Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning

Tej Chajed
MIT CSAIL
(now VMware Research & UW-Madison)

Joseph Tassarotti
Boston College
(now NYU)

Mark Theng
MIT CSAIL

Frans Kaashoek
MIT CSAIL

Nickolai Zeldovich
MIT CSAIL

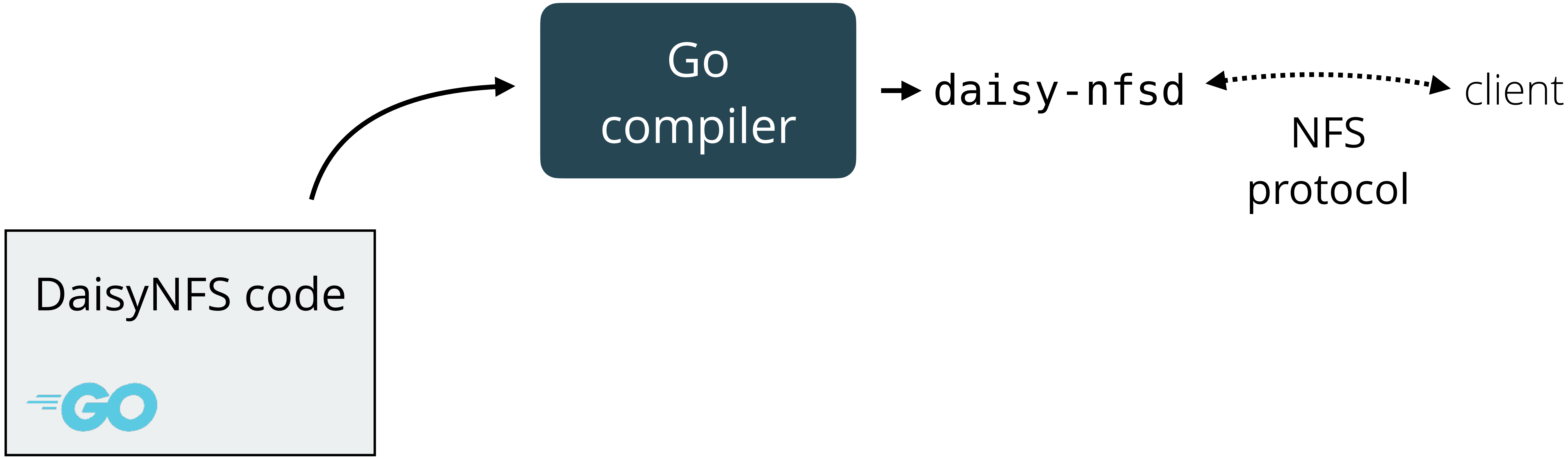
File-system correctness is important but challenging

Applications rely on file system to store data

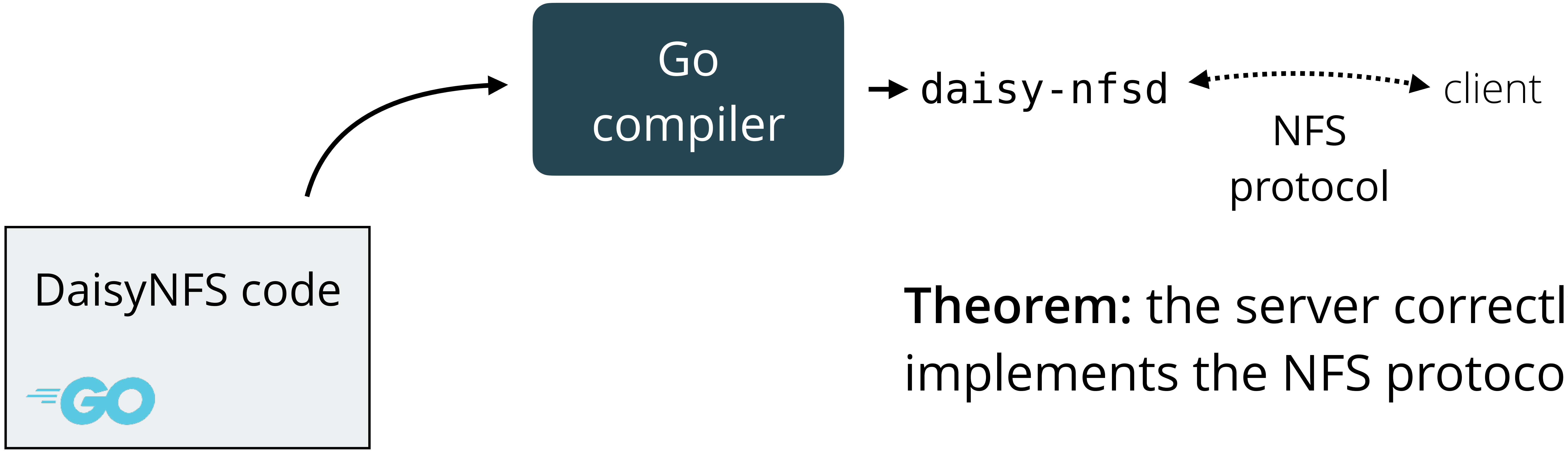
Prone to bugs due to optimizations

Bugs can lead to data loss

Approach: formal verification

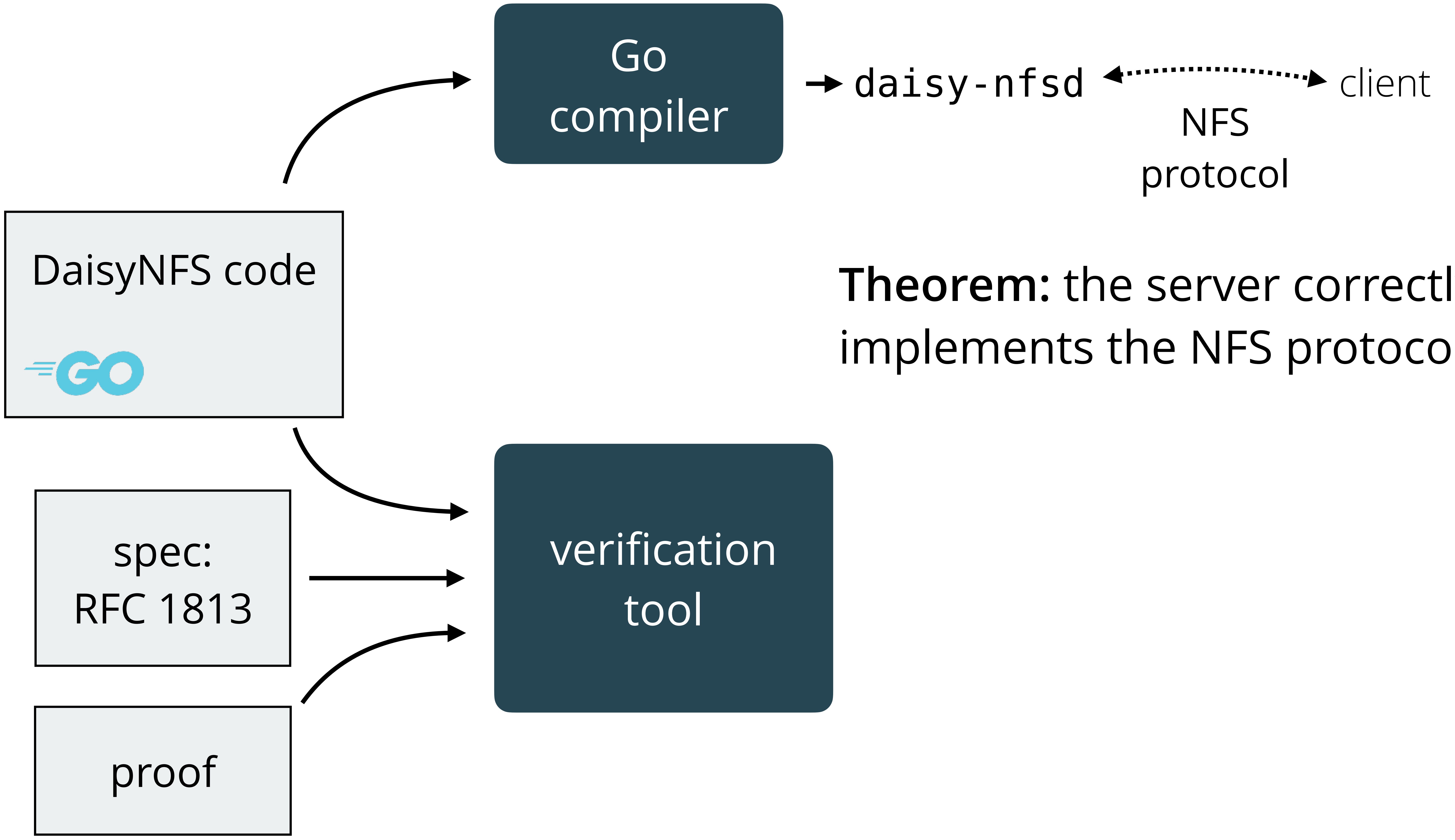


Approach: formal verification



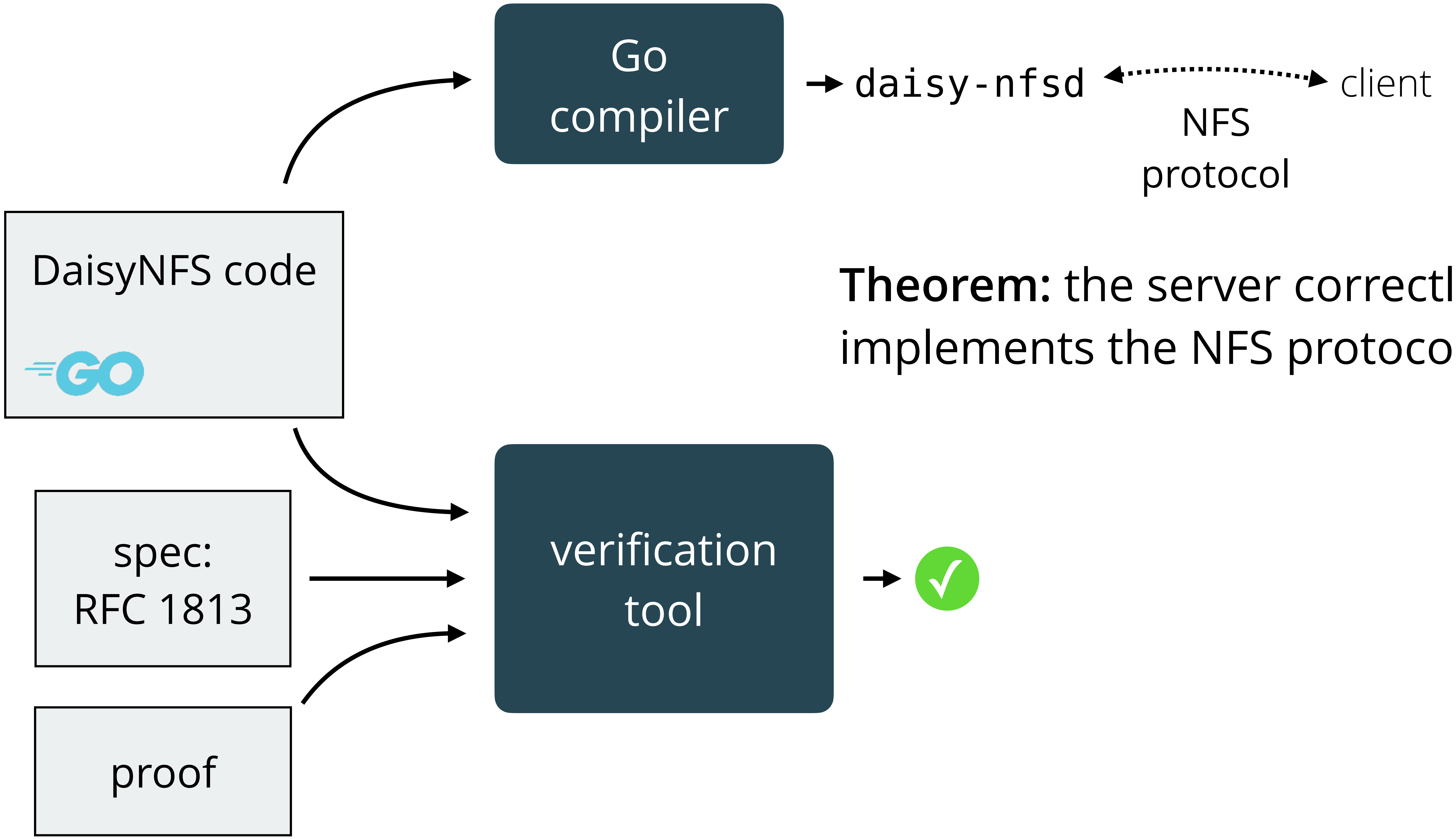
Theorem: the server correctly implements the NFS protocol.

Approach: formal verification



Theorem: the server correctly implements the NFS protocol.

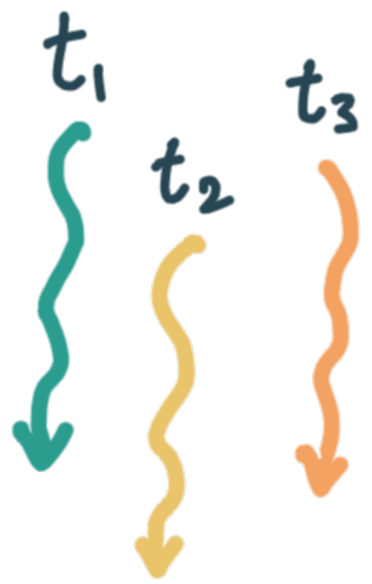
Approach: formal verification



Theorem: the server correctly implements the NFS protocol.

Challenges in verifying a file system

Crashes



Concurrency

Contribution: verification-friendly design

DaisyNFS

File-system code implemented with transactions

GoTxn

Transaction system gives atomicity

Contribution: verification-friendly design

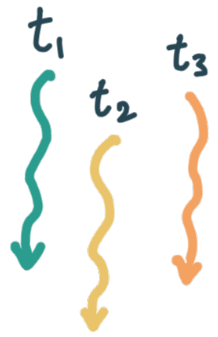
DaisyNFS

File-system code implemented with transactions

GoTxn

Transaction system gives atomicity

Crashes



Concurrency

Contribution: verification-friendly design

DaisyNFS

File-system code implemented with transactions

Sequential reasoning

GoTxn

Transaction system gives atomicity

Crashes

Concurrency

Contribution: verification-friendly design

DaisyNFS

File-system code implemented with transactions

Sequential reasoning

Simulation-transfer theorem turns sequential reasoning into concurrent & crash-safe correctness

GoTxn

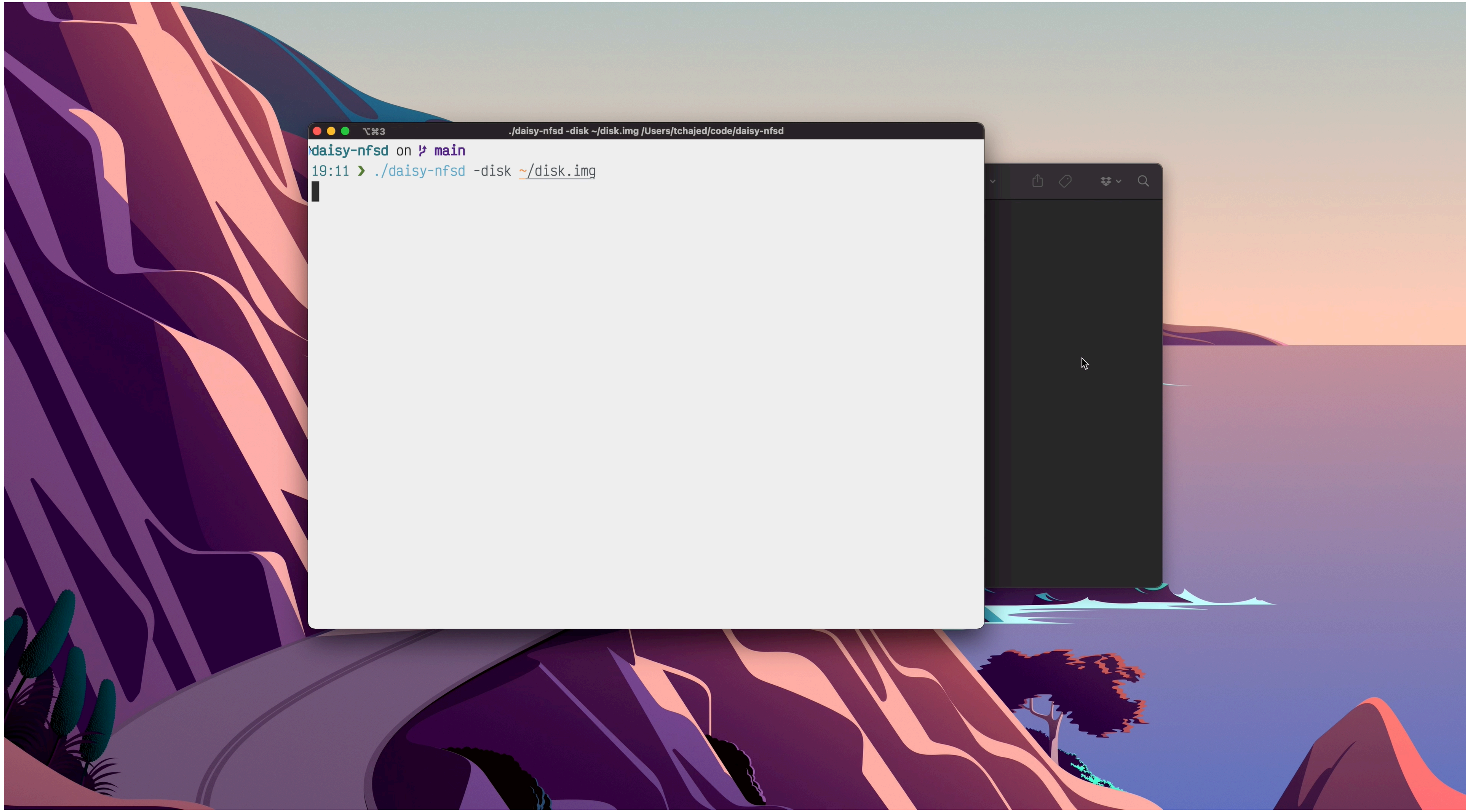
Transaction system gives atomicity

Crashes

t_1 t_2 t_3

Concurrency

DaisyNFS is a real file system



⌥⌘3 ./daisy-nfsd -disk ~/disk.img /Users/tchajed/code/daisy-nfsd

```
daisy-nfsd on main
19:11 ❯ ./daisy-nfsd -disk ~/disk.img
```

System design

```
func handleCommand(req) {
  switch req.cmd {
    case MKDIR: fs.MKDIR(…)
    case LOOKUP: fs.LOOKUP(…)
    …
  }
}
```

Go (unverified)

System design

```
func handleCommand(req) {
  switch req.cmd {
    case MKDIR: fs.MKDIR(…)
    case LOOKUP: fs.LOOKUP(…)
    …
  }
}
```

Go (unverified)

```
method MKDIR(d_ino, name)
method LOOKUP(d_ino, name)
  returns (ino:Ino)
…
```

Dafny (verified)

System design

```
func handleCommand(req) {
  switch req.cmd {
    case MKDIR: fs.MKDIR(…)
    case LOOKUP: fs.LOOKUP(…)
    …
  }
}
```

Go (unverified)

```
method MKDIR(d_ino, name)
method LOOKUP(d_ino, name)
  returns (ino:Ino)
…
```

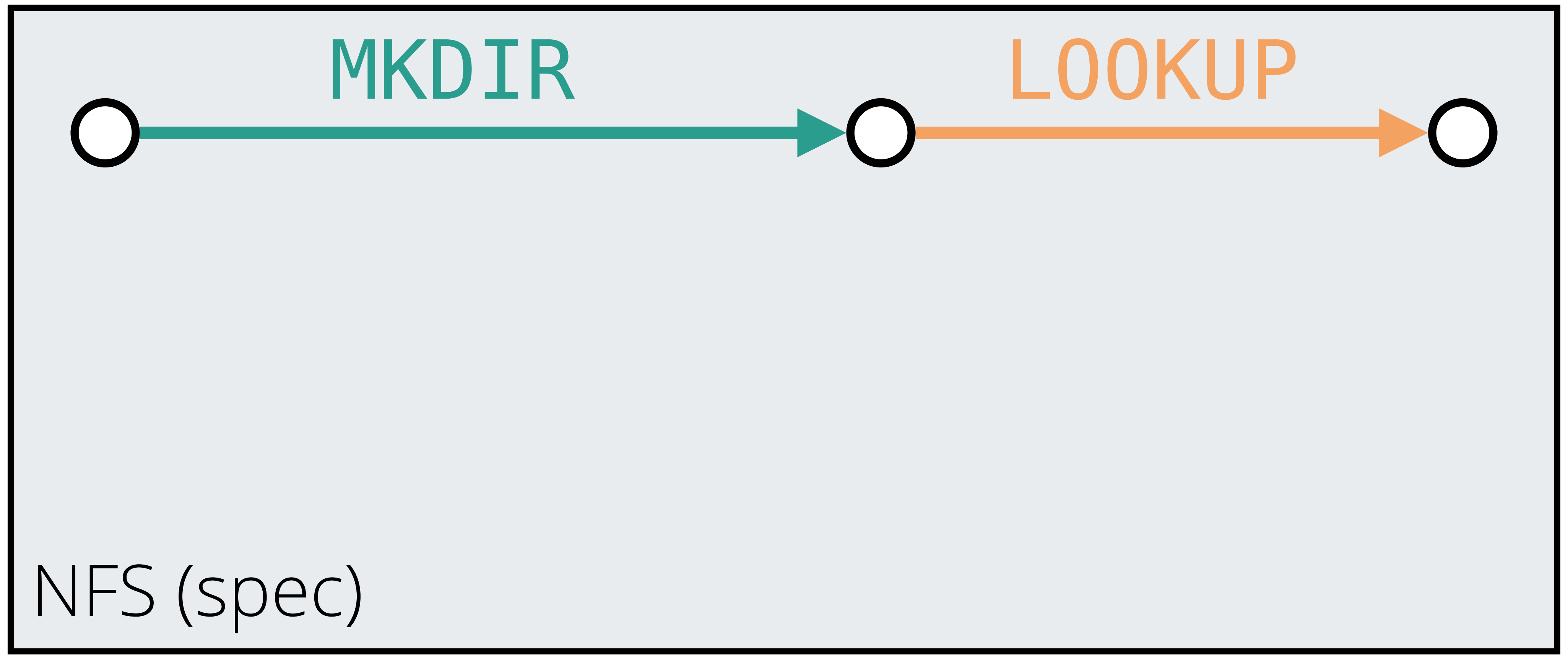
Dafny (verified)

```
tx := Begin()
tx.Read(…)
tx.Write(…)
tx.Commit()
```

Go (verified)

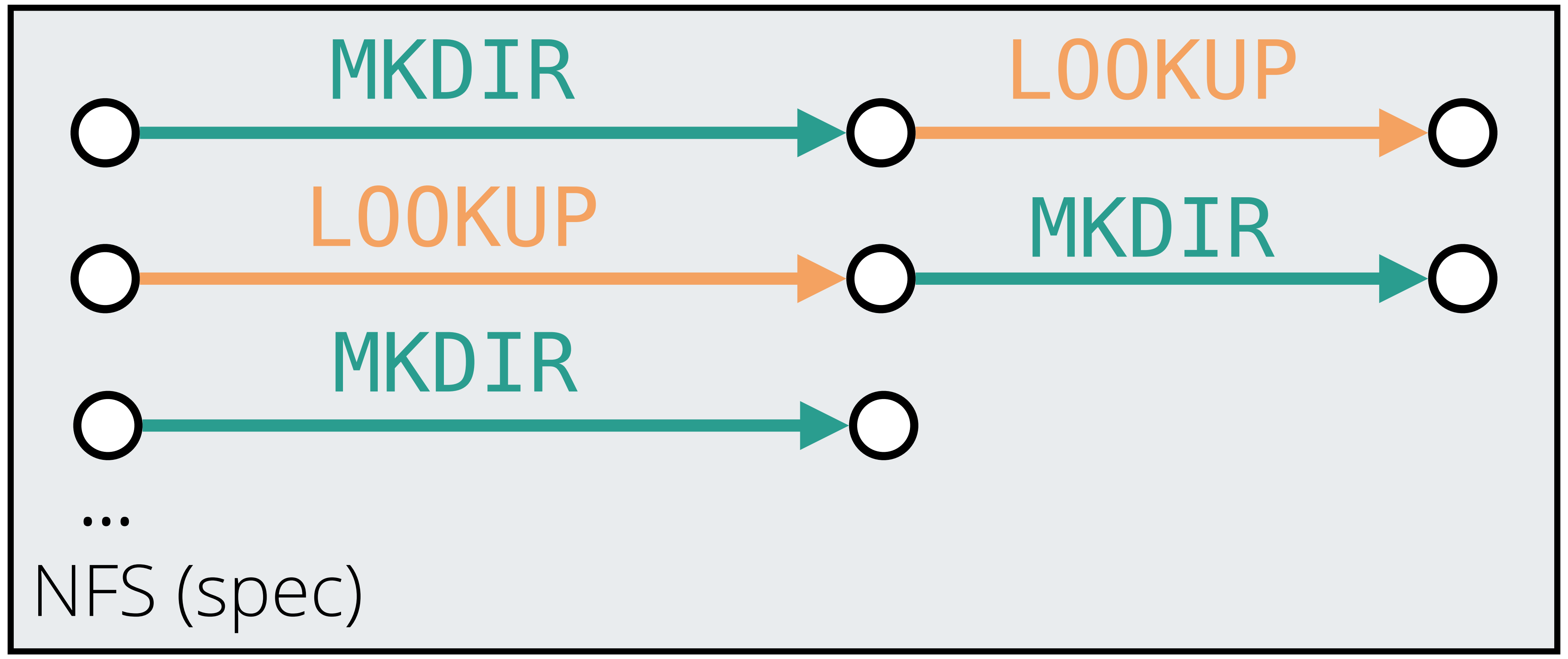
DaisyNFS’s top-level correctness theorem

`MKDIR(…)` ‖ `LOOKUP(…)`



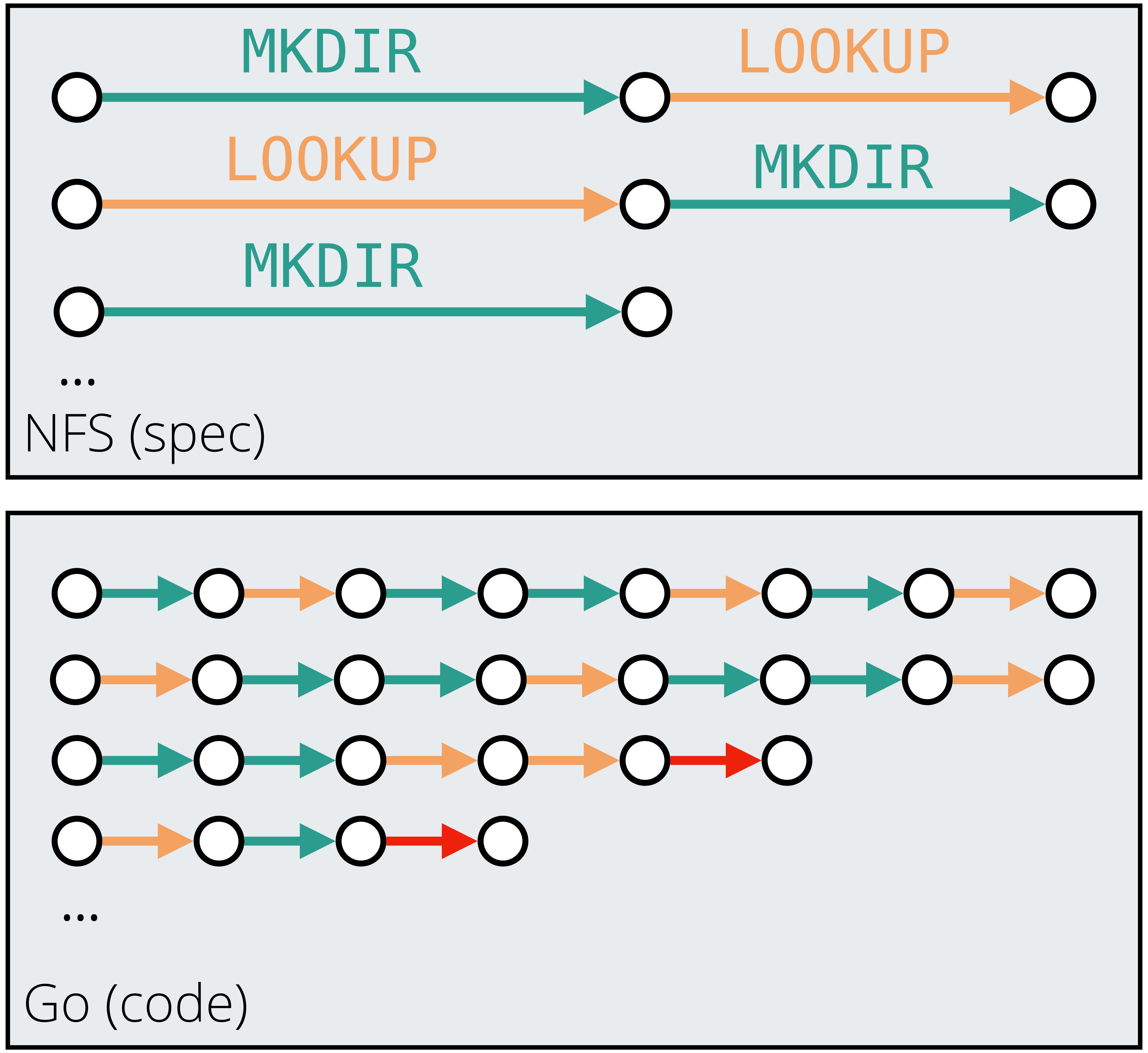
DaisyNFS's top-level correctness theorem

MKDIR(…) ‖ LOOKUP(…)

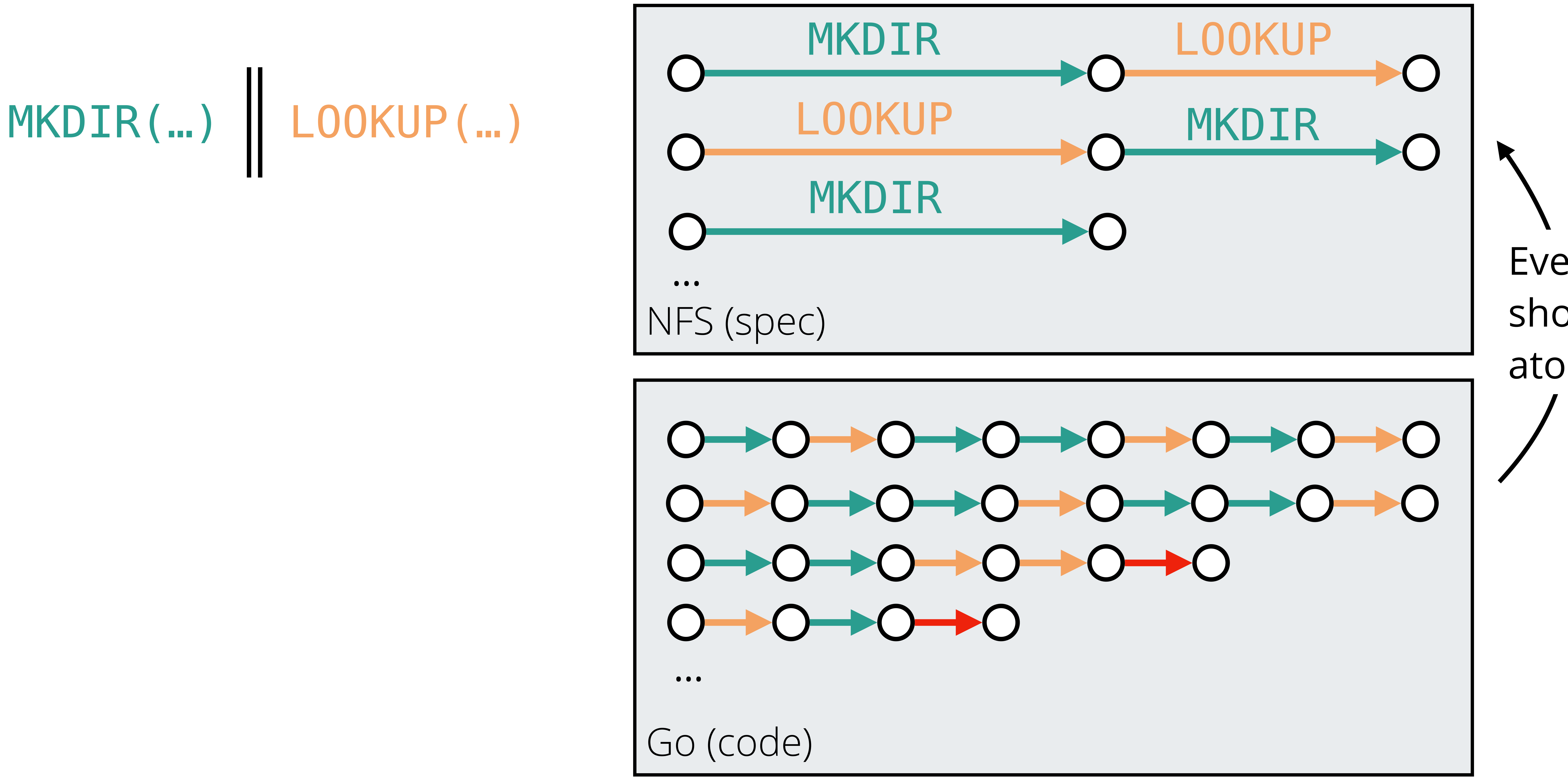


DaisyNFS's top-level correctness theorem

MKDIR(…) || LOOKUP(…)

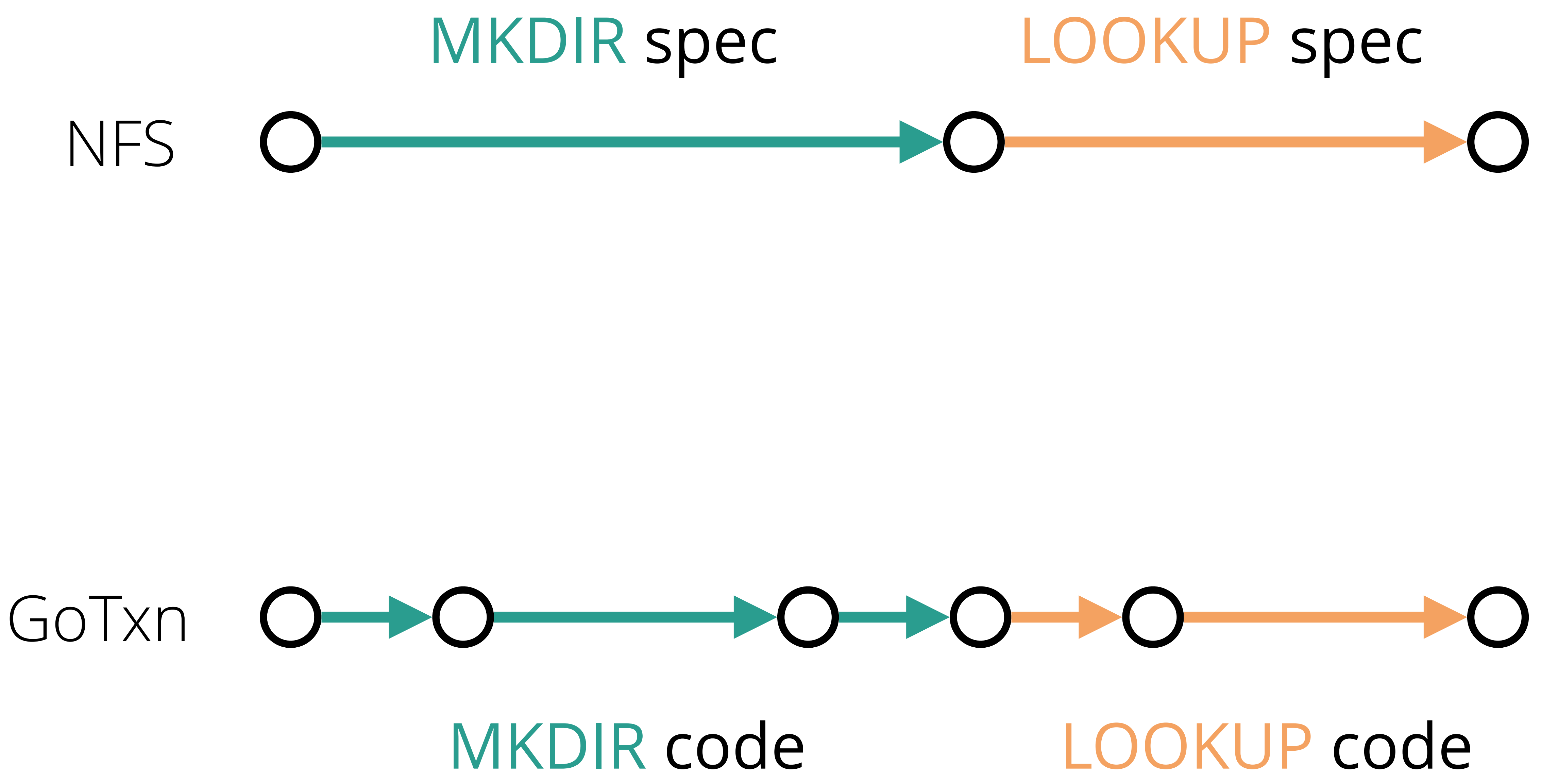


DaisyNFS's top-level correctness theorem

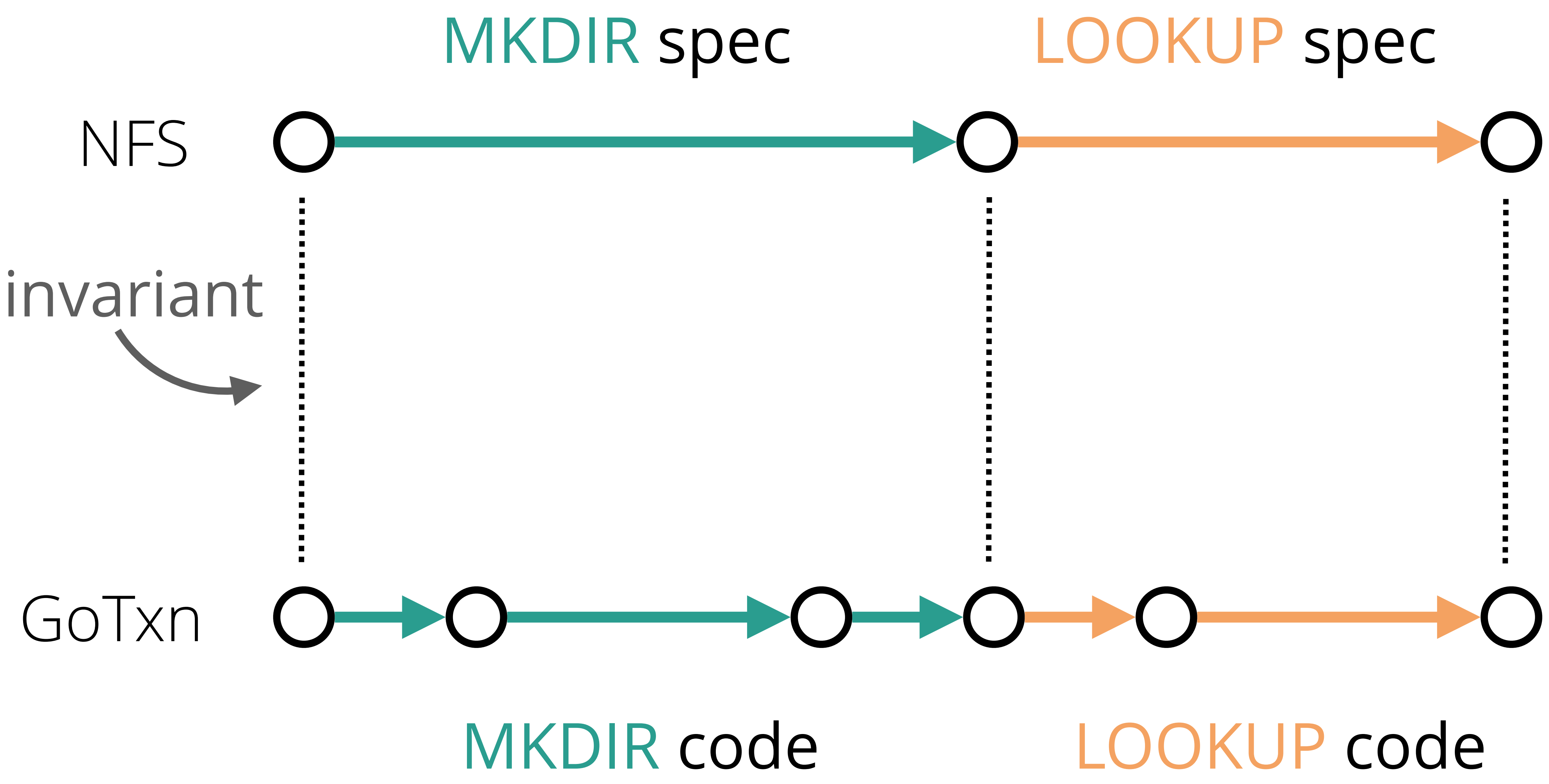


Every `daisy-nfsd` execution should have corresponding atomic execution in spec

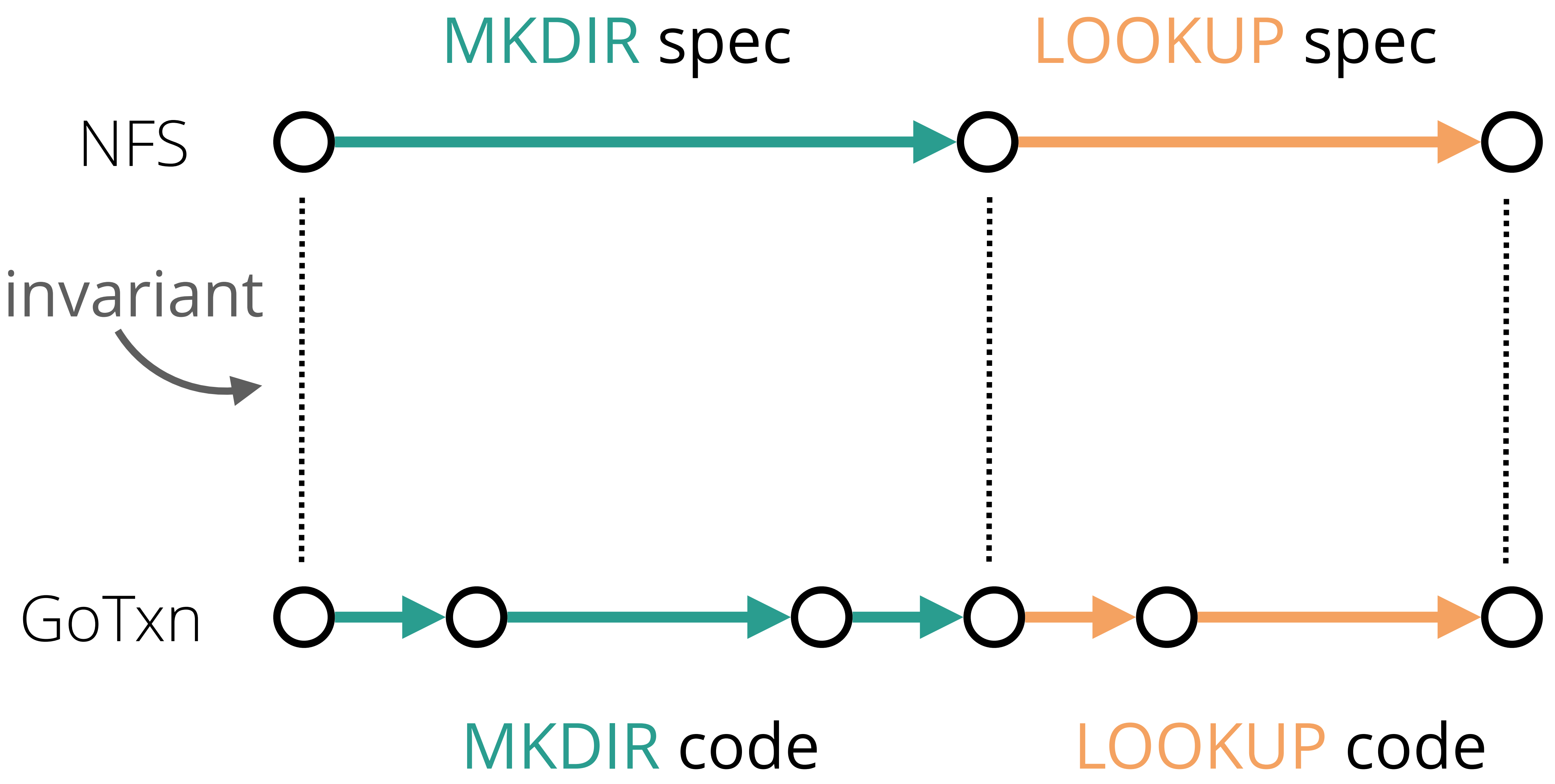
Transactions are proven with sequential reasoning



Transactions are proven with sequential reasoning



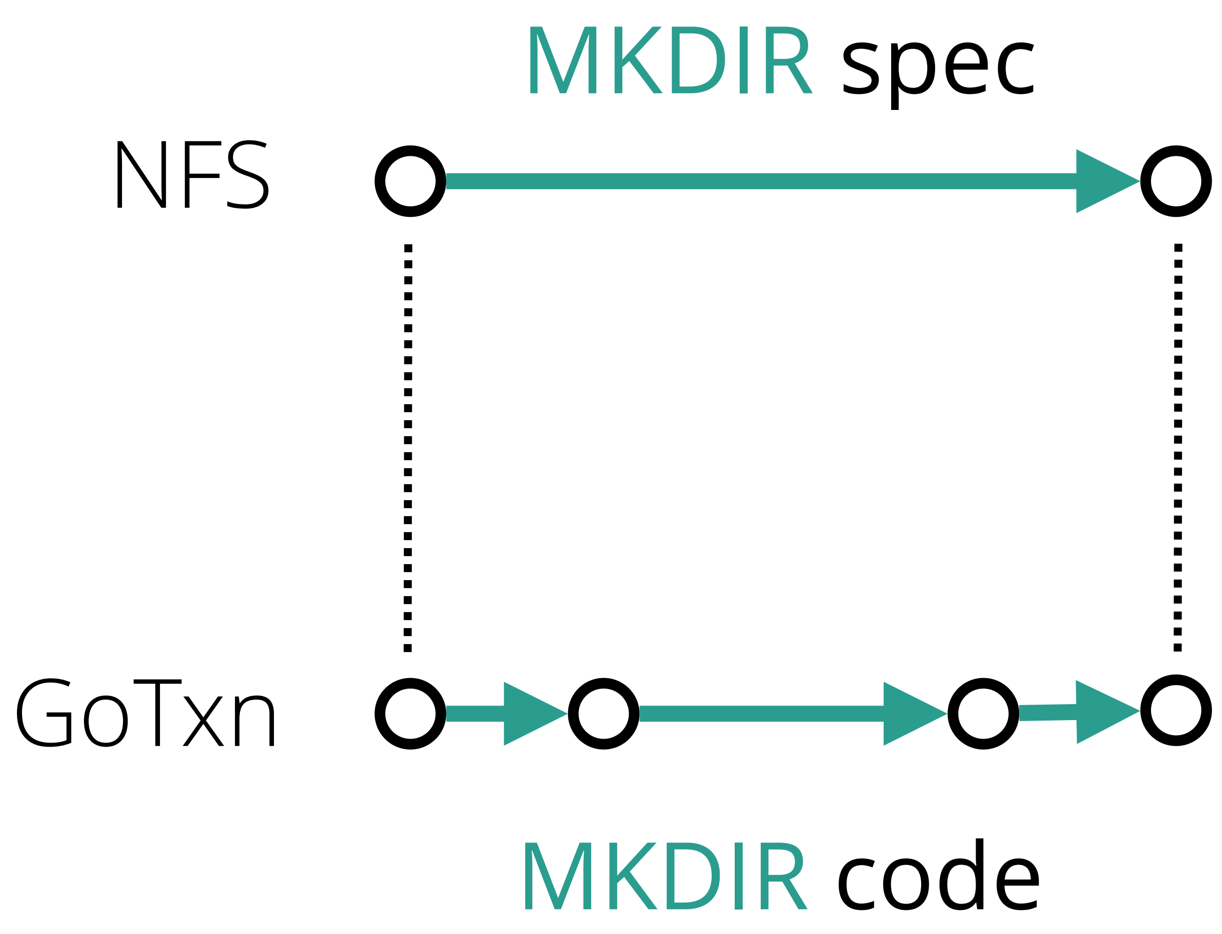
Transactions are proven with sequential reasoning



Sequential reasoning has low proof overhead

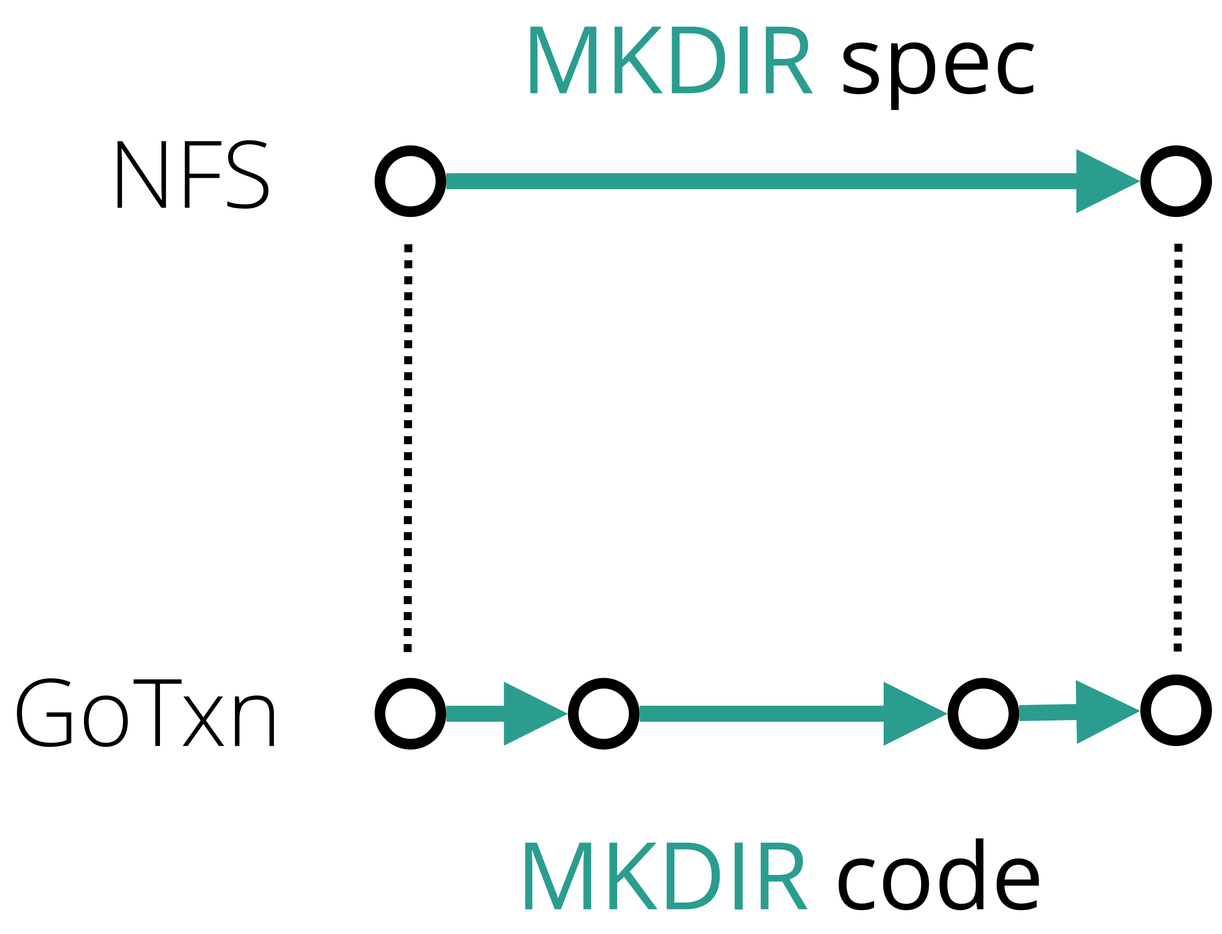
Simulation-transfer theorem

input: forward simulation for every operation

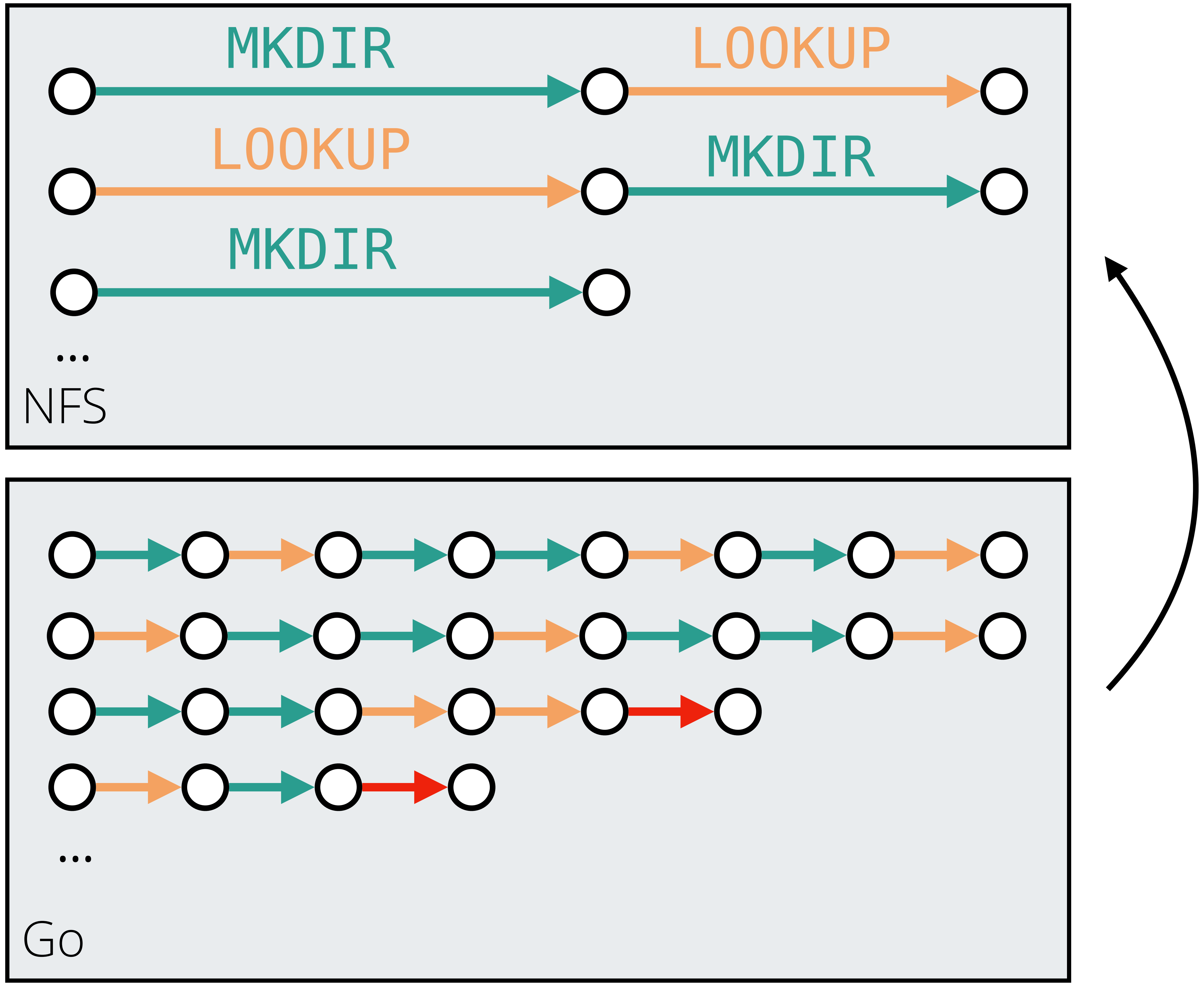


Simulation-transfer theorem

input: forward simulation for every operation

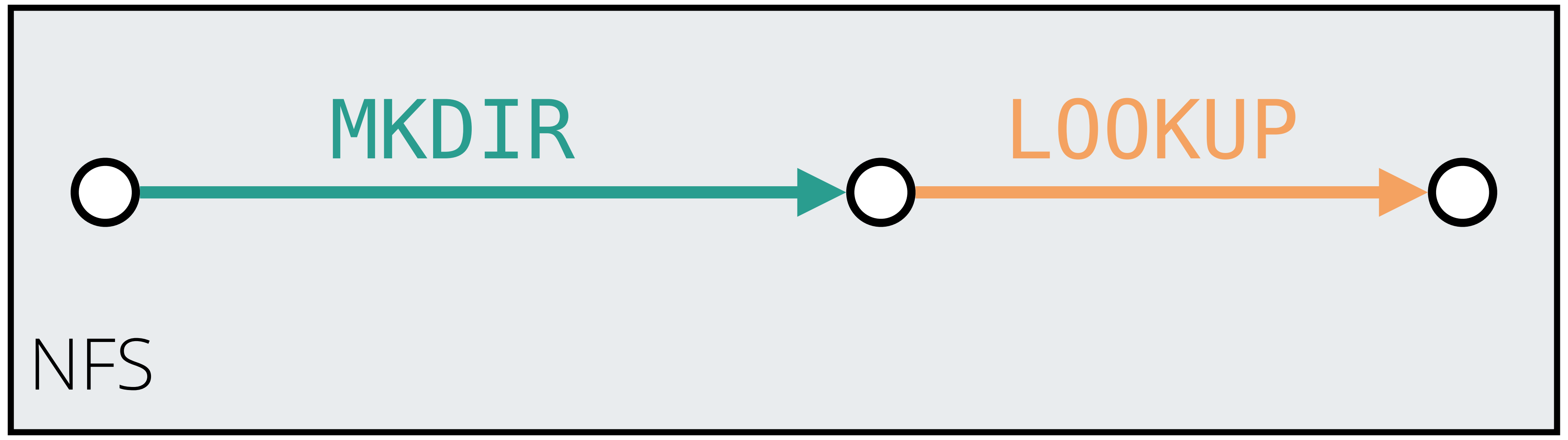


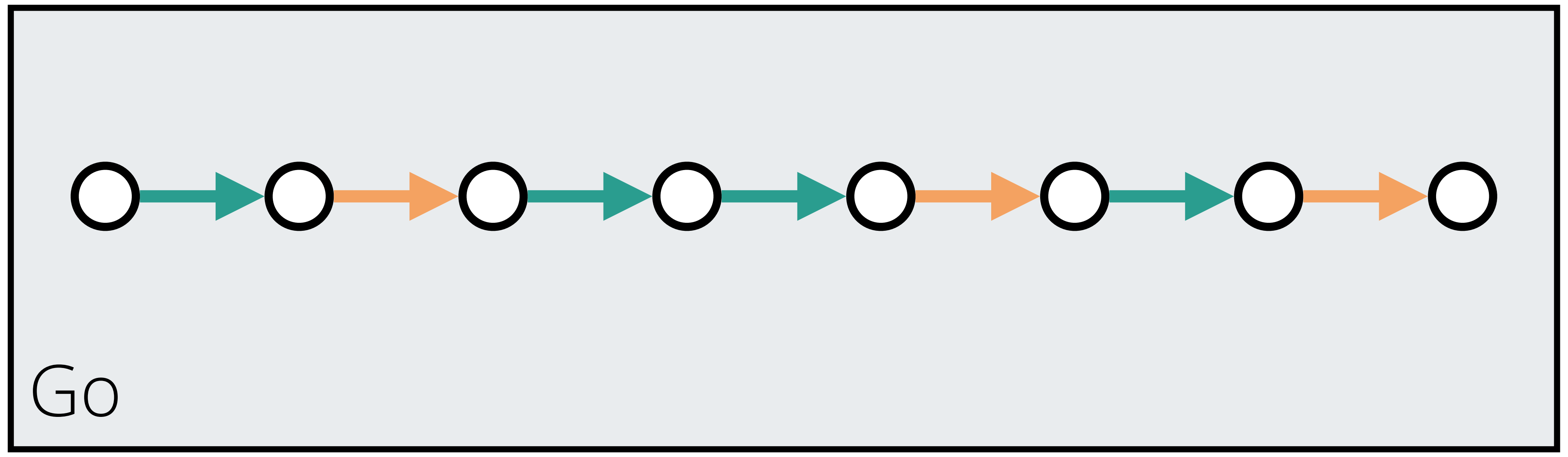
output: concurrent, crash-safe refinement



Proof: compose GoTxn and DaisyNFS proofs

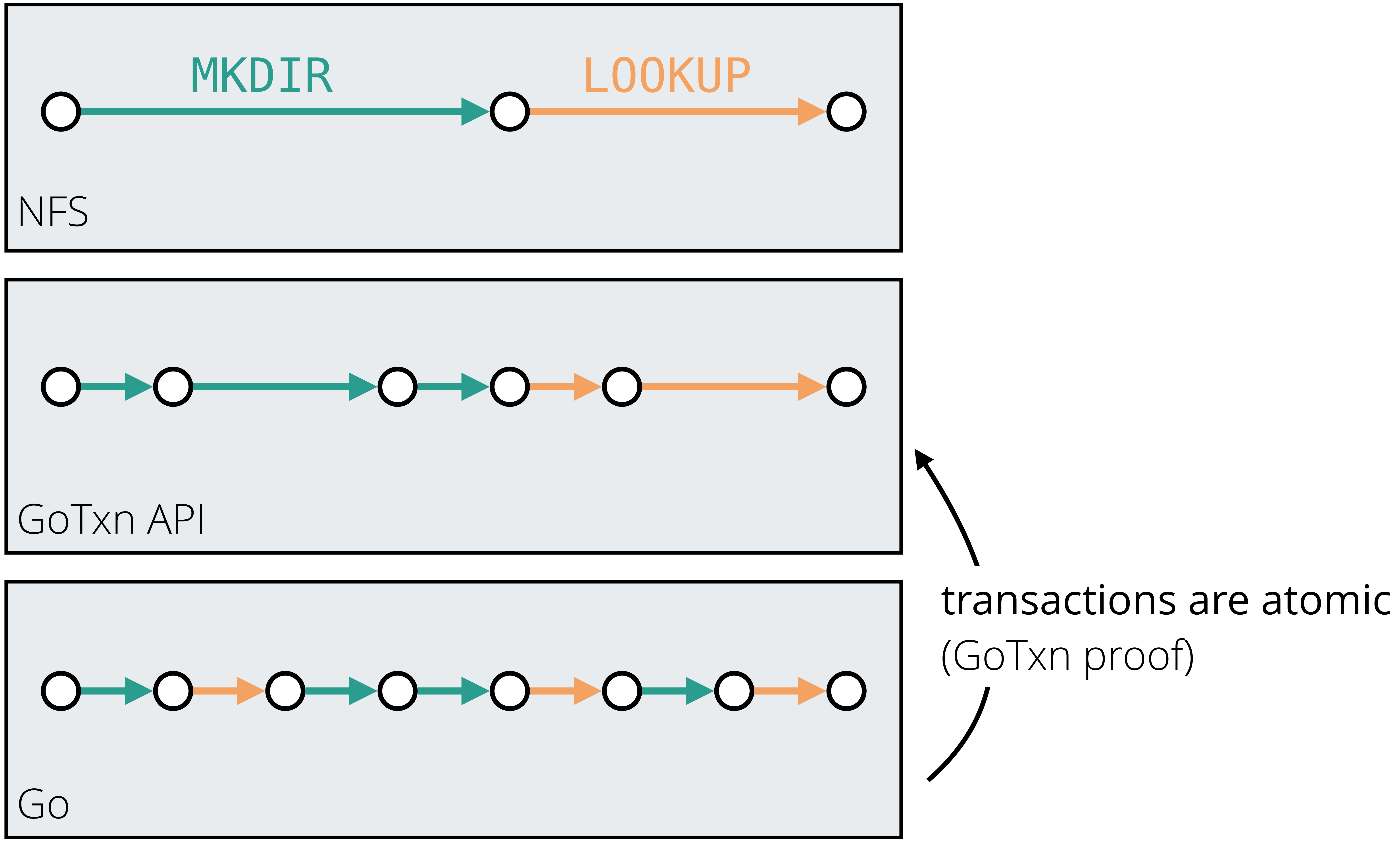
`MKDIR(…) || LOOKUP(…)`





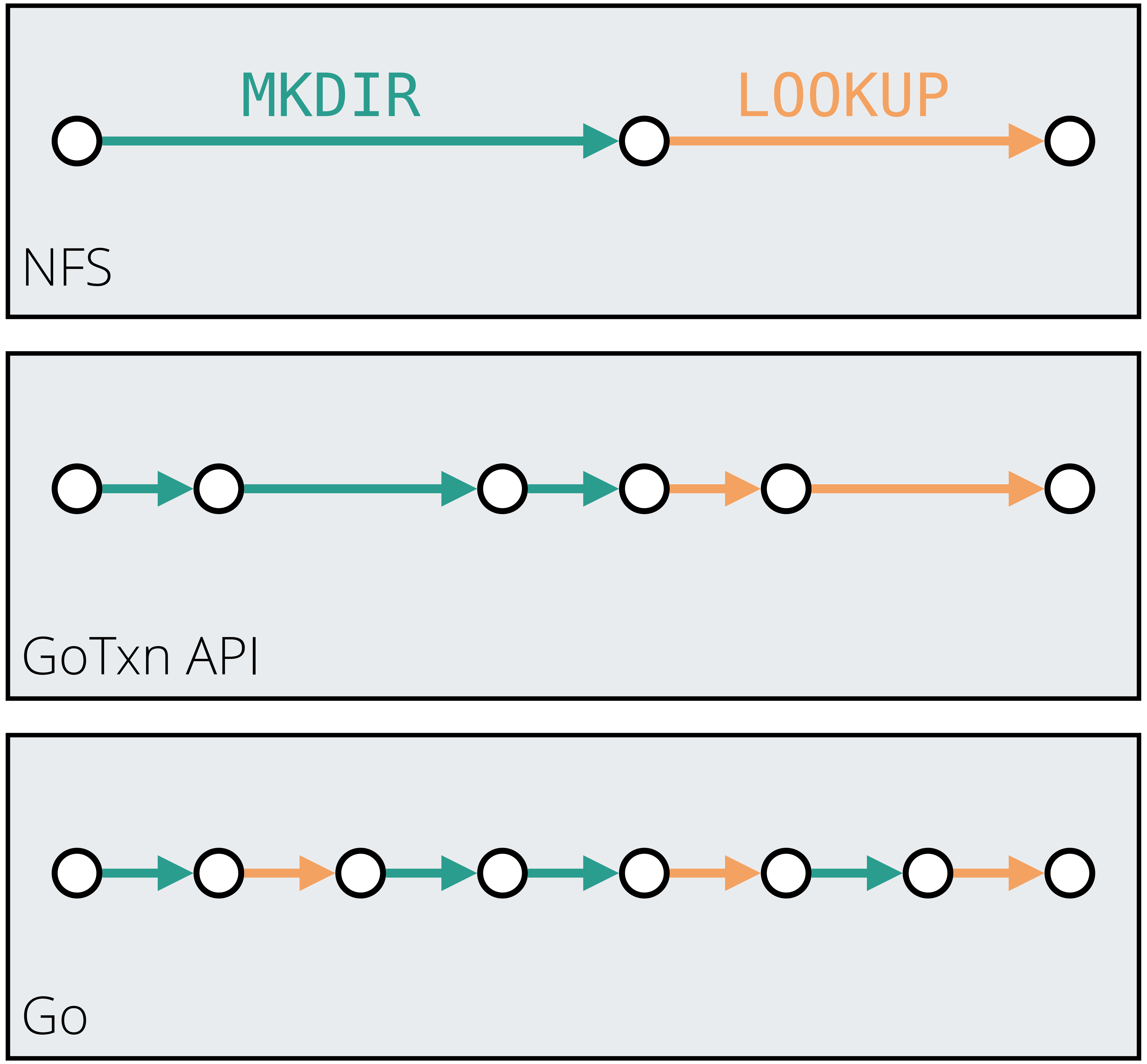
Proof: compose GoTxn and DaisyNFS proofs

MKDIR(…) ‖ LOOKUP(…)



Proof: compose GoTxn and DaisyNFS proofs

MKDIR(…) ‖ LOOKUP(…)



sequential transactions are correct
(DaisyNFS proof)

transactions are atomic
(GoTxn proof)

Theorem needs some assumptions for atomicity

All shared state must go through the transaction system

Theorem needs some assumptions for atomicity

All shared state must go through the transaction system

Challenge: how to integrate in-memory allocator state (for performance reasons)?

Naive in-memory allocator would be slow

```
method WRITE(ino) {
  tx := Begin()
  a := Alloc()
  … // add a to ino
  markUsed(tx, a)
  tx.Commit()
}
```

```
method REMOVE(ino) {
  tx := Begin()
  … // get a from ino
  Free(a)
  markFree(tx, a)
  tx.Commit()
}
```

Naive in-memory allocator would be slow

```
method WRITE(ino) {
  tx := Begin()
  a := Alloc()
  … // add a to ino
  markUsed(tx, a)
  tx.Commit()
}
```

allocator
lock

```
method REMOVE(ino) {
  tx := Begin()
  … // get a from ino
  Free(a)
  markFree(tx, a)
  tx.Commit()
}
```

Naive in-memory allocator would be slow

```
method WRITE(ino) {
  tx := Begin()
  a := Alloc()
  … // add a to ino
  markUsed(tx, a)
  tx.Commit()
}
```

allocator
lock

```
method REMOVE(ino) {
  tx := Begin()
  … // get a from ino
  Free(a)
  markFree(tx, a)
  tx.Commit()
}
```

Our solution: use allocator only as hint

```
method WRITE(ino) {
  tx := Begin()
  a := AllocHint()
  if isUsed(tx, a) {
    tx.Abort()
    return ENOSPC
  }
  … // use a as before
  tx.Commit()
}
```

No lock is held after AllocHint call

Rely on-disk allocator as source of truth

Our solution: use allocator only as hint

```
method WRITE(ino) {
  tx := Begin()
  a := AllocHint()
  if isUsed(tx, a) {
    tx.Abort()
    return ENOSPC
  }
  … // use a as before
  tx.Commit()
}
```

No lock is held after AllocHint call

Rely on-disk allocator as source of truth

`AllocHint` has spec that is sound in a transaction

```
method WRITE(ino) {
  tx := Begin()
  a := AllocHint()
  if isUsed(tx, a) {
    tx.Abort()
    return ENOSPC
  }
  … // use a as before
  tx.Commit()
}
```

Simulation-transfer theorem explicitly allows `AllocHint` and `Free`

Evaluation

Evaluation results

GoTxn reduces proof burden

Bugs found in unverified code and specification

Good performance compared to Linux

Simulation transfer reduces proof overhead

		Code
	DaisyNFS	4,000 (Dafny)
	GoTxn	1,600 (Go)

Simulation transfer reduces proof overhead

	Code	Proof
DaisyNFS	4,000 (Dafny)	6,800 (Dafny)
GoTxn	1,600 (Go)	35,000 (Perennial)

Simulation transfer reduces proof overhead

	Code	Proof
DaisyNFS	4,000 (Dafny)	6,800 (Dafny)
GoTxn	1,600 (Go)	35,000 (Perennial)

2x proof:code

20x proof:code

Bugs found in unverified code and spec

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

File handle parser panics if wrong length

bytes

mic type cast

ed

RENAME can create circular directories

CREATE/MKDIR allow empty name

Proof assumes caller provides bounded inode

RENAME allows overwrite where spec does not

Didn’t find bugs in verified parts

Bugs found in unverified code and spec

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

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

Proof had an unintentional precondition

Dafny

```
type Ino = ino:uint64 | ino < NUM_INODES

method REMOVE(ino: Ino)
  requires invariant()
```

Proof had an unintentional precondition

Dafny

```
type Ino = ino:uint64 | ino < NUM_INODES

method REMOVE(ino: Ino)
  requires invariant()
```

actually means...

```
method REMOVE(ino: uint64)
  requires invariant()
  requires ino < NUM_INODES
```

Proof had an unintentional precondition

Go (unverified)

```
…
fs.REMOVE(req.ino)
…
```

Go code is *assumed* to meet any preconditions

Dafny

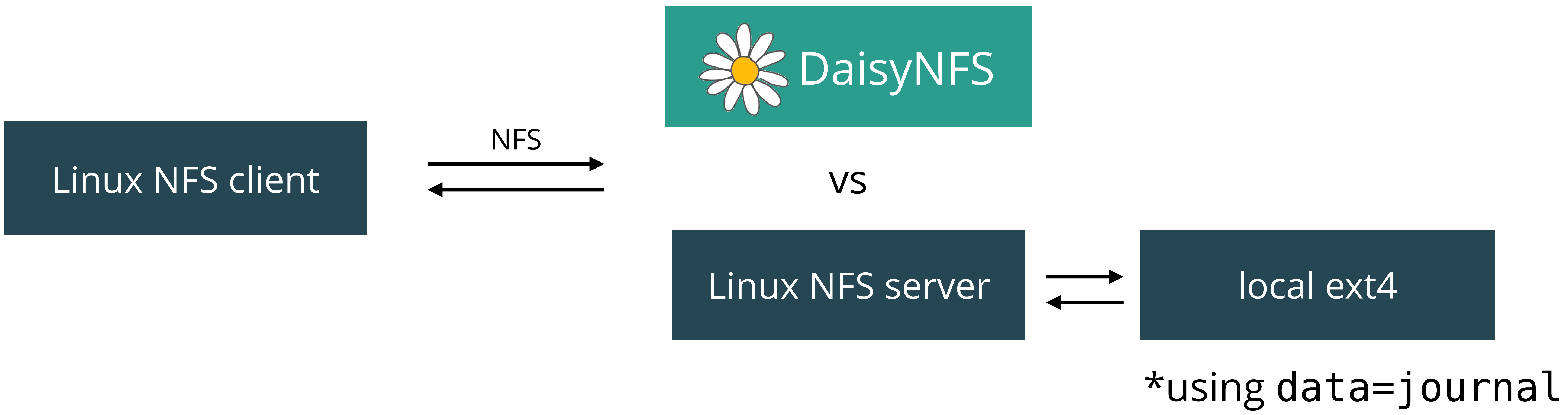
```
type Ino = ino:uint64 | ino < NUM_INODES

method REMOVE(ino: Ino)
  requires invariant()
```

actually means...

```
method REMOVE(ino: uint64)
  requires invariant()
  requires ino < NUM_INODES
```

Compare against Linux NFS server with ext4



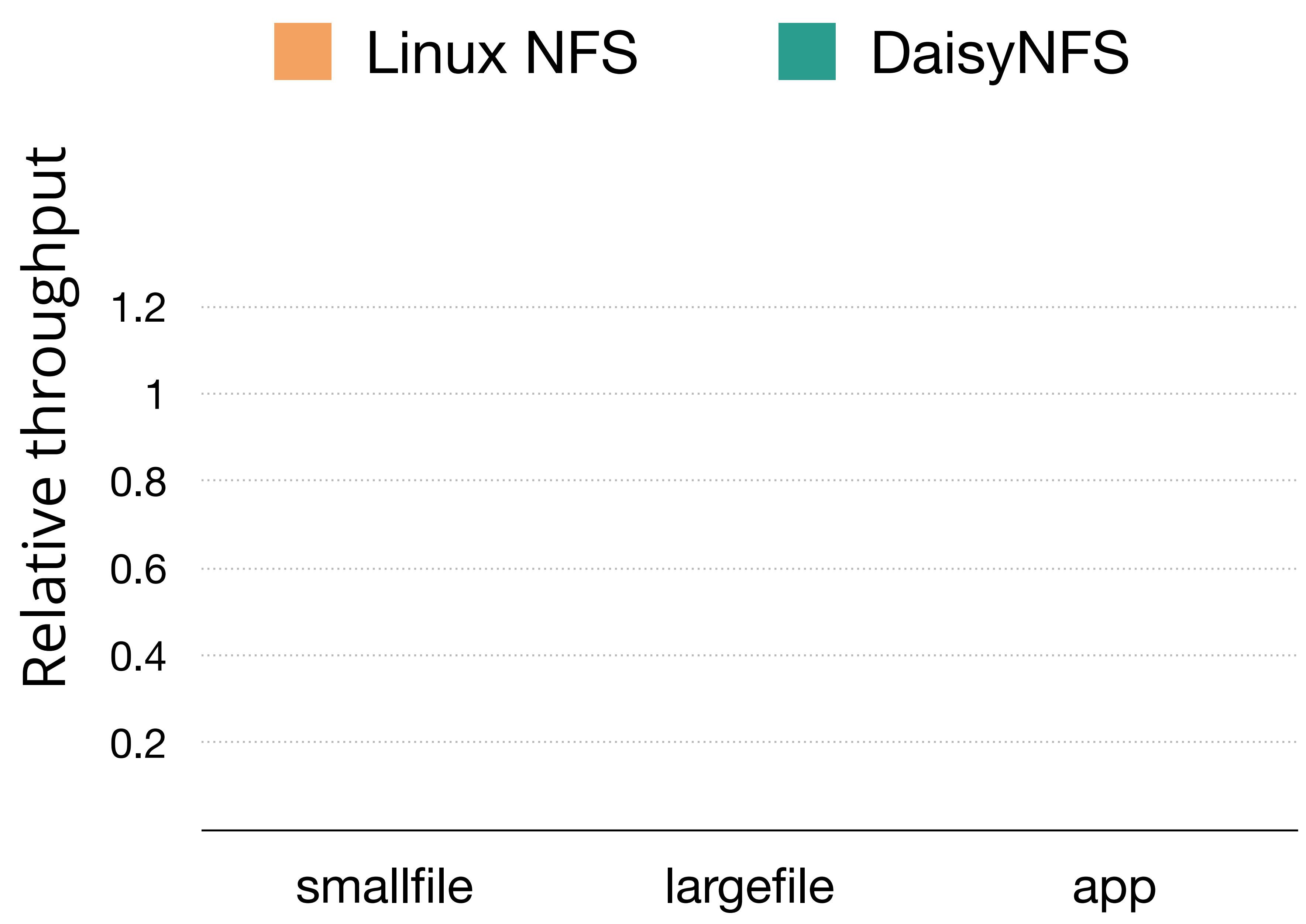
Performance evaluation setup

Hardware: i3.metal instance

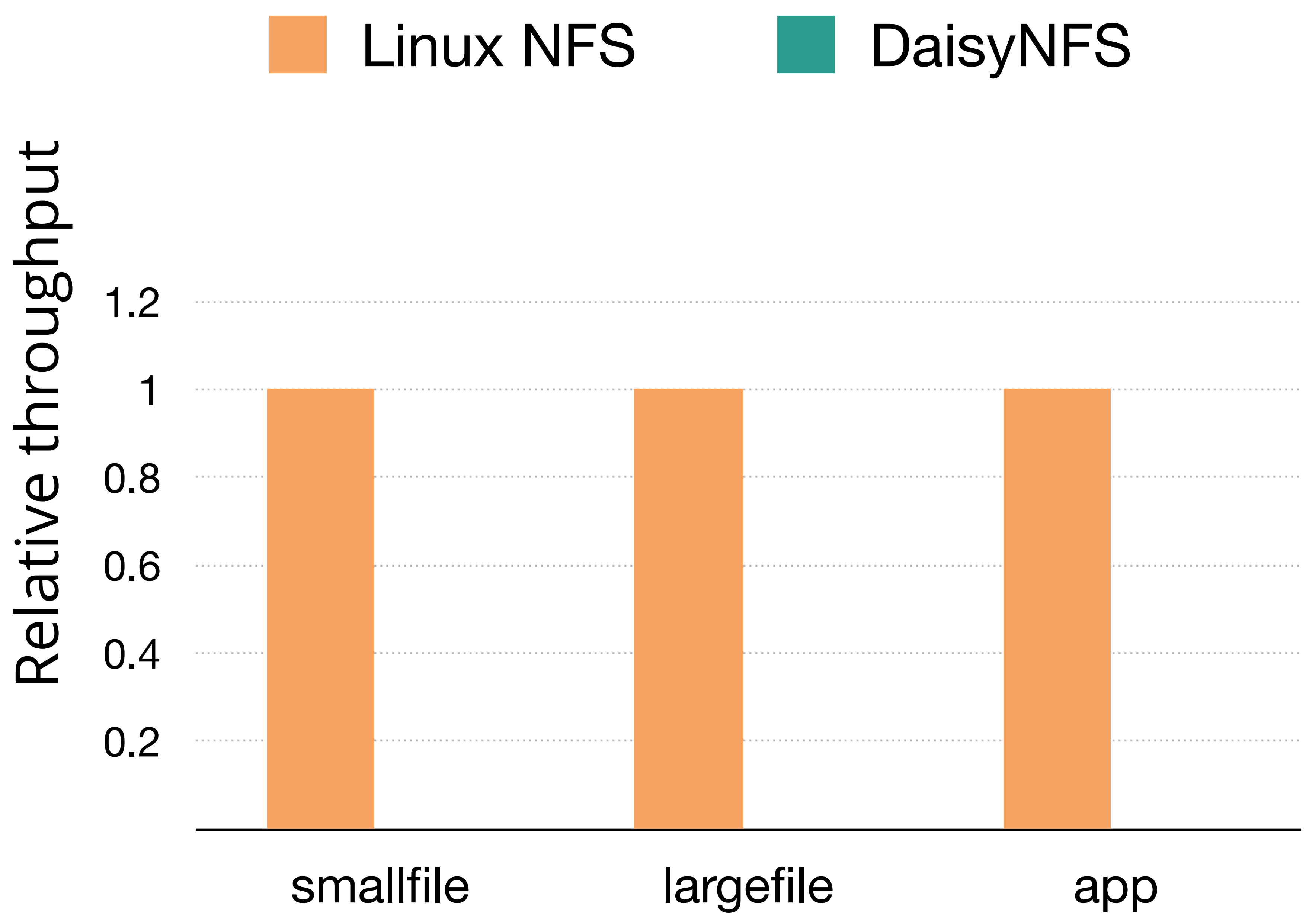
36 cores, fast NVMe drive

Benchmarks:

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

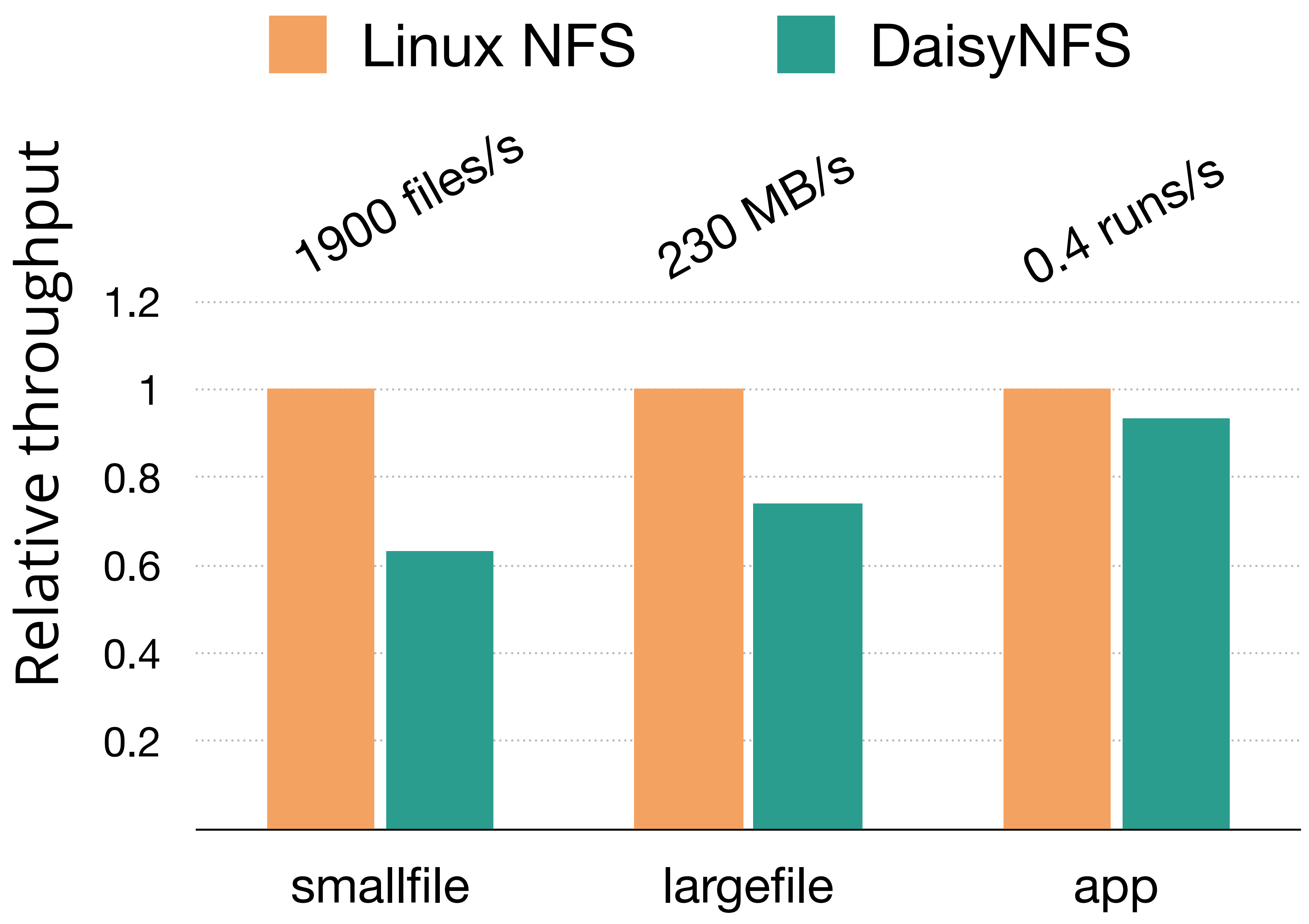


Compare DaisyNFS throughput to Linux, running on an NVMe disk

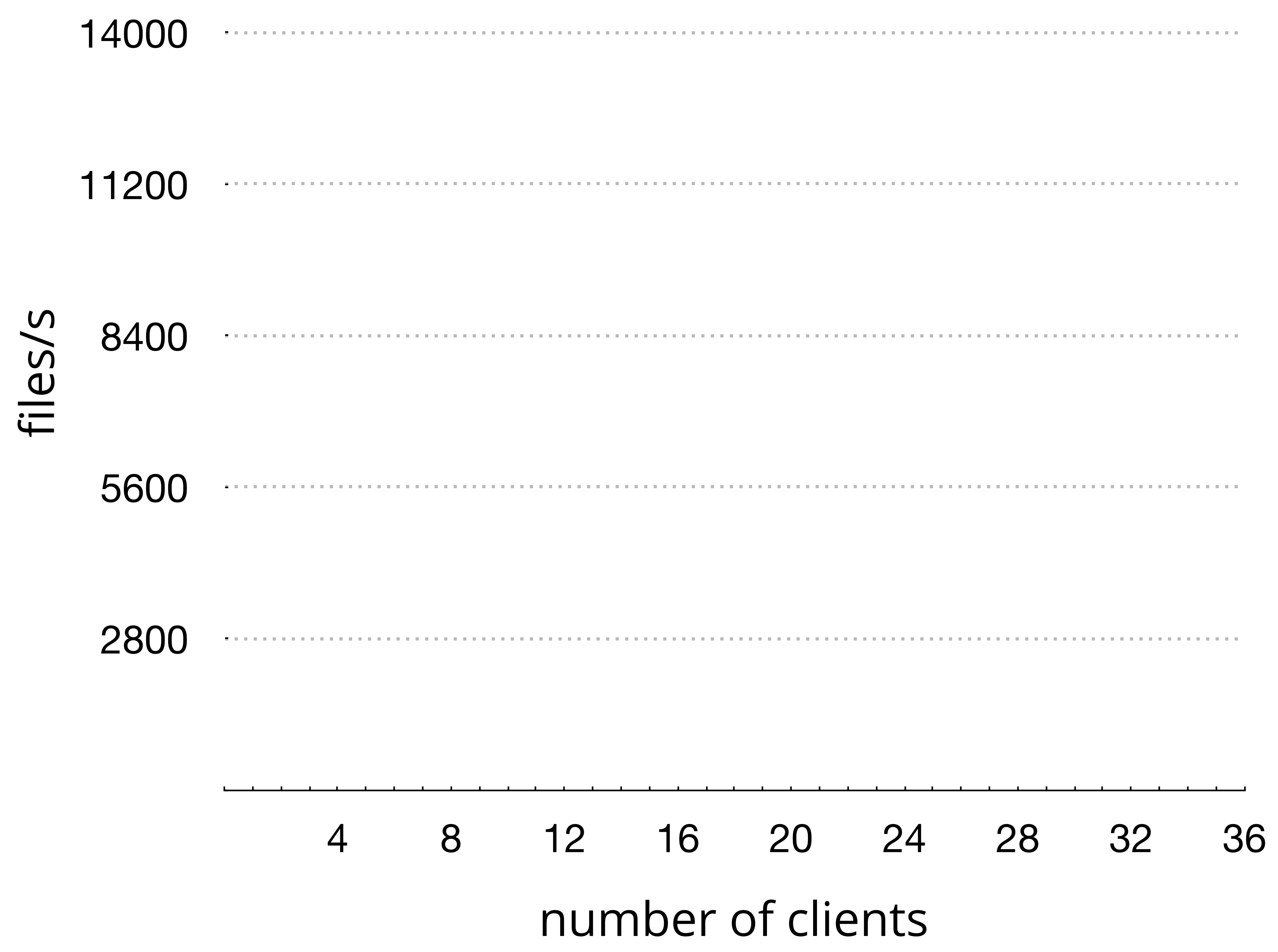


Compare DaisyNFS throughput to Linux, running on an NVMe disk

DaisyNFS gets good performance with a single client

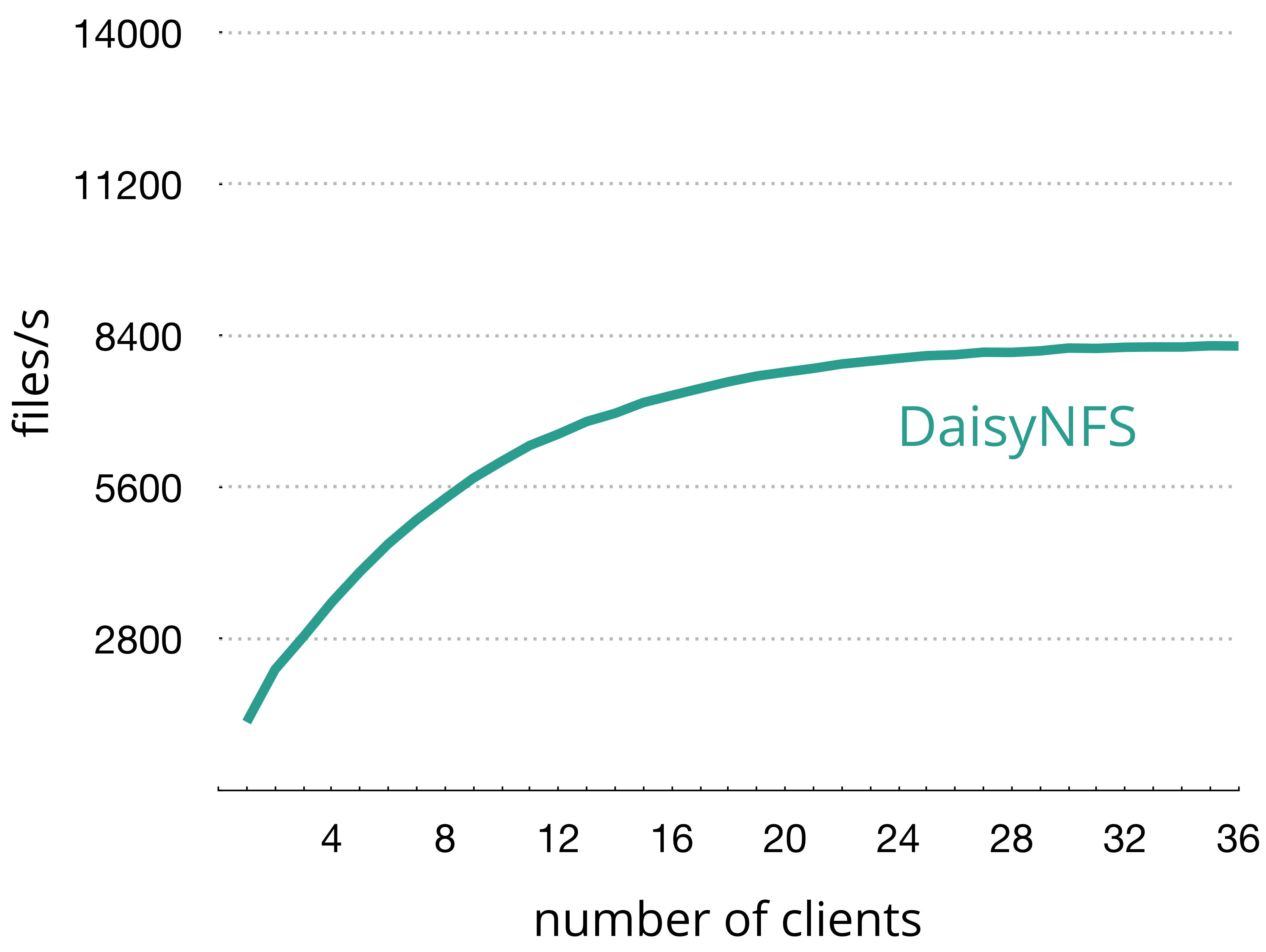


Compare DaisyNFS throughput to Linux, running on an NVMe disk



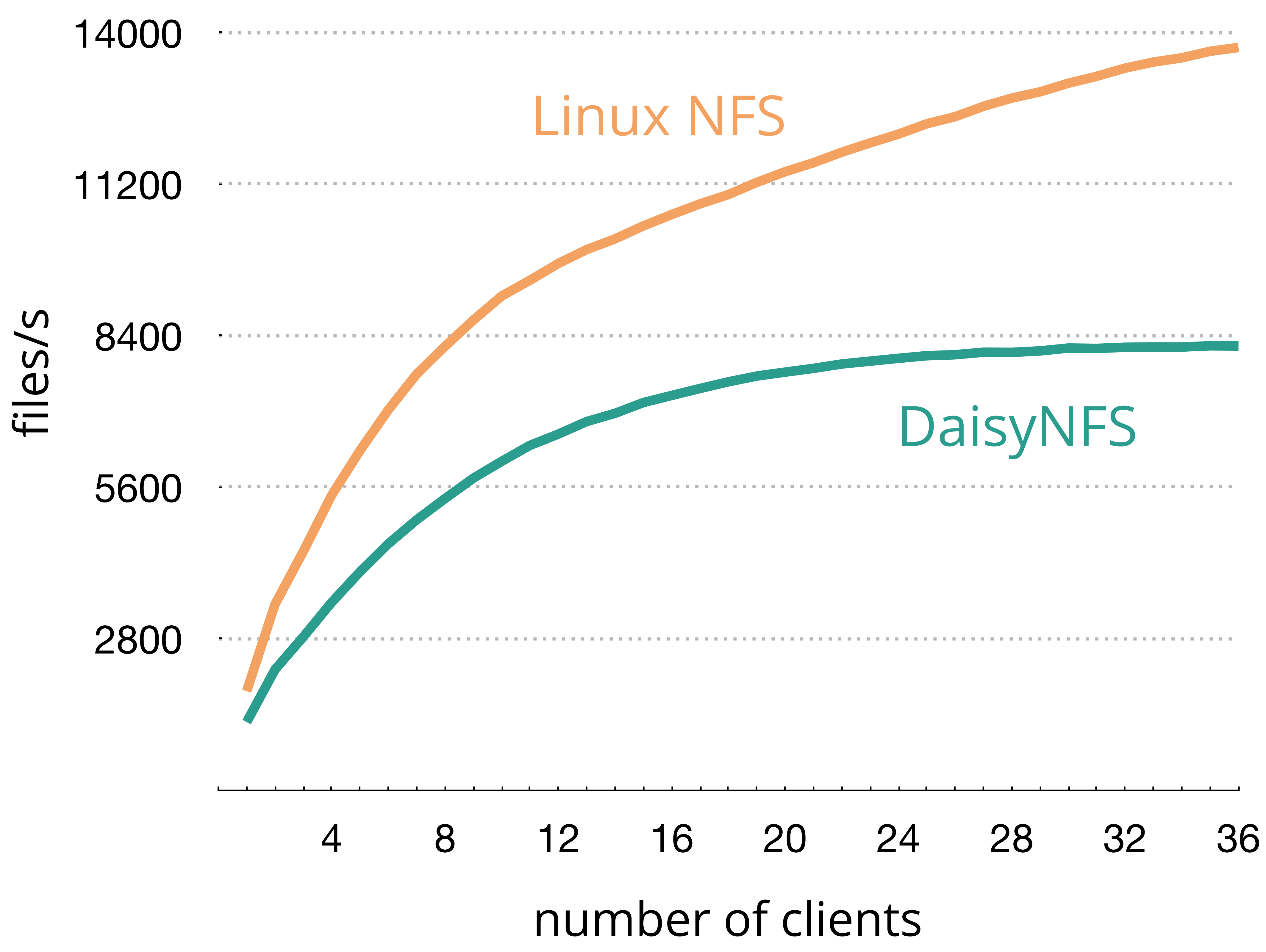
Run smallfile with many clients on an NVMe SSD

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

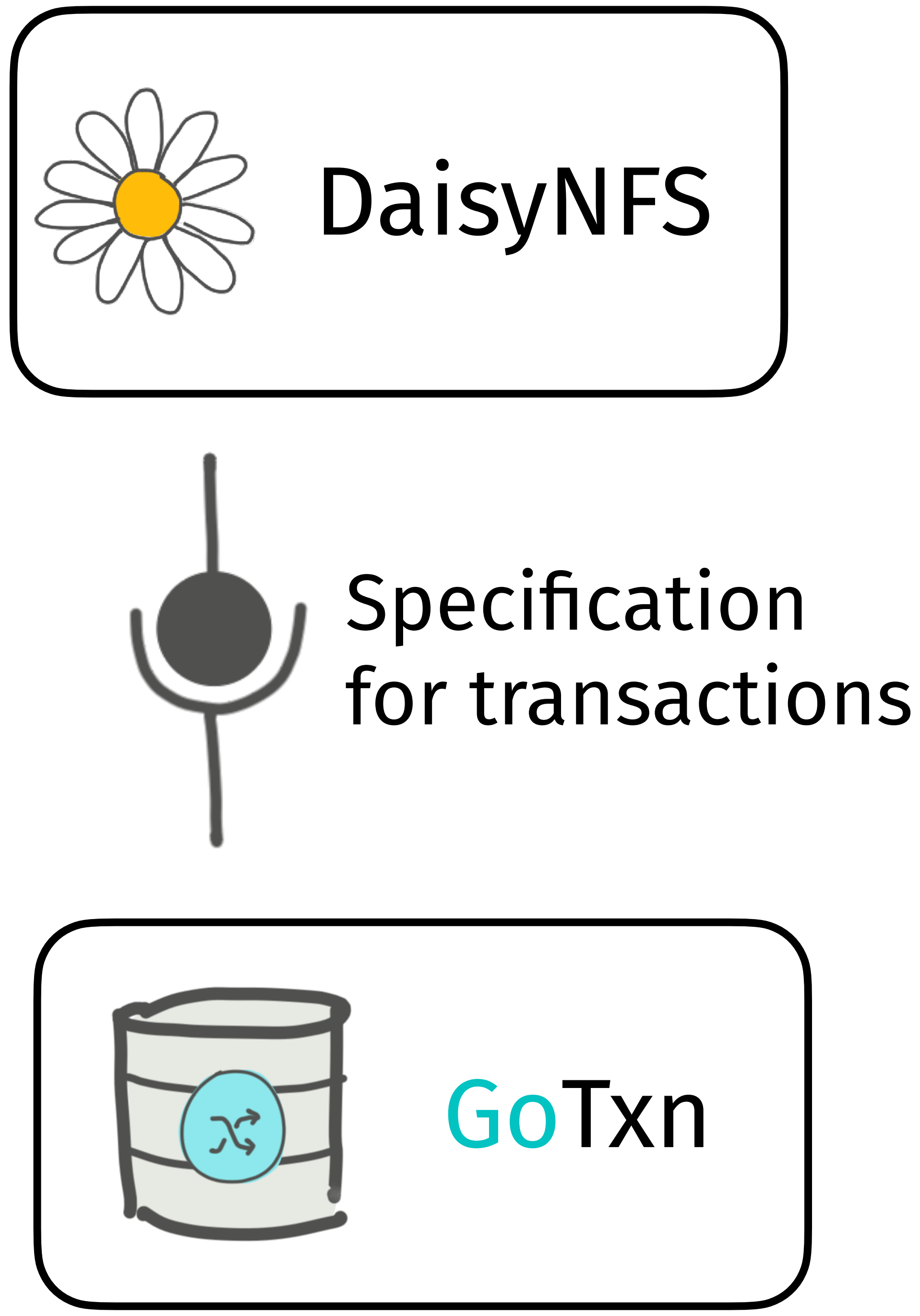
Related work

GoTxn extends GoJournal [OSDI 2021] with two-phase locking

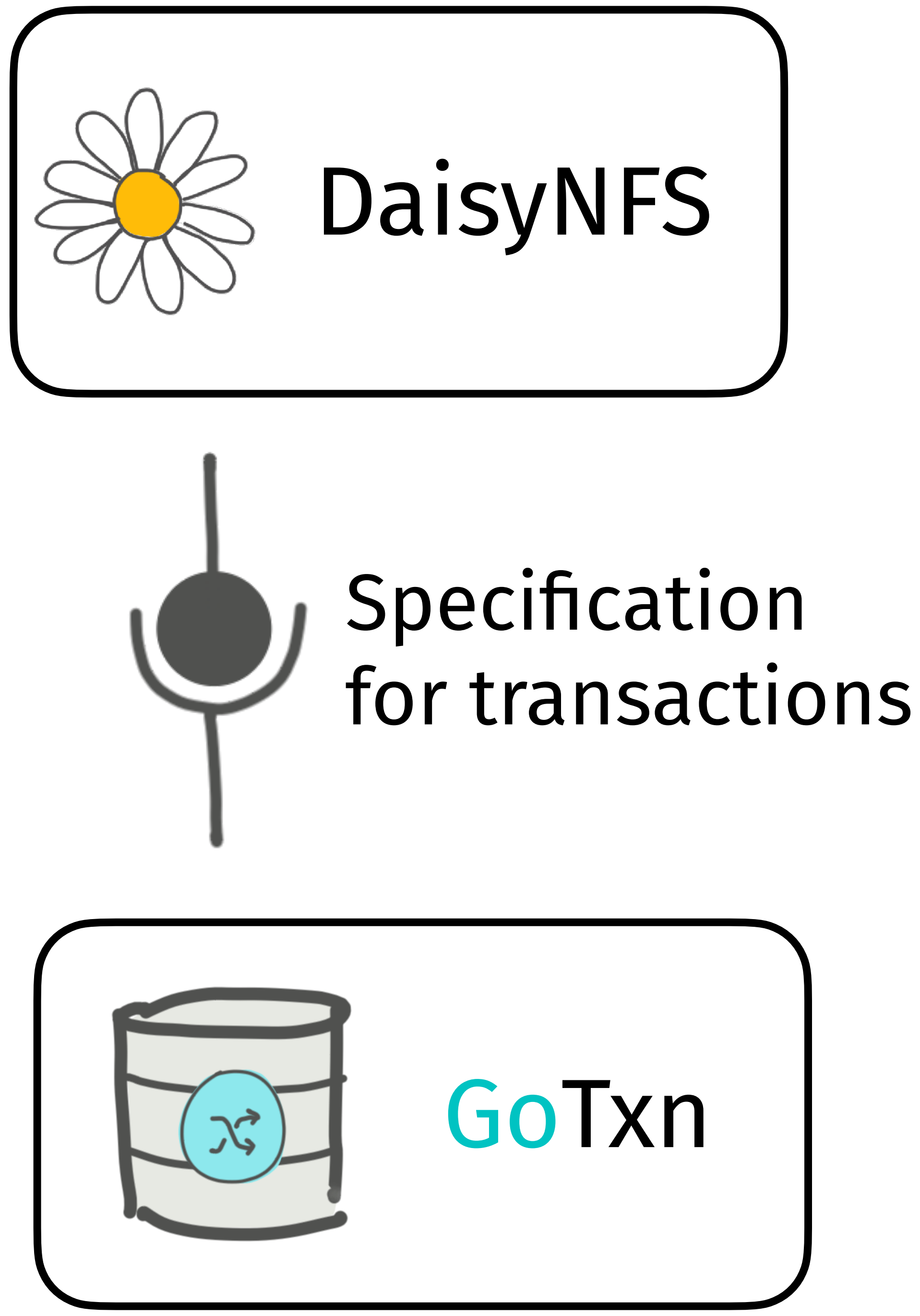
Flashix [2021] is a verified concurrent file system, does not use transactions

Isotope [FAST 2016] has an unverified file system using transactions

Conclusion

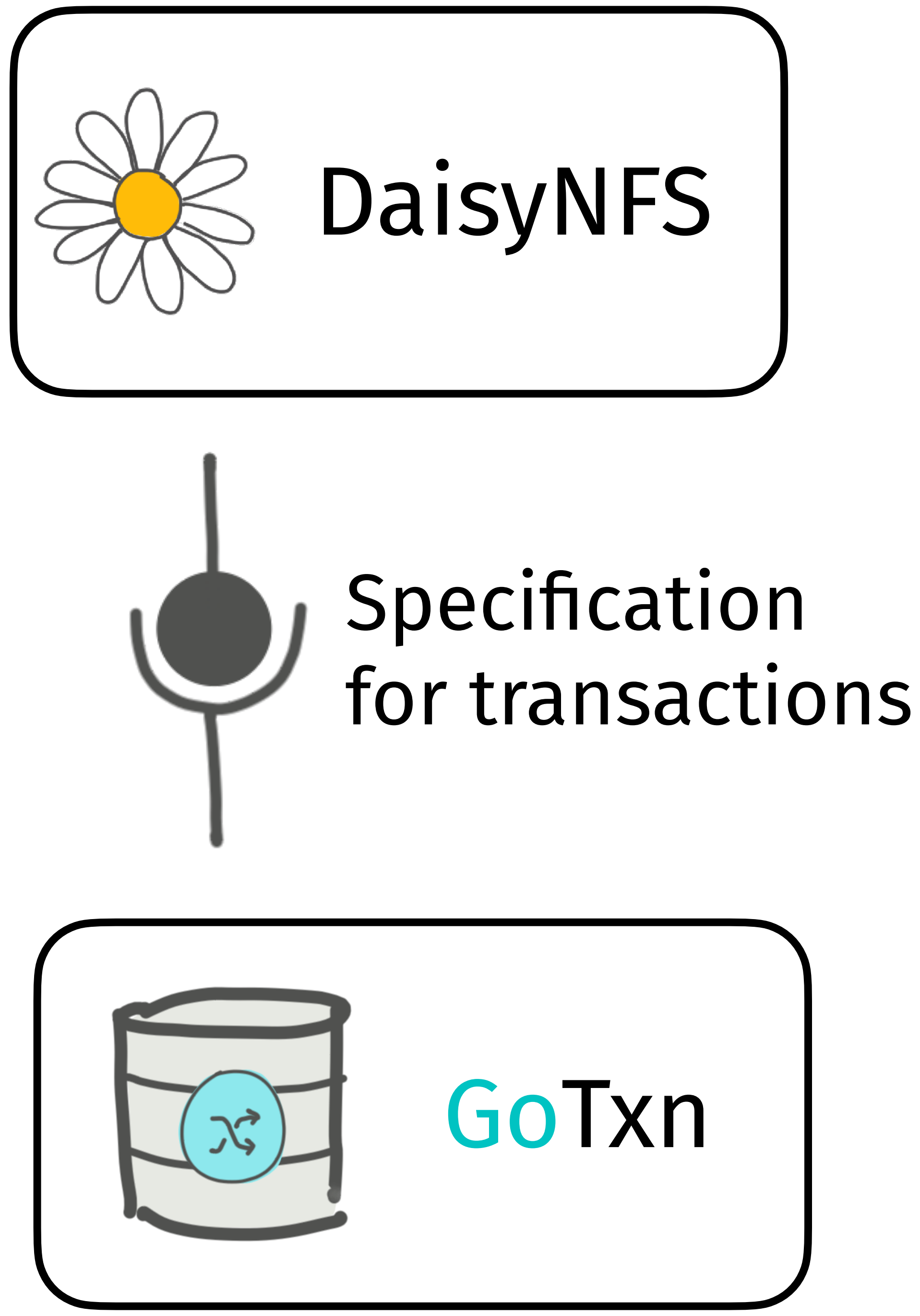


Conclusion



DaisyNFS is a verified, concurrent file system with good performance

Conclusion



DaisyNFS is a verified, concurrent file system with good performance

Simulation-transfer theorem captures how transactions turn sequential reasoning into concurrent & crash-safe correctness