Formal verification of a concurrent file system

Tej Chajed
MIT

Systems software is challenging to get right

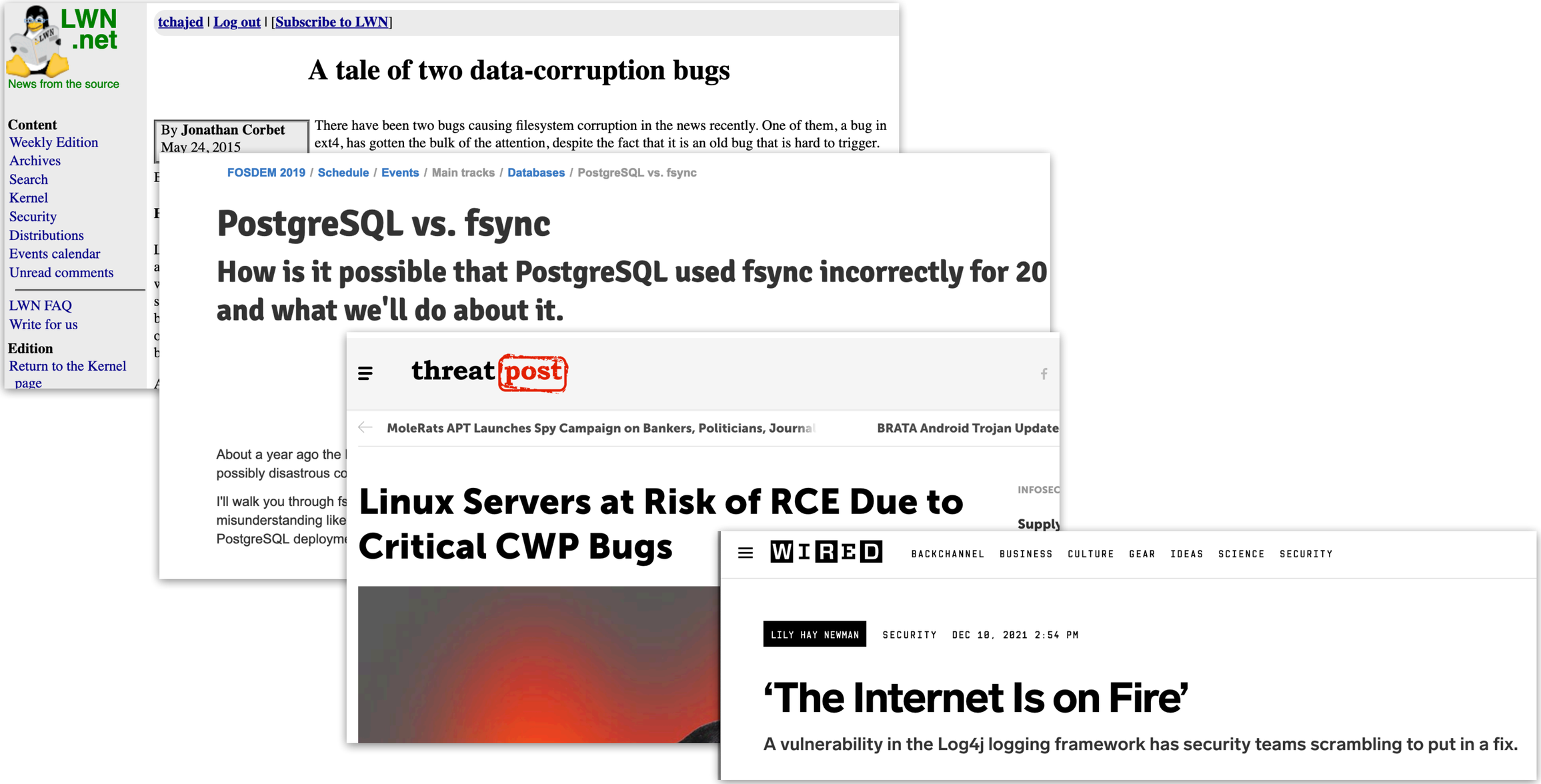
Systems software is challenging to get right

Applications exercise all corners of the system API

Runs on raw hardware: crashes, concurrency, devices

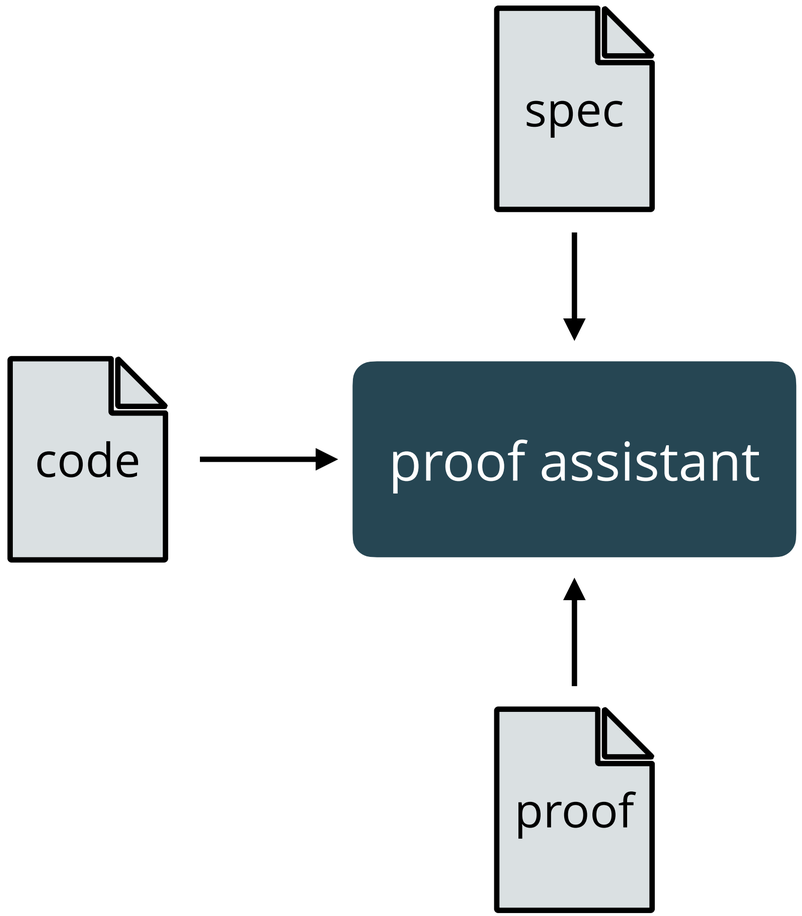
Systems software continues to have bugs

Systems software continues to have bugs

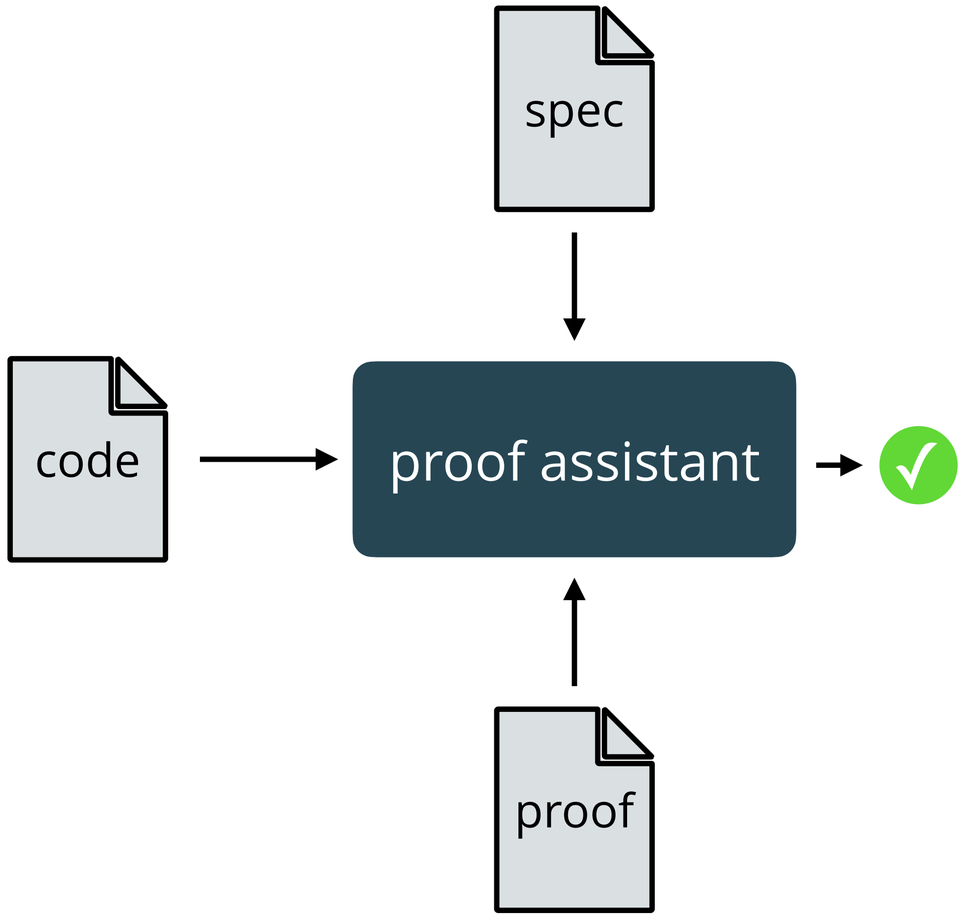


Vision: reliable systems software using verification

Approach: systems verification



Approach: systems verification



Verification can eliminate whole classes of bugs

Memory safety

Concurrency bugs

Logic errors

Security flaws

Systems verification is becoming feasible

Microkernels (seL4, CertiKOS)

Cryptography libraries (Fiat Crypto, HACL*)

Distributed systems (Ironfleet, Verdi)

File systems (FSCQ)

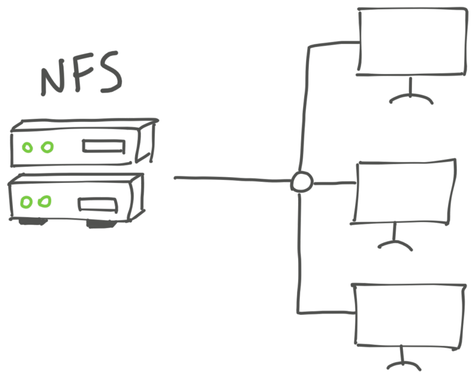
My research spans from foundations to systems

systems	OSDI ‘22	**DaisyNFS**	concurrent, crash-safe file system
	OSDI ‘21	**GoTxn**	high-performance transaction system
	SOSP ’15, SOSP ‘17	**FSCQ**	verified sequential file system
foundations	CoqPL ‘20	**Goose**	reasoning about Go code
	PLDI ‘19	**Argosy**	layered recovery procedures
	OSDI ‘18	**CSPEC**	concurrency using movers
	SOSP ’19, OSDI ‘21	**Perennial**	concurrency + crash safety
	SOSP ’15	**Crash Hoare Logic (CHL)**	sequential crash safety

My research spans from foundations to systems

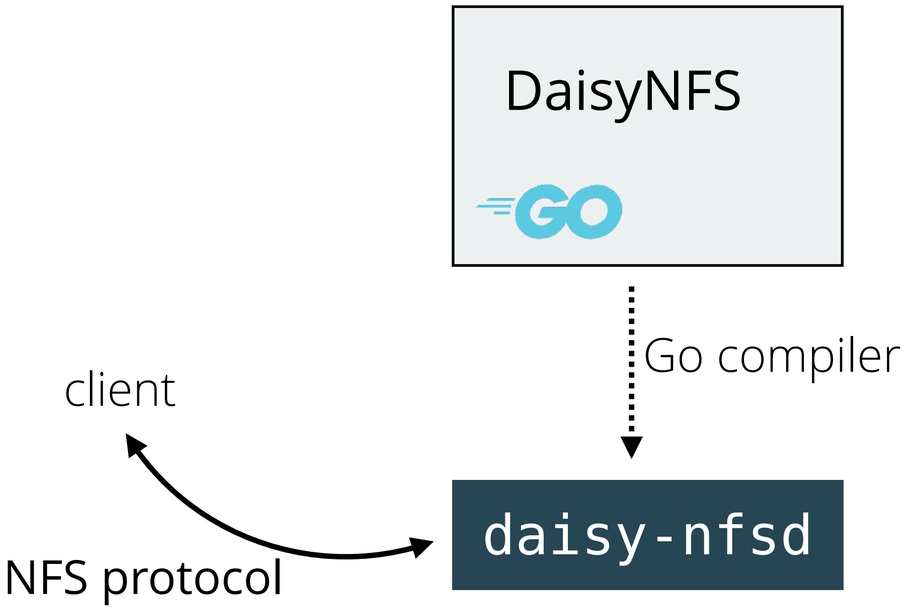
systems	OSDI ‘22	**DaisyNFS**	concurrent, crash-safe file system
	OSDI ‘21	**GoTxn**	high-performance transaction system
	SOSP ’15, SOSP ‘17	**FSCQ**	verified sequential file system
foundations	CoqPL ‘20	**Goose**	reasoning about Go code
	PLDI ‘19	**Argosy**	layered recovery procedures
	OSDI ‘18	**CSPEC**	concurrency using movers
	SOSP ’19, OSDI ‘21	**Perennial**	concurrency + crash safety
	SOSP ’15	**Crash Hoare Logic (CHL)**	sequential crash safety

NFS is a good target for verification

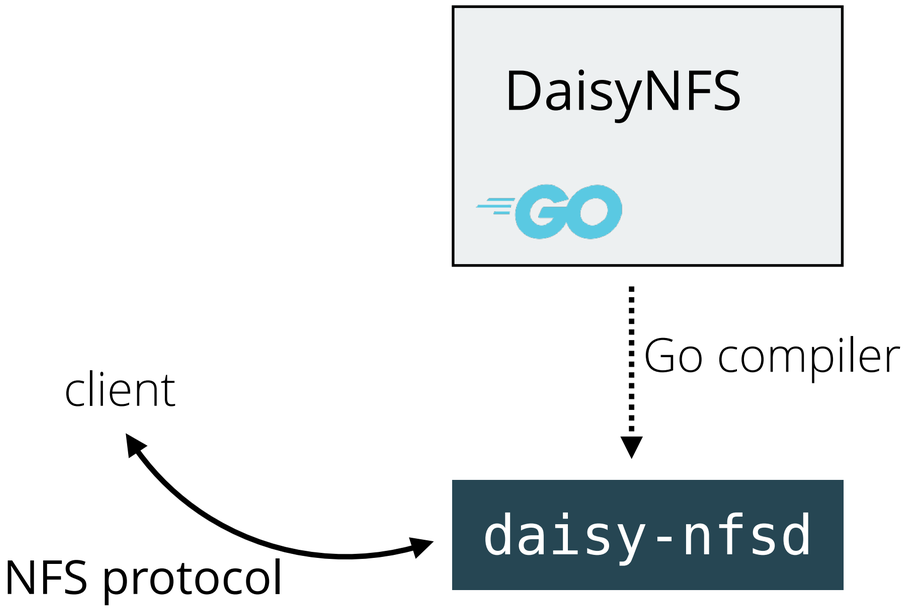


1. **Widely used**
2. **Sophisticated implementations** with concurrency & high performance
3. **Bugs are costly**, especially data loss

DaisyNFS implements an NFS server



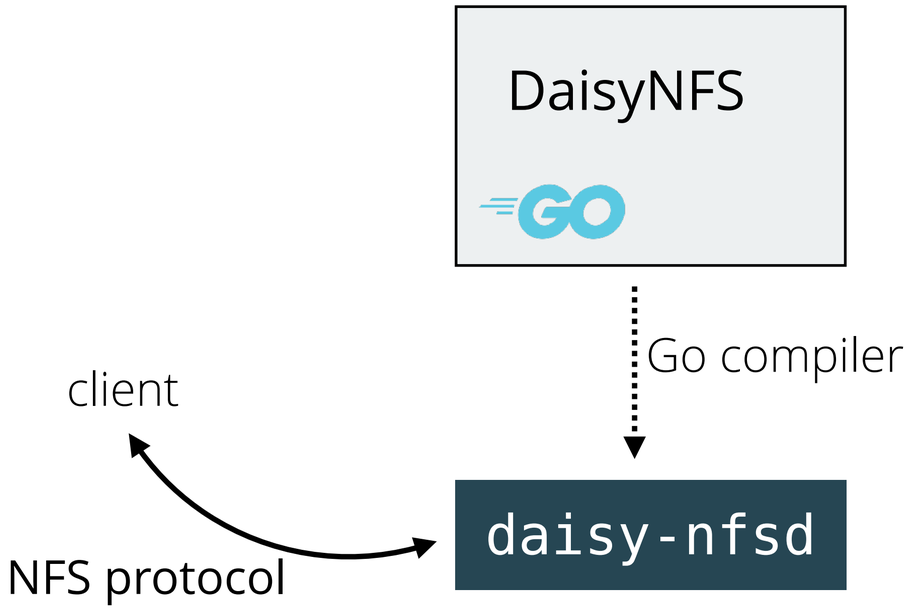
DaisyNFS implements an NFS server



What is Go?

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

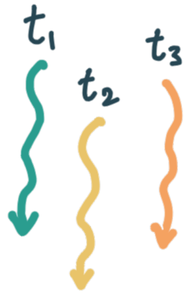
DaisyNFS is a verified NFS server



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

Challenges in verifying a file system

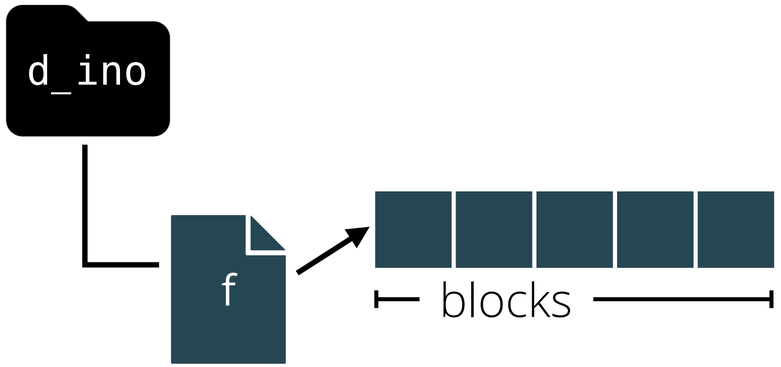
Crashes



Concurrency

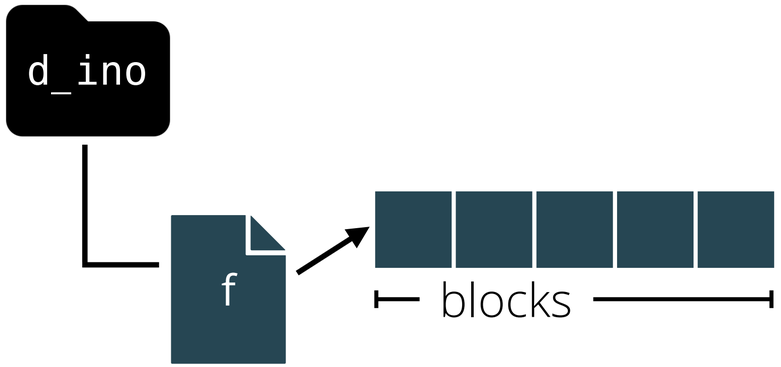
REMOVE has several steps

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

  f := unlink(d_ino, name)

  blocks := getBlocks(f)

  free(blocks)

}
```



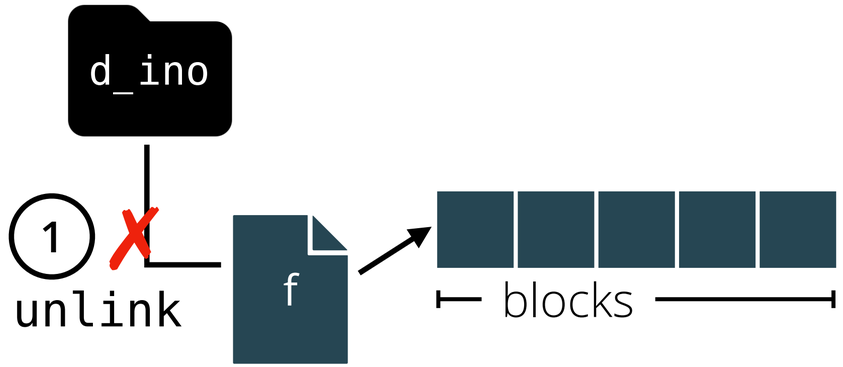
REMOVE has several steps

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

1 f := unlink(d_ino, name)

2 blocks := getBlocks(f)

3 free(blocks)

}
```



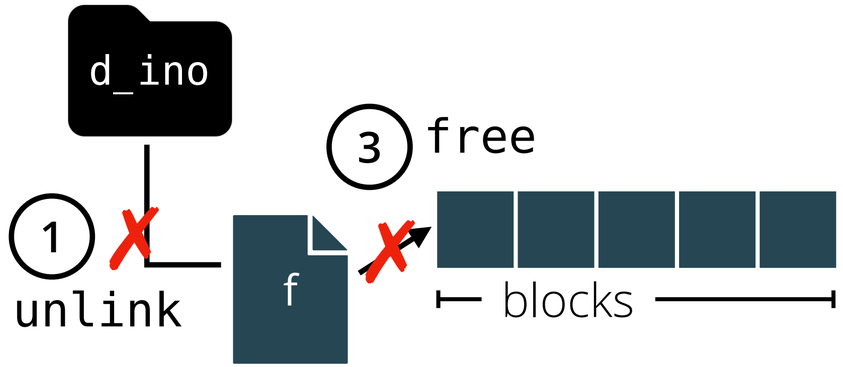
REMOVE has several steps

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

1 f := unlink(d_ino, name)

2 blocks := getBlocks(f)

3 free(blocks)

}
```



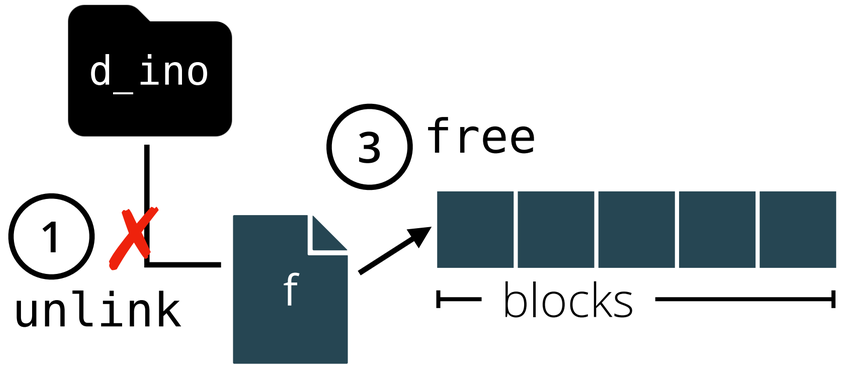
REMOVE has several steps

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

1 f := unlink(d_ino, name)

2 blocks := getBlocks(f)

3 free(blocks)

}
```



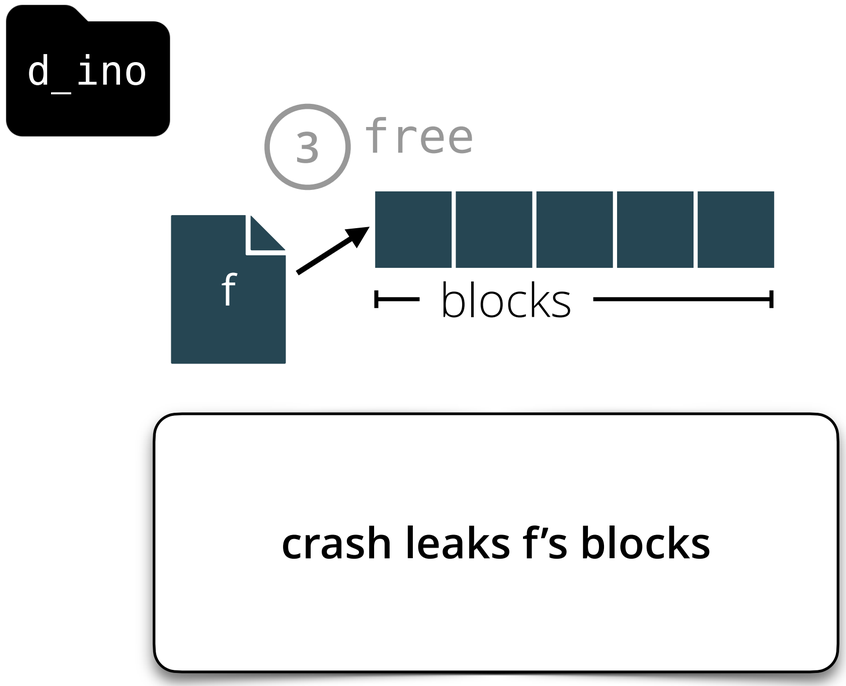
Crashes create subtle bugs

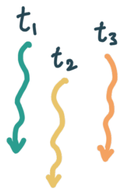
```
func REMOVE(d_ino: uint64,
            name: []byte) {

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

3 free(blocks)

}
```



Crashes create subtle bugs

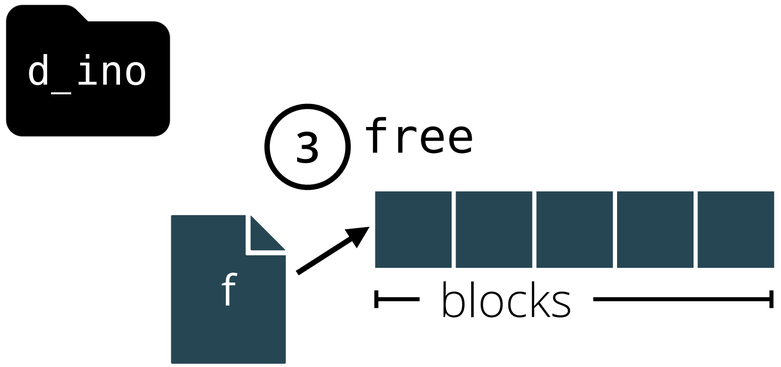
```
func REMOVE(d_ino: uint64,
            name: []byte) {

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

3 free(blocks)

}
```

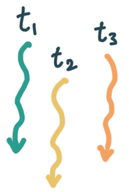




Concurrency also creates subtle bugs

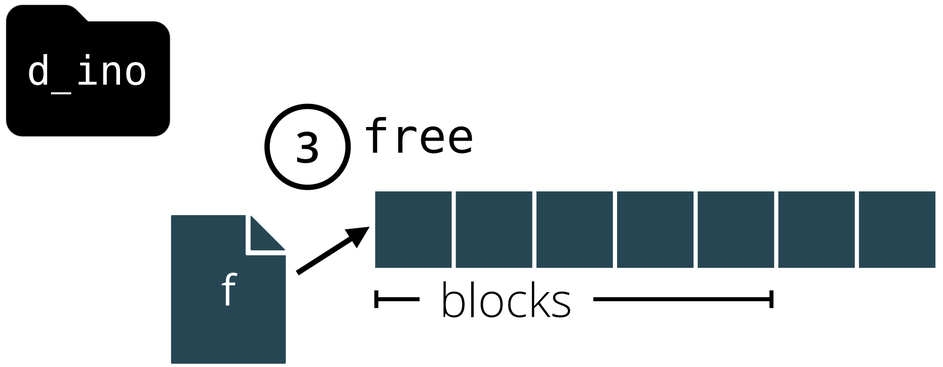
```
func REMOVE(d_ino: uint64,
            name: []byte) {

1 f := unlink(d_ino, name)

2 blocks := getBlocks(f)

3 free(blocks)

}
```

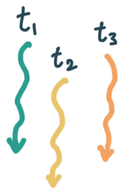




Concurrency also creates subtle bugs

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

1 f := unlink(d_ino, name)

2 blocks := getBlocks(f)

                                  concurrent
                                  append
3 free(blocks)

}
```

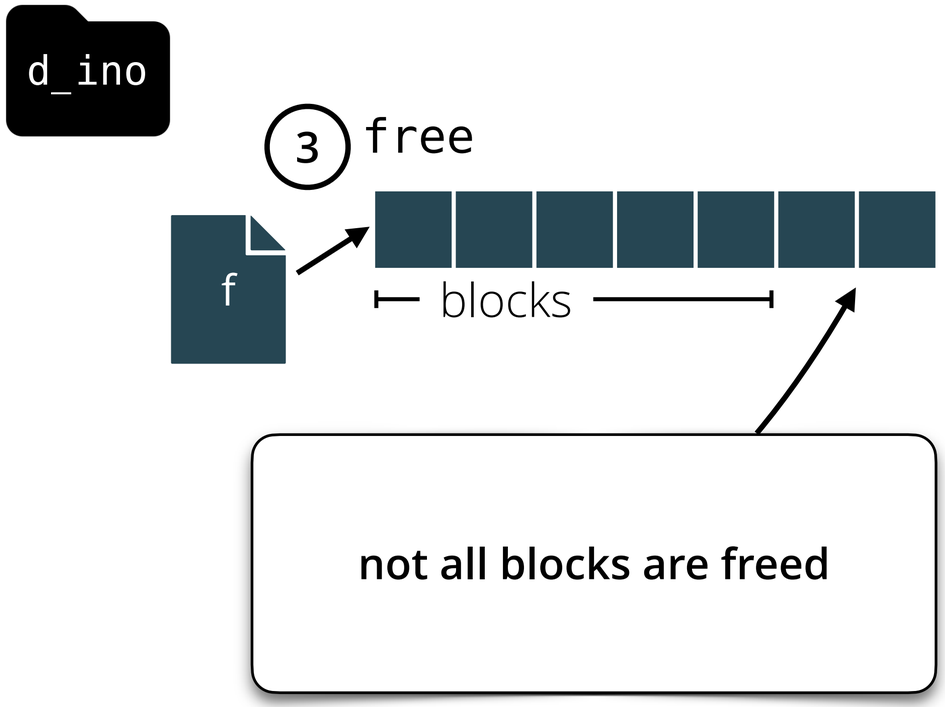




Concurrency also creates subtle bugs

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

1 f := unlink(d_ino, name)

2 blocks := getBlocks(f)

3 free(blocks)

}
```

concurrent append



Crashes and concurrency bugs can be severe

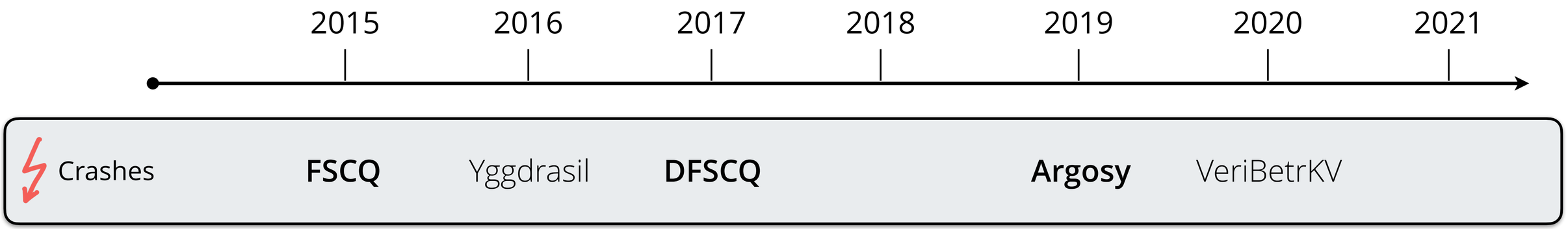
Might leak resources

Might return the wrong user’s data

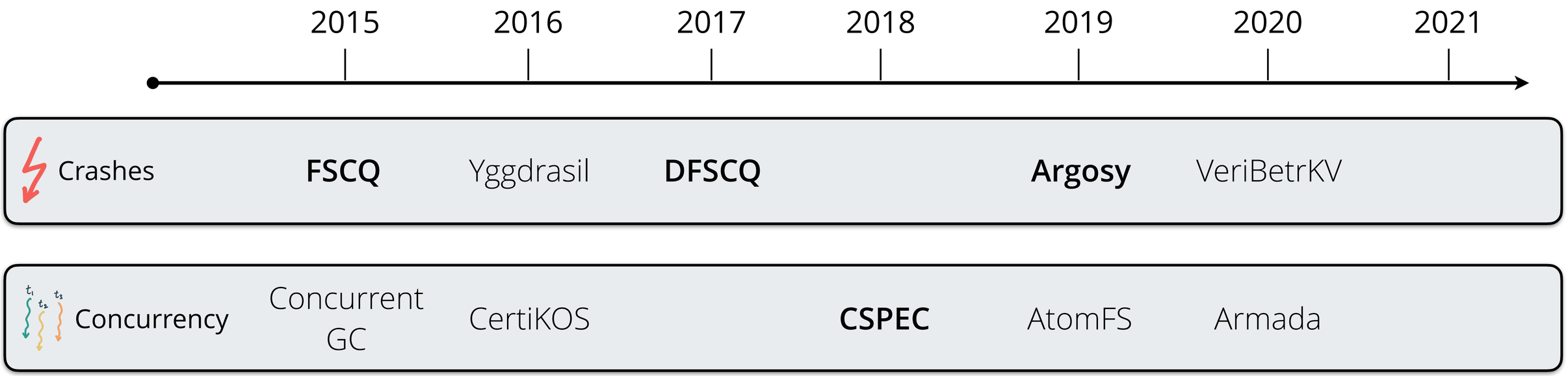
Might lose user data

My thesis is the first to verify crashes & concurrency

My thesis is the first to verify crashes & concurrency



My thesis is the first to verify crashes & concurrency



My thesis is the first to verify crashes & concurrency

Crashes — **FSCQ** — Yggdrasil — **DFSCQ** — **Argosy** — VeriBetrKV

Concurrency — Concurrent GC — CertiKOS — **CSPEC** — AtomFS — Armada

\+ — **Perennial** — **GoJournal** **DaisyNFS** Flashix

Approach: verification-friendly system design

DaisyNFS

File-system code implemented with transactions

GoTxn

Transaction system gives atomicity

Approach: verification-friendly system design

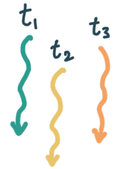
DaisyNFS

File-system code implemented with transactions

GoTxn

Transaction system gives atomicity

Crashes



Concurrency

Approach: verification-friendly system design

DaisyNFS

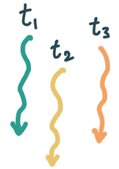
File-system code implemented with transactions

Sequential reasoning

GoTxn

Transaction system gives atomicity

Crashes



Concurrency

Approach: verification-friendly system design

DaisyNFS

File-system code implemented with transactions

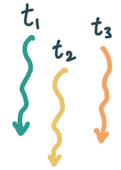
Sequential reasoning

Specification for transactions bridges the two

GoTxn

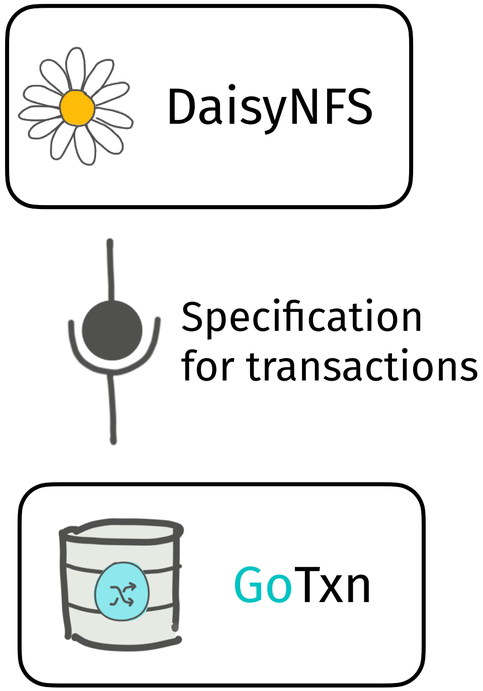
Transaction system gives atomicity

Crashes

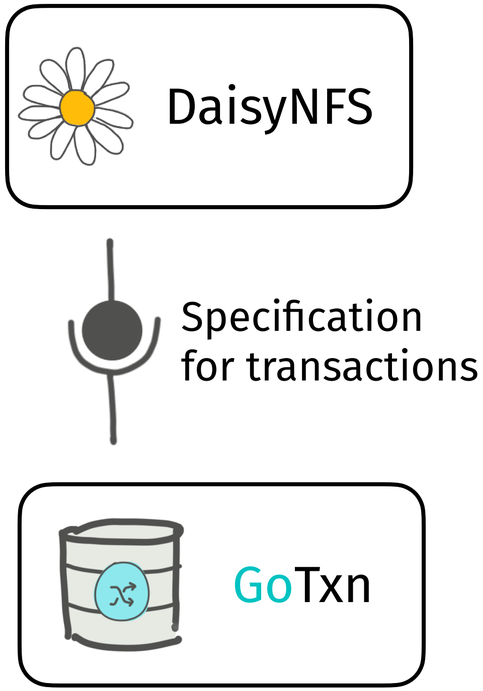


Concurrency

Contributions

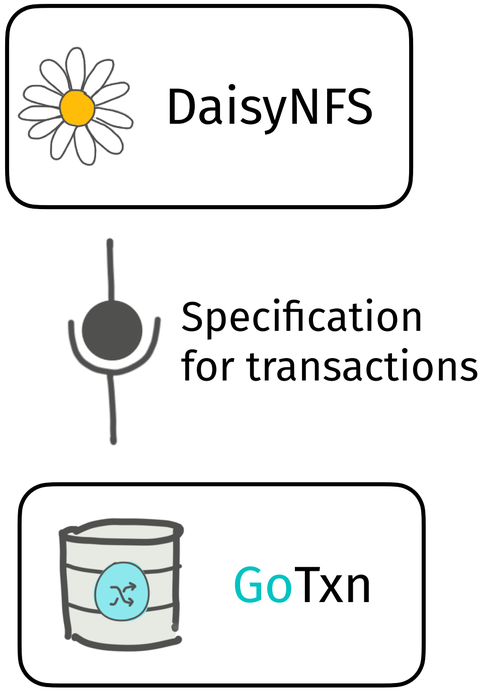


Contributions



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

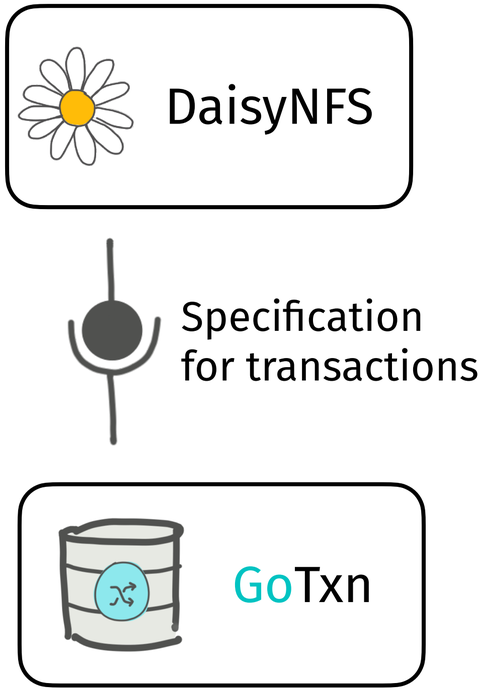
Contributions



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

Lifting specification for concurrent transactions

Contributions

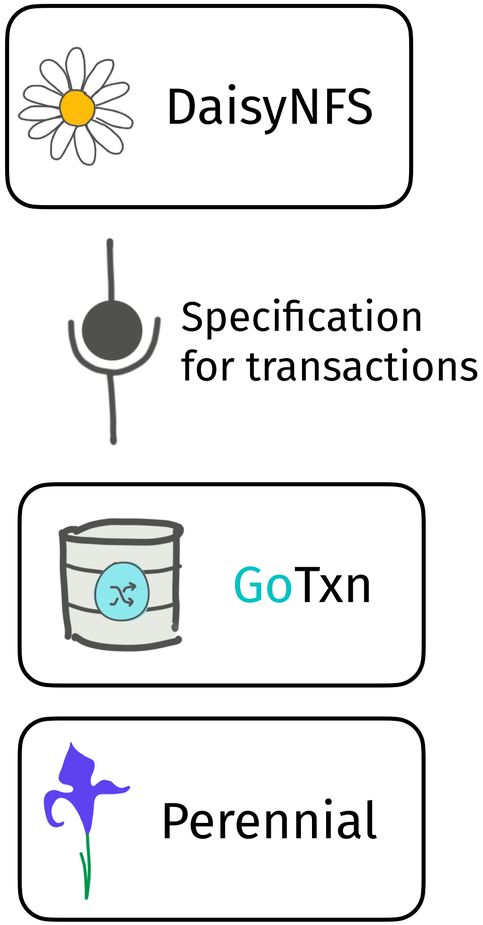


Reduce proof effort with **sequential reasoning for a concurrent system**

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

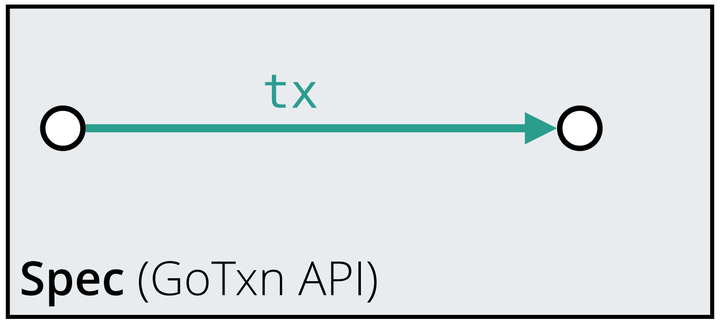
```
func             Begin() *Txn

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**

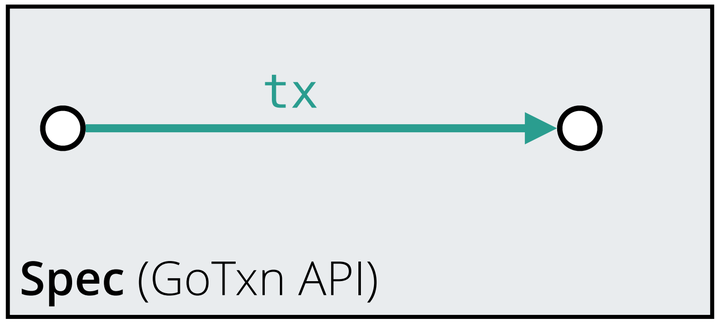
Specifying a transaction system

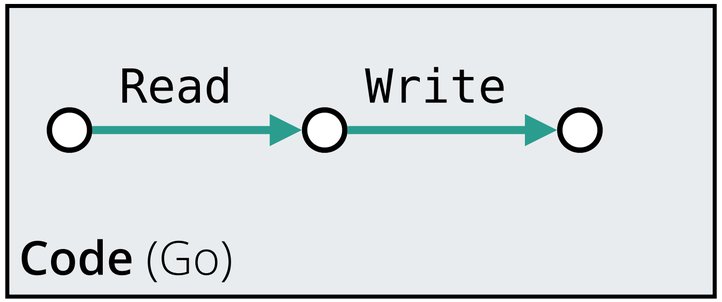
```
tx
v := tx.Read(0)
tx.Write(1, v)
```



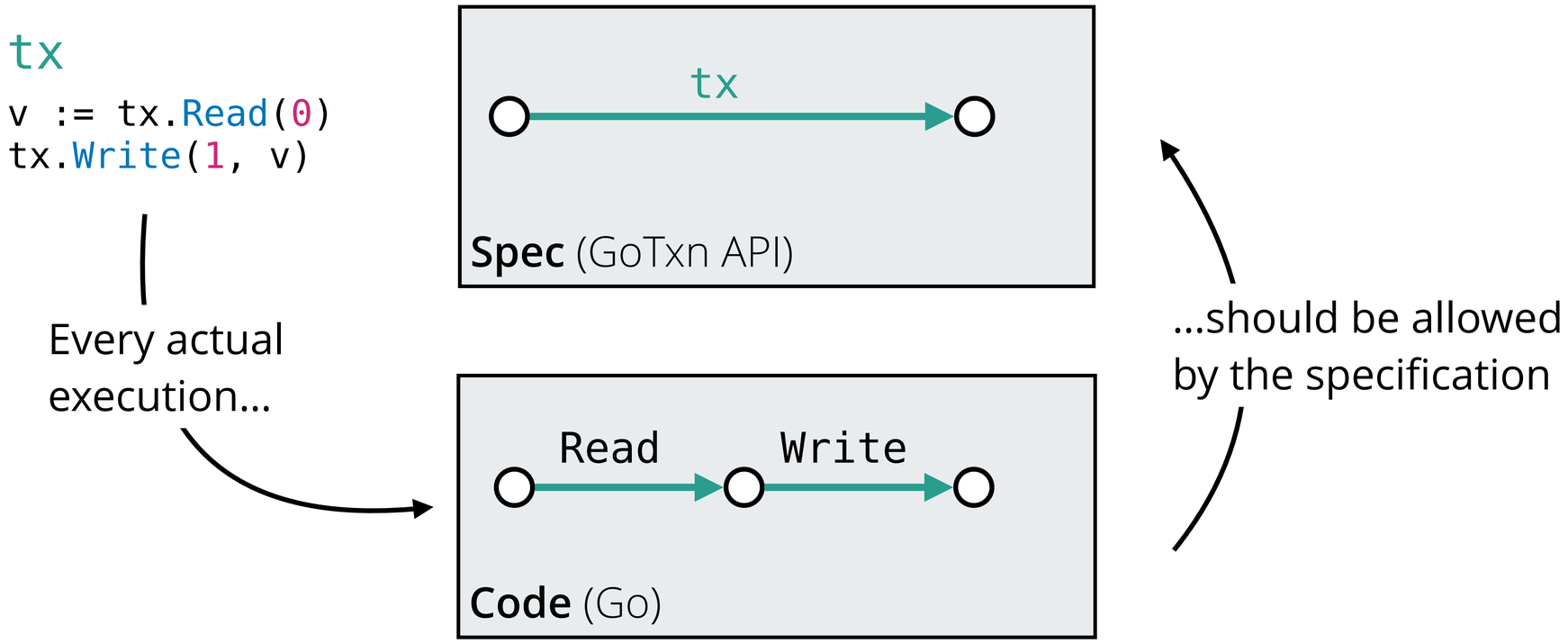
Specifying a transaction system

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



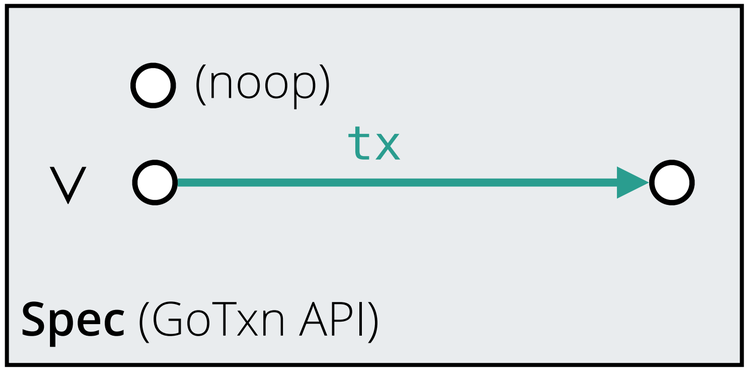


Specifying a transaction system



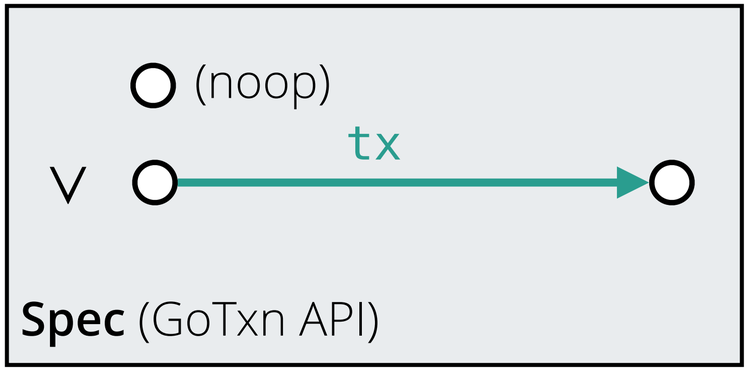
Specifying crash atomicity for transactions

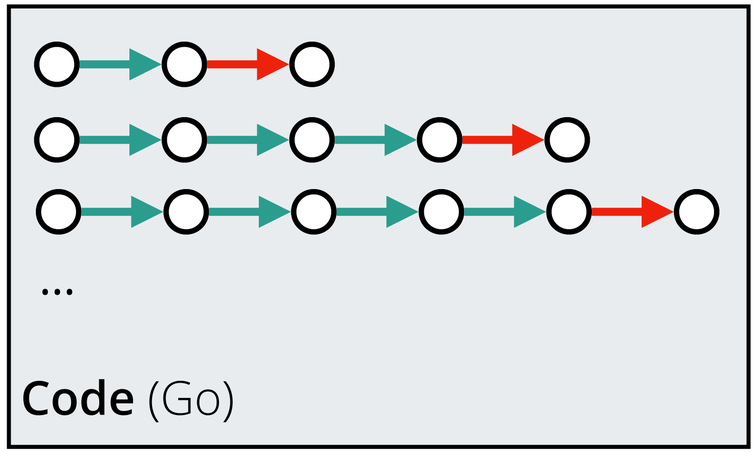
```
tx
v := tx.Read(0)
tx.Write(1, v)
```



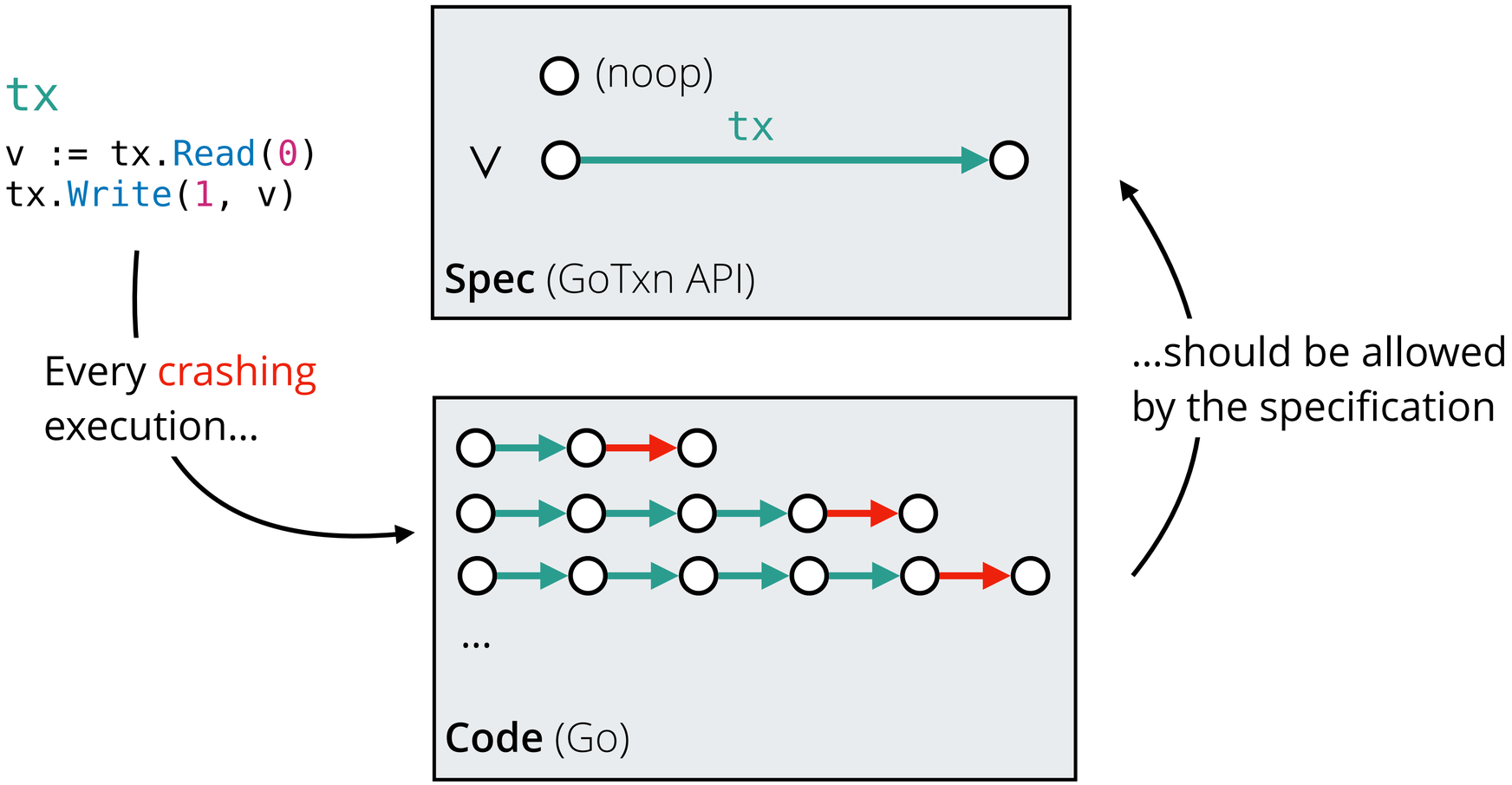
Specifying crash atomicity for transactions

```
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

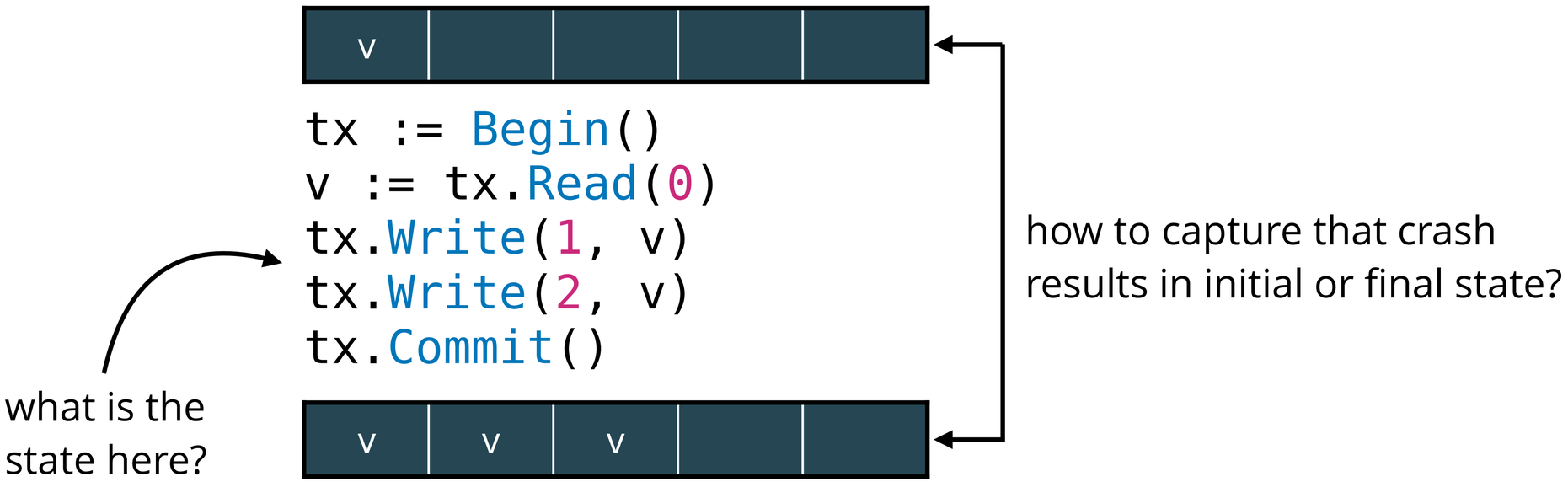

```
tx := Begin()
v := tx.Read(0)
tx.Write(1, v)
tx.Write(2, v)
tx.Commit()
```

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

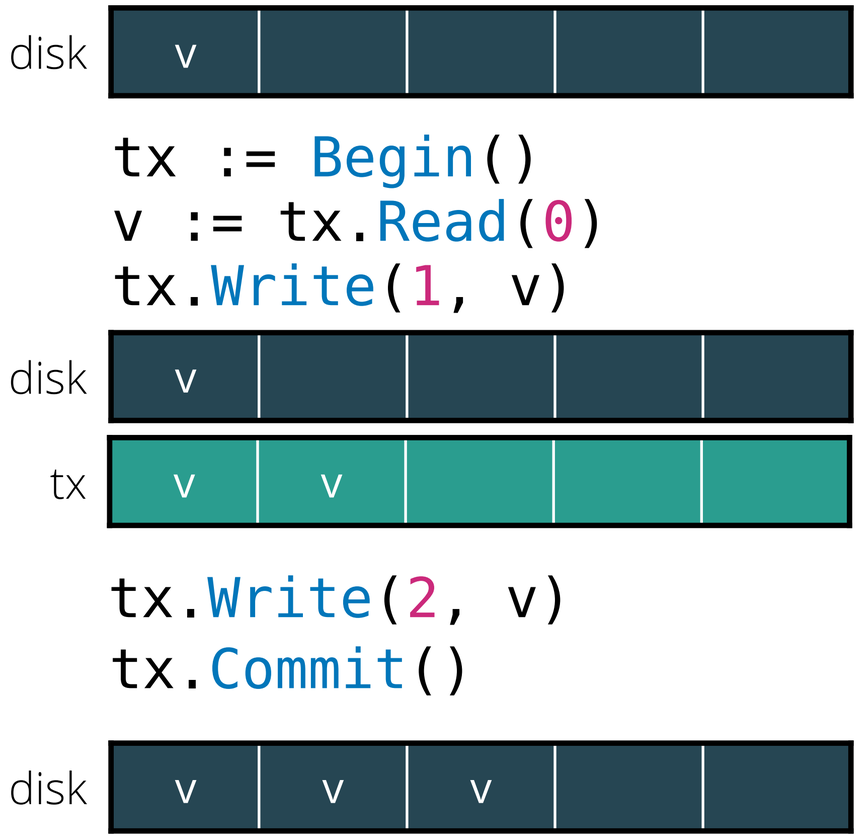


Specifying sequential transactional API


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

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

Specifying sequential transactional API



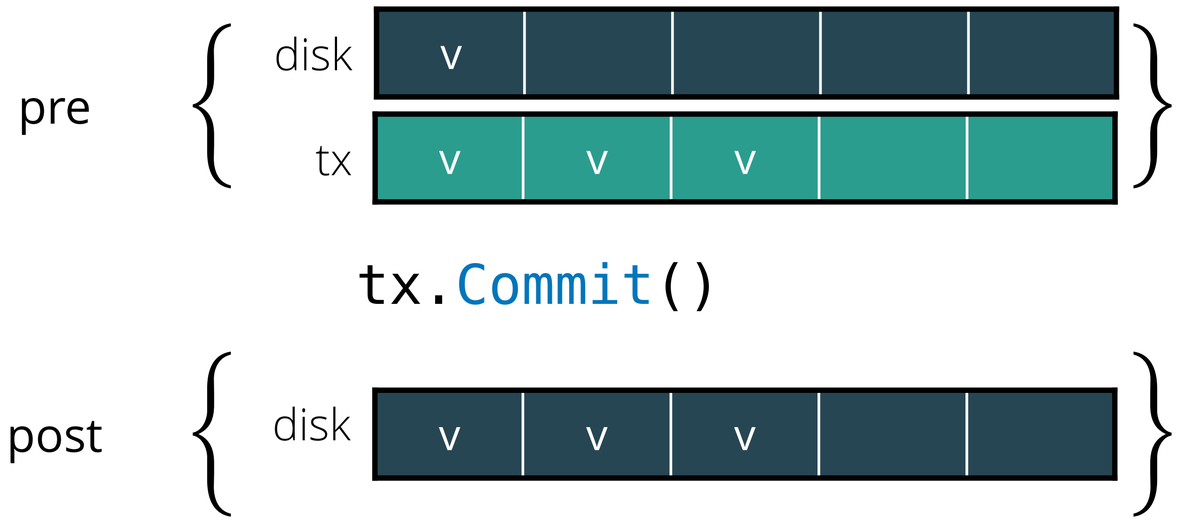
transaction's
in-memory view

Specifying sequential transactional API


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

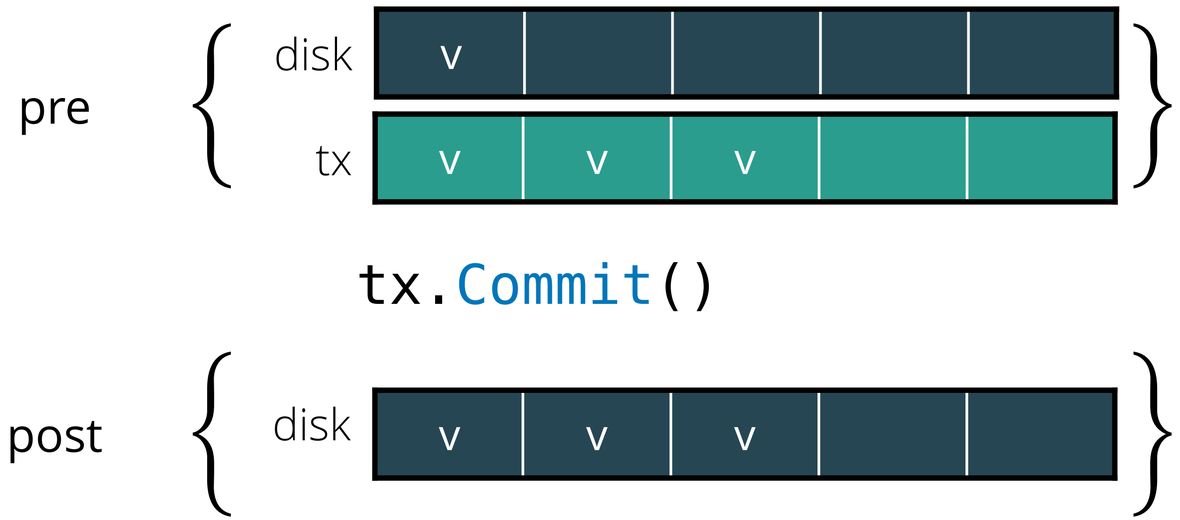

```
tx.Commit()
```

Hoare Logic to specify Commit without crashes



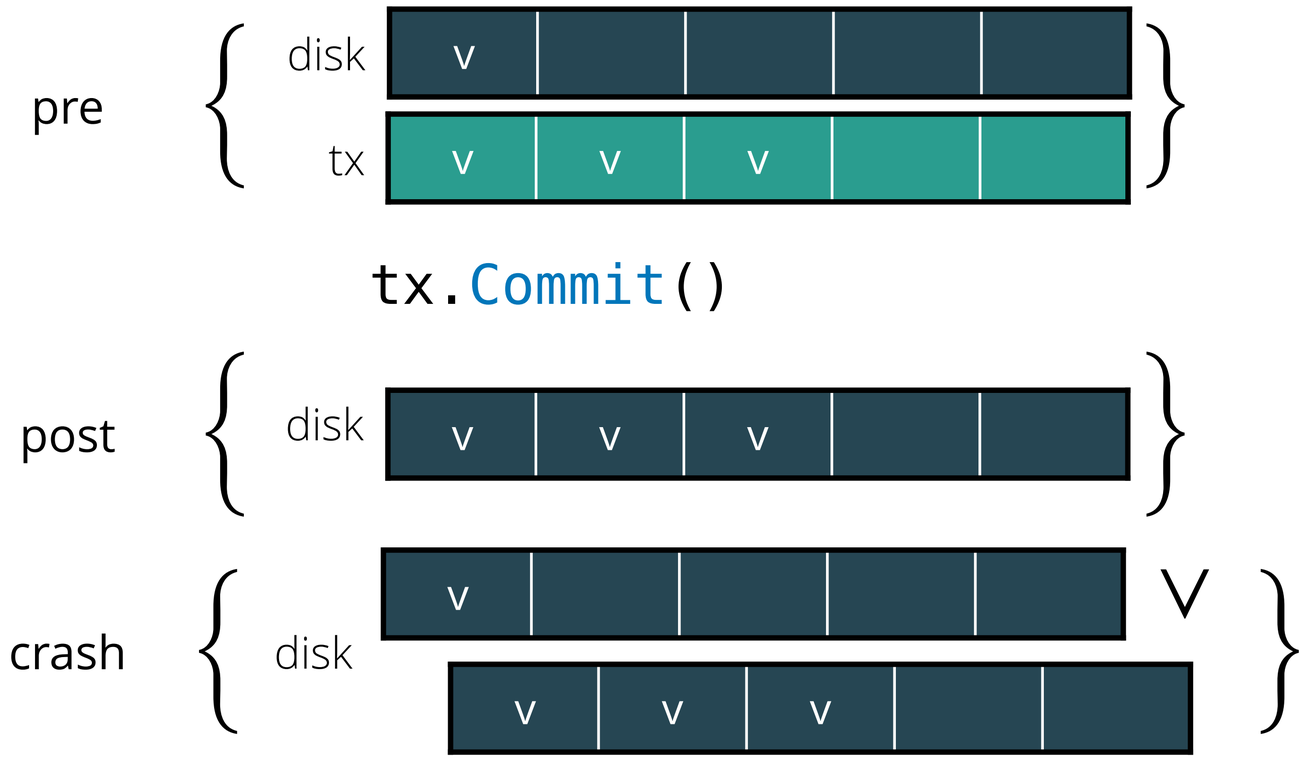
Perennial logic adds crash conditions

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



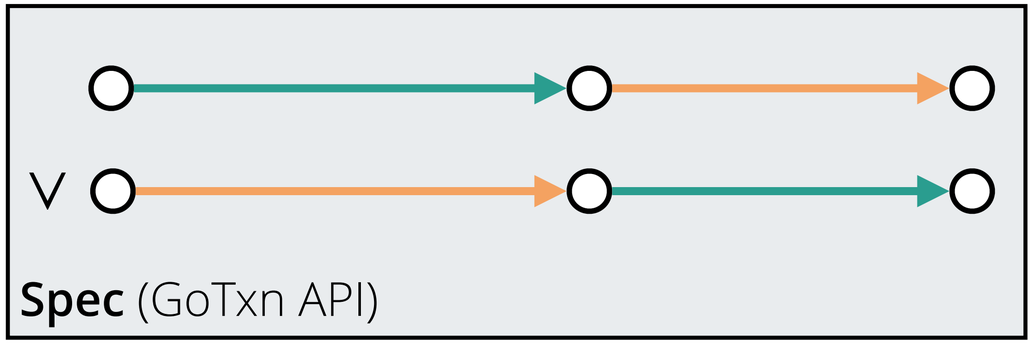
Perennial logic adds crash conditions

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

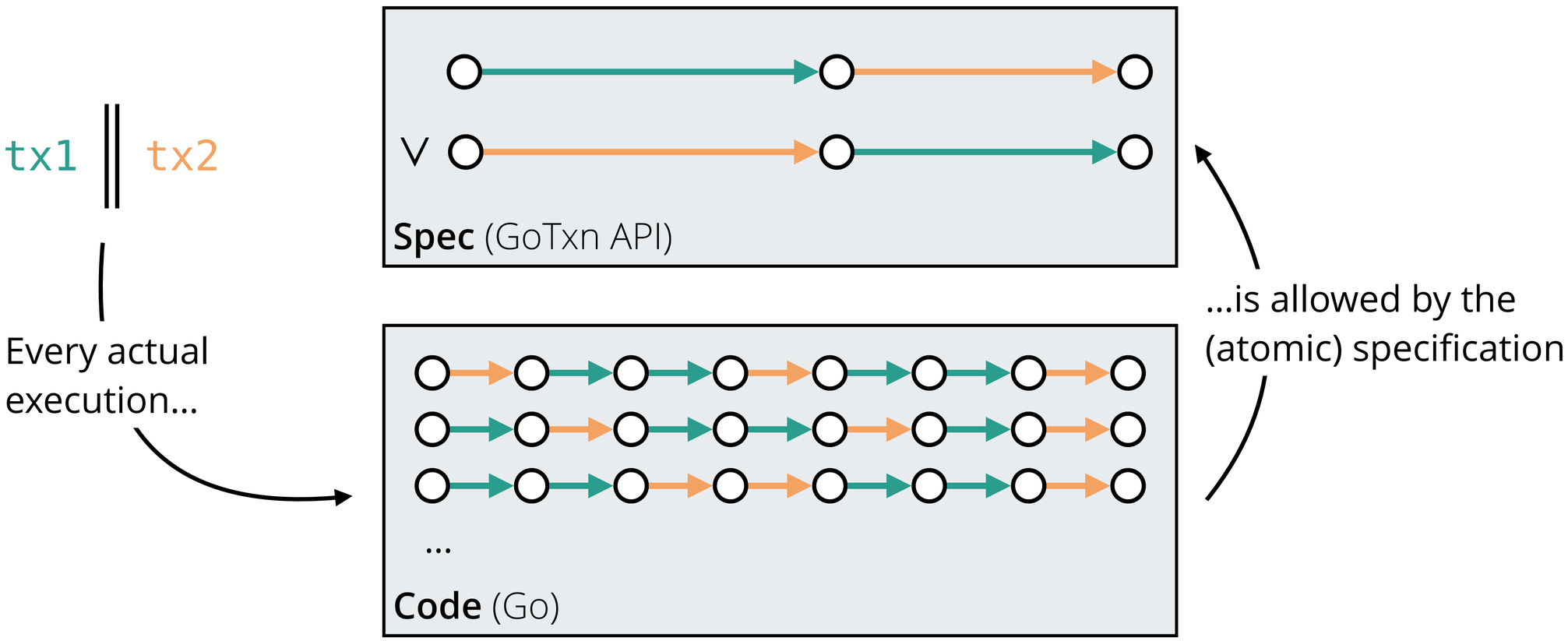


Generalizing to include concurrency

`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


```
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


```
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]

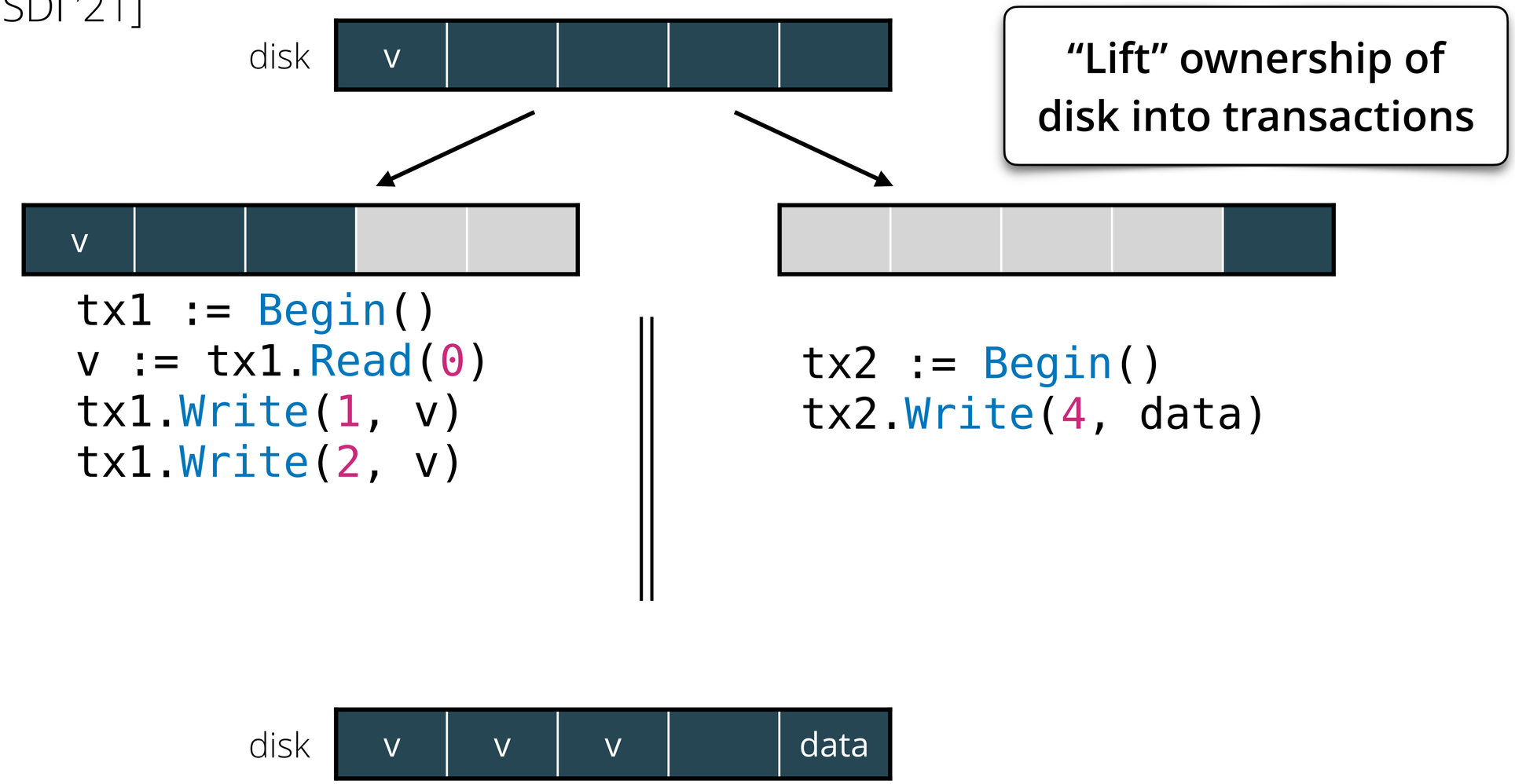

```
tx1 := Begin()
v := tx1.Read(0)
tx1.Write(1, v)
tx1.Write(2, v)
```

‖

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

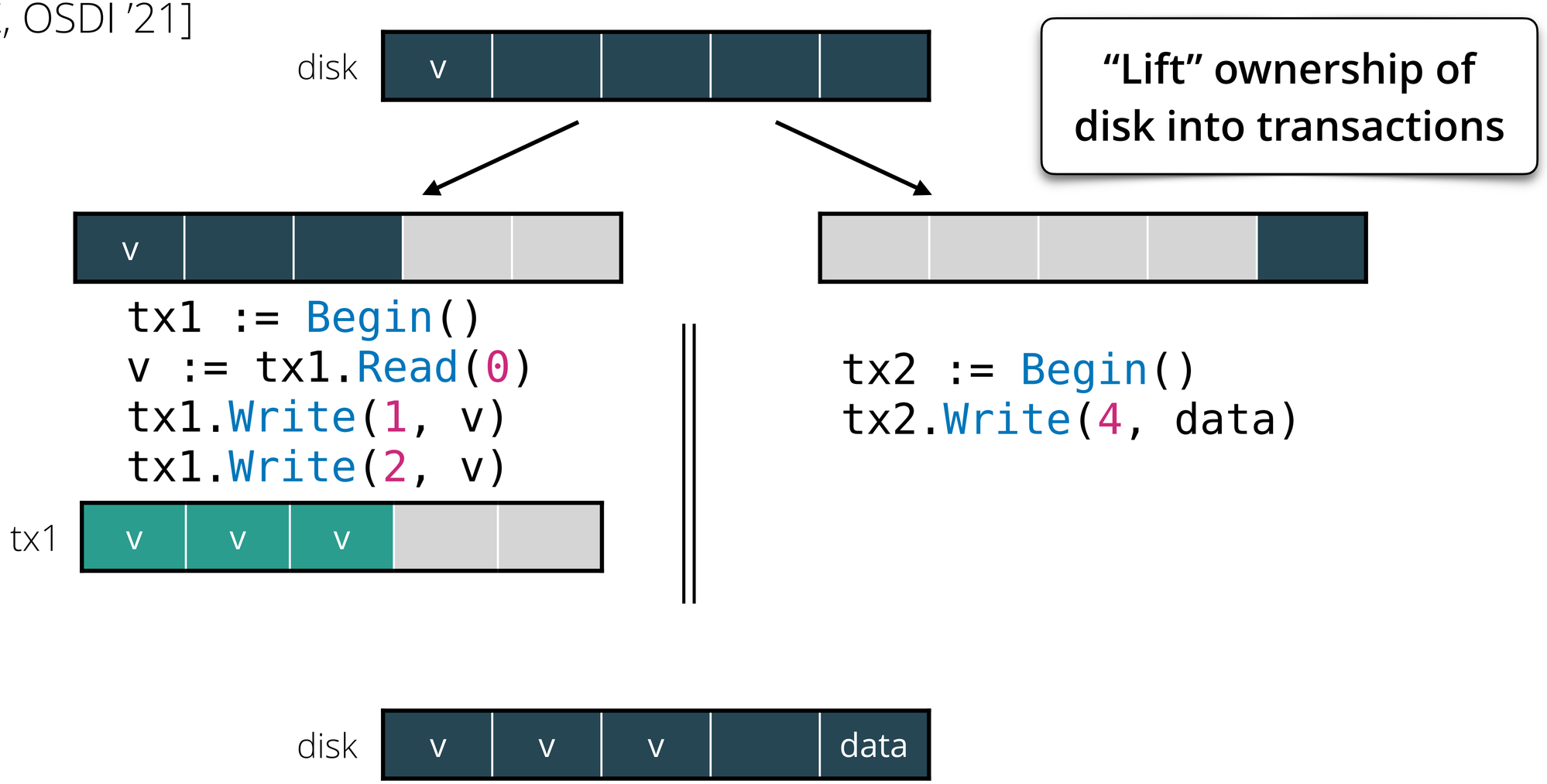
Idea: lifting-based specification

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



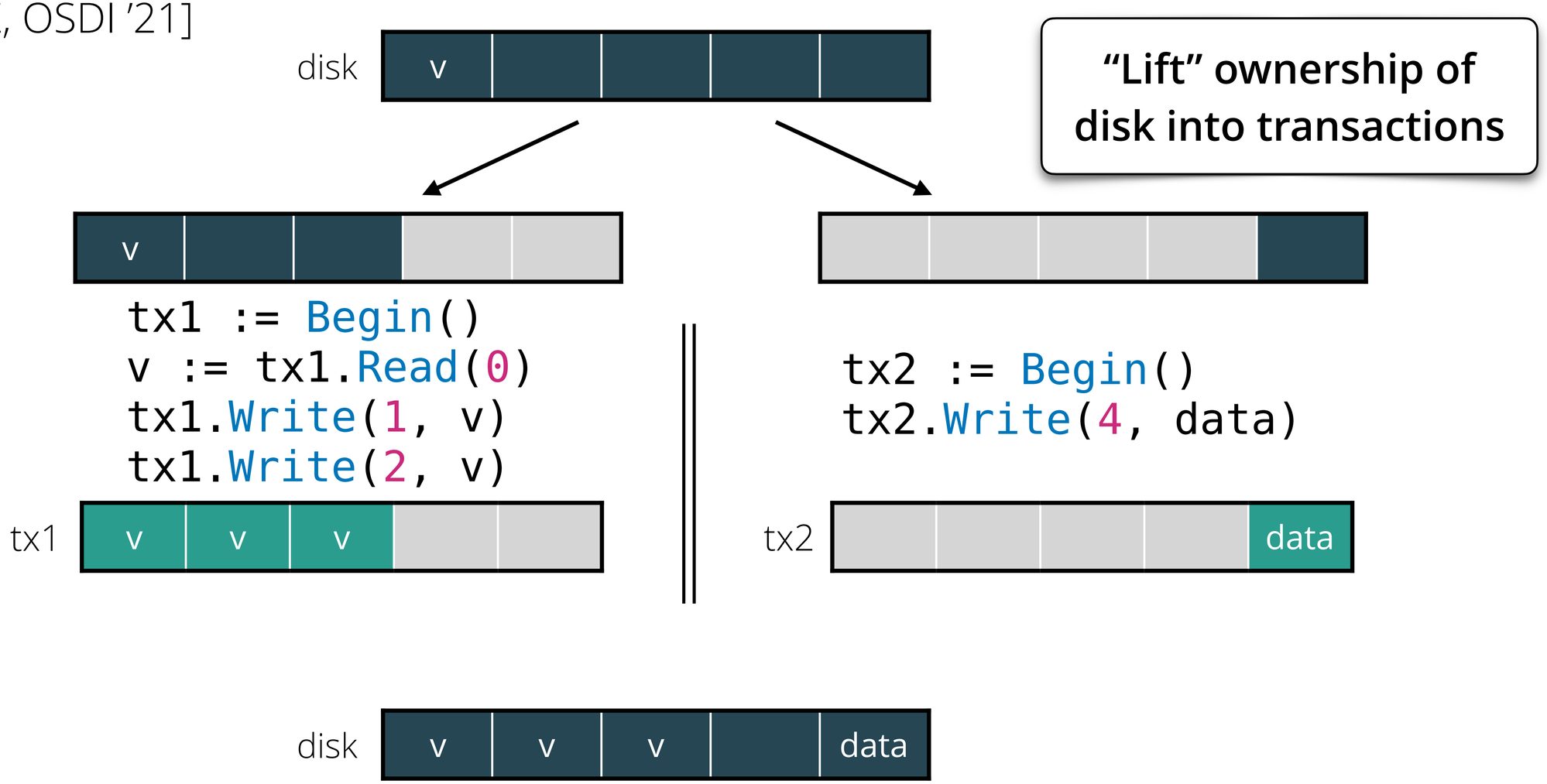
Idea: lifting-based specification

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



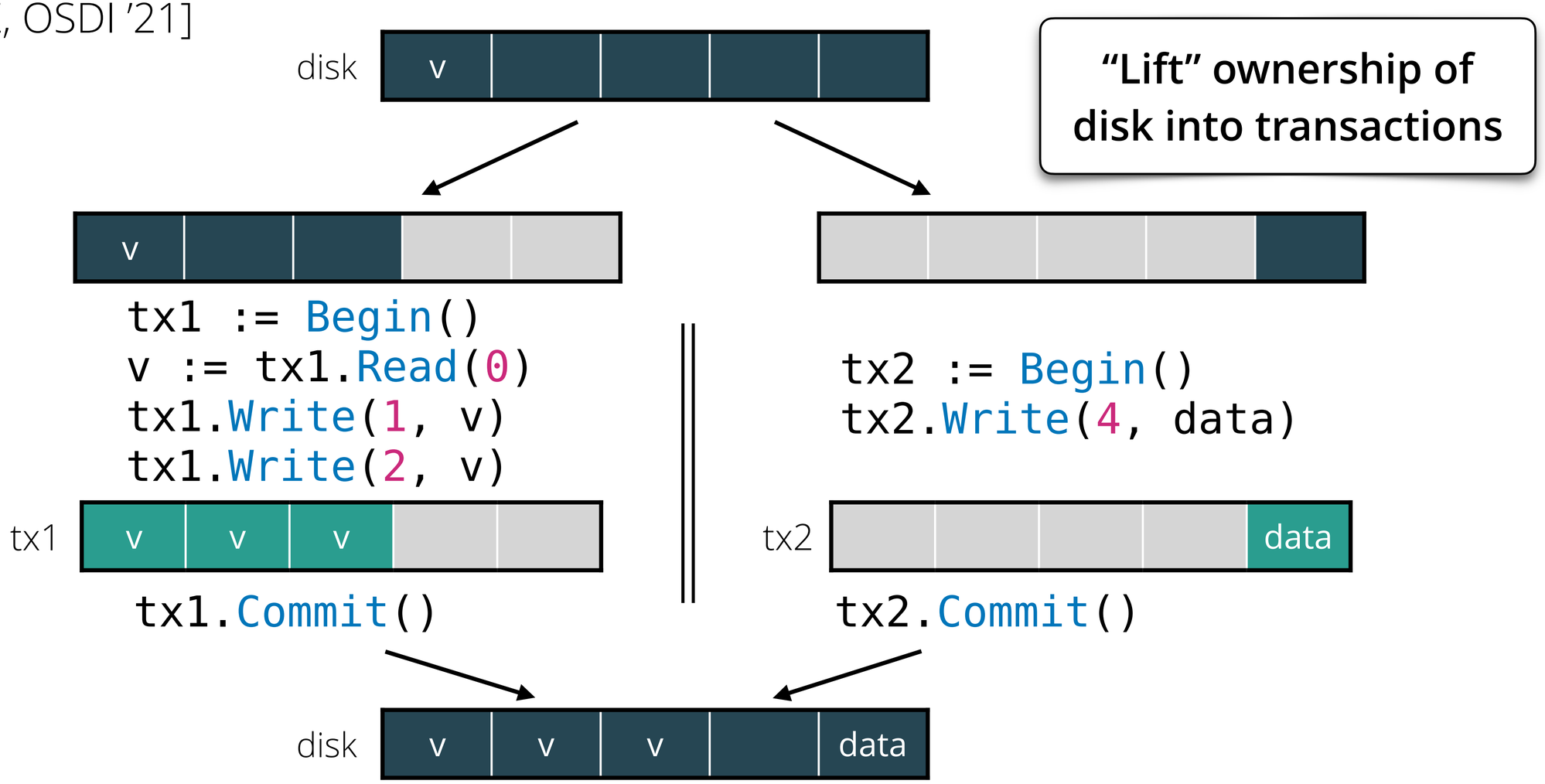
Idea: lifting-based specification

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

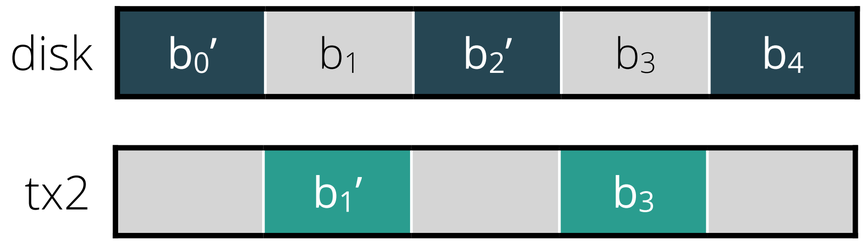


Idea: lifting-based specification

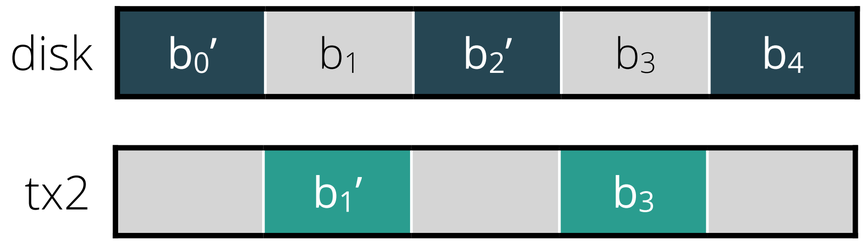
[**C**TTJKZ, OSDI '21]

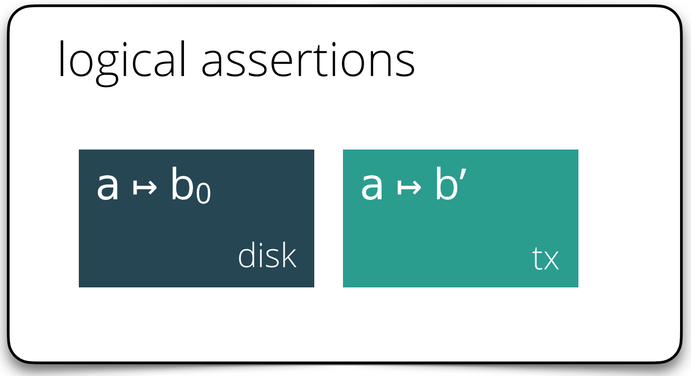


Separation logic describes lifting

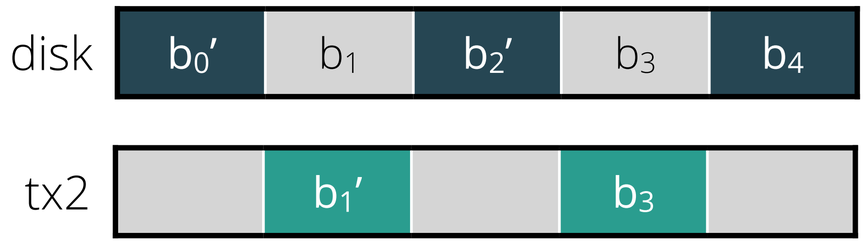


Separation logic describes lifting



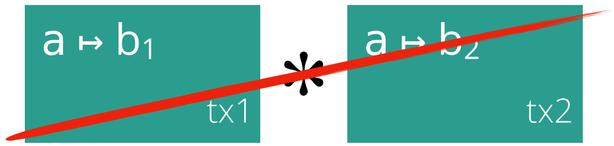


Separation logic describes lifting



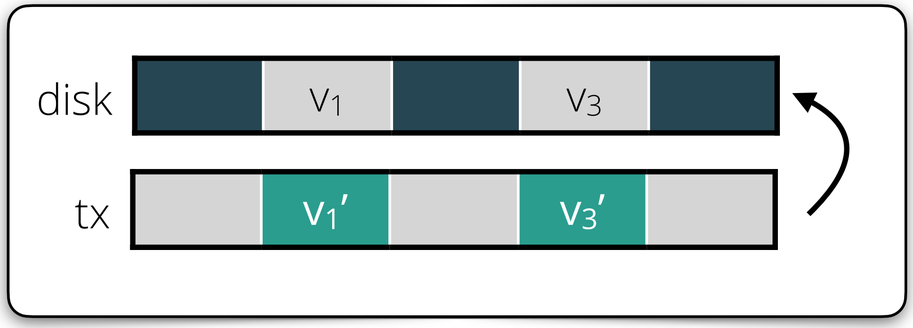
logical assertions

disjointness



Commit spec captures atomicity

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

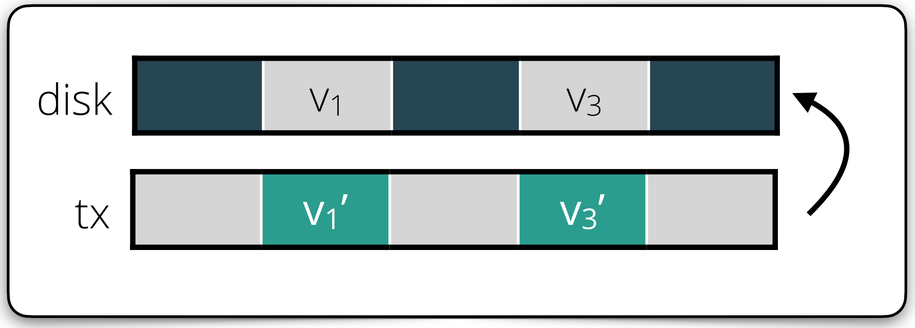


Commit spec captures atomicity

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

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

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



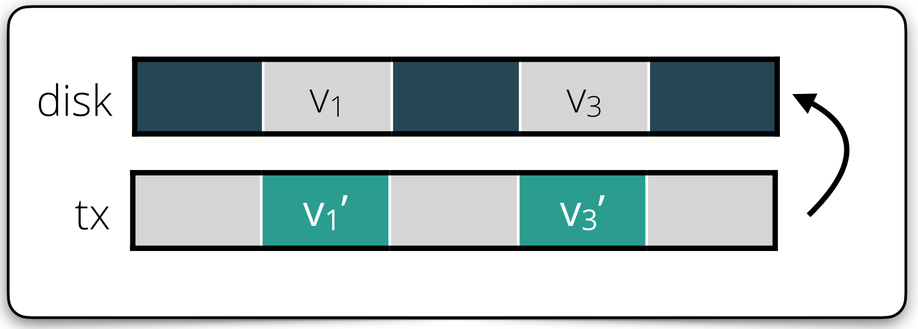
Commit spec captures atomicity

[**C**TTJKZ, 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

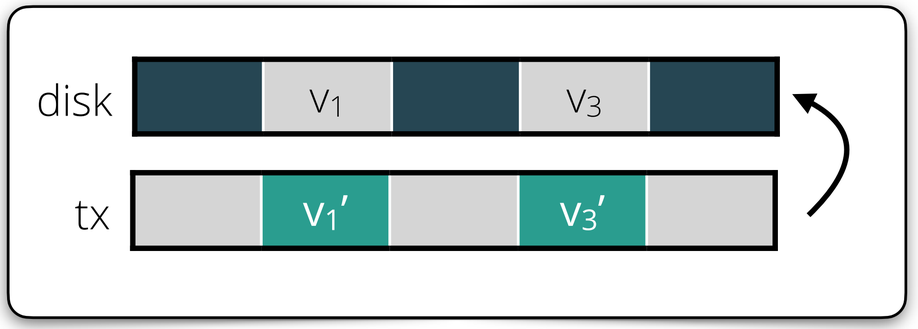
[**C**TTJKZ, OSDI '21]

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

```
tx.Commit()
```

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

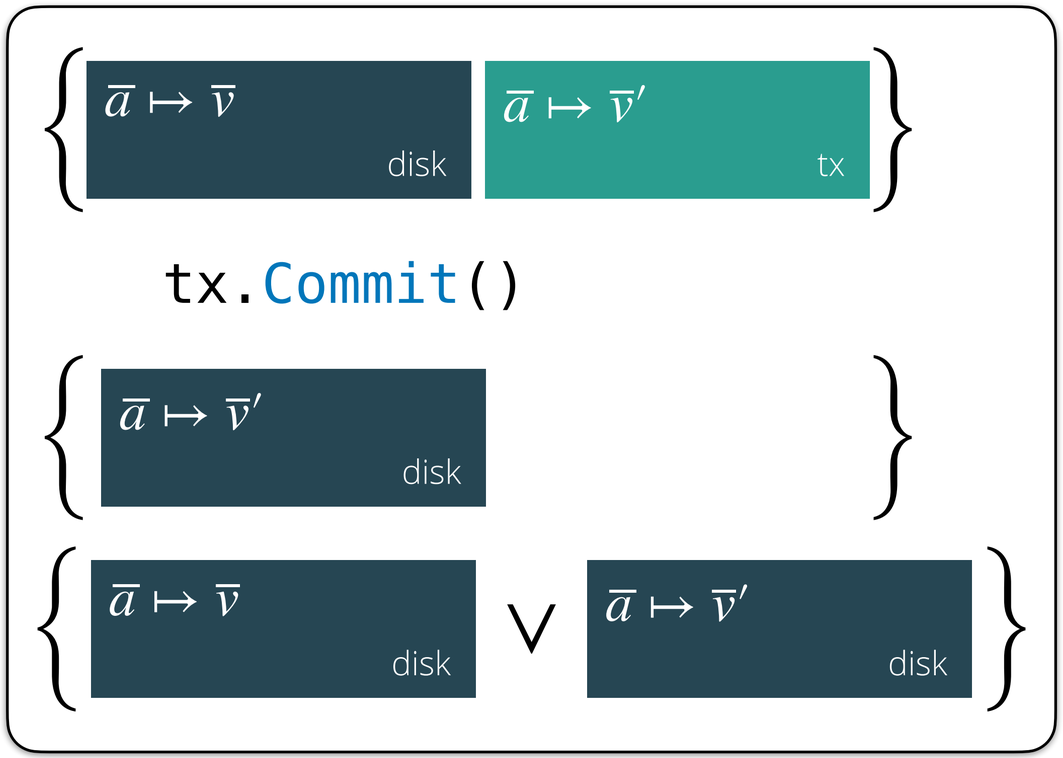
$\{$ $\overline{a} \mapsto \overline{v}$ disk $\vee$ $\overline{a} \mapsto \overline{v}'$ disk $\}$

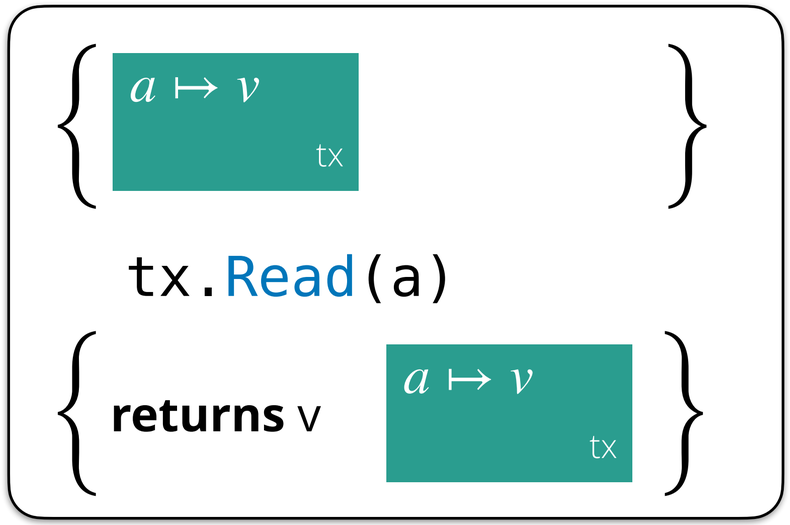


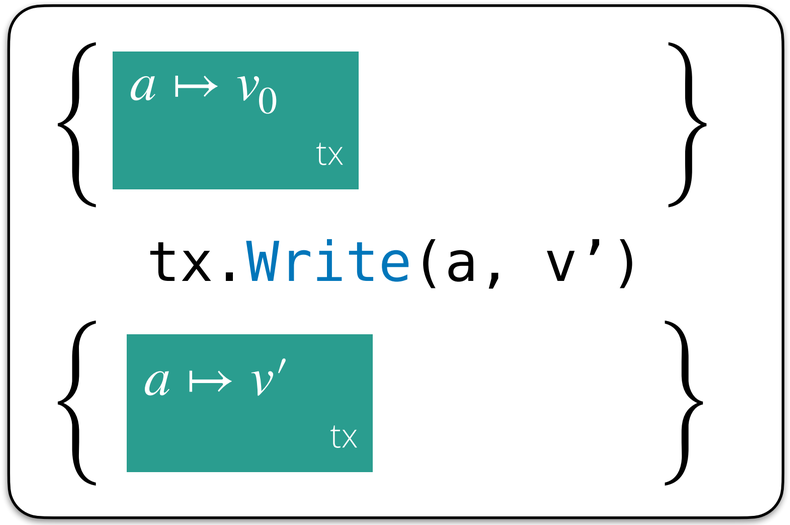
crash condition is atomic

Lifting specification describes the GoTxn API

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

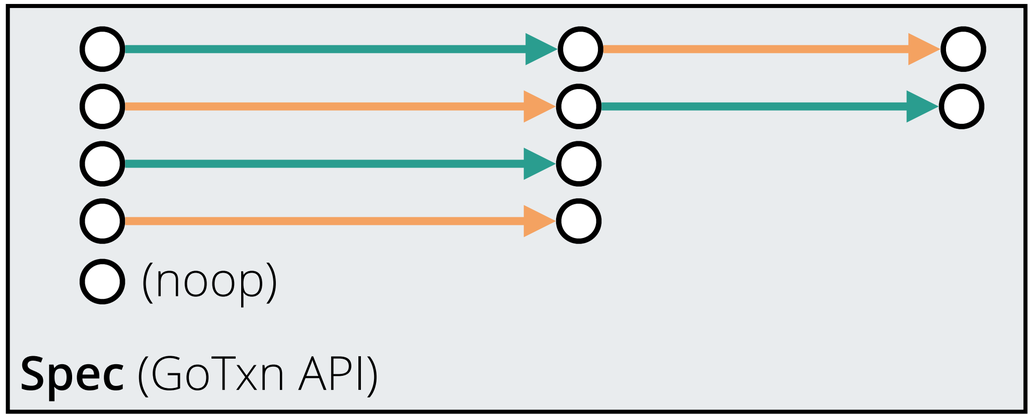




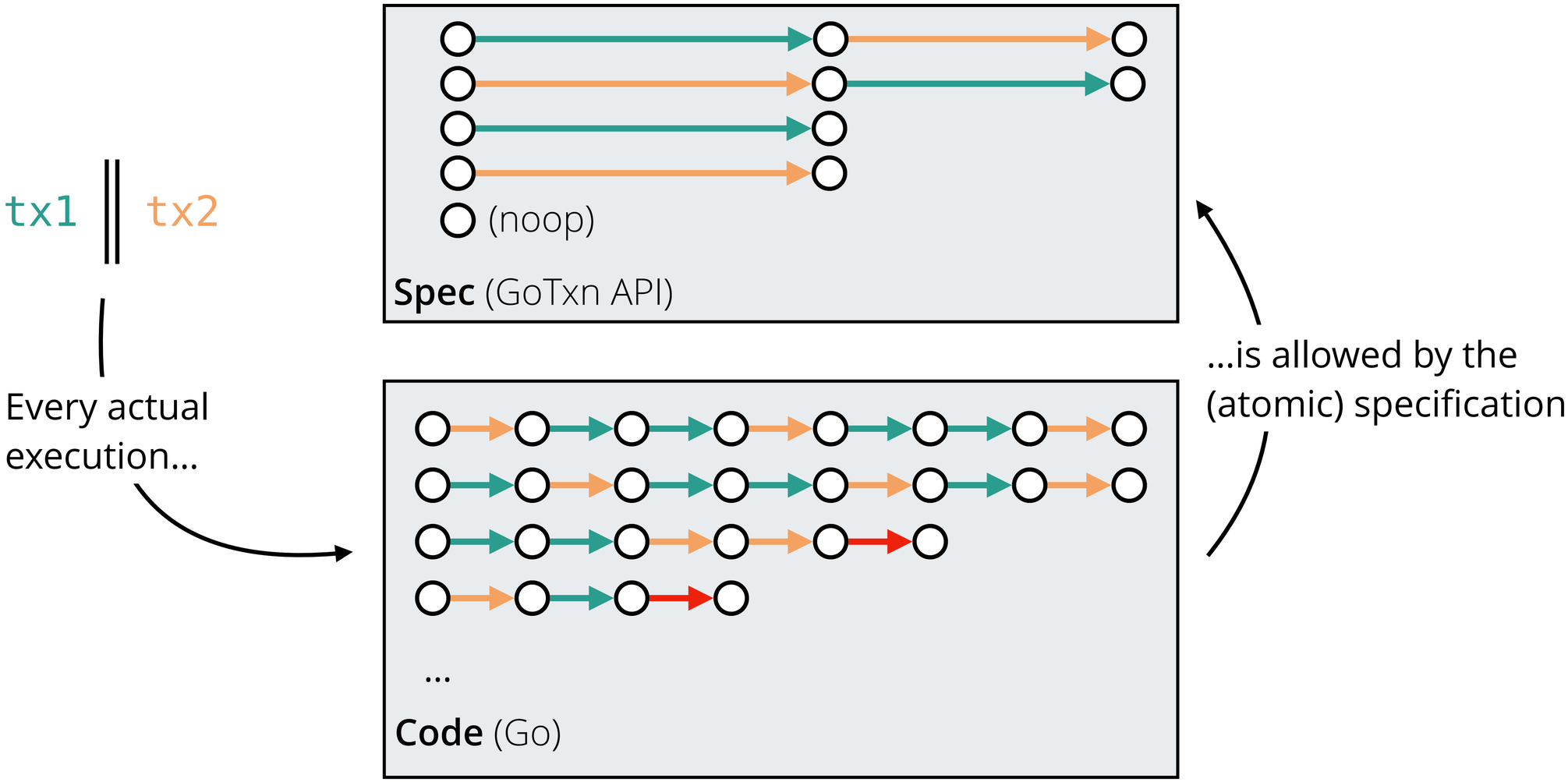


Complete GoTxn specification

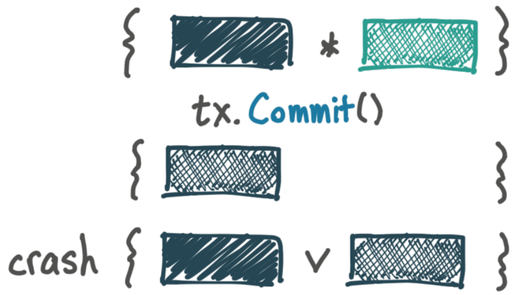
`tx1` ‖ `tx2`



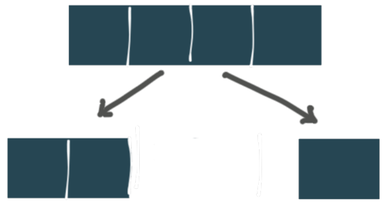
Complete GoTxn specification



Summary of specifying transaction system



Perennial logic supports specifying and proving crash and concurrent behavior



Lifting specification describes concurrent transactions

Roadmap

DaisyNFS

File-system code implemented with transactions

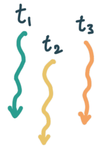
Specification
for transactions

Specification that bridges the two ✓

GoTxn

Transaction system gives atomicity

Crashes



Concurrency

Roadmap

DaisyNFS

File-system code implemented with transactions

Specification
for transactions

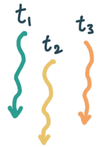
Specification that bridges the two

✓

GoTxn

Transaction system gives atomicity

Crashes



Concurrency

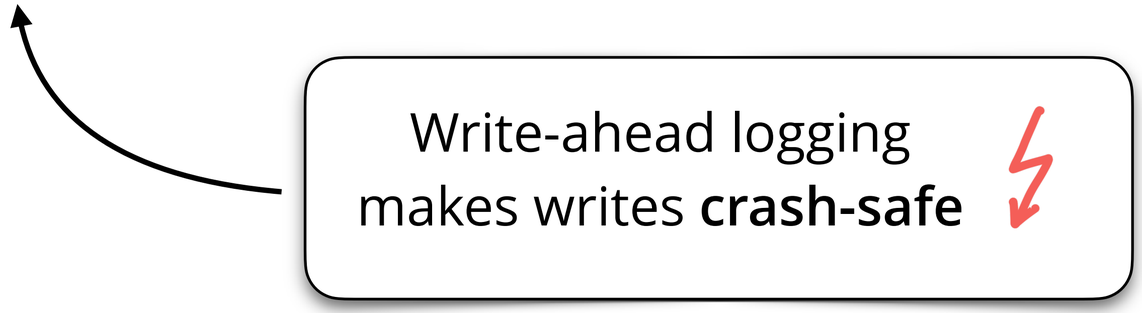

GoTxn

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()
```



Implementing GoTxn

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

Two-phase locking
handles **concurrency**

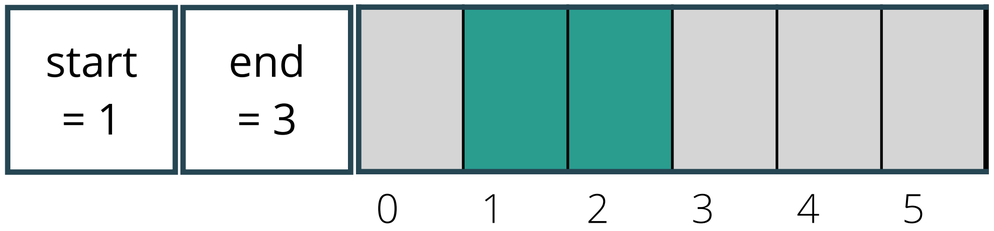
Write-ahead logging
makes writes **crash-safe**

Write-ahead logging API

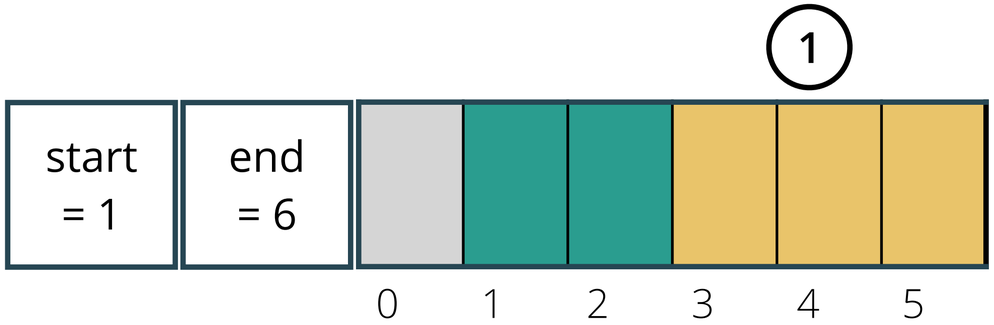
```
func Multiwrite(ws []Update)
type Update struct {
   addr  uint64
   block []byte
}

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

Write-ahead log (WAL)

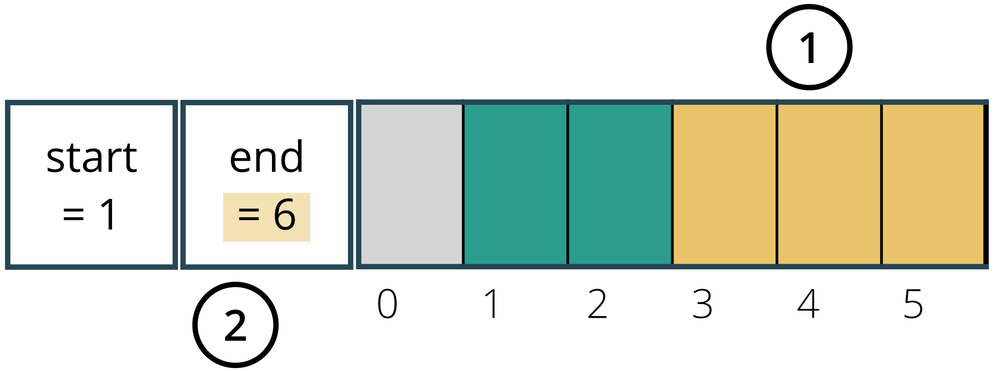
Multi-block atomicity come from circular log



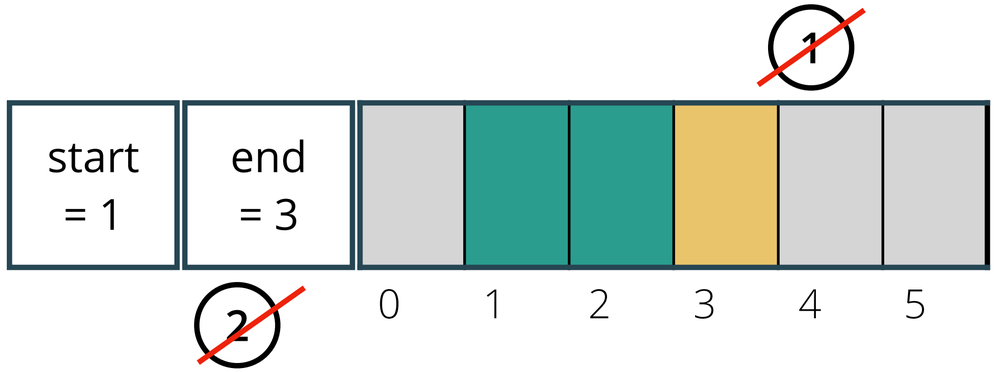
Multi-block atomicity come from circular log



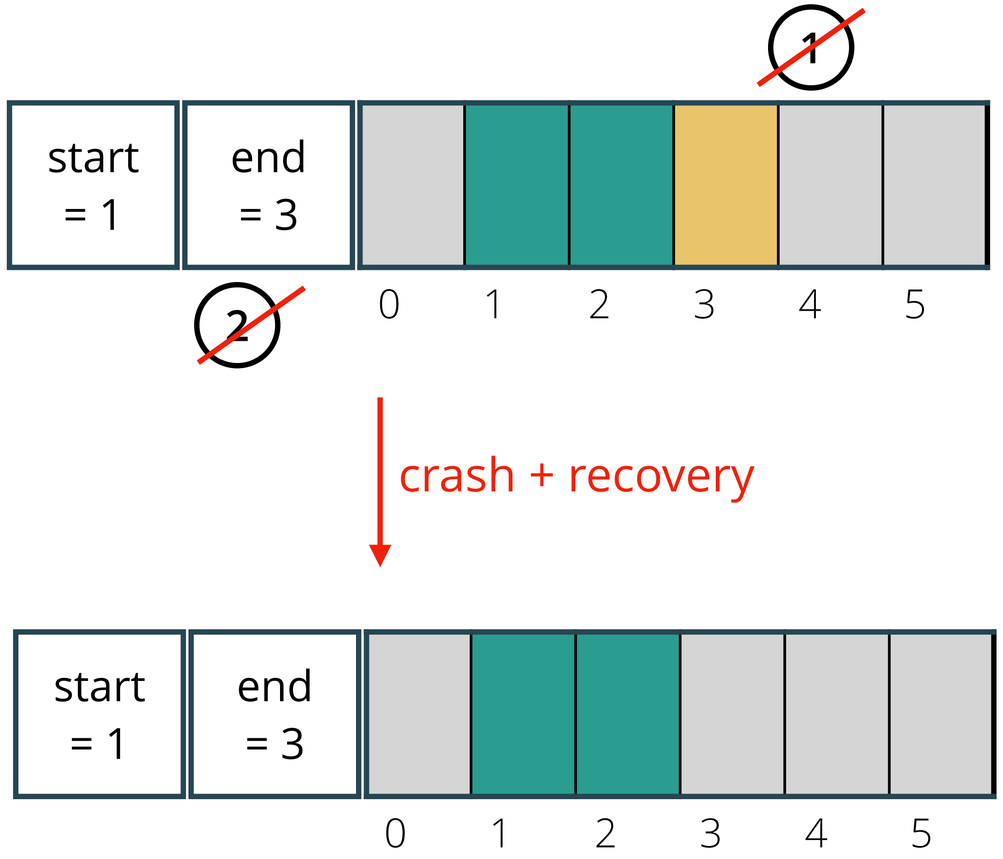
Multi-block atomicity come from circular log



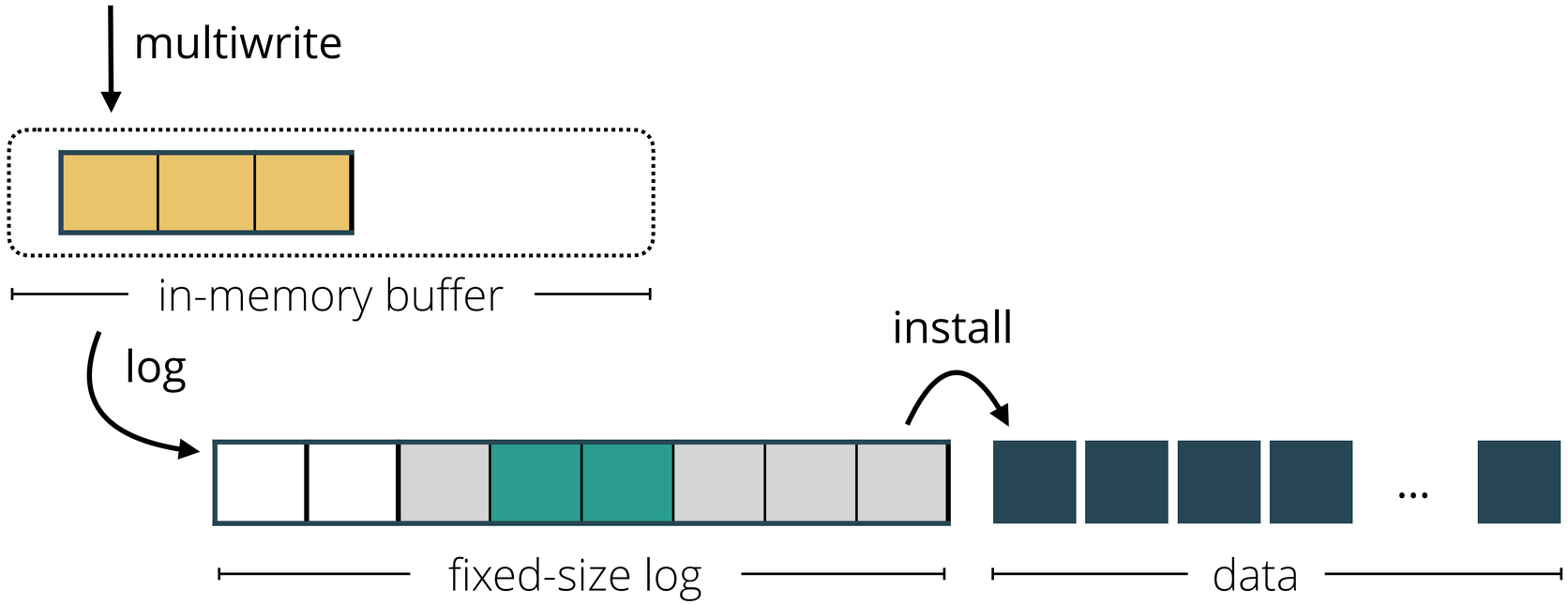
Crash never leaves a partial multiwrite



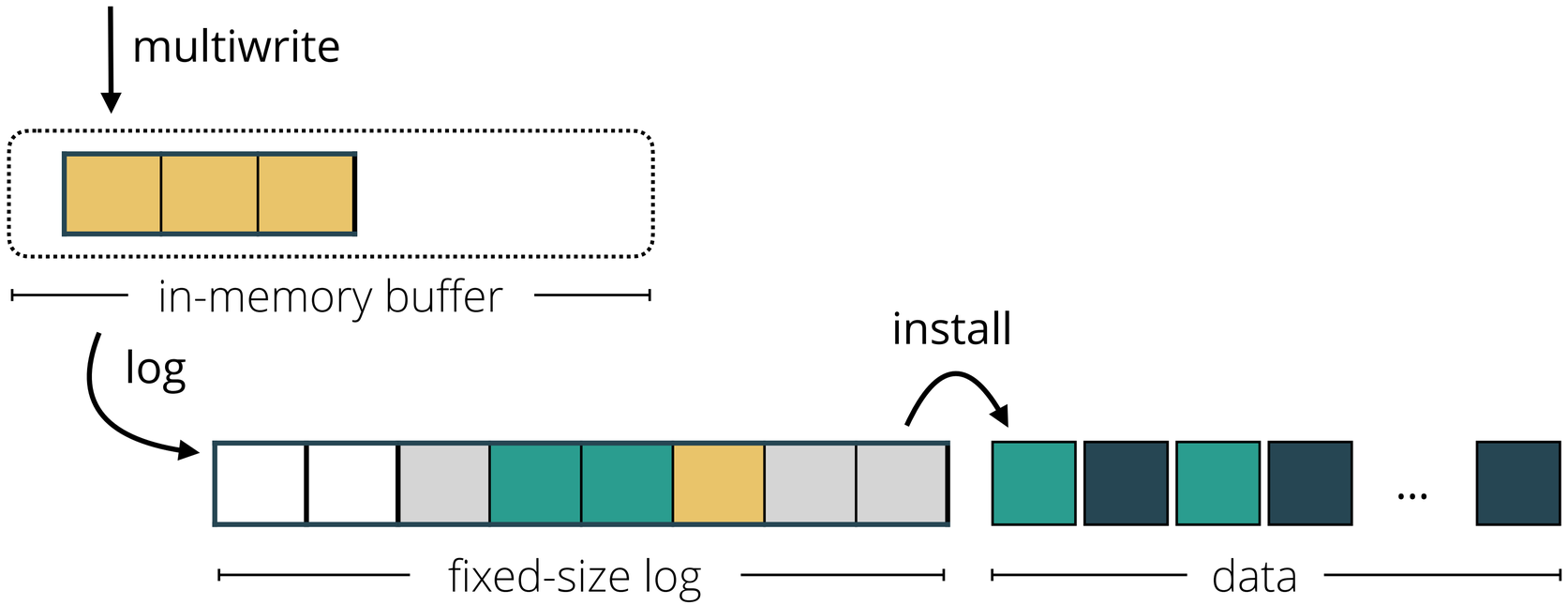
Crash never leaves a partial multiwrite



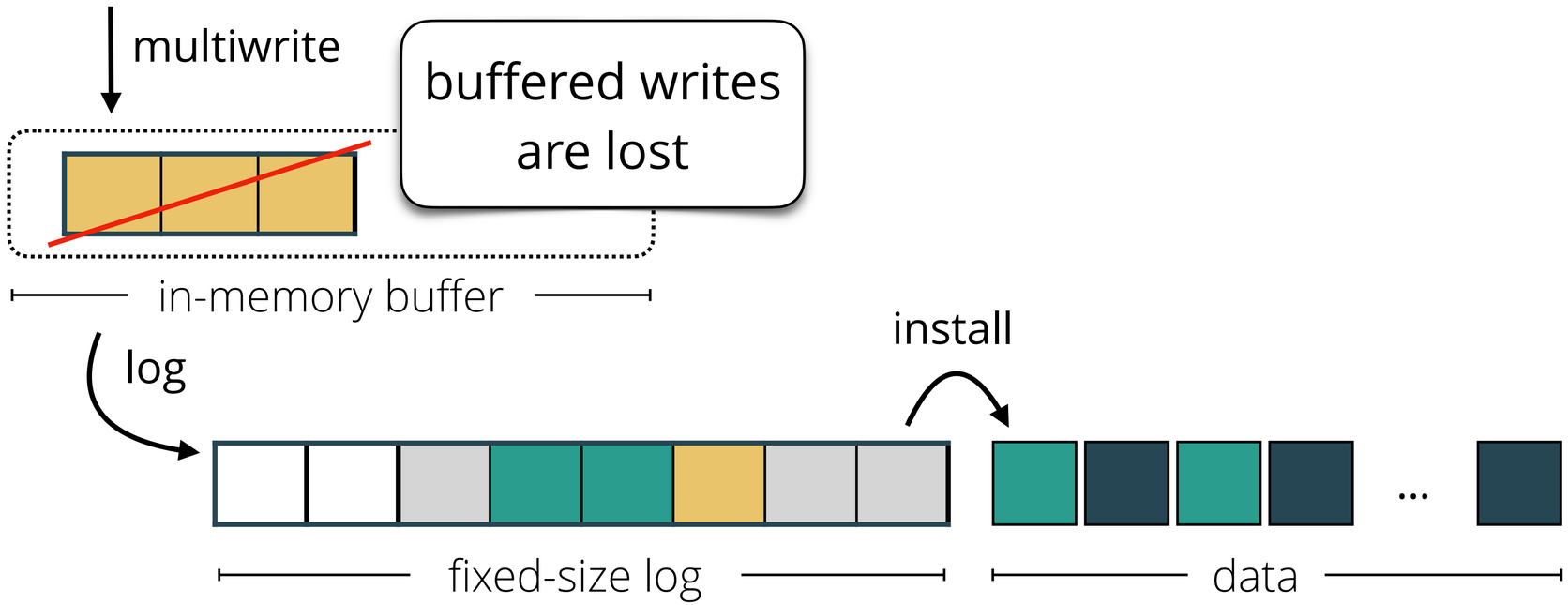
Writing, logging, and installation are concurrent



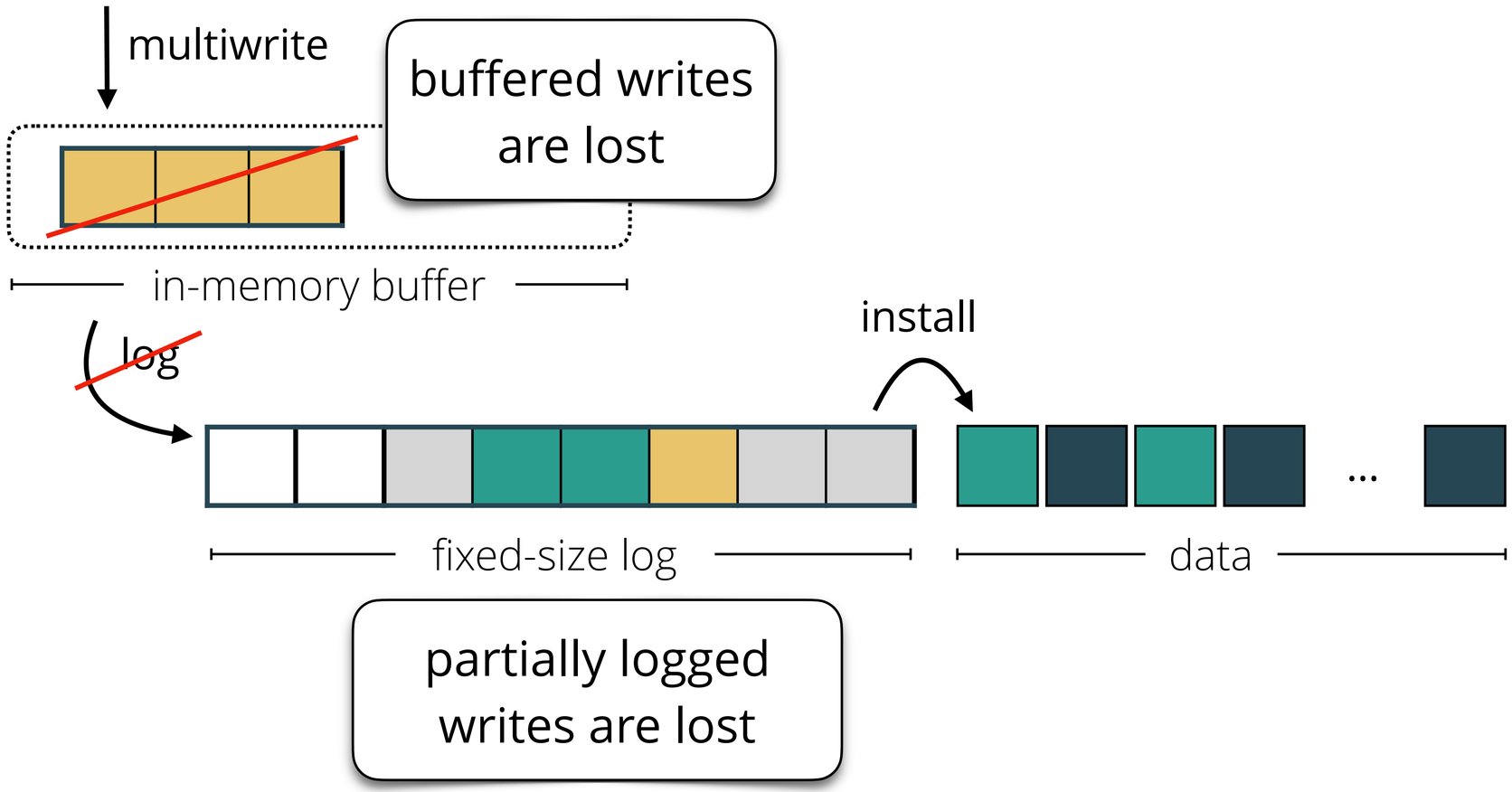
Crashes are complicated in the WAL



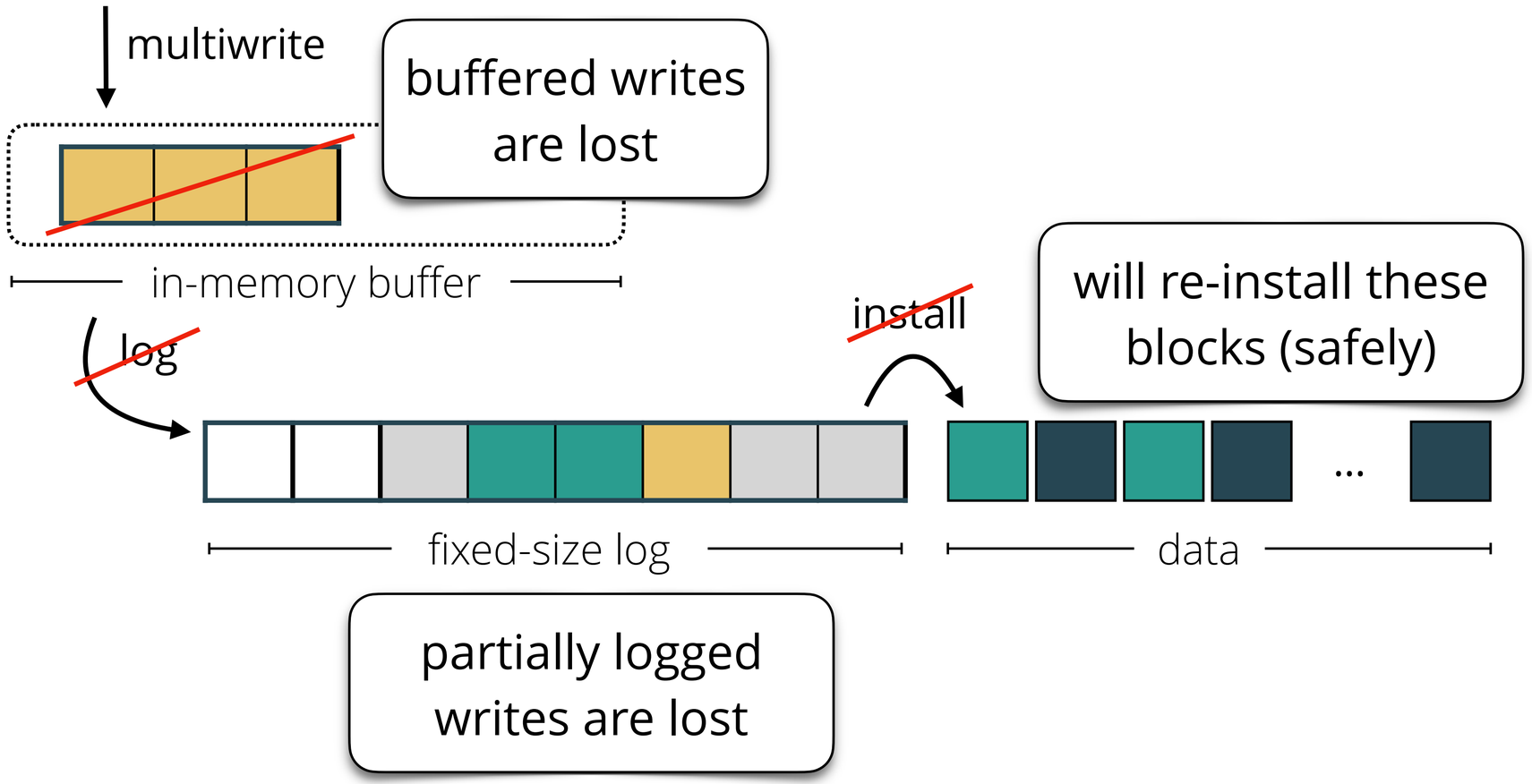
Crashes are complicated in the WAL



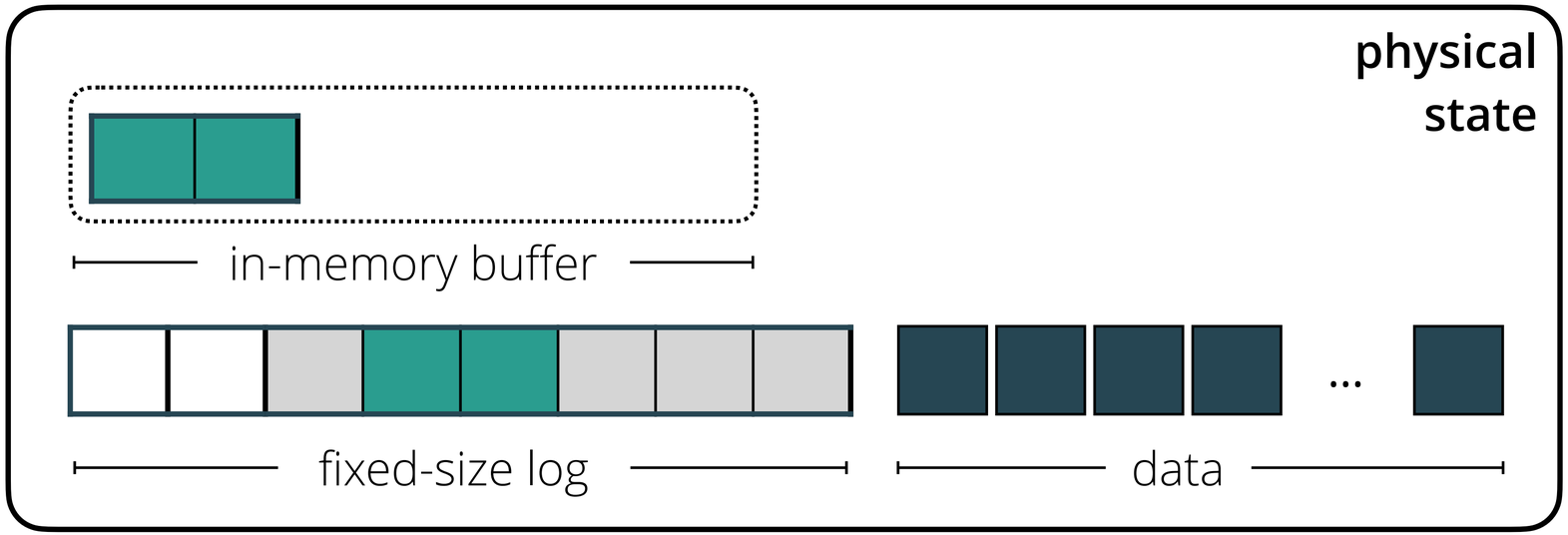
Crashes are complicated in the WAL



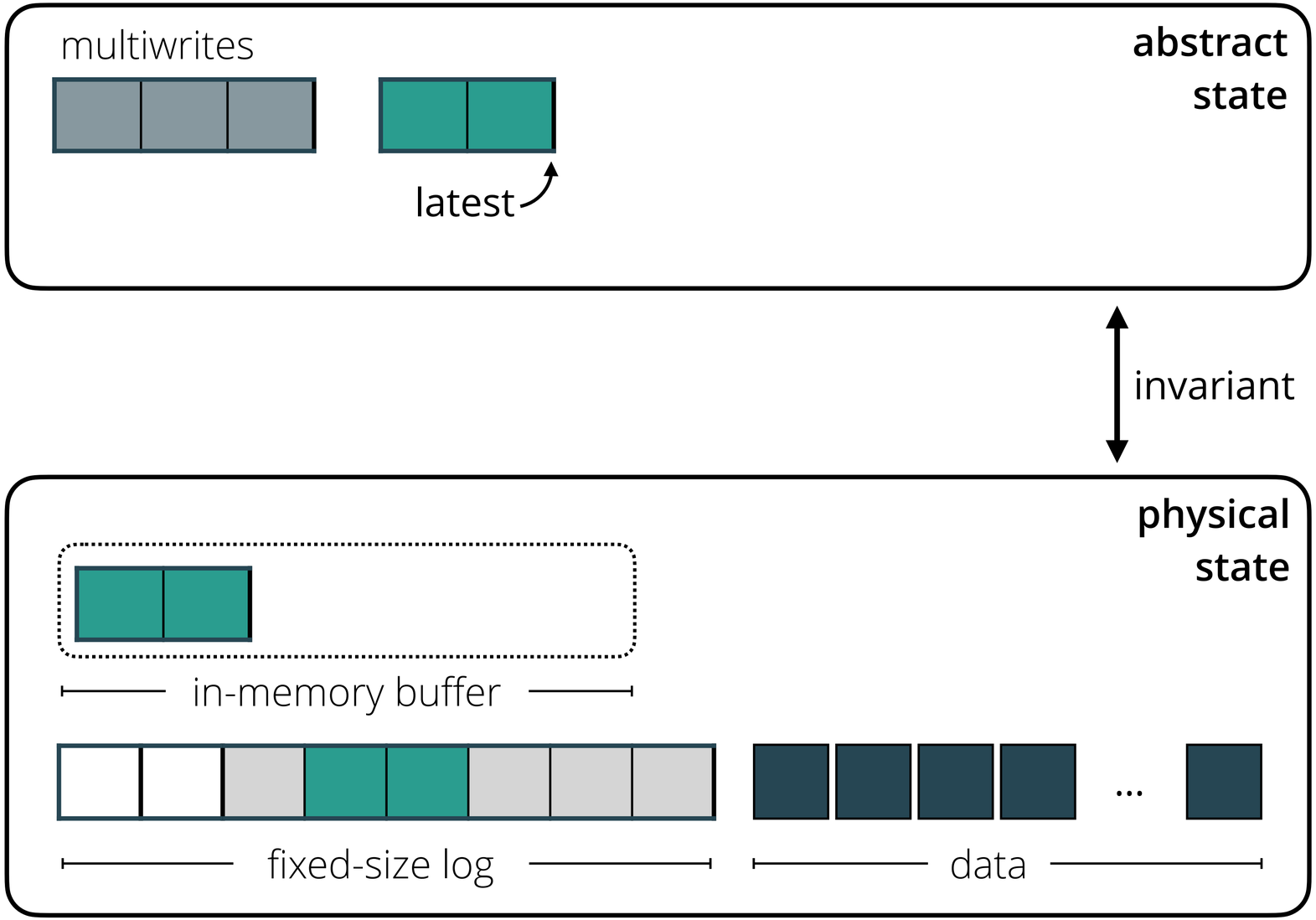
Crashes are complicated in the WAL



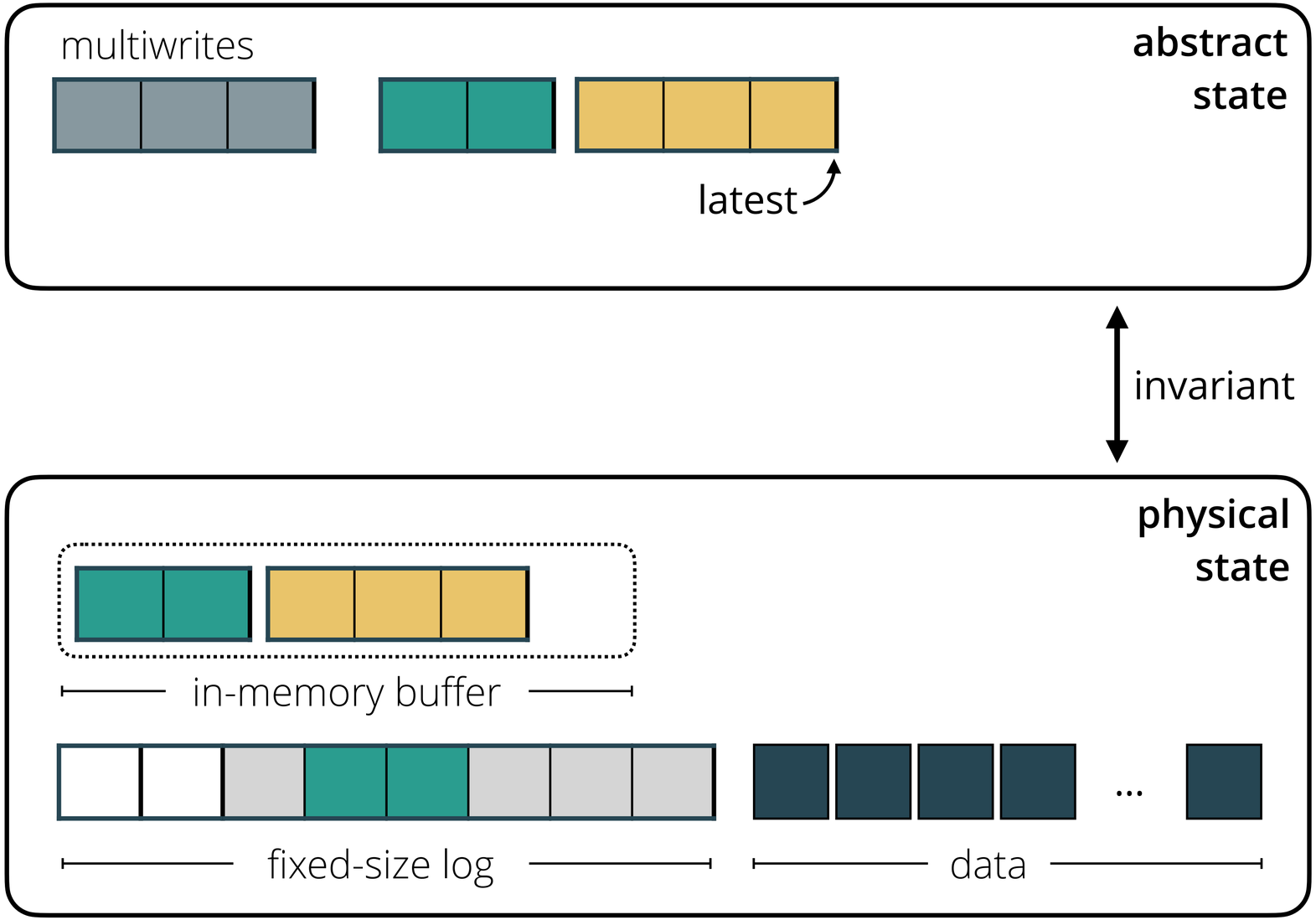
Idea: model WAL as a history of multiwrites



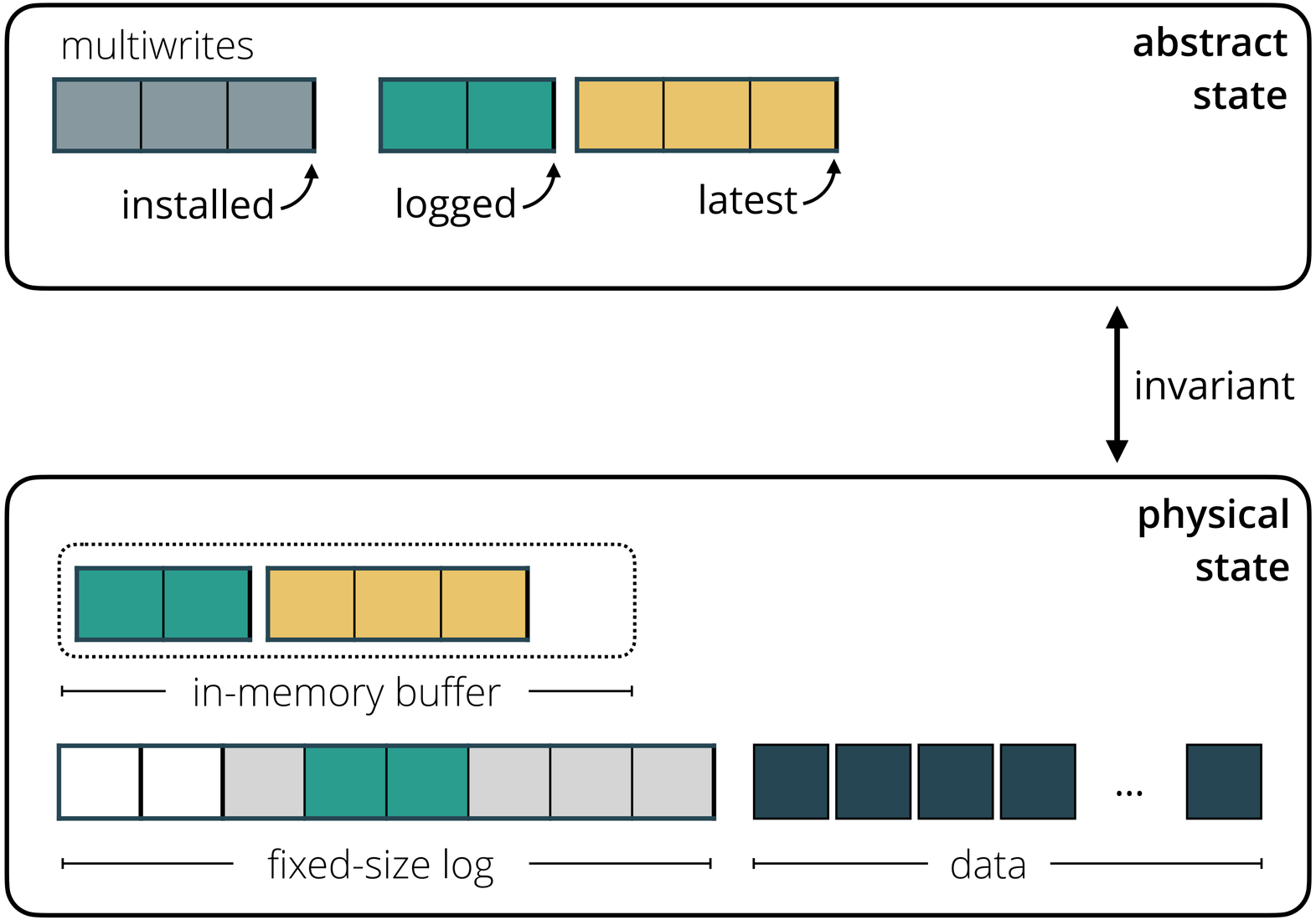
Idea: model WAL as a history of multiwrites



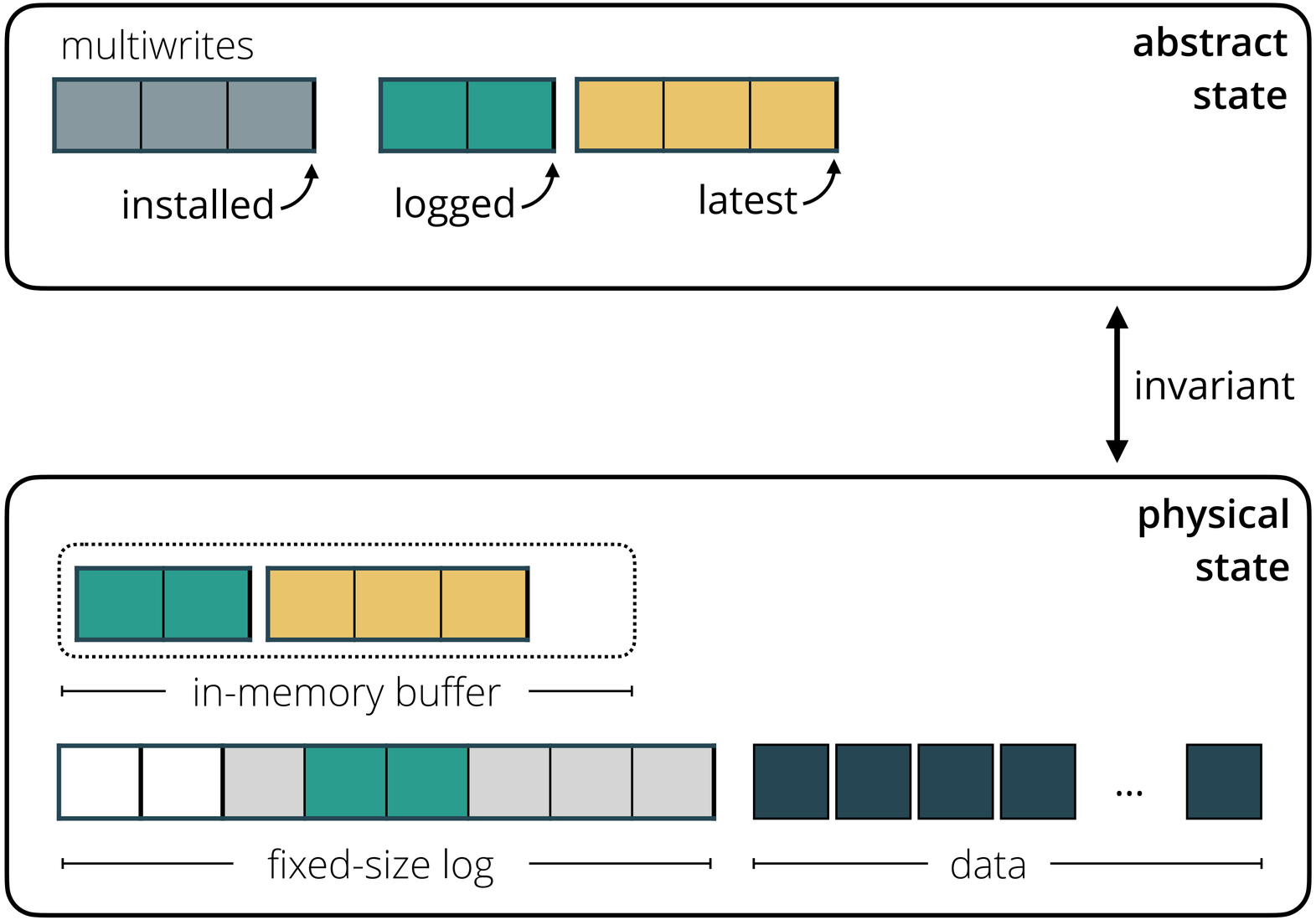
Idea: model WAL as a history of multiwrites



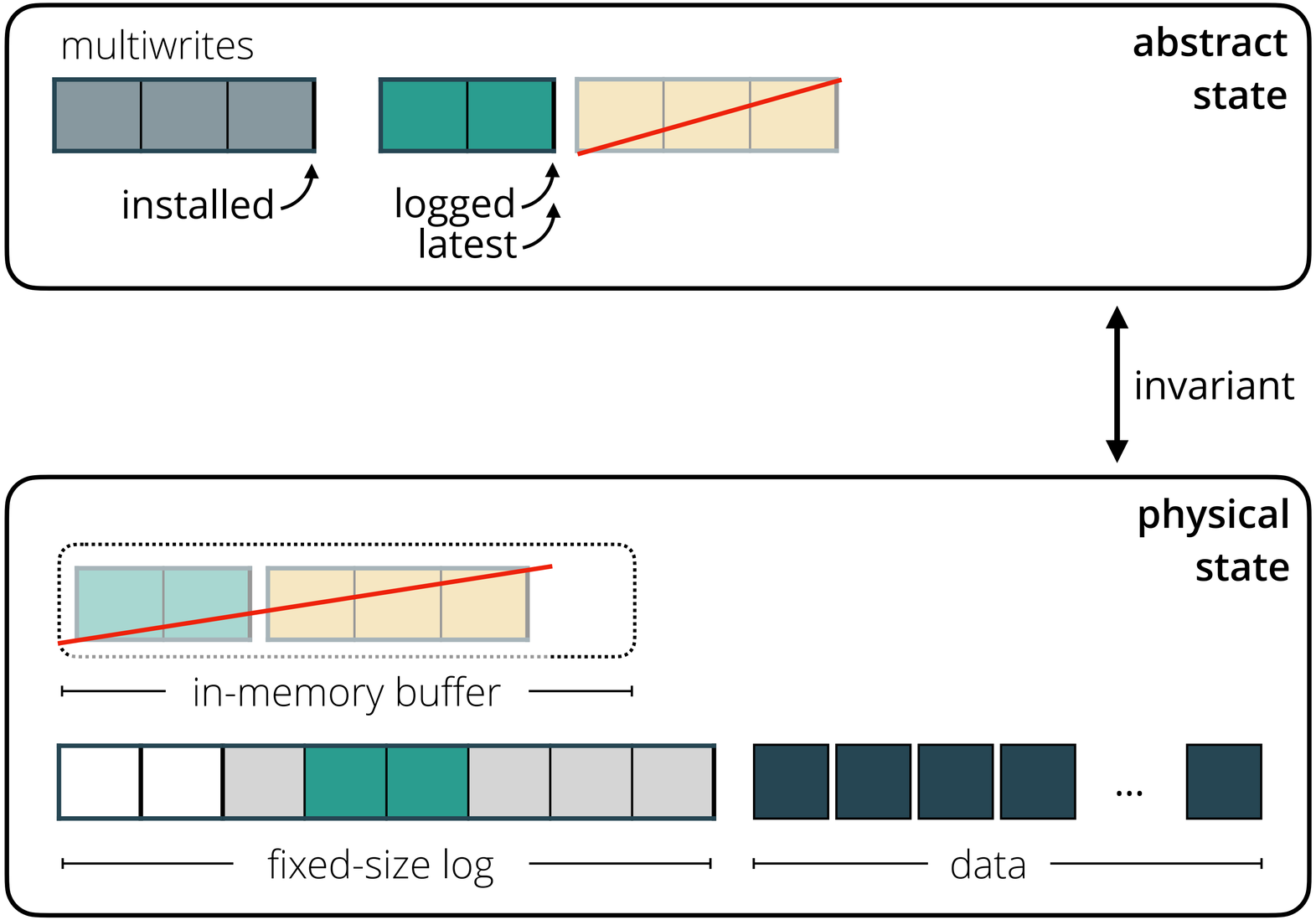
Idea: model WAL as a history of multiwrites



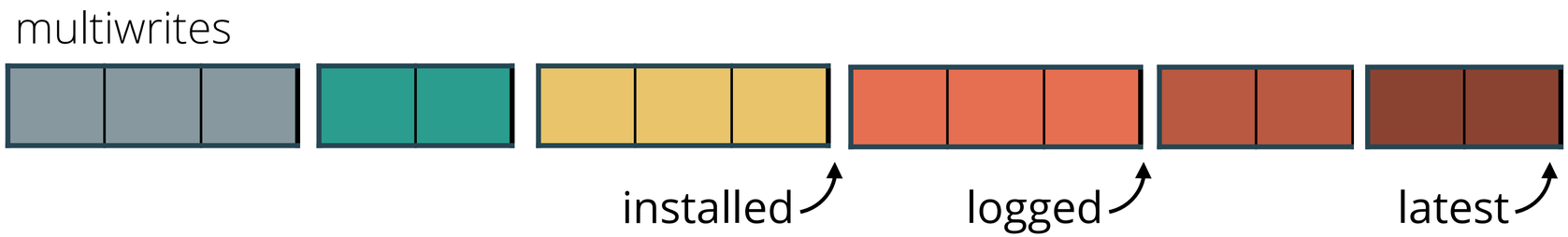
History abstract state crisply expresses atomicity



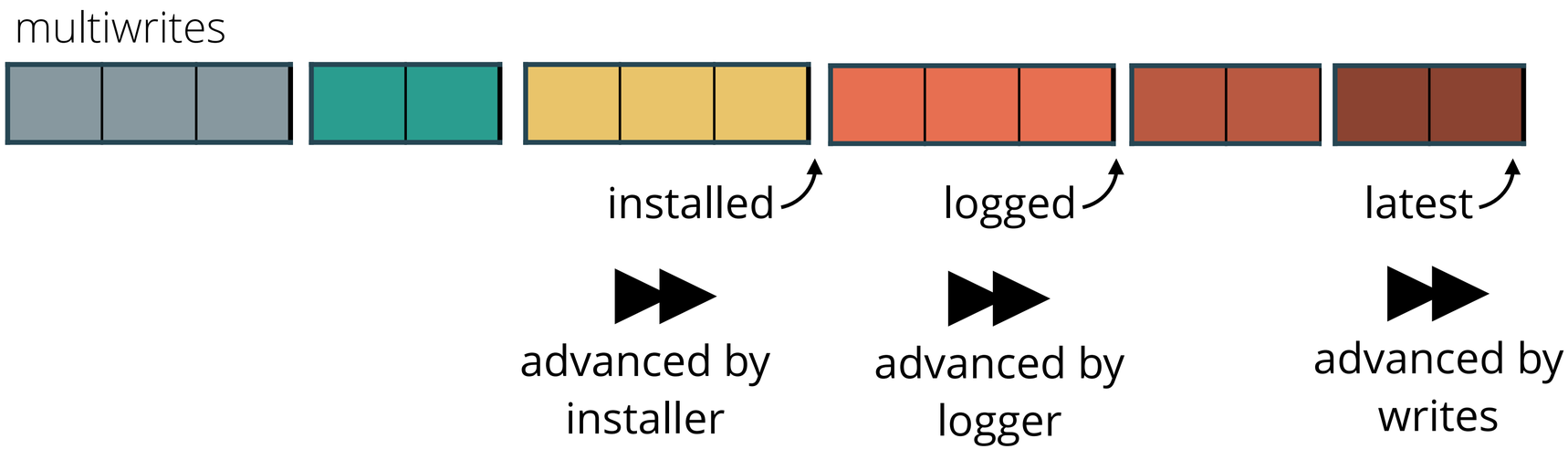
History abstract state crisply expresses atomicity



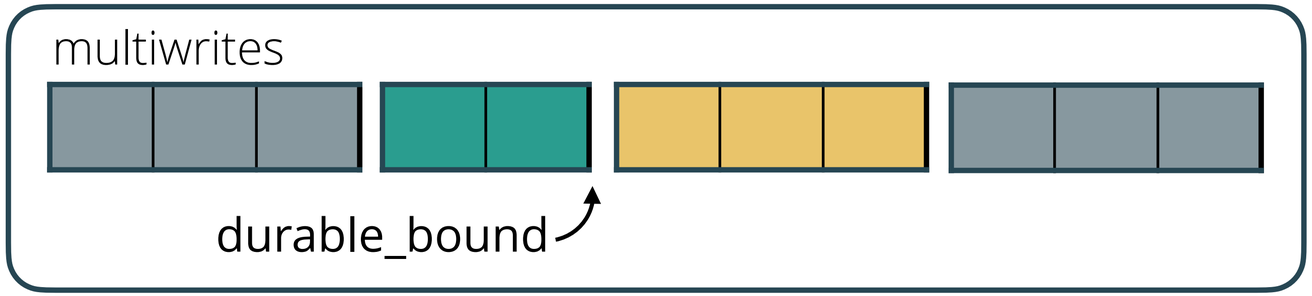
Pointers can all advance concurrently



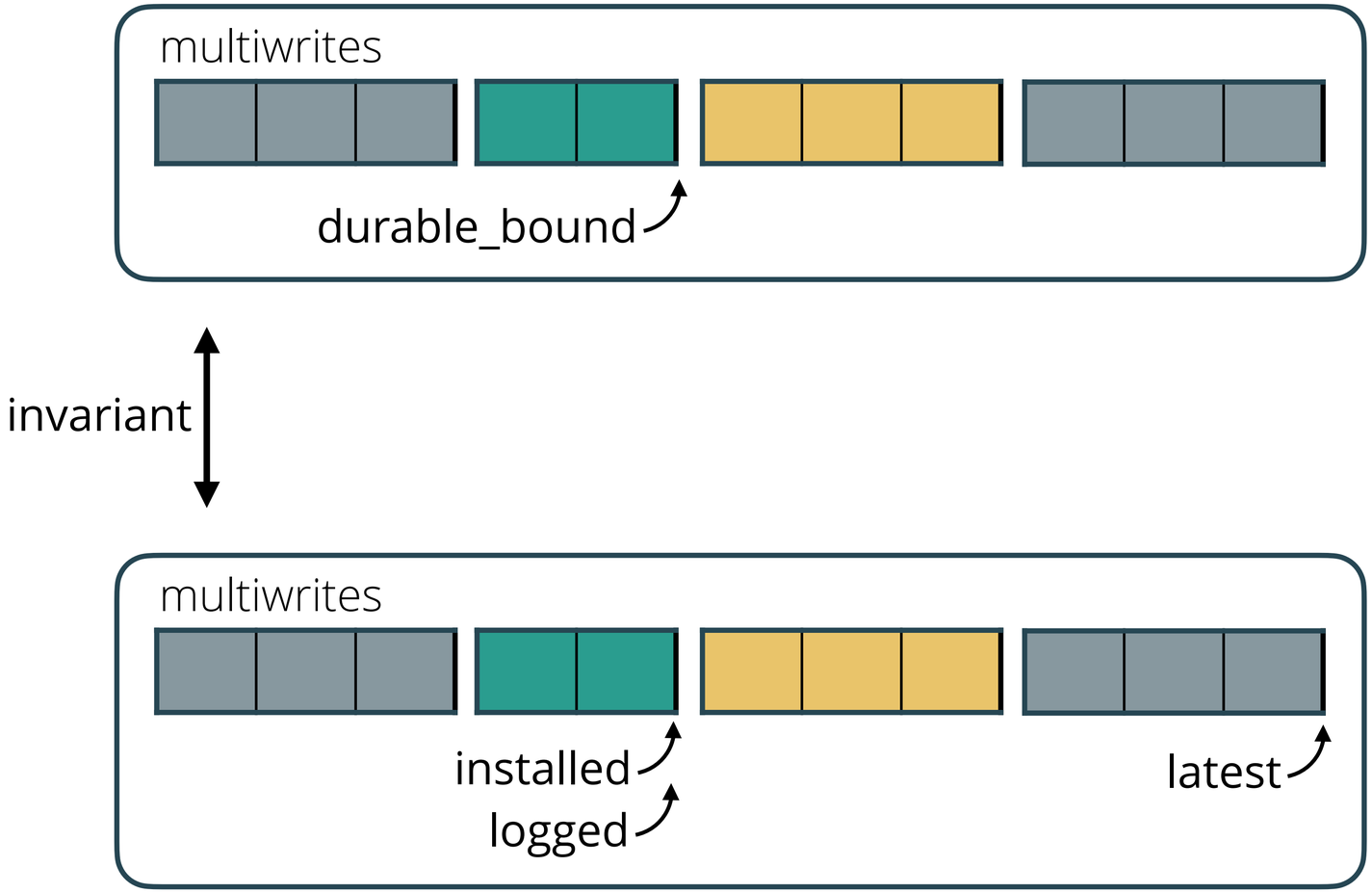
Pointers can all advance concurrently



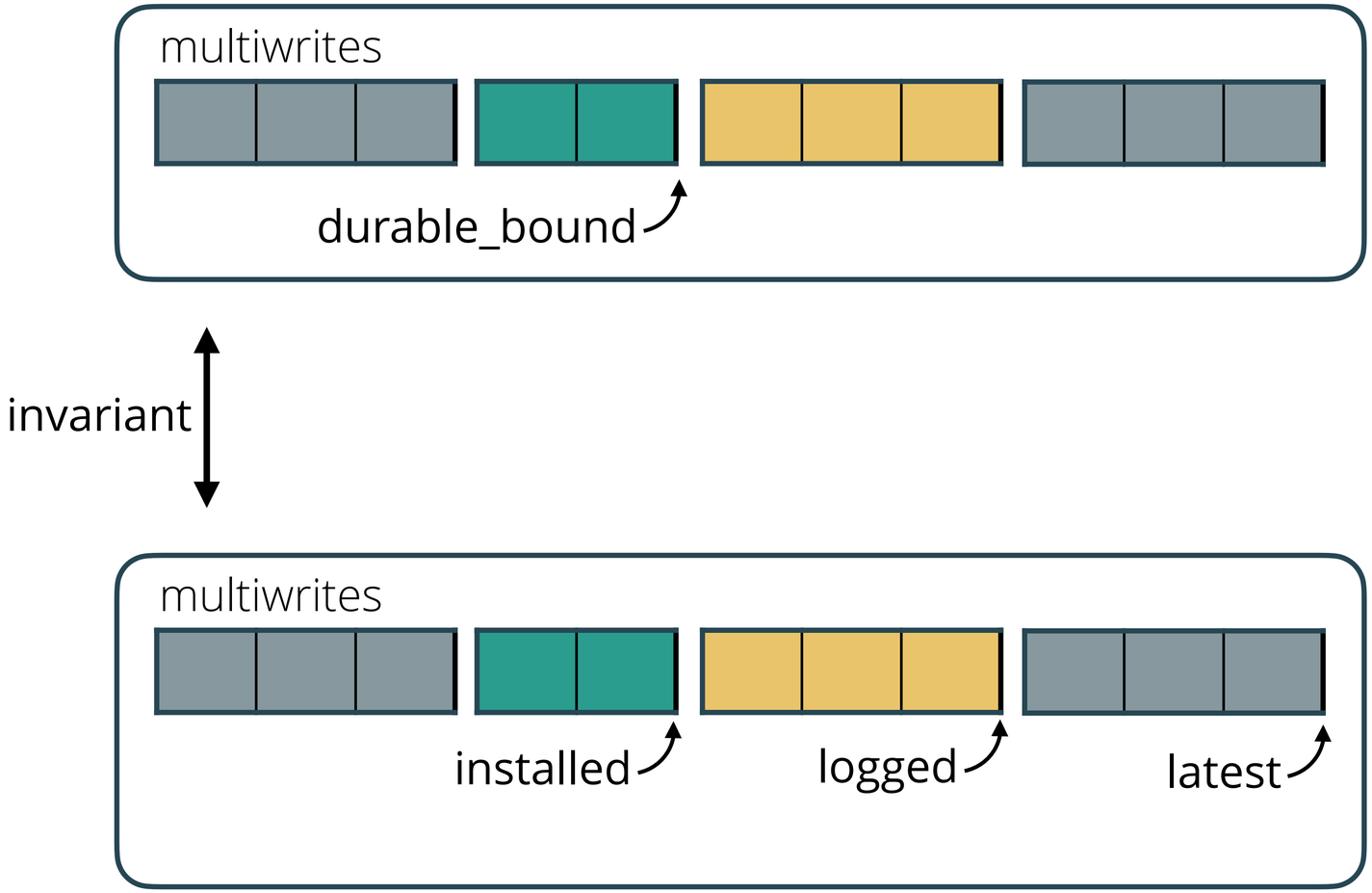
Durable bound hides concurrency for rest of proof



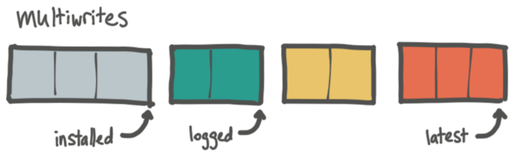
Durable bound hides concurrency for rest of proof



Durable bound hides concurrency for rest of proof



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

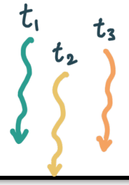
Specification that bridges the two

✓

GoTxn

Transaction system gives atomicity

Crashes



Concurrency

✓

Roadmap

DaisyNFS

File-system code implemented with transactions

Sequential reasoning

Specification for transactions

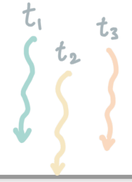
Specification that bridges the two

✓

GoTxn

Transaction system gives atomicity

Crashes



Concurrency

✓

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
Request for Comments: 1813
Category: Informational

B. Callaghan
B. Pawlowski
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.

Defining NFS protocol mathematically

Defining NFS protocol mathematically

```
type FilesysData = map<Ino, File>

datatype File =
  | ByteFile(data: seq<byte>, attrs: Attrs)
  | Directory(dir: map<Path, Ino>, attrs: Attrs)

type Ino = uint64
type Path = seq<byte>
datatype Attrs = Attrs(mode: uint32, …)
```

Defining NFS protocol mathematically

Captures the state of the file system

```
type FilesysData = map<Ino, File>

datatype File =
  | ByteFile(data: seq<byte>, attrs: Attrs)
  | Directory(dir: map<Path, Ino>, attrs: Attrs)

type Ino = uint64
type Path = seq<byte>
datatype Attrs = Attrs(mode: uint32, …)
```

Defining NFS protocol mathematically

Captures the state of the file system

```
type FilesysData = map<Ino, File>

datatype File =
  | ByteFile(data: seq<byte>, attrs: Attrs)
  | Directory(dir: map<Path, Ino>, attrs: Attrs)

type Ino = uint64
type Path = seq<byte>
datatype Attrs = Attrs(mode: uint32, …)
```

```
predicate REMOVE_spec(fs: FilesysData, fs’: FilesysData,
                      args: RemoveArgs, r: RemoveReply)
predicate MKDIR_spec(fs: FilesysData, fs’: FilesysData,
                     args: MkdirArgs, r: MkdirReply)
…
```

Defining NFS protocol mathematically

Captures the state of the file system

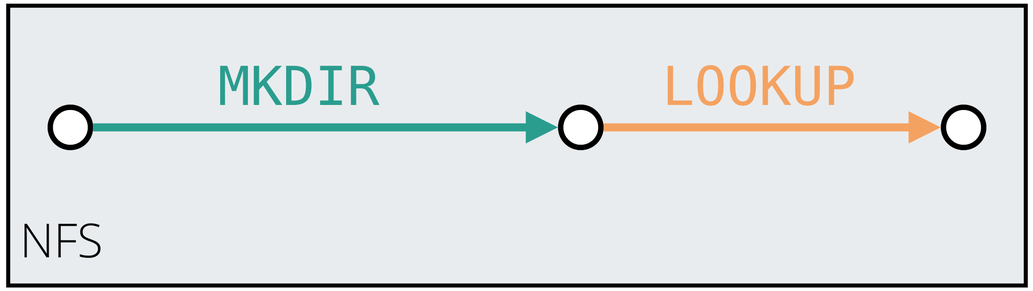
```
type FilesysData = map<Ino, File>

datatype File =
  | ByteFile(data: seq<byte>, attrs: Attrs)
  | Directory(dir: map<Path, Ino>, attrs: Attrs)

type Ino = uint64
type Path = seq<byte>
datatype Attrs = Attrs(mode: uint32, …)
```

Defines what REMOVE is allowed to do

```
predicate REMOVE_spec(fs: FilesysData, fs': FilesysData,
                      args: RemoveArgs, r: RemoveReply)
predicate MKDIR_spec(fs: FilesysData, fs': FilesysData,
                     args: MkdirArgs, r: MkdirReply)
…
```

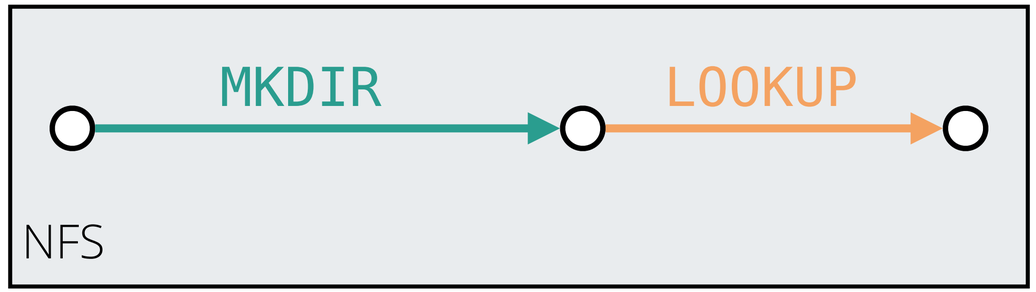
DaisyNFS’s top-level specification

`MKDIR(…)` ‖ `LOOKUP(…)`



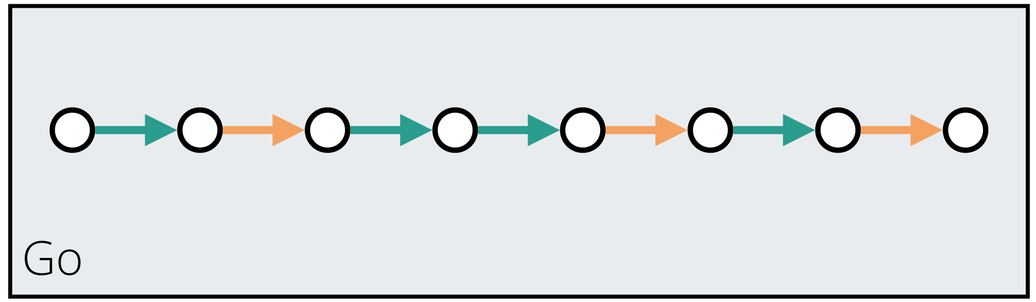
DaisyNFS's top-level specification

`MKDIR(…)` ‖ `LOOKUP(…)`



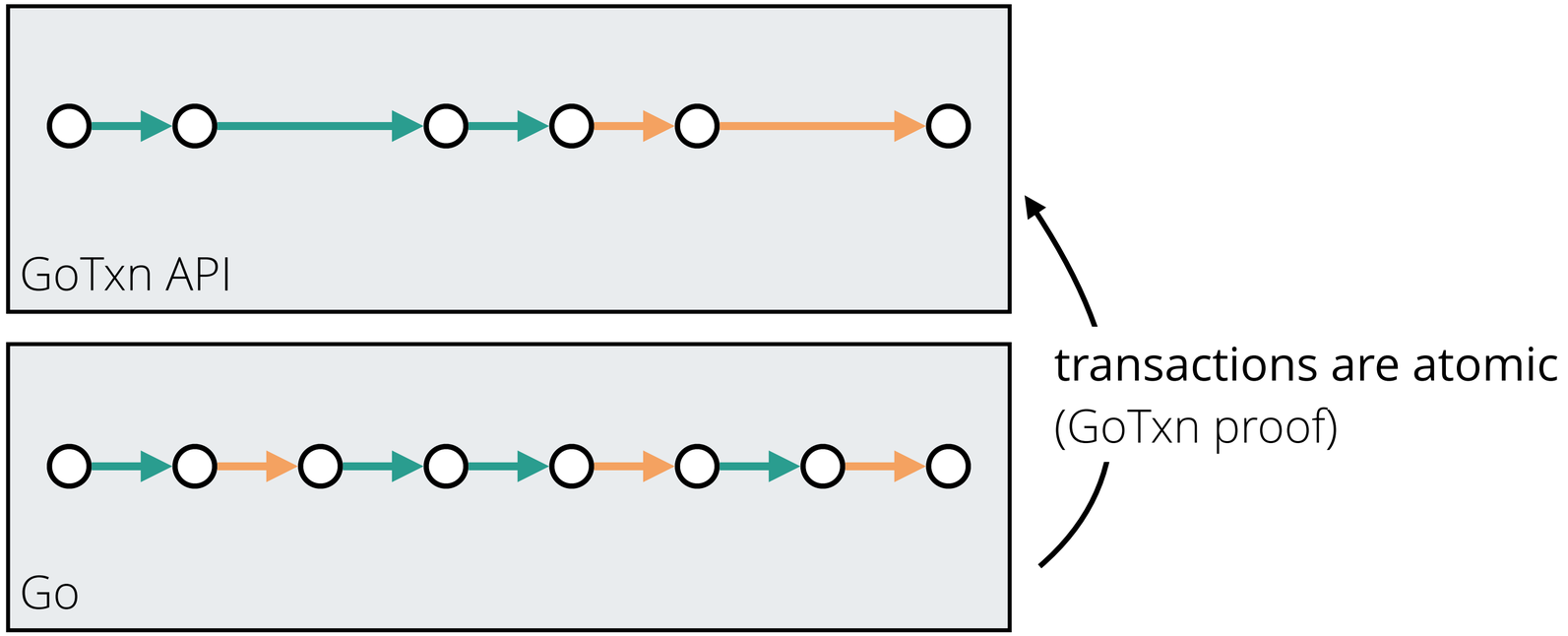
should follow (atomic) NFS specification

Every `daisy-nfsd` concurrent execution…



Proof: compose GoTxn and DaisyNFS proofs

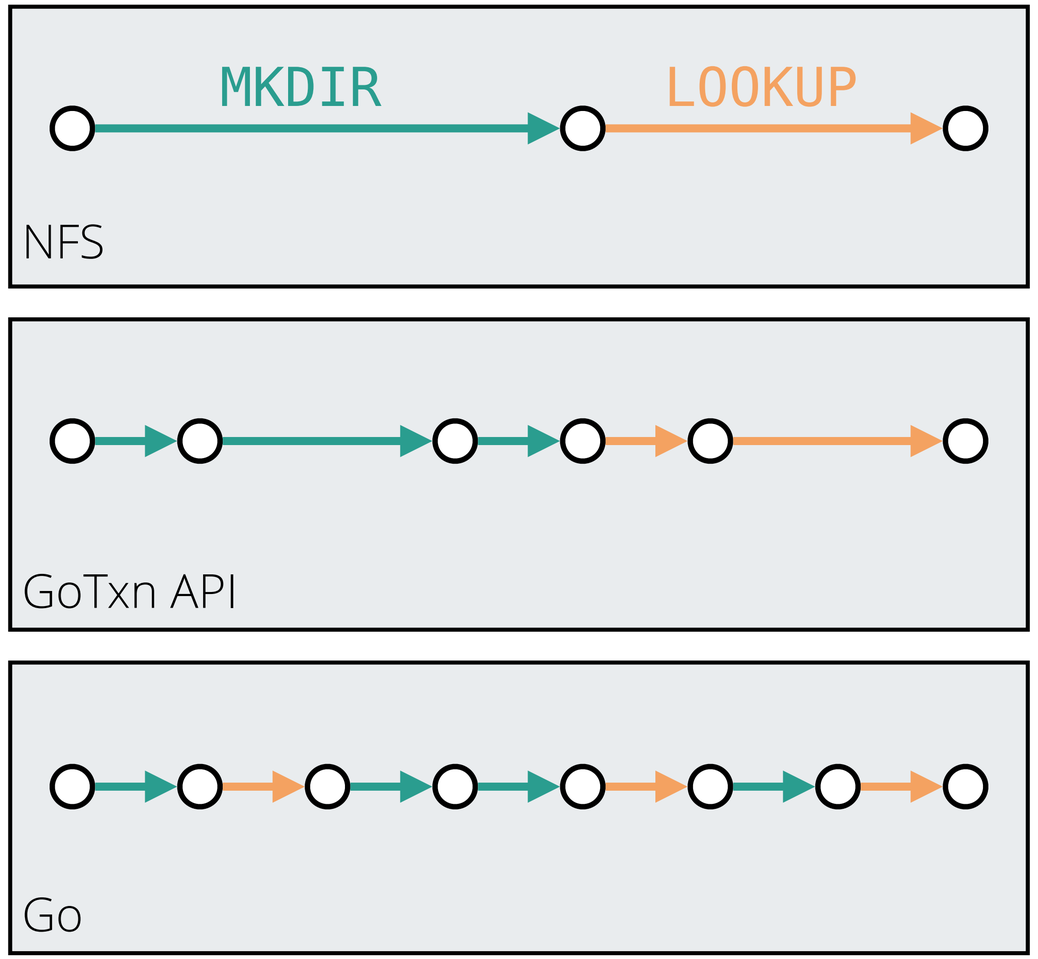
`MKDIR(…)` ‖ `LOOKUP(…)`



`daisy-nfsd` concurrent execution

Proof: compose GoTxn and DaisyNFS proofs

`MKDIR(…)` ‖ `LOOKUP(…)`

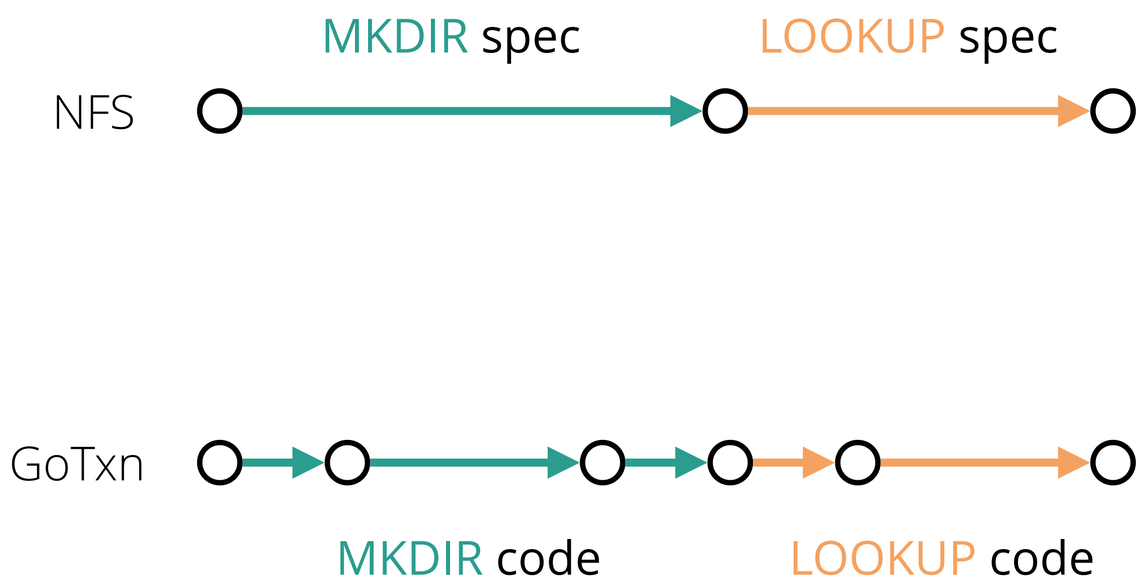


sequential transactions are correct (DaisyNFS proof)

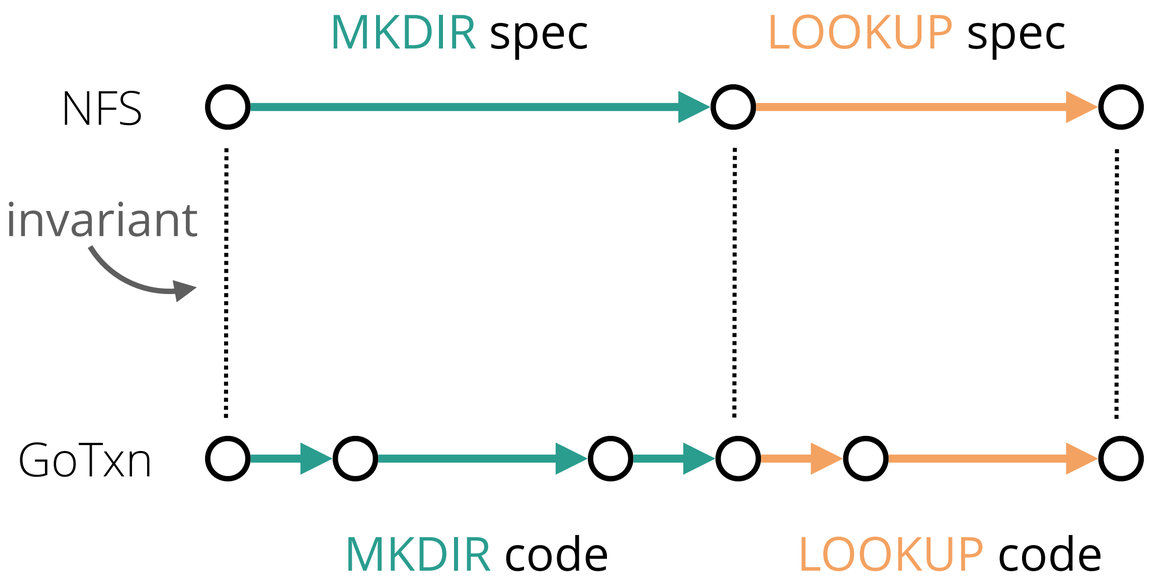
transactions are atomic (GoTxn proof)

`daisy-nfsd` concurrent execution

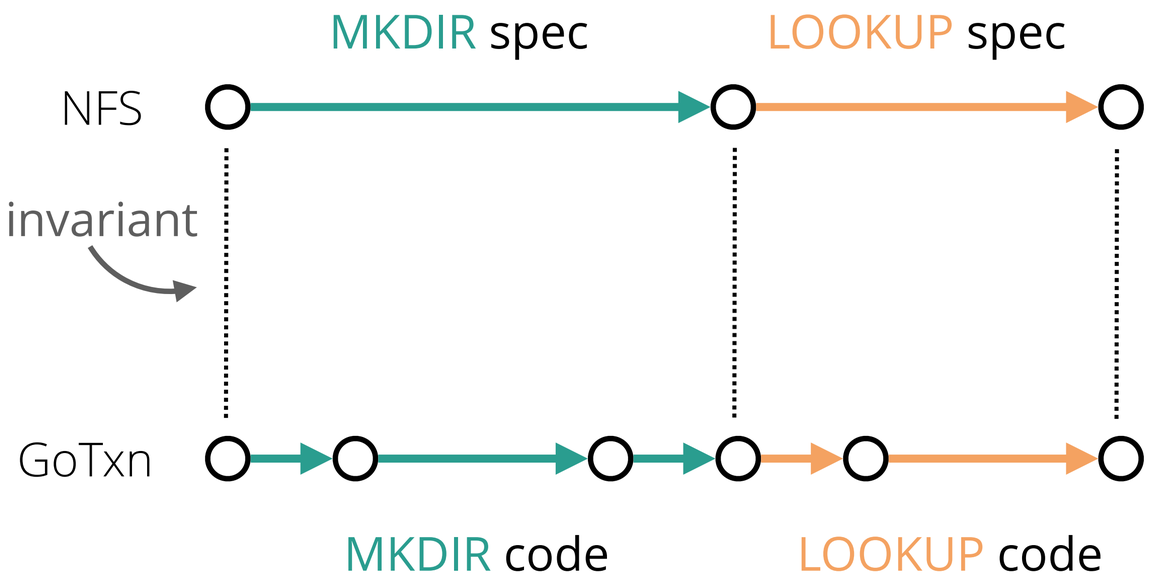
Transactions are proven with sequential reasoning



Transactions are proven with sequential reasoning



Transactions are proven with sequential reasoning



Sequential reasoning is highly automated

Verify operations using Dafny

Dafny (verified)

```
method REMOVE(tx, args)
  returns (r: RemoveReply)
  requires invariant()
  ensures invariant()
  ensures
    REMOVE_spec(old(fs), fs, args, r)
```

Verify operations using Dafny

Dafny (verified)

```
method REMOVE(tx, args)
  returns (r: RemoveReply)
  requires invariant()
  ensures invariant()
  ensures
    REMOVE_spec(old(fs), fs, args, r)
```

```
func REMOVE(args) RemoveReply {
  tx := Begin()
  r := fs.REMOVE(tx, args)
  tx.Commit()

  return r
}
```

Go (unverified)

Implementing freeing using bounded transactions

Dafny (verified)

```
method REMOVE(tx, args)
  returns (r: RemoveReply)
  requires invariant()
  ensures invariant()
  ensures
    REMOVE_spec(old(fs), fs, args, r)
```

```
func REMOVE(args) RemoveReply {
  tx := Begin()
  r := fs.REMOVE(tx, args)
  tx.Commit()

  return r
}
```

Go (unverified)

Implementing freeing using bounded transactions

Dafny (verified)

```
method REMOVE(tx, args)
  returns (r: RemoveReply)
  requires invariant()
  ensures invariant()
  ensures
    REMOVE_spec(old(fs), fs, args, r)

method ZeroFreeSpace(ino)
  requires invariant()
  ensures invariant()
  ensures fs == old(fs)
```

```
func REMOVE(args) RemoveReply {
  tx := Begin()
  r := fs.REMOVE(tx, args)
  tx.Commit()

  return r
}
```

Go (unverified)

Implementing freeing using bounded transactions

Dafny (verified)

```
method REMOVE(tx, args)
  returns (r: RemoveReply)
  requires invariant()
  ensures invariant()
  ensures
    REMOVE_spec(old(fs), fs, args

method ZeroFreeSpace(ino)
  requires invariant()
  ensures invariant()
  ensures fs == old(fs)
```

```
func REMOVE(args) RemoveReply {
  tx := Begin()
  r := fs.REMOVE(tx, args)
  tx.Commit()
  go fs.ZeroFreeSpace(r.ino)
  return r
}
```

Logical no-op but physically recovers space

Implementing freeing using bounded transactions

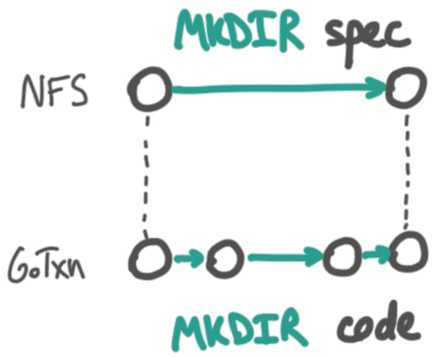
Dafny (verified)

```
method REMOVE(tx, args)
  returns (r: RemoveReply)
  requires invariant()
  ensures invariant()
  ensures
    REMOVE_spec(old(fs), fs, args

method ZeroFreeSpace(ino)
  requires invariant()
  ensures invariant()
  ensures fs == old(fs)
```

```
func REMOVE(args) RemoveReply {
  tx := Begin()
  r := fs.REMOVE(tx, args)
  tx.Commit()
  go fs.ZeroFreeSpace(r.ino)
  return r
}
```

Logical no-op but physically recovers space

Summary

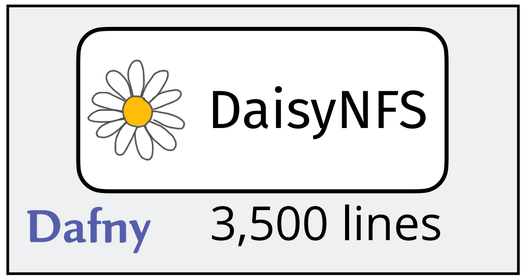


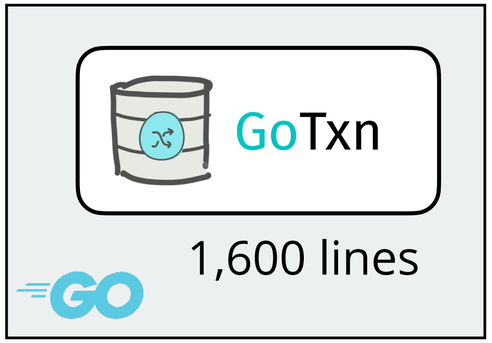
Sequential reasoning for concurrent system

Formalized RFC 1813

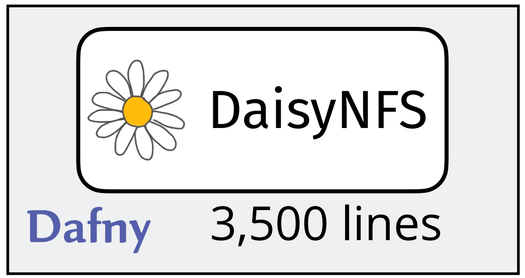
Fit operations into fixed-size transactions

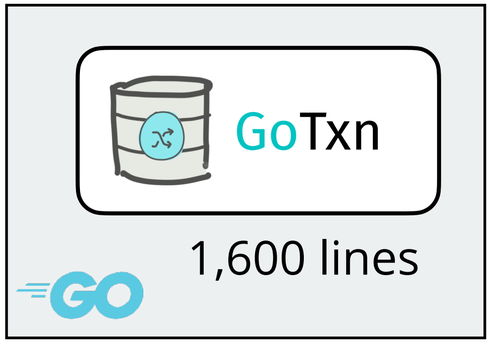
Implementation: code and verification

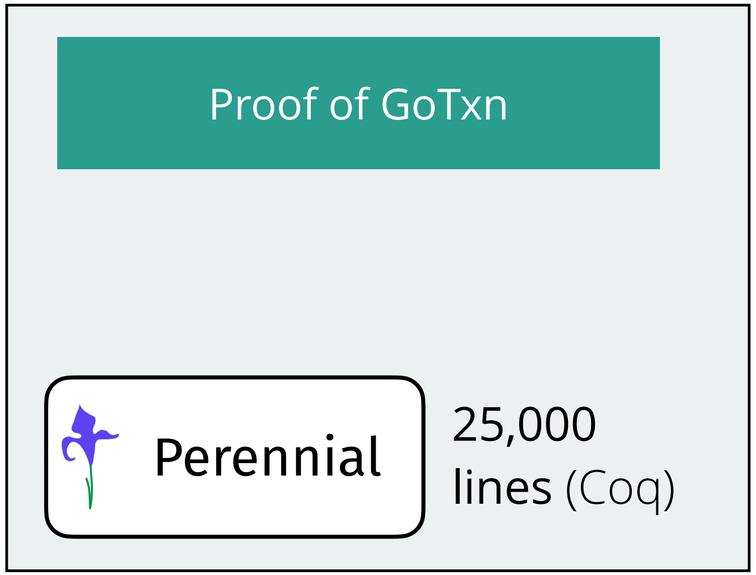




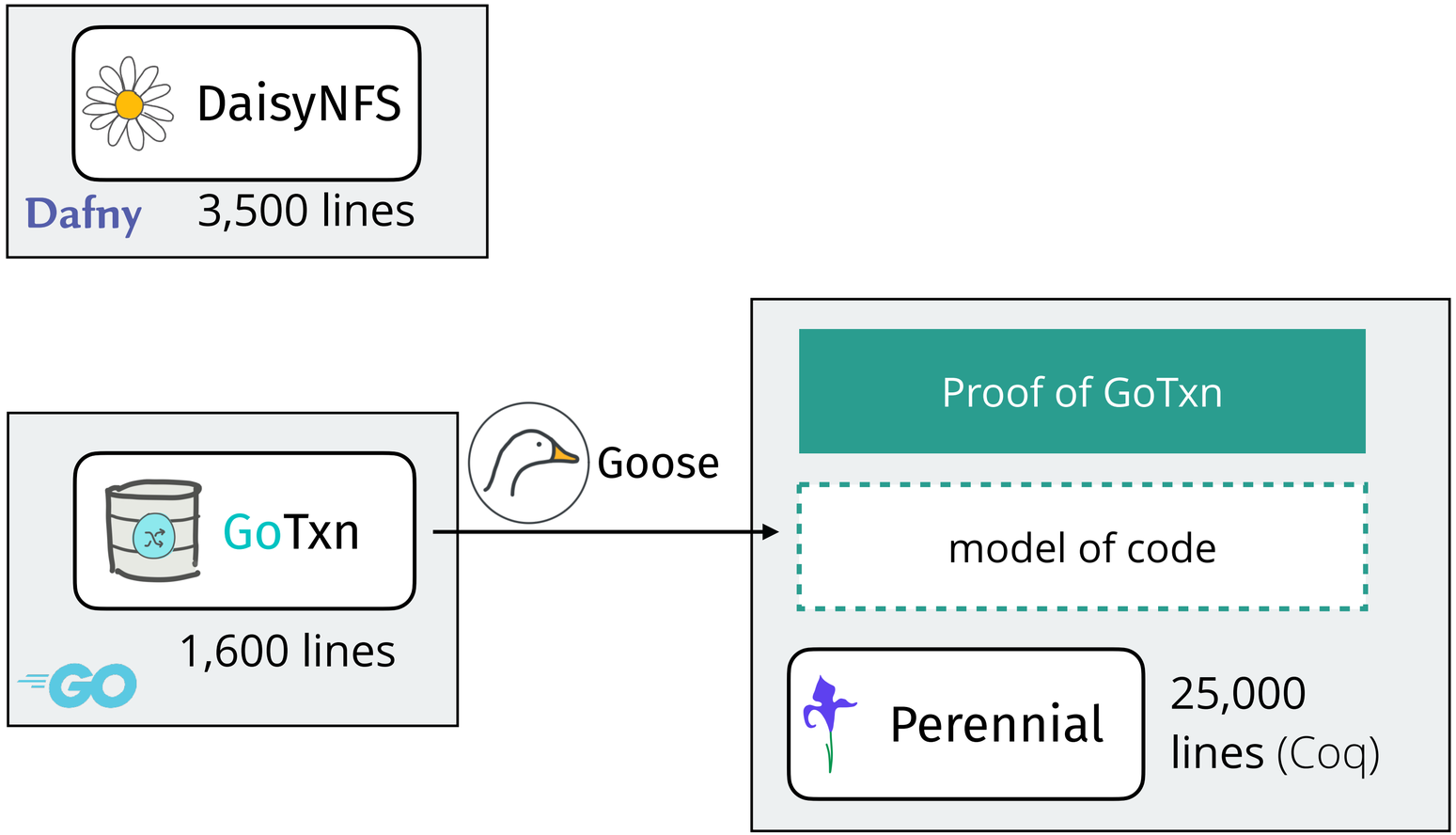
Implementation: code and verification



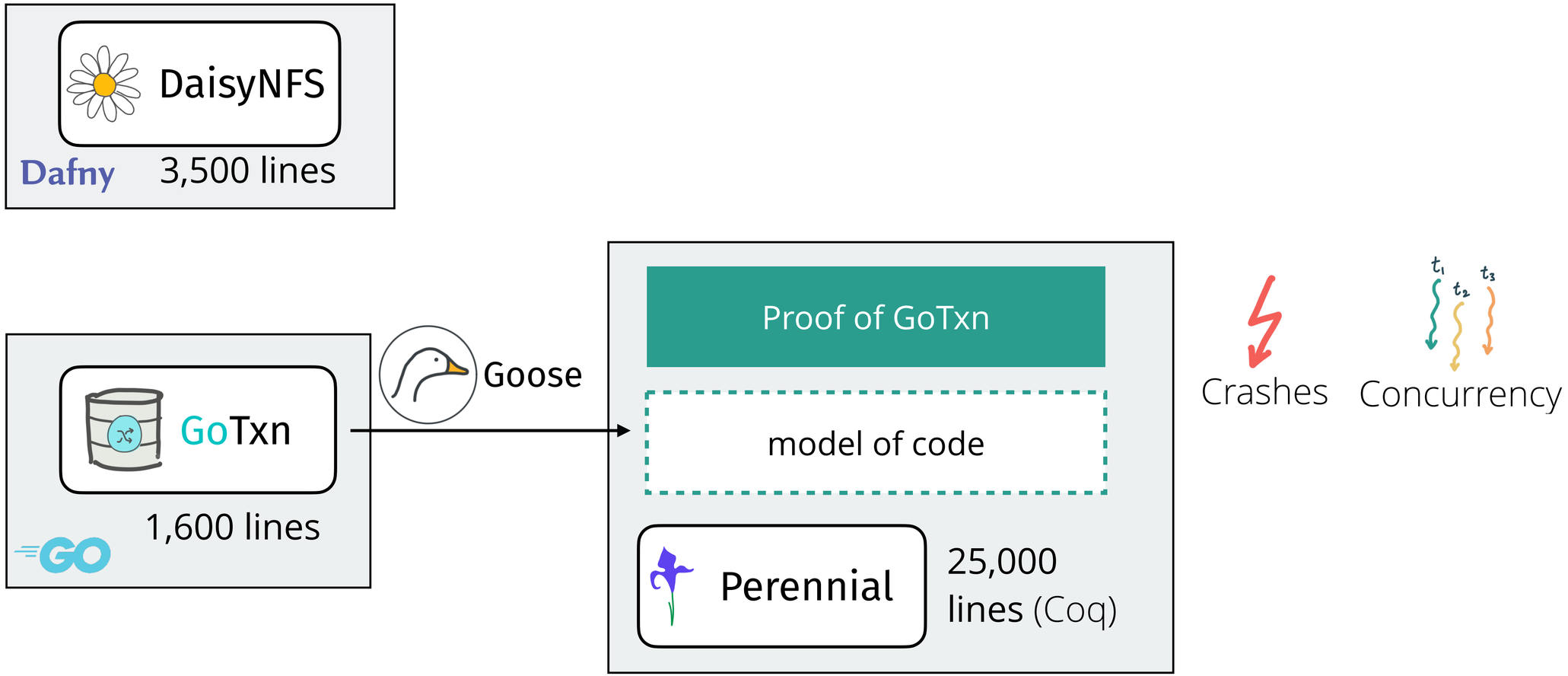




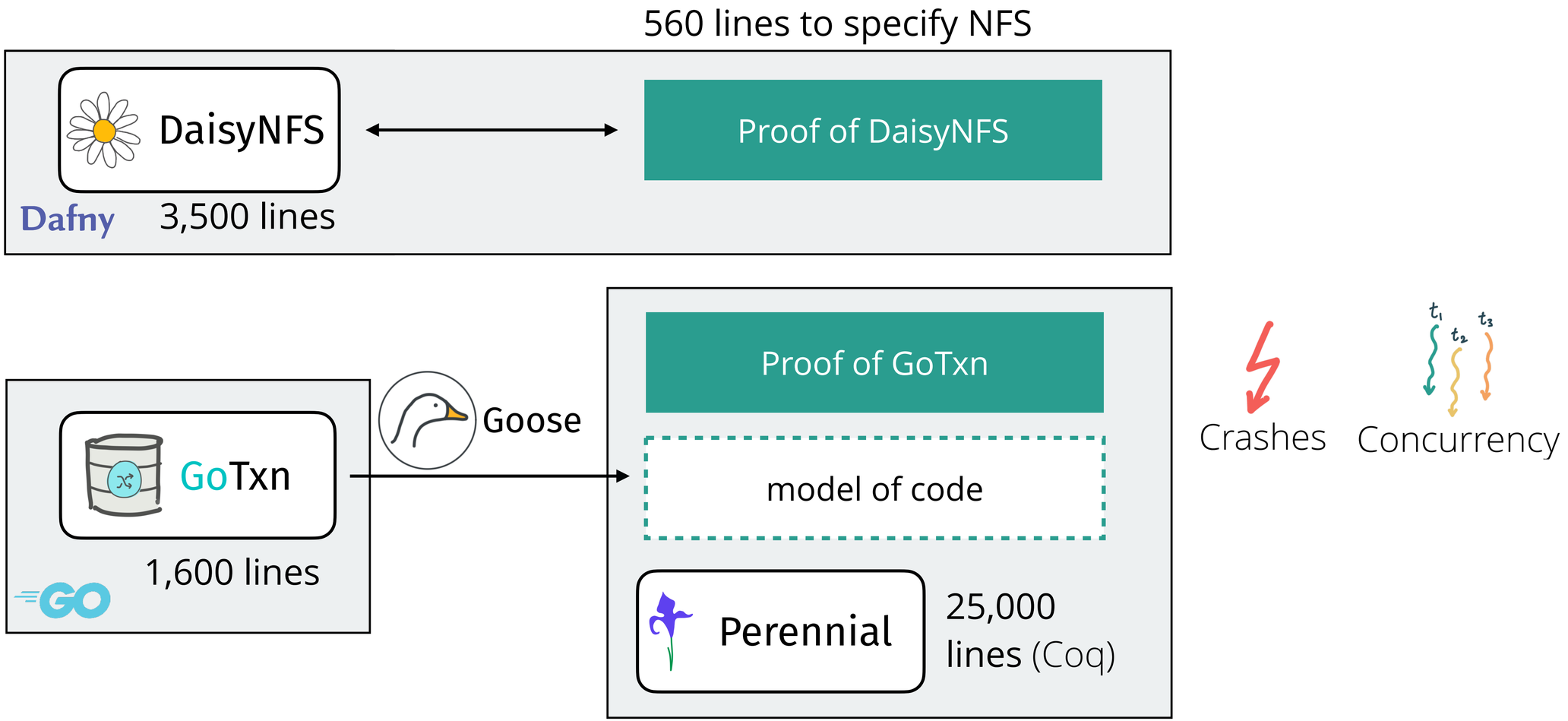
Implementation: code and verification



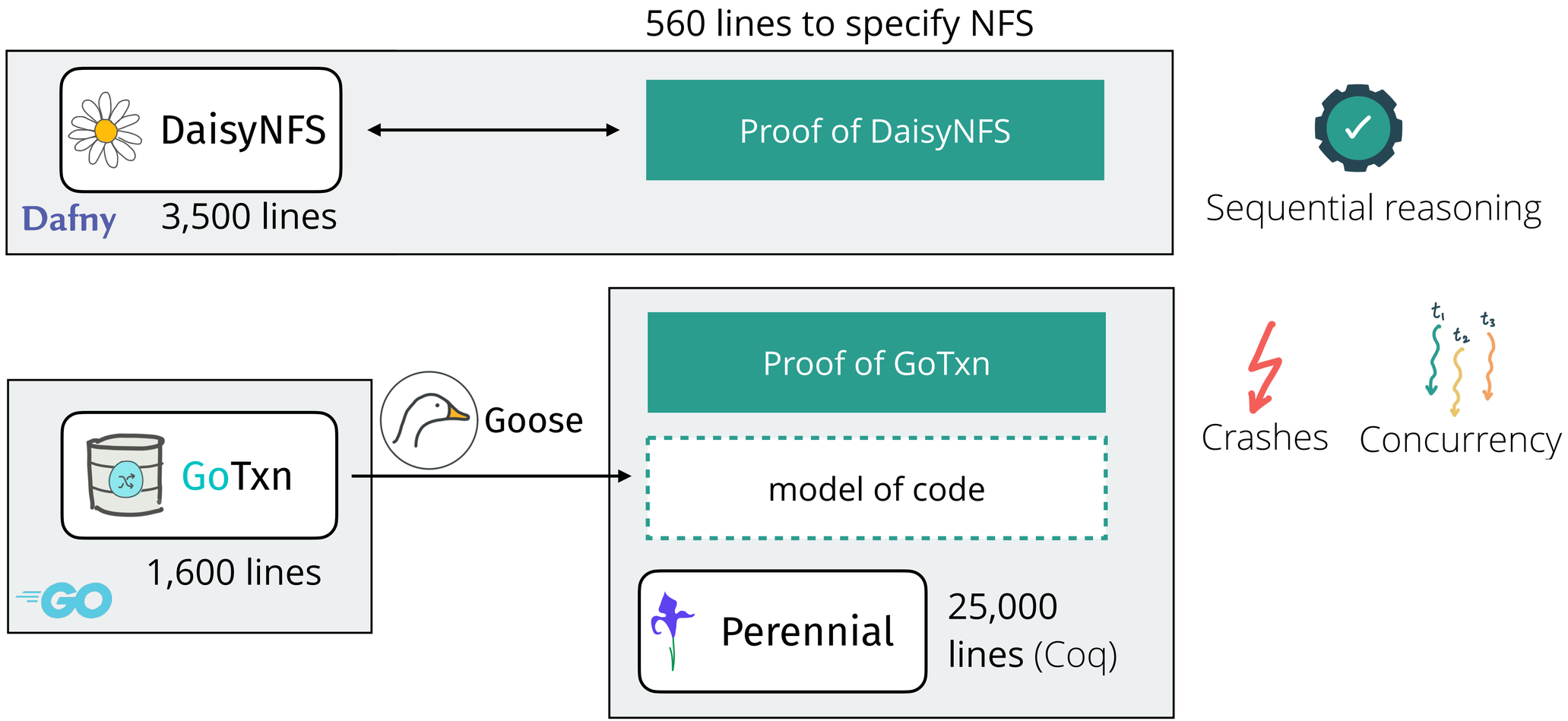
Implementation: code and verification



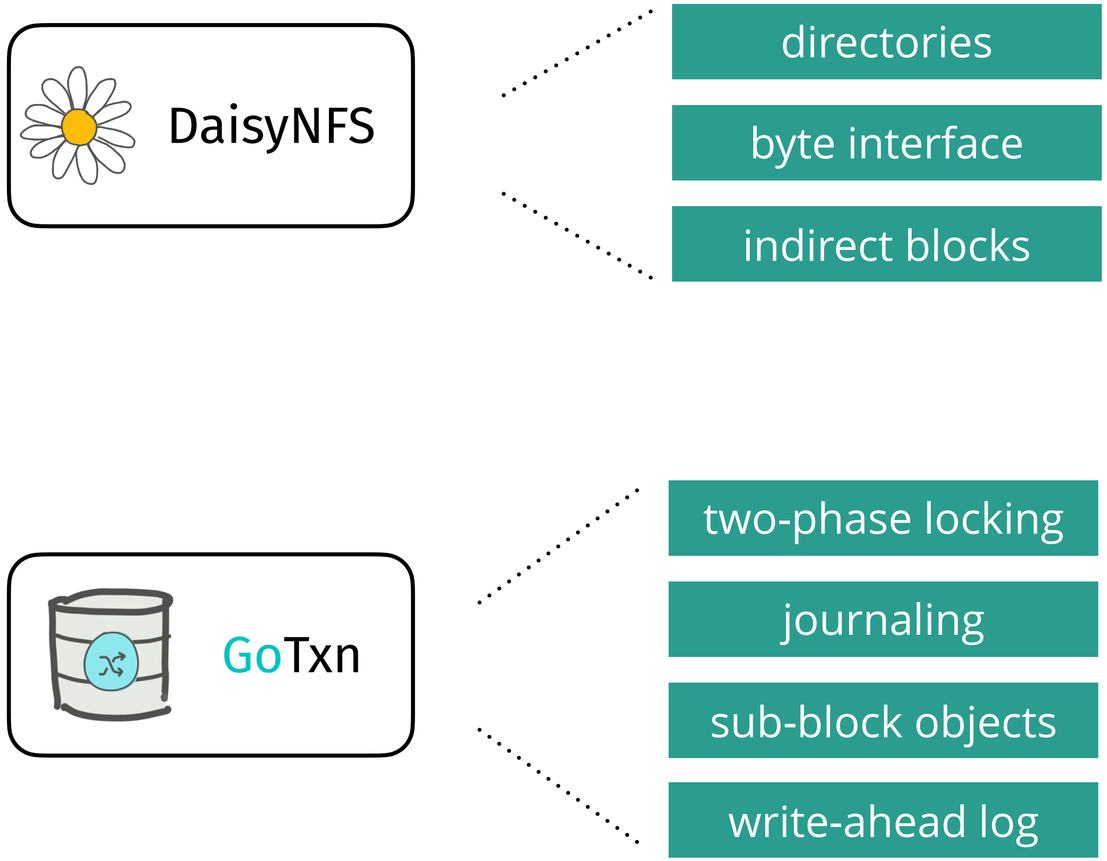
Implementation: code and verification



Implementation: code and verification



Implementation: code



Implementation: code

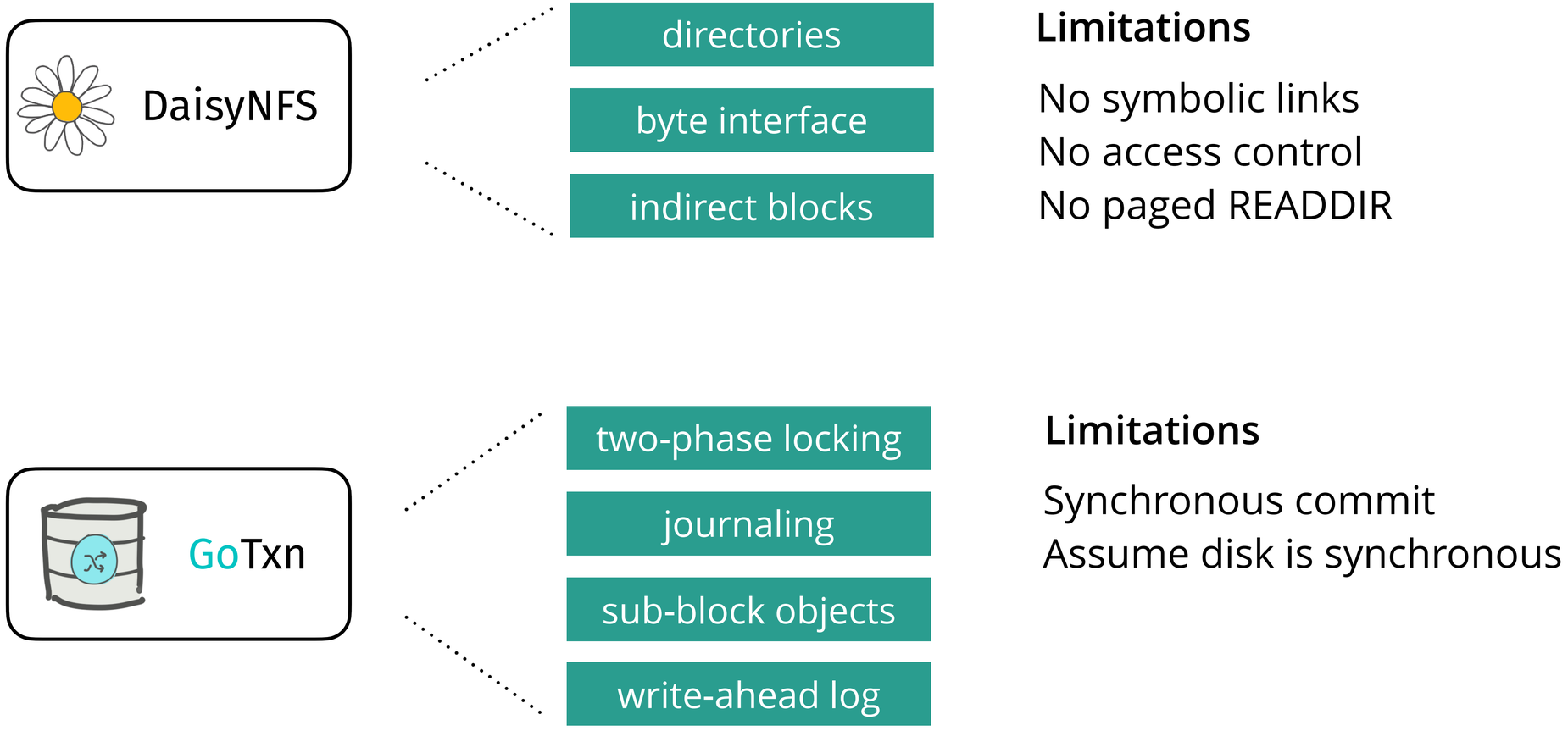
- directories
- byte interface
- indirect blocks

Limitations

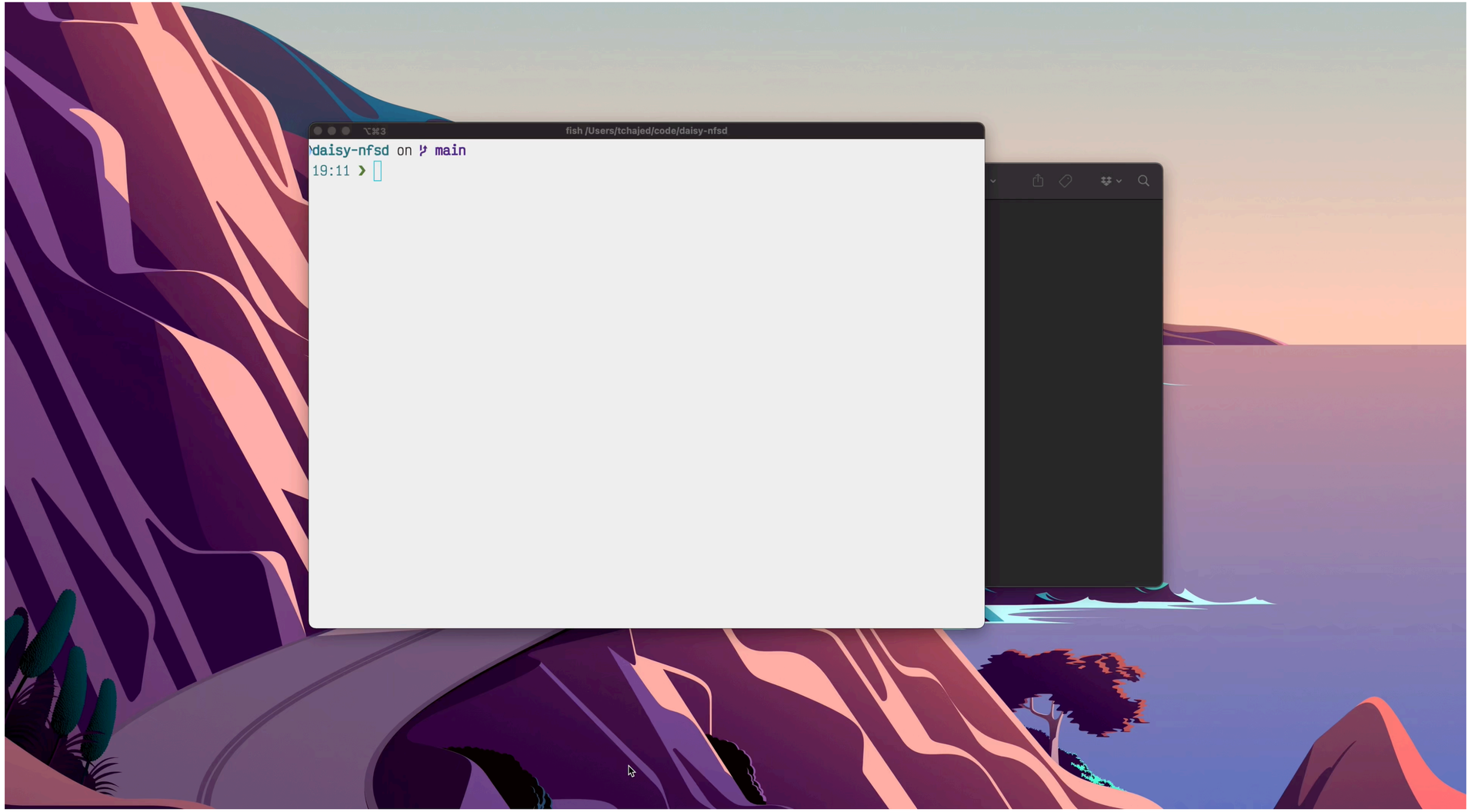
No symbolic links
No access control
No paged READDIR

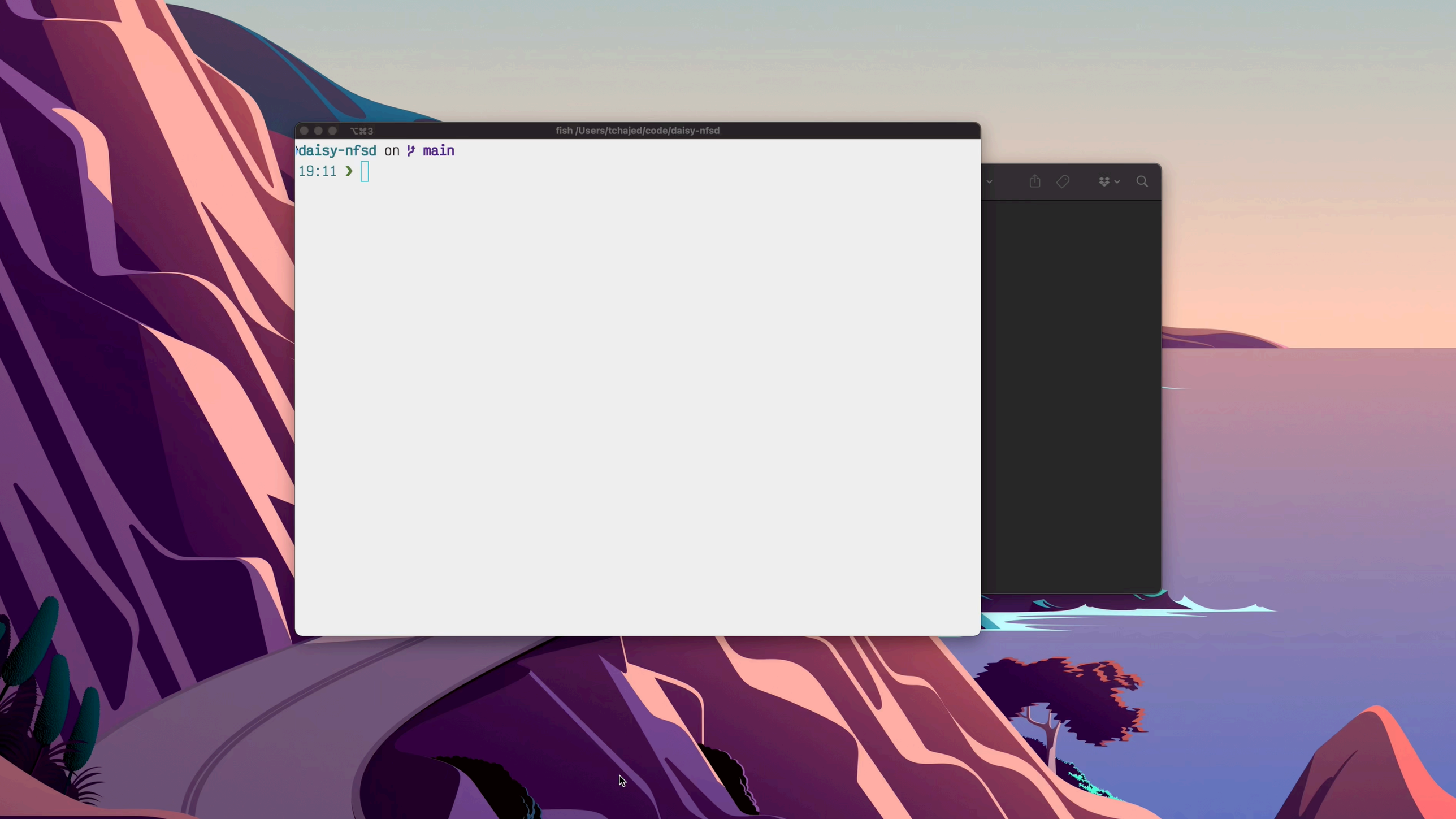
- two-phase locking
- journaling
- sub-block objects
- write-ahead log

Implementation: code



DaisyNFS is a real file system




⌥⌘3
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)

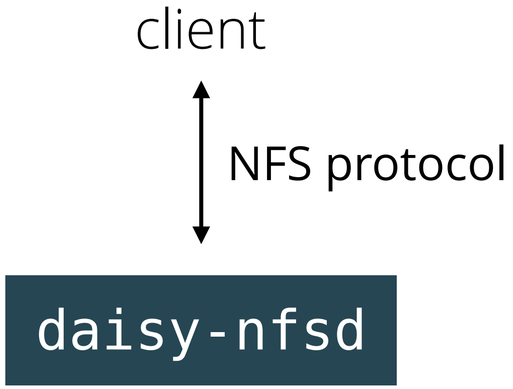
GoTxn greatly reduces proof overhead

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

2x proof:code

20x proof:code

Assumptions in the DaisyNFS proof



Theorem: the server correctly implements the NFS protocol.

Trusted computing base (TCB)

Unverified code (e.g., network protocol)

Dafny specification of NFS

Goose accurately models Go

GoTxn spec in Dafny is faithful

Testing methodologies

Unit test Dafny support libraries

Compare Goose model to Go executions

Run fsstress and fsx-linux test suites

CrashMonkey for crash testing [OSDI '18]

Symbolic execution over NFS server

Bugs found in unverified code and spec

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

File handle parser panics if wrong length

Didn’t find bugs in verified parts

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

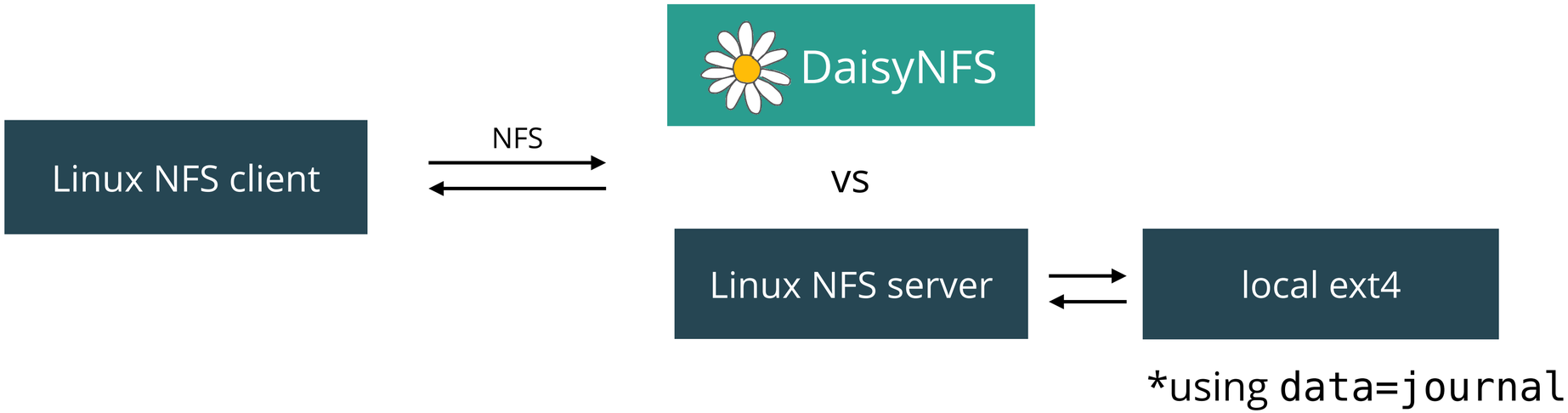
Bugs found in unverified code and spec

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

Bugs found in unverified code and spec

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

Compare against Linux NFS server with ext4

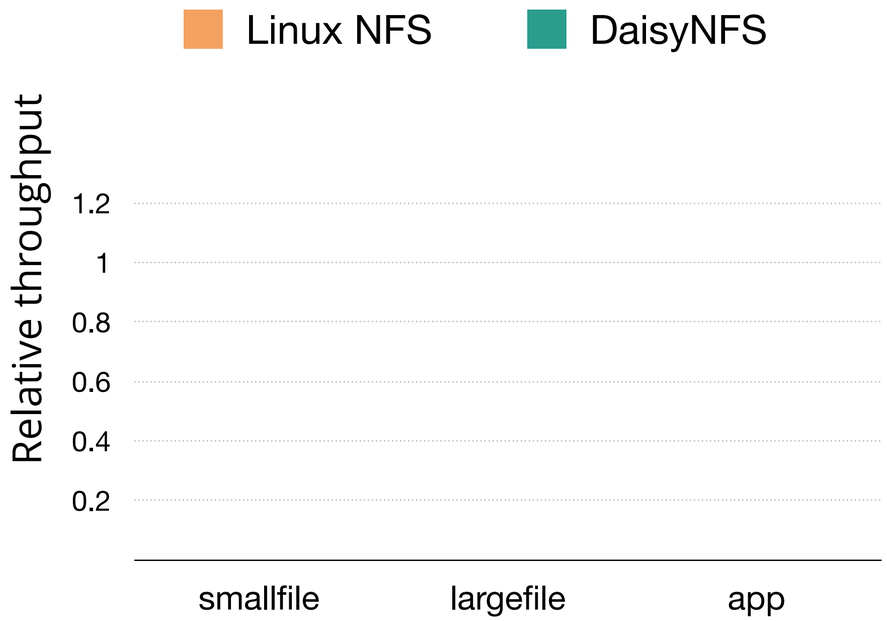


Performance evaluation setup

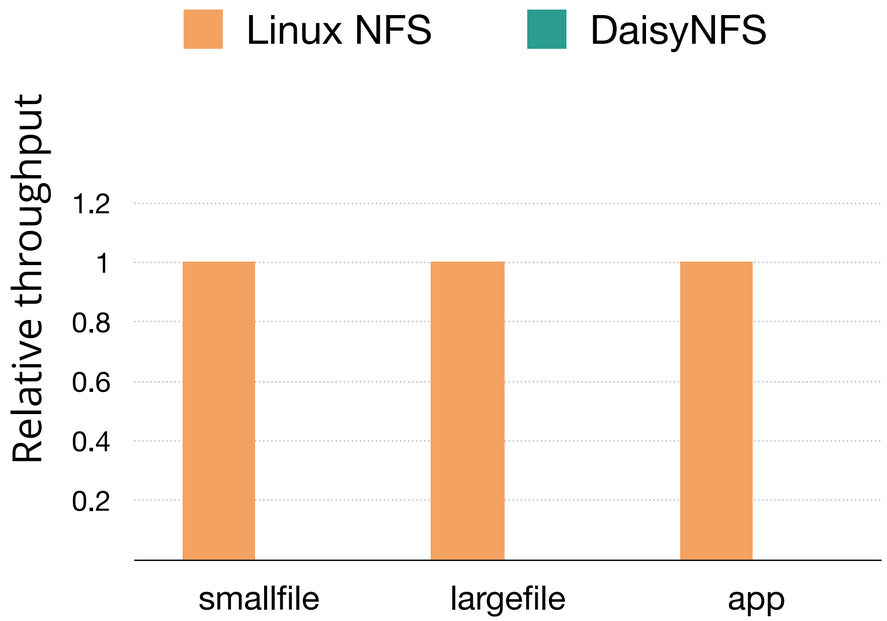
Hardware: i3.metal instance
36 cores at 2.3GHz

Benchmarks:

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

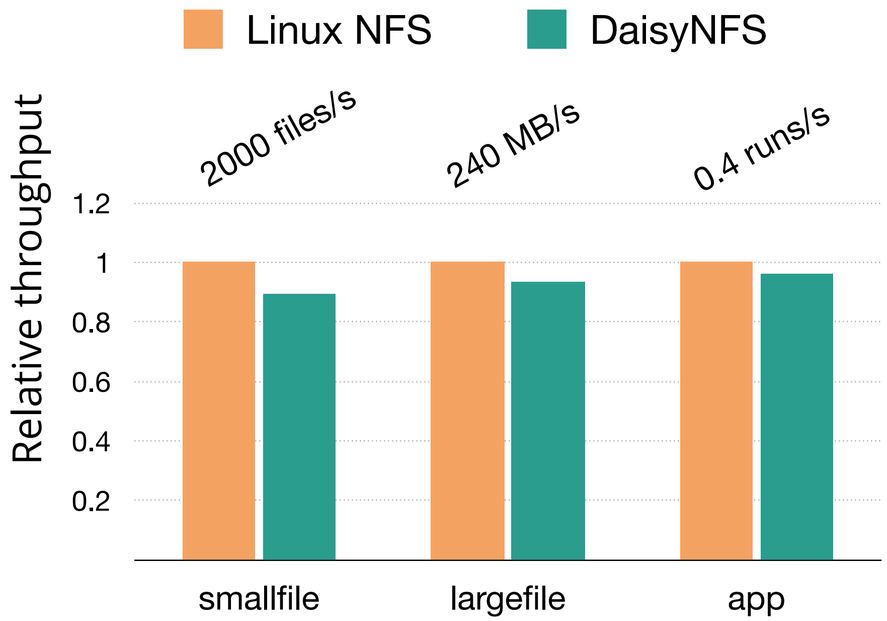


Compare DaisyNFS throughput to Linux, running on an in-memory disk

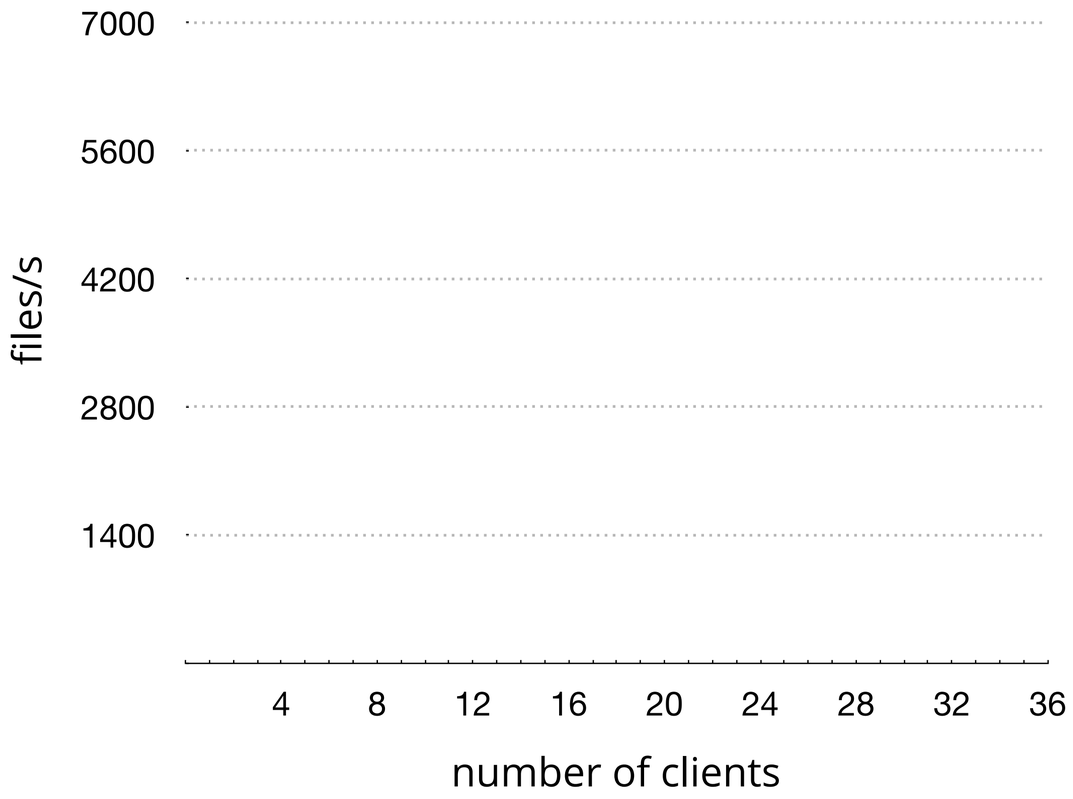


Compare DaisyNFS throughput to Linux, running on an in-memory disk

DaisyNFS gets good performance with a single client

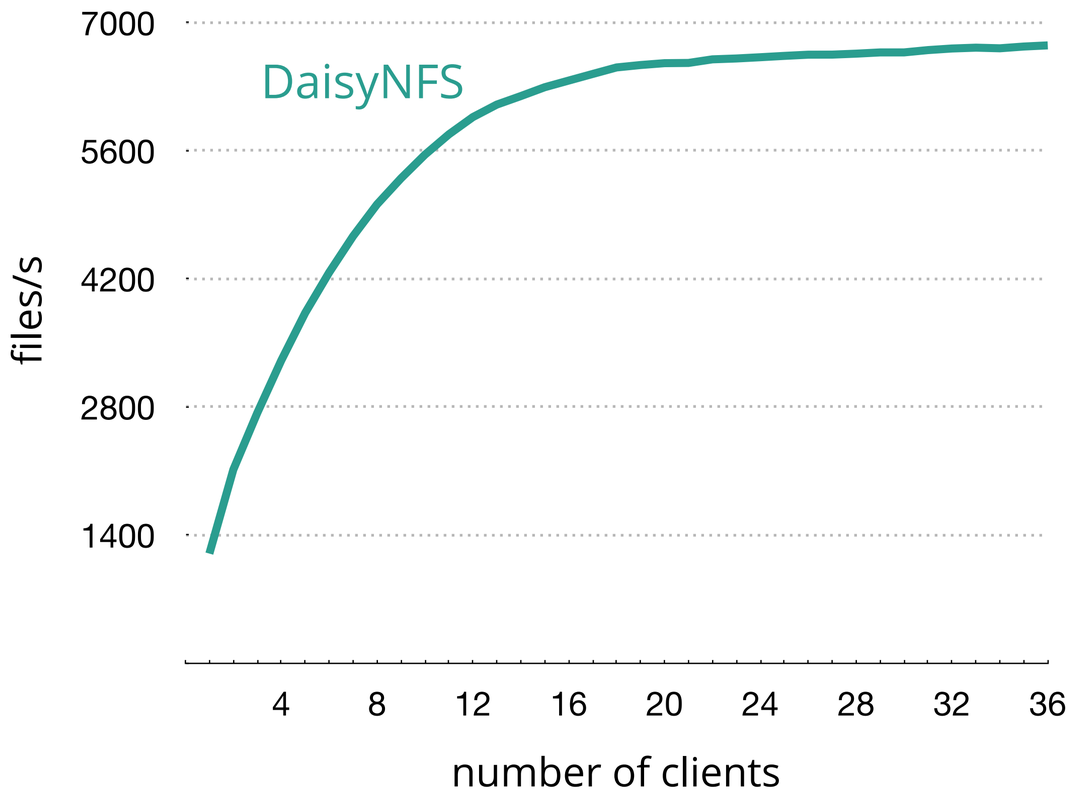


Compare DaisyNFS throughput to Linux, running on an in-memory disk



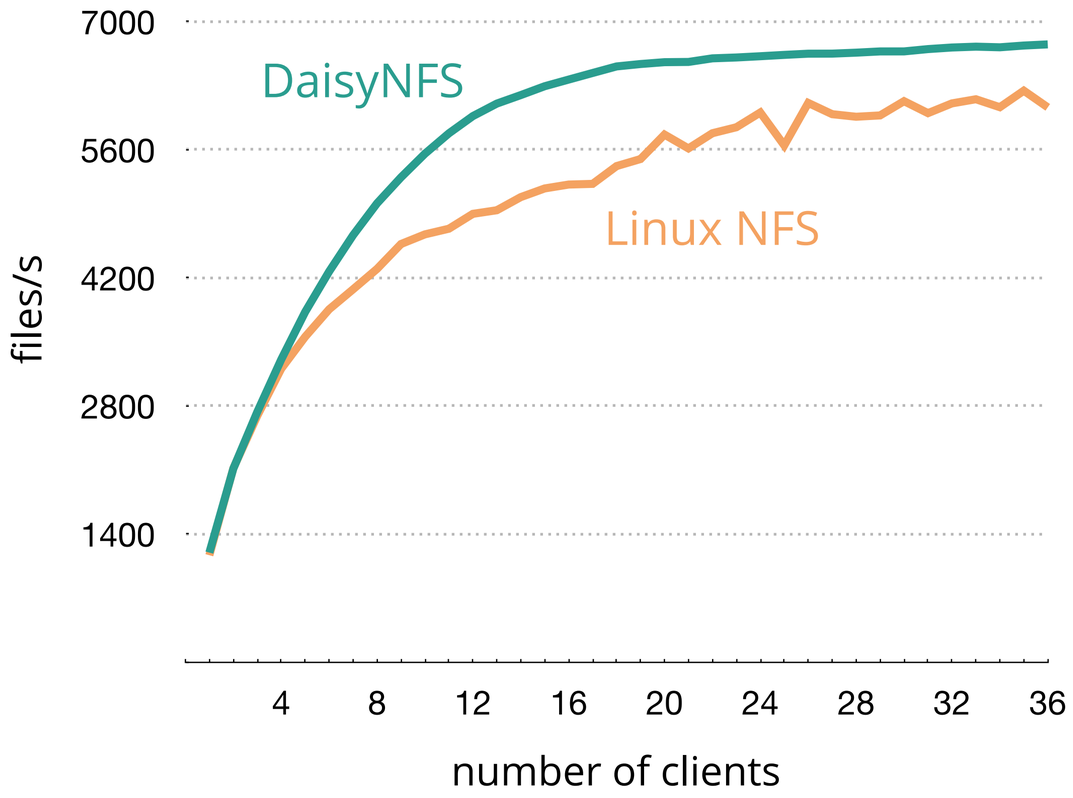
Run smallfile with many clients on an NVMe SSD

DaisyNFS can take advantage of multiple clients

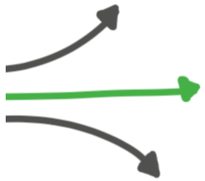


Run smallfile with many clients on an NVMe SSD

DaisyNFS can take advantage of multiple clients



Run smallfile with many clients on an NVMe SSD



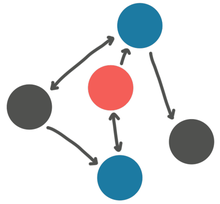
Future directions

Imagine a future of reliable systems software

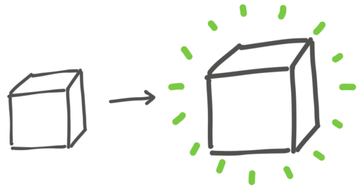
File systems

Security

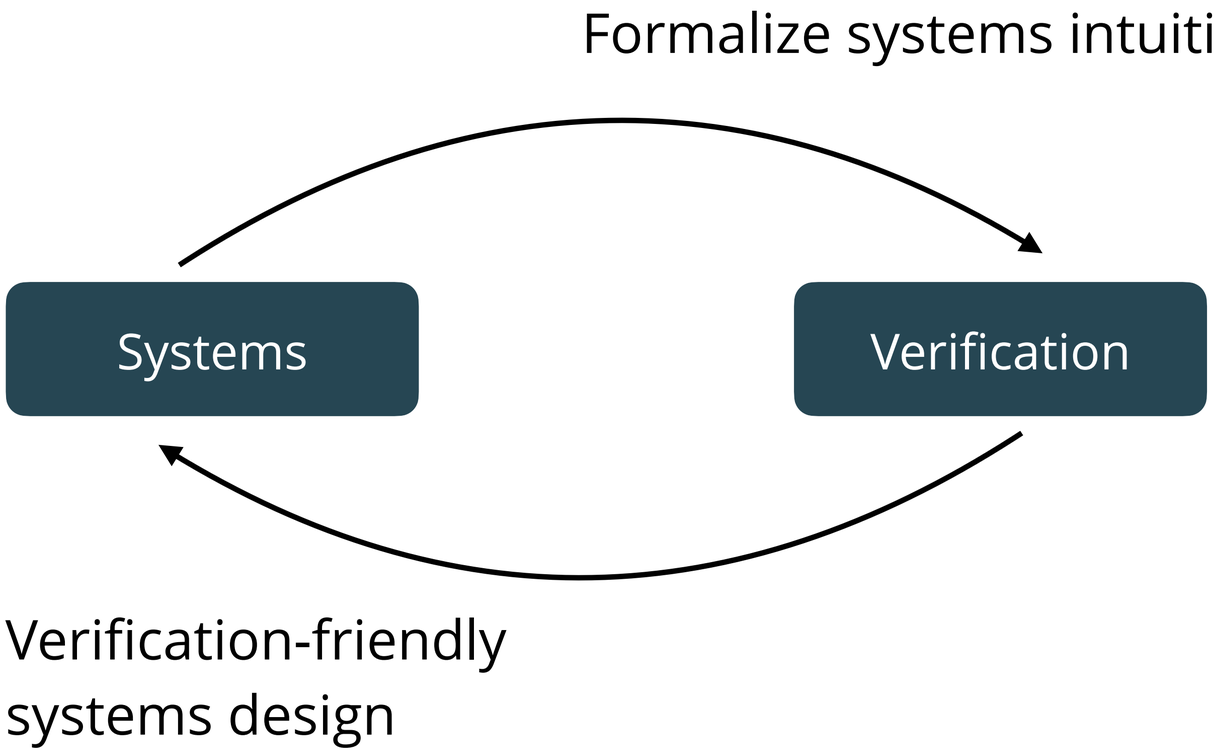
Distributed systems

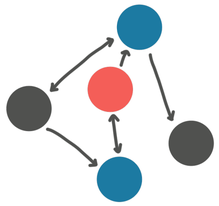


Software upgrades



My research approach

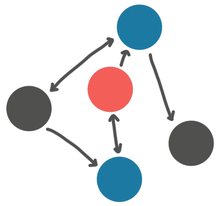




Distributed systems are complex and multi-tiered

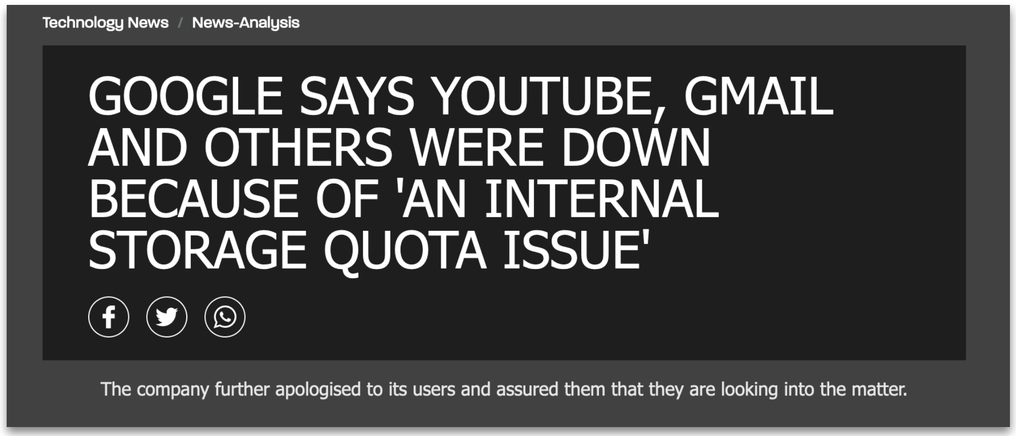
Reliability is important in cloud services

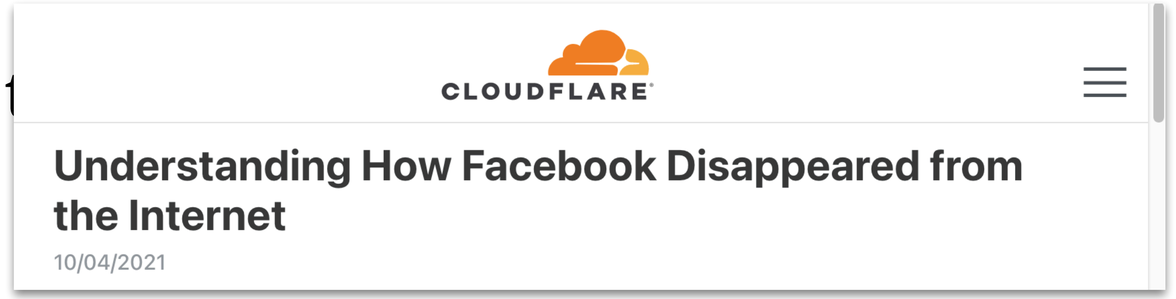
Bugs cause systems to go down

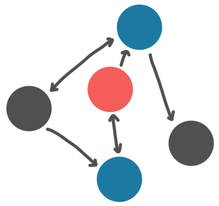


Distributed systems are complex and multi-tiered

ud services





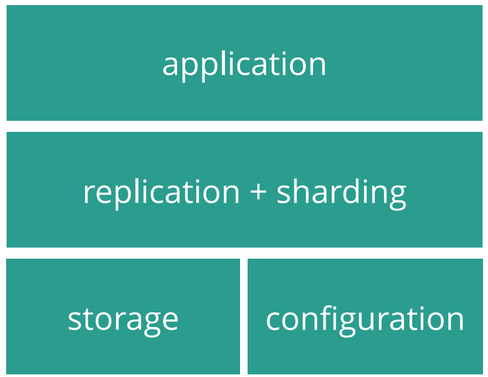


Distributed systems are complex and multi-tiered

Reliability is important in cloud services

Bugs cause systems to go down

Goal: prove a multi-service distributed system



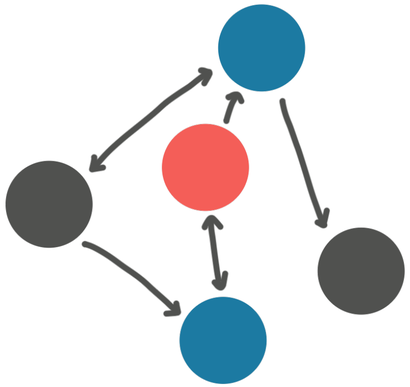
Real systems are made from many smaller components

Proof burden should be manageable

Approach: alternate protocol proofs to RPC-style implementations

```
reply, err := RemoteCall(…)
if err != nil {
  if timeout(err) { … }
  // handle other errors
}
// normal processing
```

Implementation uses
remote procedures



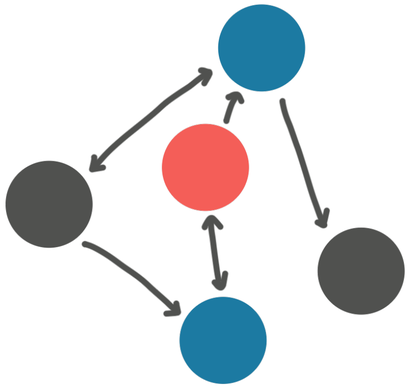
Protocol reasoning is over
whole system

Approach: alternate protocol proofs to RPC-style implementations

```
reply, err := RemoteCall(…)
if err != nil {
  if timeout(err) { … }
  // handle other errors
}
// normal processing
```

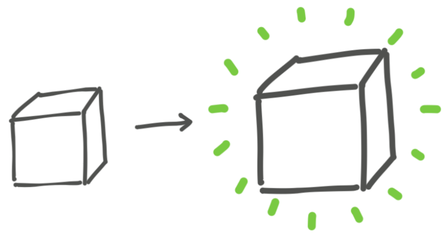
Implementation uses remote procedures

Specification should bridge the two



Protocol reasoning is over whole system

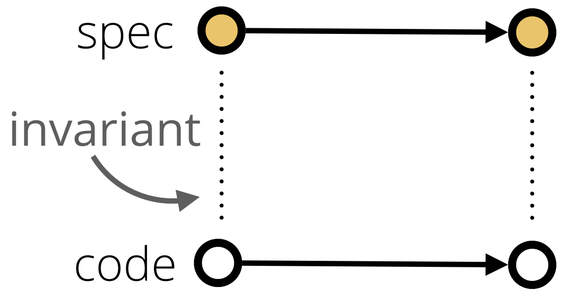
Verifying safety of software upgrades



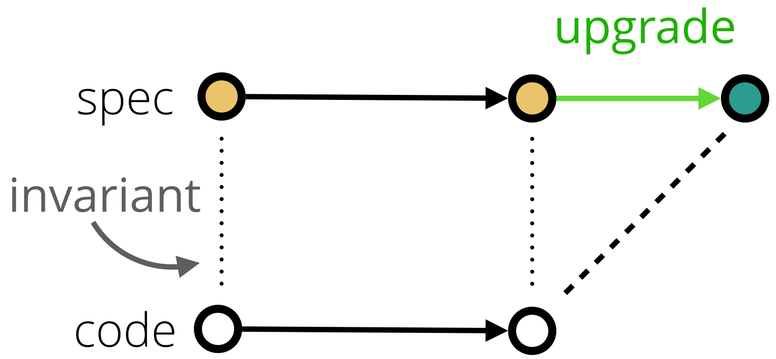
Upgrading a verified system deserves proof

File systems, distributed systems, and applications have upgrades

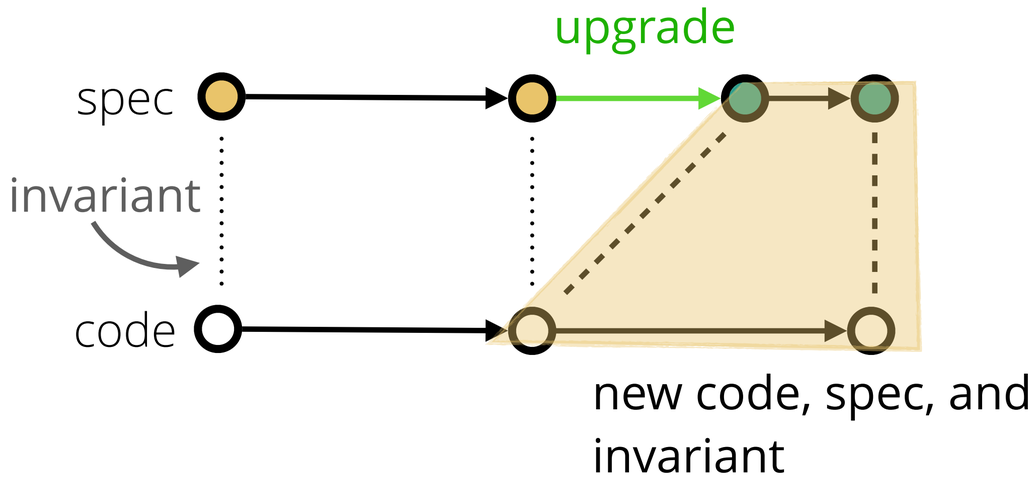
Approach: new specifications and proof patterns for software upgrades



Approach: new specifications and proof patterns for software upgrades



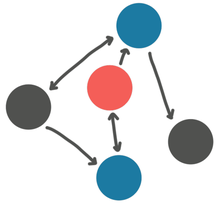
Approach: new specifications and proof patterns for software upgrades



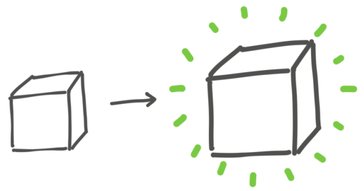
Verification beyond these domains

Security

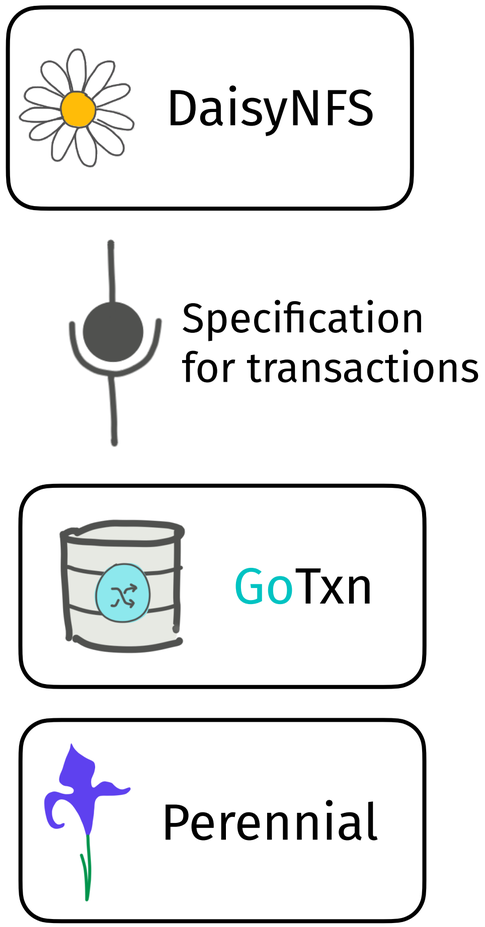
Distributed systems



Software upgrades



Vision: reliable systems software with verification



Verified a concurrent, crash-safe file system

Goal: expand reach of verification in other domains

Tej Chajed
tchajed@mit.edu

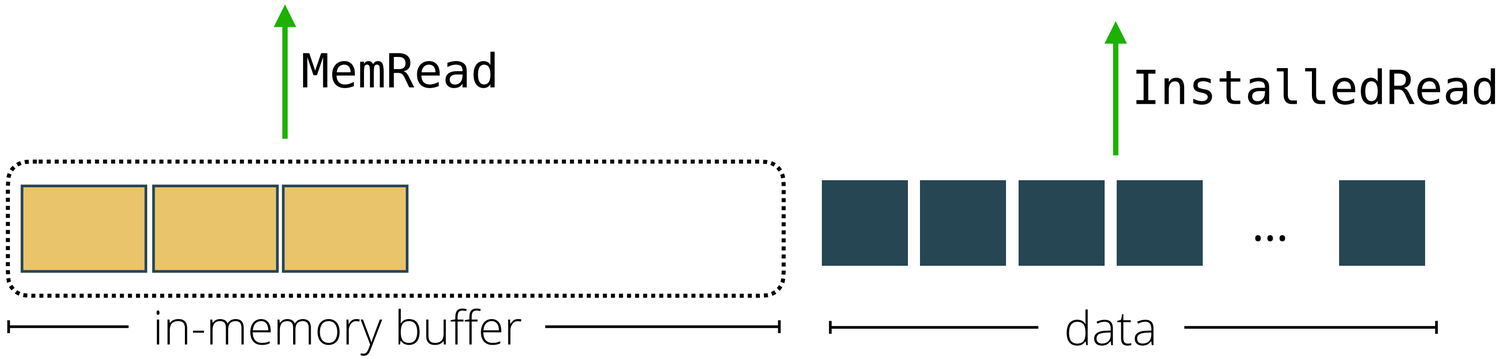
GoTxn

Reading is deceptively simple

```
func Read(a) Block {
  v, ok := MemRead(a)
  if ok {
    return v
  }
  return InstalledRead(a)
}
```

Reading is deceptively simple

```
func Read(a) Block {
  v, ok := MemRead(a)
  if ok {
    return v
  }
  return InstalledRead(a)
}
```

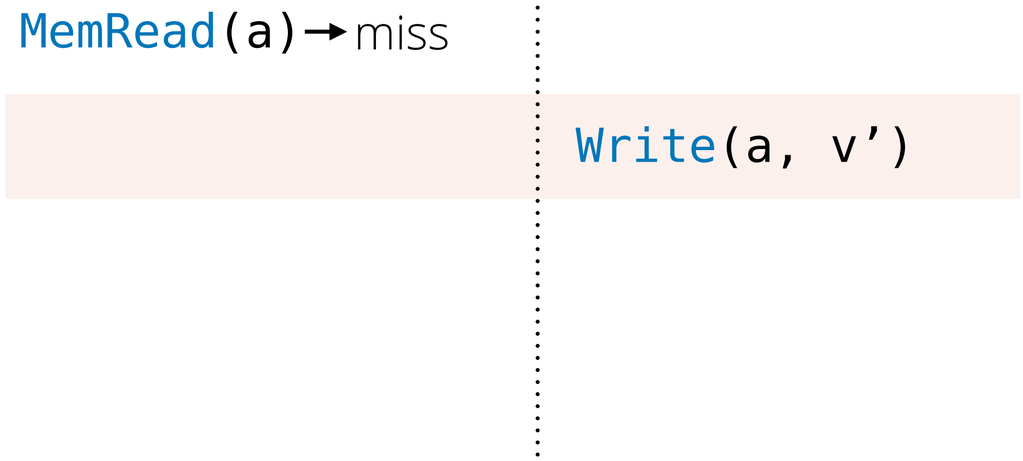


Is Read atomic?

```
v, ok := MemRead(a)
if ok {
  return v
}
return InstalledRead(a)
```

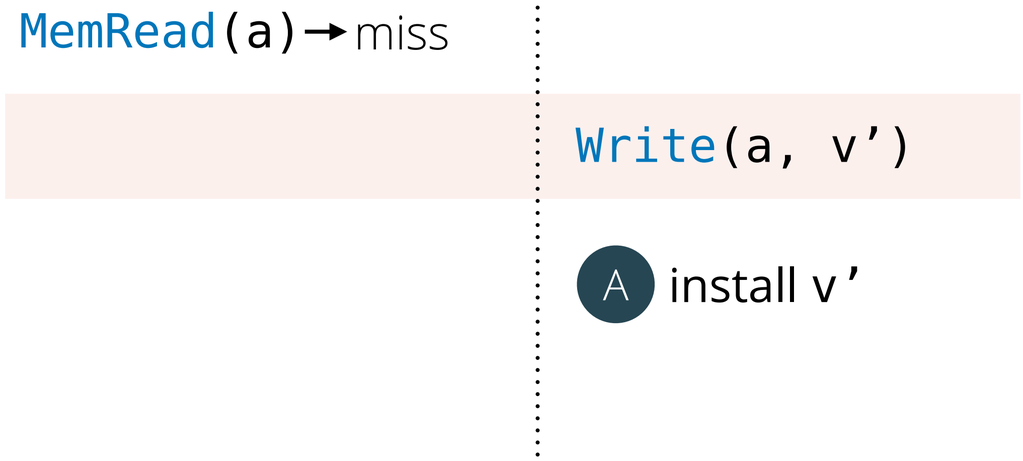
Is Read atomic?

```
v, ok := MemRead(a)
if ok {
  return v
}
return InstalledRead(a)
```



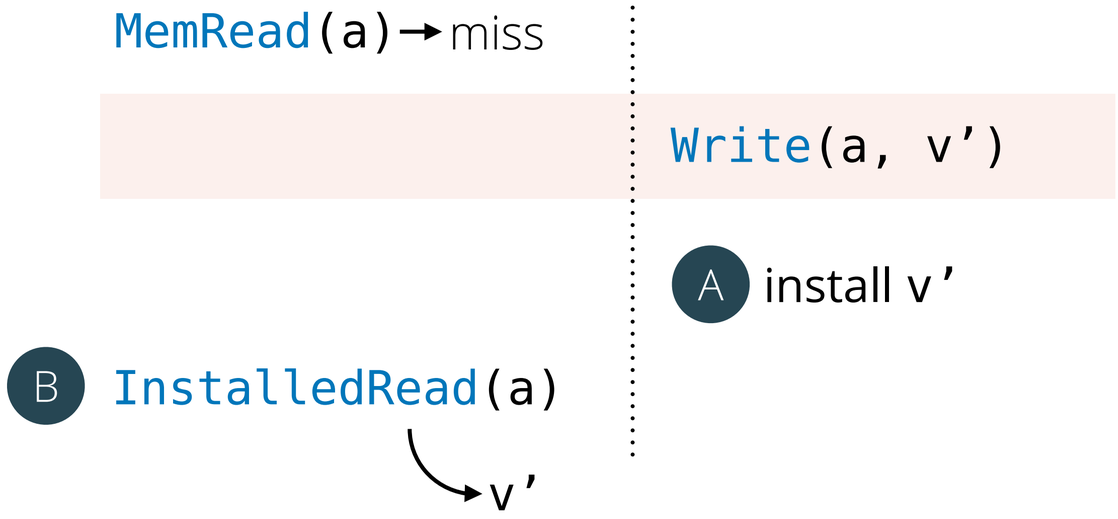
Is Read atomic?

```
v, ok := MemRead(a)
if ok {
  return v
}
return InstalledRead(a)
```



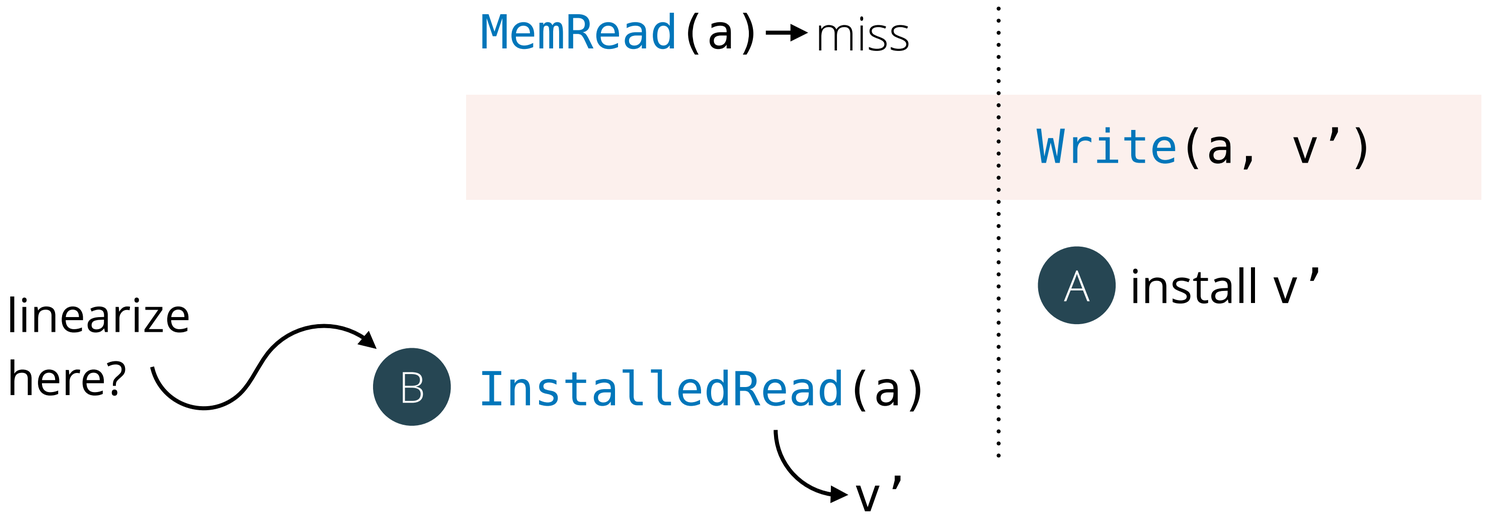
Is Read atomic?

```
v, ok := MemRead(a)
if ok {
  return v
}
return InstalledRead(a)
```



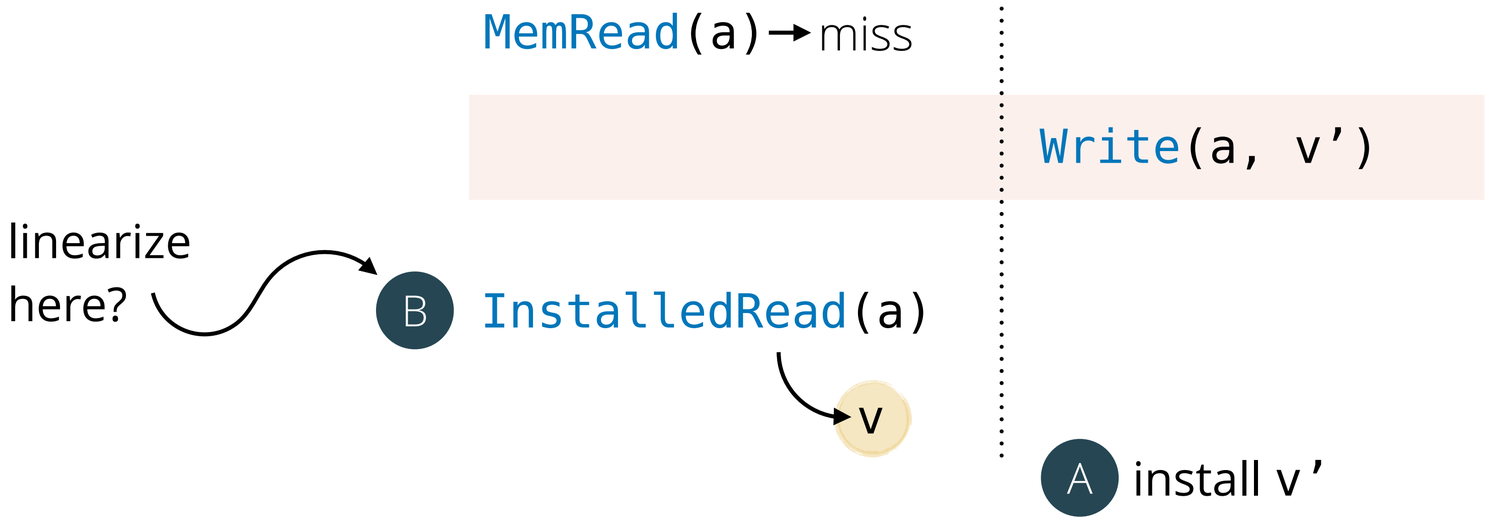
Is Read atomic?

```
v, ok := MemRead(a)
if ok {
  return v
}
return InstalledRead(a)
```



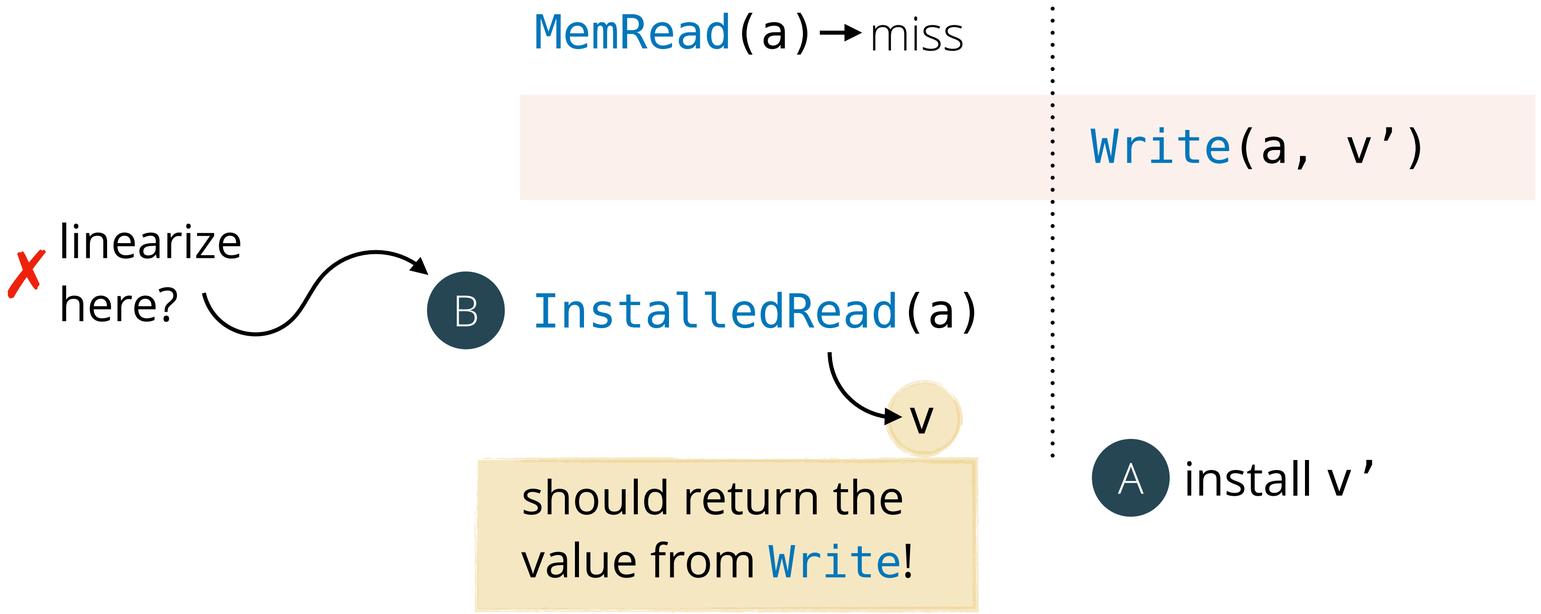
Is Read atomic?

```
v, ok := MemRead(a)
if ok {
  return v
}
return InstalledRead(a)
```



Is Read atomic?

```
v, ok := MemRead(a)
if ok {
  return v
}
return InstalledRead(a)
```



Read is atomic in a future-dependent way

```
v, ok := MemRead(a)
if ok {
  return v
}
return InstalledRead(a)
```