Formal verification of a concurrent file system

Tej Chajed
MIT

Systems software is challenging to get right

Expose an interface to applications

Ext4 PostgreSQL Linux

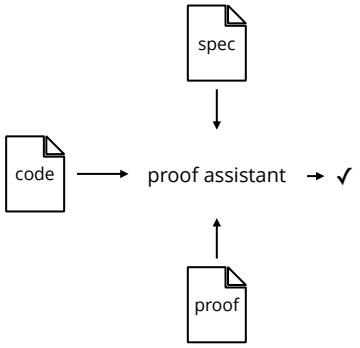
Run in a complex environment

Systems software continues to have bugs

Vision: reliable systems software using verification

Ext4 ✓ PostgreSQL ✓ Linux ✓

Approach: systems verification



Verification can eliminate whole classes of bugs

Memory safety

Concurrency bugs

Logic errors

Security flaws

Systems verification is practical

Microkernels (e.g., seL4, CertiKOS)

Cryptography libraries (e.g., Fiat Crypto, HACL*)

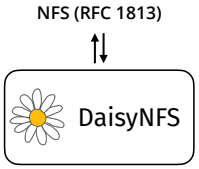
Distributed systems (e.g., Ironfleet, Verdi)

File systems (e.g., FSCQ)

My research spans from foundations to systems

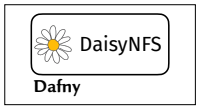
systems	draft '22	**DaisyNFS**	concurrent, crash-safe file system
	OSDI '21	**GoTxn**	high-performance transaction system
	SOSP '15, SOSP '17	**FSCQ**	verified sequential file system
foundations	CoqPL '20	**Goose**	reasoning about Go code
	PLDI '19	**Argosy**	layered recovery procedures
	OSDI '18	**CSPEC**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

DaisyNFS is a verified NFS server



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

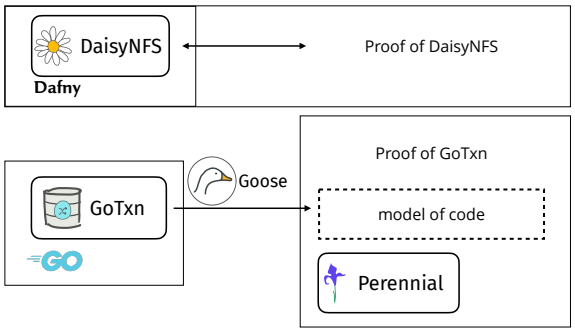
System design



File-system code:
uses one transaction per operation, implements all the features

Transaction system:
automatically gives atomicity, internally concurrent

Verification overview



Main research contribution

First approach for verifying a *high-performance, concurrent, crash-safe* file system.

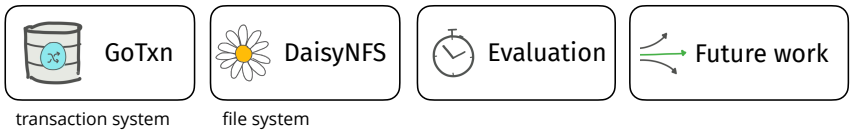
Related work

crash safety:
FSCQ [SOSP '15], Yggdrasil [OSDI '16], DFSCQ [SOSP '17], Argosy [PLDI '19], VeriBetrFS [OSDI '20]

concurrency:
Concurrent GC [MSR Tech Report '15], CertiKOS [OSDI '16], AtomFS [SOSP '19]

crash safety + concurrency:
Flashix [2021]

Outline



GoTxn

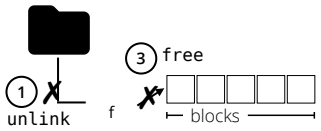
File system needs crash safety

Crash could be due to power failure, kernel panic, etc.

After reboot, file system should recover old data

REMOVE has several steps

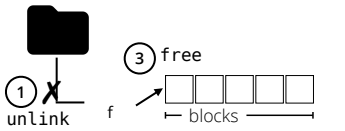
```
func REMOVE(d_ino: uint64,
            name: []byte) {
1 f := unlink(d_ino, name)
2 blocks := getBlocks(f)
3 free(blocks)
}
```



Crashes create subtle bugs

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

crash

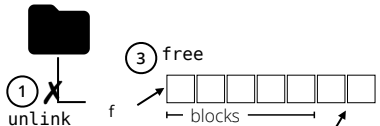


crash leaks f's blocks

Concurrency also create subtle bugs

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

concurrent append



not all blocks are freed

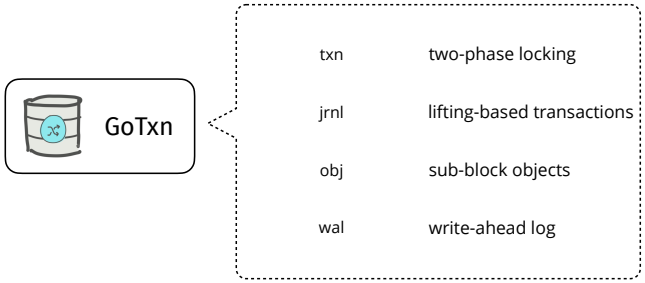
Transactions automatically give atomicity

```
func           Begin() *Txn

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

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

GoTxn

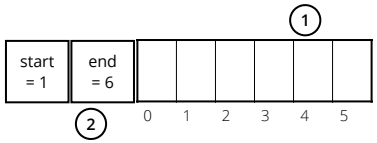
Code between `Begin()` and `Commit()` is **atomic both on crash and to other threads**

Crashes & concurrency are challenges throughout GoTxn

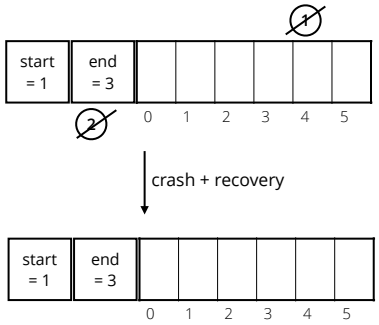


Multi-block atomic writes come from on-disk log

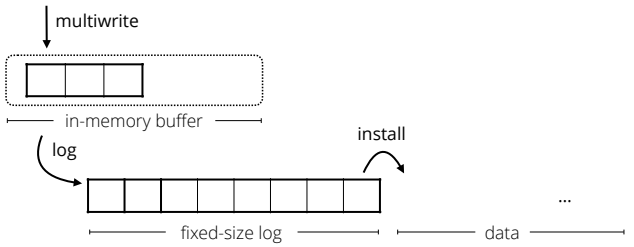
Multi-block atomic writes come from on-disk log



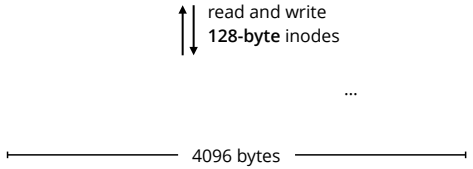
Crash never leaves a partial multiwrite



Writing, logging, and installation are concurrent

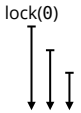


Object layer supports concurrent sub-block access



Two-phase locking handles concurrency

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




```
obj.Multiwrite(…)
tx.releaseAll()
```

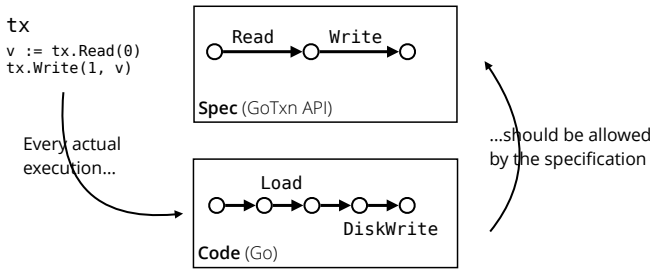

Verification challenges

Concurrency and crash safety issues in every layer

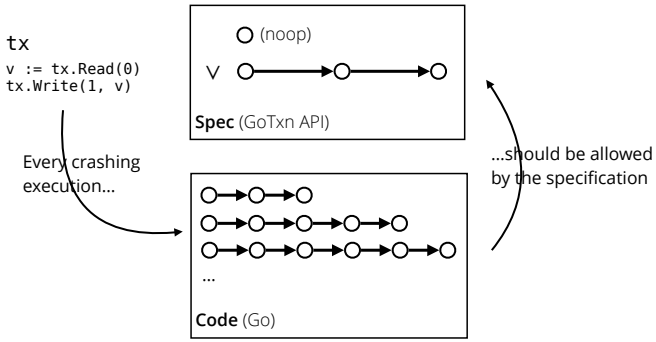
Composing proofs of layers

Specifying the transactional API

Specifying a transaction system

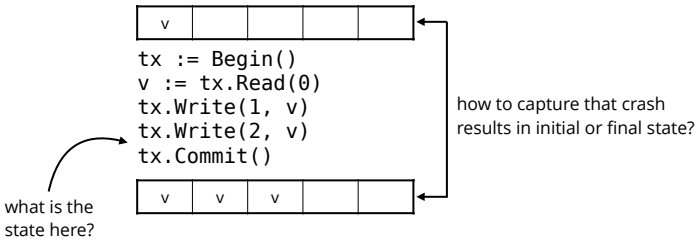


Crash atomicity for sequential transactions



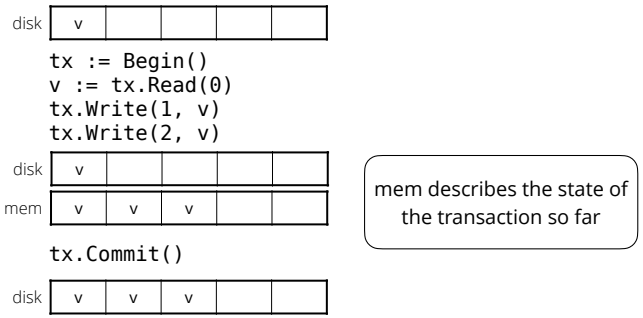
Specifying sequential transactional API

[SOSP '15]

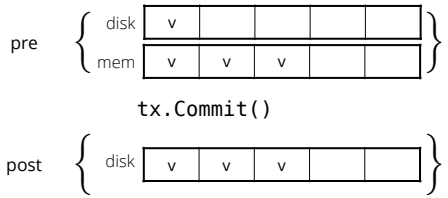


Specifying sequential transactional API

[SOSP '15]

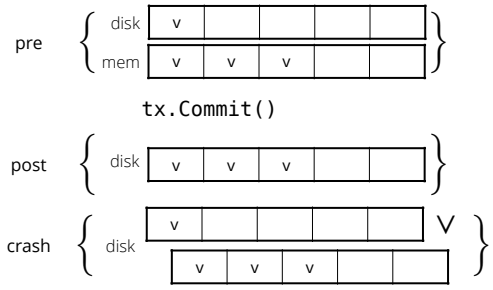


Hoare Logic to specify Commit without crashes

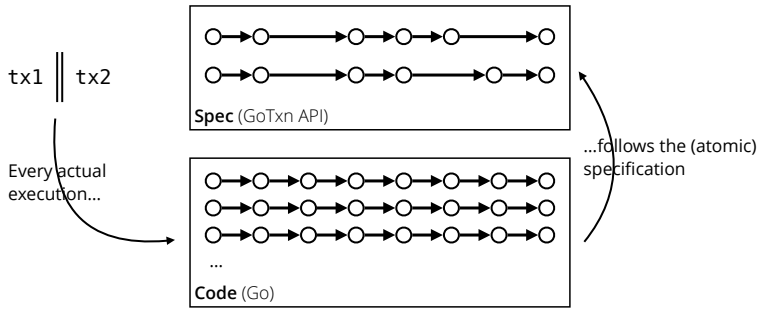


Crash Hoare Logic for sequential crash safety

[SOSP '15]



GoTxn also needs to handle concurrency



Challenge: specifying concurrent transactions

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

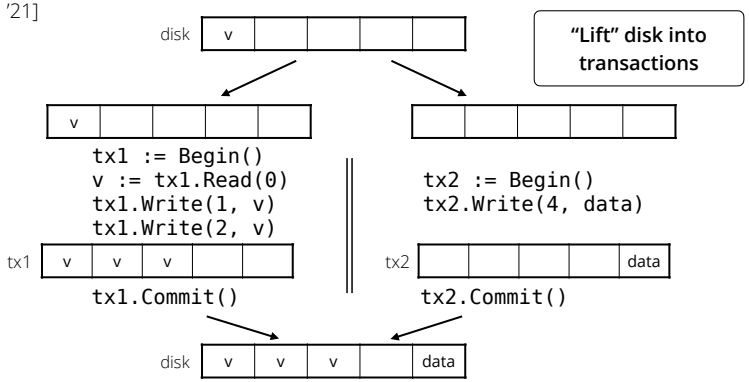
‖

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

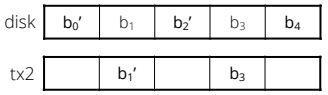
How to specify effect of concurrent transactions?

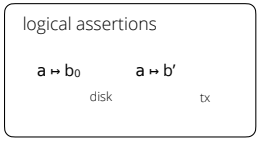
Idea: lifting-based specification

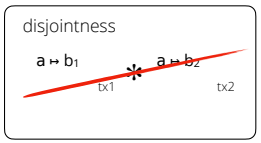
[OSDI '21]



Separation logic describes lifting

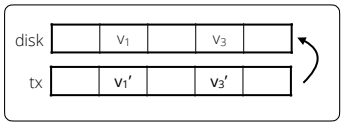






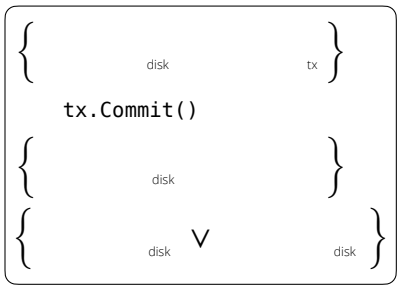
Commit spec captures atomicity


```
tx.Commit()
```

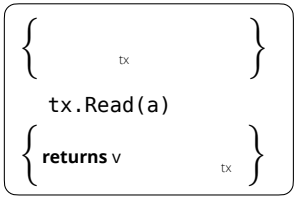


crash condition is atomic

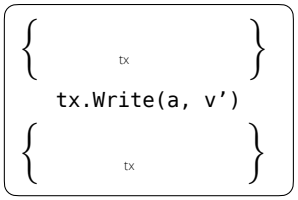
Lifting specification describes the GoTxn API




```
tx.Commit()
```

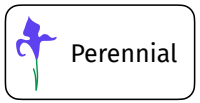




```
tx.Read(a)
```


```
tx.Write(a, v')
```

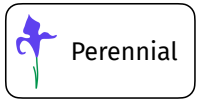
Perennial allows us to encode the spec



Builds on Iris [JFP 2018]

Extend with crash conditions

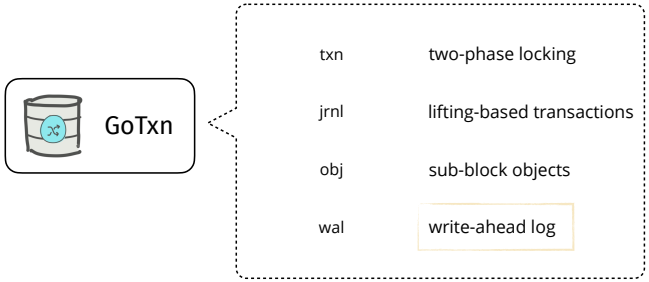
Perennial and Goose make the proof possible



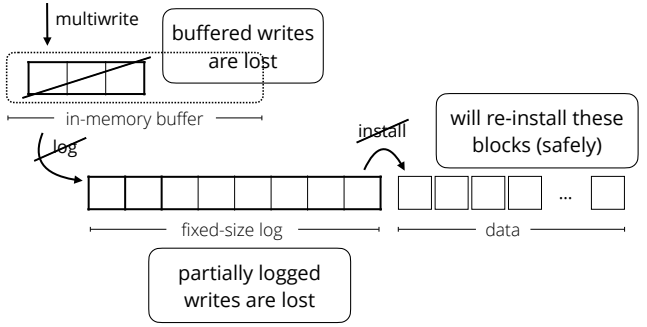
Reason about concurrency and crash safety in each layer

Connect proofs to efficient code

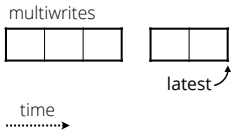
Excerpt from proof: focus on WAL



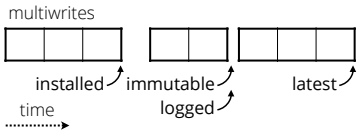
Crashes are complicated in the WAL



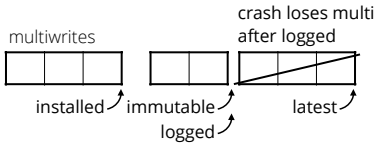
Idea: model WAL using history of multiwrites



Idea: model WAL using history of multiwrites

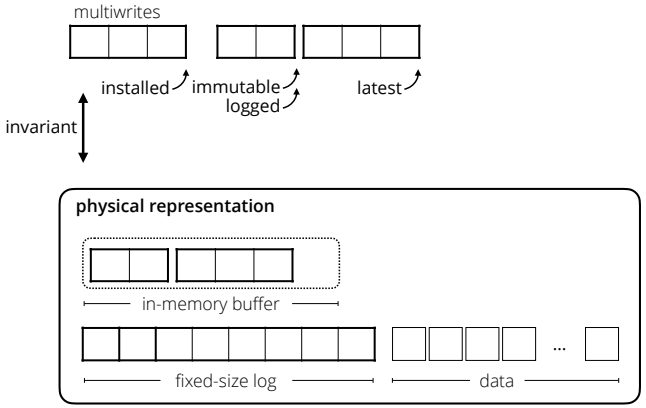


Crashes are easily expressed in this abstraction

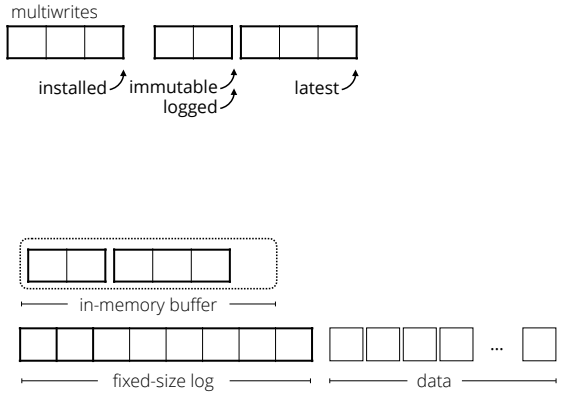


captures all relevant aspects of physical state

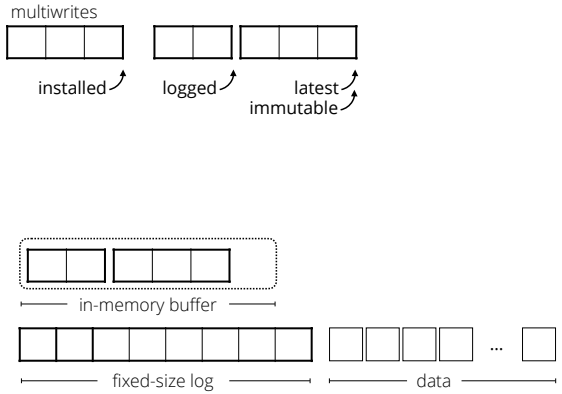
Invariant connects abstraction to physical representation



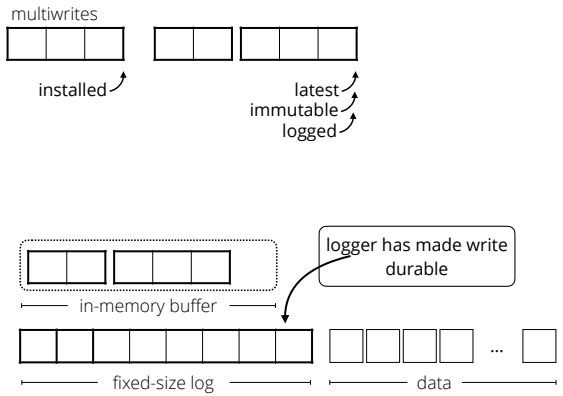
Pointers advance as write propagates



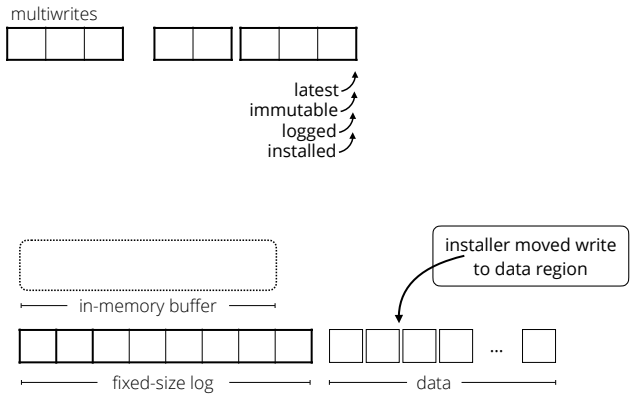
Write propagation: flush



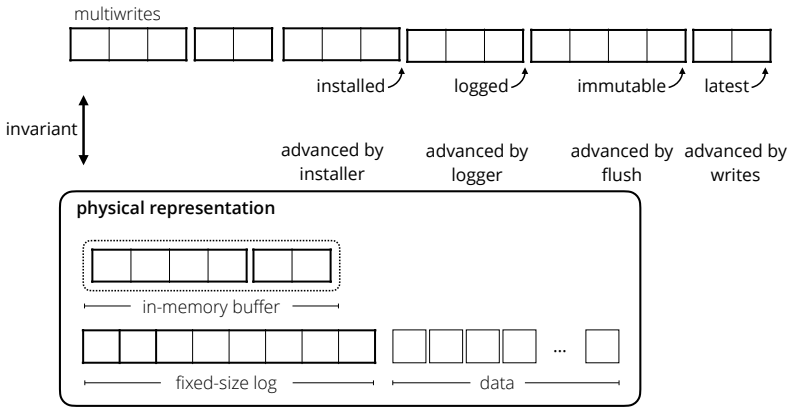
Write propagation: logging



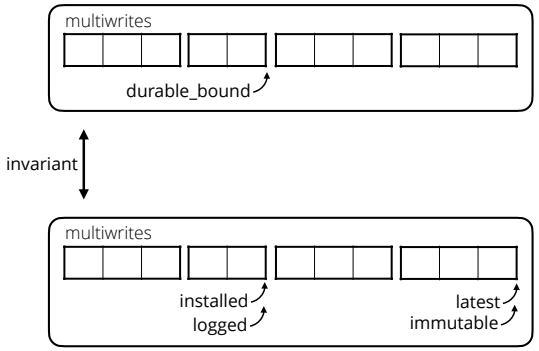
Write propagation: installation



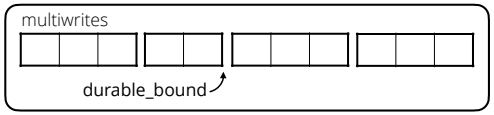
Pointers can all advance concurrently



Further abstraction hides concurrency



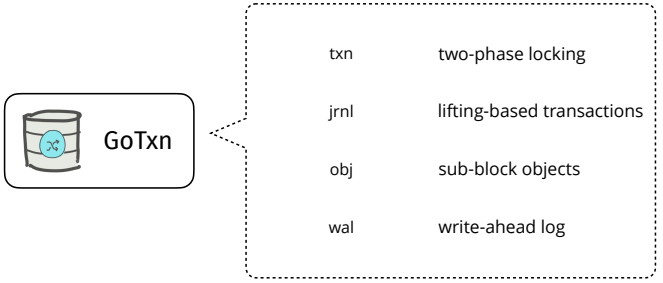
Further abstraction hides concurrency



`Flush()` advances durable_bound

On crash keep writes before durable_bound

Summary



DaisyNFS

DaisyNFS is a verified file system on top of GoTxn


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

NFS


```
func Begin() *Txn

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

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

transactions

Challenges

Specification: formalizing NFS

Proof: leveraging atomicity from GoTxn

Implementation: fitting operations into transactions

Specification: how to formalize NFS (RFC 1813)?

INFORMATIONAL

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

NFS Version 3 Protocol Specification

Status of this Memo

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

IESG Note

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

Abstract

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

Define abstract state for NFS

```
type Ino = uint64
type Path = seq<byte>
datatype Attrs = Attrs(mode: uint32, …)
datatype File =
  | ByteFile(data: seq<byte>, attrs: Attrs)
  | Directory(dir: map<Path, Ino>, attrs: Attrs)

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

Captures the state of the file system

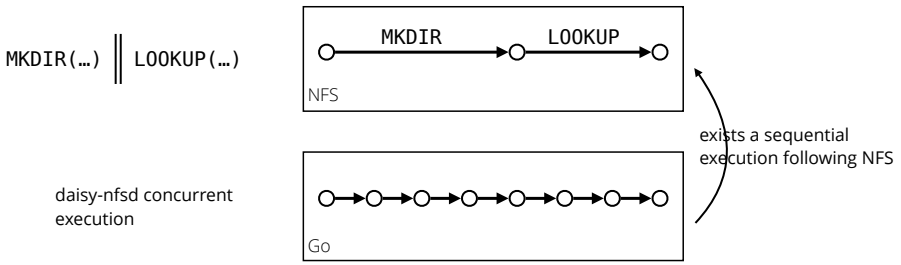
Mathematical model of each operation

```
predicate REMOVE_spec(fs: FilesysData, fs': FilesysData,
                      args: RemoveArgs)
{
  && args.d_ino in fs
  && fs[args.d_ino].Directory?
  && var d := fs[args.d_ino].dir;
     && args.name in d
     && var d' := delete(d, args.name);
        fs' == fs[args.d_ino := Directory(d', d.attrs)]
}
```

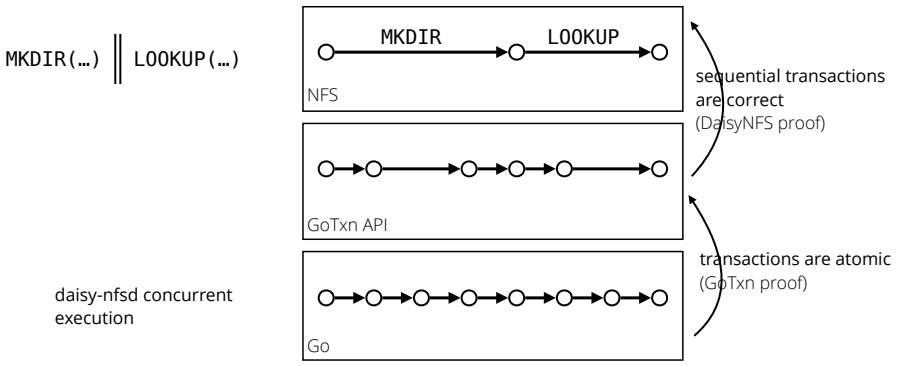
conditions needed for REMOVE to make sense

essence of REMOVE is this change to d_ino

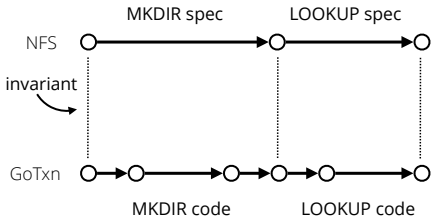
DaisyNFS's top-level specification



Proof: compose GoTxn and DaisyNFS proofs



Transactions are proven with sequential reasoning



Verify operations using Dafny

Dafny (verified)

```
method REMOVE(tx, args)
  returns (r: RemoveReply)
  requires Valid() ensures Valid()
  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)

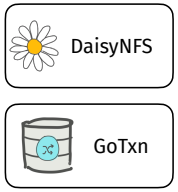
```
method REMOVE(tx, args)
  returns (r: RemoveReply)
  requires Valid() ensures Valid()
  ensures
    REMOVE_spec(old(fs), fs, args, r)

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

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

Go (unverified)

Logical no-op but physically recovers space

Summary



Specified a mathematical version of RFC 1813

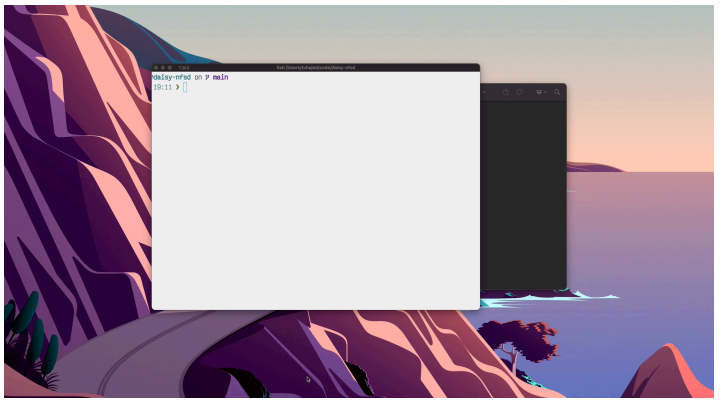
Proof strategy uses GoTxn atomicity

Implemented file system using transactions

Implementation

	Total lines
DaisyNFS	9,600 (Dafny)
GoTxn	36,000 (Perennial)
Perennial	20,000 (Coq)
Goose	3,500 (Go)

DaisyNFS is a real file system



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	Proof
DaisyNFS	3,200	6,400
GoTxn	1,600 (Go)	35,000 (Perennial)

2x proof:code

20x proof:code

Assumptions in the DaisyNFS proof

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

Trusted computing base (TCB)

Unverified code (e.g., network protocol)

Dafny specification of NFS

GoTxn spec in Dafny is faithful

Goose accurately models Go

Testing methodology

Unit tested Dafny support libraries

Compare Goose model to Go executions

Ran fsstress and fsx-linux test suites

Ran CrashMonkey for crash testing [OSDI '18]

Symbolic execution over NFS protocol

Bugs found in unverified code and spec

XDR decoder for strings can allocate 2^{32} bytes
File handle parser panics if wrong length
Panic on unexpected enum value
WRITE panics if not enough input bytes
Directory REMOVE panics in dynamic type cast

Unverified glue code

Didn't find bugs in verified parts

The names "." and ".." are allowed
RENAME can create circular directories
CREATE/MKDIR allow empty name
Proof assumes caller provides bounded inode
RENAME allows overwrite where spec does not

Missing from specification

Compare against Linux NFS server with ext4

Linux NFS client ⇄ (NFS) DaisyNFS

vs

Linux NFS server ⇄ local ext4

*using `data=journal`

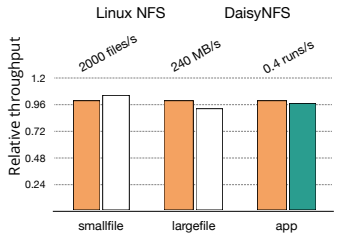
Performance evaluation setup

Hardware: i3.metal instance
36 cores at 2.3GHz, NVMe SSD

Benchmarks:

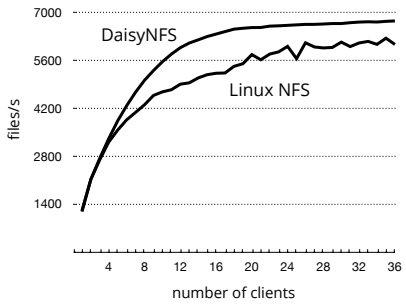
- smallfile: metadata heavy
- largefile: lots of data
- app: `git clone` + `make`

DaisyNFS gets good performance with a single client



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

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

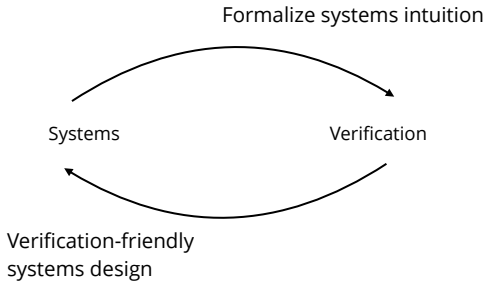
File systems

Security

Distributed systems

Software upgrades

My research approach



Verification is promising for secure systems

System needs to cover all inputs but attacker only needs one vulnerability

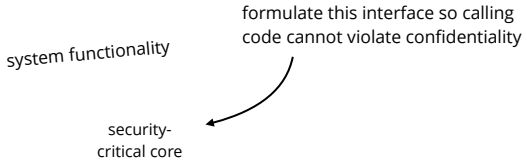
Some verified libraries exist but not systems

Goal: end-to-end security guarantees

Specifying application-level security

Prove confidentiality and integrity guarantees

Approach: specifications for confidentiality

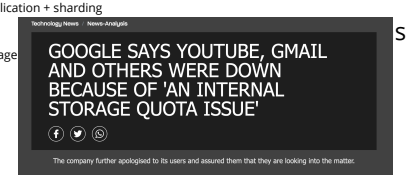


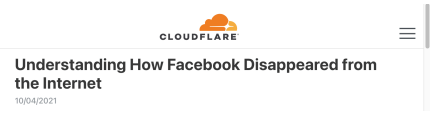
Distributed systems are complex and multi-tiered

ud services

replication + sharding

storage





Goal: prove a multi-service distributed system

Real systems are made from many smaller components

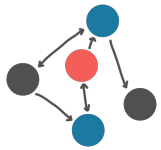
Proof burden should be manageable

Approach: alternate protocol proofs to RPC-style implementations

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

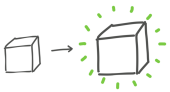
Implementation uses remote procedures

Specification should bridge the two



Protocol reasoning is over whole system

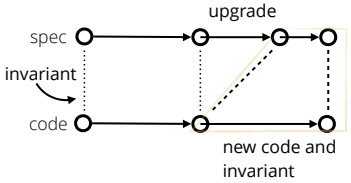
Verifying safety of software upgrades



Upgrading a verified system deserves proof

File systems, distributed systems, and applications have upgrades

Approach: new specifications and proof patterns for software upgrades

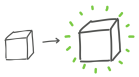


Verification beyond these domains

Security

Distributed systems

Software upgrades



Vision: reliable systems software with verification

Verified a concurrent, crash-safe file system

Expand the reach of verification in other domains

Perennial