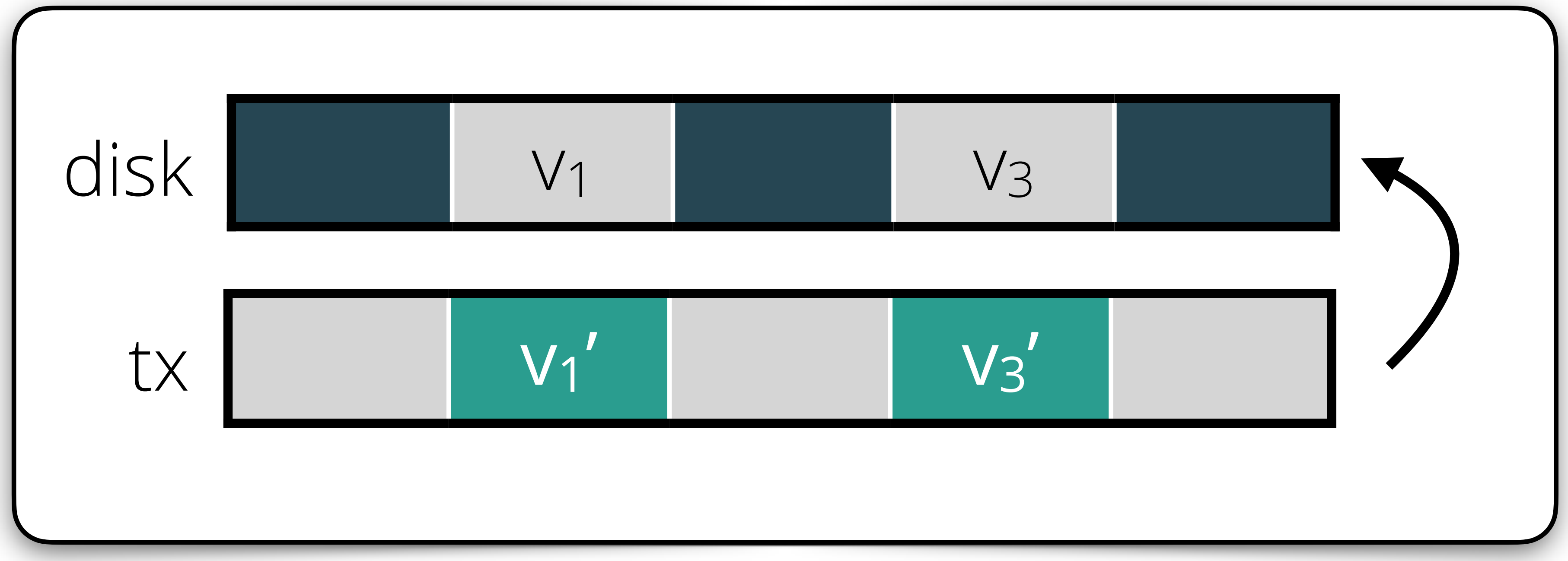
Separation logic describes lifting





Commit spec captures atomicity

[**C**TTJKZ, OSDI ’21]

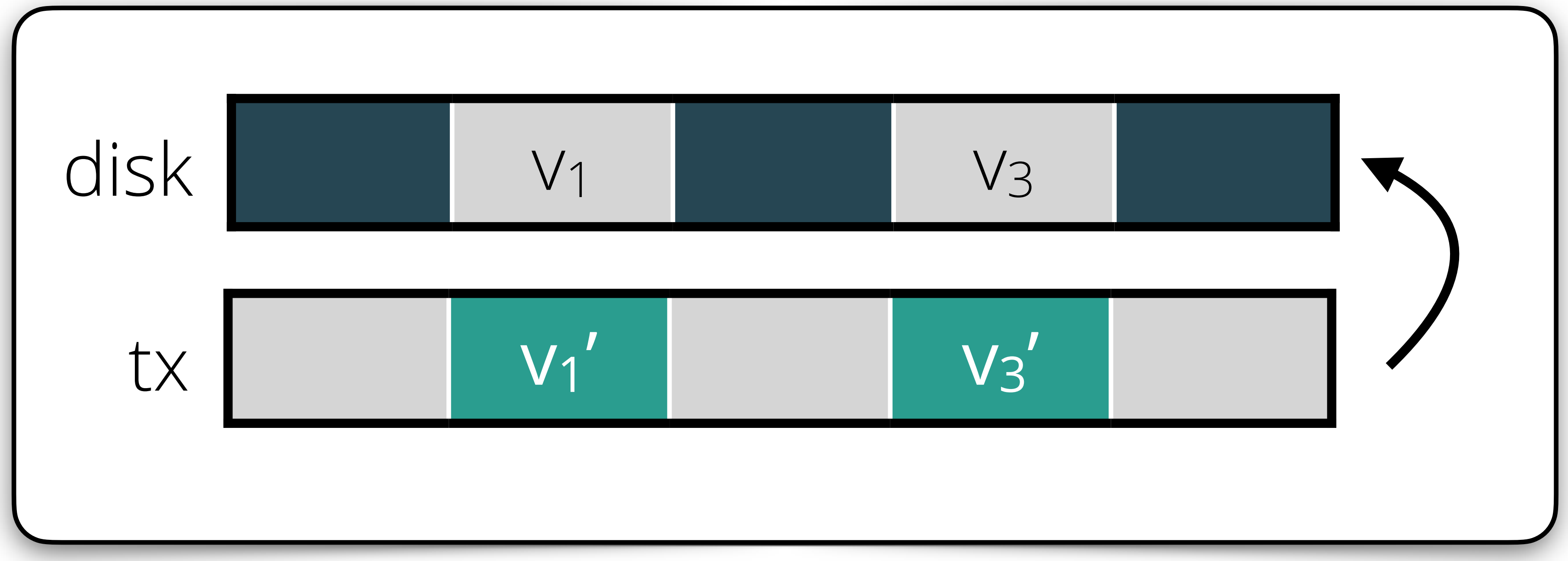


Commit spec captures atomicity

[CTTJKZ, OSDI '21]

$\overline{a} \mapsto \overline{v}$ disk

$\overline{a} \mapsto \overline{v}'$ tx



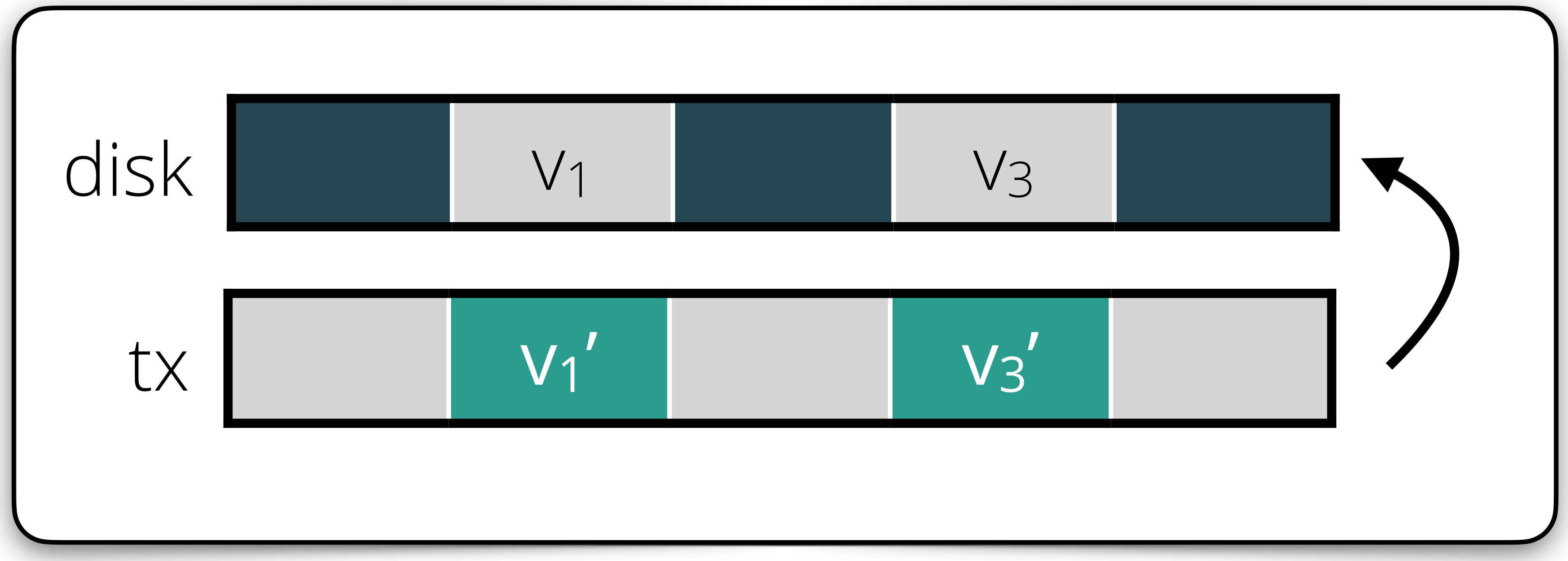
Commit spec captures atomicity

[CTTJKZ, OSDI '21]

$\{\, \overline{a} \mapsto \overline{v}$ (disk) $\;\; \overline{a} \mapsto \overline{v}'$ (tx) $\,\}$

```
tx.Commit()
```

$\{\, \overline{a} \mapsto \overline{v}'$ (disk) $\,\}$



Commit spec captures atomicity

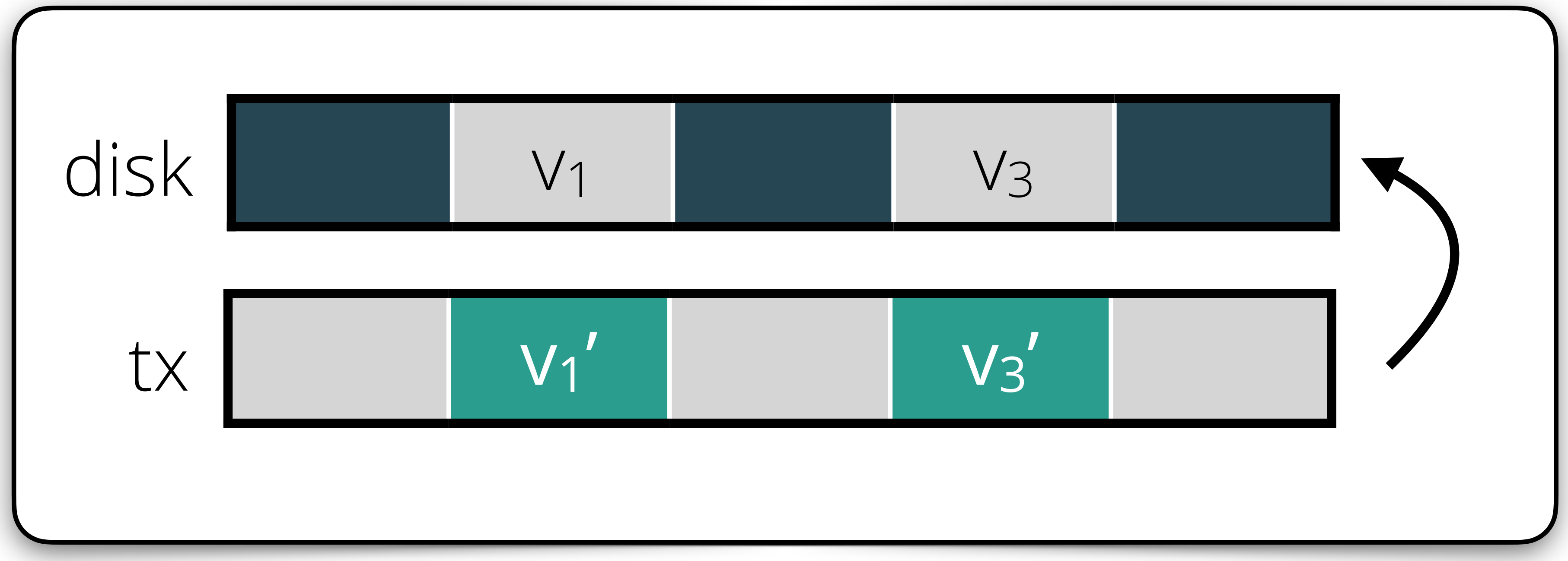
[CTTJKZ, OSDI '21]

$\left\{ \overline{a} \mapsto \overline{v} \;\; \text{disk} \quad \overline{a} \mapsto \overline{v}' \;\; \text{tx} \right\}$

`tx.Commit()`

$\left\{ \overline{a} \mapsto \overline{v}' \;\; \text{disk} \right\}$

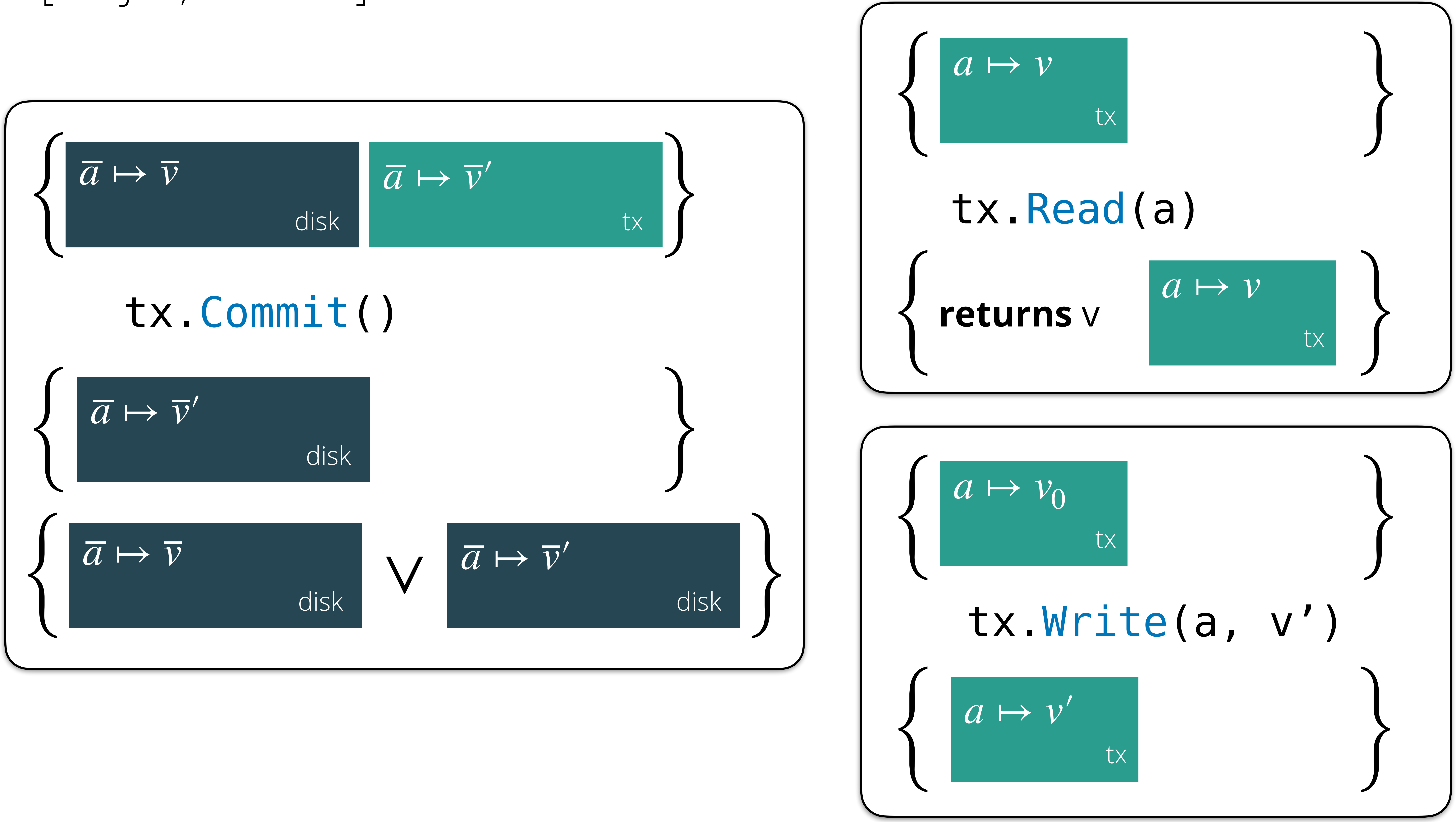
$\left\{ \overline{a} \mapsto \overline{v} \;\; \text{disk} \;\vee\; \overline{a} \mapsto \overline{v}' \;\; \text{disk} \right\}$



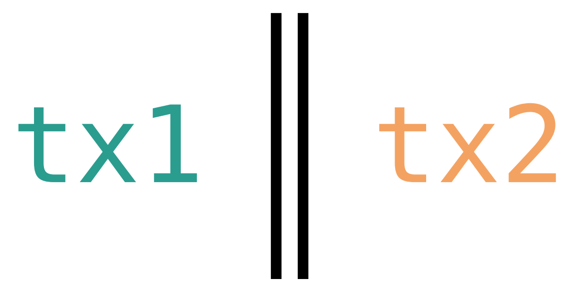
crash condition is atomic

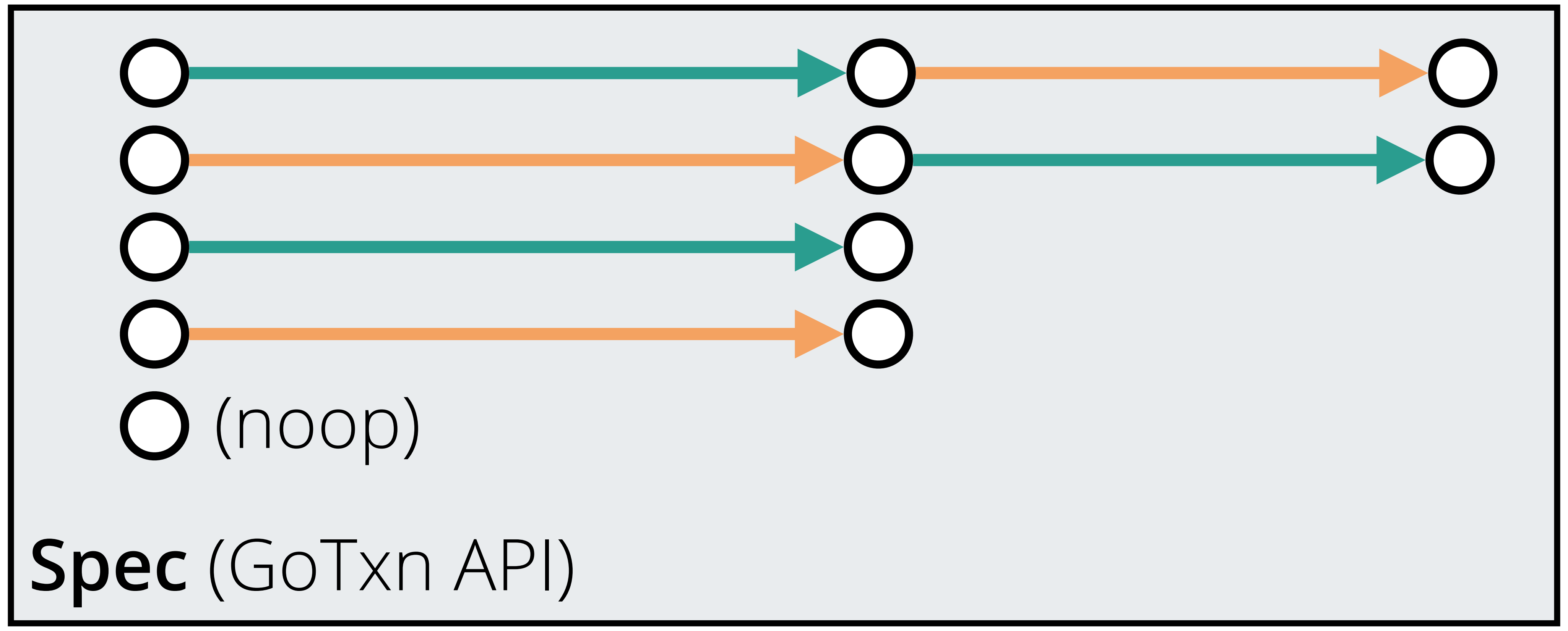
Lifting specification describes the GoTxn API

[**C**TTJKZ, OSDI '21]

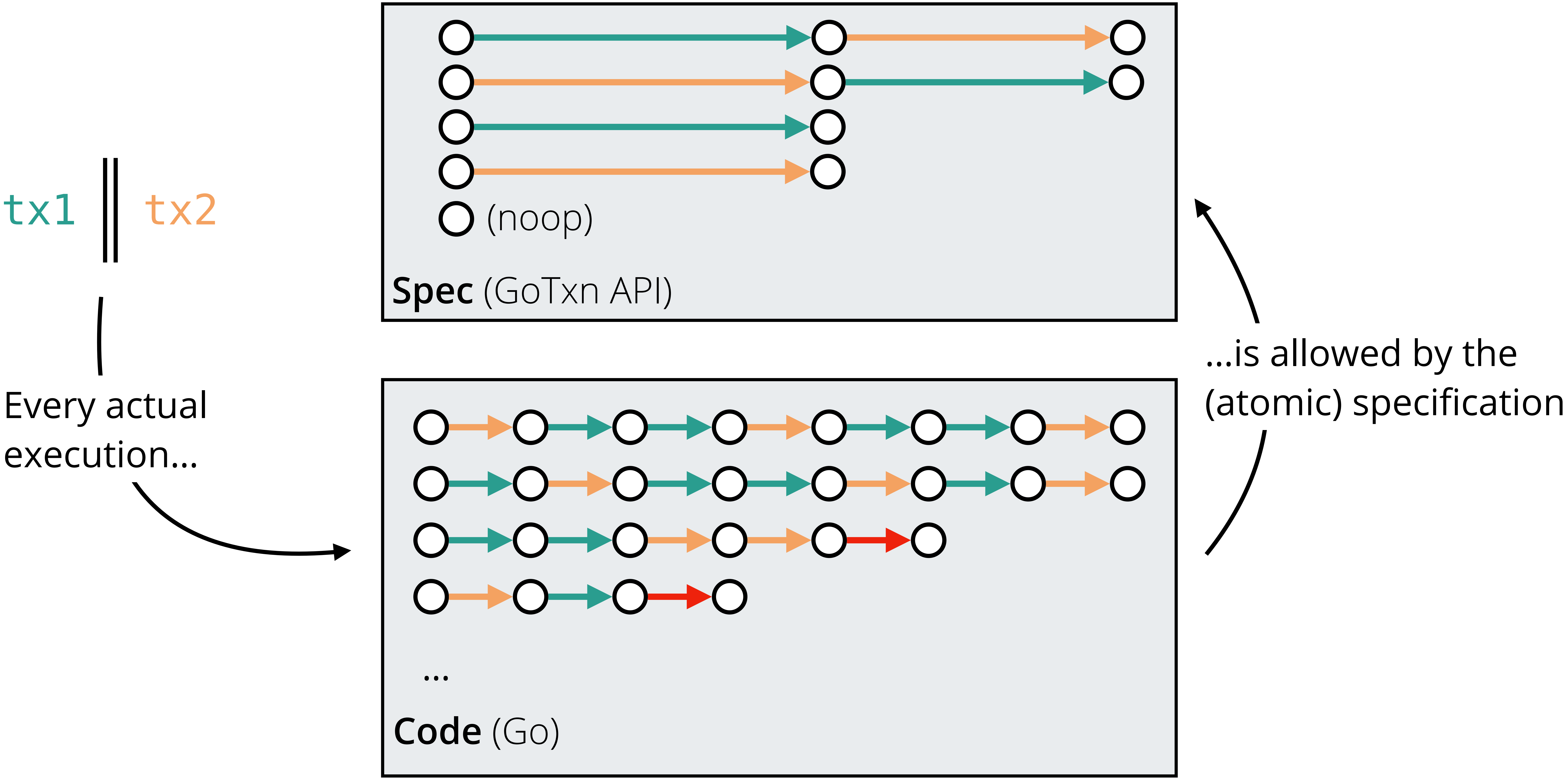


Complete GoTxn specification





Complete GoTxn specification



Technical note: Coq proof shows refinement

$$
\begin{aligned}
& e_c \; : \texttt{Goose<Disk>} \\
\preceq \; & e_s \; : \texttt{Goose<Txn>}
\end{aligned}
$$

Technical note: Coq proof shows refinement

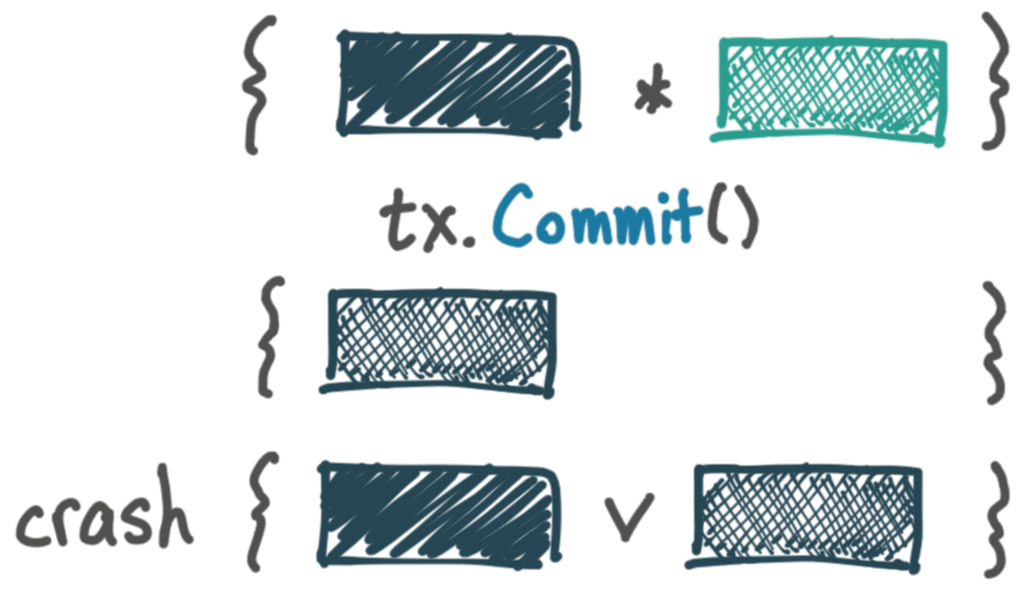
e_c : `Goose<Disk>`

$\preceq$ e_s : `Goose<Txn>`

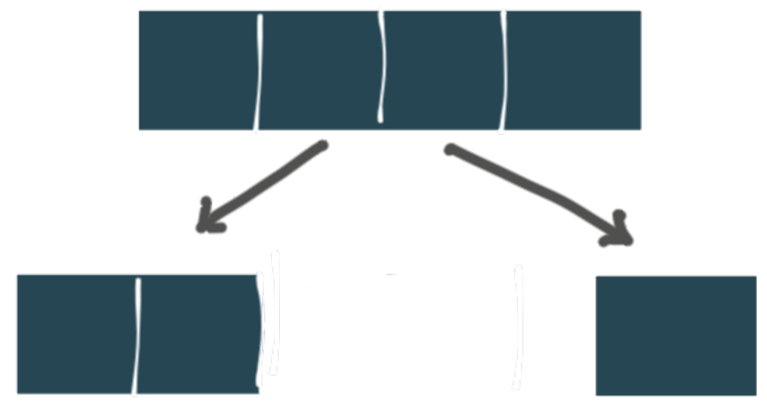
```
tx := Begin()
v := tx.Read(0)
tx.Write(1, v)
tx.Commit()
```

```
atomically {
  v ← Read(0);
  Write(1, v);
}
```

Summary of specifying transaction system



Perennial logic supports specifying and proving crash and concurrent behavior



Lifting specification describes concurrent transactions

Perennial

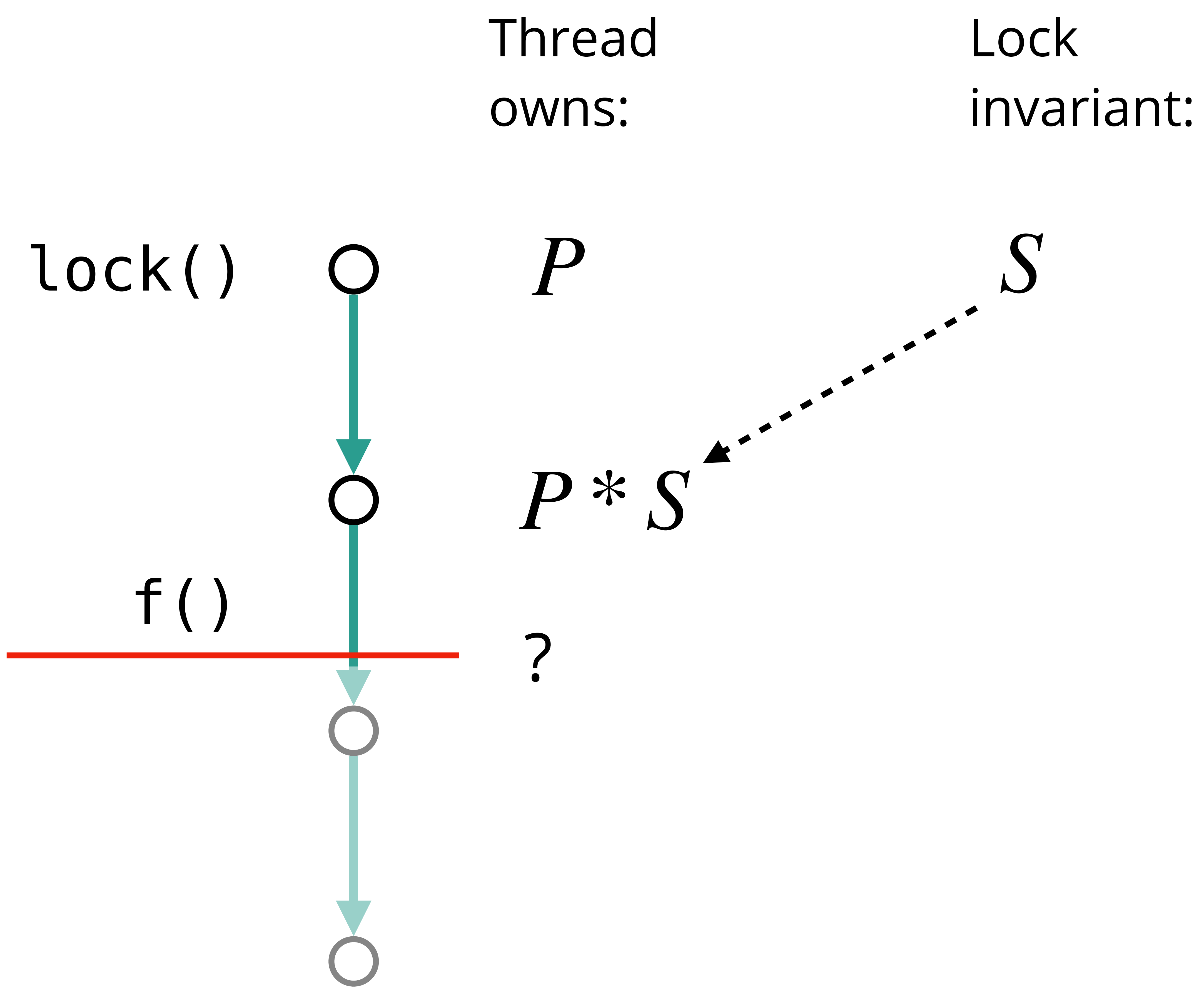
Key judgment: Hoare "quadruple"

$$\{P\}\ e\ \{Q\}\ \{T\}$$

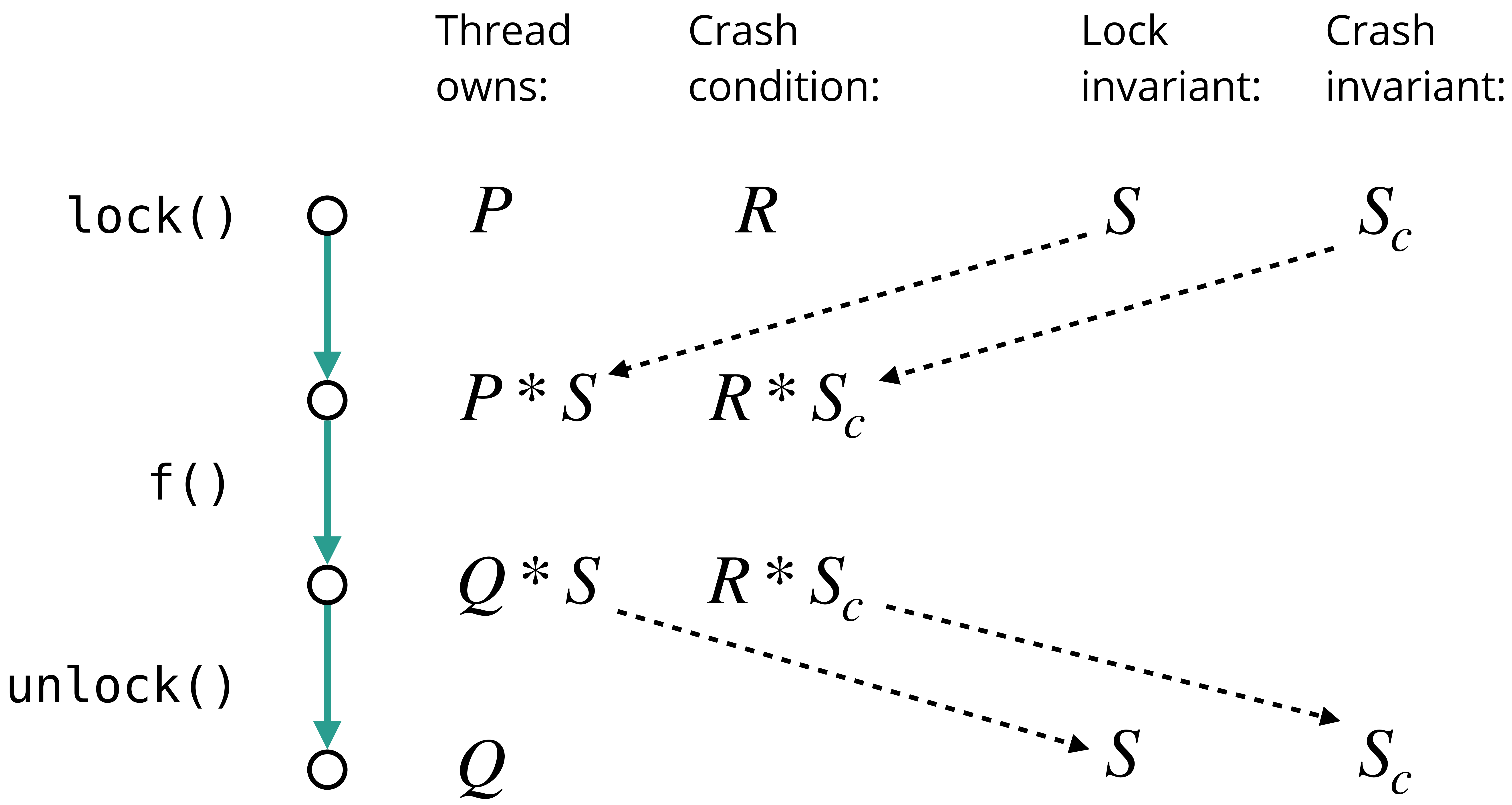
"crash condition"

If we halt e during its execution, T will hold

Ownership with crashes is tricky



Crash locks support locking durable state



Other challenges

Prove recovery is idempotent

Durable linearizability specifications

Connection to Go code

Roadmap

DaisyNFS — **File-system code** implemented with transactions

Specification for transactions — **Specification** that bridges the two ✓

GoTxn — **Transaction system** gives atomicity

Crashes

Concurrency

Roadmap

DaisyNFS — **File-system code** implemented with transactions

Specification for transactions — **Specification** that bridges the two ✓

GoTxn — **Transaction system** gives atomicity

- Crashes
- Concurrency

GoTxn

Implementing GoTxn

```
tx := Begin()
v := tx.Read(0)
tx.Write(1, v)
tx.Write(2, v)
tx.Commit()
```

Implementing GoTxn

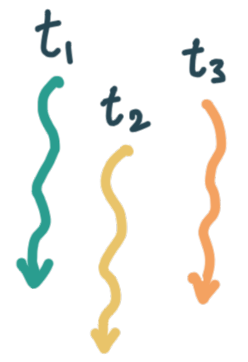
```
tx := Begin()
v := tx.Read(0)
tx.Write(1, v)
tx.Write(2, v)
tx.Commit()
```

Write-ahead logging
makes writes **crash-safe**

Implementing GoTxn

```
tx := Begin()
v := tx.Read(0)
tx.Write(1, v)
tx.Write(2, v)
tx.Commit()
```

Two-phase locking handles **concurrency**



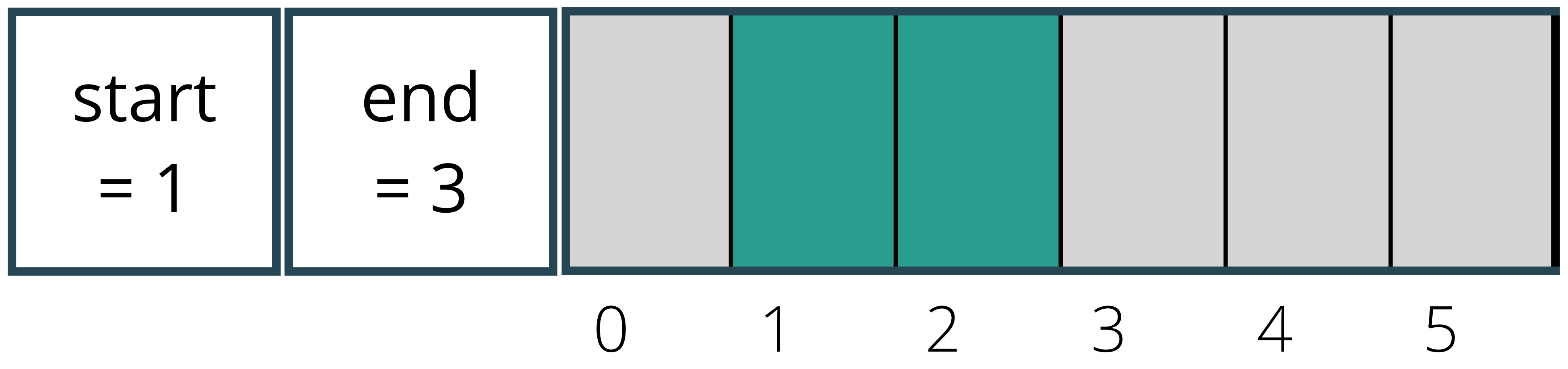
Write-ahead logging makes writes **crash-safe**

Write-ahead logging API

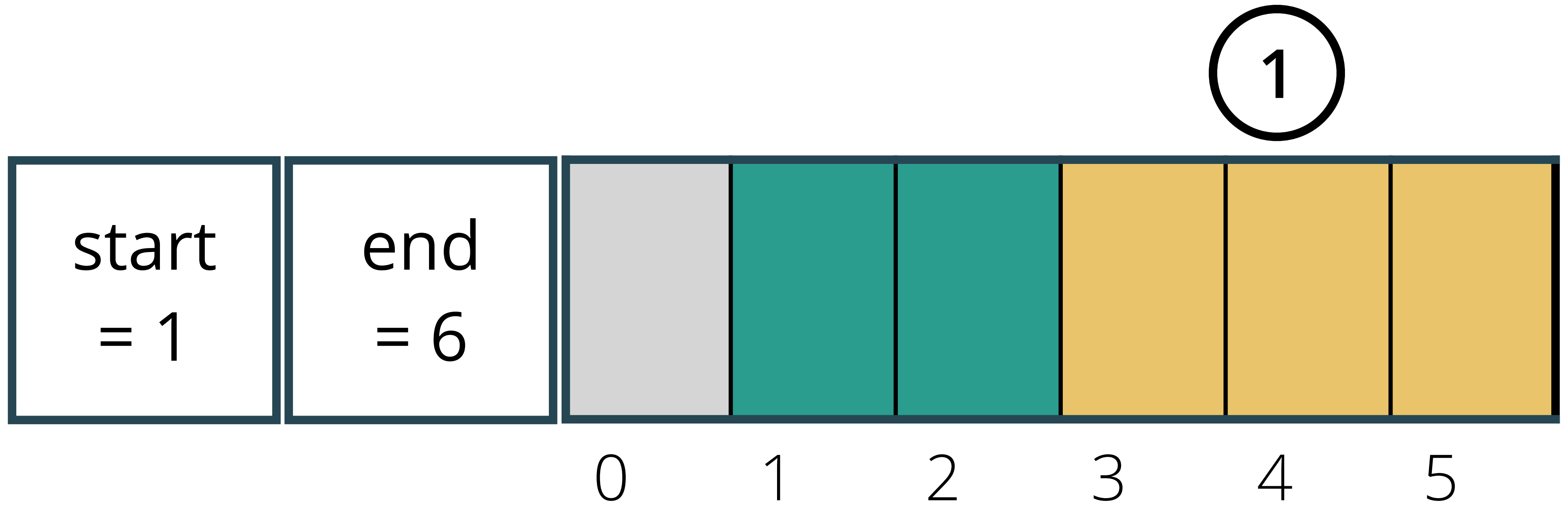
```
func Multiwrite(ws []Update)
type Update struct {
   addr  uint64
   block []byte
}

func Read(a uint64) []byte
func Flush()
```

Write-ahead log (WAL)

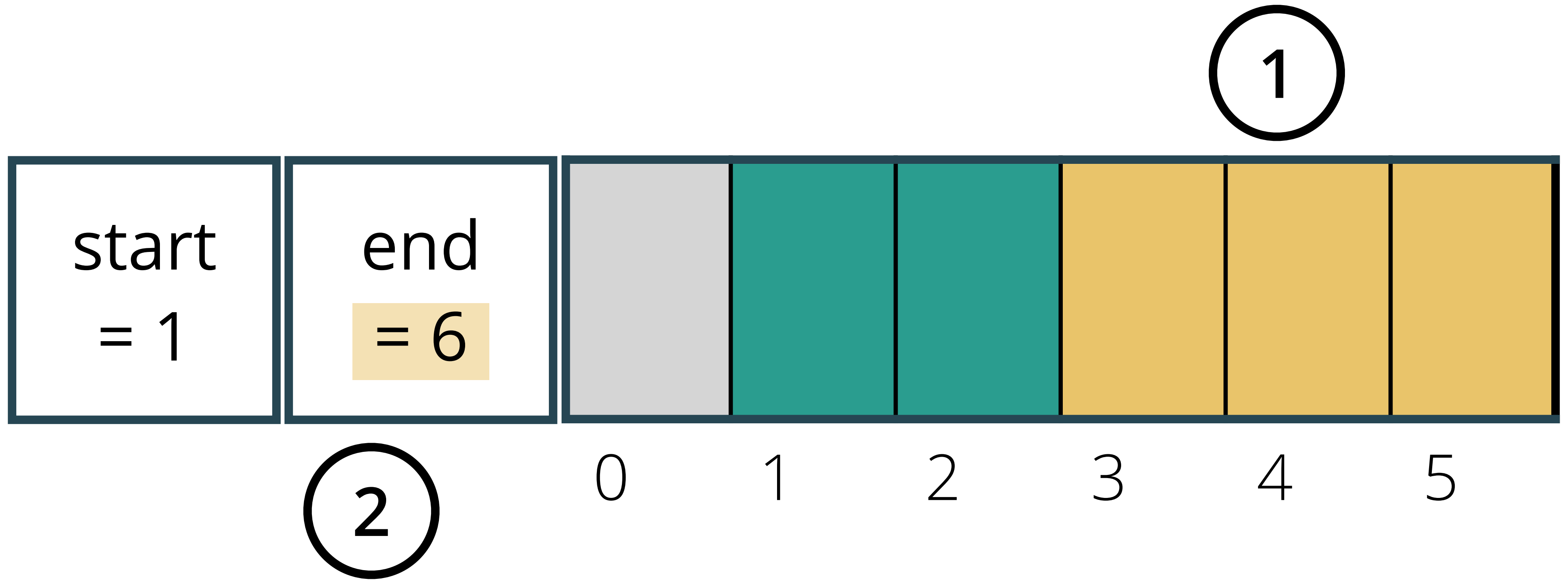
Multi-block atomicity come from circular log



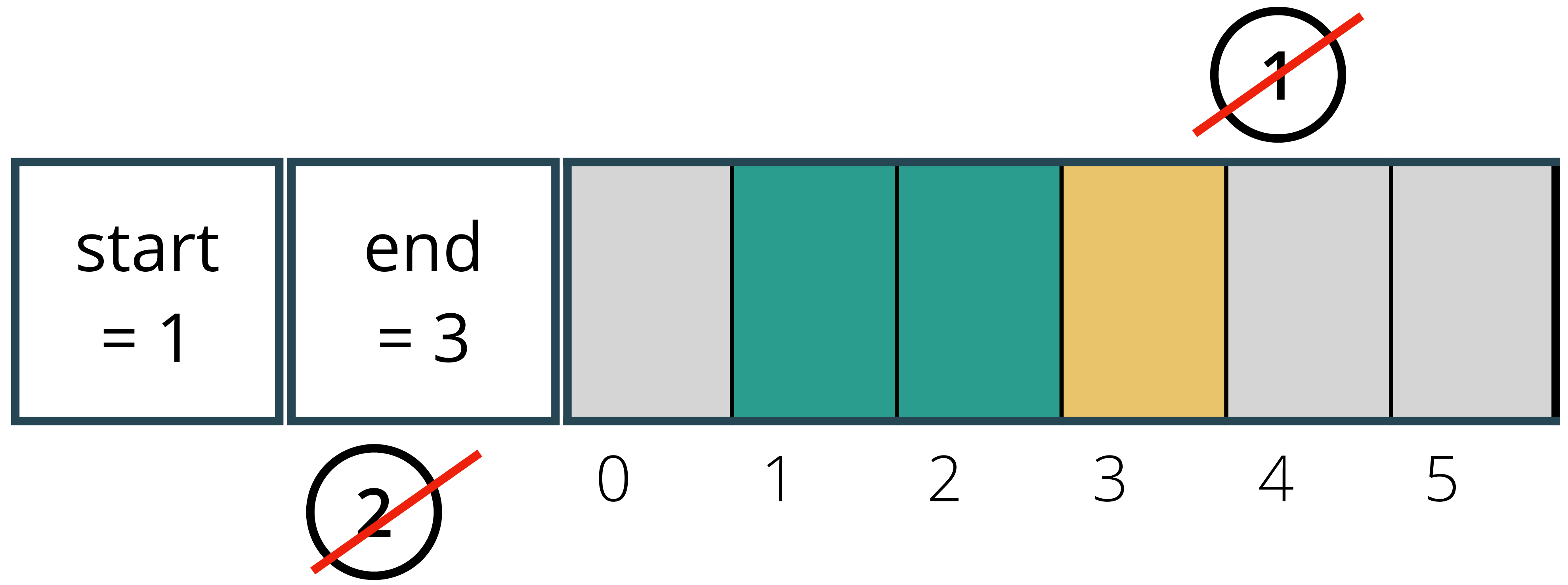
Multi-block atomicity come from circular log



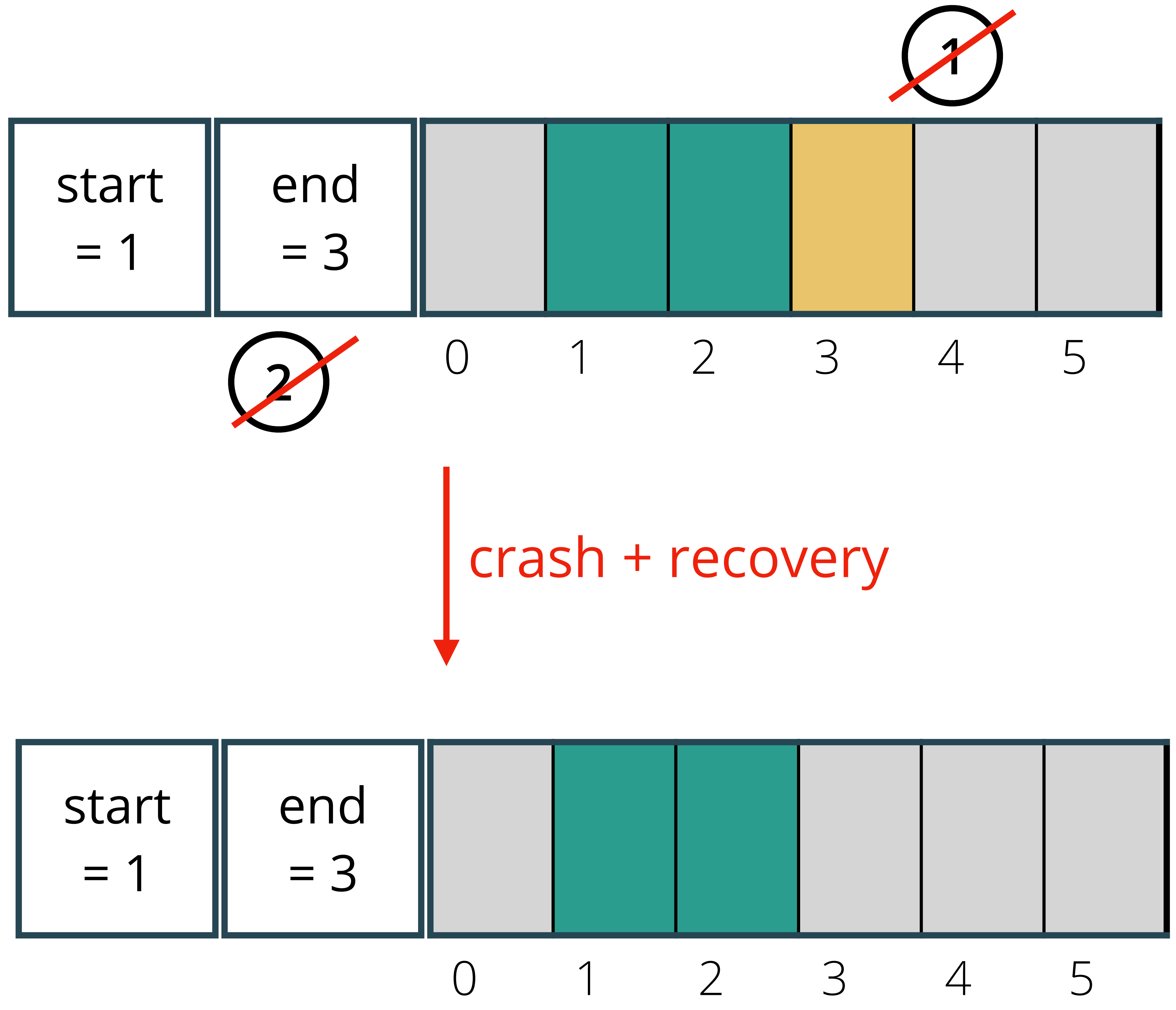
Multi-block atomicity come from circular log



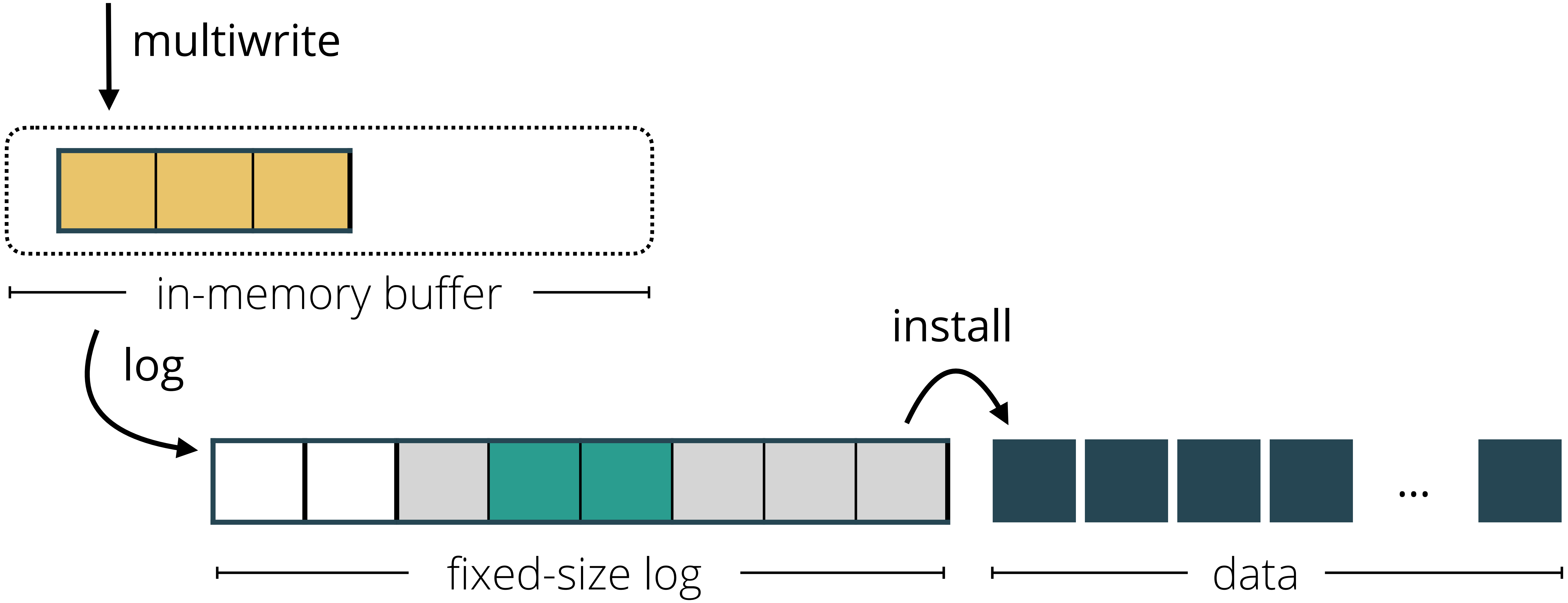
Crash never leaves a partial multiwrite



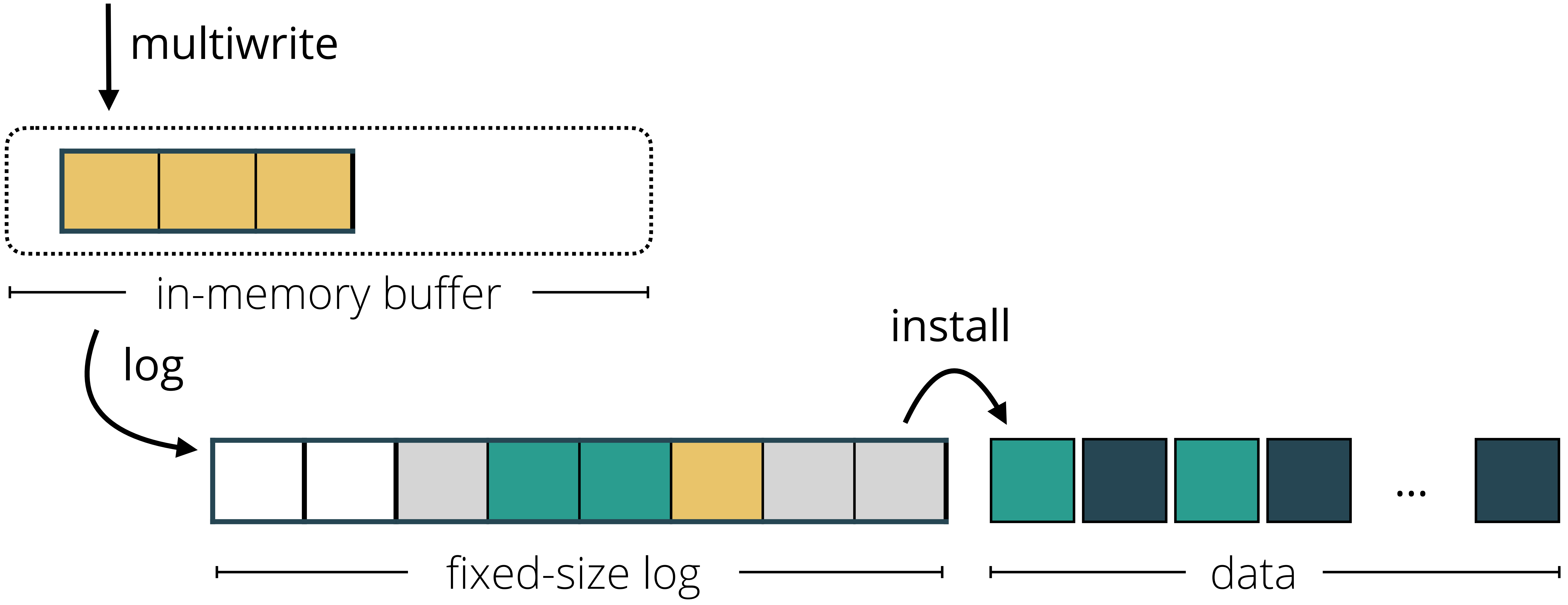
Crash never leaves a partial multiwrite



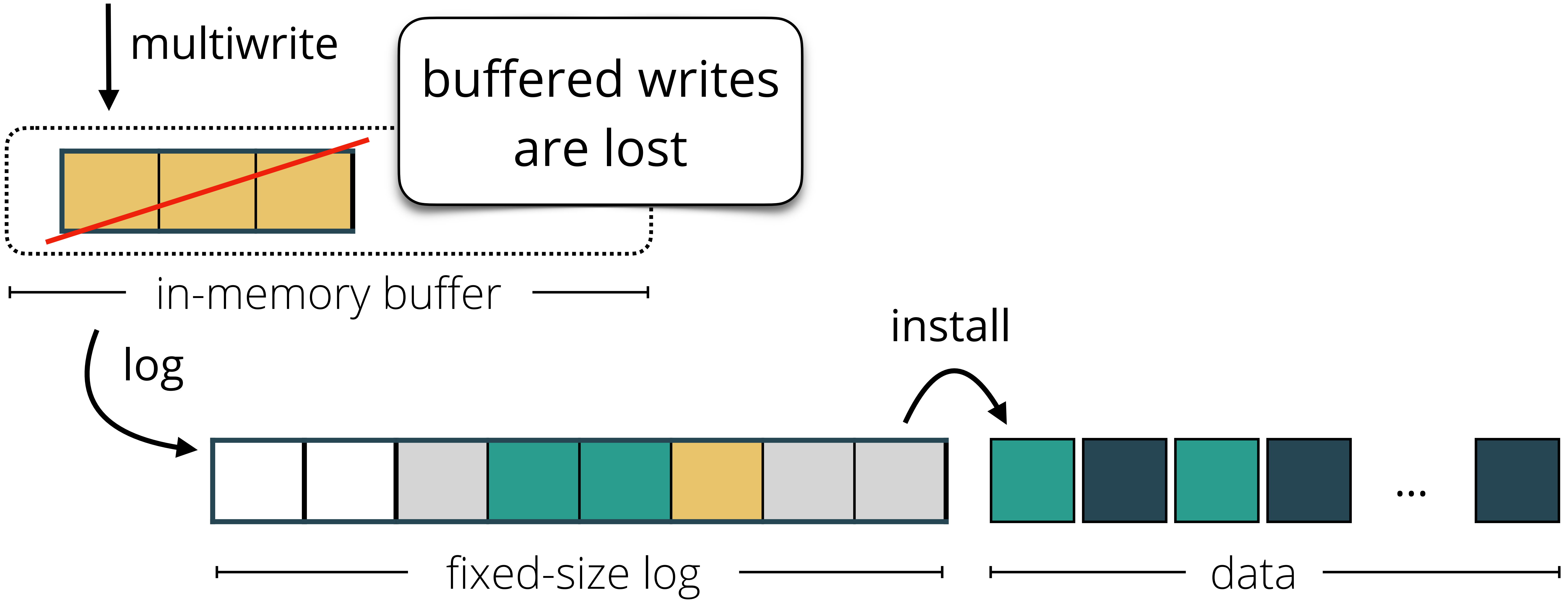
Writing, logging, and installation are concurrent



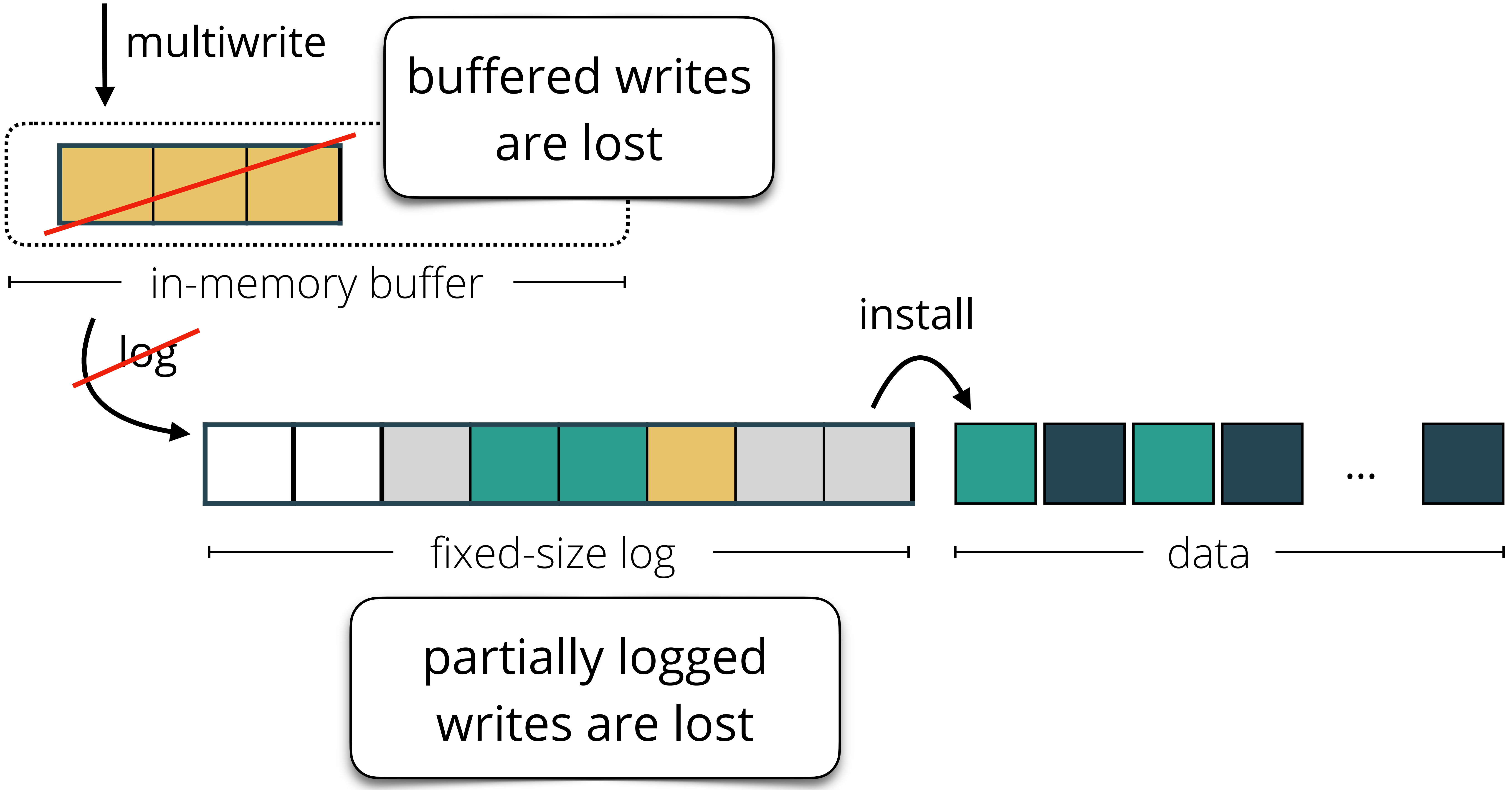
Crashes are complicated in the WAL



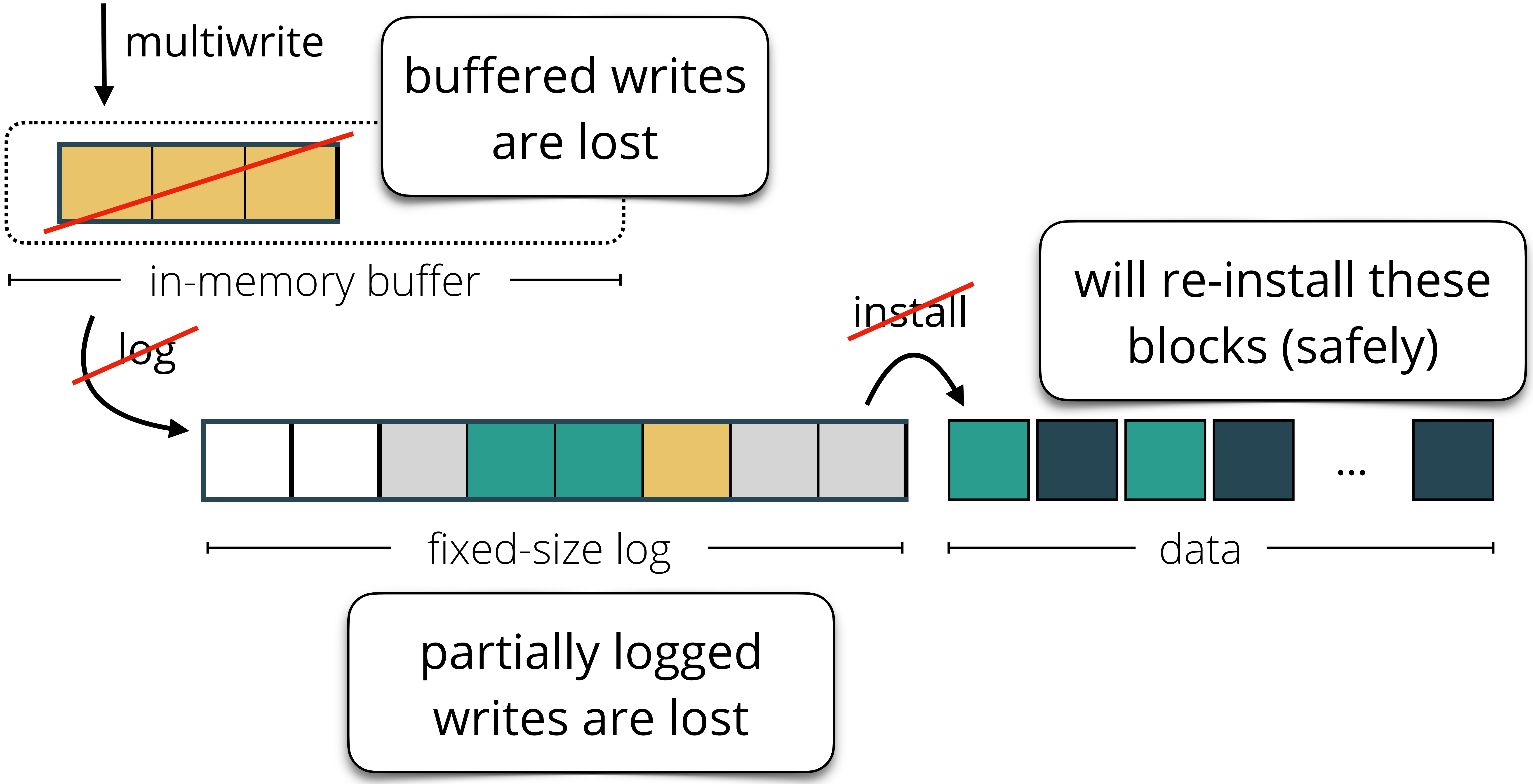
Crashes are complicated in the WAL



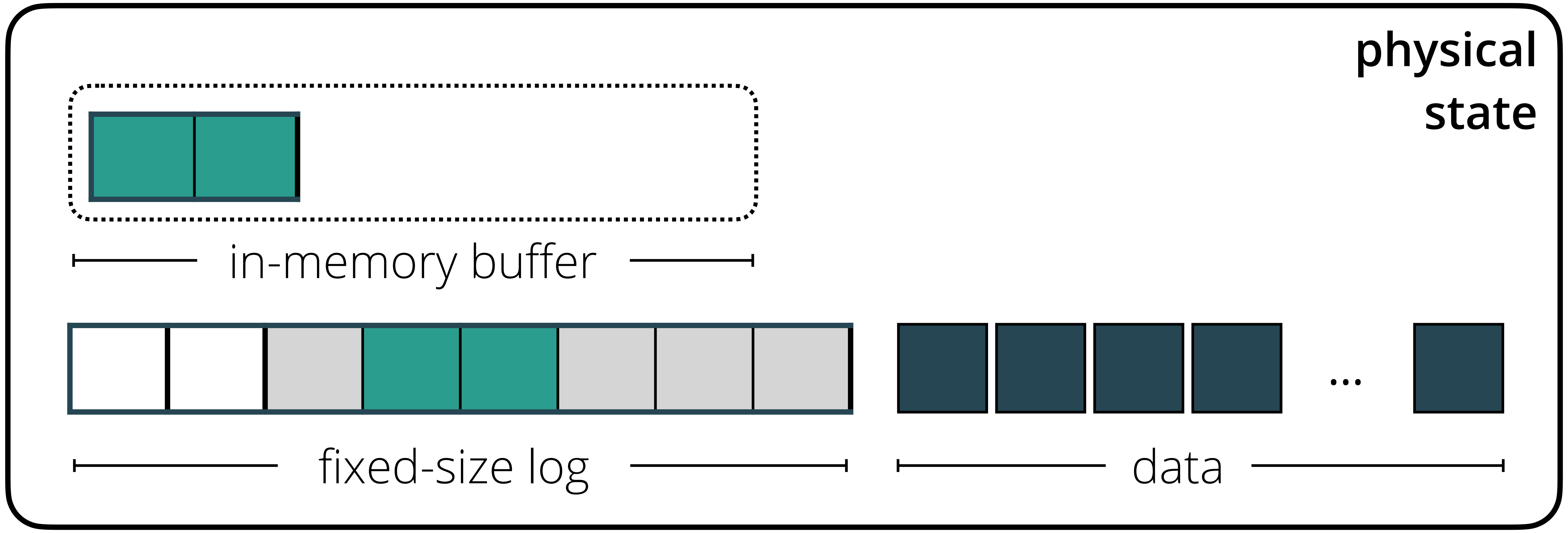
Crashes are complicated in the WAL



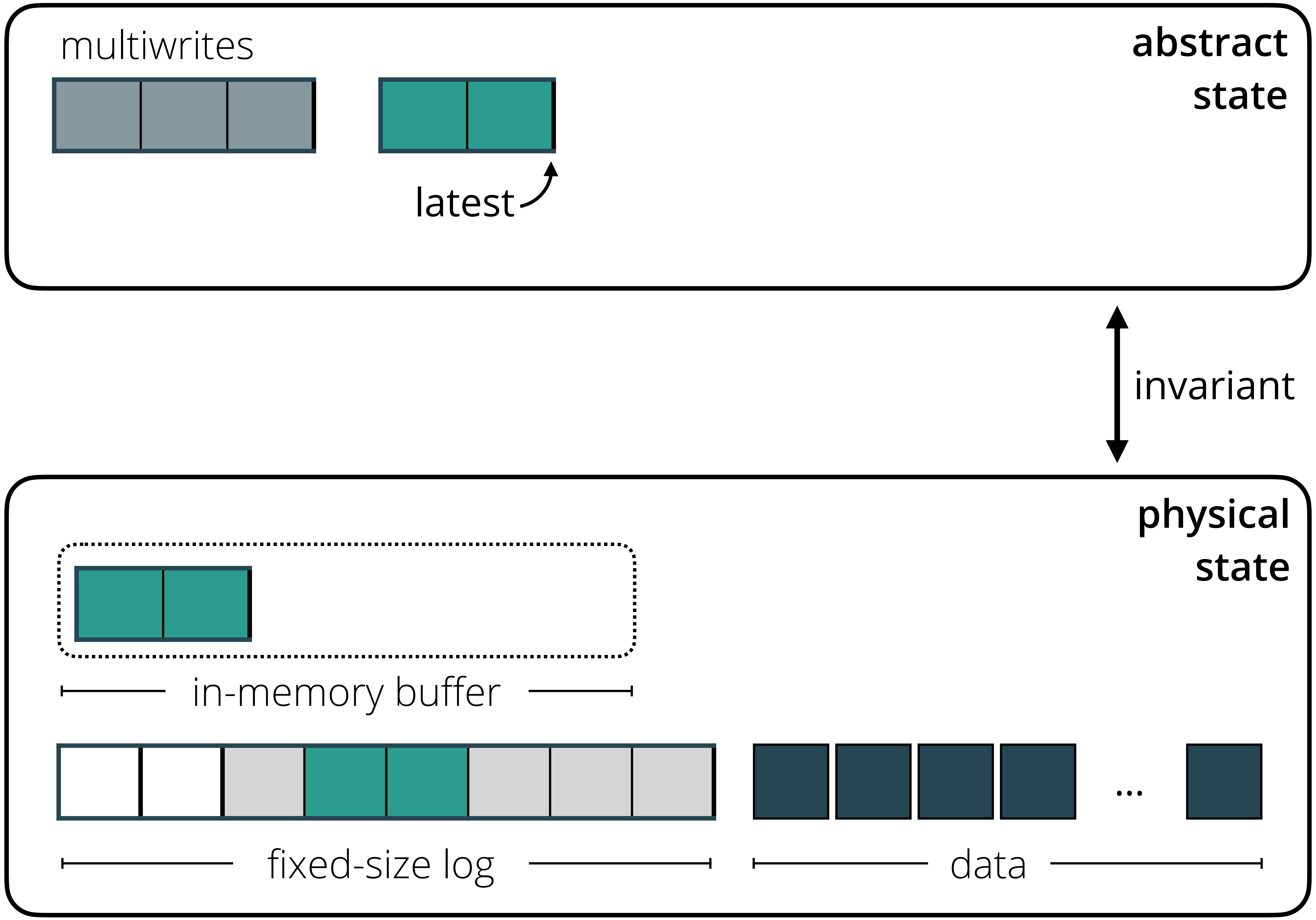
Crashes are complicated in the WAL



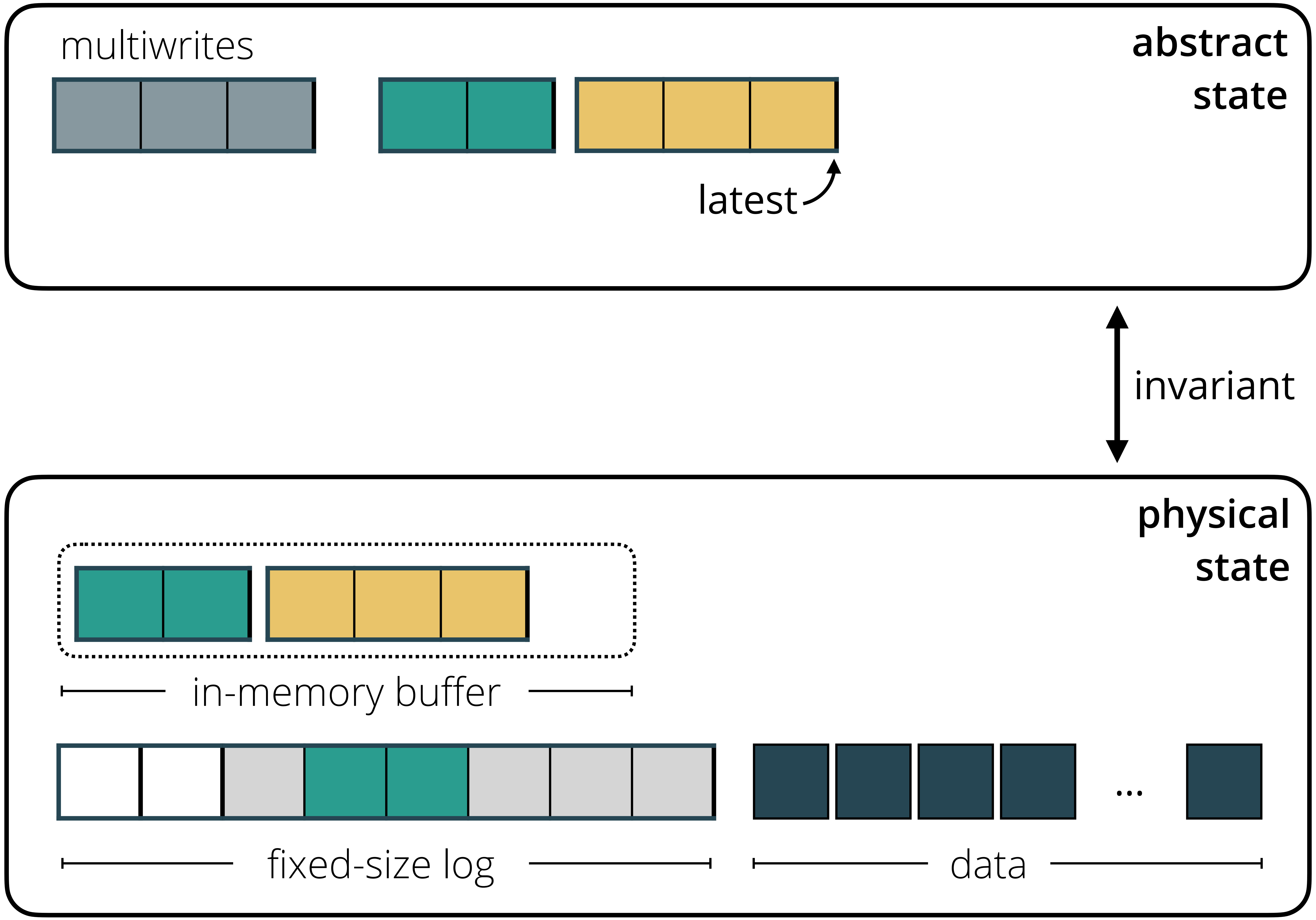
Idea: model WAL as a history of multiwrites



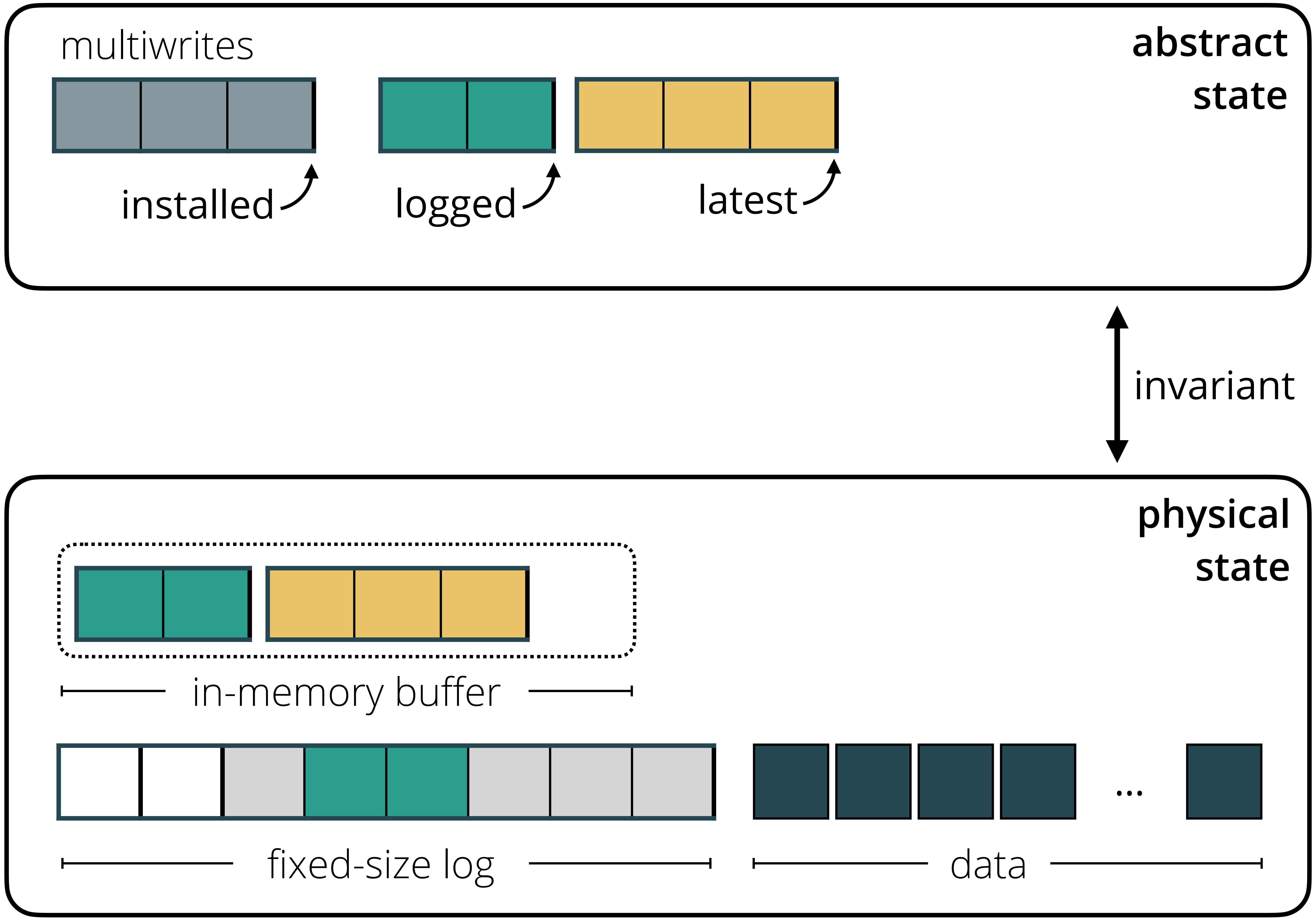
Idea: model WAL as a history of multiwrites



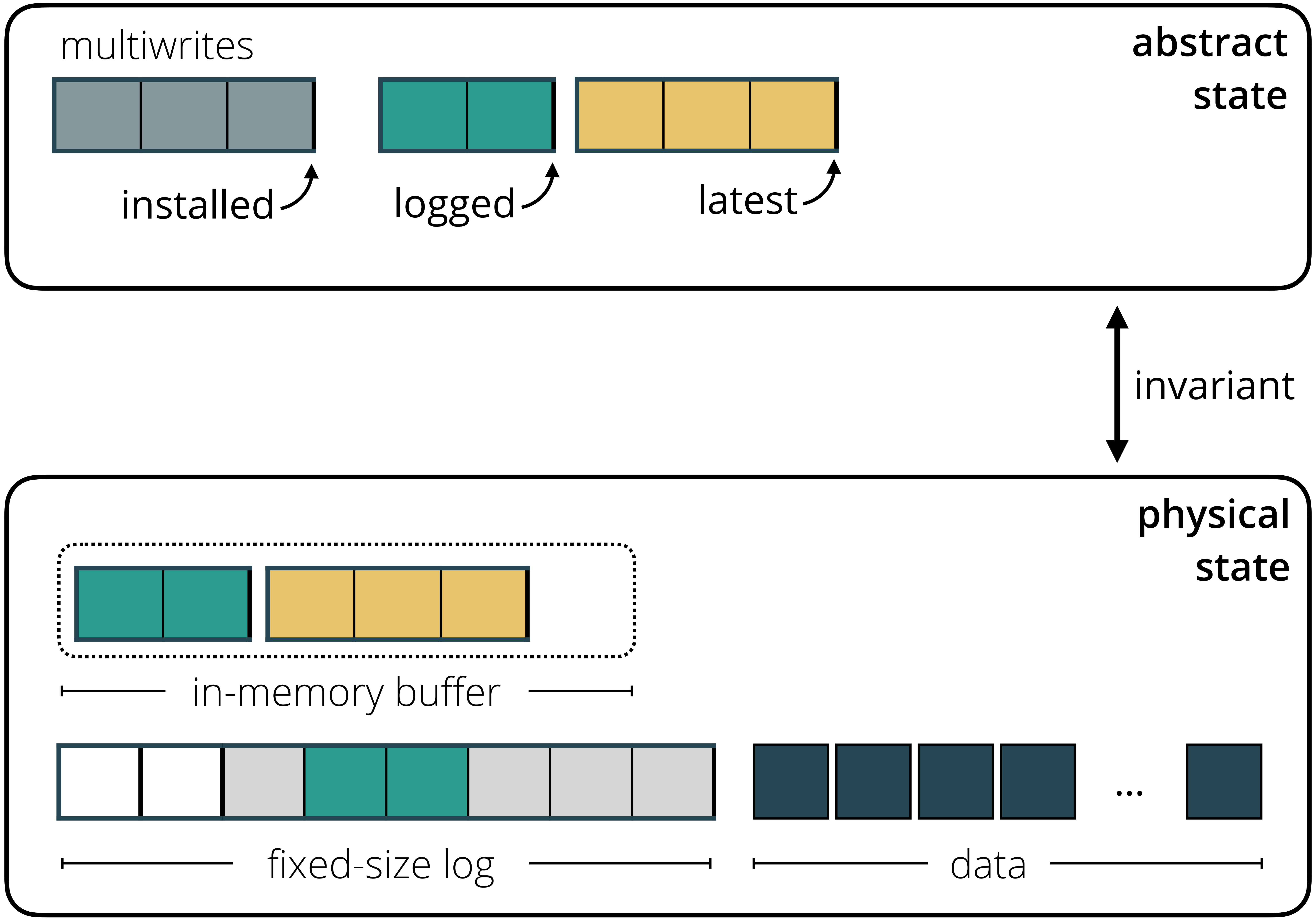
Idea: model WAL as a history of multiwrites



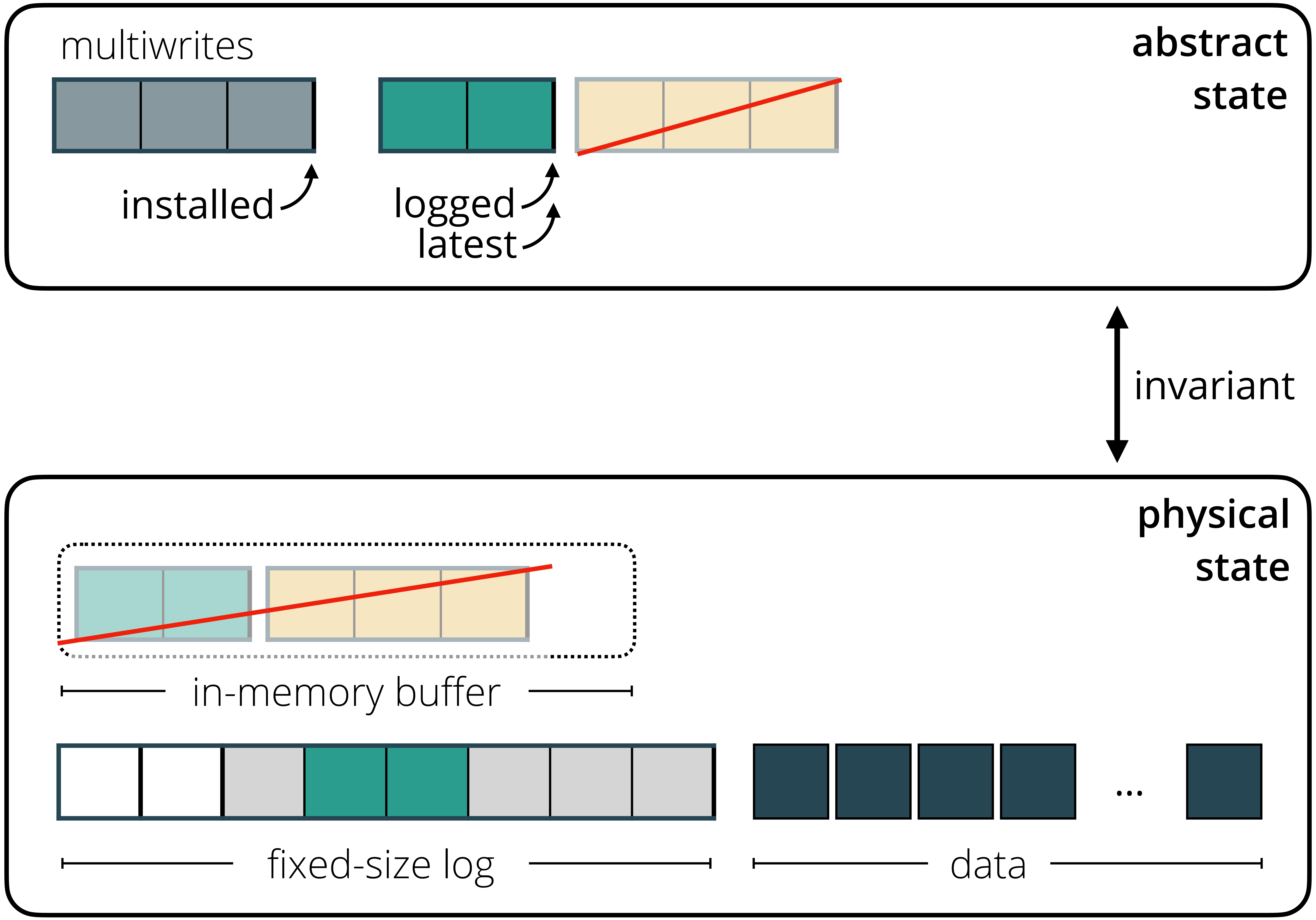
Idea: model WAL as a history of multiwrites



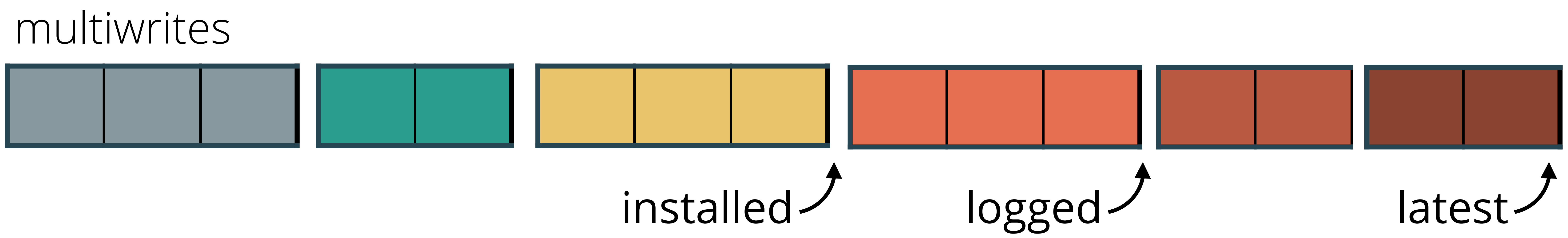
History abstract state crisply expresses atomicity



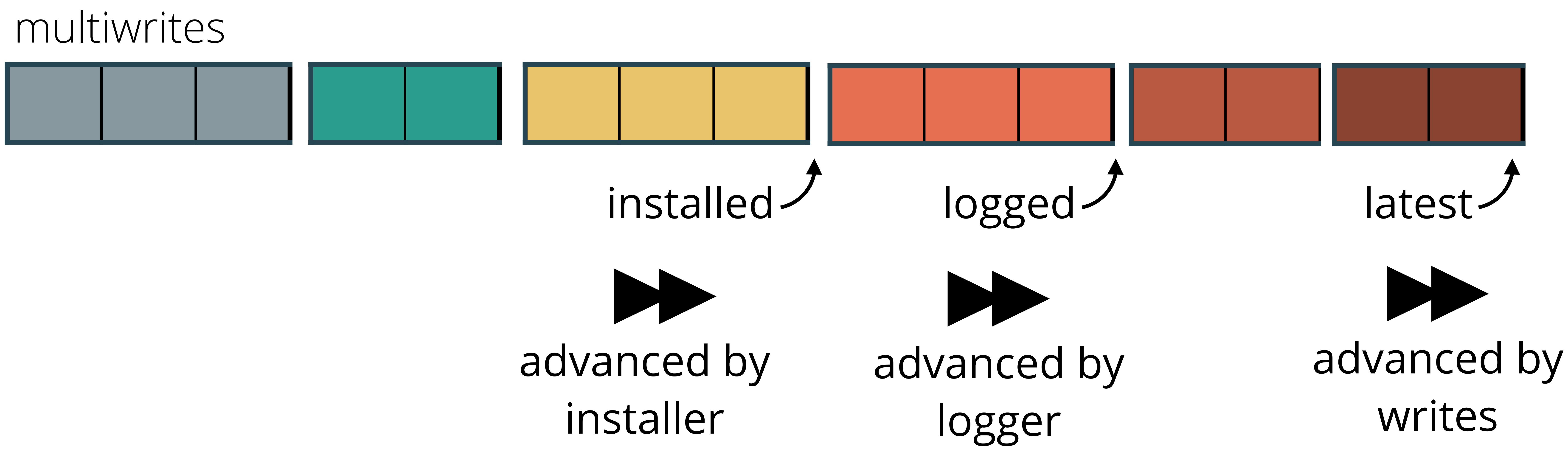
History abstract state crisply expresses atomicity



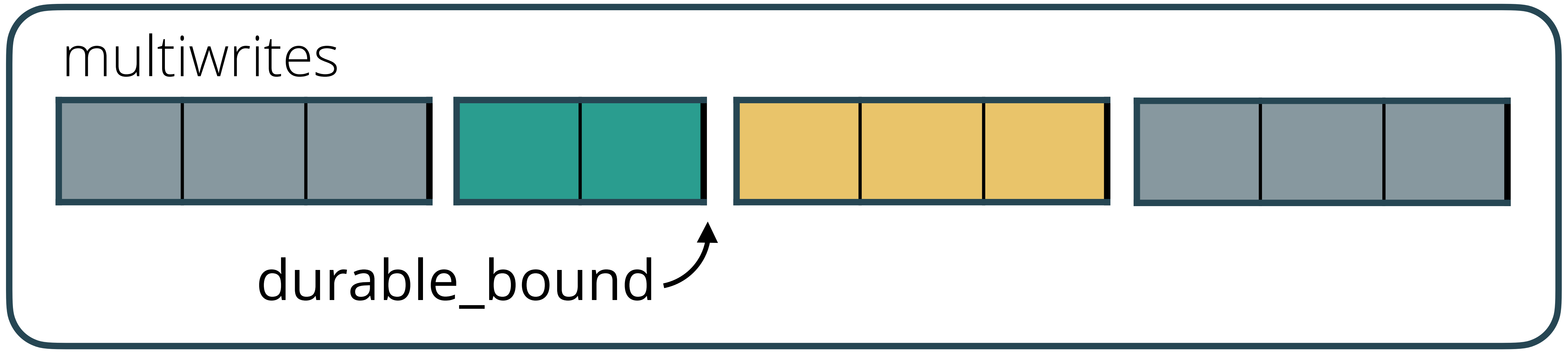
Pointers can all advance concurrently



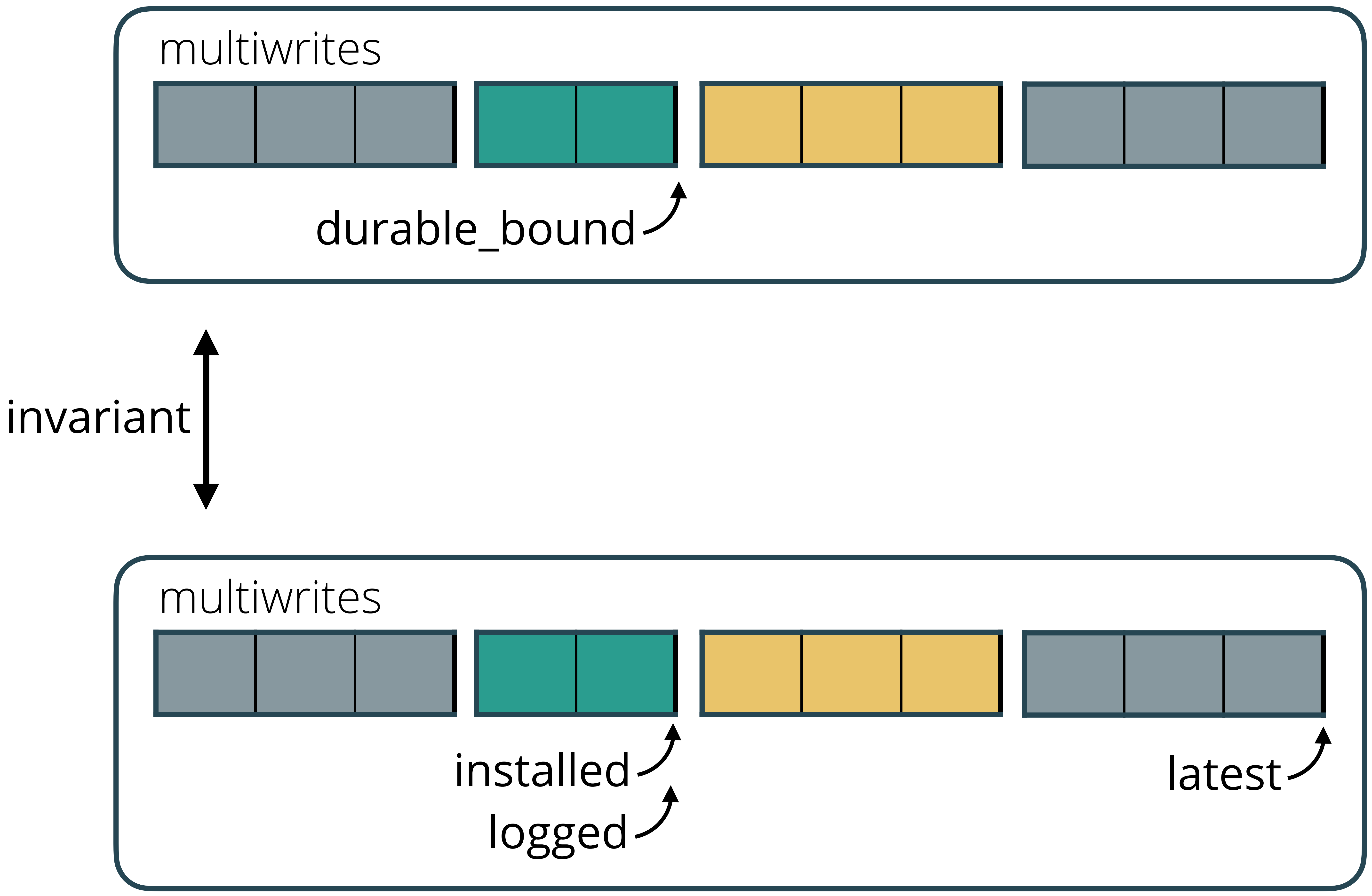
Pointers can all advance concurrently



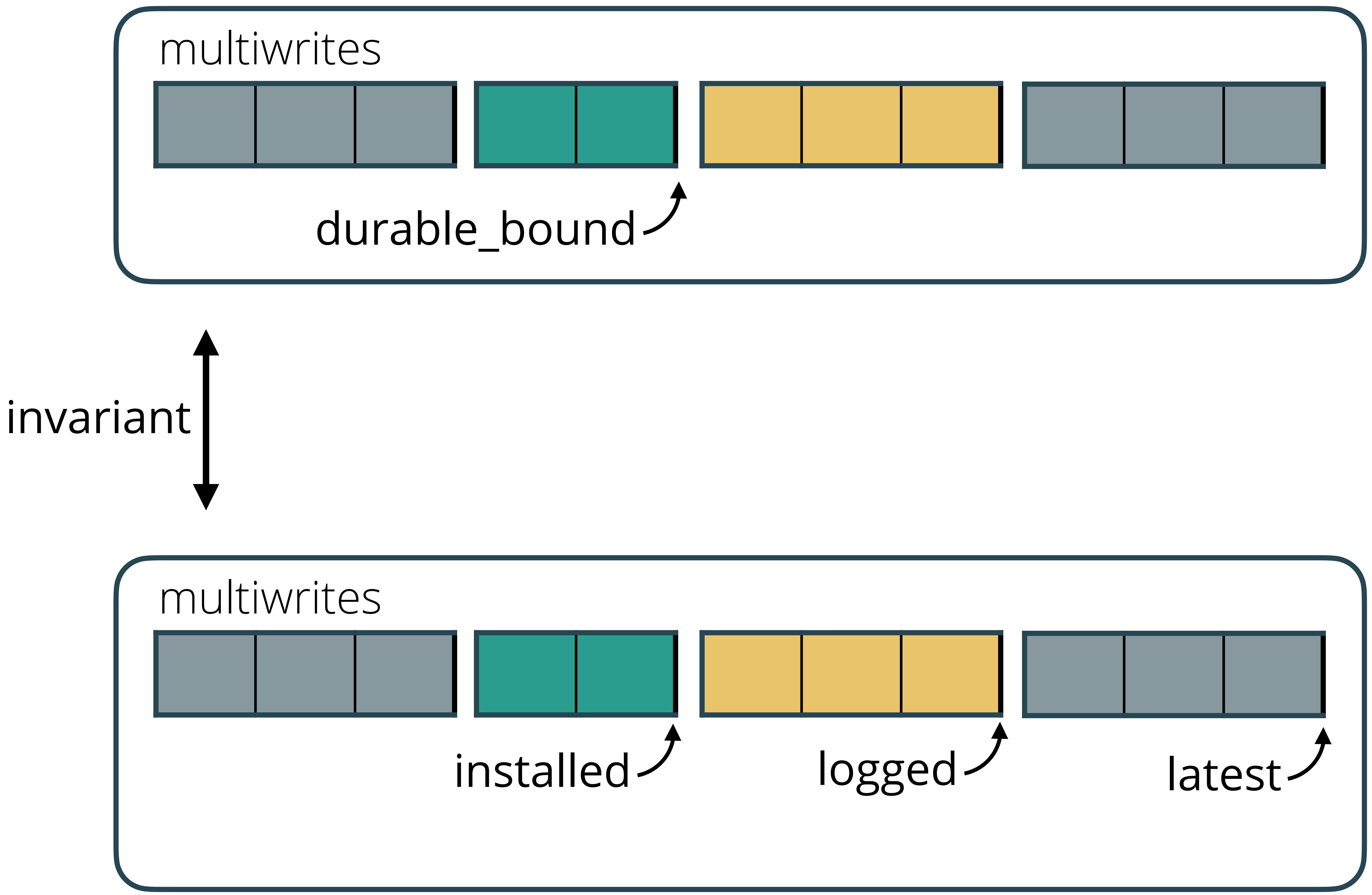
Durable bound hides concurrency for rest of proof



Durable bound hides concurrency for rest of proof



Durable bound hides concurrency for rest of proof



Future work

Can we make this proof less messy?

Can we make it easier to improve the logging design?

Future work

Ca

Ca

vim invariant.v ~/c/p/s/p/wal

invariant.v

```
Record wal_names := mkWalNames
  { circ_name: circ_names;
    cs_name : gname;
    txns_ctx_name : gname;
    txns_name : gname;
    (* TODO: rename being_installed_start_txn to installed_txn since they are now always the same *)
    being_installed_start_txn_name : gname;
    already_installed_name : gname;
    diskEnd_avail_name : gname;
    start_avail_name : gname;
    stable_txn_ids_name : gname;
    logger_pos_name : gname;
    (* TODO: this is the logger's next transaction id? *)
    logger_txn_id_name : gname;
    (* this is the pos/txnid captured by the installer when it starts installing *)
    (* this is used for the lock invariant *)
    installer_pos_mem_name : gname;
    installer_txn_id_mem_name : gname;
    (* this is used for the wal invariant *)
    installer_pos_name : gname;
    installer_txn_id_name : gname;
    (* this is the in-memory diskEnd (not the on-disk diskEnd) *)
    (* it's used to break up has_updates for the circular queue so that the installer can Advance just to
    that point *)
    diskEnd_mem_name : gname;
    diskEnd_mem_txn_id_name : gname;
    installed_pos_mem_name : gname;
    installed_txn_id_mem_name : gname;
    (* the on-disk diskEnd for the interface invariant instead of the lock invariant *)
    diskEnd_name : gname;
    diskEnd_txn_id_name : gname;
    base_disk_name : gname;
  }.
```

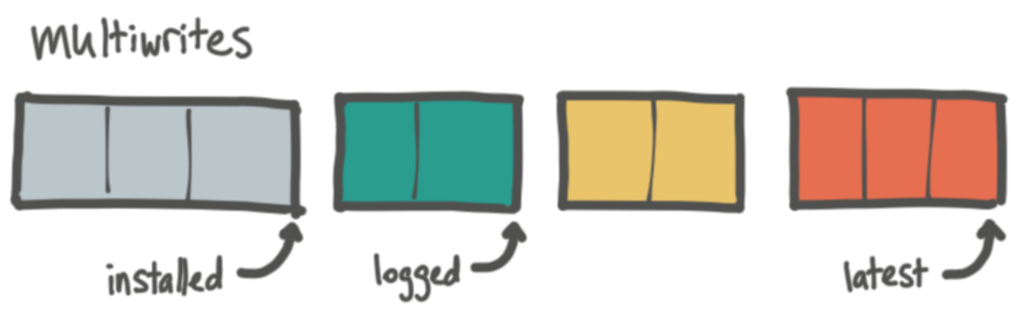
master   coq   83:1   7%

Future work

Can we make this proof less messy?

Can we make it easier to improve the logging design?

Summary of proving the WAL in GoTxn



Abstract state for write-ahead logging based on history of multiwrites and internal pointers

Lower bound on durable state hides concurrency

Roadmap

DaisyNFS

File-system code implemented with transactions

Sequential reasoning

Specification for transactions

Specification that bridges the two ✓

GoTxn

Transaction system gives atomicity

Crashes

t_1 t_2 t_3 Concurrenc ✓

Roadmap

DaisyNFS

File-system code implemented with transactions

Sequential reasoning

Specification for transactions

Specification that bridges the two ✓

GoTxn

Transaction system gives atomicity

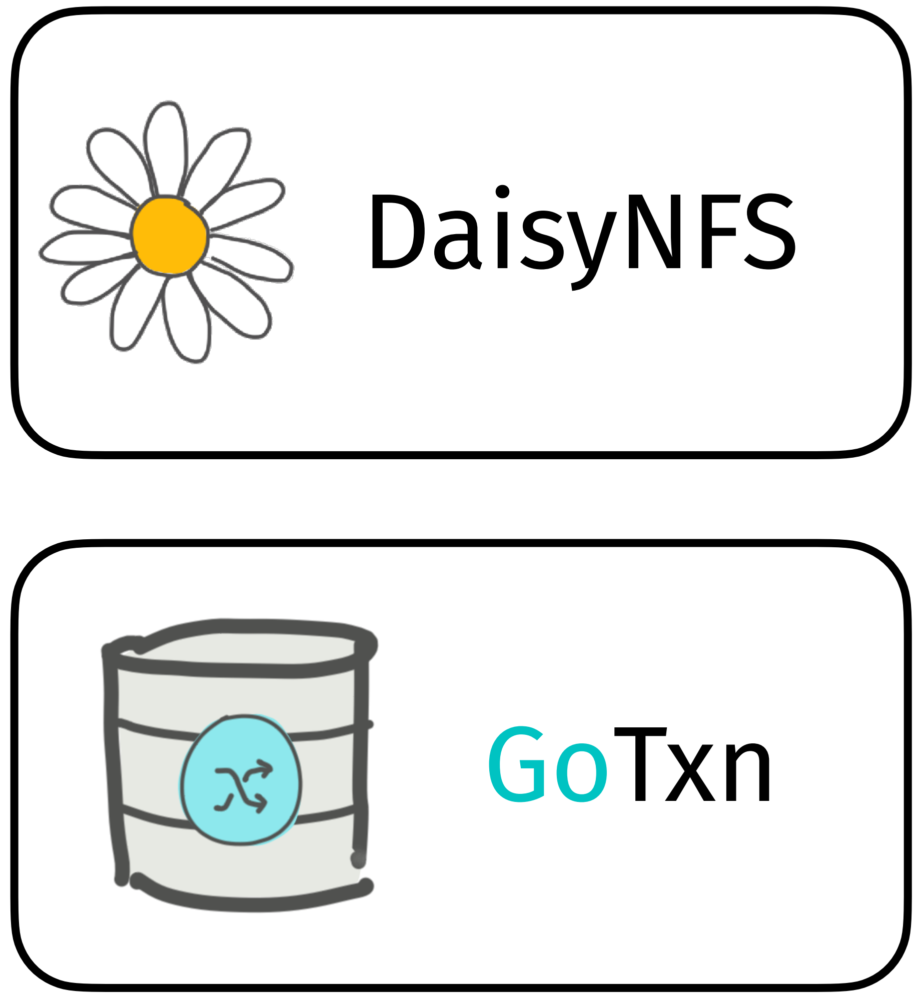
Crashes

Concurrenc ✓

DaisyNFS

DaisyNFS is a verified file system on top of GoTxn

[**C**TTKZ, OSDI ’22]




```
GETATTR, SETATTR, READ, WRITE,
CREATE, REMOVE, MKDIR, RENAME,
LOOKUP, READDIR,
FSINFO, PATHCONF, FSSTAT
```

NFS

```
func Begin() *Txn

func (tx *Txn) Read(…)
func (tx *Txn) Write(…)

func (tx *Txn) Commit()
```

transactions

Challenges

Specification: formalizing NFS

Proof: leveraging atomicity from GoTxn

Implementation: fitting operations into transactions

Specification: how to formalize NFS (RFC 1813)?

INFORMATIONAL

Network Working Group — B. Callaghan
Request for Comments: 1813 — B. Pawlowski
Category: Informational — P. Staubach
Sun Microsystems, Inc.
June 1995

NFS Version 3 Protocol Specification

Status of this Memo

This memo provides information for the Internet community. This memo does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

IESG Note

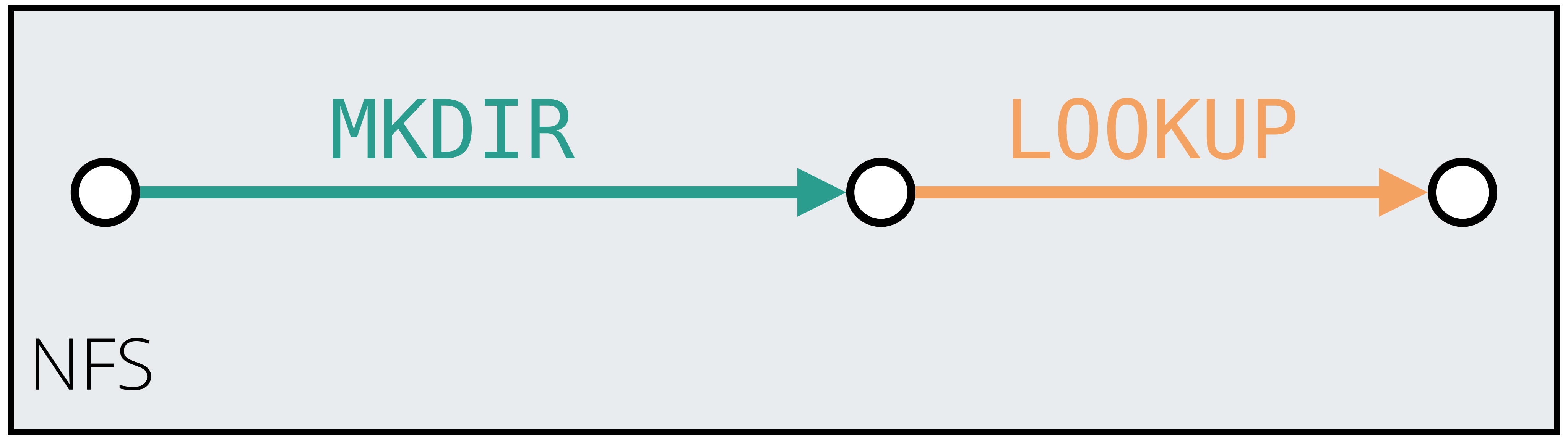
Internet Engineering Steering Group comment: please note that the IETF is not involved in creating or maintaining this specification. This is the significance of the specification not being on the standards track.

Abstract

This paper describes the NFS version 3 protocol. This paper is provided so that people can write compatible implementations.

DaisyNFS's top-level specification

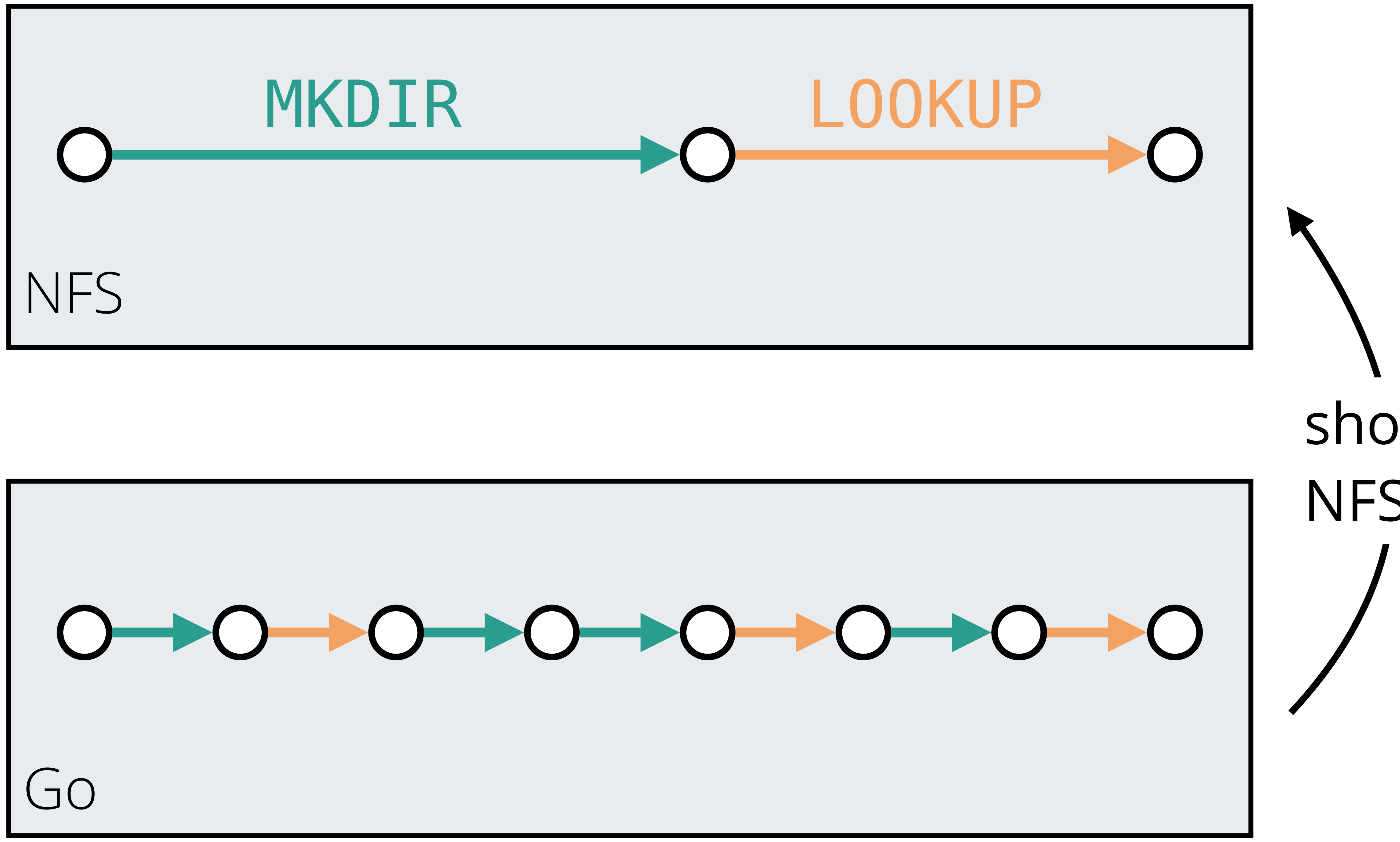
`MKDIR(…)` ‖ `LOOKUP(…)`



DaisyNFS's top-level specification

`MKDIR(…)` ∥ `LOOKUP(…)`

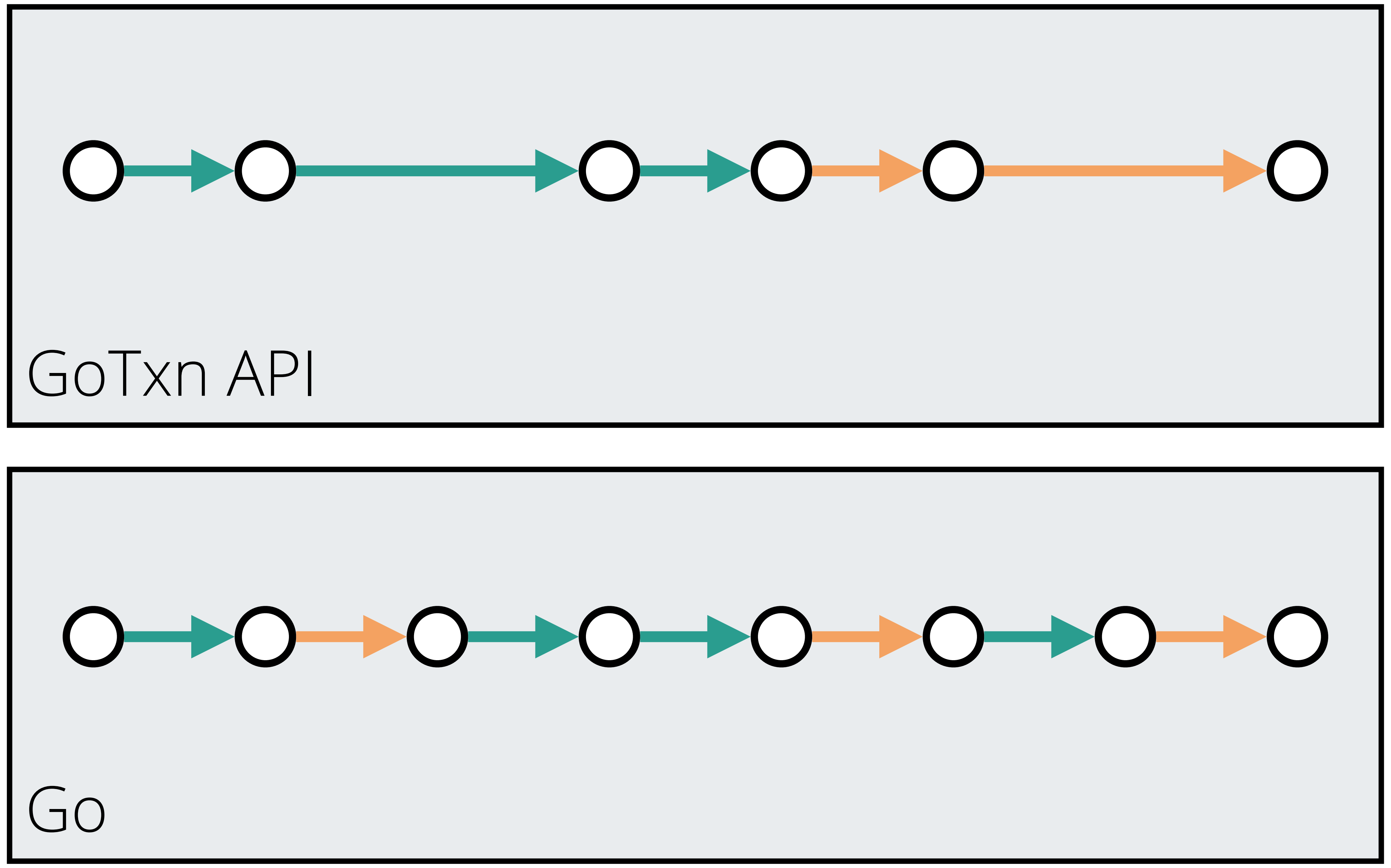
Every `daisy-nfsd` concurrent execution…



should follow (atomic) NFS specification

Proof: compose GoTxn and DaisyNFS proofs

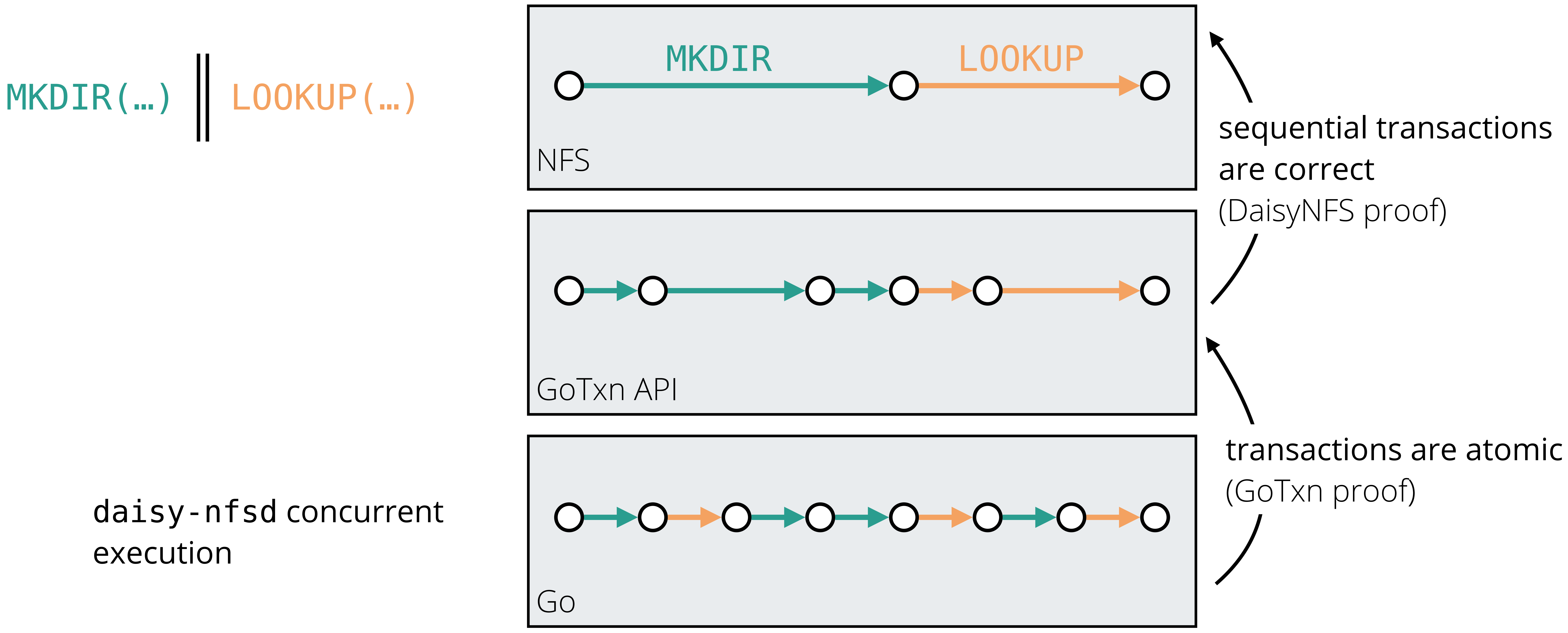
`MKDIR(…)` ‖ `LOOKUP(…)`



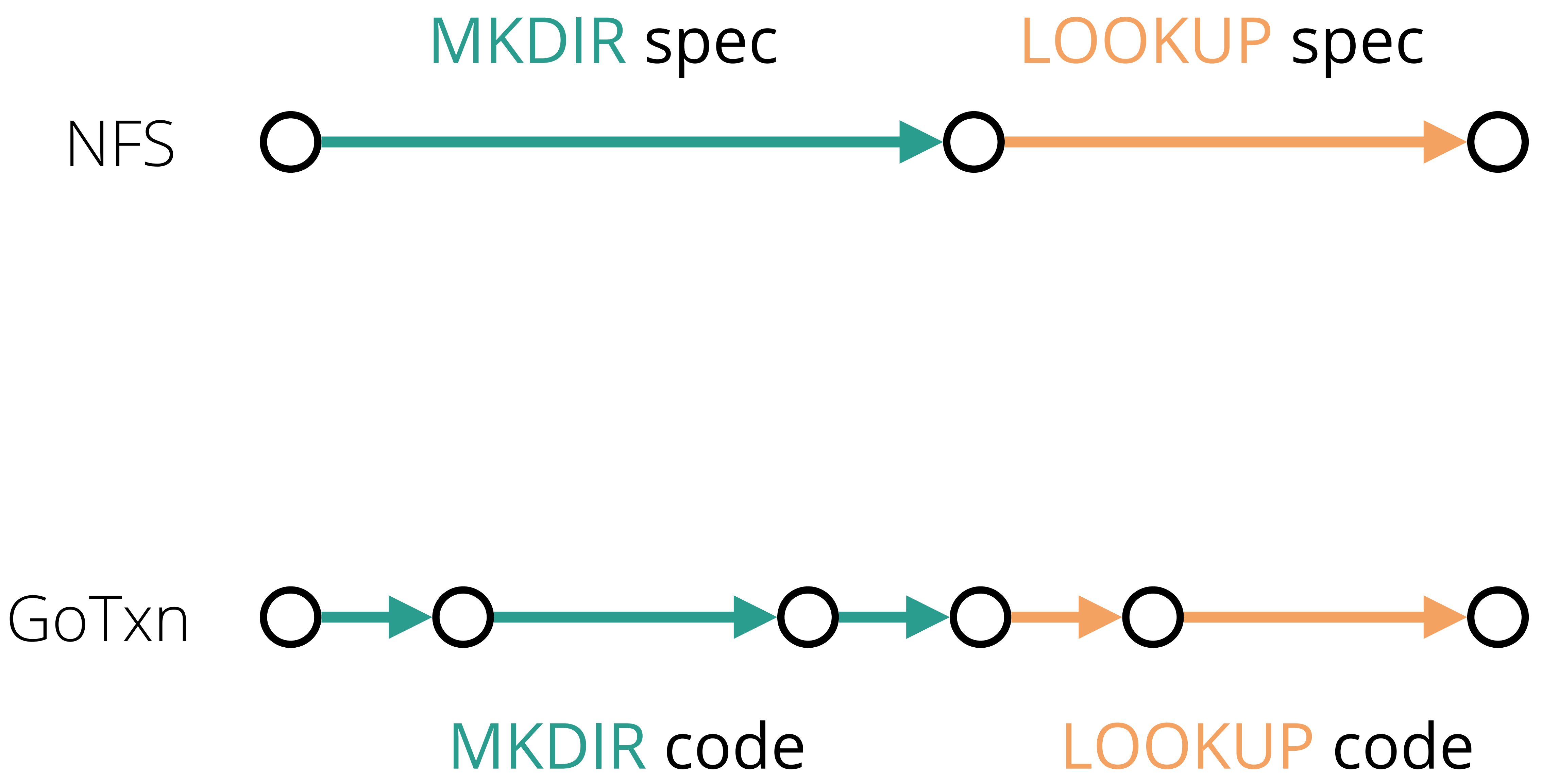
`daisy-nfsd` concurrent execution

transactions are atomic (GoTxn proof)

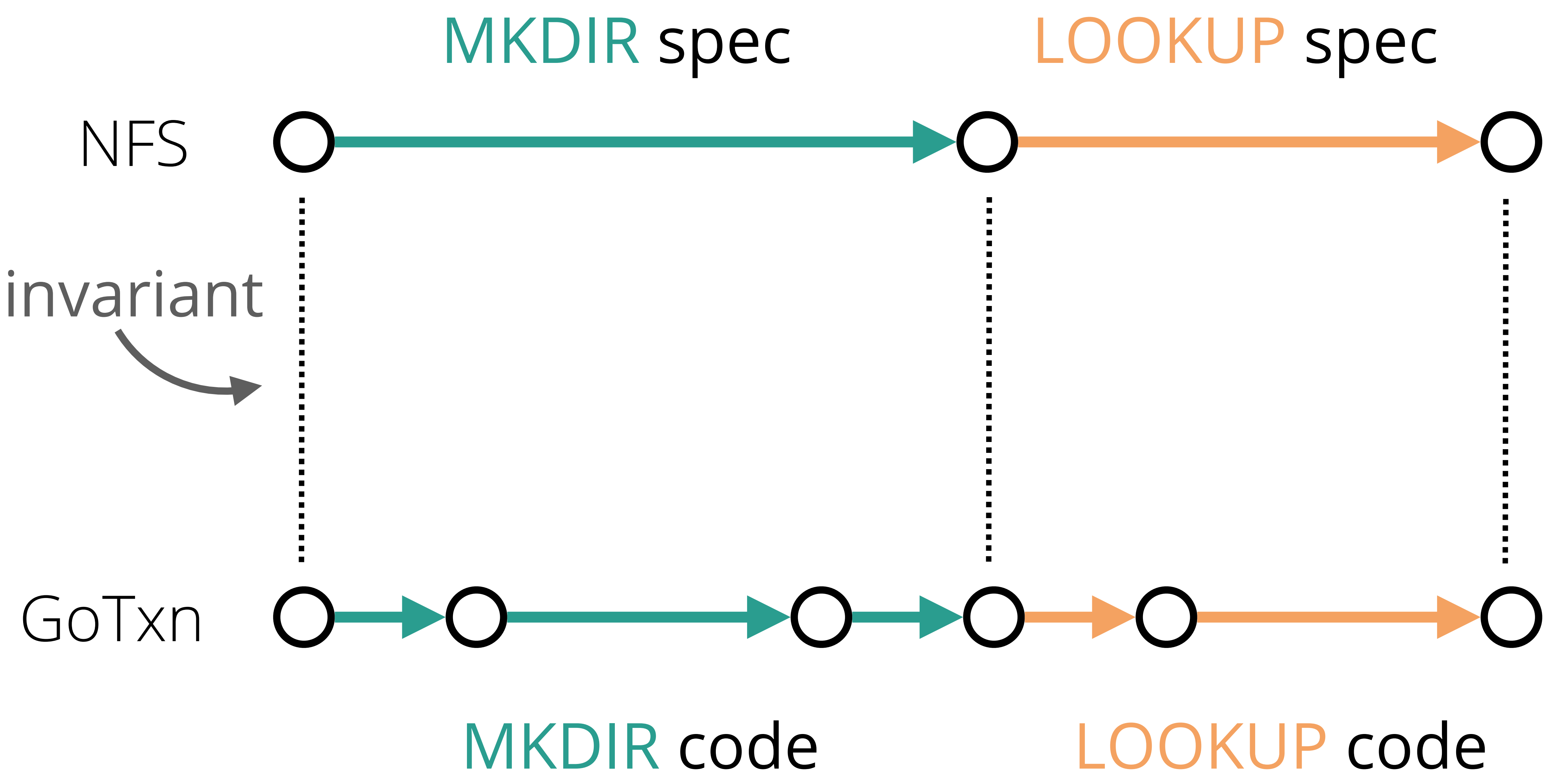
Proof: compose GoTxn and DaisyNFS proofs



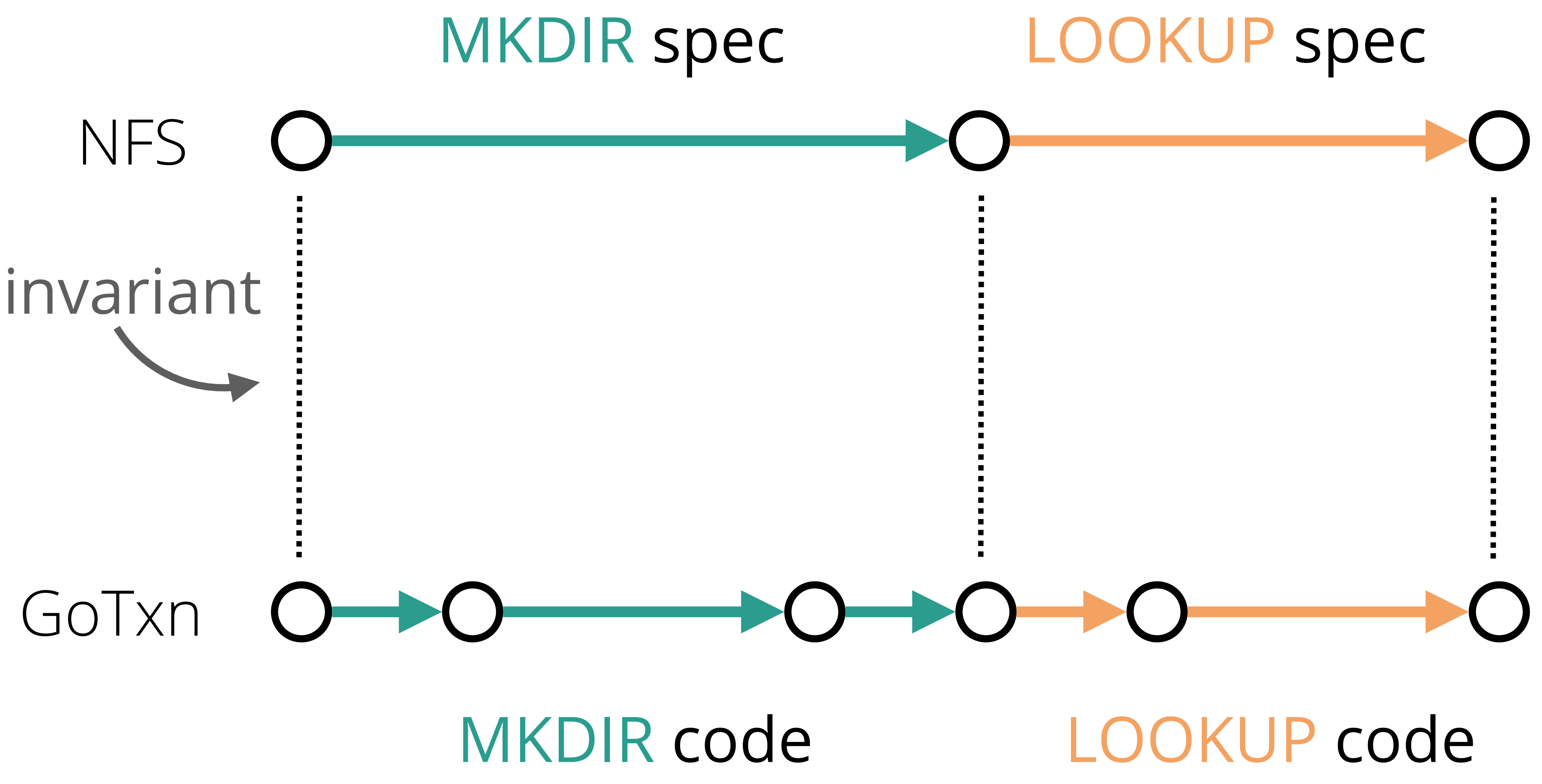
Transactions are proven with sequential reasoning



Transactions are proven with sequential reasoning



Transactions are proven with sequential reasoning



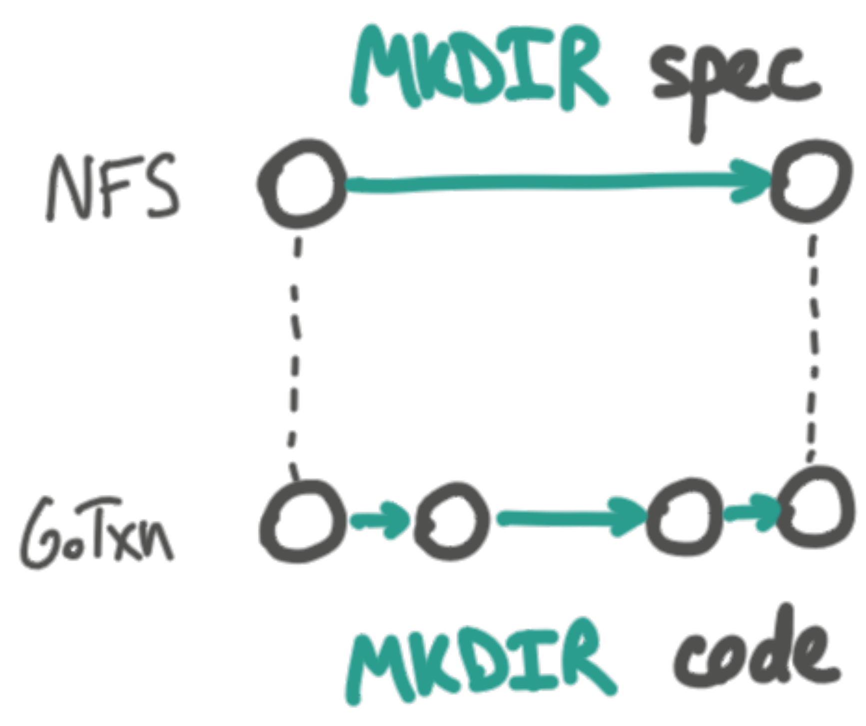
Sequential reasoning is highly automated

Future work: verify entirely in Iris

Can we use logical atomicity (with crashes) for transaction spec?

Can we do the proof with low overhead in Iris?

Summary



Sequential reasoning for concurrent system

Formalized RFC 1813

Fit operations into fixed-size transactions

Implementation: code and verification

DaisyNFS

Dafny 3,500 lines

GoTxn

GO 1,600 lines

Implementation: code and verification

DaisyNFS

Dafny 3,500 lines

GoTxn

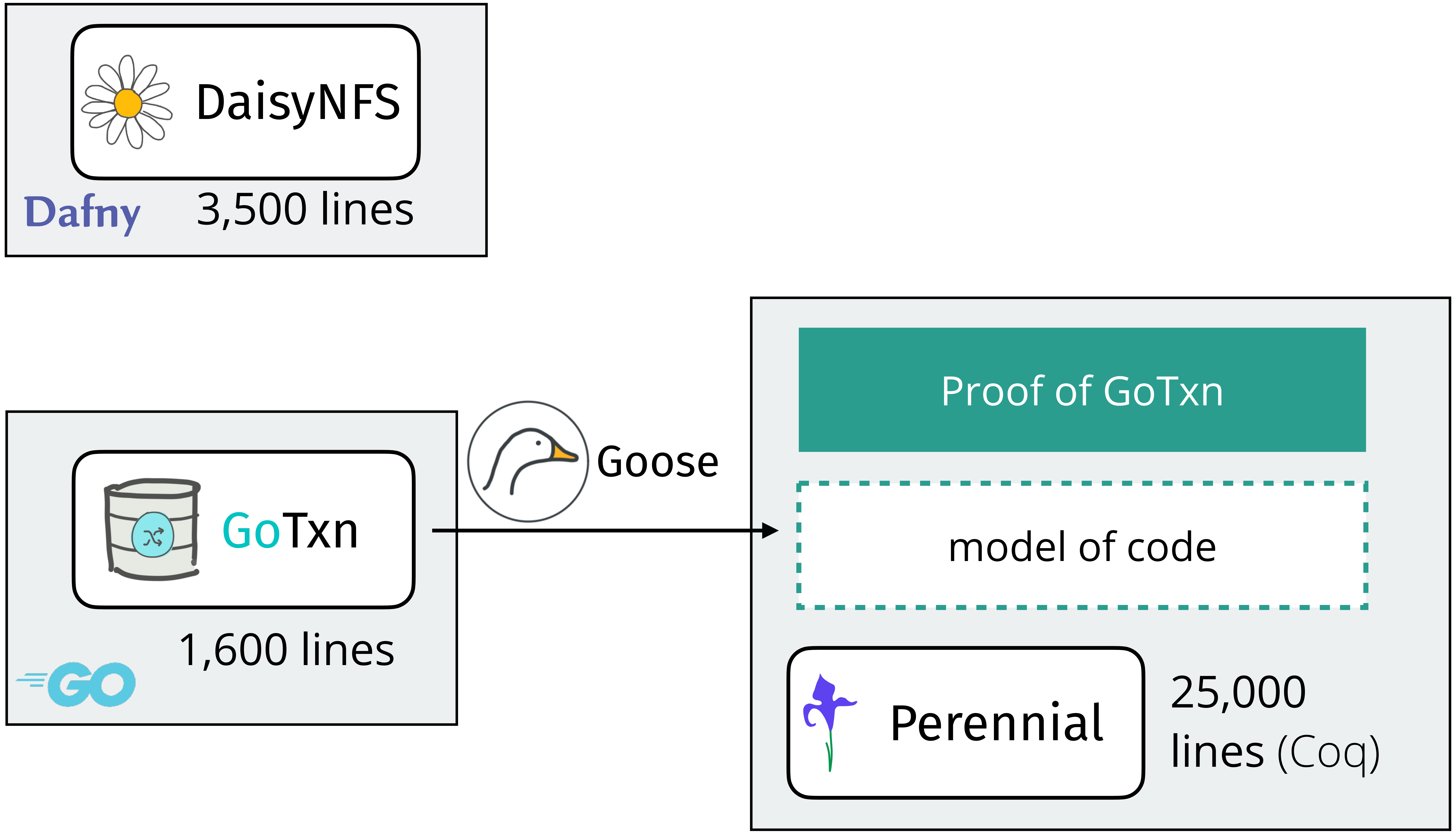
GO 1,600 lines

Proof of GoTxn

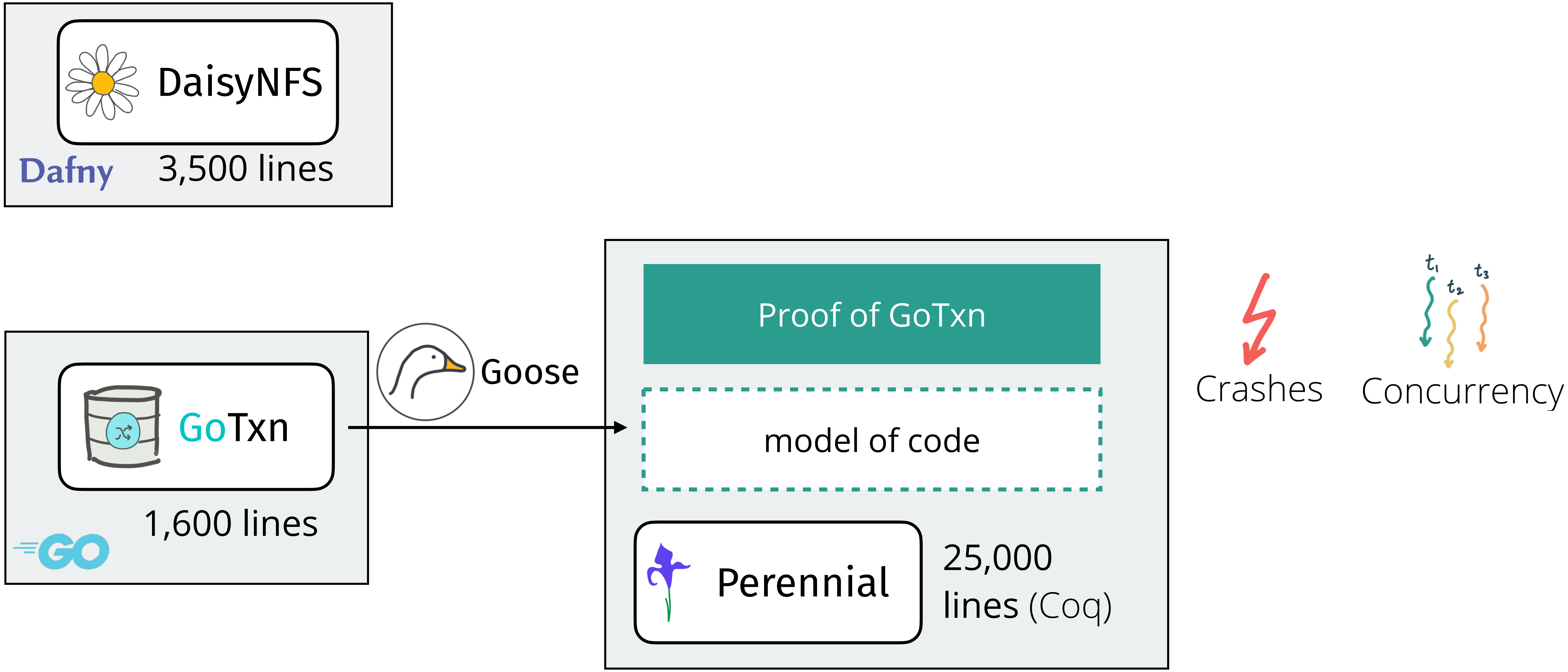
Perennial

25,000 lines (Coq)

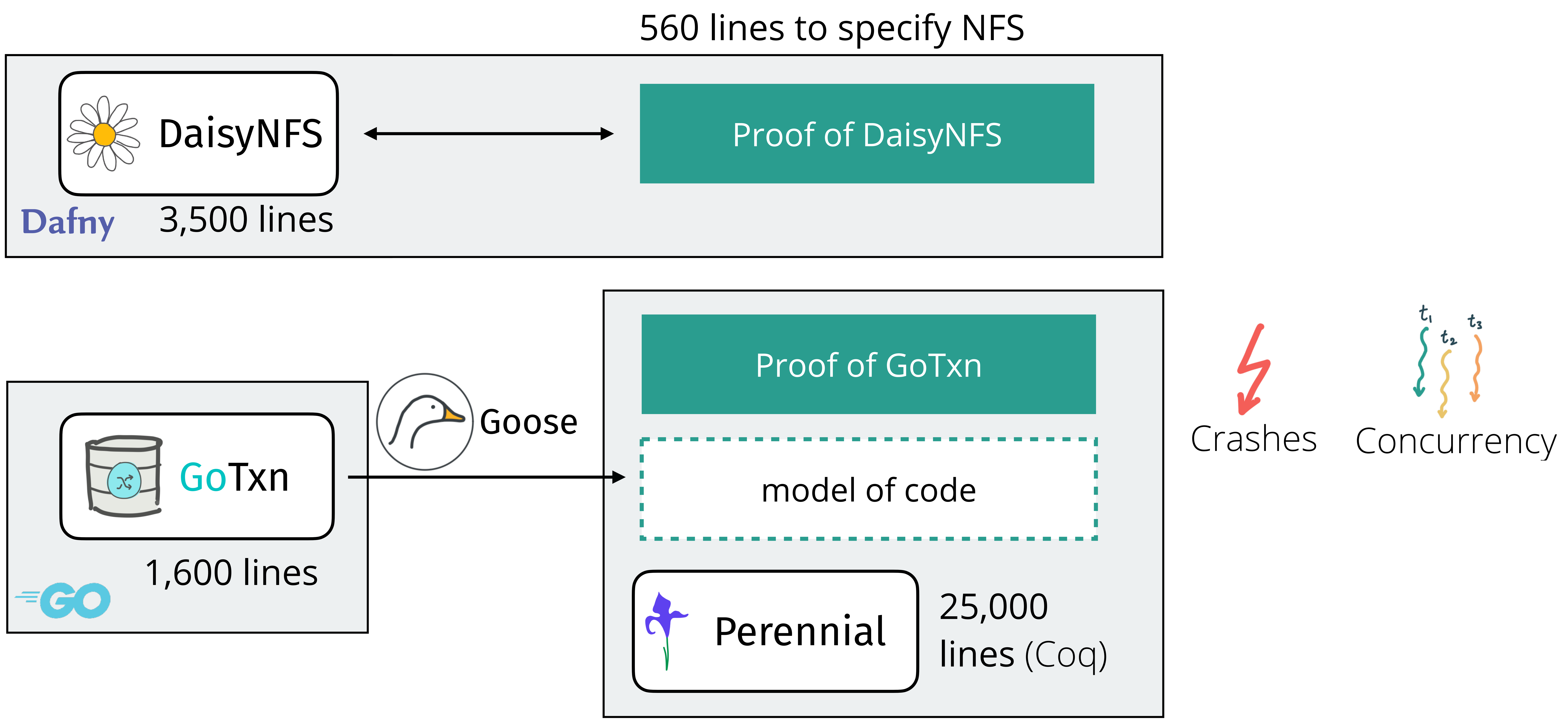
Implementation: code and verification



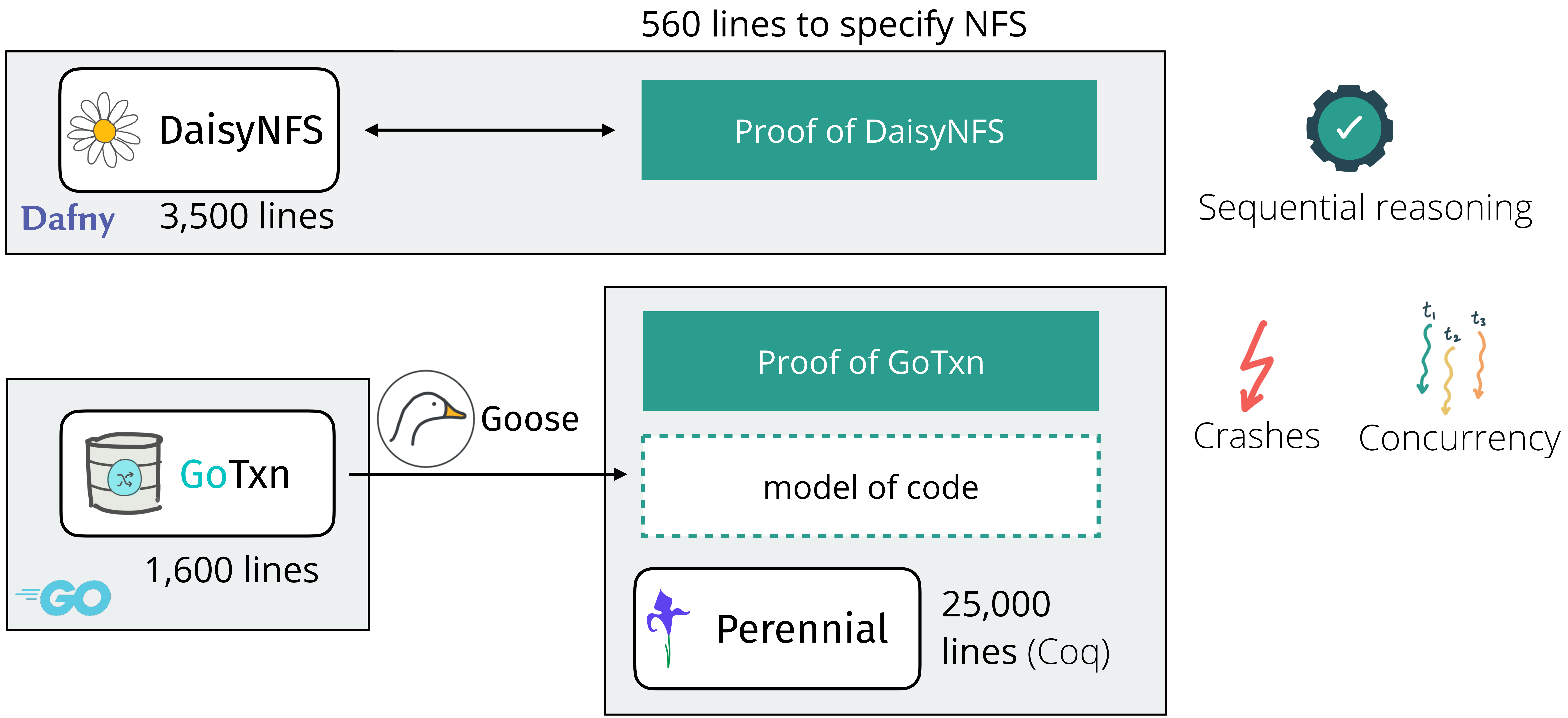
Implementation: code and verification



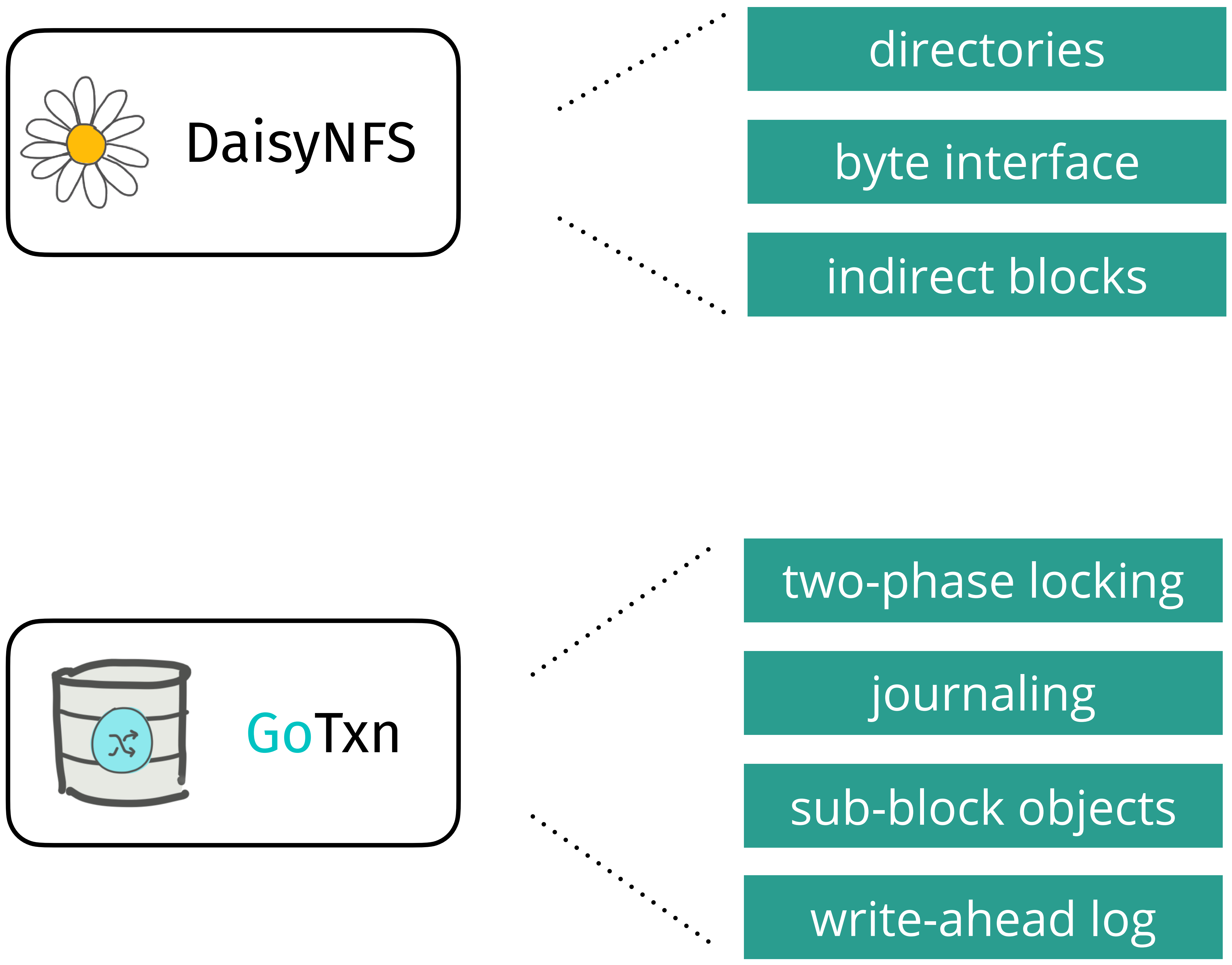
Implementation: code and verification



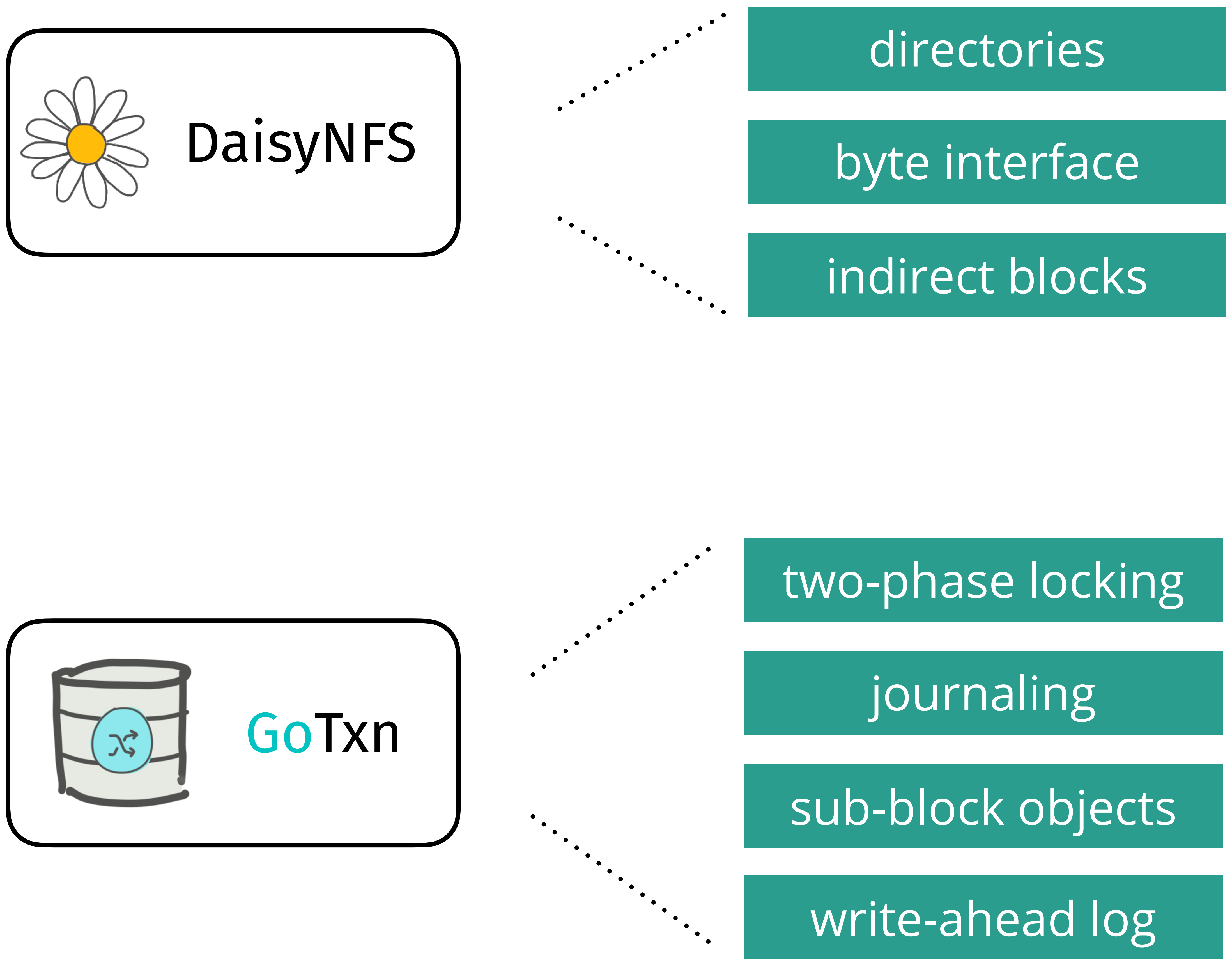
Implementation: code and verification



Implementation: code



Implementation: code



Limitations

No symbolic links
No access control
No paged READDIR

Implementation: code

DaisyNFS

- directories
- byte interface
- indirect blocks

Limitations

No symbolic links
No access control
No paged READDIR

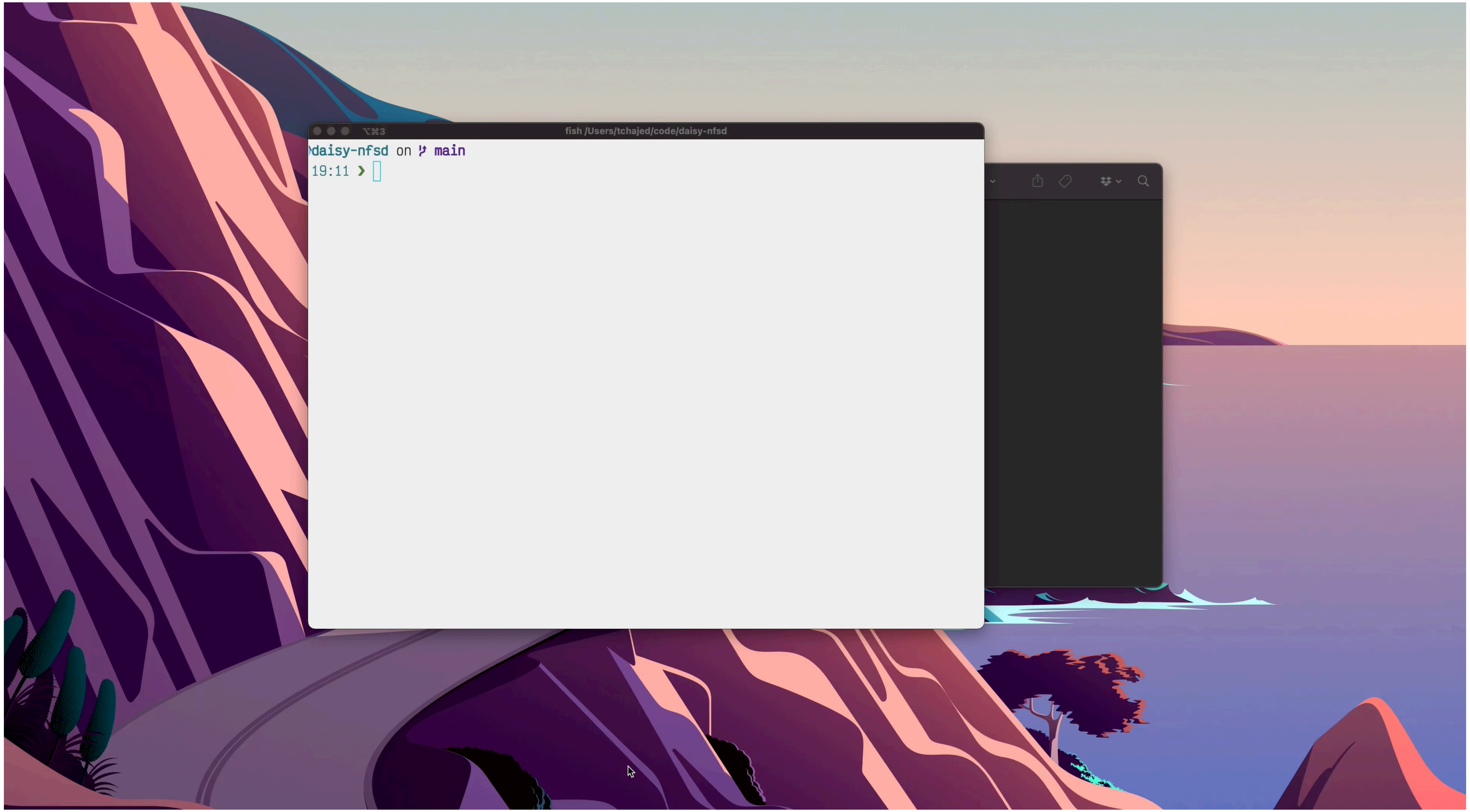
GoTxn

- two-phase locking
- journaling
- sub-block objects
- write-ahead log

Limitations

Synchronous commit
Assume disk is synchronous

DaisyNFS is a real file system




fish /Users/tchajed/code/daisy-nfsd
daisy-nfsd on main
19:11 ❯

Evaluation

Evaluation questions

Does GoTxn reduce the proof burden?

What is assumed in the DaisyNFS proof?

Does DaisyNFS get acceptable performance?

GoTxn greatly reduces proof overhead

		Code
	DaisyNFS	3,500
	GoTxn	1,600 (Go)

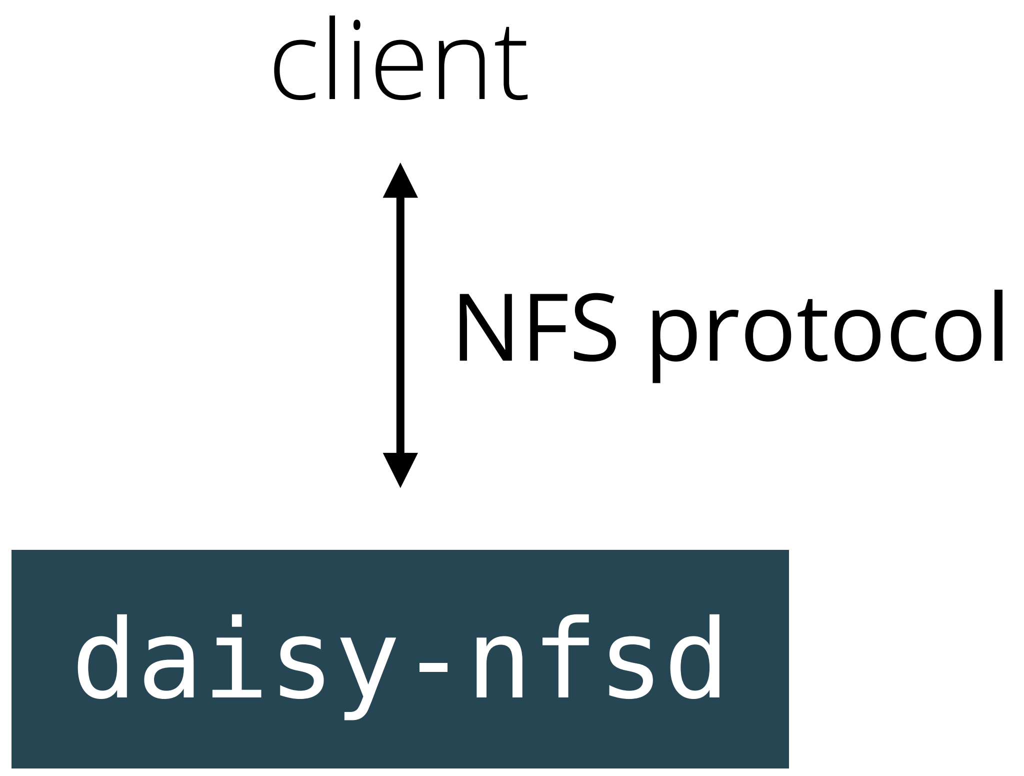
GoTxn greatly reduces proof overhead

		Code	Proof
	DaisyNFS	3,500	6,600
	GoTxn	1,600 (Go)	35,000 (Perennial)

GoTxn greatly reduces proof overhead

		Code	Proof	
	DaisyNFS	3,500	6,600	**2x proof:code**
	GoTxn	1,600 (Go)	35,000 (Perennial)	**20x proof:code**

Assumptions in the DaisyNFS proof



Assuming correctness of:

- Unverified glue code
- NFS specification state machine
- Tooling
- GoTxn specification in Dafny

Theorem: the server correctly implements the NFS protocol.

Bugs found in unverified code and spec

XDR decoder for strings can allocate 2^{32} bytes

File handle parser panics if wrong length

bytes

mic type cast

ed

RENAME can create circular directories

CREATE/MKDIR allow empty name

Proof assumes caller provides bounded inode

RENAME allows overwrite where spec does not

Didn’t find bugs in verified parts

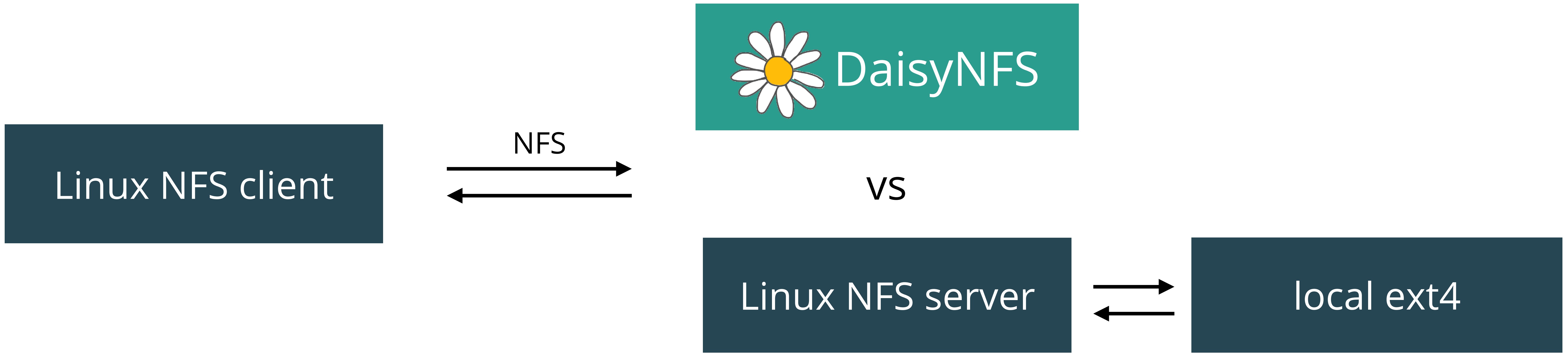
Bugs found in unverified code and spec

Bug	Category
XDR decoder for strings can allocate 2^{32} bytes	Unverified glue code
File handle parser panics if wrong length	Unverified glue code
Panic on unexpected enum value	Unverified glue code
WRITE panics if not enough input bytes	Unverified glue code
Directory REMOVE panics in dynamic type cast	Unverified glue code
The names "`.`" and "`..`" are allowed	Missing from specification
RENAME can create circular directories	Missing from specification
CREATE/MKDIR allow empty name	Missing from specification
Proof assumes caller provides bounded inode	Missing from specification
RENAME allows overwrite where spec does not	Missing from specification

Bugs found in unverified code and spec

Bug	Category
XDR decoder for strings can allocate 2^{32} bytes	Unverified glue code
File handle parser panics if wrong length	Unverified glue code
Panic on unexpected enum value	Unverified glue code
WRITE panics if not enough input bytes	Unverified glue code
Directory REMOVE panics in dynamic type cast	Unverified glue code
The names "`.`" and "`..`" are allowed	Missing from specification
RENAME can create circular directories	Missing from specification
CREATE/MKDIR allow empty name	Missing from specification
Proof assumes caller provides bounded inode	Missing from specification
RENAME allows overwrite where spec does not	Missing from specification

Compare against Linux NFS server with ext4



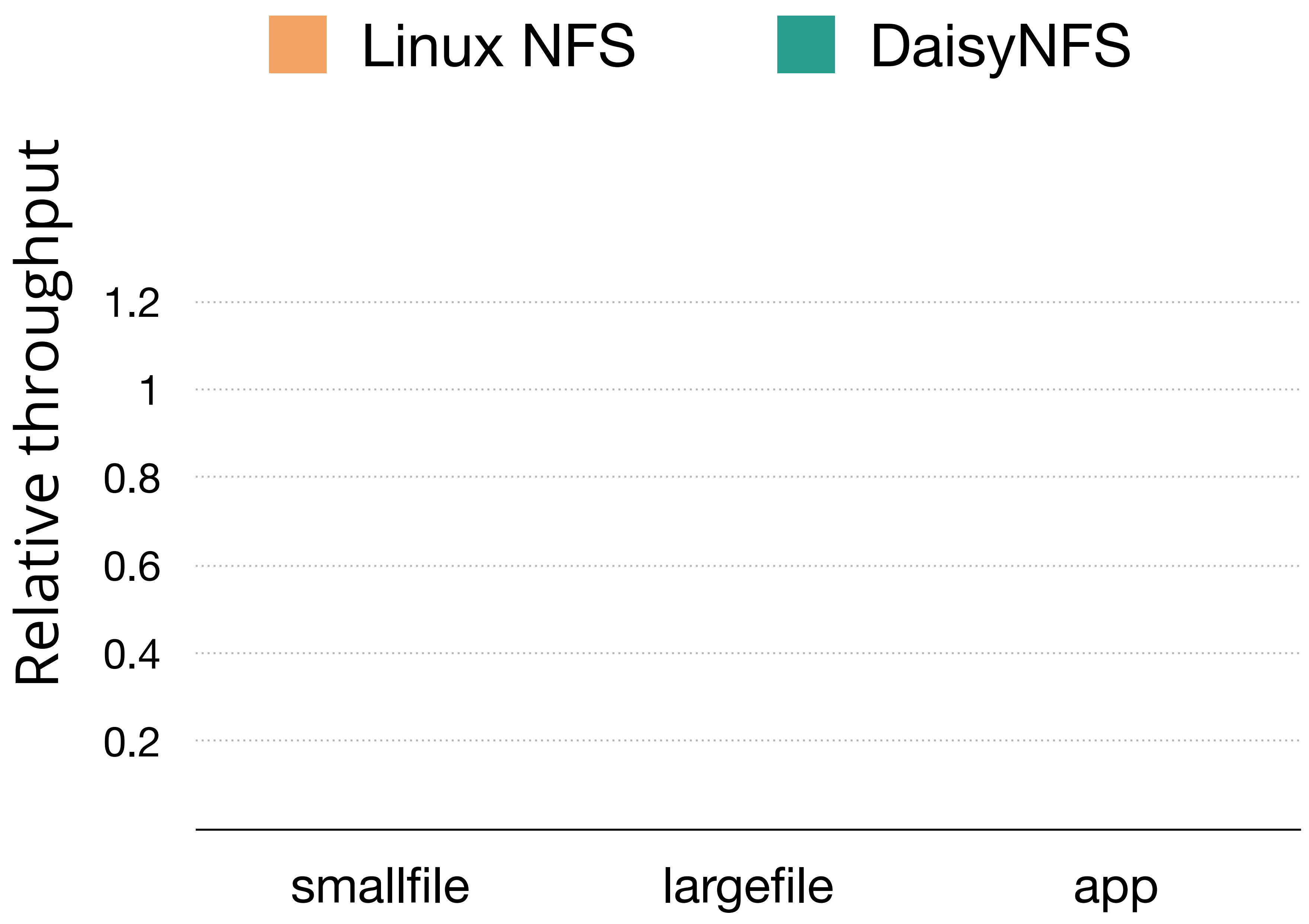
*using `data=journal`

Performance evaluation setup

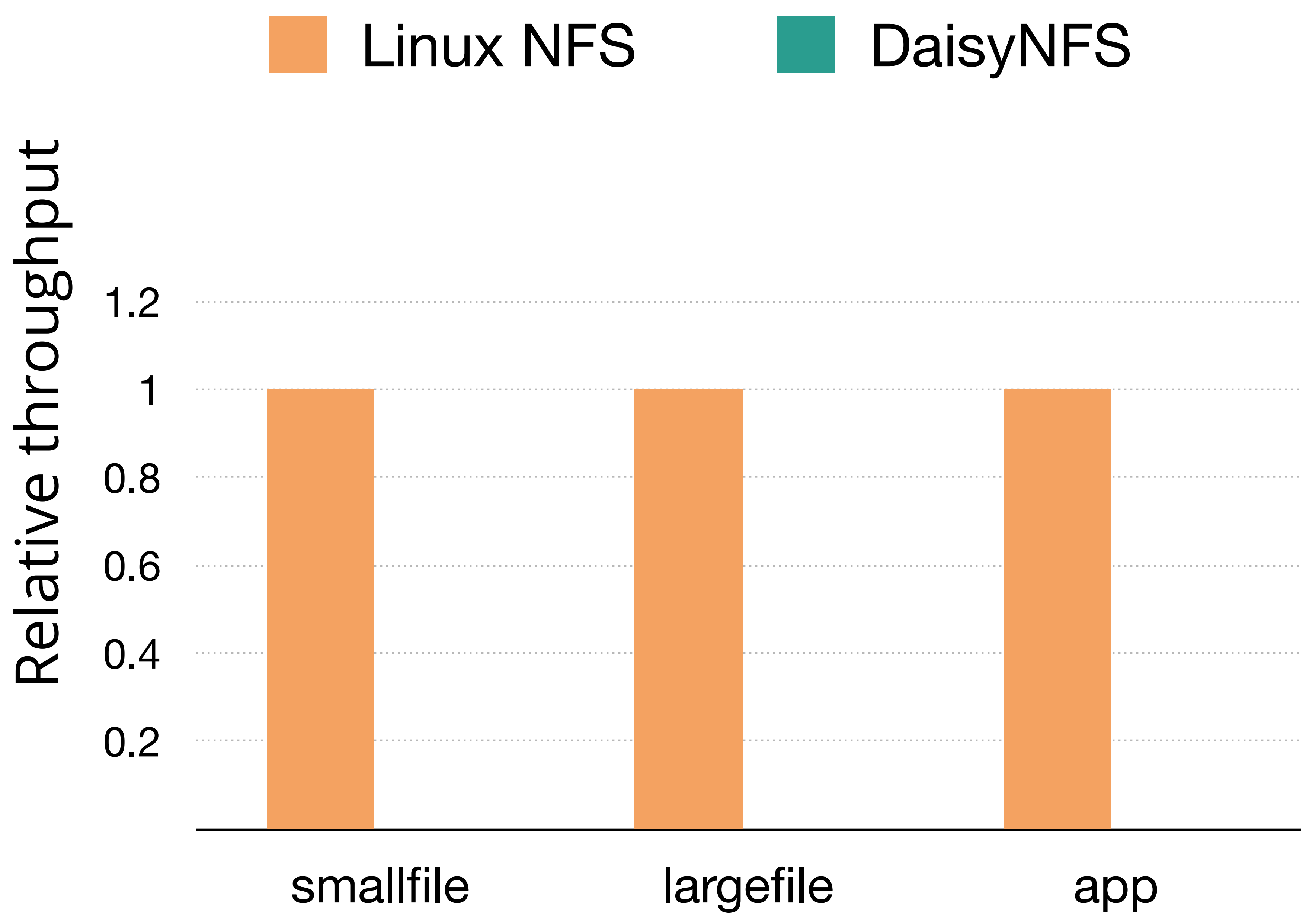
Hardware: i3.metal instance
36 cores at 2.3GHz

Benchmarks:

- smallfile: metadata heavy
- largefile: lots of data
- app: `git clone` + `make`

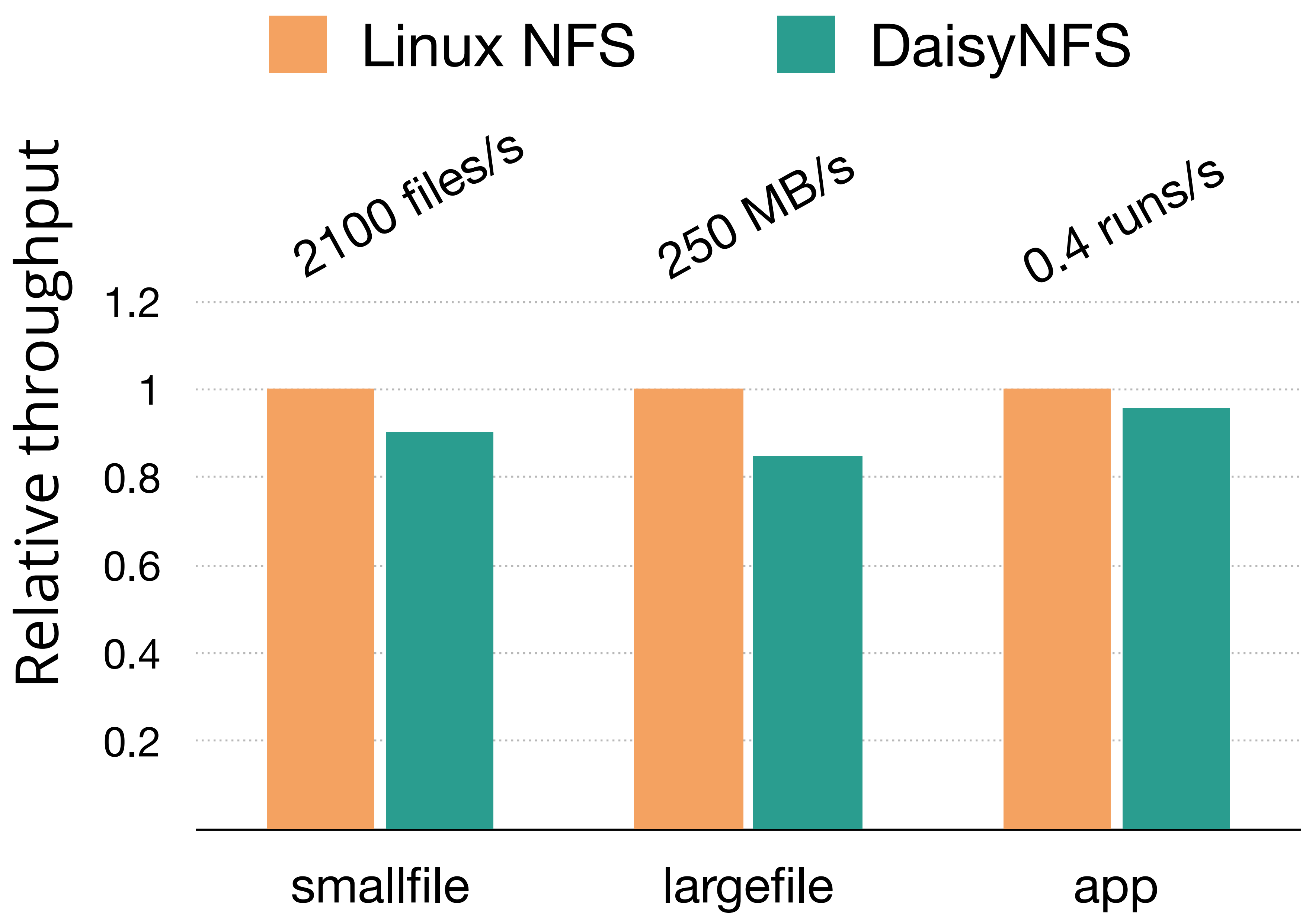


Compare DaisyNFS throughput to Linux, running on an in-memory disk

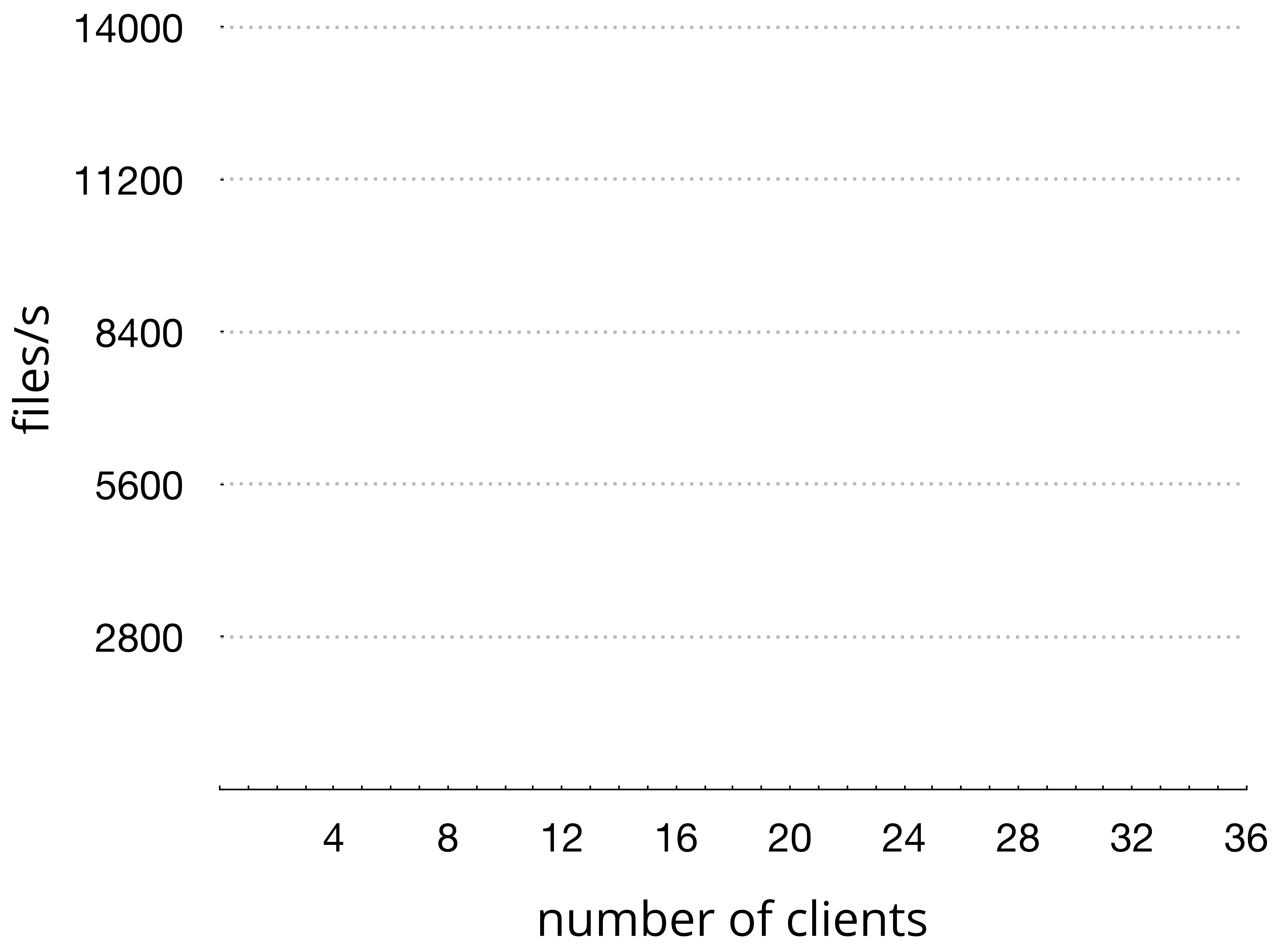


Compare DaisyNFS throughput to Linux, running on an in-memory disk

DaisyNFS gets good performance with a single client

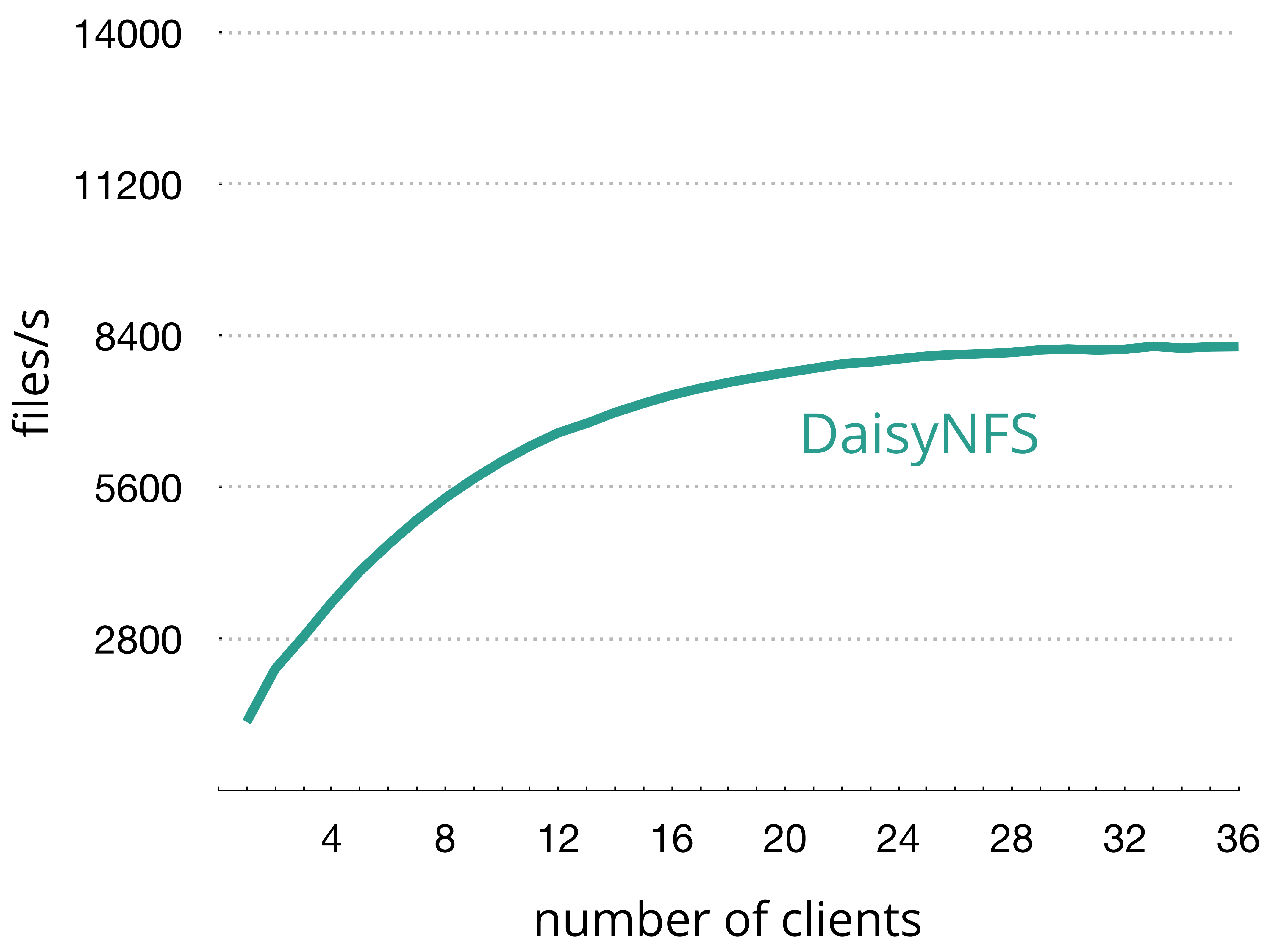


Compare DaisyNFS throughput to Linux, running on an in-memory disk



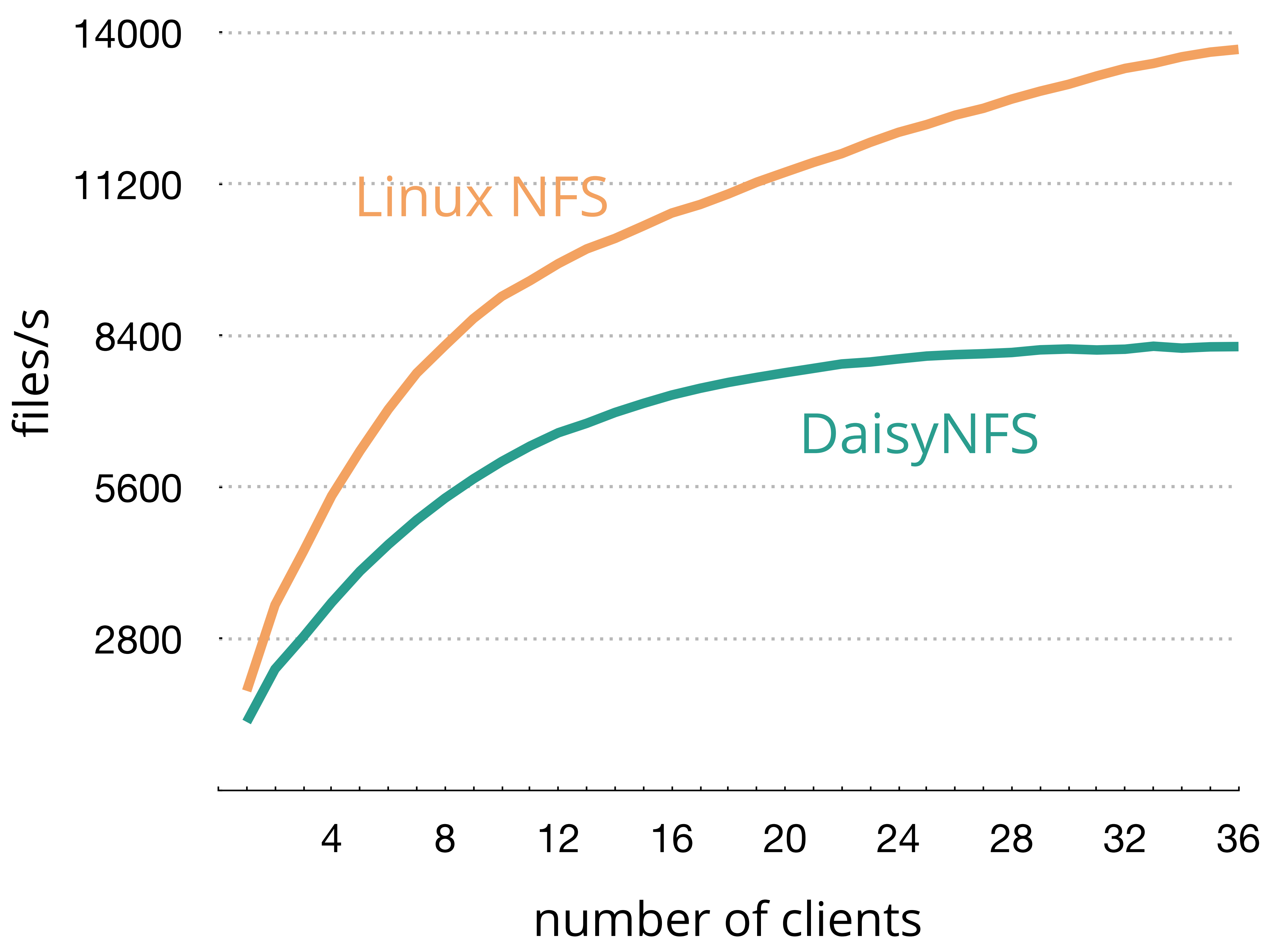
Run smallfile with many clients on an NVMe SSD

DaisyNFS can take advantage of multiple clients



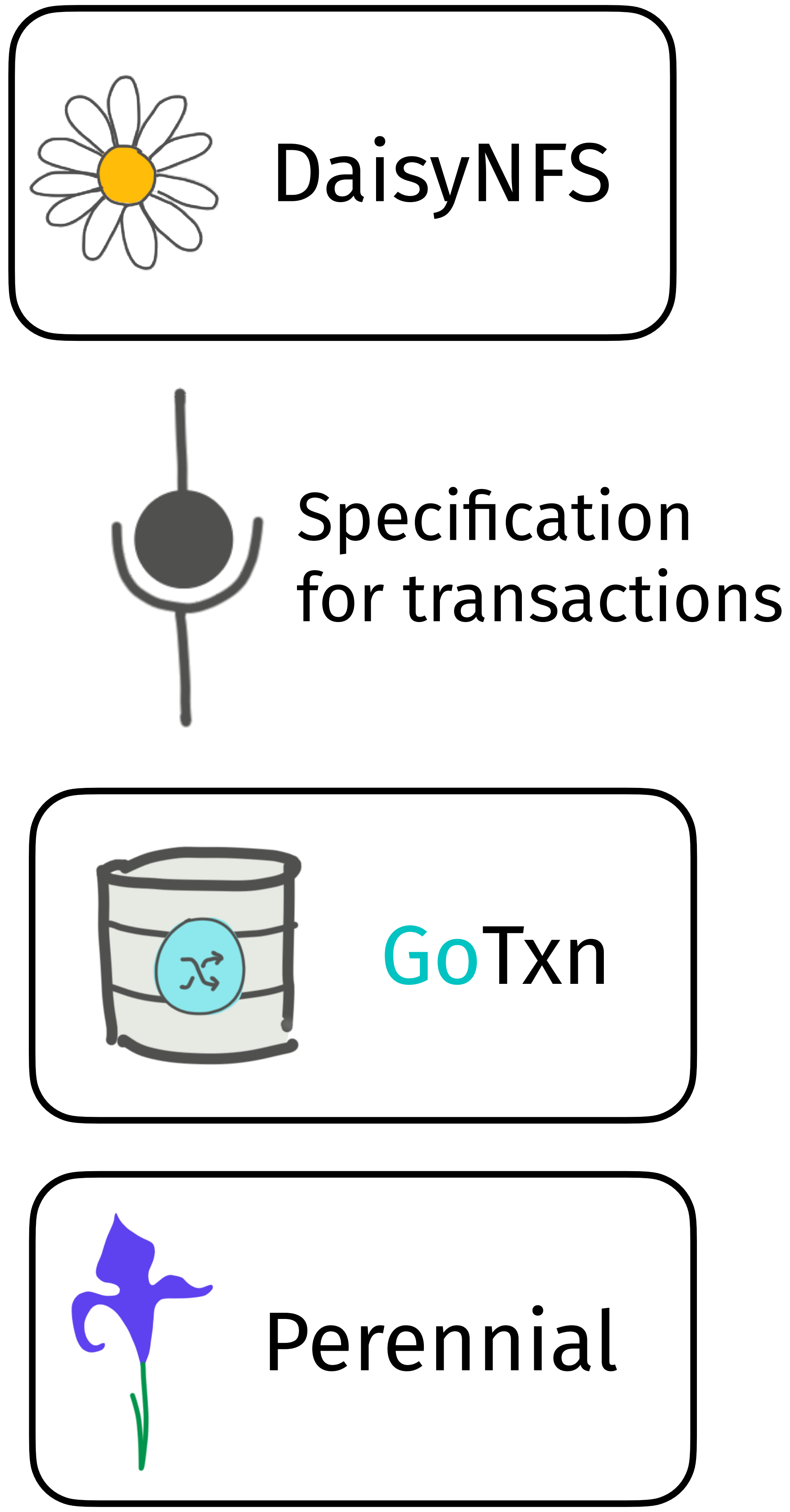
Run smallfile with many clients on an NVMe SSD

DaisyNFS can take advantage of multiple clients



Run smallfile with many clients on an NVMe SSD

DaisyNFS is a verified concurrent file system



Verified a file system combining automated and interactive proofs

Built on a program logic for crashes + concurrency

Tej Chajed
tchajed@gmail.com