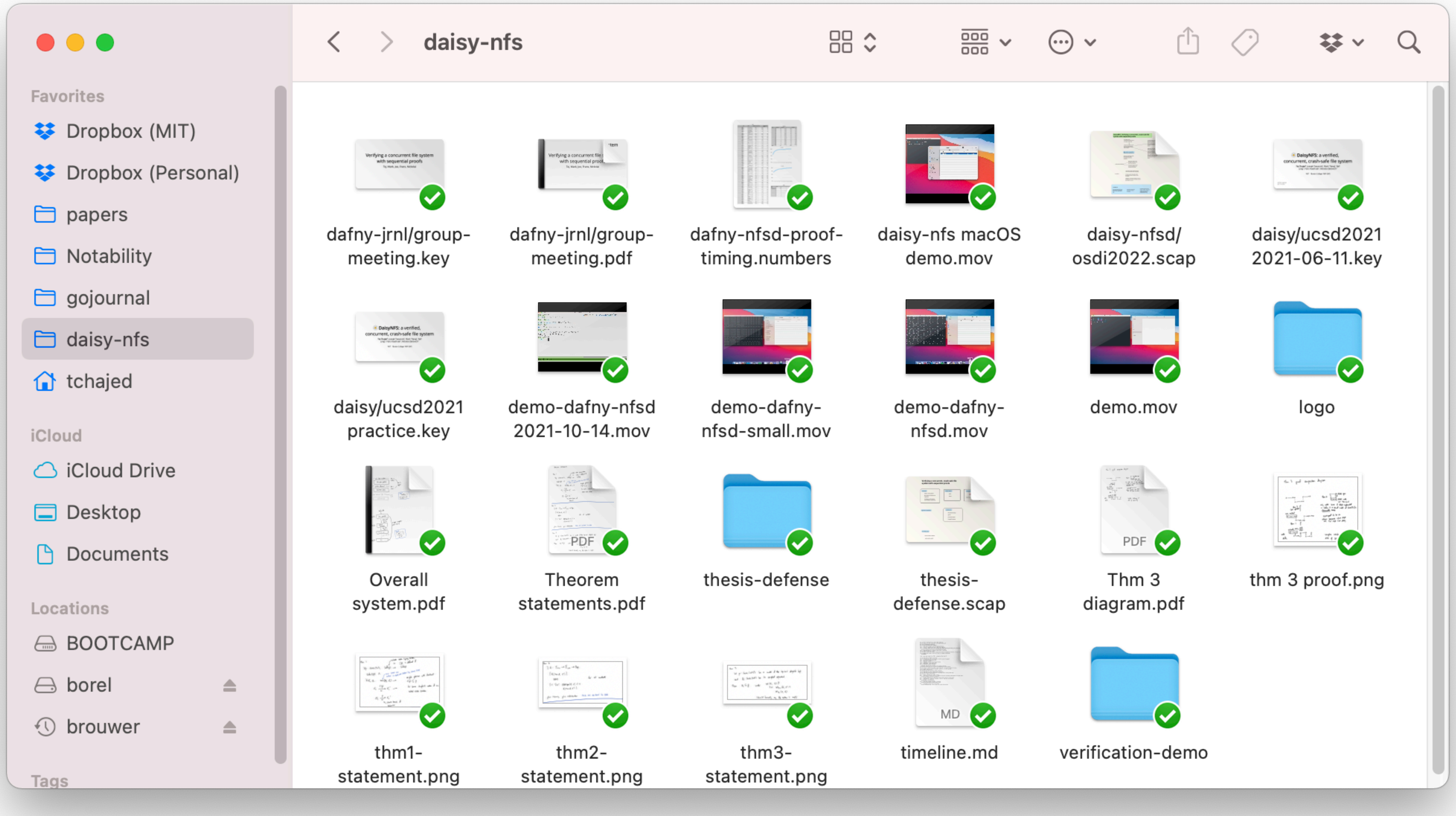
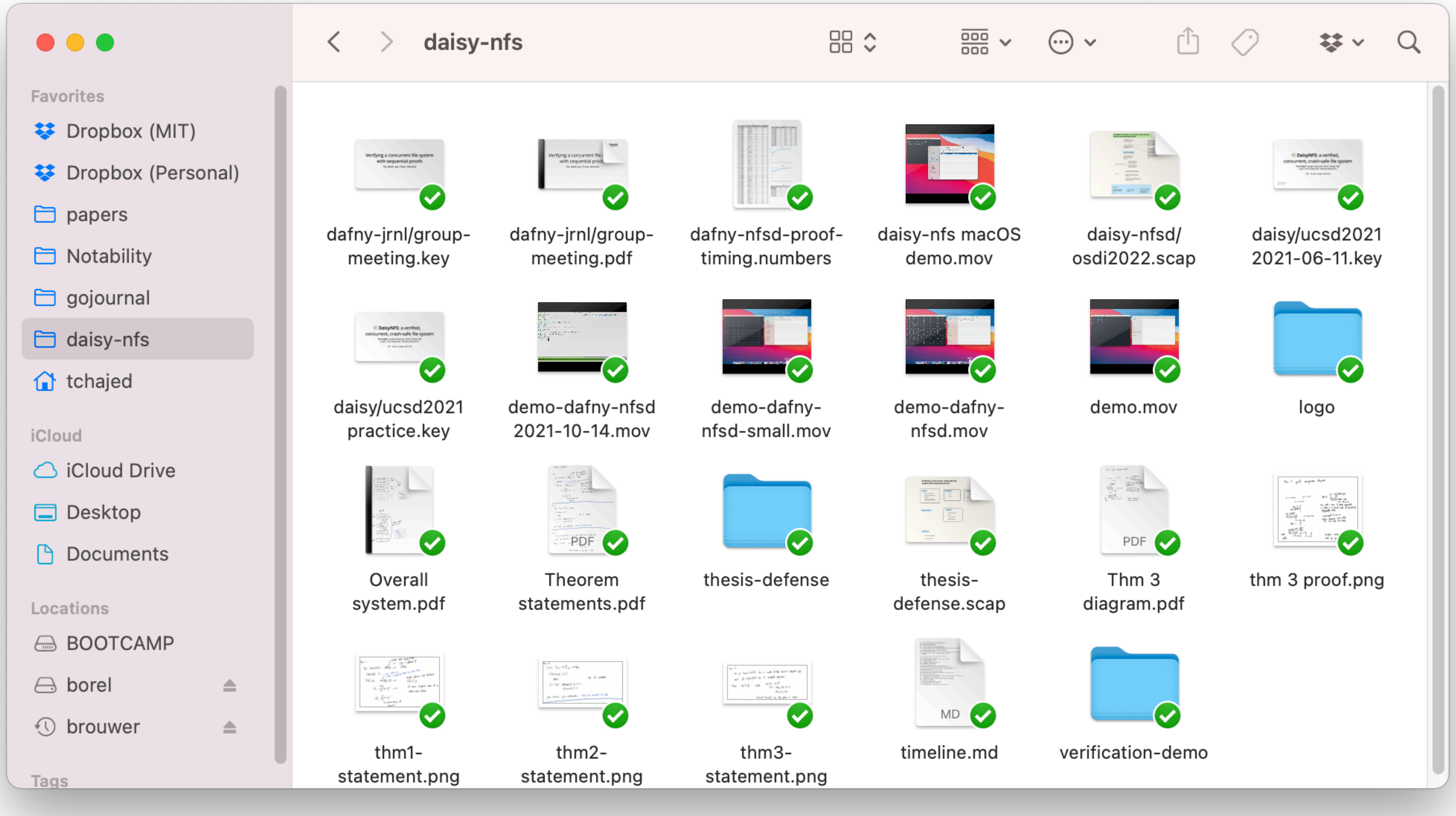
Verifying a concurrent, crash-safe file system with sequential reasoning

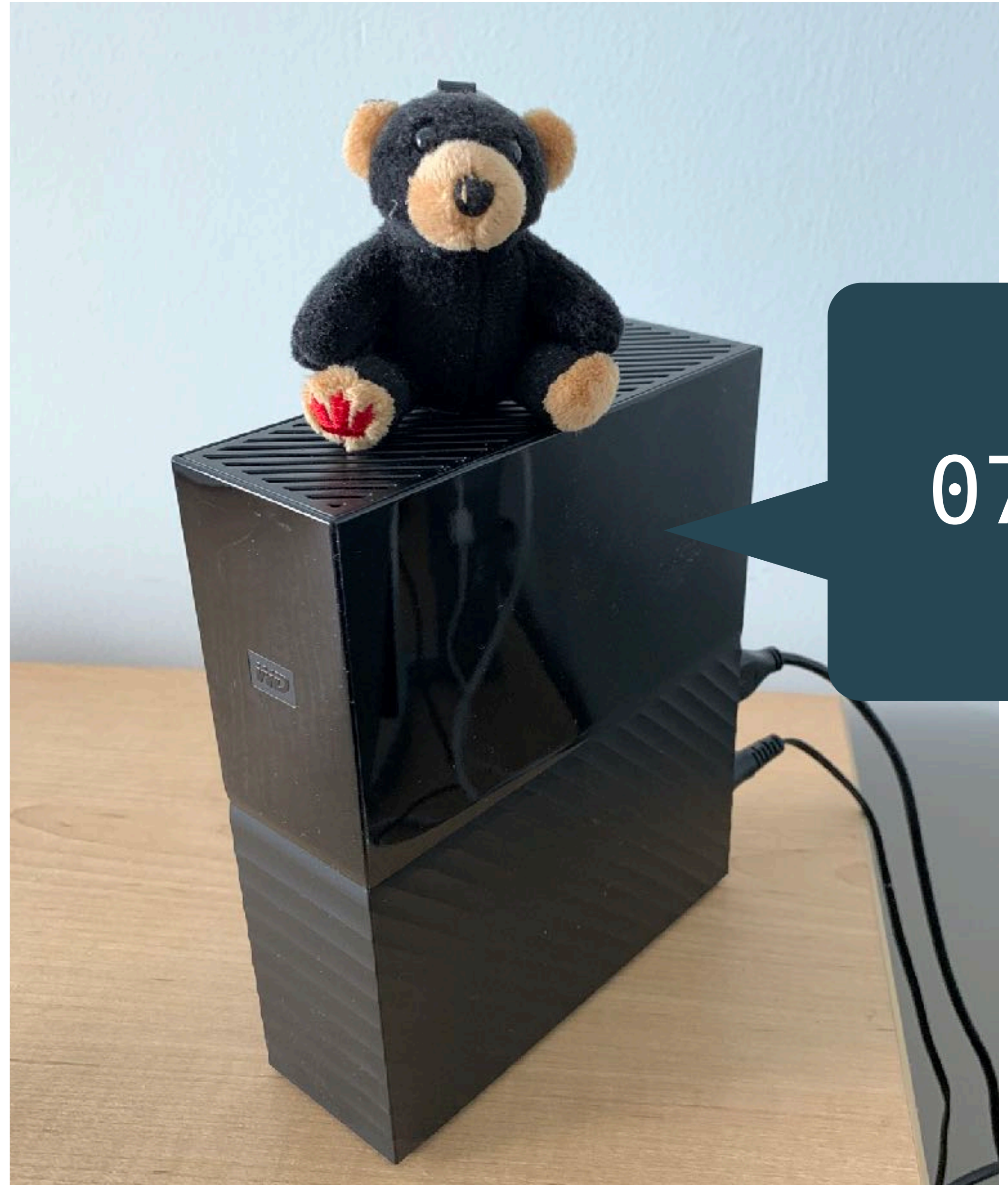
Tej Chajed

PhD defense

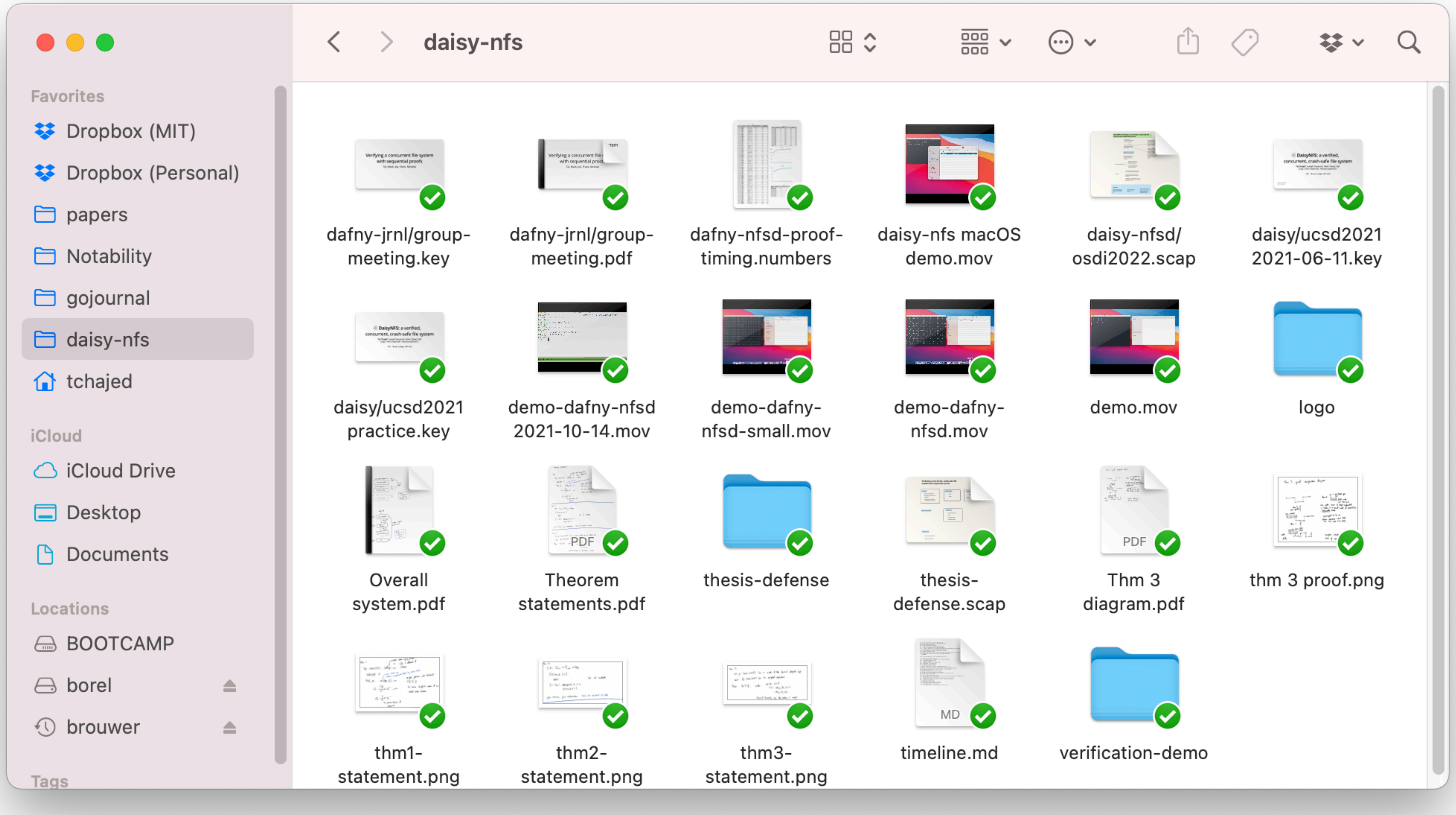
October 21st, 2021


daisy-nfs
Favorites
Dropbox (MIT)
Dropbox (Personal)
papers
Notability
gojournal
daisy-nfs
tchajed
iCloud
iCloud Drive
Desktop
Documents
Locations
BOOTCAMP
borel
brouwer
Tags
dafny-jrnl/group-meeting.key
dafny-jrnl/group-meeting.pdf
dafny-nfsd-proof-timing.numbers
daisy-nfs macOS demo.mov
daisy-nfsd/osdi2022.scap
daisy/ucsd2021 2021-06-11.key
daisy/ucsd2021 practice.key
demo-dafny-nfsd 2021-10-14.mov
demo-dafny-nfsd-small.mov
demo-dafny-nfsd.mov
demo.mov
logo
Overall system.pdf
Theorem statements.pdf
thesis-defense
thesis-defense.scap
Thm 3 diagram.pdf
thm 3 proof.png
thm1-statement.png
thm2-statement.png
thm3-statement.png
timeline.md
verification-demo

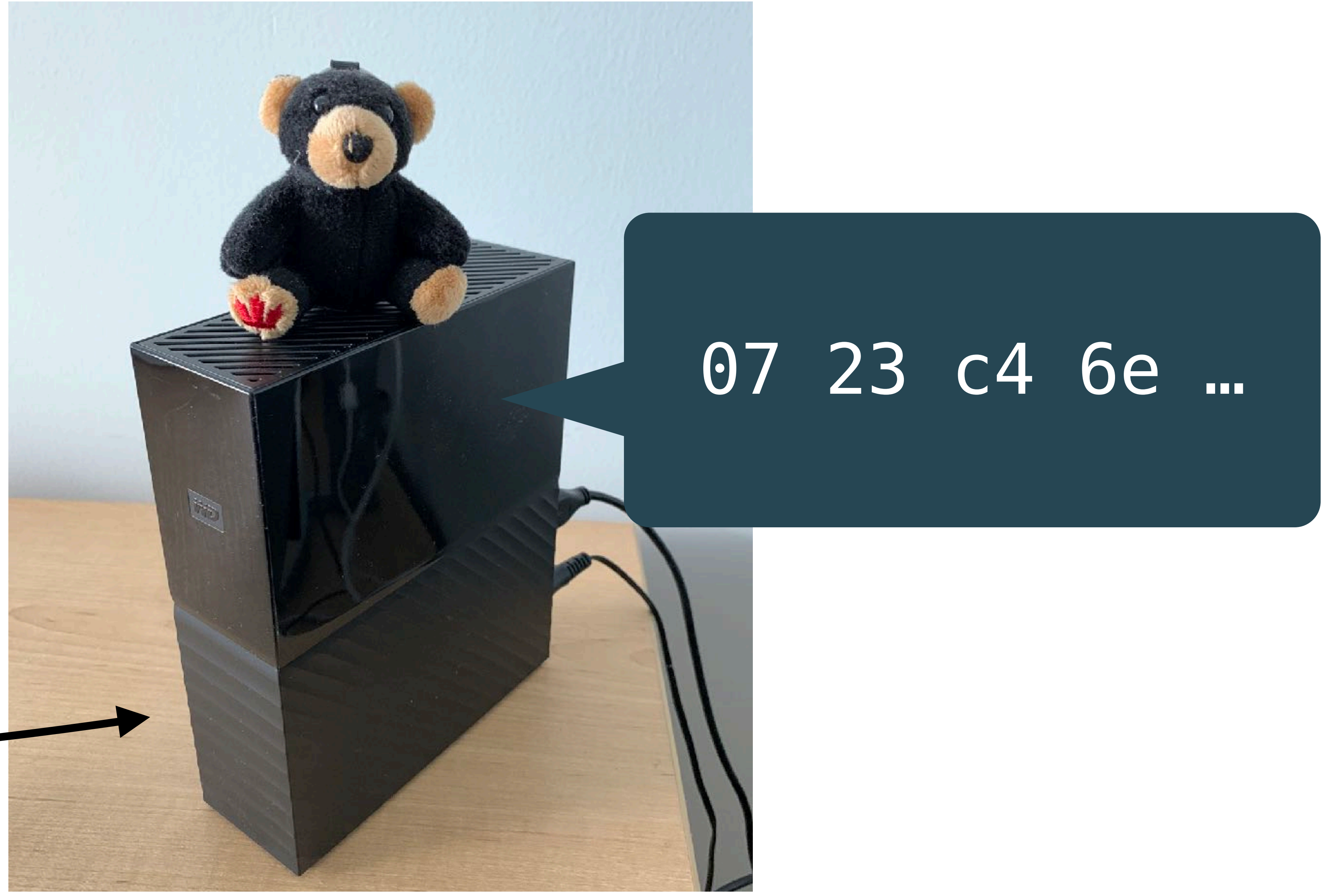

daisy-nfs
Favorites
Dropbox (MIT)
Dropbox (Personal)
papers
Notability
gojournal
daisy-nfs
tchajed
iCloud
iCloud Drive
Desktop
Documents
Locations
BOOTCAMP
borel
brouwer
Tags
dafny-jrnl/group-meeting.key
dafny-jrnl/group-meeting.pdf
dafny-nfsd-proof-timing.numbers
daisy-nfs macOS demo.mov
daisy-nfsd/osdi2022.scap
daisy/ucsd2021 2021-06-11.key
daisy/ucsd2021 practice.key
demo-dafny-nfsd 2021-10-14.mov
demo-dafny-nfsd-small.mov
demo-dafny-nfsd.mov
demo.mov
logo
Overall system.pdf
Theorem statements.pdf
thesis-defense
thesis-defense.scap
Thm 3 diagram.pdf
thm 3 proof.png
thm1-statement.png
thm2-statement.png
thm3-statement.png
timeline.md
verification-demo

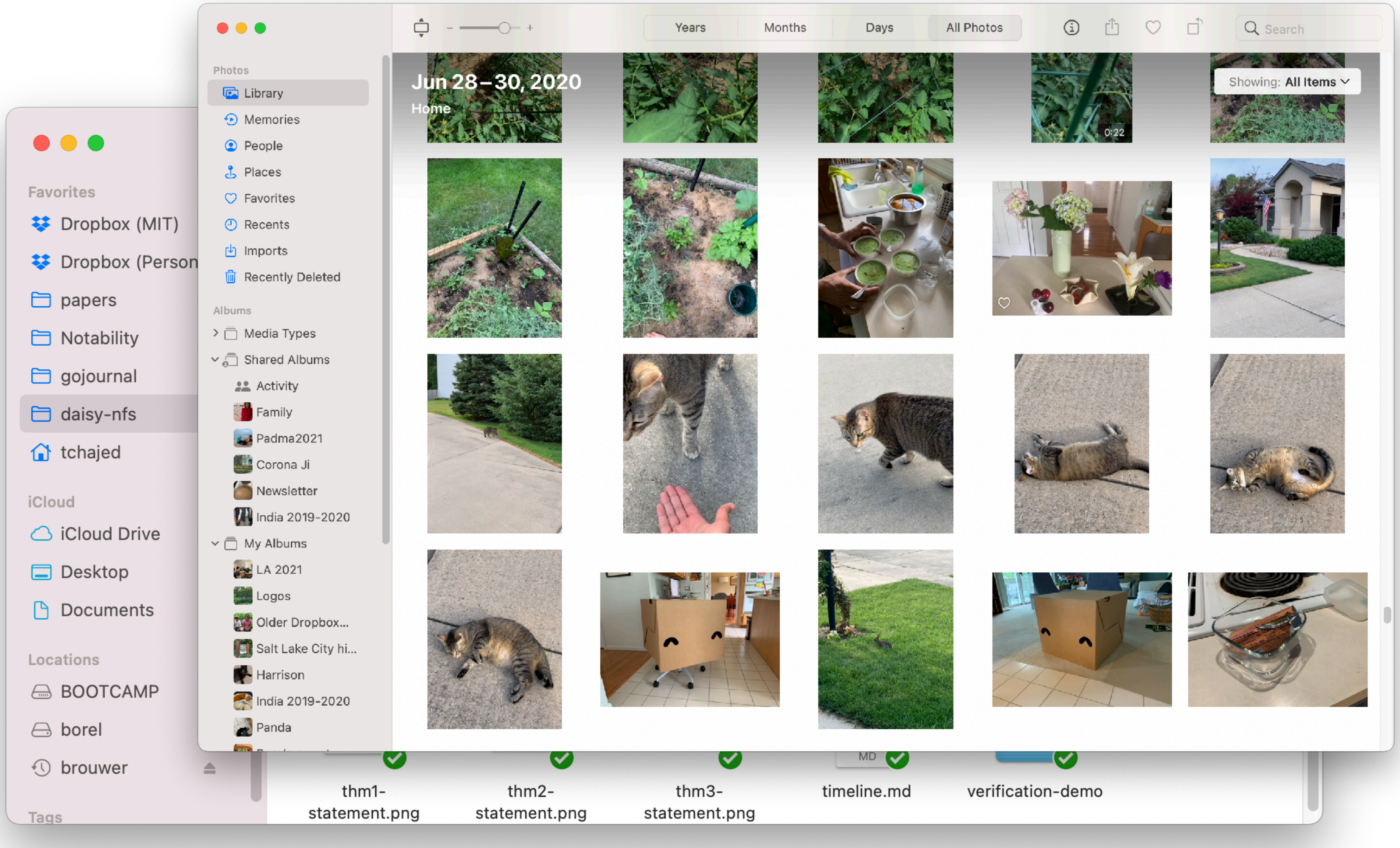


`07 23 c4 6e …`

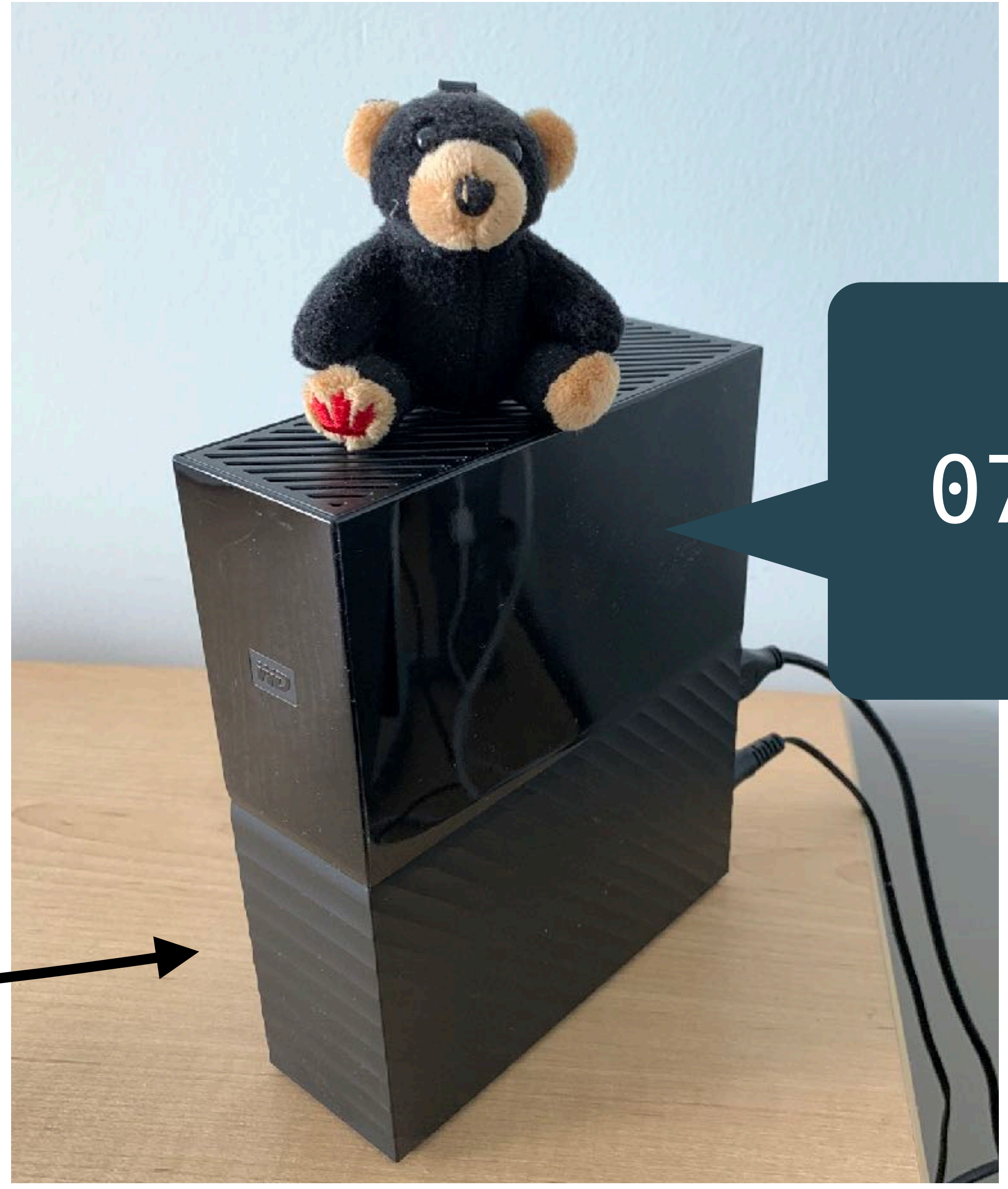

daisy-nfs
Favorites
Dropbox (MIT)
Dropbox (Personal)
papers
Notability
gojournal
daisy-nfs
tchajed
iCloud
iCloud Drive
Desktop
Documents
Locations
BOOTCAMP
borel
brouwer
Tags
dafny-jrnl/group-meeting.key
dafny-jrnl/group-meeting.pdf
dafny-nfsd-proof-timing.numbers
daisy-nfs macOS demo.mov
daisy-nfsd/osdi2022.scap
daisy/ucsd2021 2021-06-11.key
daisy/ucsd2021 practice.key
demo-dafny-nfsd 2021-10-14.mov
demo-dafny-nfsd-small.mov
demo-dafny-nfsd.mov
demo.mov
logo
Overall system.pdf
Theorem statements.pdf
thesis-defense
thesis-defense.scap
Thm 3 diagram.pdf
thm 3 proof.png
thm1-statement.png
thm2-statement.png
thm3-statement.png
timeline.md
verification-demo

file system

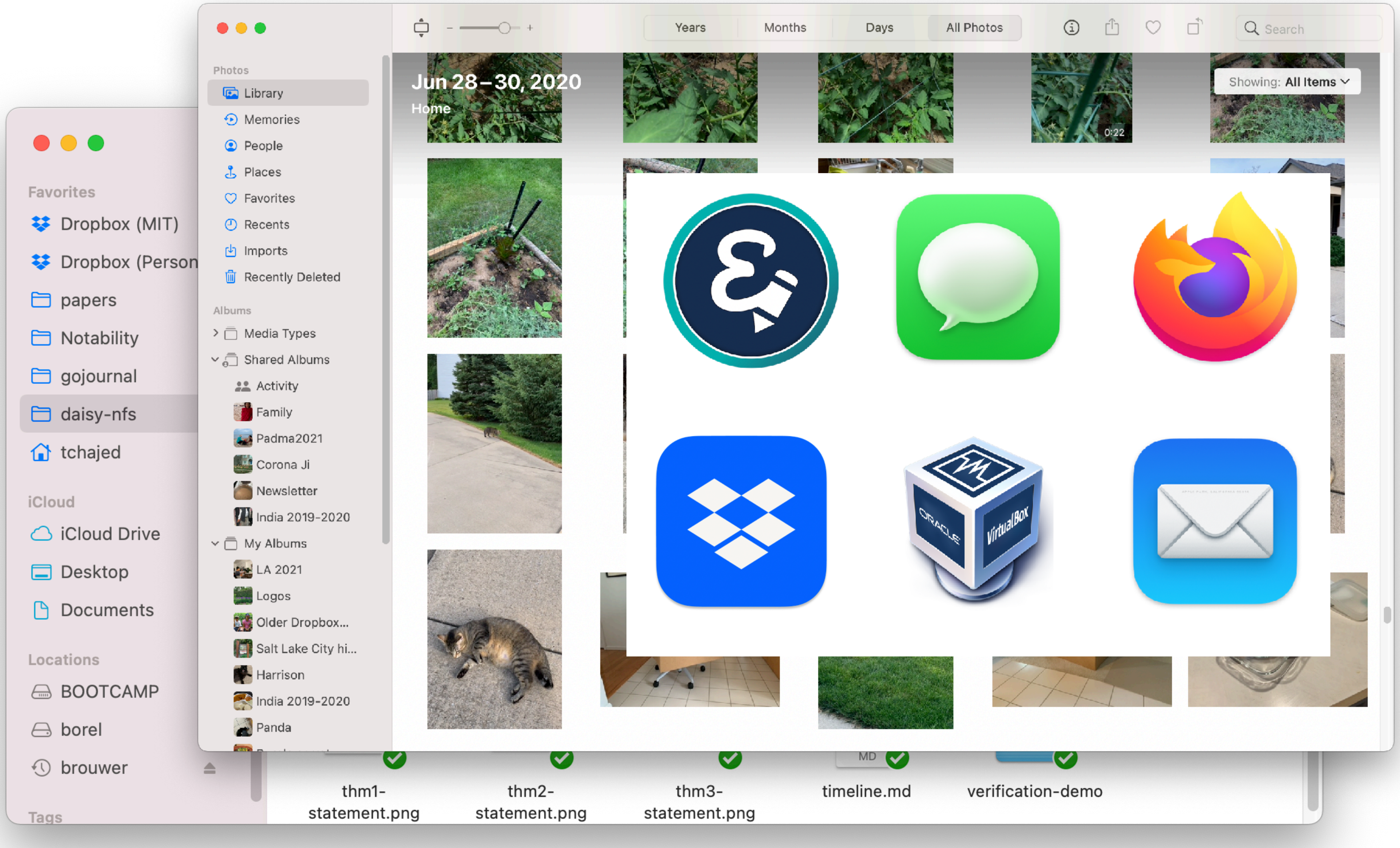

07 23 c4 6e …



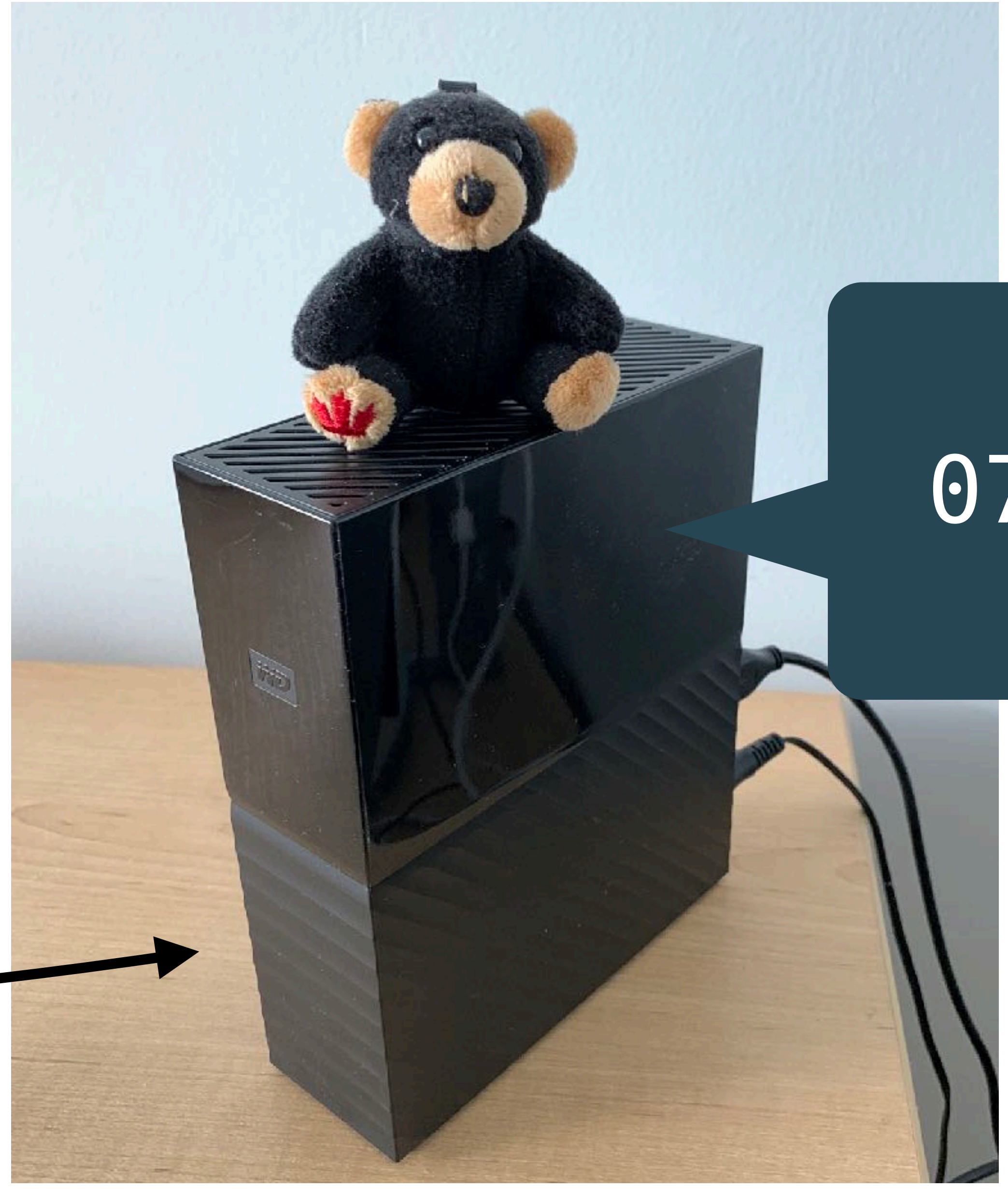
file system



07 23 c4 6e …



file system



07 23 c4 6e …

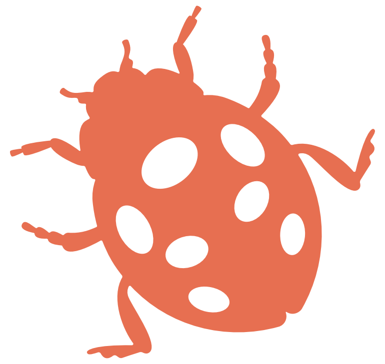
Important that the file system be correct

Responsible for storing all persistent data

Nearly all applications rely on it

Bugs can cause permanent data loss

File systems are just programs and therefore **they have bugs**



File systems struggle with
crash safety + concurrency + high performance

File systems struggle with
crash safety + concurrency + high performance

a *crash* is any sudden interruption, like a power failure

File systems struggle with
crash safety + concurrency + high performance

a *crash* is any sudden interruption, like a power failure

concurrency comes from devices, simultaneous user requests

File systems struggle with **crash safety + concurrency + high performance**

a *crash* is any sudden interruption, like a power failure

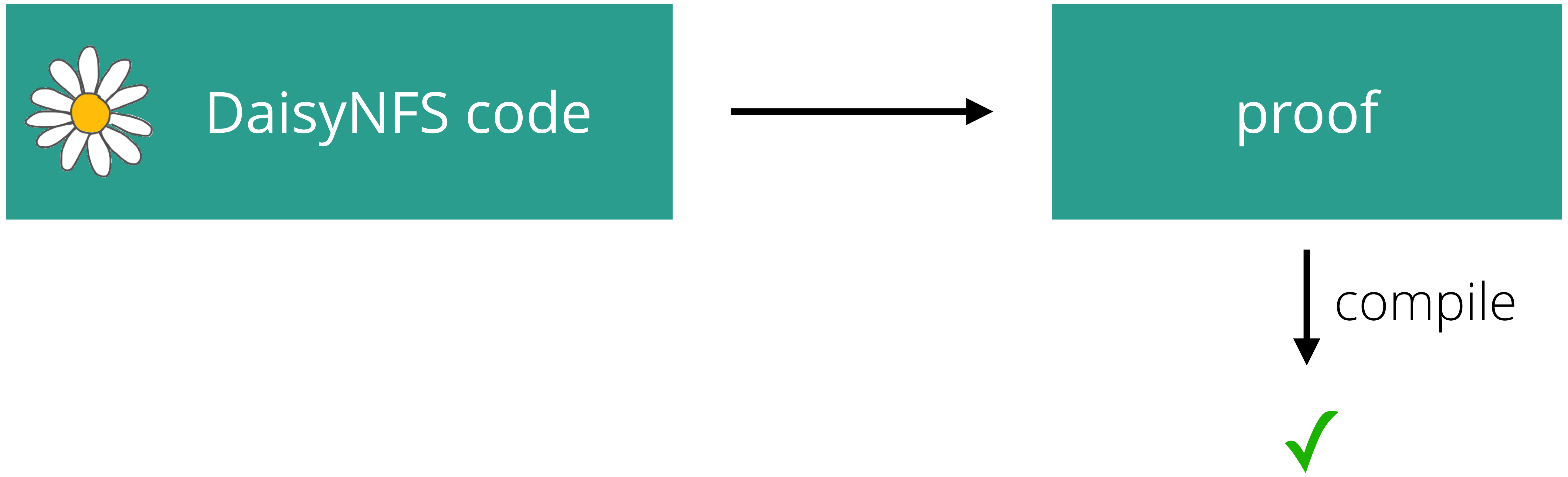
concurrency comes from devices, simultaneous user requests

high performance makes both of these hard

DaisyNFS is a new, verified file system

DaisyNFS code

DaisyNFS is a new, verified file system



DaisyNFS code

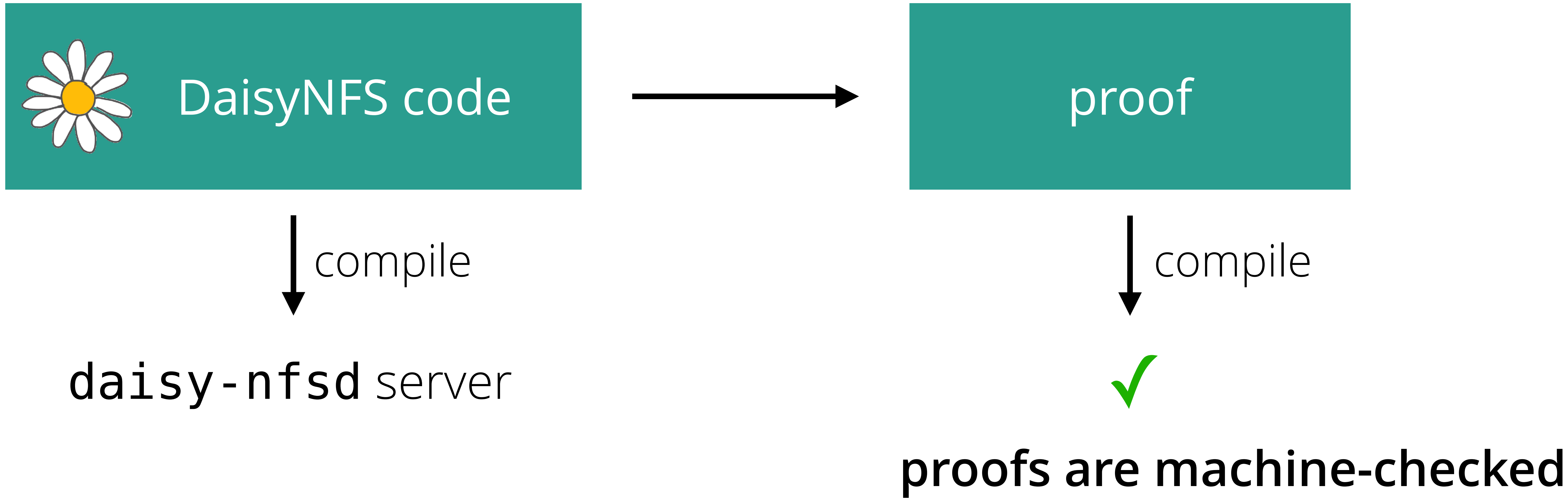
proof

compile

✓

proofs are machine-checked

DaisyNFS is a new, verified file system



DaisyNFS code

proof

compile

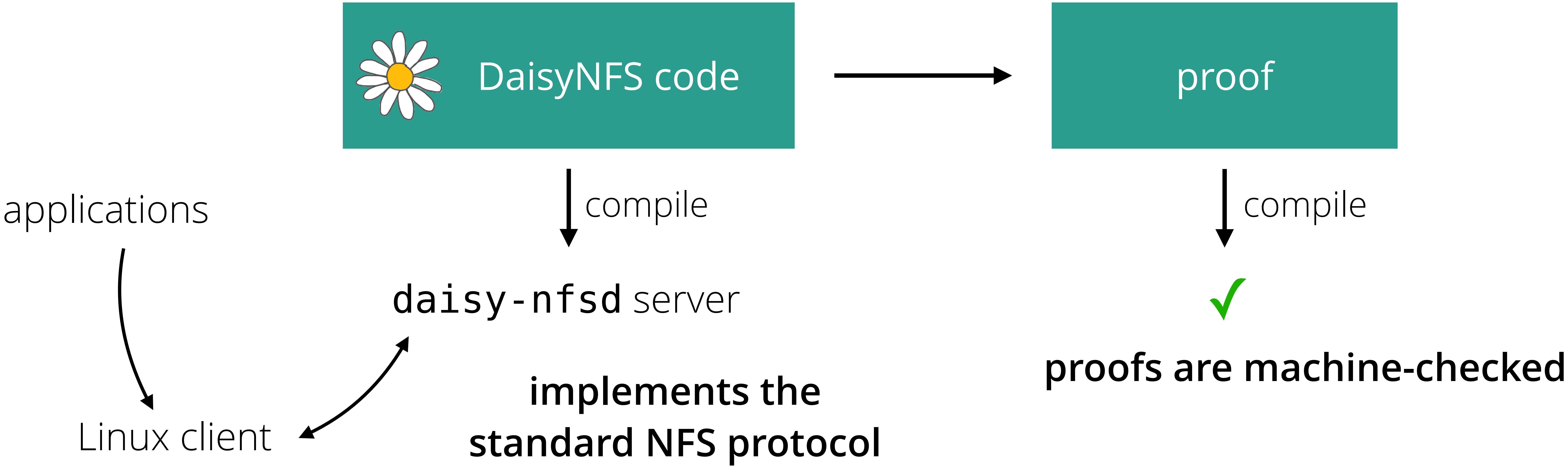
compile

`daisy-nfsd` server

✓

proofs are machine-checked

DaisyNFS is a new, verified file system



DaisyNFS code

proof

compile

compile

applications

Linux client

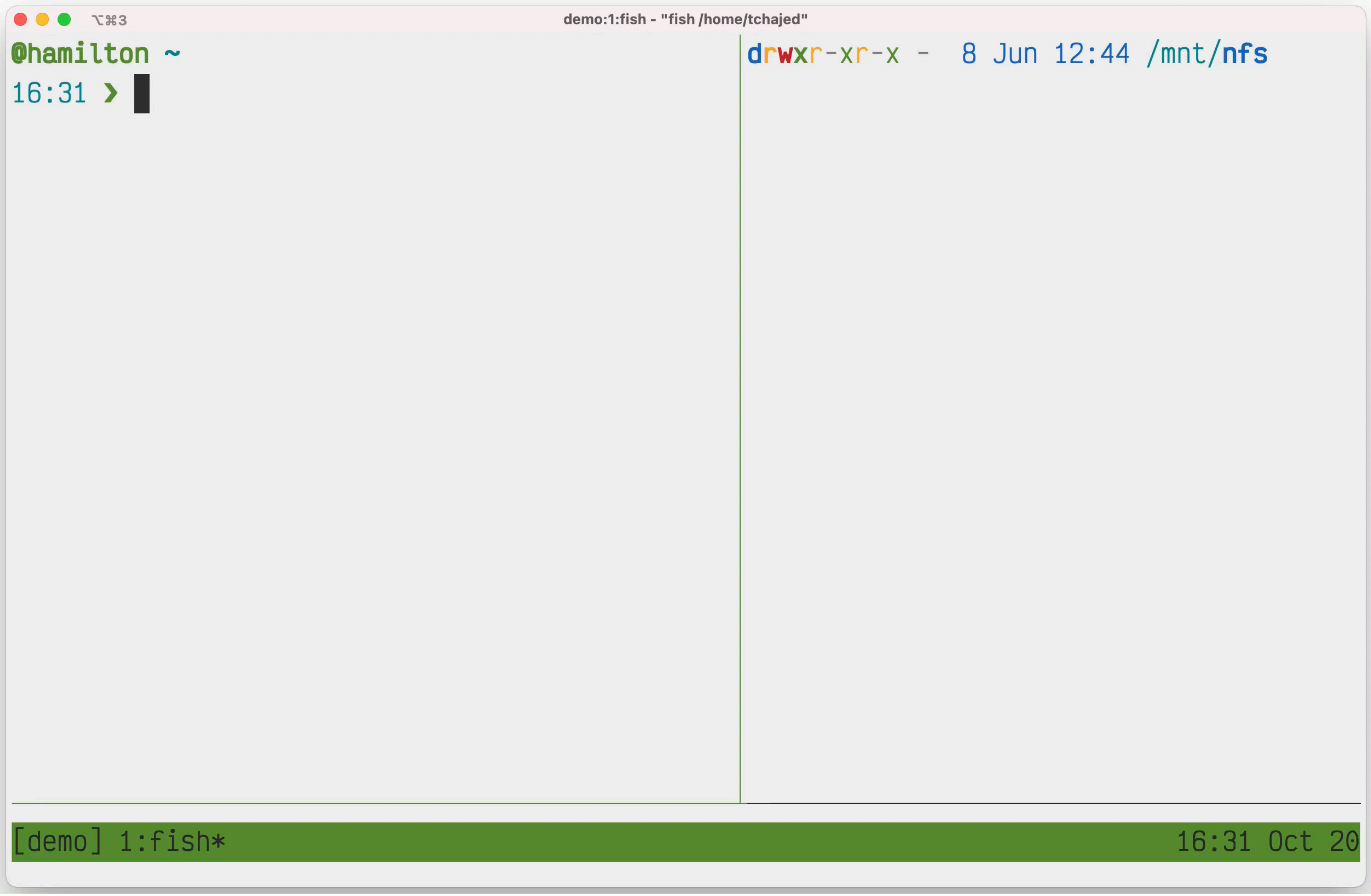
`daisy-nfsd` server

✓

implements the standard NFS protocol

proofs are machine-checked

DaisyNFS is a real file system

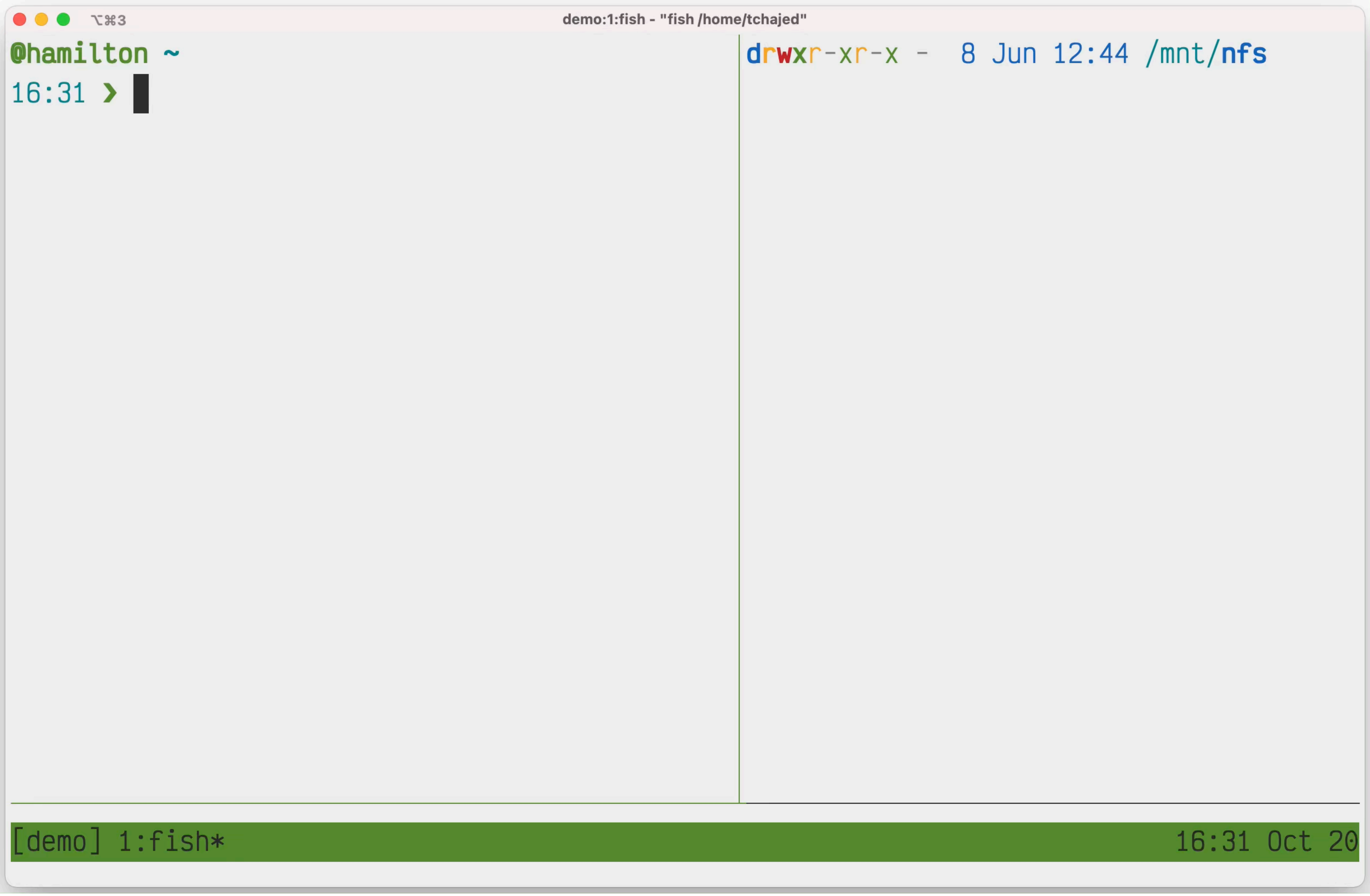


```
@hamilton ~
16:31 ❯ 
```

```
drwxr-xr-x -  8 Jun 12:44 /mnt/nfs
```

```
[demo] 1:fish*                                        16:31 Oct 20
```

DaisyNFS is a real file system



```
@hamilton ~
16:31 ❯
```

```
drwxr-xr-x -  8 Jun 12:44 /mnt/nfs
```

```
[demo] 1:fish*                                                    16:31 Oct 20
```

Approach: formal verification

Give **mathematical proof** that code does what it's supposed to

Formalize desired behavior as a **specification**

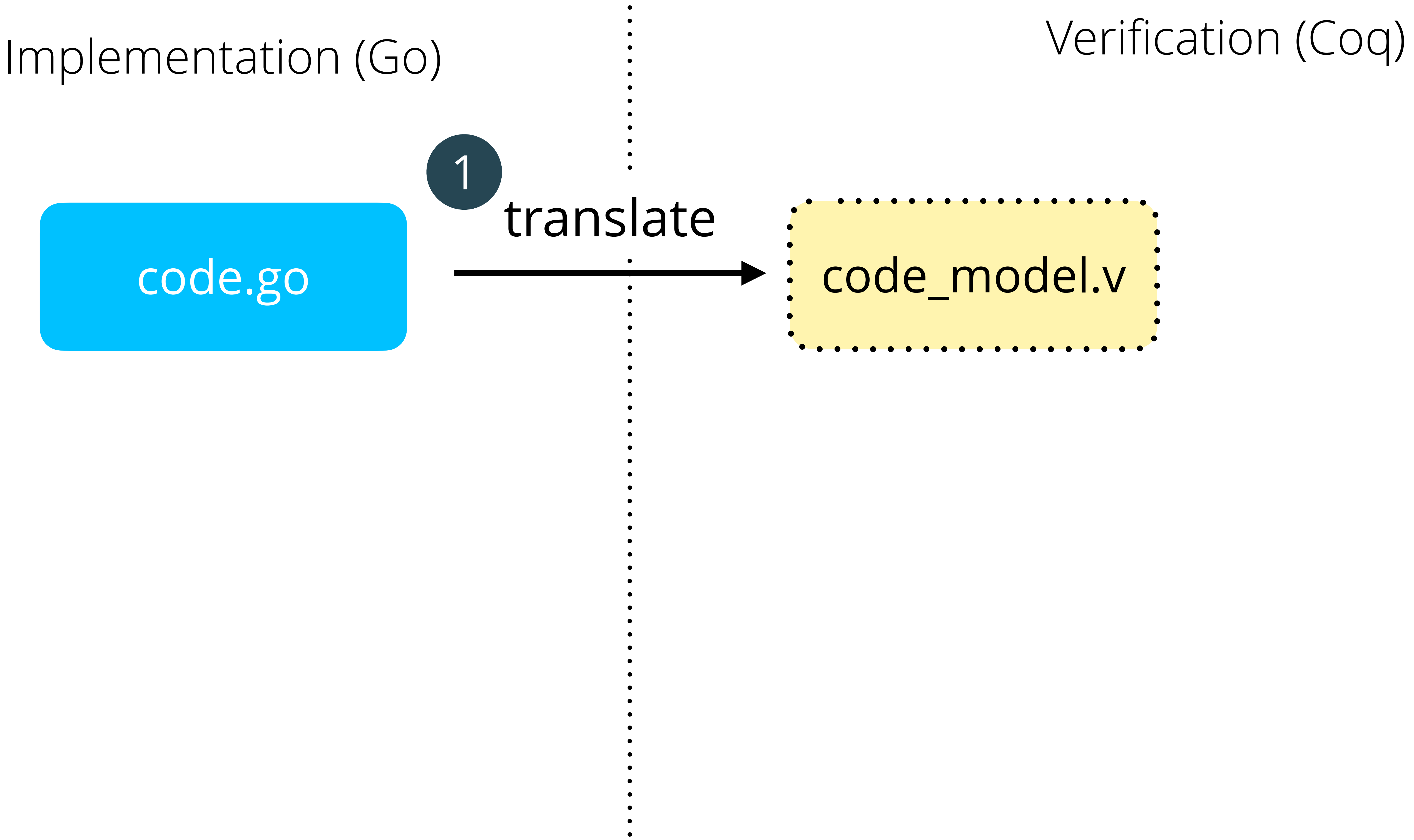
Approach: formal verification

Implementation (Go)

code.go

Verification (Coq)

Approach: formal verification



Implementation (Go)

Verification (Coq)

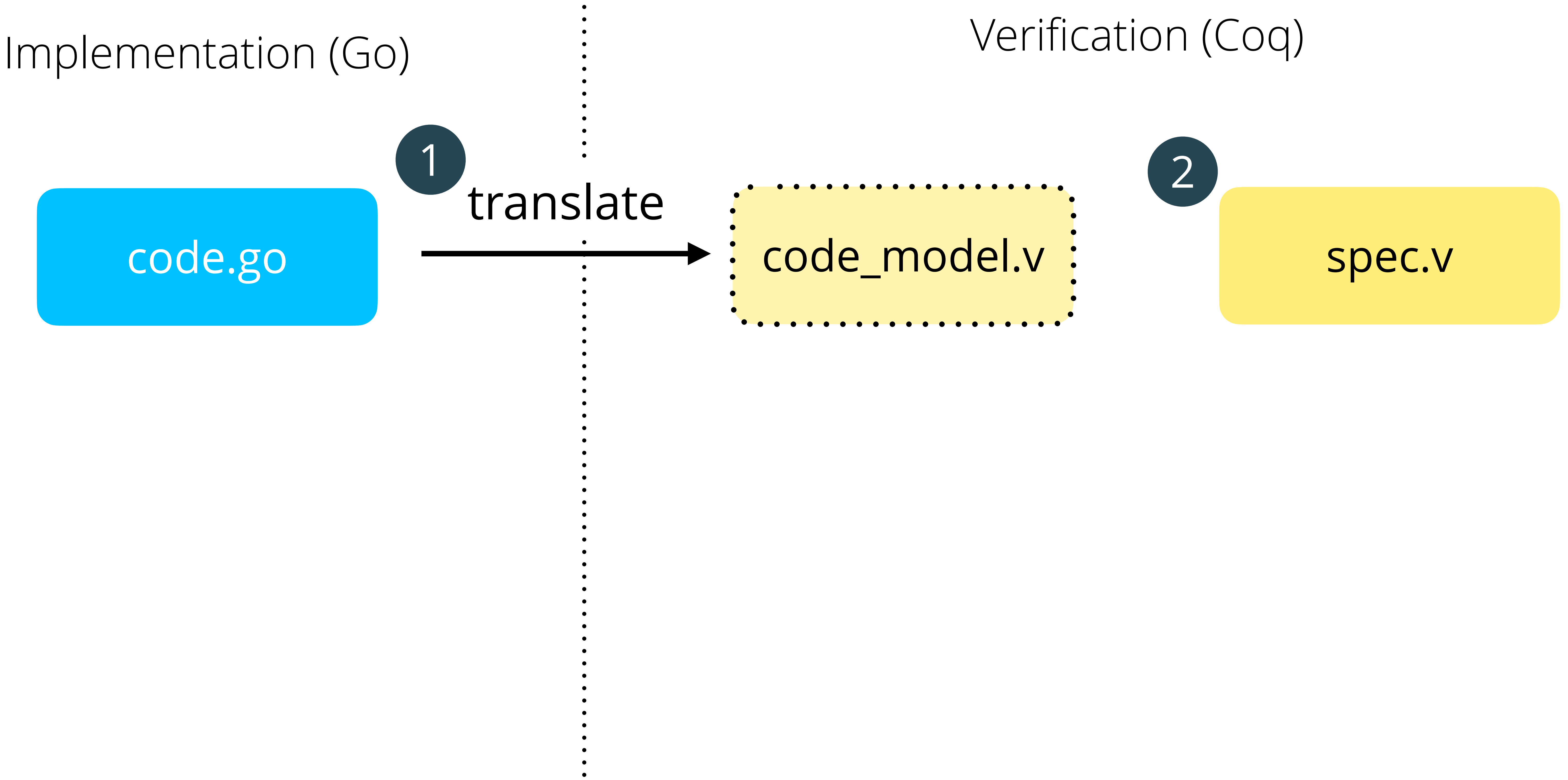
①

code.go

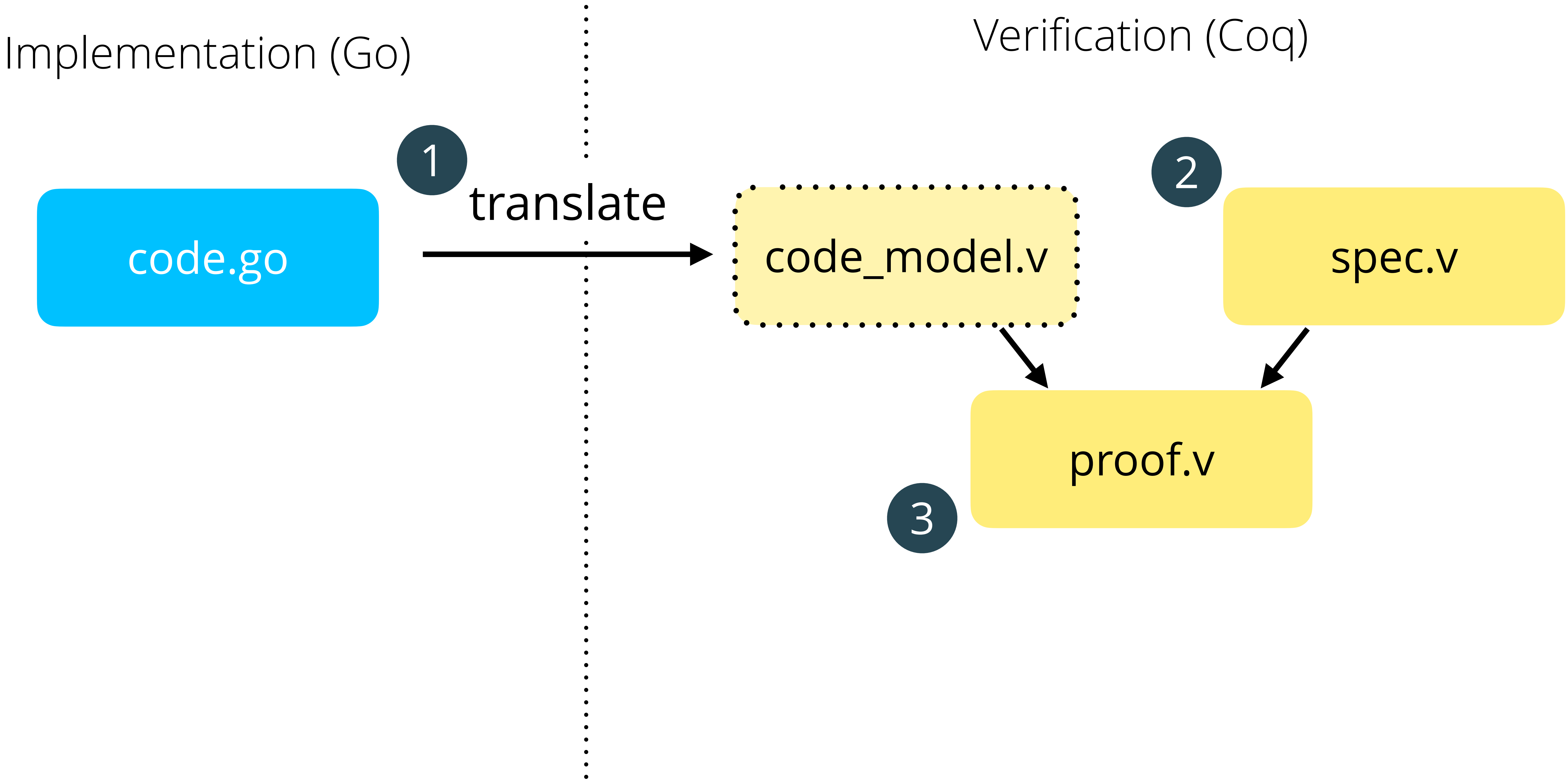
translate

code_model.v

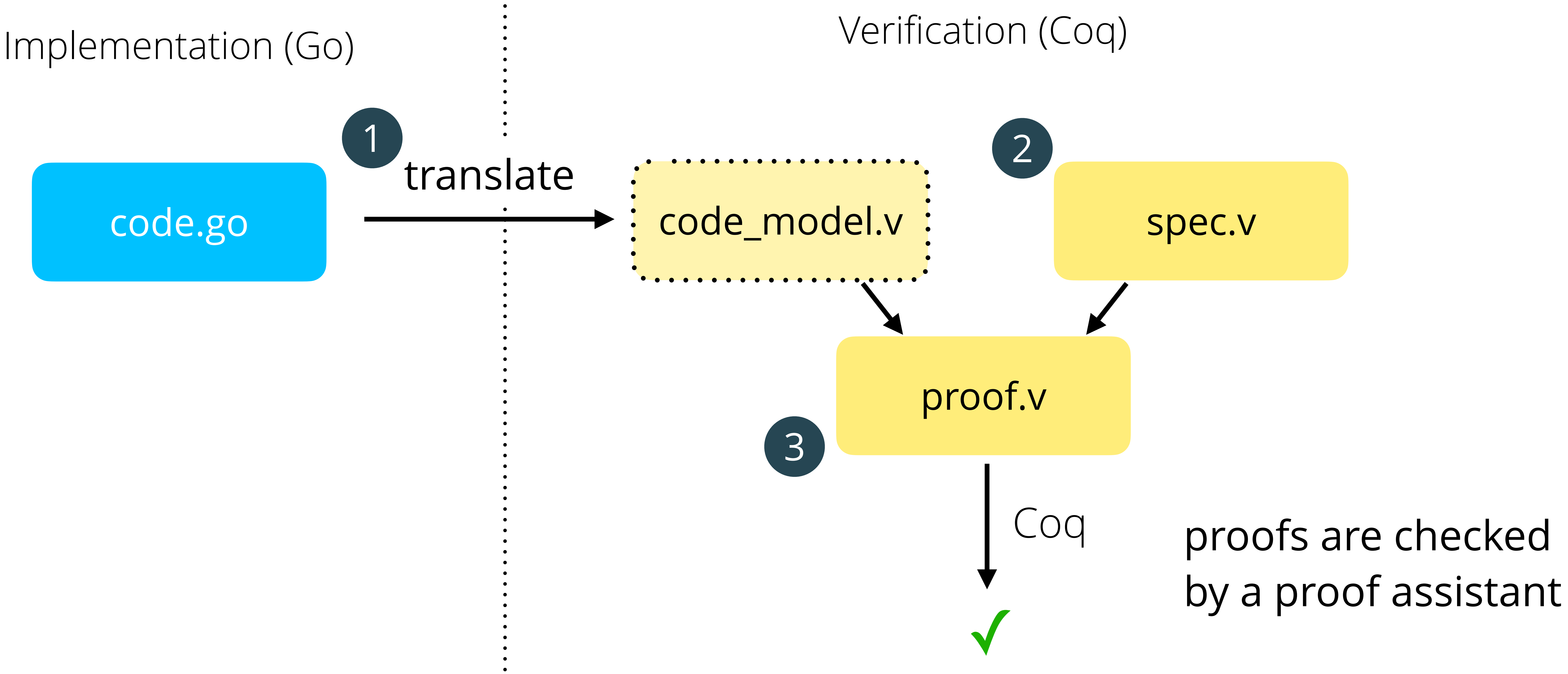
Approach: formal verification



Approach: formal verification

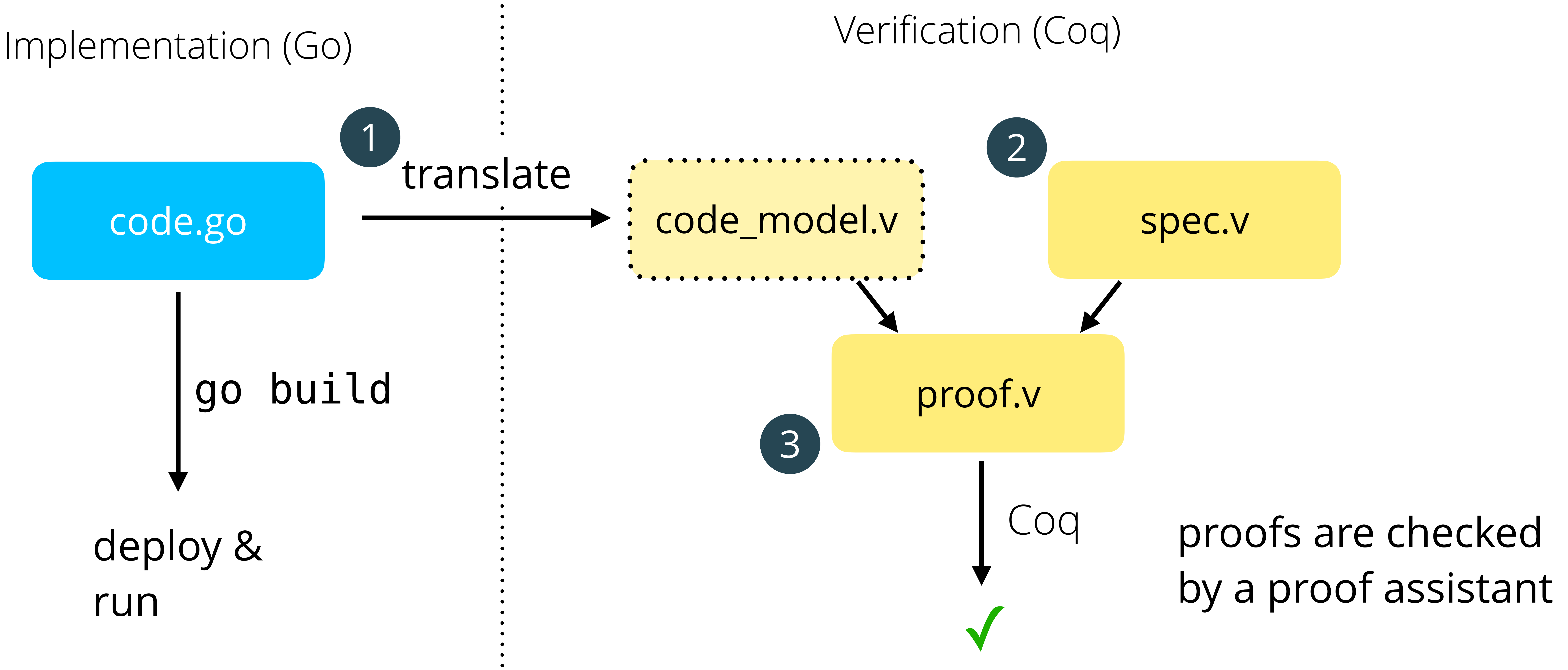


Approach: formal verification



proofs are checked
by a proof assistant

Approach: formal verification



Implementation (Go)

Verification (Coq)

① code.go

translate

code_model.v

② spec.v

③ proof.v

`go build`

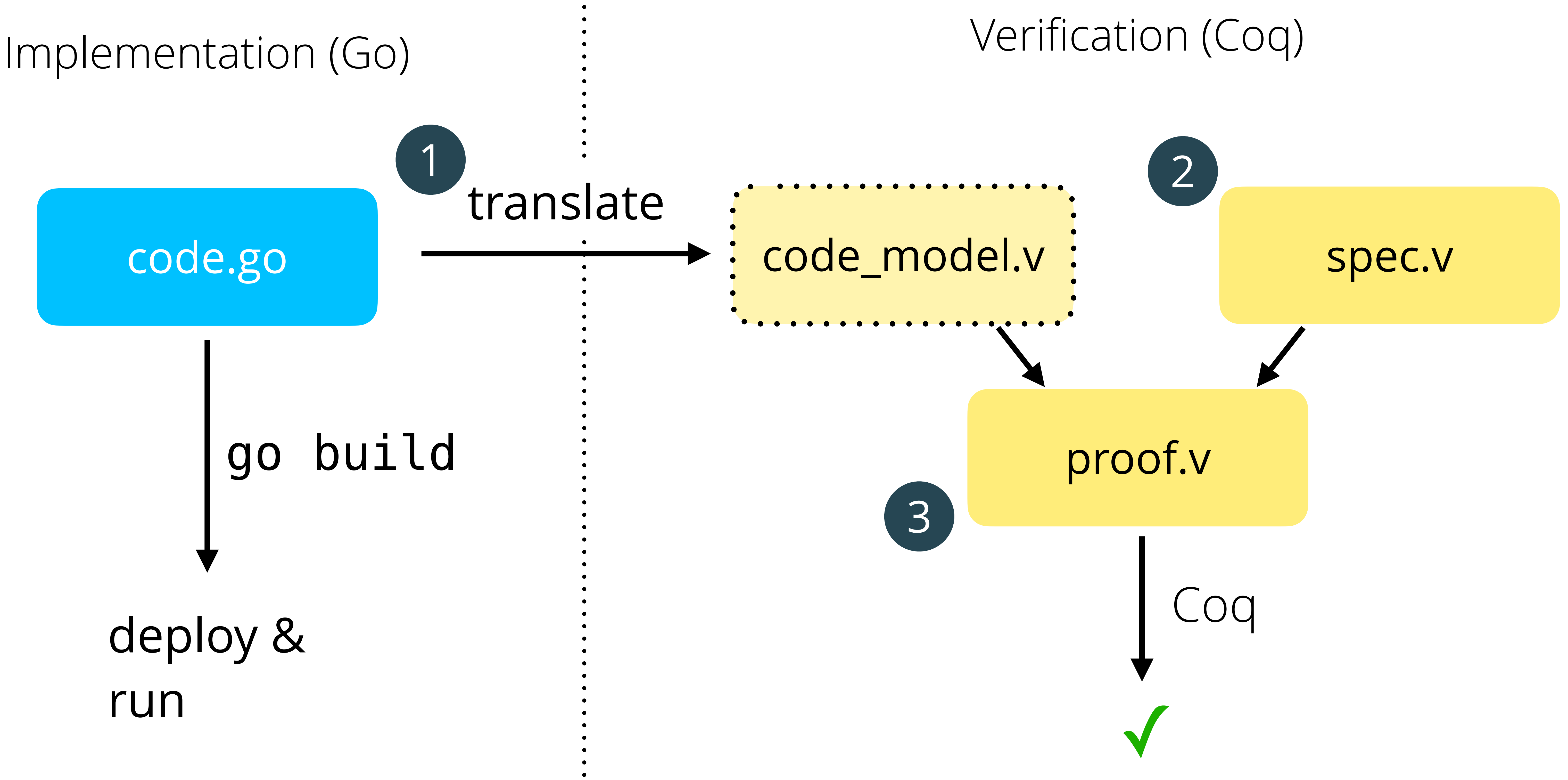
deploy & run

Coq

✓

proofs are checked by a proof assistant

Verification of storage systems is challenging



Implementation (Go)

Verification (Coq)

① translate

code.go

code_model.v

② spec.v

③ proof.v

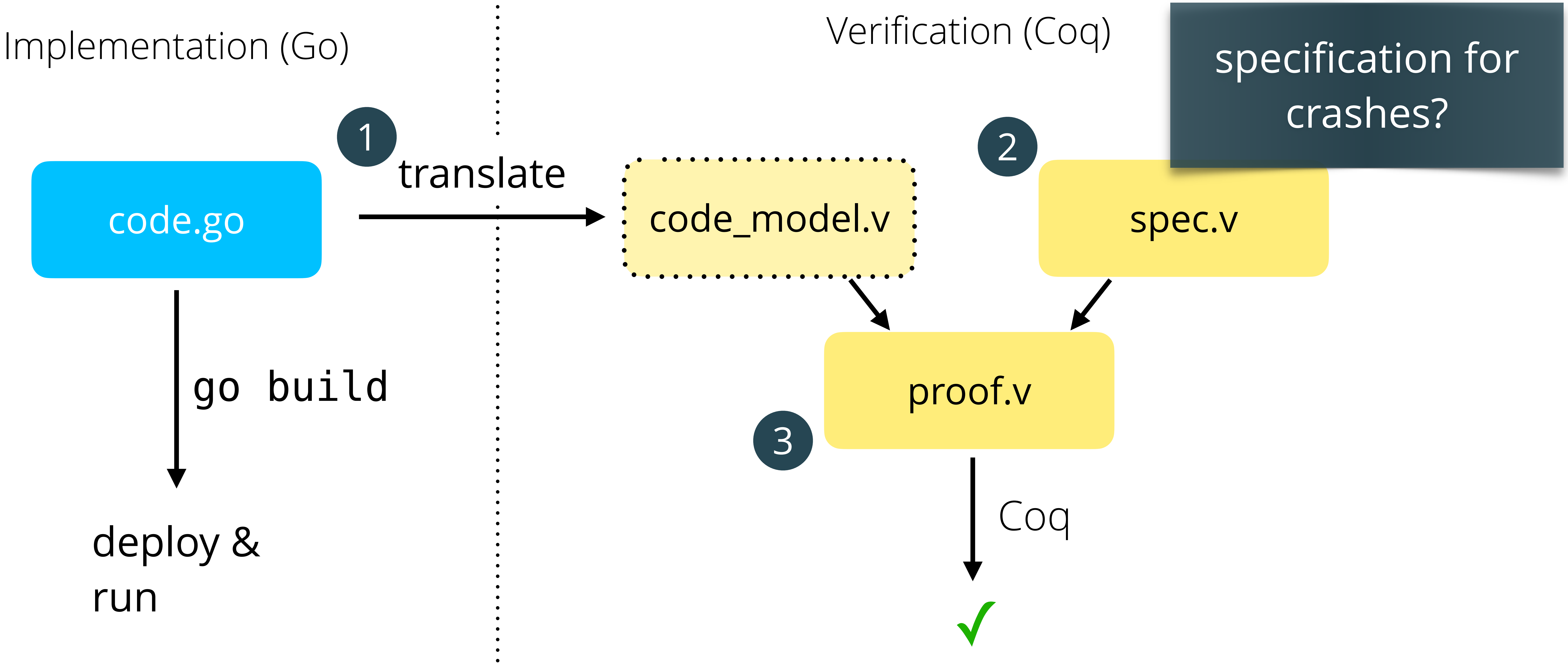
`go build`

deploy & run

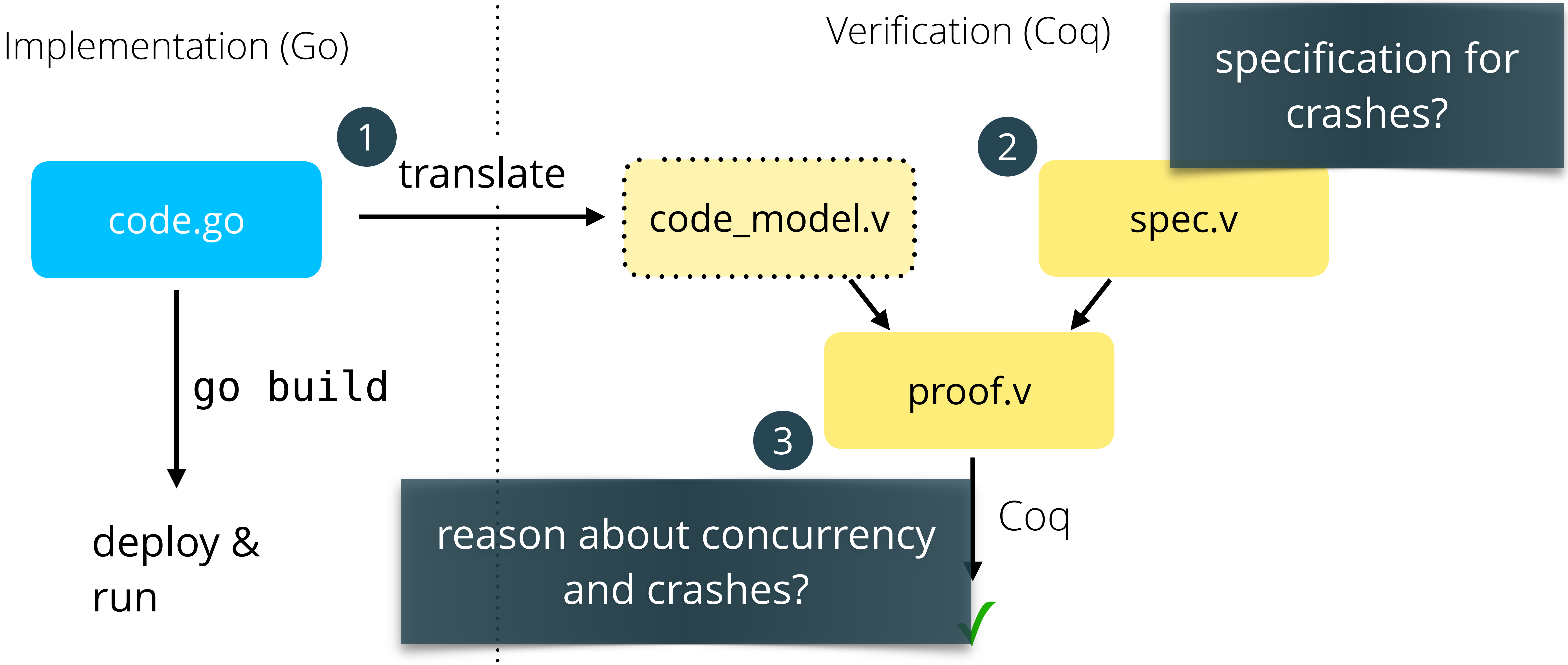
Coq

✓

Verification of storage systems is challenging



Verification of storage systems is challenging



Implementation (Go)

Verification (Coq)

code.go

① translate

code_model.v

② spec.v

specification for crashes?

proof.v

go build

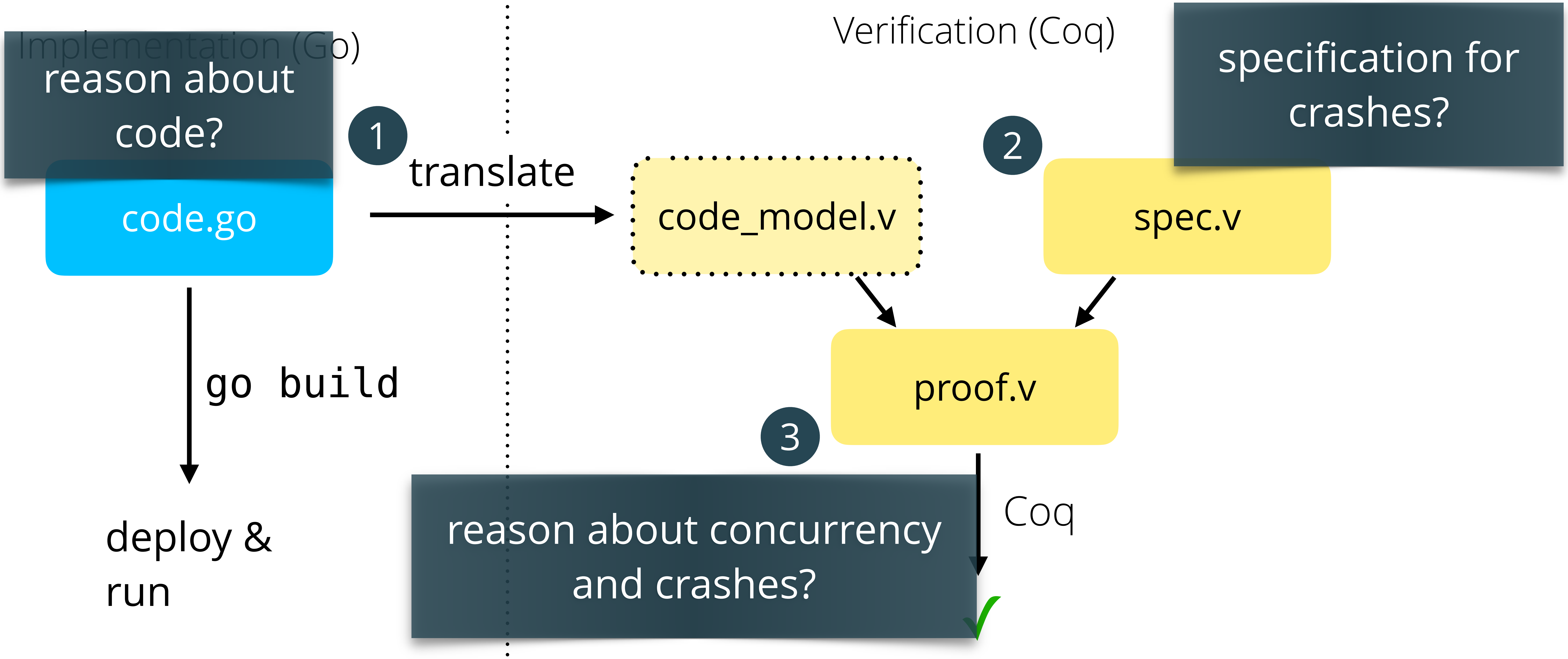
deploy & run

③

reason about concurrency and crashes?

Coq ✓

Verification of storage systems is challenging

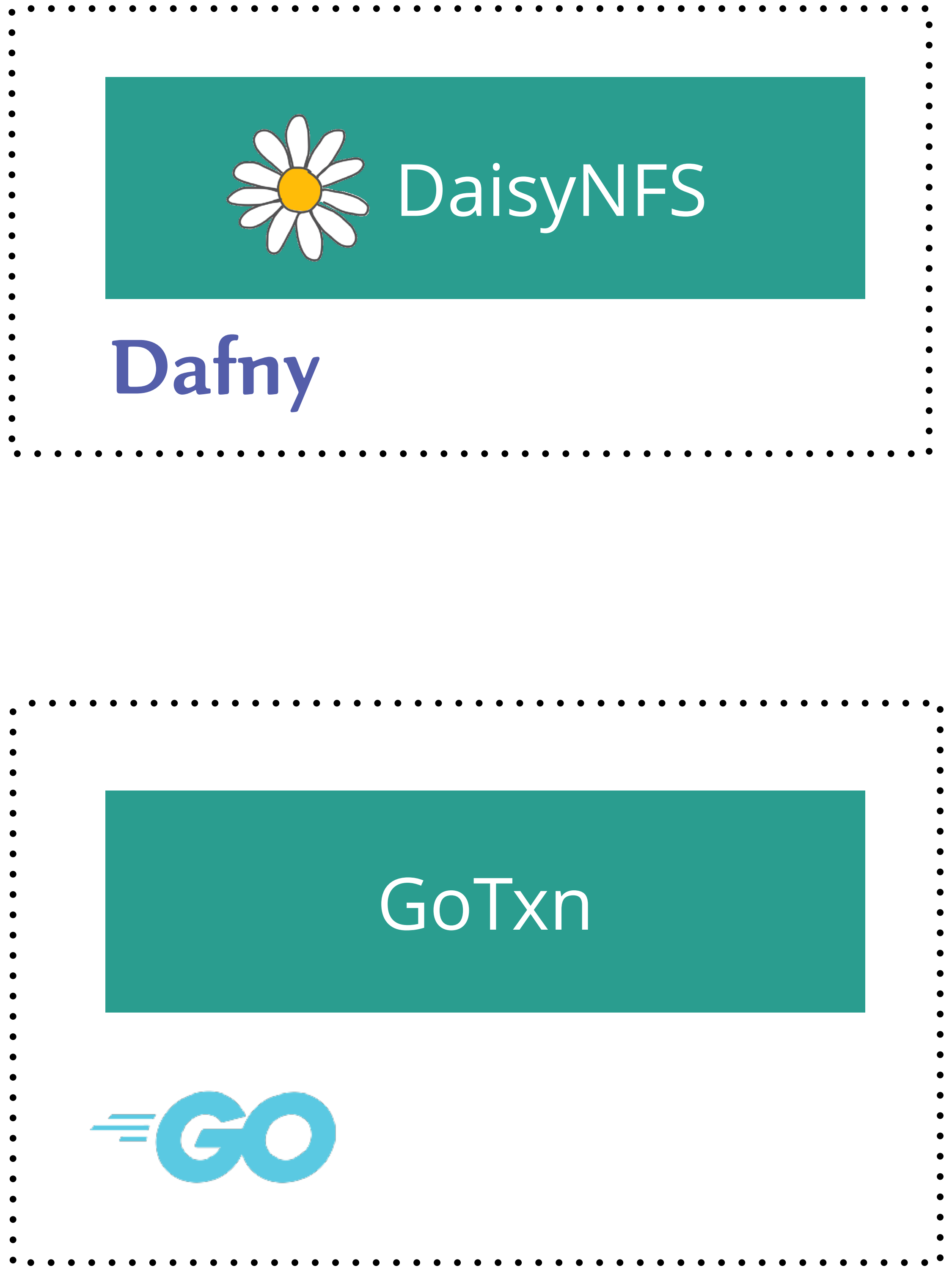


Verifying a file system is a daunting task

Still need to reason about **crash safety** + **concurrency** for a **high performance** implementation

DaisyNFS organizes the system and proof to make this manageable

DaisyNFS architecture



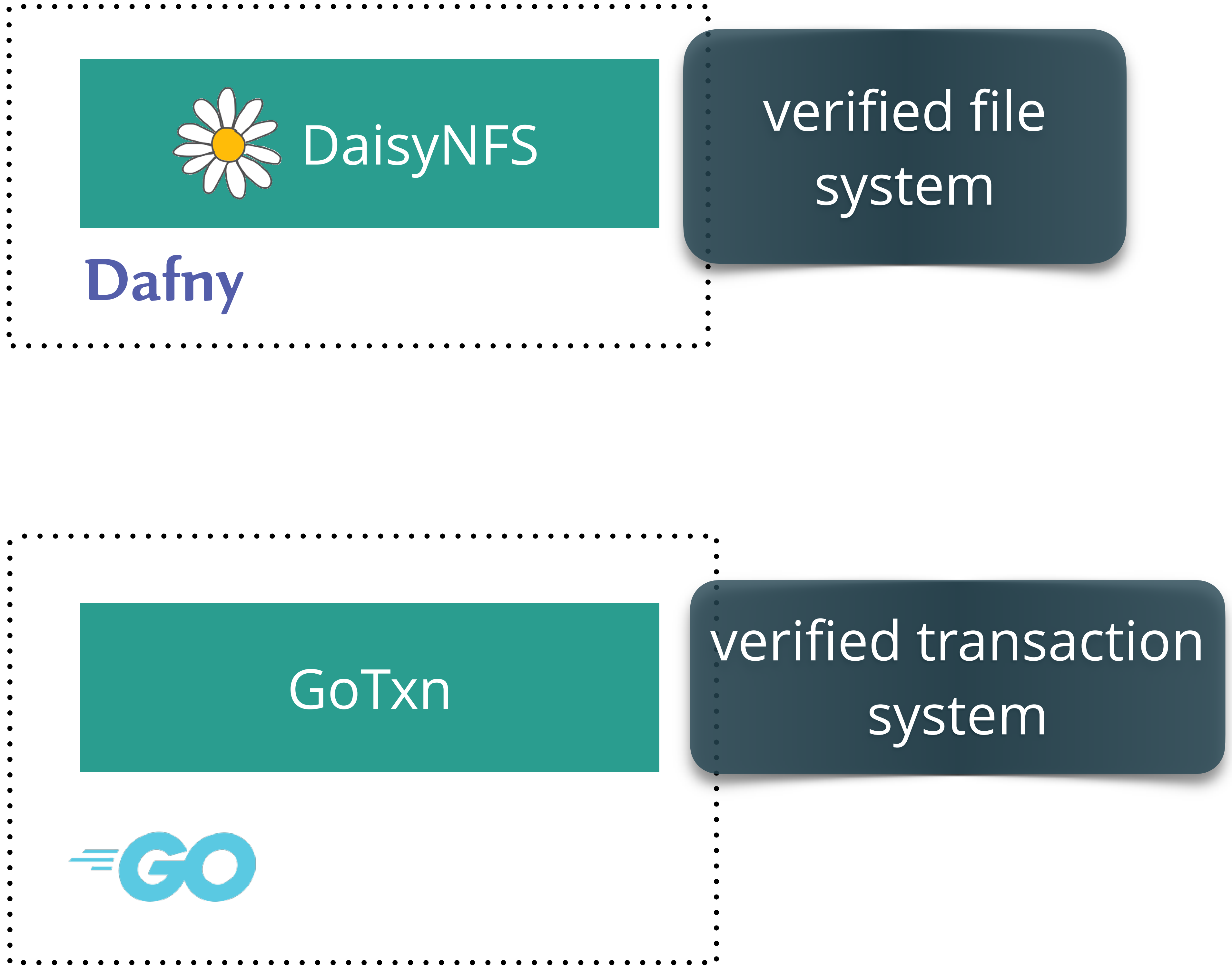
DaisyNFS

Dafny

GoTxn

GO

DaisyNFS architecture



DaisyNFS

verified file system

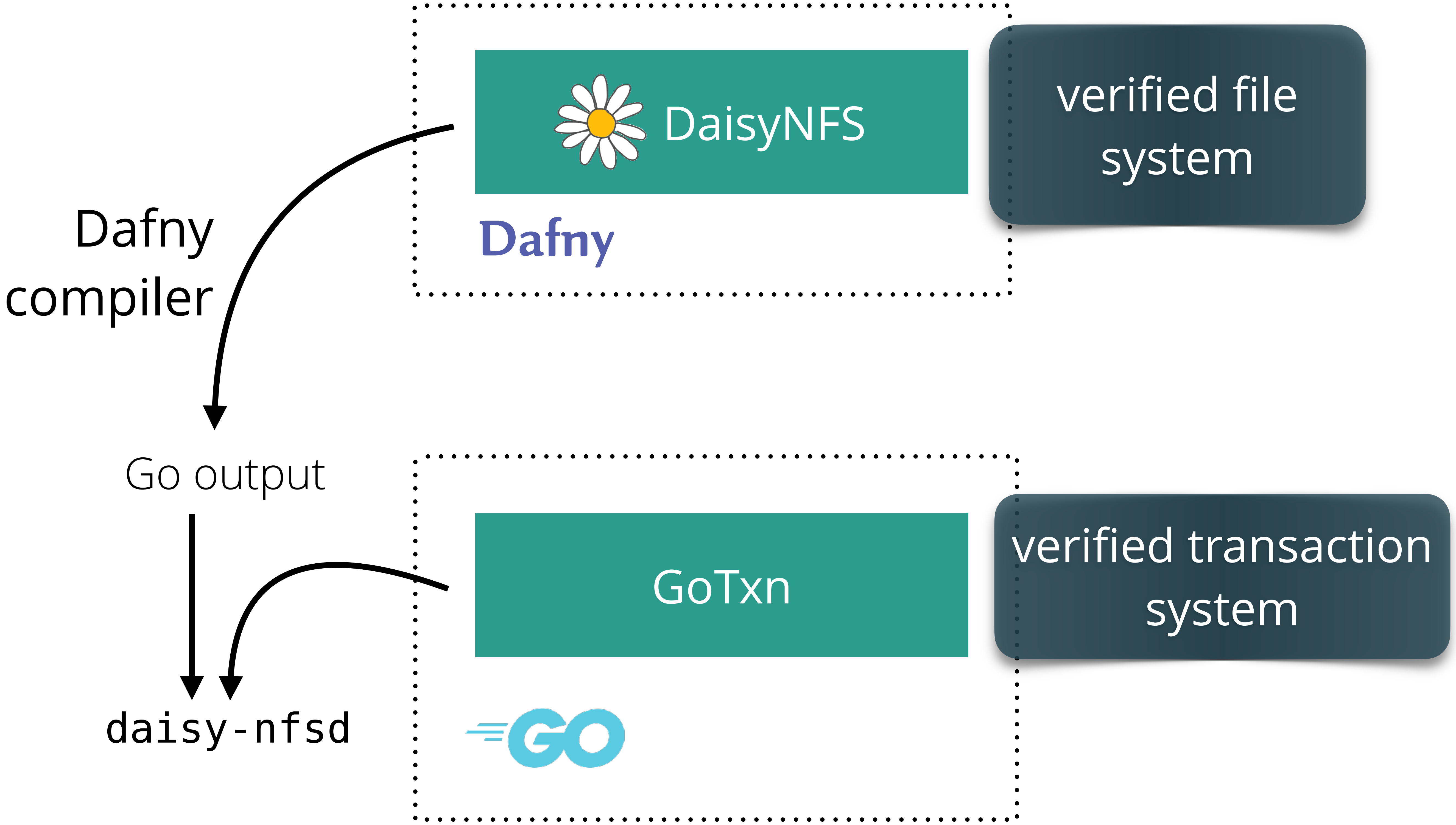
Dafny

GoTxn

verified transaction system

GO

DaisyNFS architecture



DaisyNFS

Dafny

verified file system

Dafny compiler

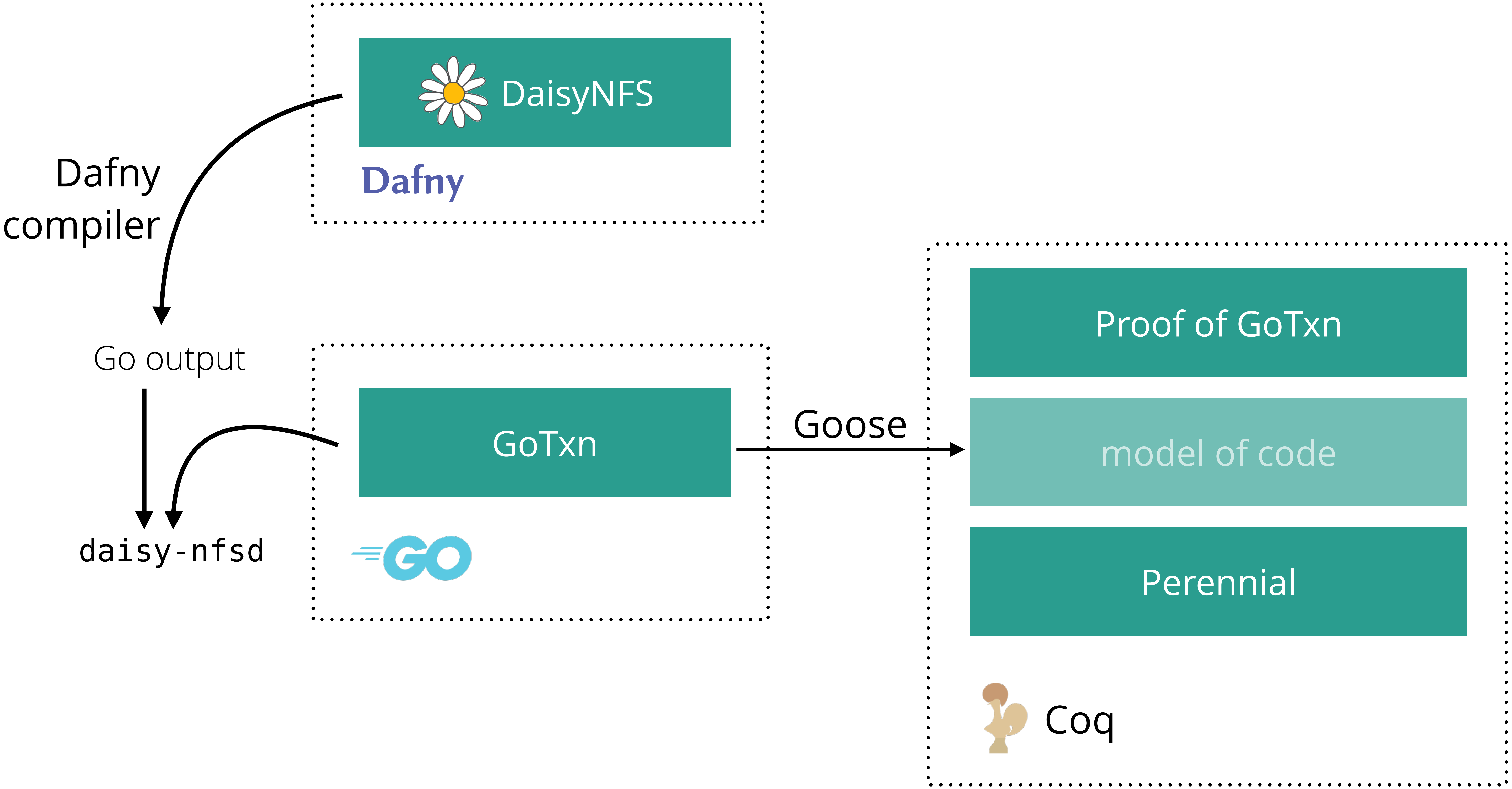
Go output

`daisy-nfsd`

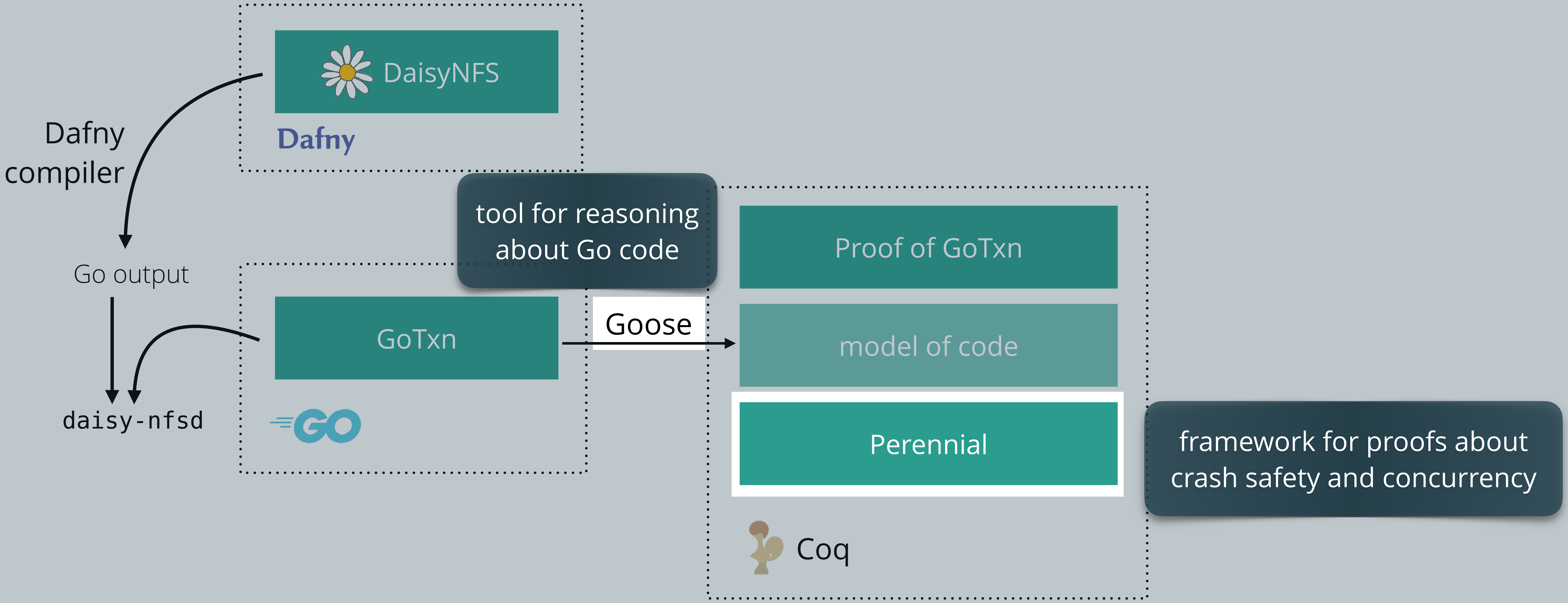
GoTxn

verified transaction system

DaisyNFS architecture



DaisyNFS architecture



DaisyNFS

Dafny

Dafny compiler

Go output

daisy-nfsd

tool for reasoning about Go code

GoTxn

GO

Goose

Proof of GoTxn

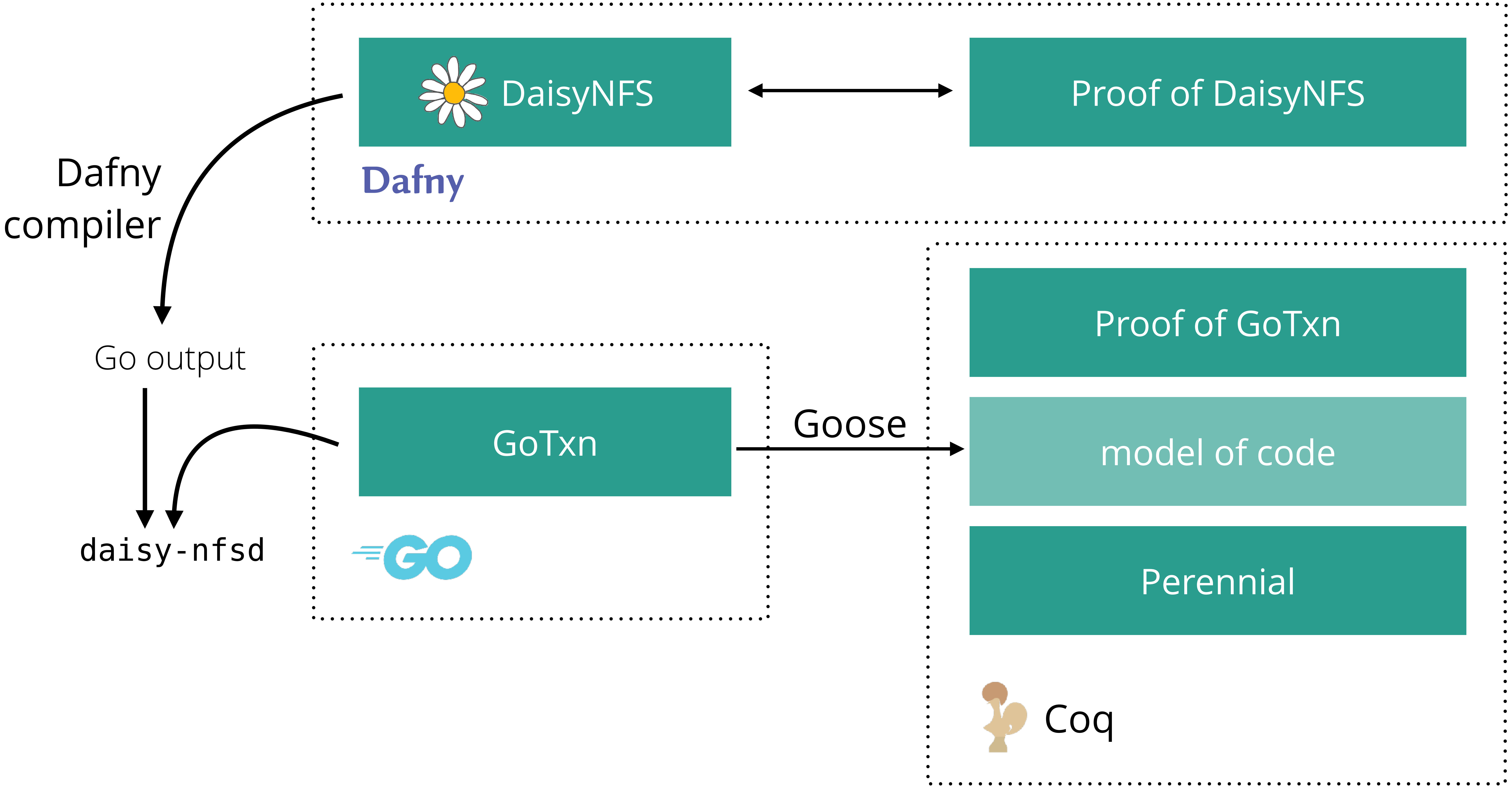
model of code

Perennial

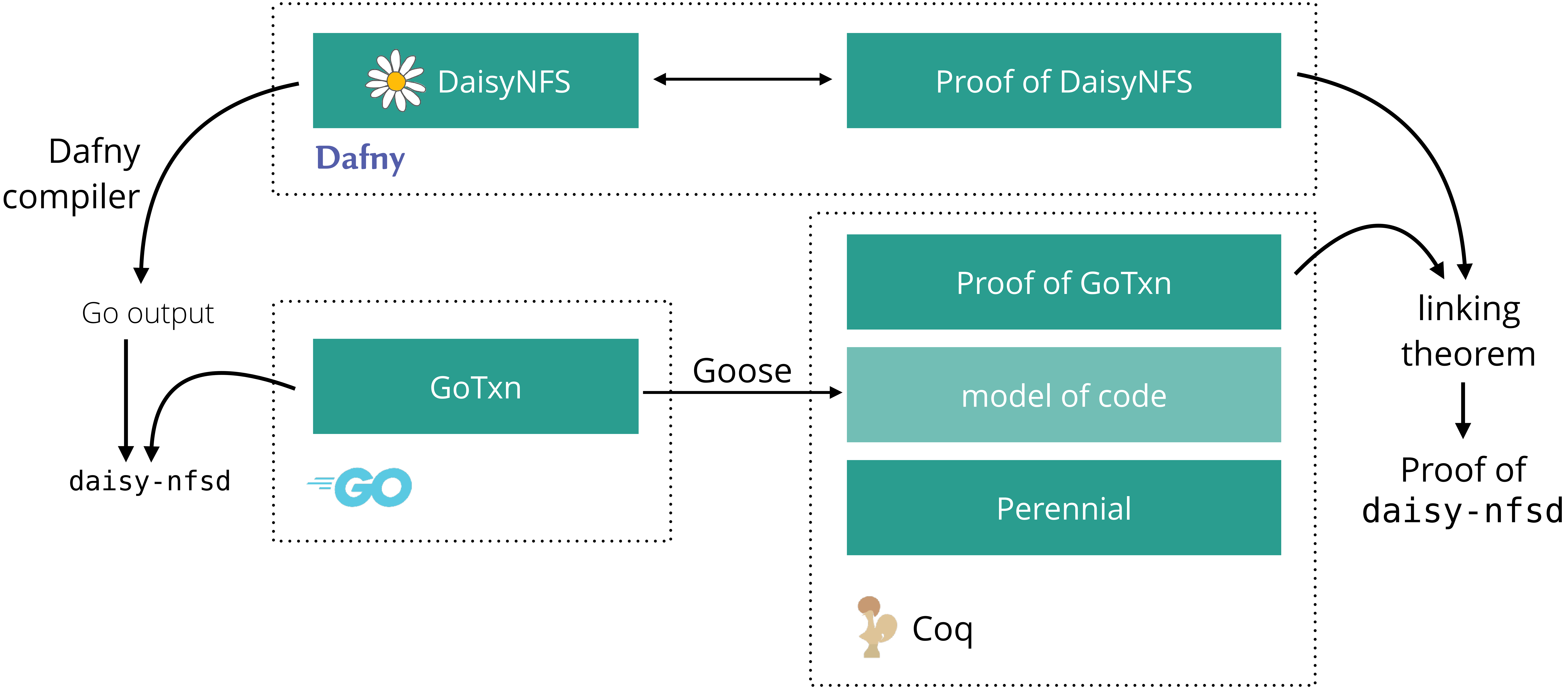
framework for proofs about crash safety and concurrency

Coq

DaisyNFS architecture



DaisyNFS architecture



DaisyNFS

Proof of DaisyNFS

Dafny

Dafny compiler

Go output

`daisy-nfsd`

GoTxn

Goose

Proof of GoTxn

model of code

Perennial

Coq

linking theorem

Proof of `daisy-nfsd`

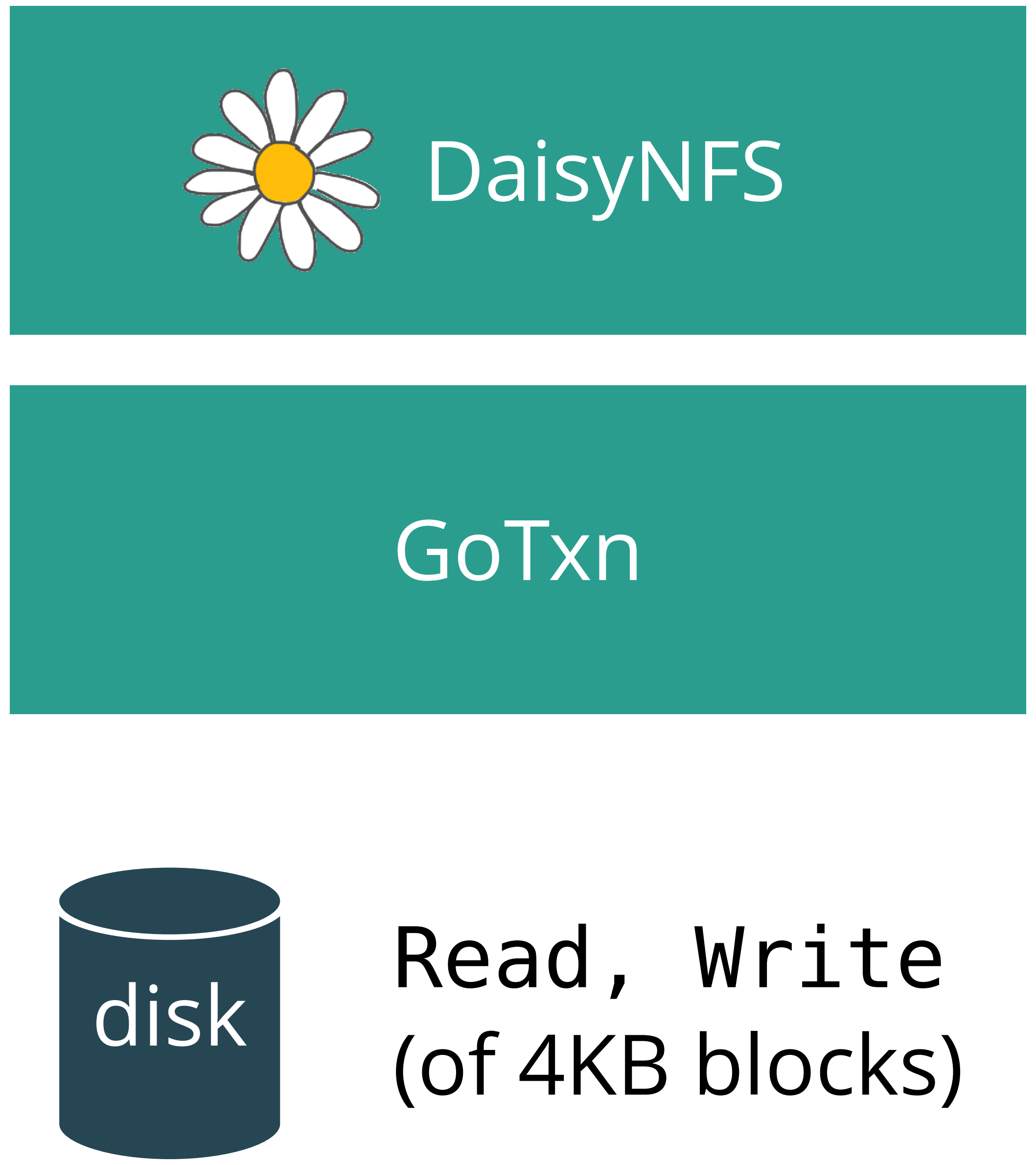
Contributions

Perennial + **Goose:** foundations for verifying storage systems written in Go

GoTxn: handles crash safety and concurrency to enable sequential reasoning

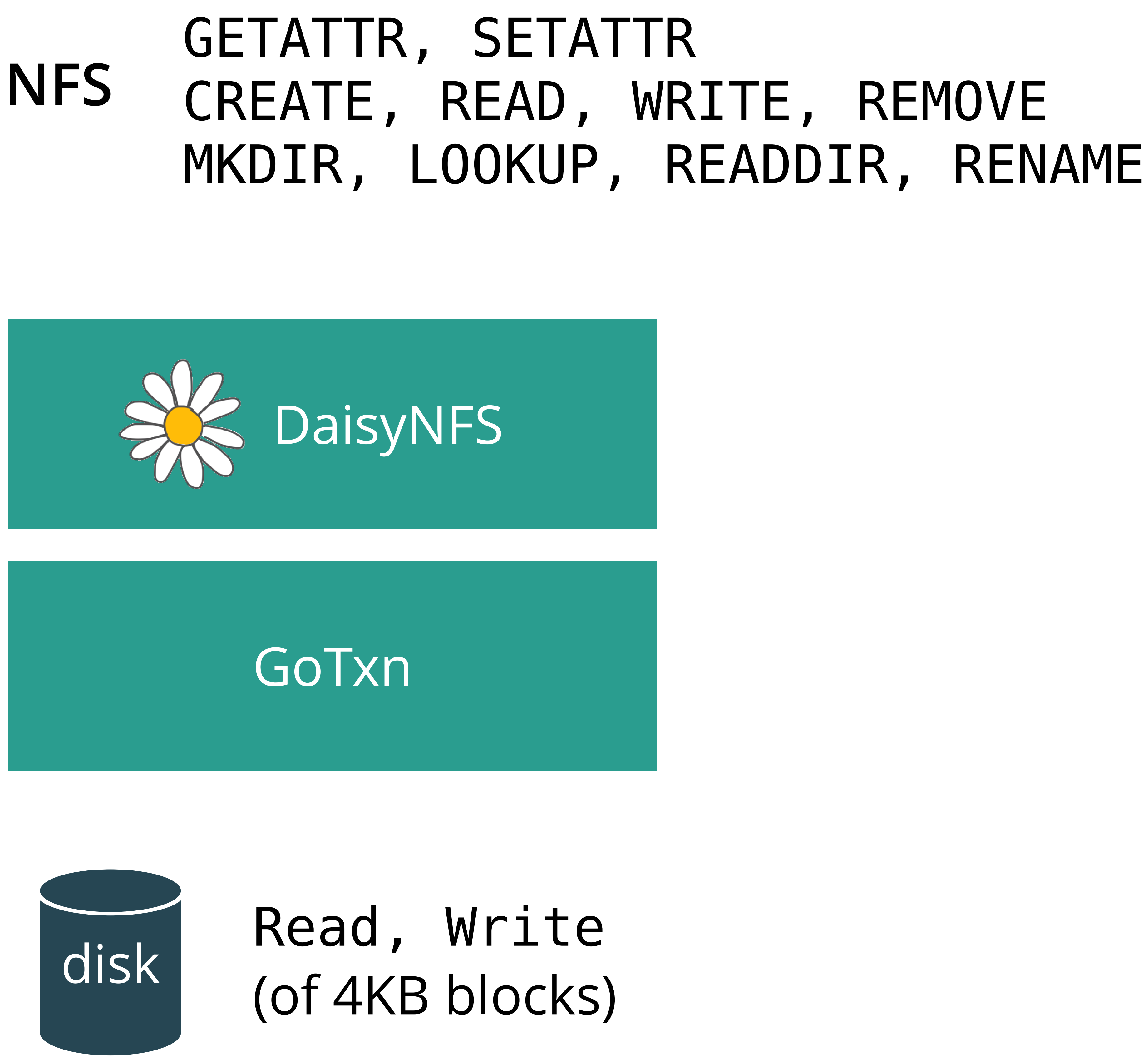
DaisyNFS: a verified concurrent, crash-safe file system

What did we prove?



`Read, Write`
(of 4KB blocks)

What did we prove?



NFS

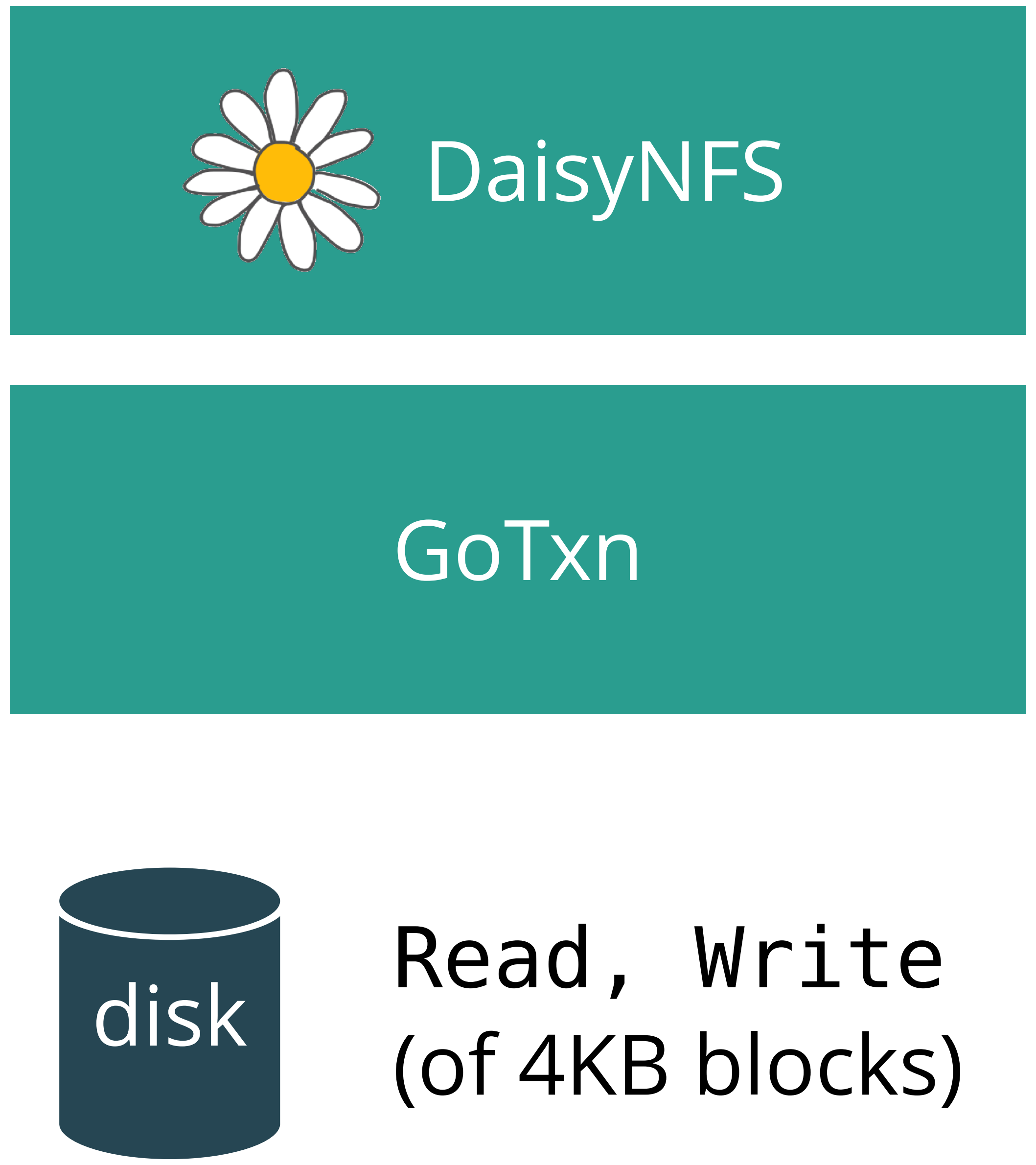
`GETATTR, SETATTR`
`CREATE, READ, WRITE, REMOVE`
`MKDIR, LOOKUP, READDIR, RENAME`

`Read, Write`
(of 4KB blocks)

What did we prove?

NFS

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



`Read, Write`
(of 4KB blocks)

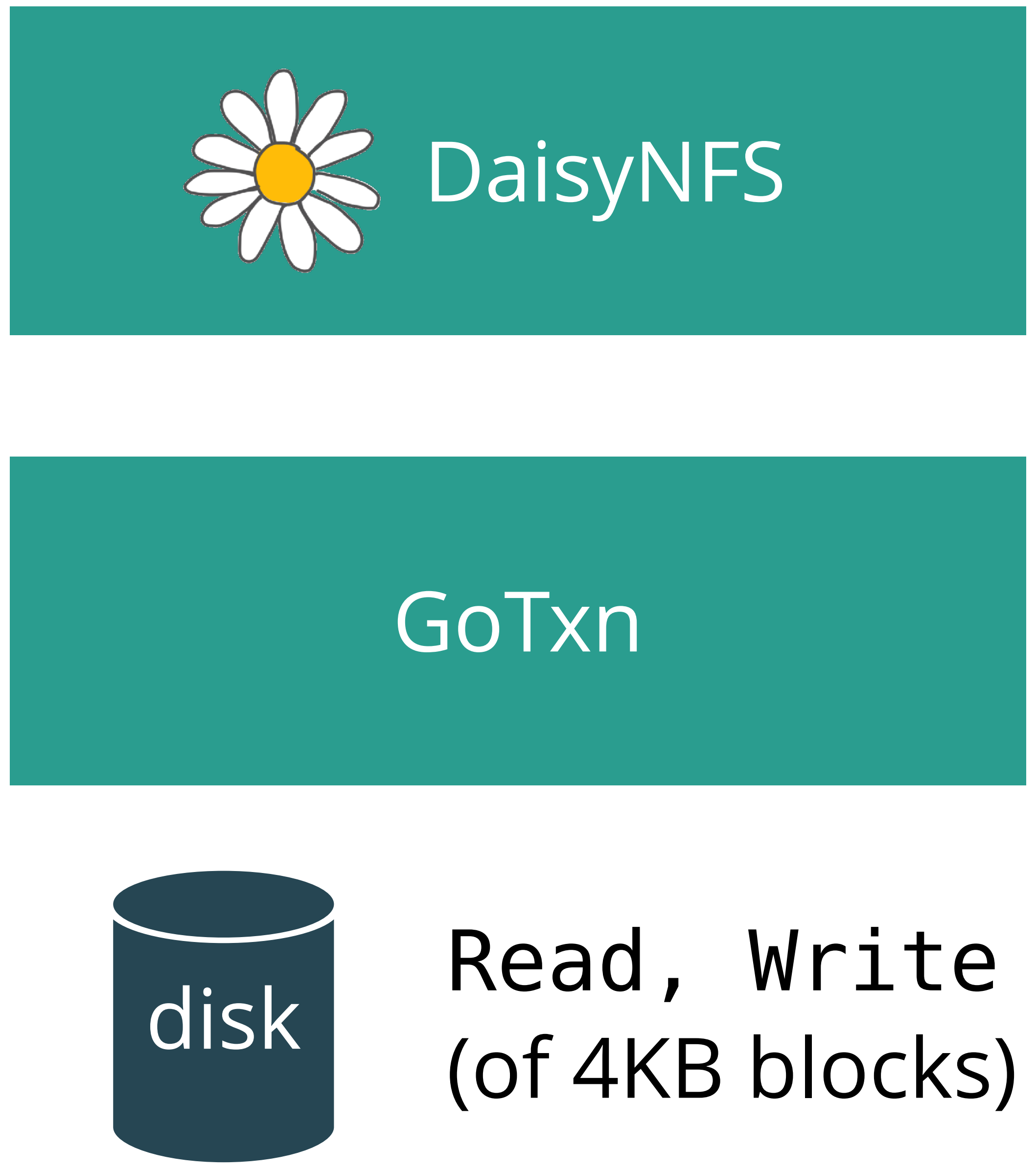
Theorem: Every NFS operation appears to execute atomically and correctly, despite crashes and concurrency.

Design and implementation of DaisyNFS

Verifying a high-performance transaction system

Evaluating DaisyNFS

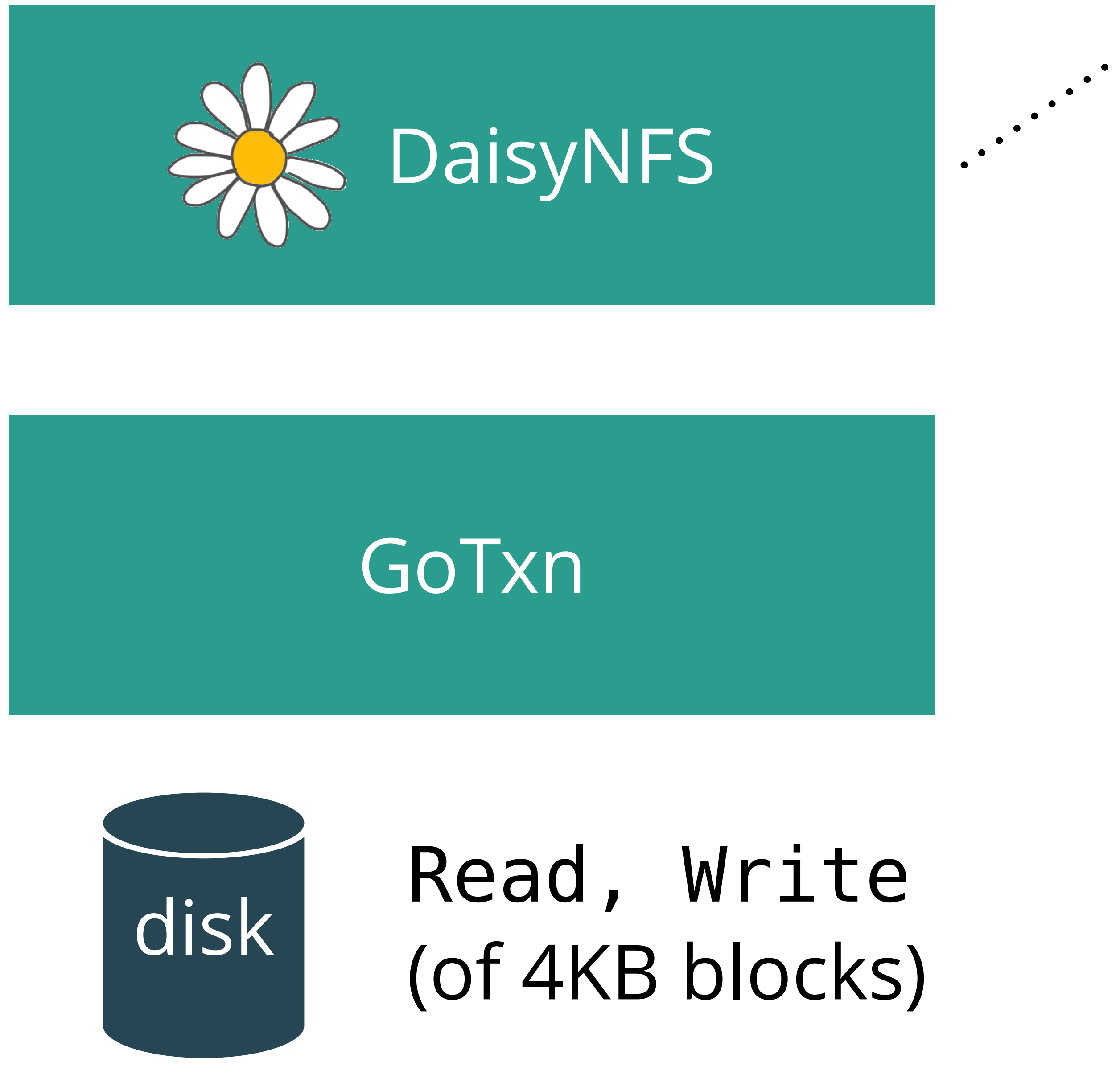
DaisyNFS accesses the disk through a transaction system



`Read, Write`
(of 4KB blocks)

DaisyNFS accesses the disk through a transaction system

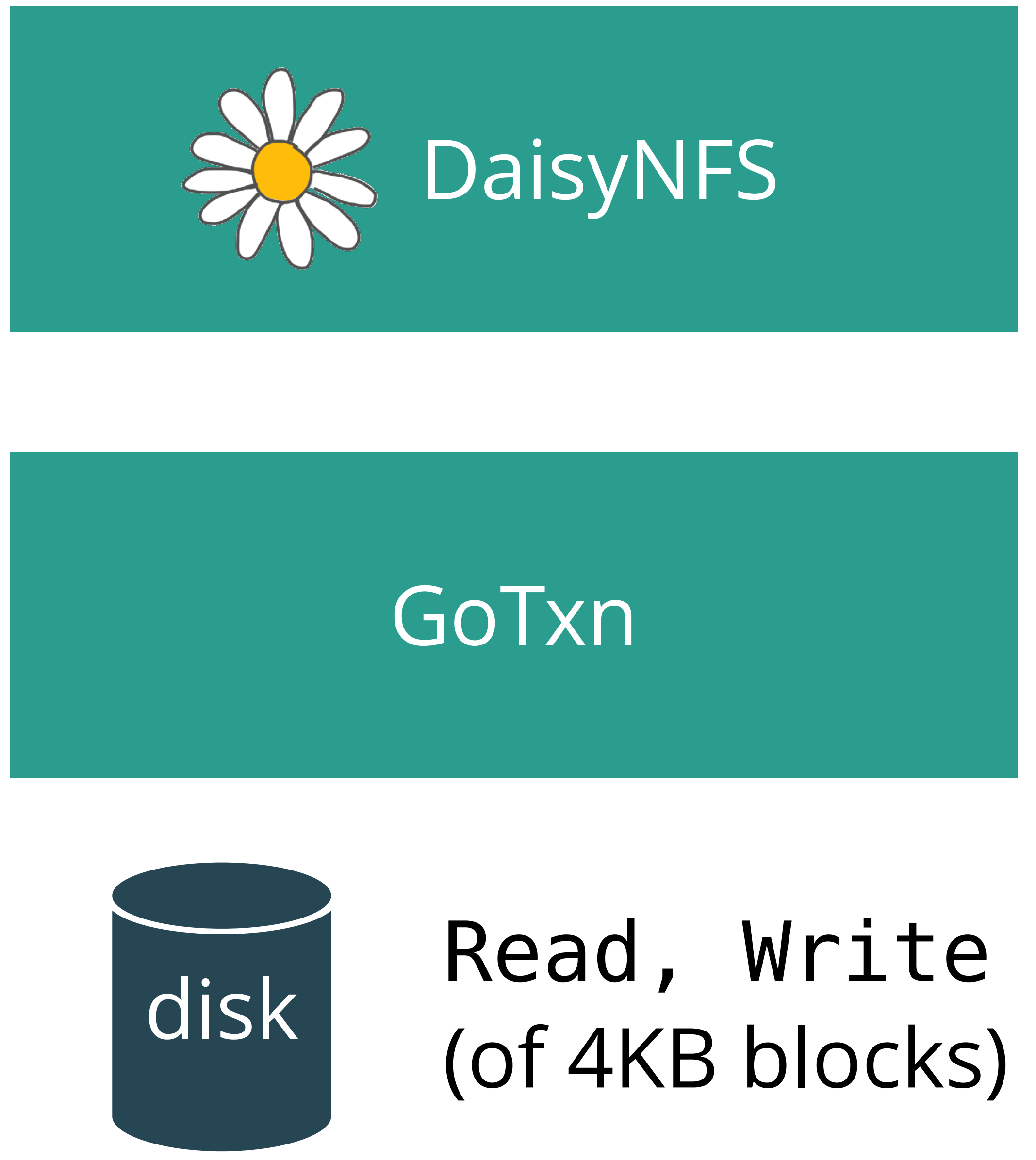
```
tx := Begin()
v := tx.Read(3)
tx.Write(7, v)
tx.Commit()
```



`Read, Write`
(of 4KB blocks)

Each operation runs within a transaction

DaisyNFS accesses the disk through a transaction system



```
tx := Begin()
v := tx.Read(3)
tx.Write(7, v)
tx.Commit()
```

Each operation runs within a transaction

```
func Begin() *Txn

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

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

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

Transactions isolate difficult reasoning and leave simpler sequential reasoning

```
tx := Begin()
v := tx.Read(3)
tx.Write(7, v)
tx.Commit()
```

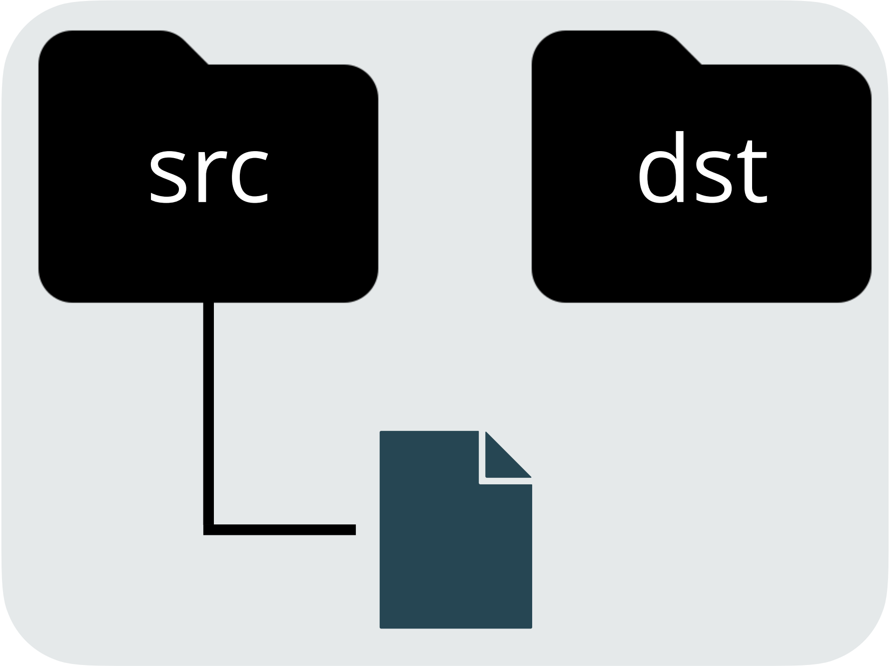
Operations are atomic — without worrying about crash safety or concurrency

GoTxn

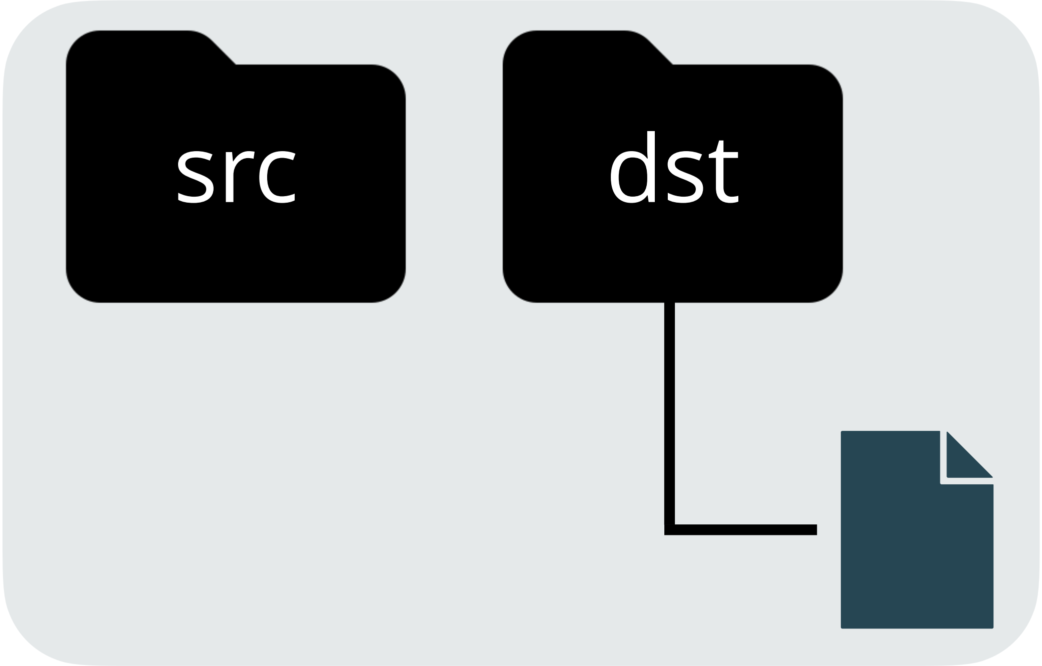
Fine-grained concurrency and crashes mean things are hard

Crash atomicity is a key correctness challenge in file systems

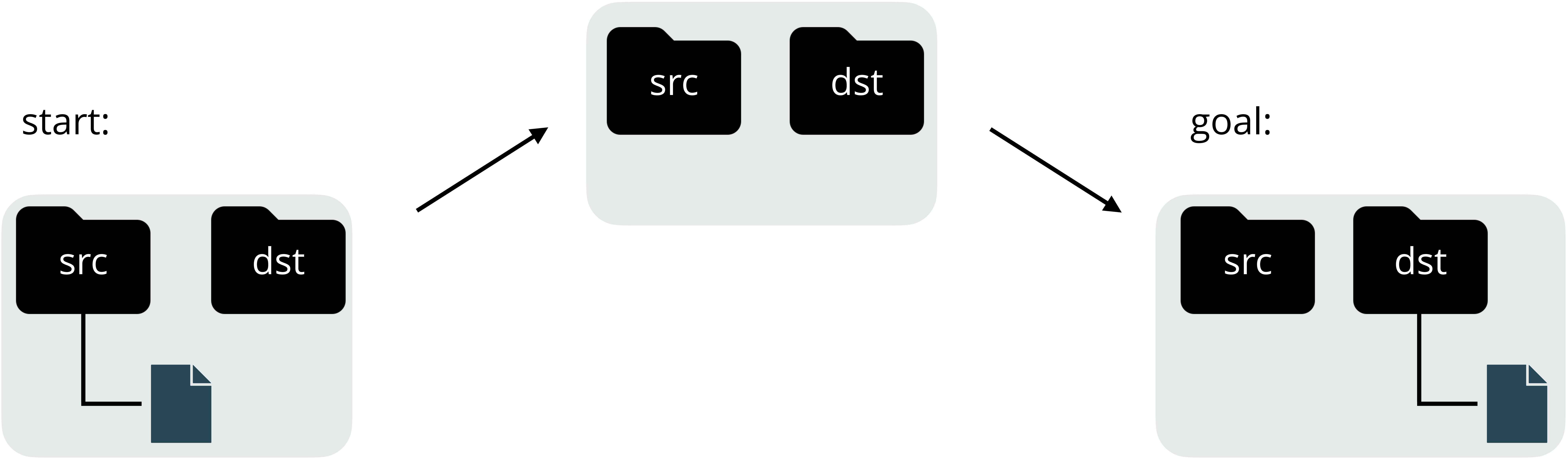
start:



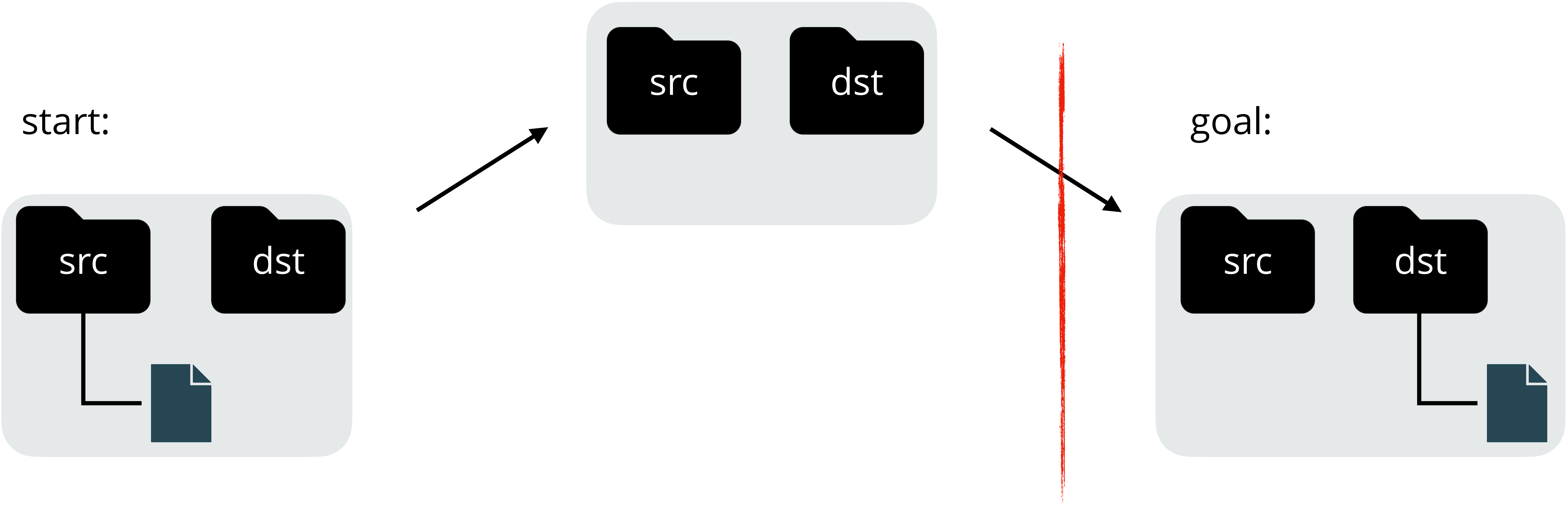
goal:



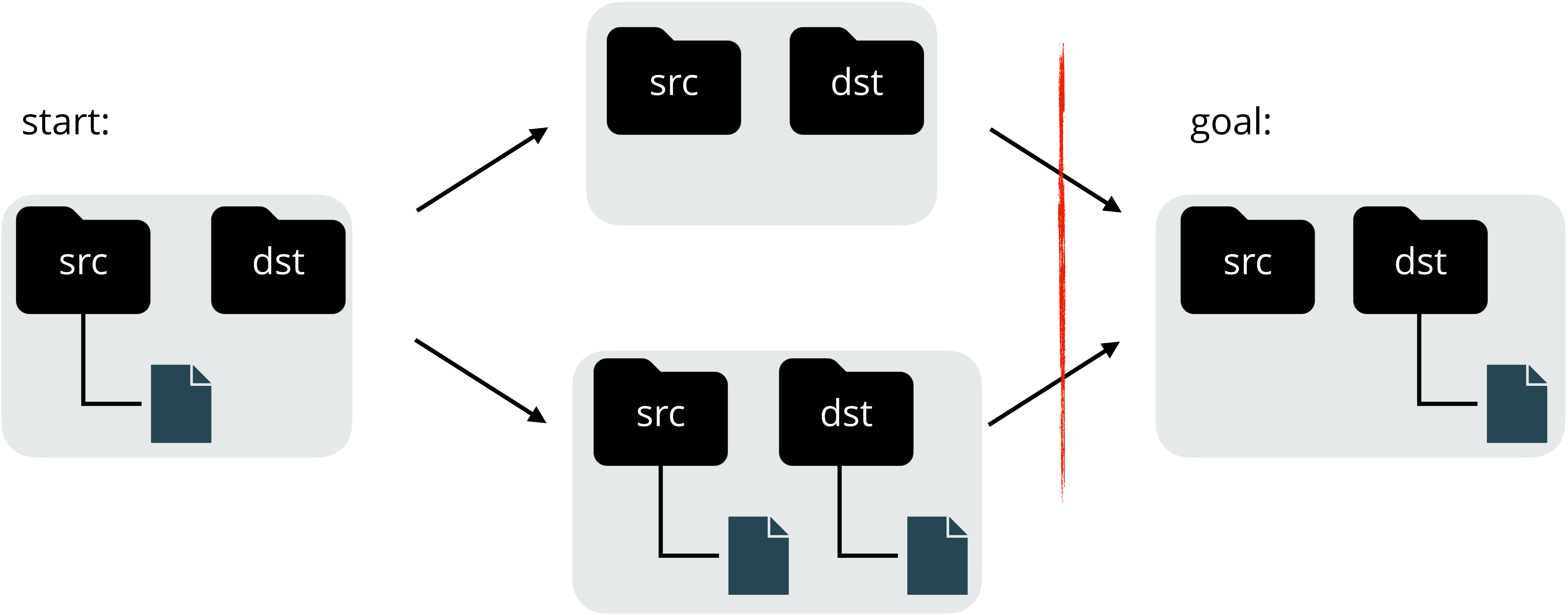
Crash atomicity is a key correctness challenge in file systems



Crash atomicity is a key correctness challenge in file systems



Crash atomicity is a key correctness challenge in file systems



Common approach is to use journaling

One solution: *journaling* is a way to write multiple values atomically

Simplifies crash atomicity but journaling is subtle to use correctly

Journal gathers up writes and issues them at once

```
op := Begin()
v := op.Read(3)
op.Write(7, v)
op.Write(8, v)
op.Commit()
```

Journal gathers up writes and issues them at once

```
op := Begin()
v := op.Read(3)
op.Write(7, v)
op.Write(8, v)
op.Commit()
```

Journal gathers up writes and issues them at once

```
op := Begin()
v := op.Read(3)
op.Write(7, v)
op.Write(8, v)
op.Commit()
```

both writes go to disk together on `Commit()`

Journal gathers up writes and issues them at once

```
op := Begin()
v := op.Read(3)
op.Write(7, v)
op.Write(8, v)
op.Commit()
```

both writes go to disk together on `Commit()`

Journal gathers up writes and issues them at once

```
op := Begin()
v := op.Read(3)
op.Write(7, v)
op.Write(8, v)
op.Commit()
```

code needs to guarantee other threads don’t touch 3, 7, 8

both writes go to disk together on `Commit()`

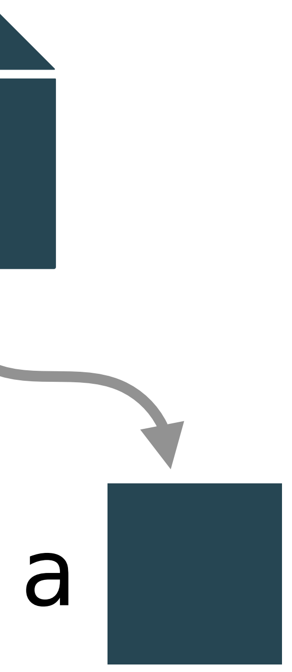
Journal gathers up writes and issues them at once

```
op := Begin()
v := op.Read(3)
op.Write(7, v)
op.Write(8, v)
op.Commit()
```

code needs to guarantee other threads don’t touch 3, 7, 8

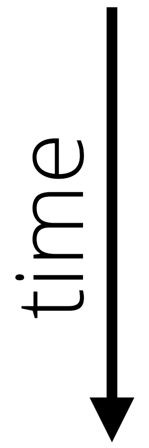
both writes go to disk together on `Commit()`

Potential bug even with journaling

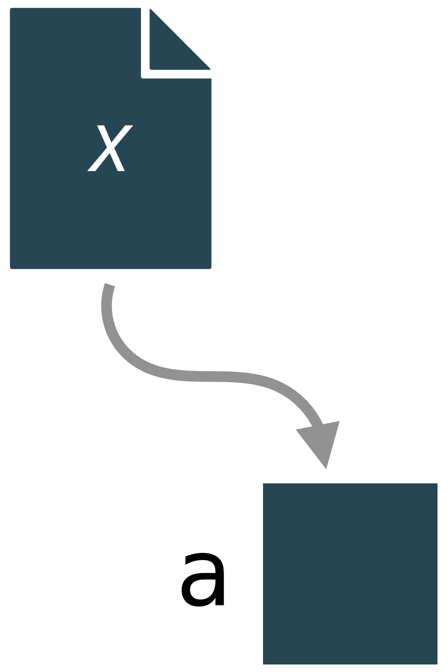


Potential bug even with journaling

deleting file *x*:



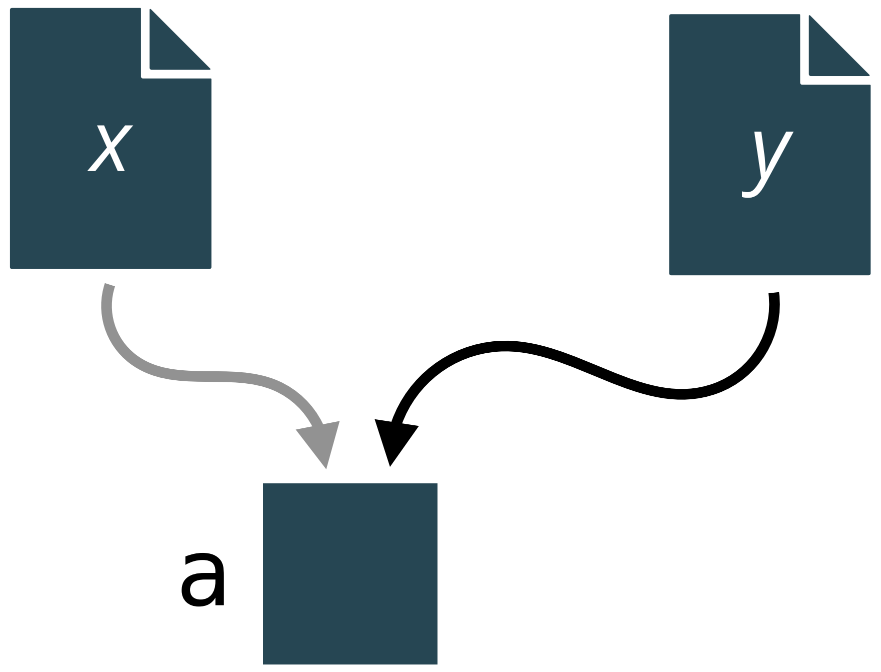
```
op := Begin()
…
free(a)



op.Commit()
```



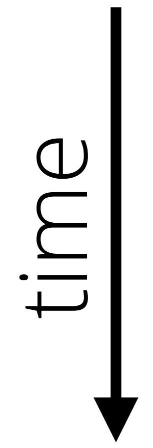
Potential bug even with journaling

time

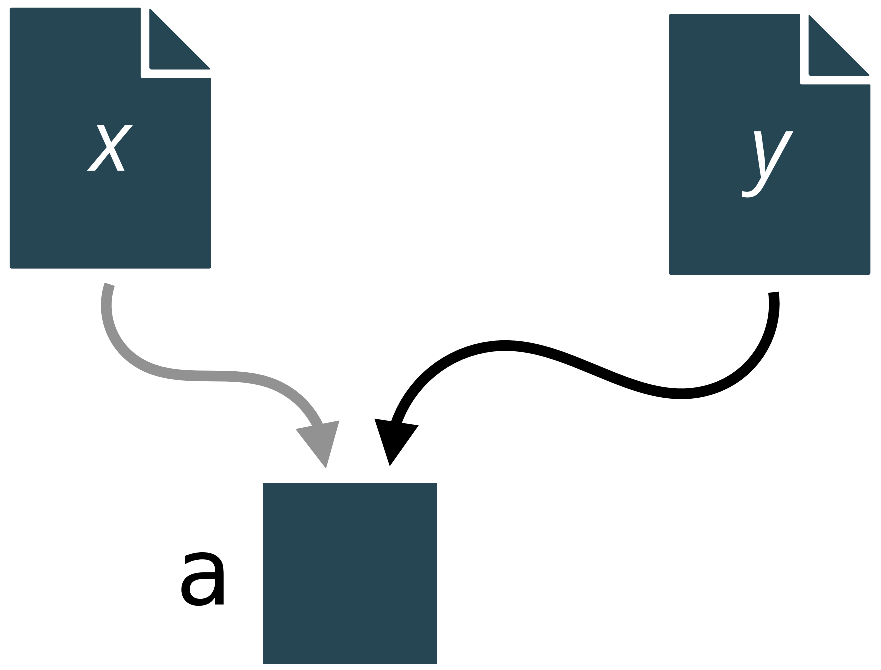
```
deleting file x:      appending to y:

op := Begin()
…                     op := Begin()
free(a)               alloc() → a
                      …
                      op.Commit()


op.Commit()
```



Potential bug even with journaling



deleting file *x*:	appending to *y*:
`op := Begin()`	
…	`op := Begin()`
`free(a)`	`alloc() → a`
	…
	`op.Commit()`
	crash
`op.Commit()`	



Potential bug even with journaling

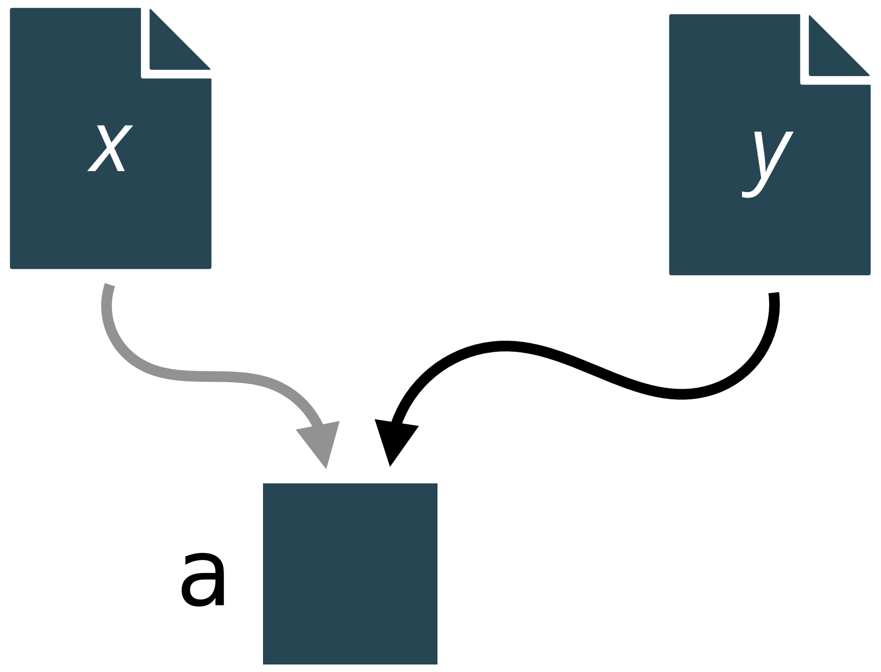
time

deleting file *x*: / appending to *y*:

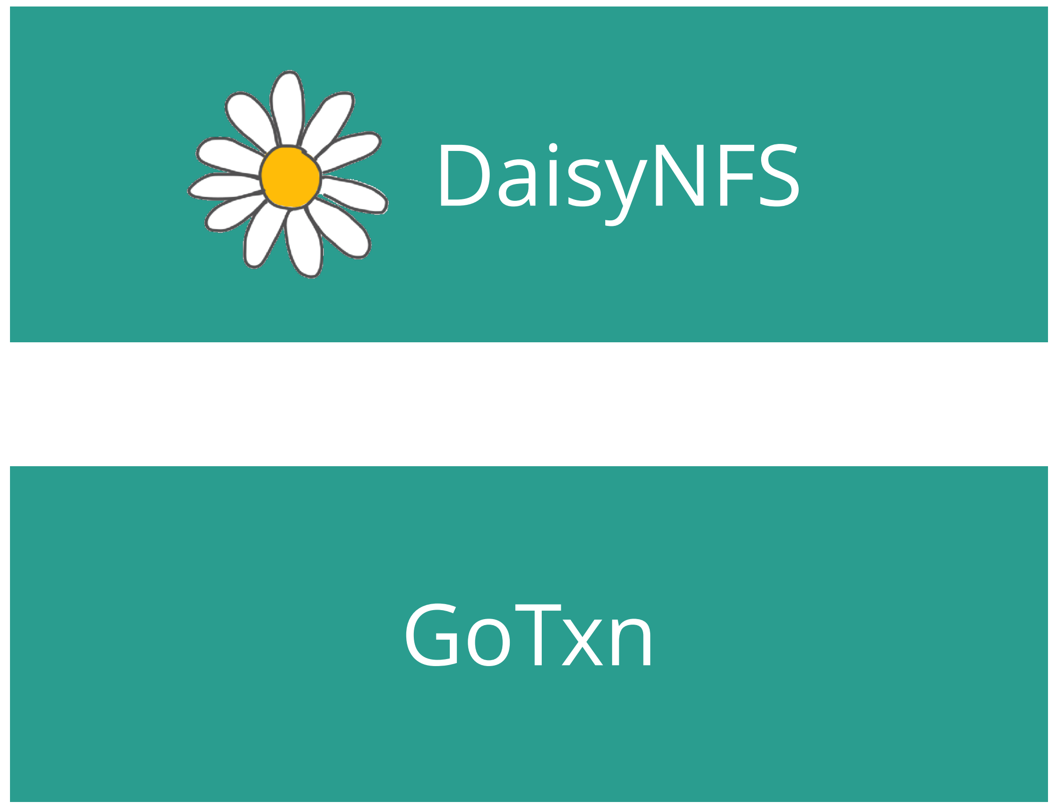
```
op := Begin()
…                  op := Begin()
free(a)            alloc() → a
                   …
                   op.Commit()
```

crash → both *x* and *y* contain block a

```
op.Commit()
```



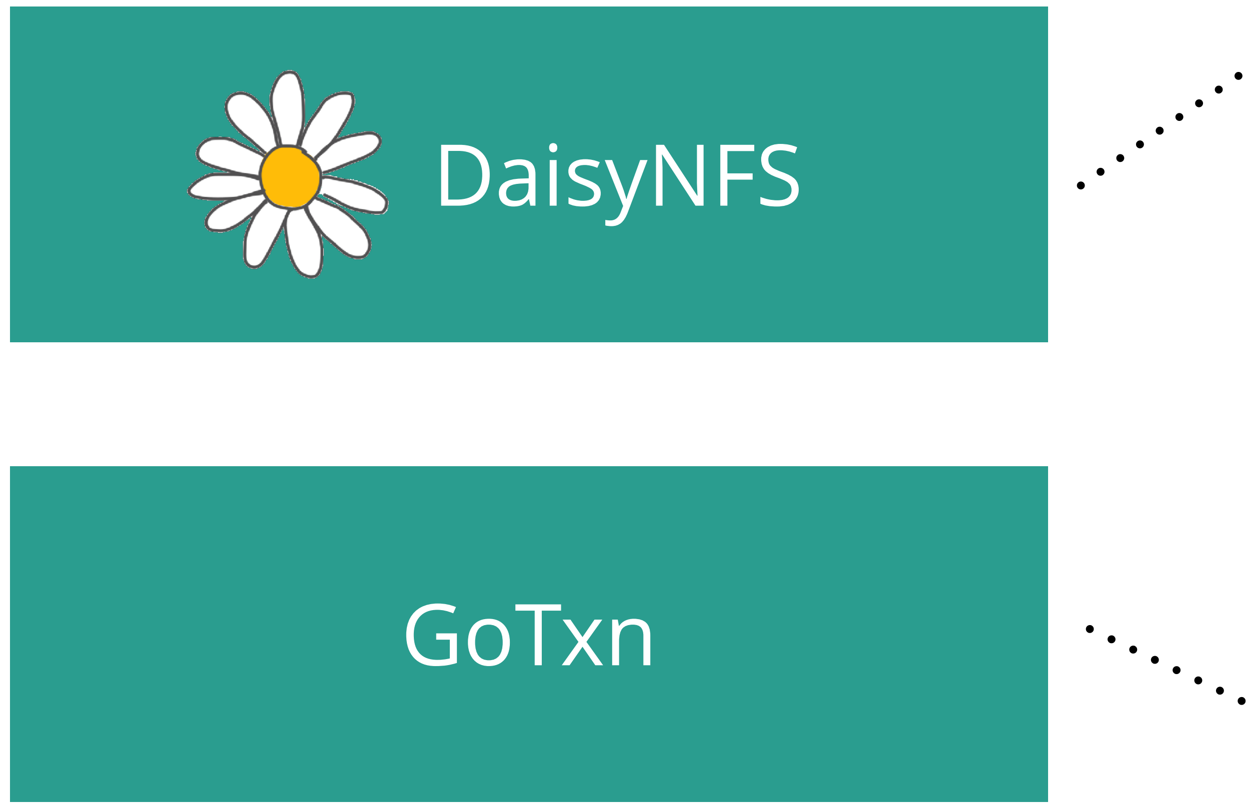
Designed a file system around transactions



```
func Begin() *Txn

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

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

Unlike journaling, provides strong **atomicity** guarantee

Designed a file system around transactions



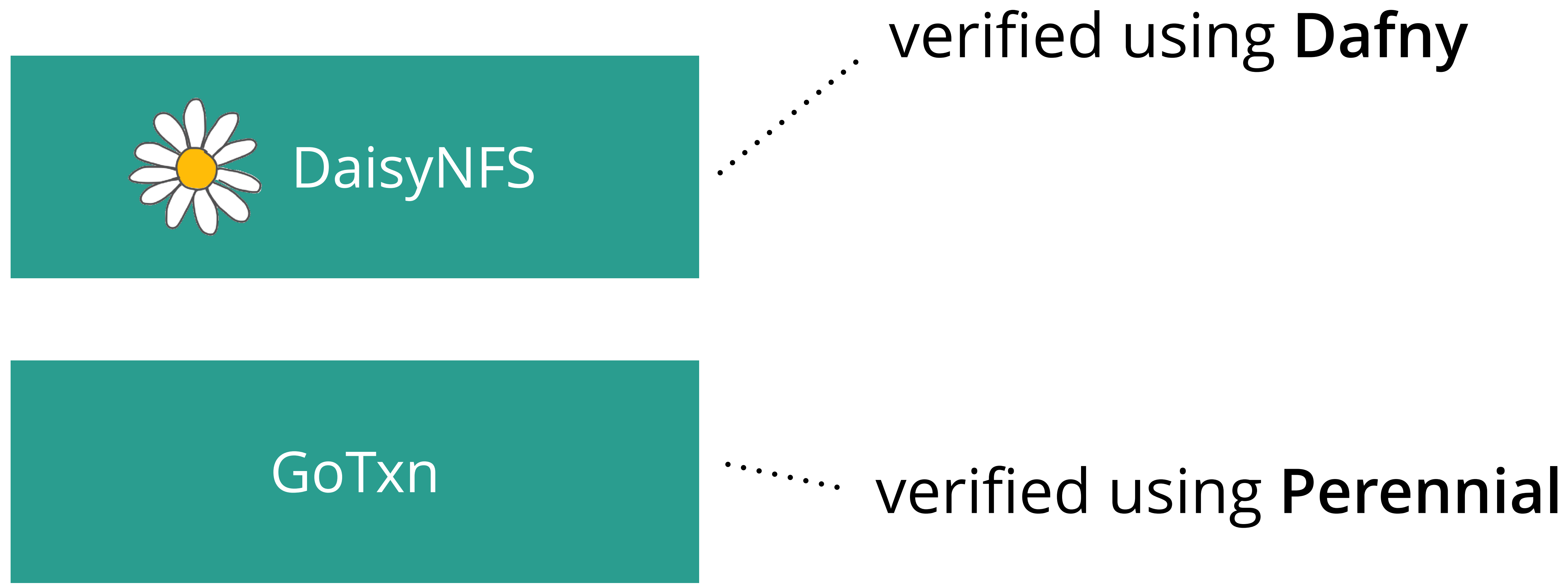
```
tx := Begin()
v := tx.Read(3)
tx.Write(7, v)
tx.Commit()
```

```
func Begin() *Txn

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

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

Design that fits all file-system code into transactions

Unlike journaling, provides strong **atomicity** guarantee

Transactions are so sequential that we verify them without a concurrency framework



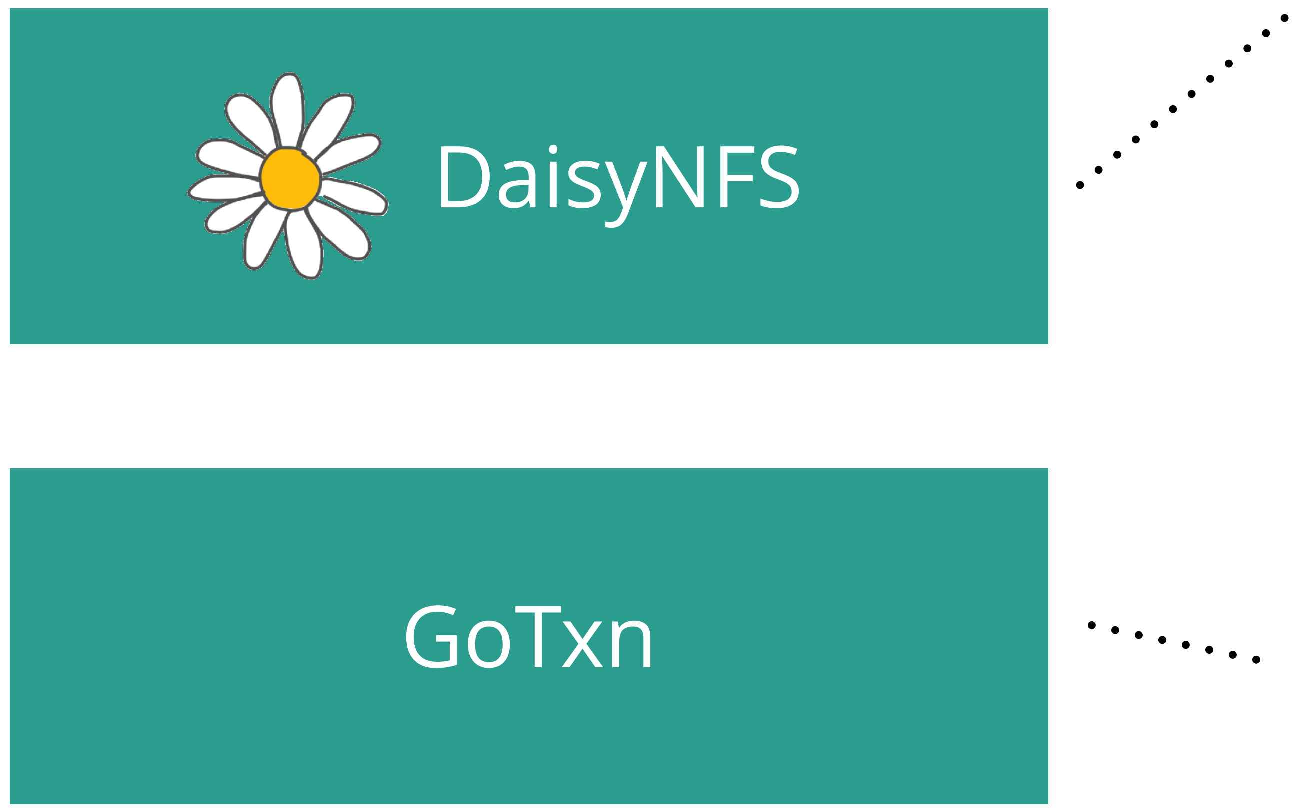
DaisyNFS

verified using **Dafny**

GoTxn

verified using **Perennial**

Transactions are so sequential that we verify them without a concurrency framework



verified using **Dafny**

existing, widely-used verification system

verified using **Perennial**

our own custom infrastructure

Transactions are so sequential that we verify them without a concurrency framework

DaisyNFS

verified using **Dafny**

existing, widely-used verification system
2x as much proof as code

GoTxn

verified using **Perennial**

our own custom infrastructure
20x as much proof as code

Sequential reasoning helps because each operation needs to do a lot

internal view

user's view

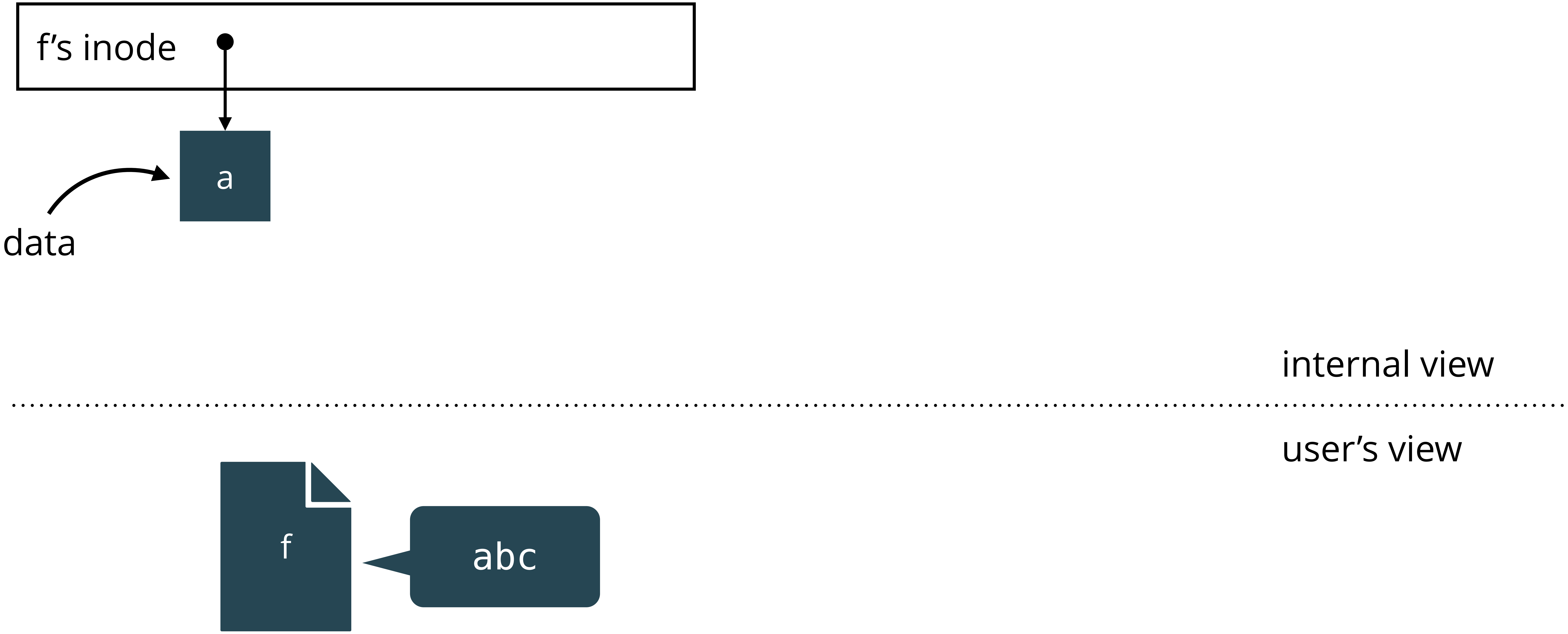
Sequential reasoning helps because each operation needs to do a lot

f’s inode

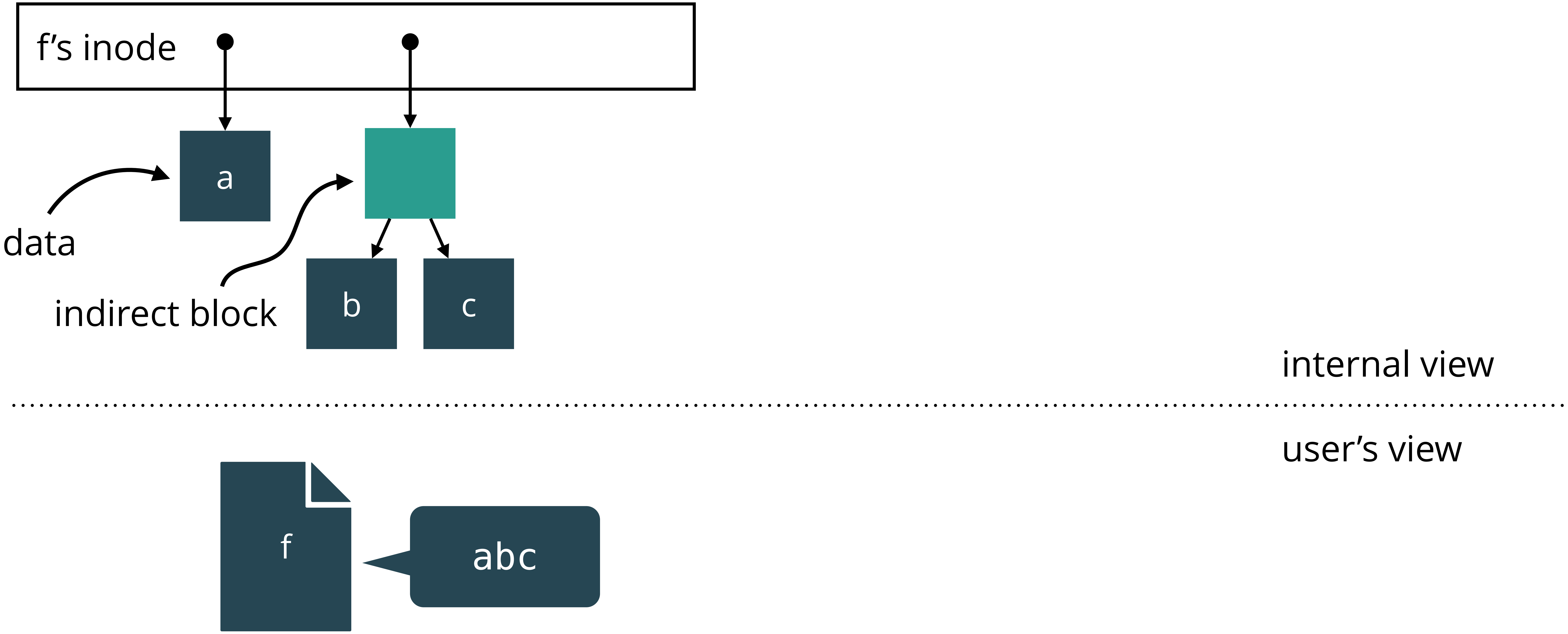
internal view

user’s view

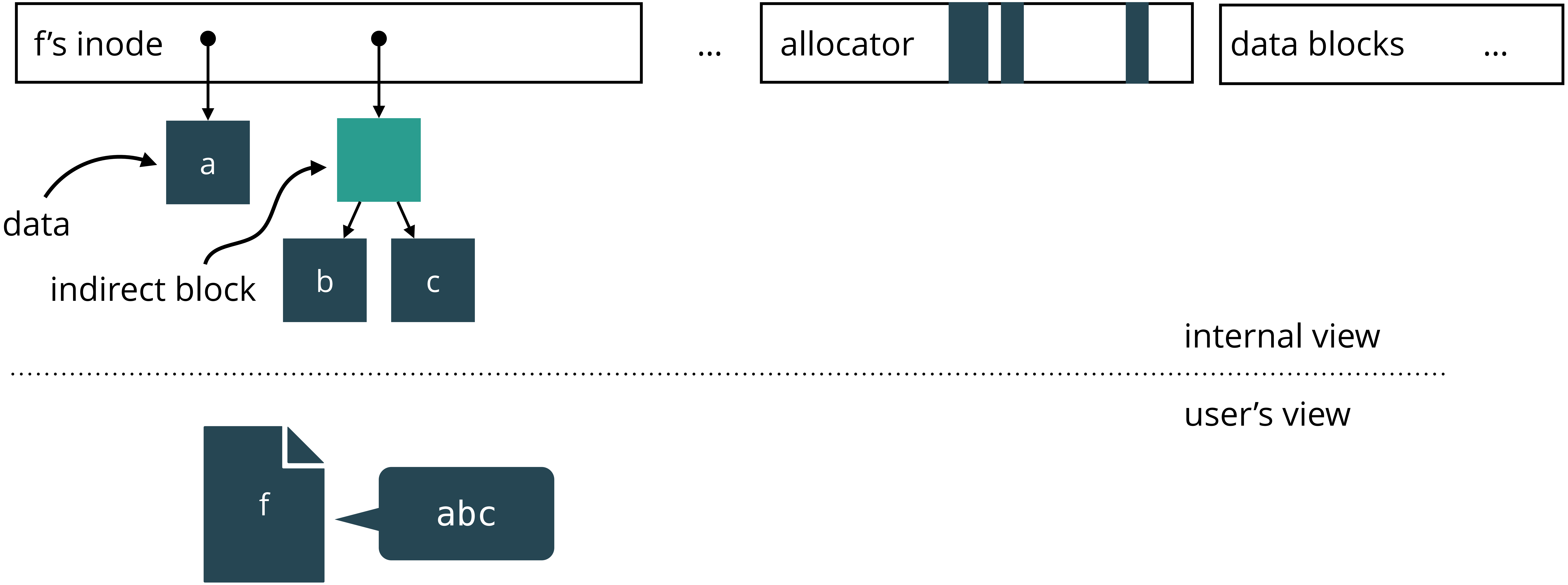
Sequential reasoning helps because each operation needs to do a lot



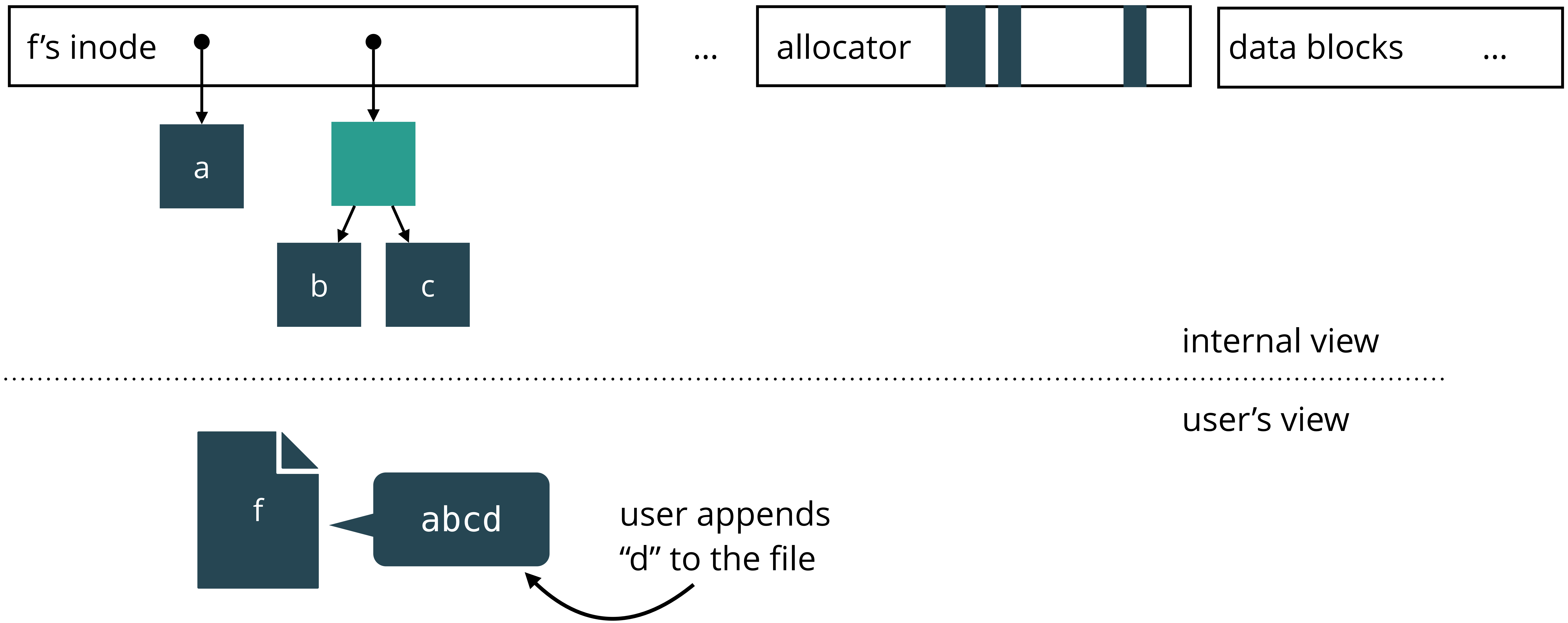
Sequential reasoning helps because each operation needs to do a lot



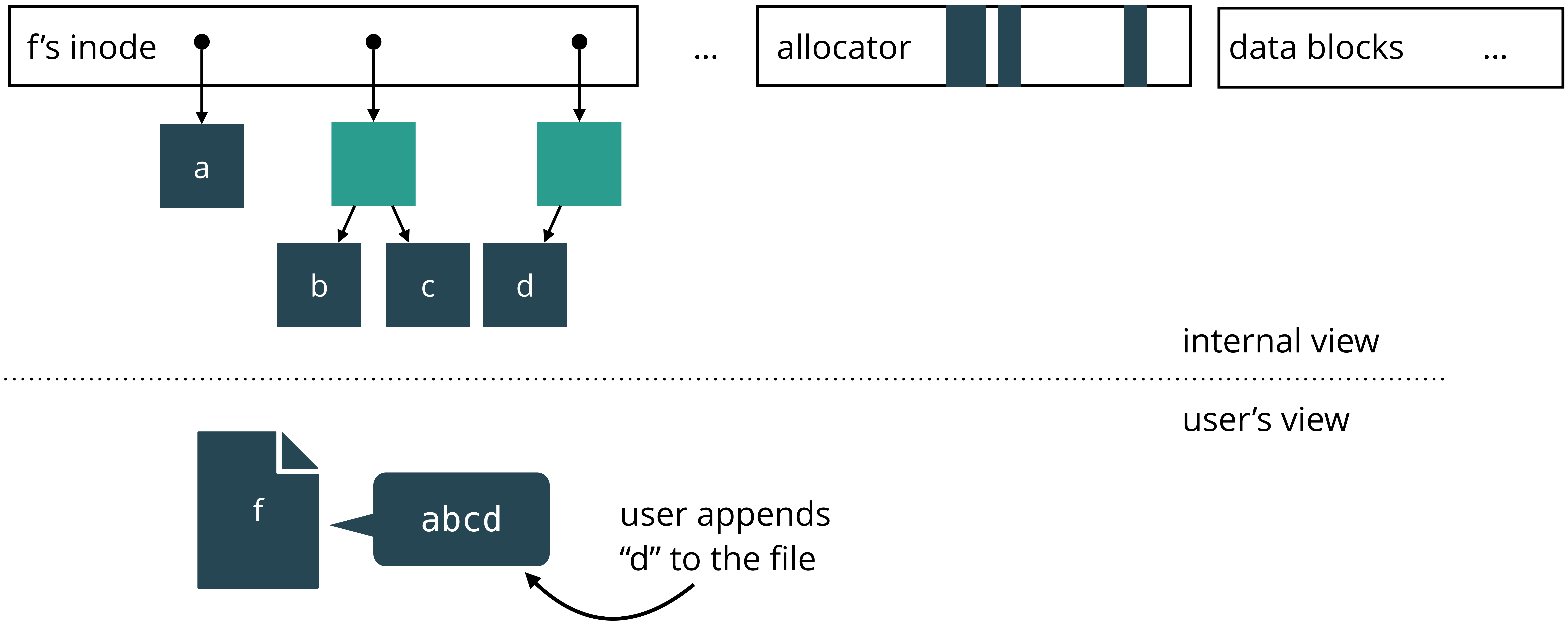
Sequential reasoning helps because each operation needs to do a lot



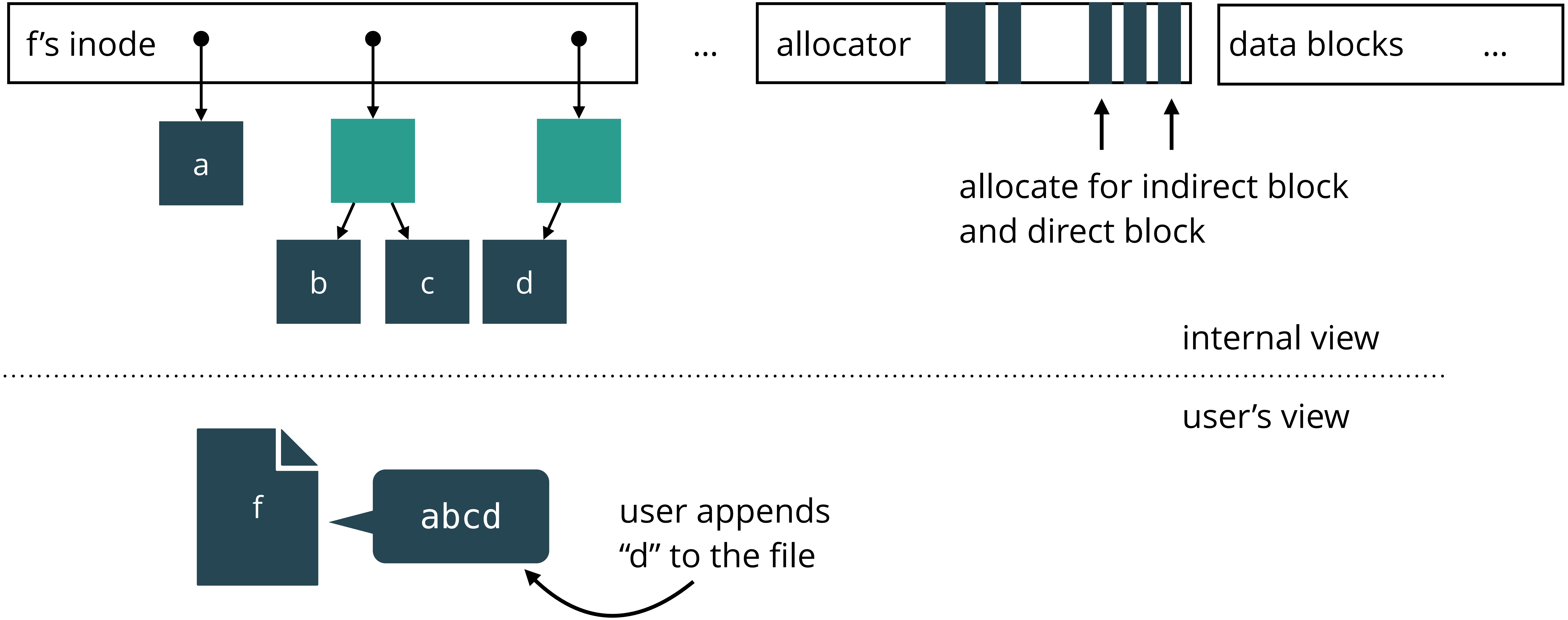
Sequential reasoning helps because each operation needs to do a lot



Sequential reasoning helps because each operation needs to do a lot



Sequential reasoning helps because each operation needs to do a lot



allocate for indirect block
and direct block

internal view

user's view

user appends
"d" to the file

Careful specification of the transaction system enables this division of proof

code

```
tx := Begin()
v := tx.Read(3)
tx.Write(7, v)
tx.Commit()
```

Intuitively, think of this block of code as being atomic

Careful specification of the transaction system enables this division of proof

code

```
tx := Begin()
v := tx.Read(3)
tx.Write(7, v)
tx.Commit()
```

Intuitively, think of this block of code as being atomic

spec

```
atomically {
  v ← Read(3);
  Write(7, v);
}
```

Specification formalizes this by relating code programs to simpler spec programs

Design and implementation of DaisyNFS

Verifying a high-performance transaction system

Evaluating DaisyNFS

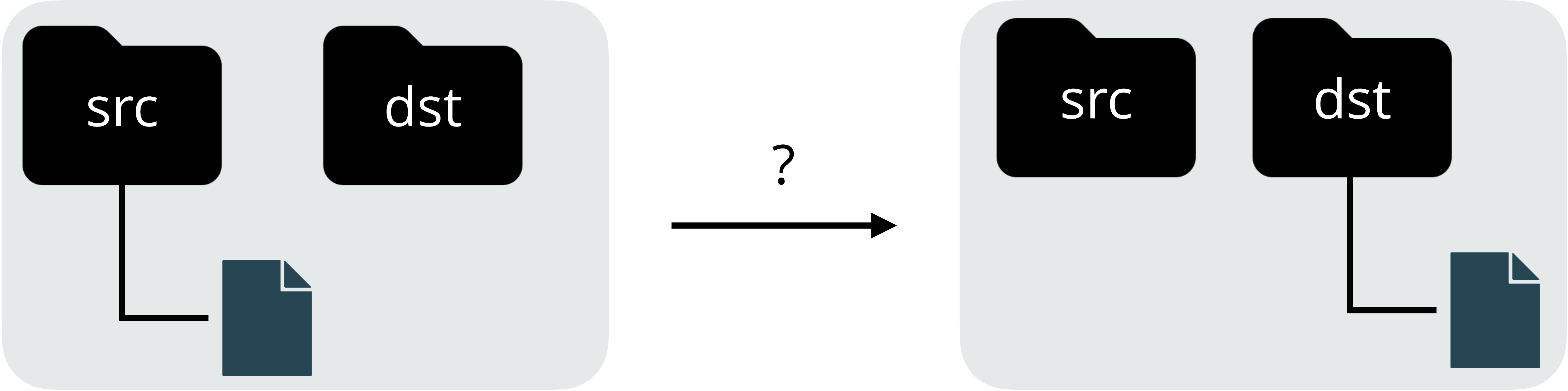
DaisyNFS

GoTxn

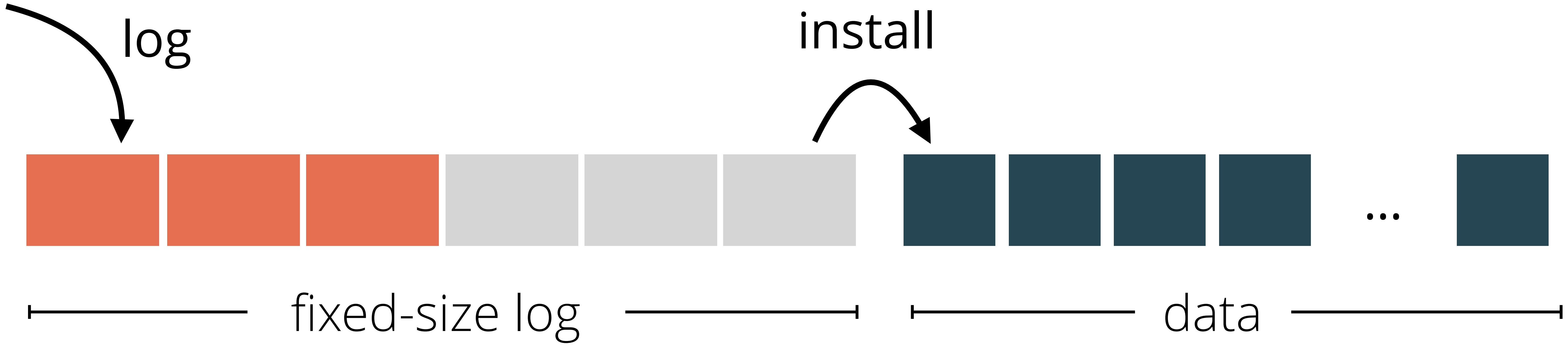
DaisyNFS

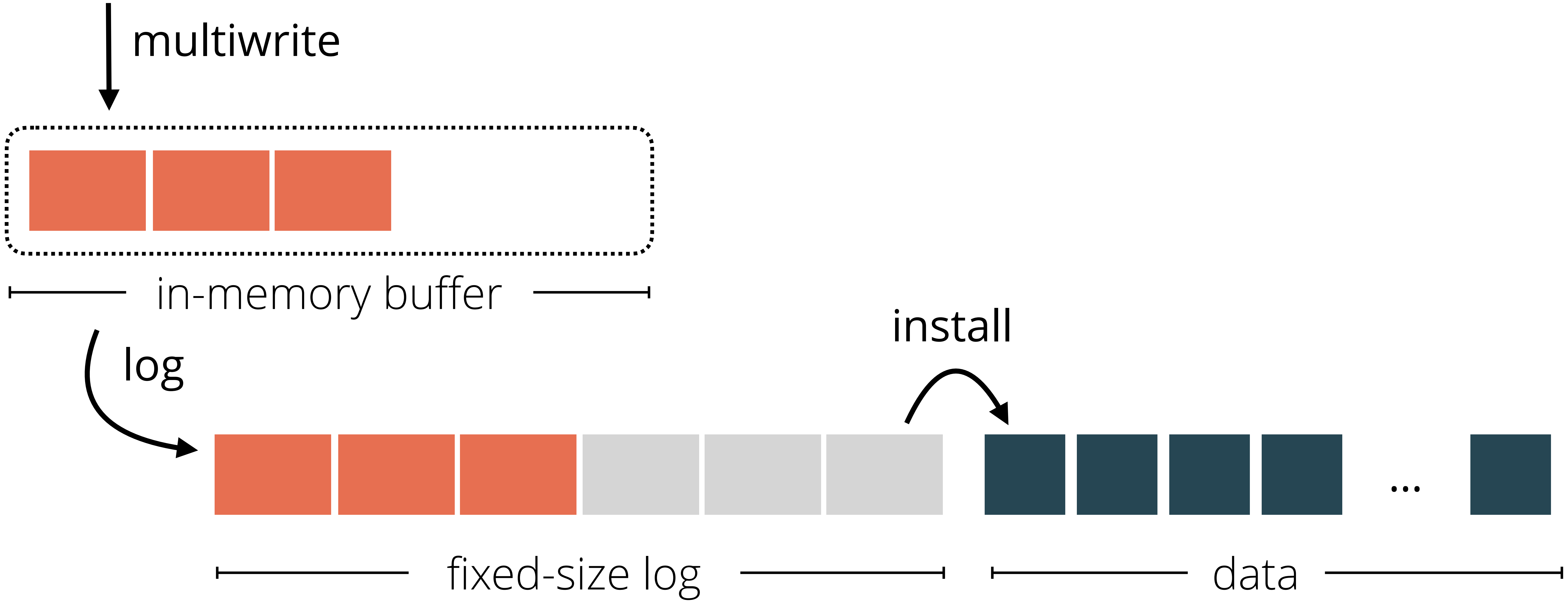
GoTxn

Recall: RENAME needs to update two things atomically

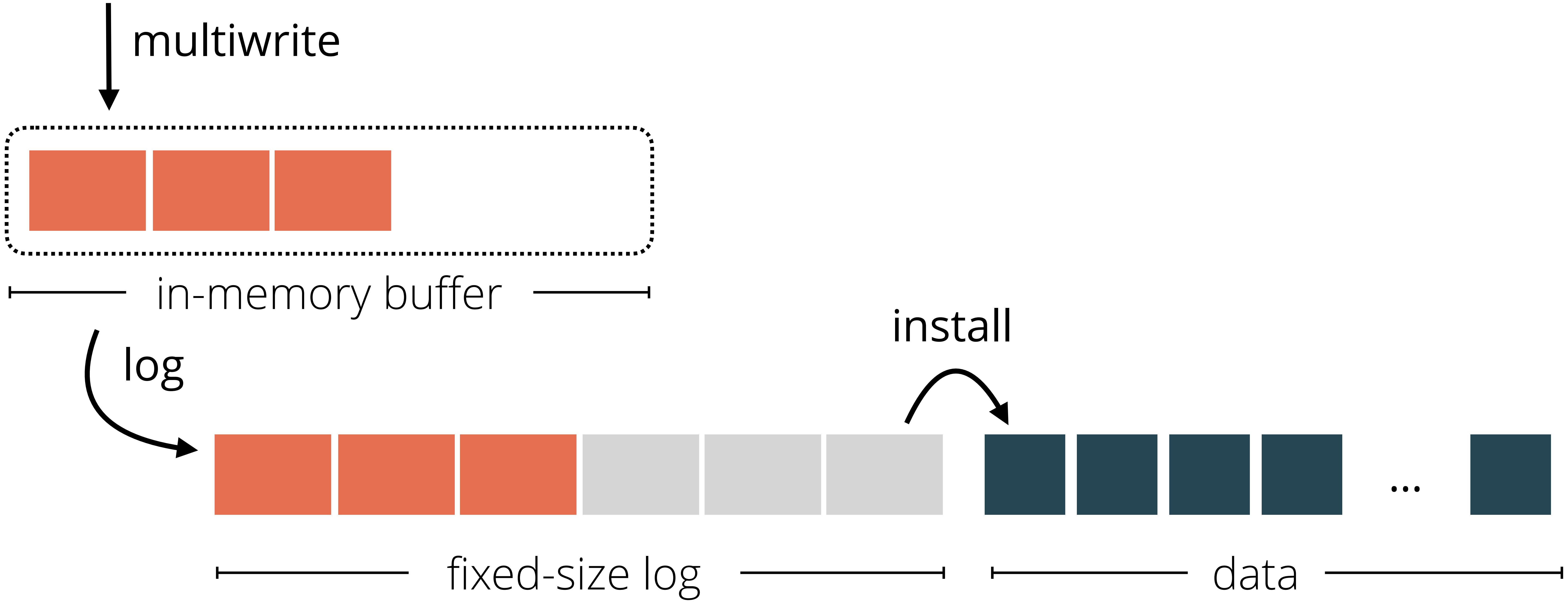


Write-ahead logging is the core atomicity primitive



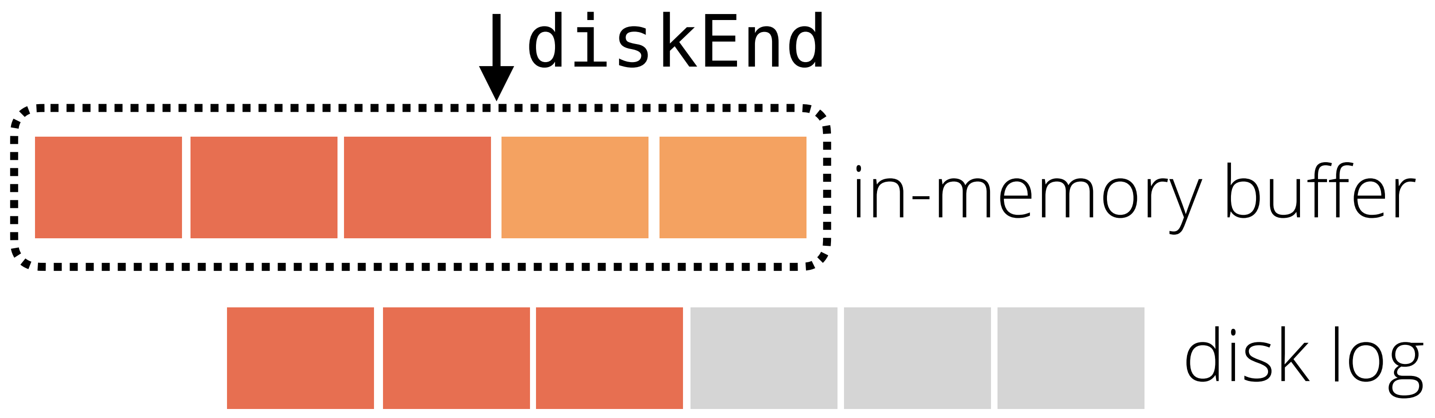

multiwrite
in-memory buffer
log
install
...
fixed-size log
data

Writes, logging, and installation are all concurrent



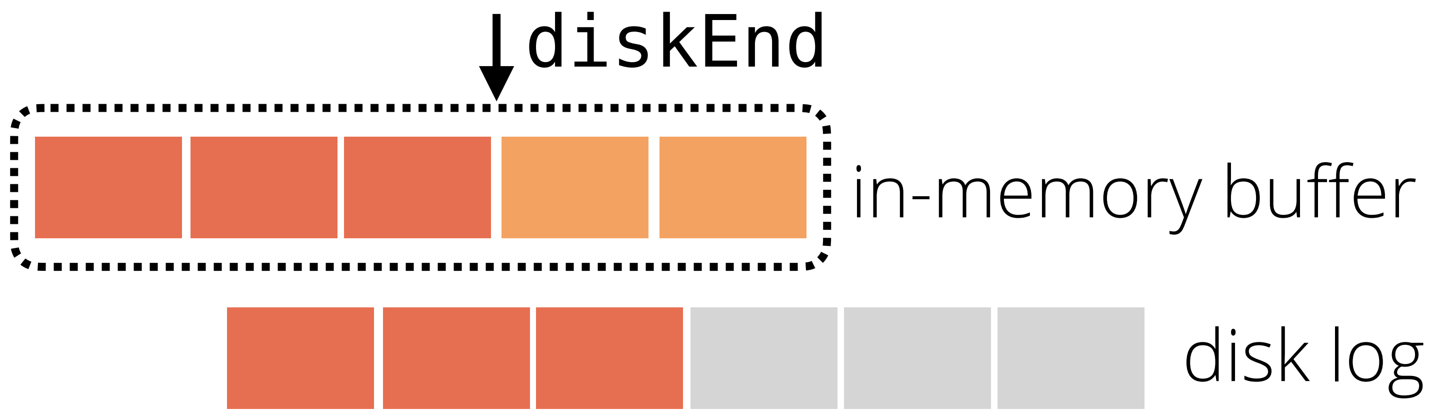
Logging happens lock-free in the background

```
var diskEnd uint64
for {











}
```



Logging happens lock-free in the background

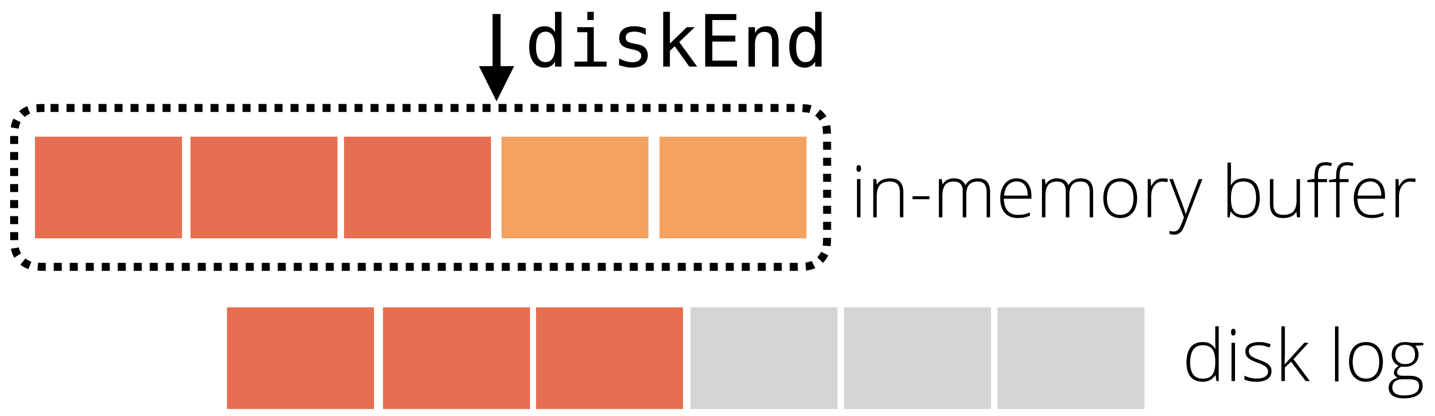
```
var diskEnd uint64
for {
    l.memLock.Lock()
    newBufs := l.memLog.takeFrom(diskEnd)
    l.memLock.Unlock()



}
```



grab `newBufs` (orange writes to be written)

Logging happens lock-free in the background

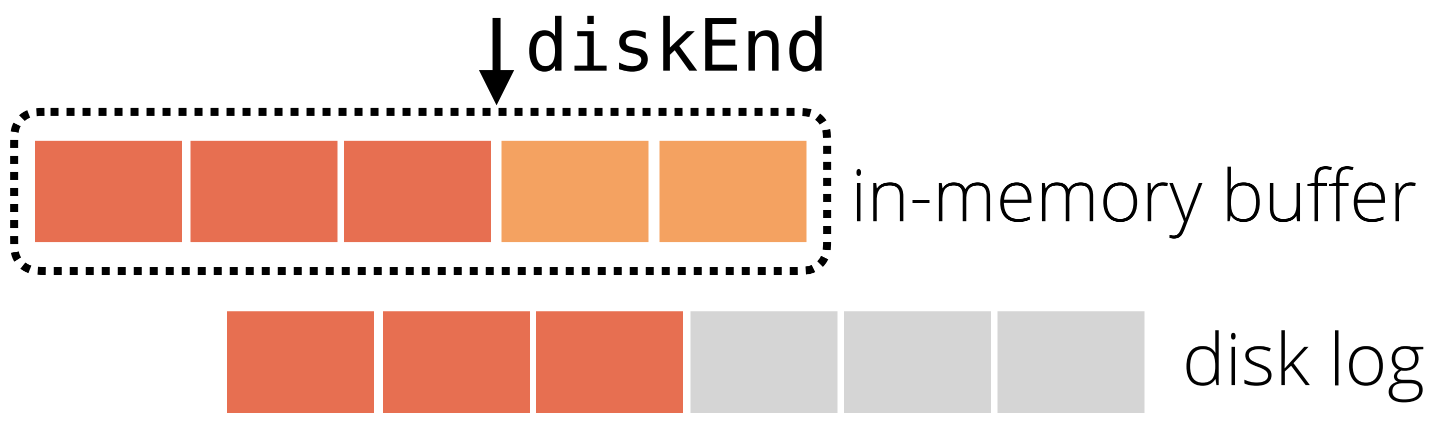
```
var diskEnd uint64
for {
    l.memLock.Lock()
    newBufs := l.memLog.takeFrom(diskEnd)
    l.memLock.Unlock()

    circ.Append(diskEnd, newBufs)




}
```



grab `newBufs` (orange writes to be written)

append `newBufs` to log

Logging happens lock-free in the background



```
var diskEnd uint64
for {
     l.memLock.Lock()
     newBufs := l.memLog.takeFrom(diskEnd)
     l.memLock.Unlock()

     circ.Append(diskEnd, newBufs)

     l.memLock.Lock()
     diskEnd += len(newBufs)
     l.memLock.Unlock()
}
```

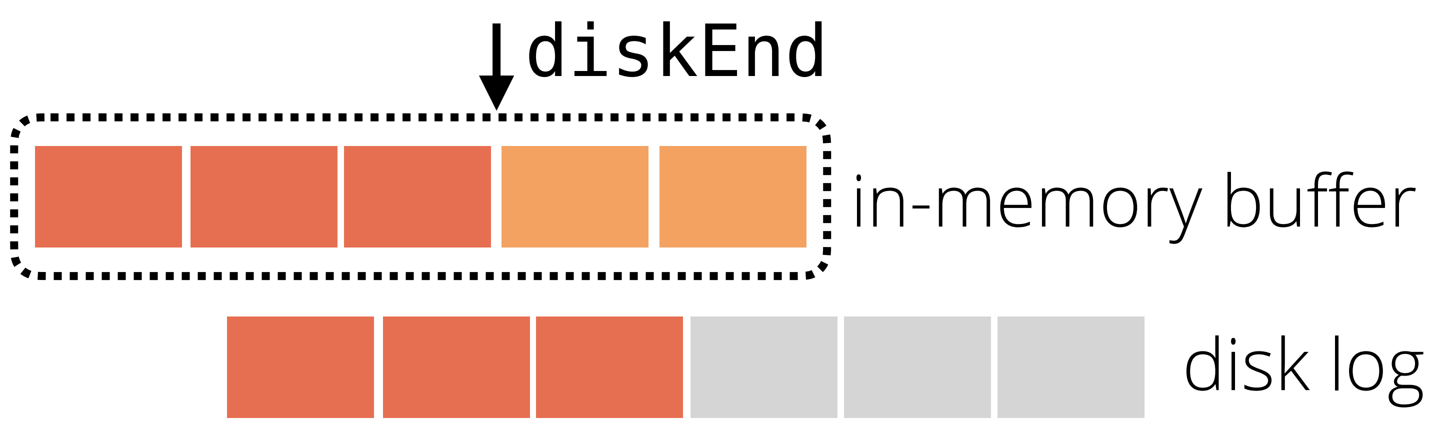
grab `newBufs` (orange writes to be written)

append `newBufs` to log

record that this batch is durable

Logging happens lock-free in the background

```
var diskEnd uint64
for {
    l.memLock.Lock()
    newBufs := l.memLog.takeFrom(diskEnd)
    l.memLock.Unlock()

    circ.Append(diskEnd, newBufs)

    l.memLock.Lock()
    diskEnd += len(newBufs)
    l.memLock.Unlock()

    // wait for a bit
}
```



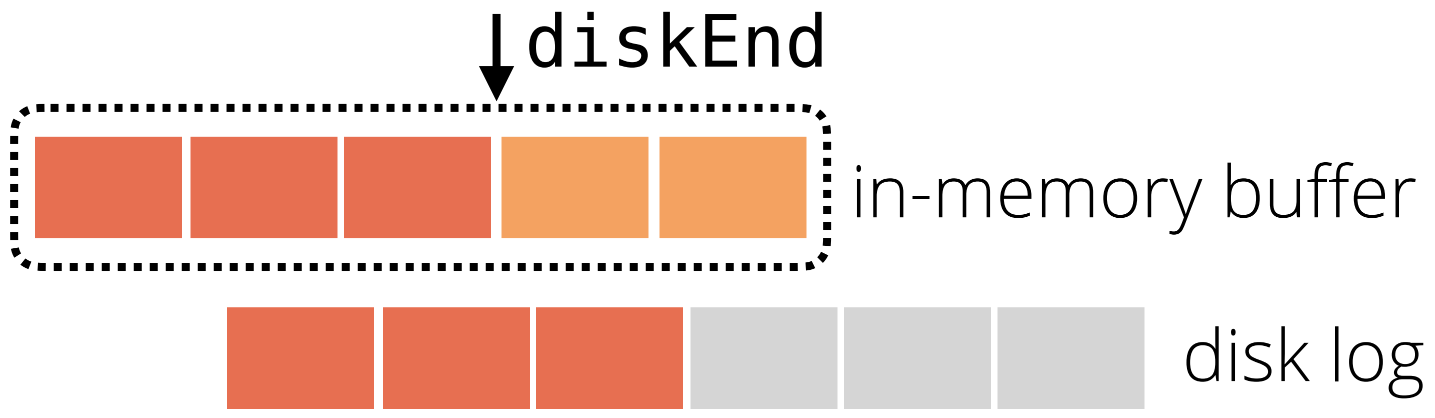
grab `newBufs` (orange writes to be written)

append `newBufs` to log

record that this batch is durable

Logging happens lock-free in the background

```
var diskEnd uint64
for {
    l.memLock.Lock()
    newBufs := l.memLog.takeFrom(diskEnd)
    l.memLock.Unlock()

    circ.Append(diskEnd, newBufs)

    l.memLock.Lock()
    diskEnd += len(newBufs)
    l.memLock.Unlock()

    // wait for a bit
}
```



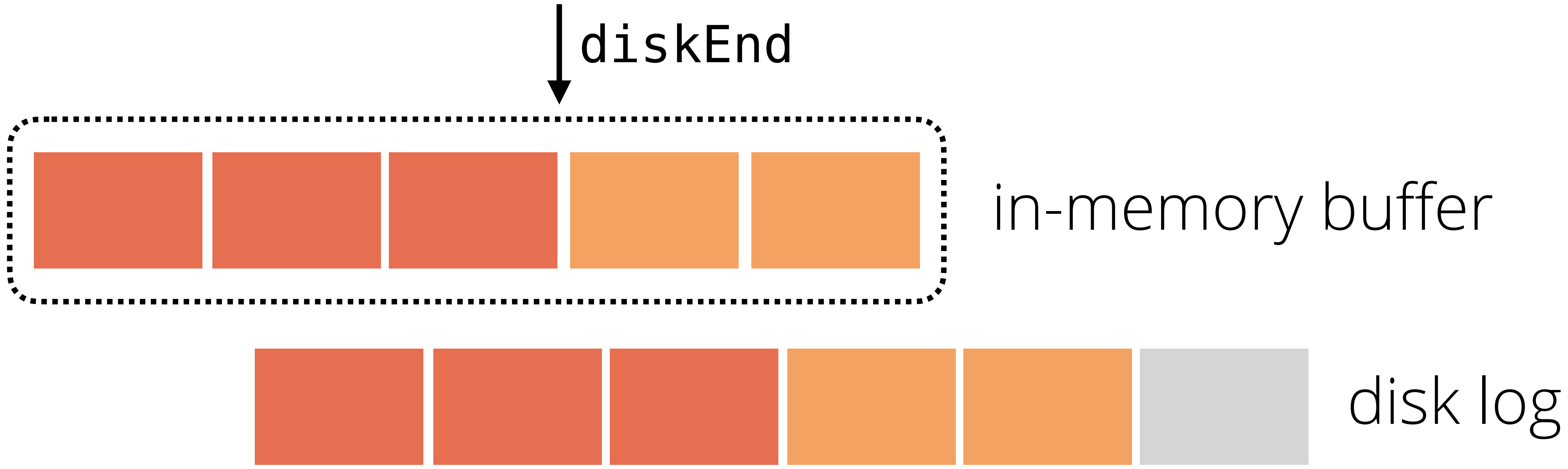
grab `newBufs` (orange writes to be written)

append `newBufs` to log

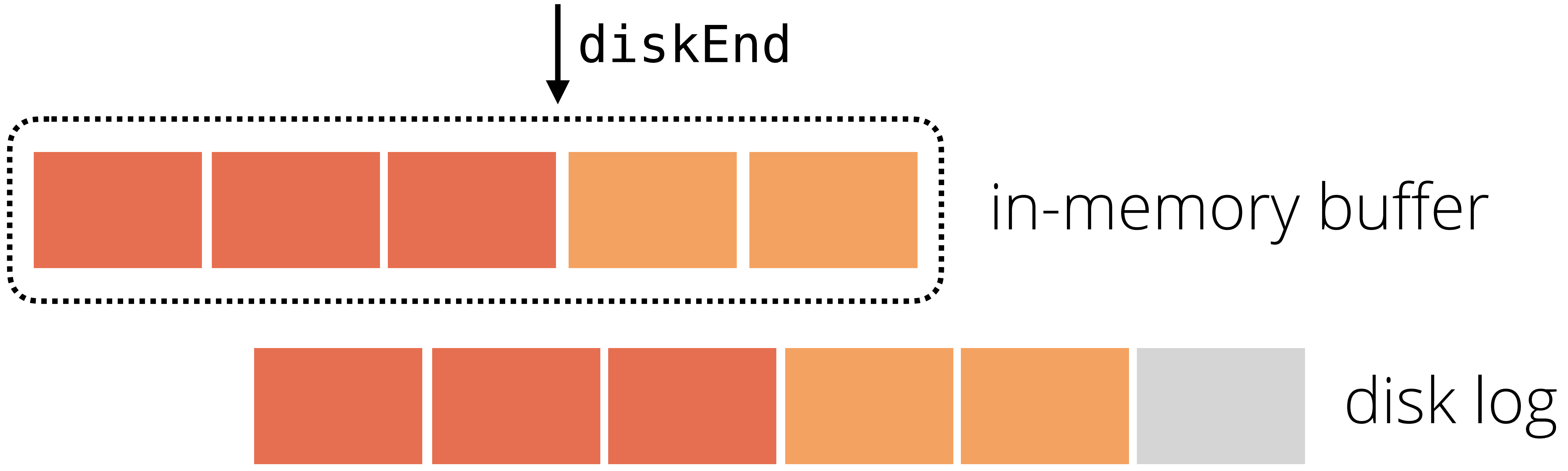
record that the batch is durable

after this operation, `diskEnd` doesn't reflect what is durable

Proof uses general concurrency techniques to reason about lock-free region

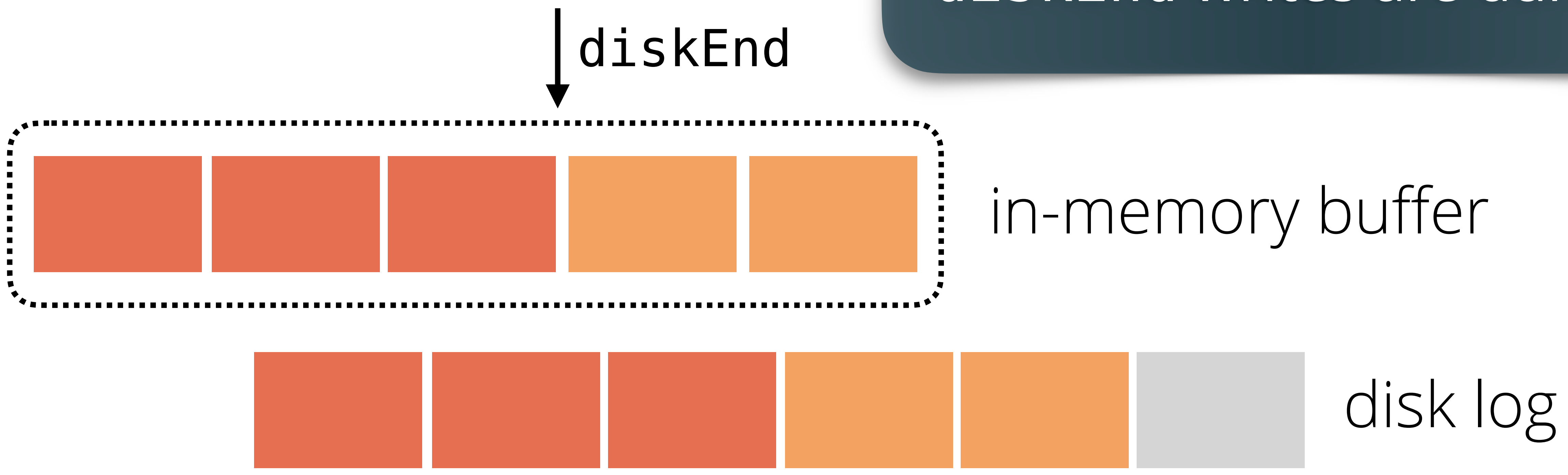


Proof uses general concurrency techniques to reason about lock-free region



logger knows exactly
`diskEnd`+2 writes are durable

Proof uses general concurrency techniques to reason about lock-free region



other threads know *at least* `diskEnd` writes are durable

logger knows exactly `diskEnd+2` writes are durable

Many other challenges in GoTxn proof

What’s the specification for each internal layer?

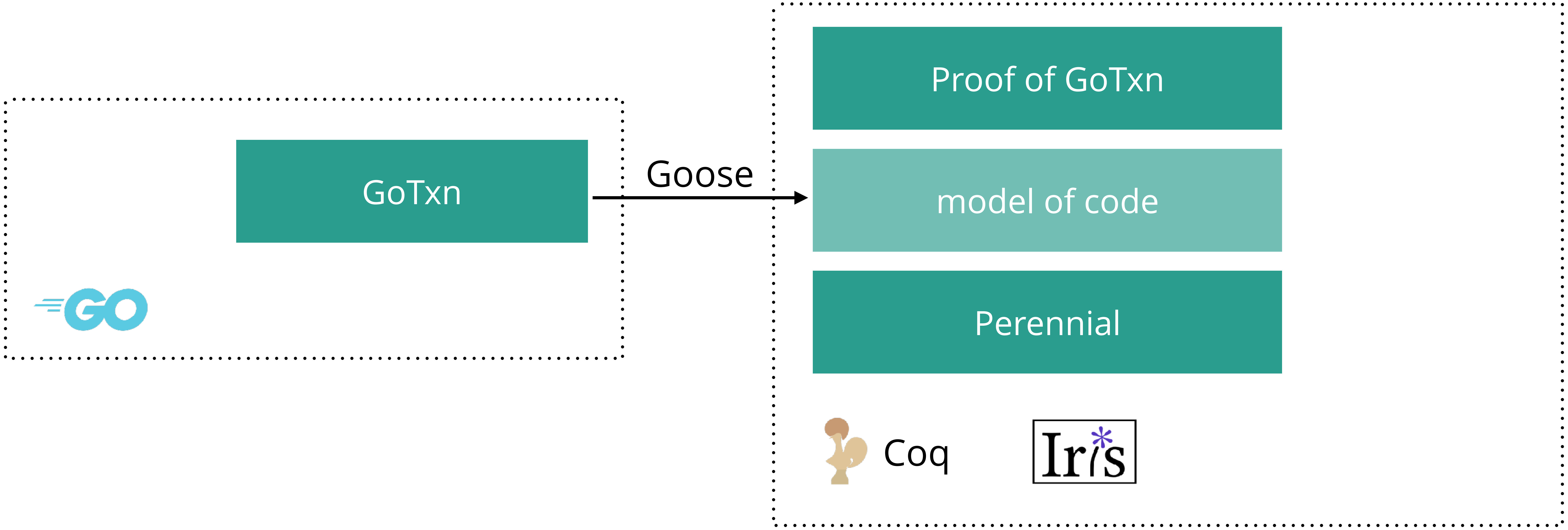
Invariants for lock-free installation, concurrency within a block, two-phase locking

Design and implementation of DaisyNFS

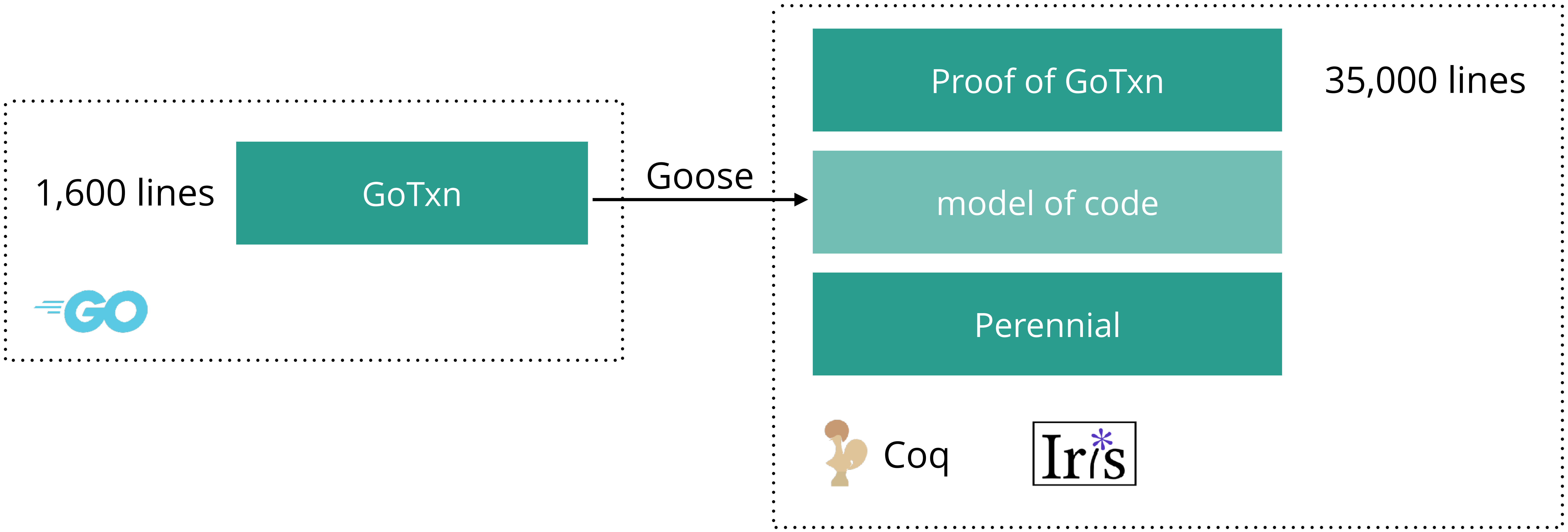
Verifying a high-performance transaction system

Evaluating DaisyNFS

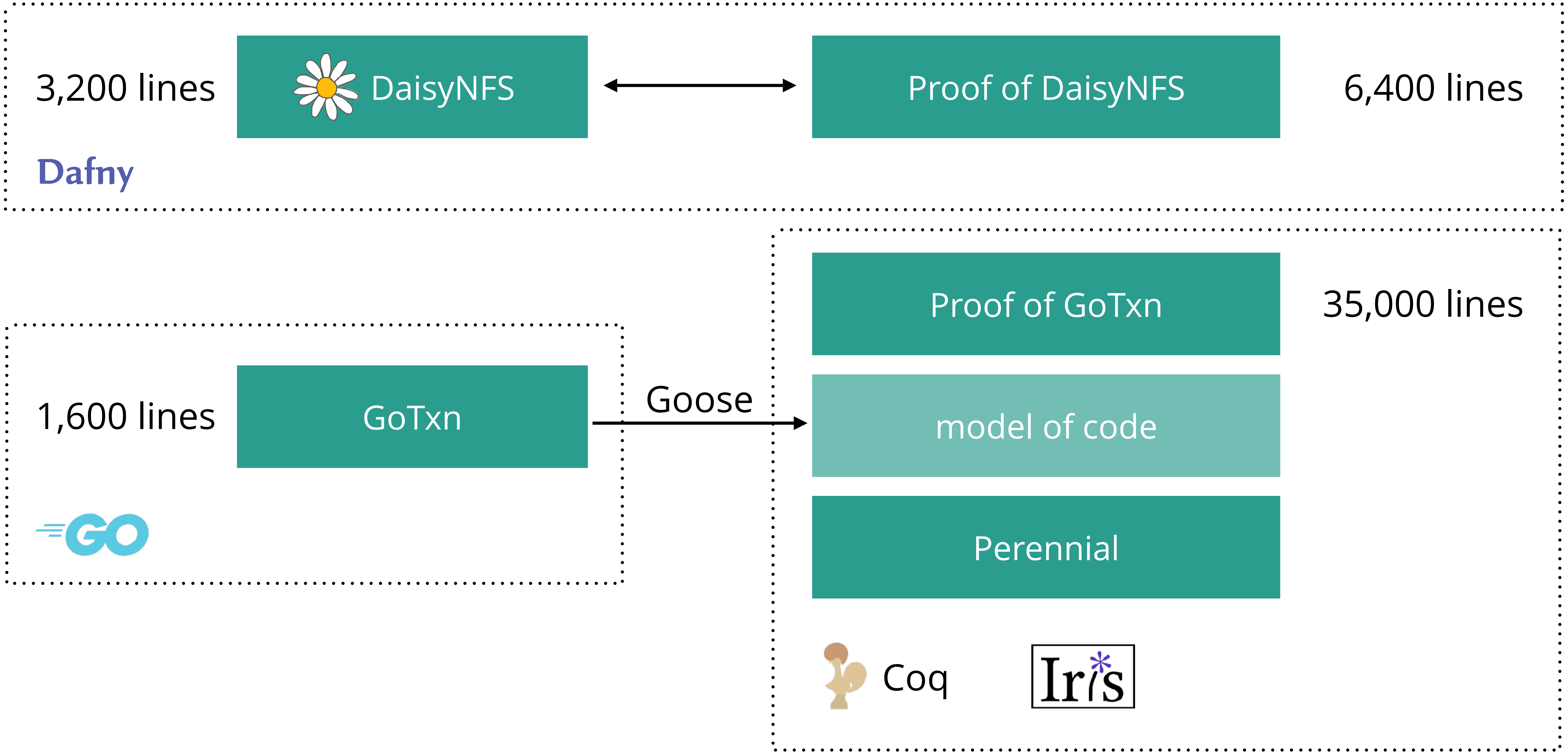
Most of the proof is for GoTxn



Most of the proof is for GoTxn



Most of the proof is for GoTxn



3,200 lines

DaisyNFS

Proof of DaisyNFS

6,400 lines

Dafny

Proof of GoTxn

35,000 lines

1,600 lines

GoTxn

Goose

model of code

Perennial

GO

Coq

Iris

Limitations

Limitations

limit performance

Only synchronous Commit

Must use transactions from Dafny

Limitations

limit performance

- Only synchronous Commit
- Must use transactions from Dafny

limits to proof

- Could still have deadlock
- Linking theorem proven on paper

Proof assumptions

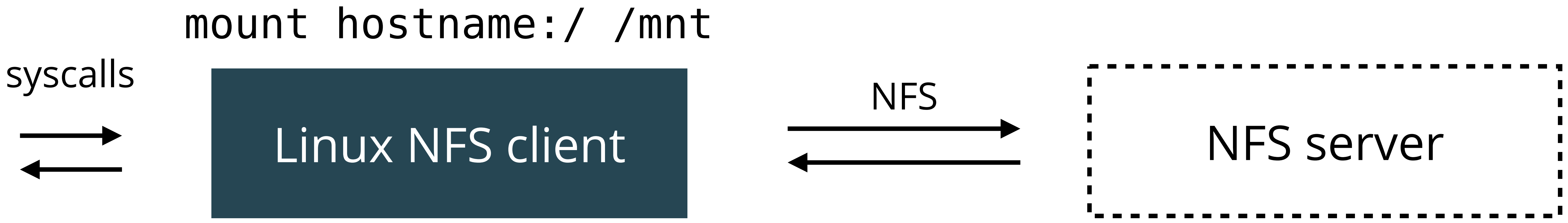
We assume that:

Goose accurately models Go

Disk has atomic 4KB reads and writes

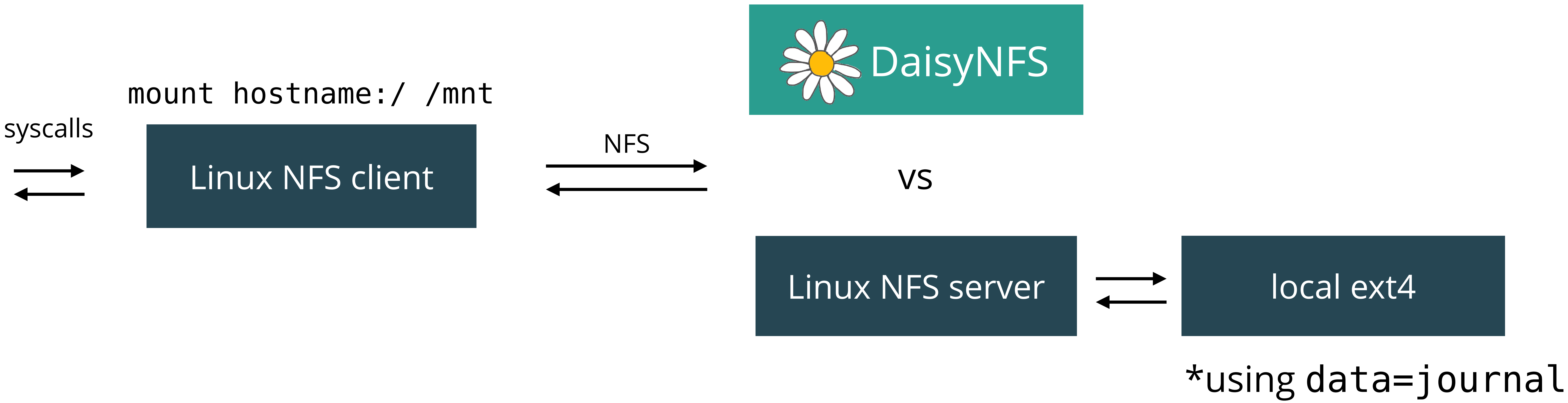
NFS specification is written correctly

Evaluate performance using an NFS client



`mount hostname:/ /mnt`

Compare against Linux NFS



`mount hostname:/ /mnt`

syscalls

Linux NFS client

NFS

DaisyNFS

vs

Linux NFS server

local ext4

*using `data=journal`

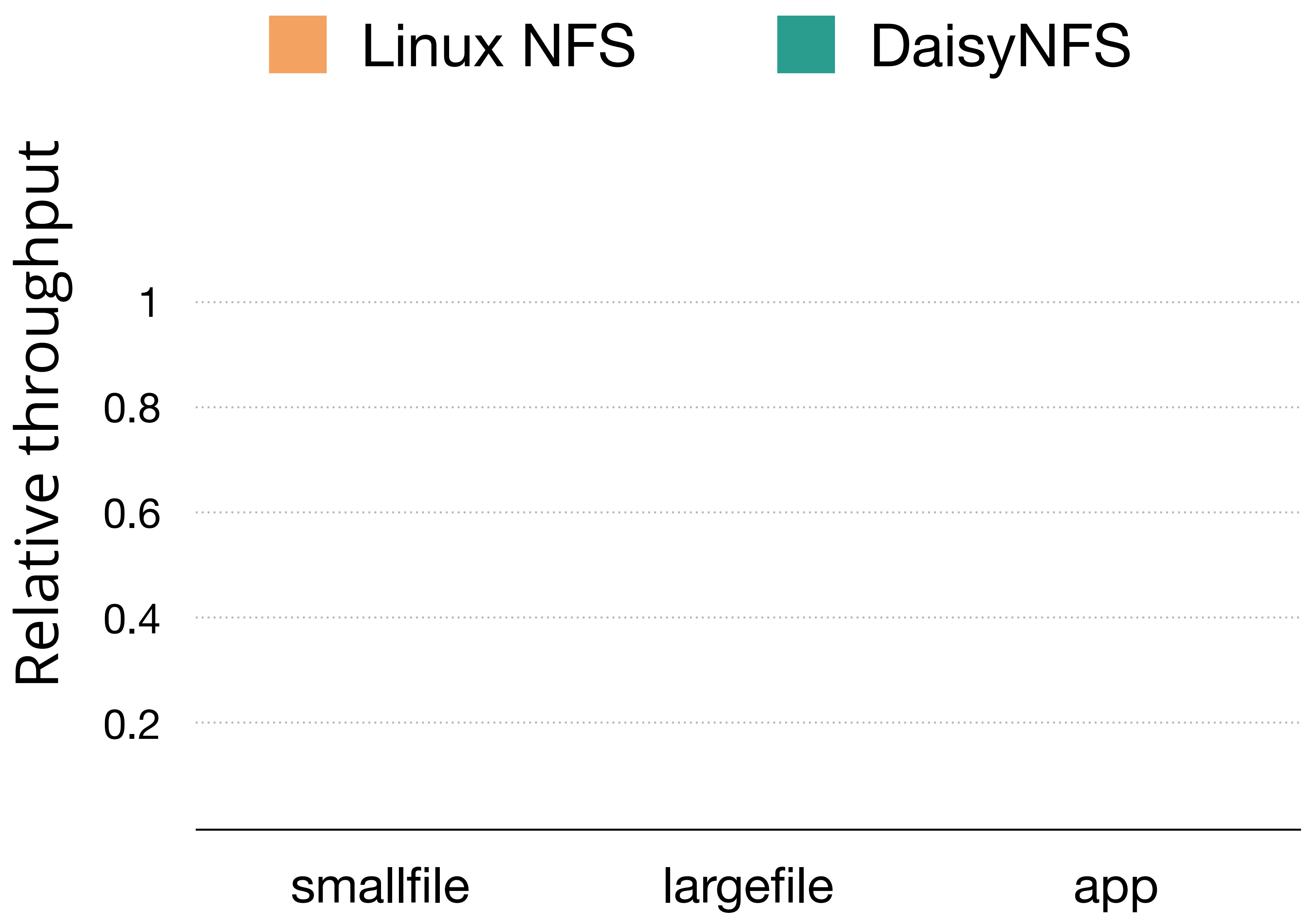
Performance evaluation setup

Hardware: i3.metal instance

36 cores at 2.3GHz, NVMe SSD

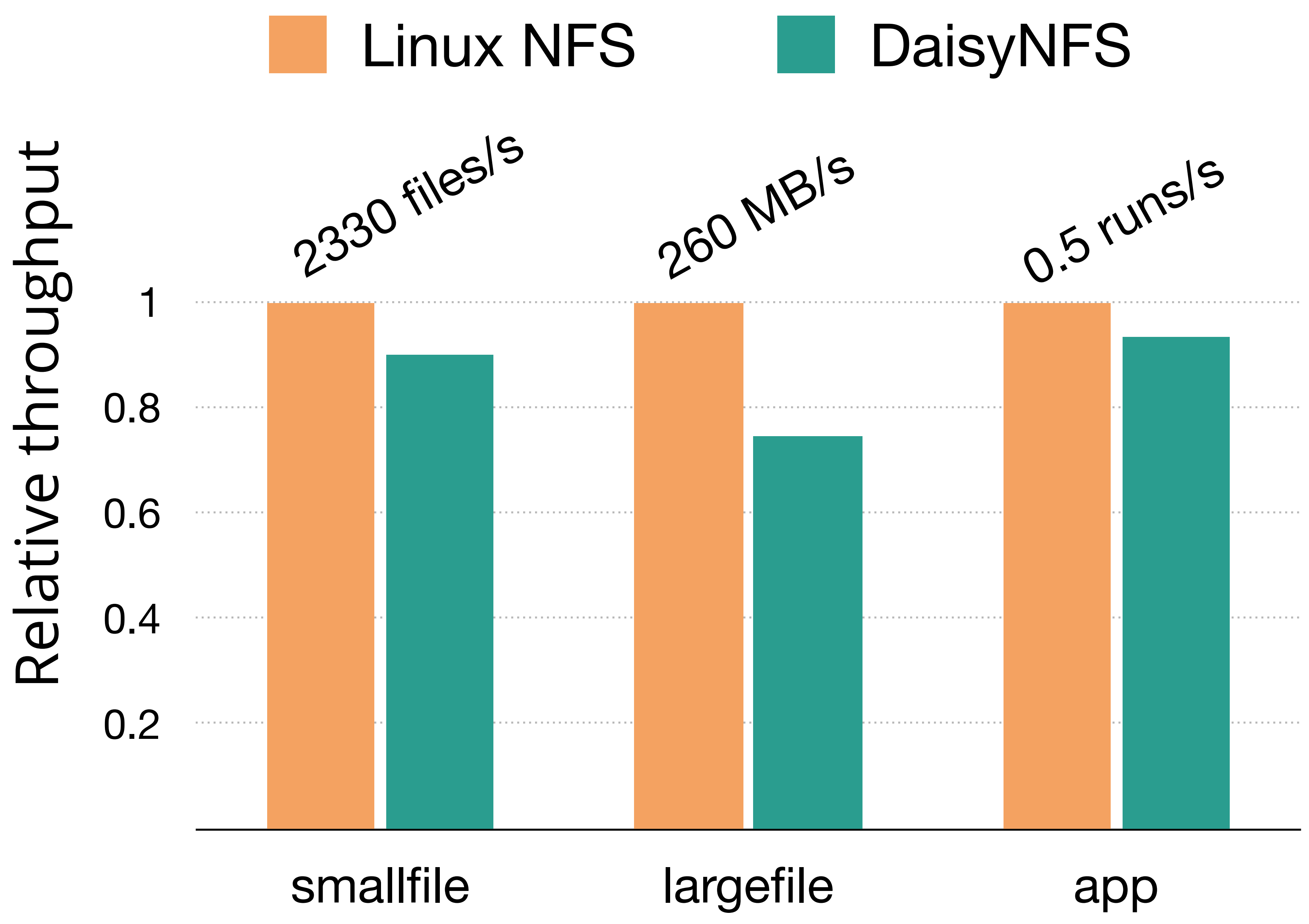
Benchmarks:

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

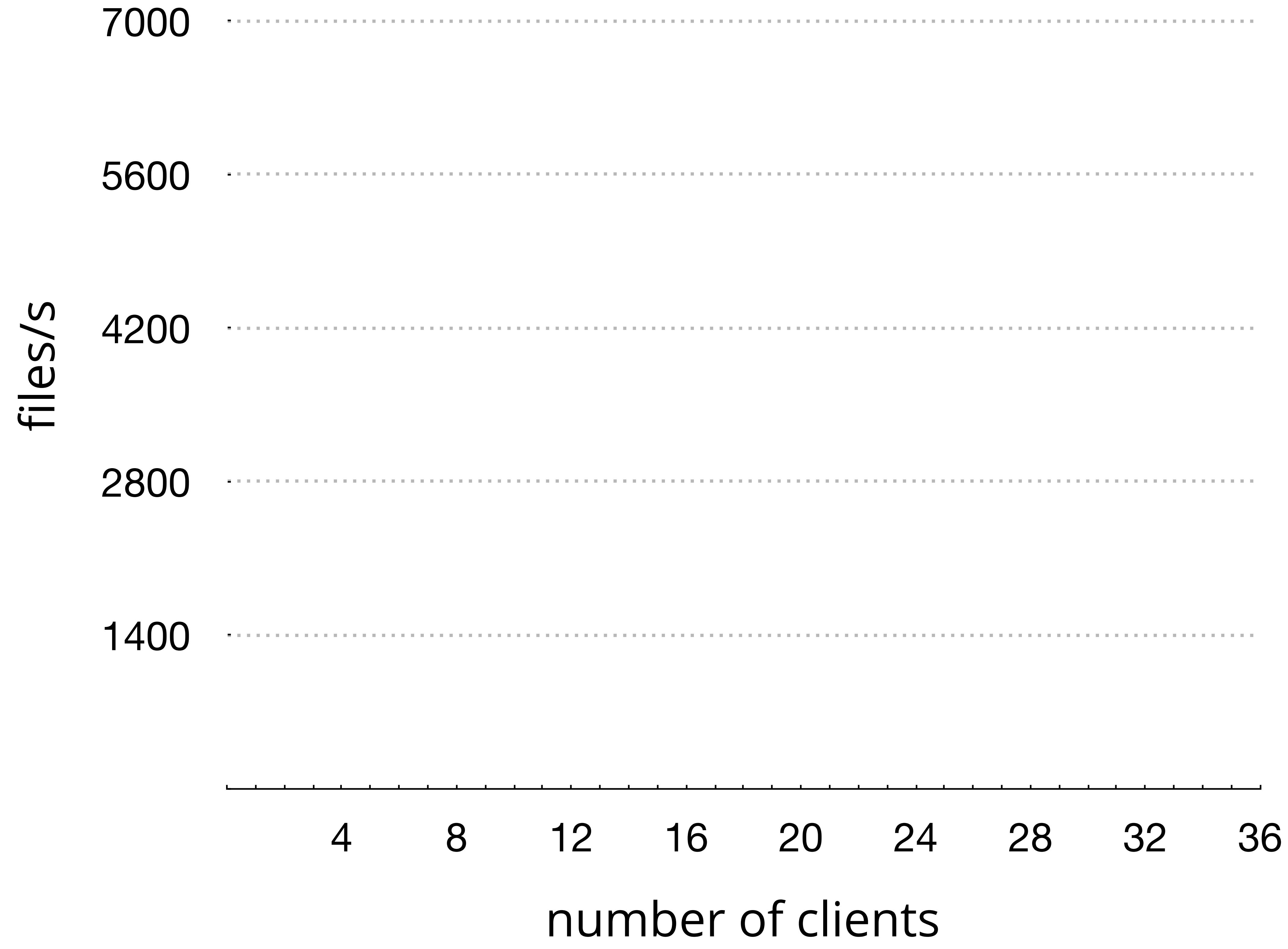


Compare DaisyNFS throughput to Linux, running on an in-memory disk

DaisyNFS gets comparable performance even with a single client

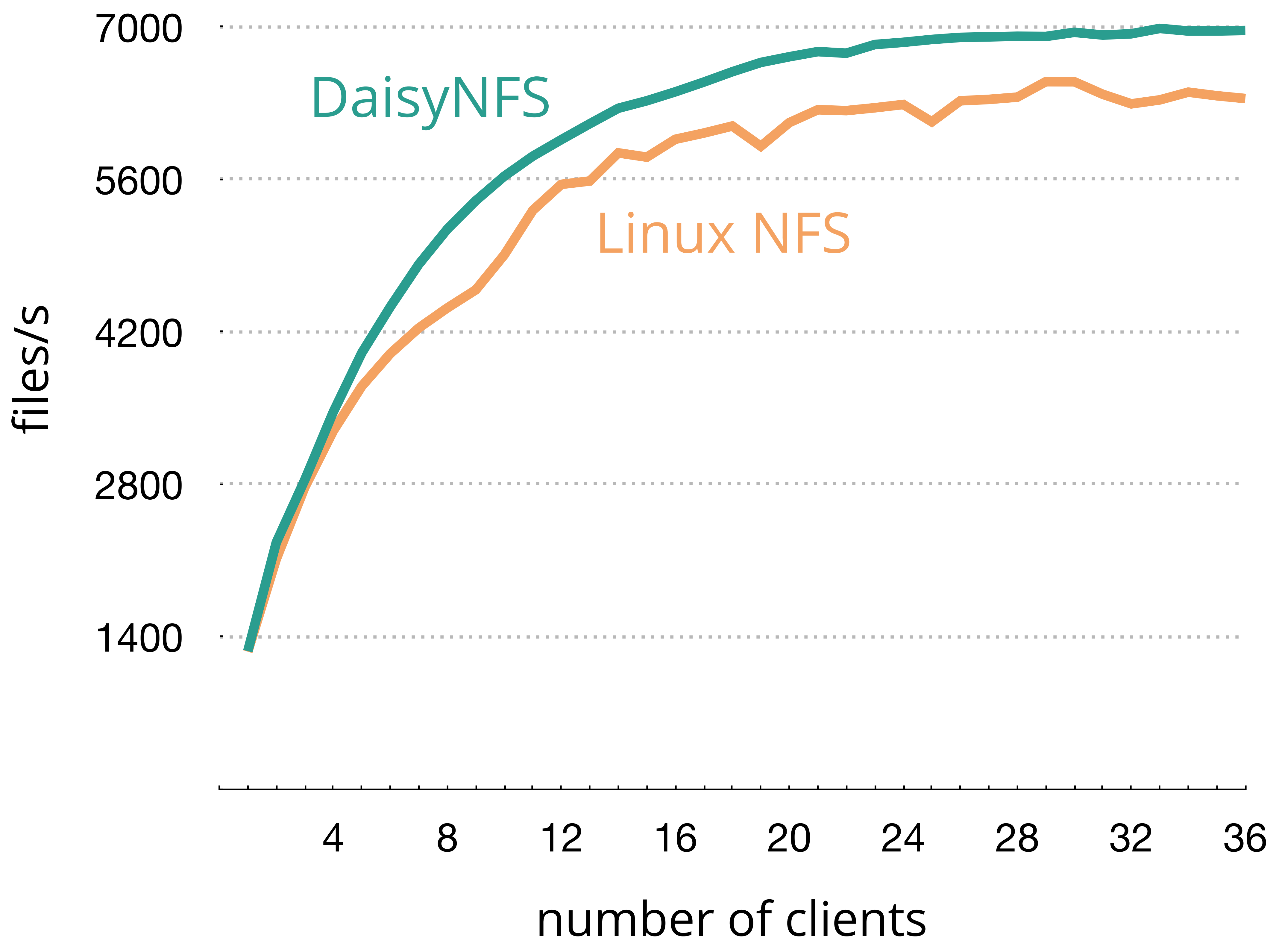


Compare DaisyNFS throughput to Linux, running on an in-memory disk



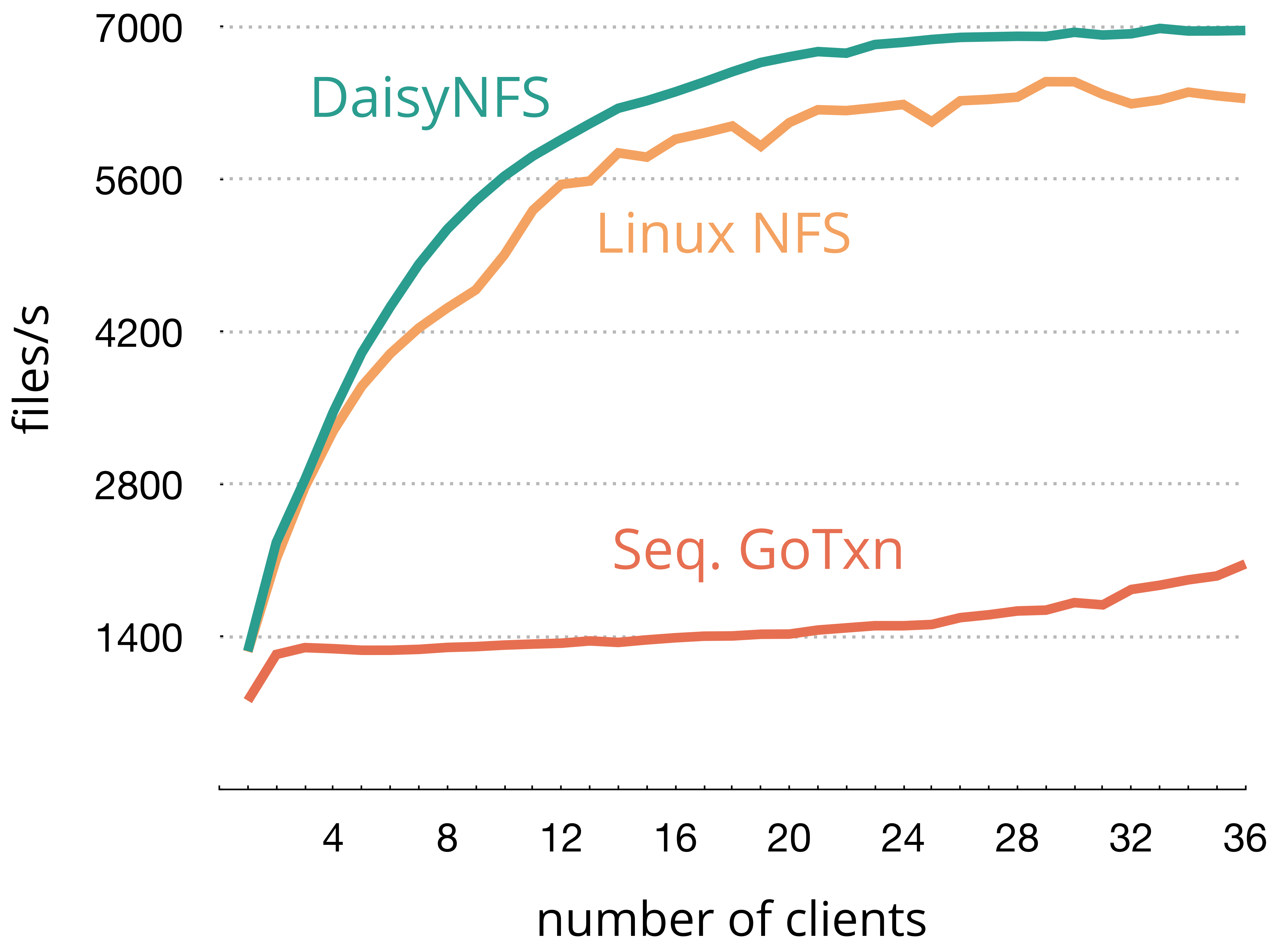
Run smallfile with many clients on an NVMe SSD

DaisyNFS can take advantage of multiple clients



Run smallfile with many clients on an NVMe SSD

Concurrency in the transaction system matters



Seq. GoTxn is DaisyNFS but with locks around tricky concurrent parts of WAL

Related work

Related work

crash safety and concurrency:
Flashix concurrent file system, ShardStore

crash safety:
FSCQ, Yggdrasil, VeriBetrFS

concurrency:
Concurrent GC, CertiKOS, AtomFS

Other related work

Goose: VST and CH20 for reasoning about C

Perennial: builds on top of Iris

GoTxn: verified transaction algorithms but not systems

DaisyNFS: builds upon DFSCQ and Yggdrasil

Each system is general-purpose

Perennial and Goose can be applied to other storage systems, languages, and hardware

GoTxn can be used to build other storage systems, in Dafny or Perennial

Summary

New foundations (**Perennial** and **Goose**) make verification of concurrent storage systems possible

GoTxn isolates the difficult reasoning so proofs on top use sequential reasoning

Verified **DaisyNFS**, a concurrent, crash-safe file system with performance comparable to Linux

Acknowledgments

Frans Kaashoek

Nickolai Zeldovich

Joe Tassarotti

PDOS

Neha, Austin, Srivatsa, Julian, Jelle, Haogang, Shoumik, Cody, Frank, Amy, Malte, Joe, David, Jon

Atalay, Anish, Derek, Jonathan, Akshay, Lily, Josh, Inho, Zain, Ariel, Kevin, Alex, Upamanyu, Ralf, Yun-Sheng

Mentees

PDOS

Daniel Ziegler
Alex Konradi
Lef Ionnadis
Sydney Gibson
Sharon Lin

MIT PL

Sara Achour, Clément Pit-Claudel, Ben Sherman,
Sam Gruetter, Thomas Bourgeat, and many others

CSC board games

especially Leilani, Nathan, Max, Jon, and Ajay

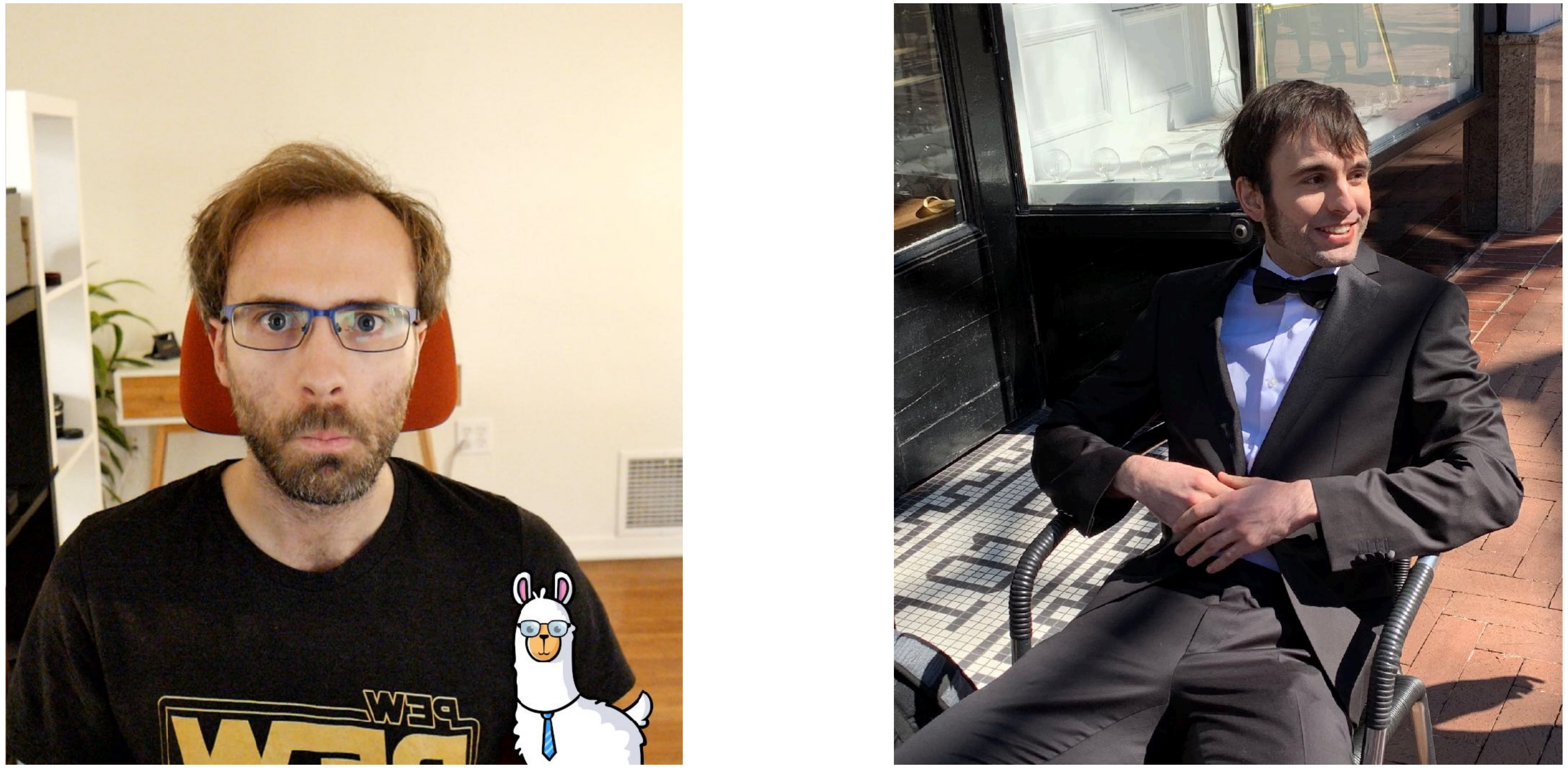
CSC board games

especially Leilani, Nathan, Max, Jon, and Ajay

CSC

MIT
EECS
COMMUNICATION LAB
mitcommlab.mit.edu/eecs/

250 Elm



Many others