



# Building a File System on the Celeste File Store

**Glenn Skinner**  
Senior Staff Engineer  
Sun Microsystems

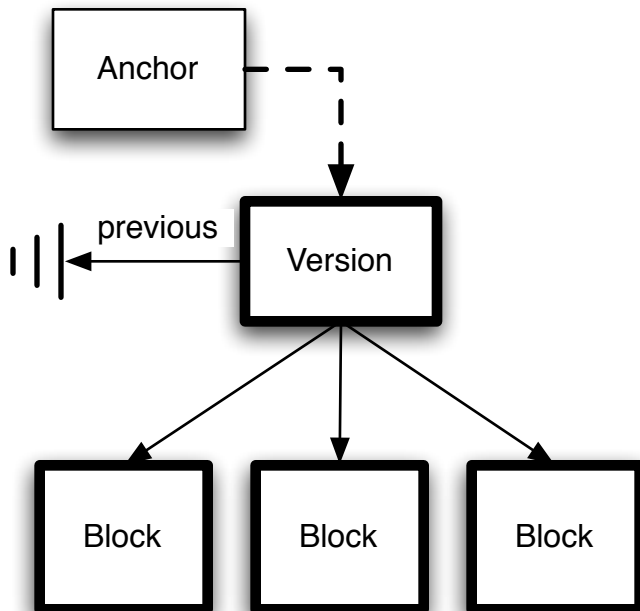


**2008  
Sun Labs  
Open House**



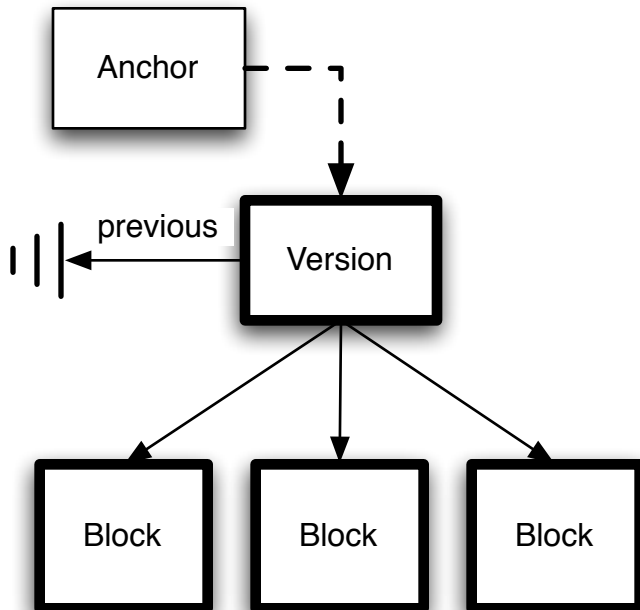
# Celeste Objects

- Normal case: content determines address
  - > Immutable, cacheable, and self-describing
- Updatable objects
  - > Content attested via byzantine agreement protocol



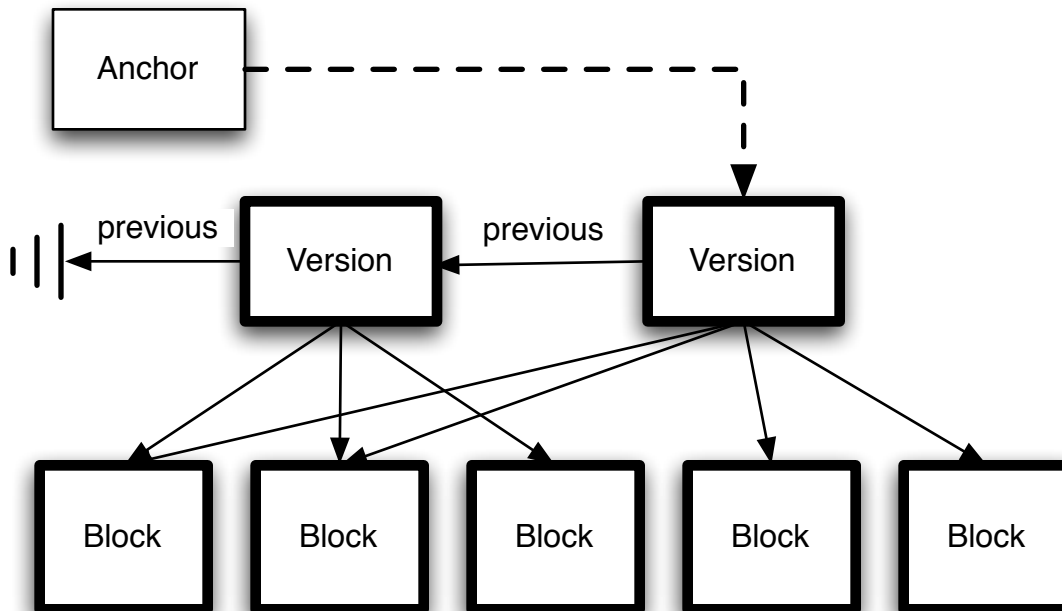
# File Updates

- The Anchor object changes value...



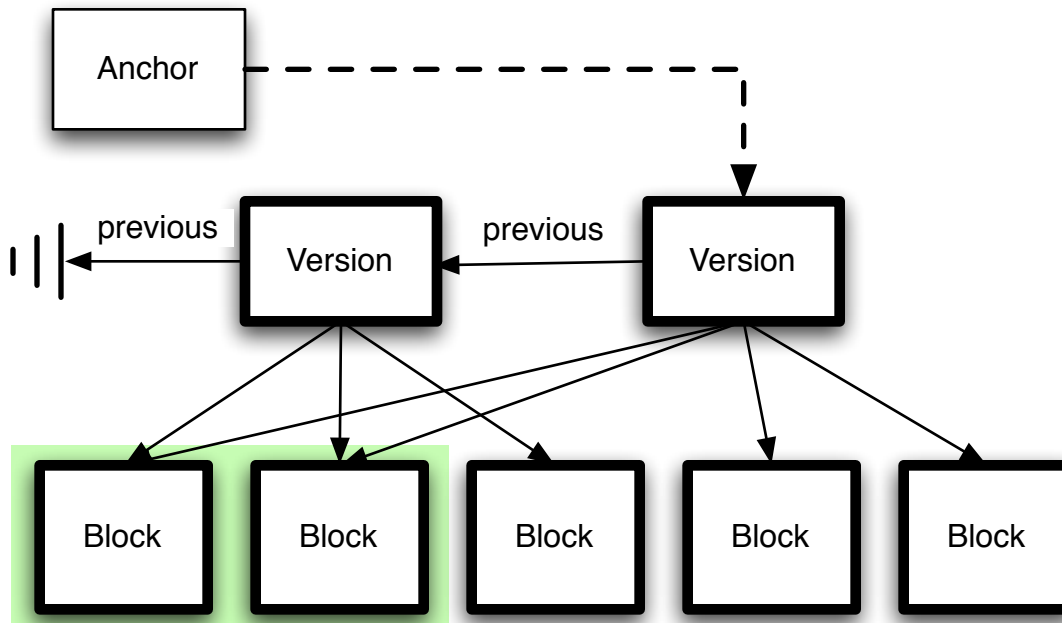
# File Updates

- The Anchor object changes value...
- ... producing a new version...



# File Updates

- The Anchor object changes value...
- ... producing a new version...
- ... that shares blocks with the old.



# Celeste Files

- Structured as collections of lower-level objects
  - > As depicted on previous slides.
  - > These objects are scattered across multiple hosts.
    - > Focus is on availability and robustness, not performance!
- Named by their anchor objects
  - > Therefore, each file has a 256-bit object id as its name.
- Accessed by a Celeste-specific API
  - > Uses Celeste-specific vocabulary: `DOLRObjectId`, `CelesteOperation`, `DOLRResponsibleParty.Signature`, ...
  - > Mapping to traditional file system APIs is non-obvious.
    - > Some parts are; I won't discuss those.

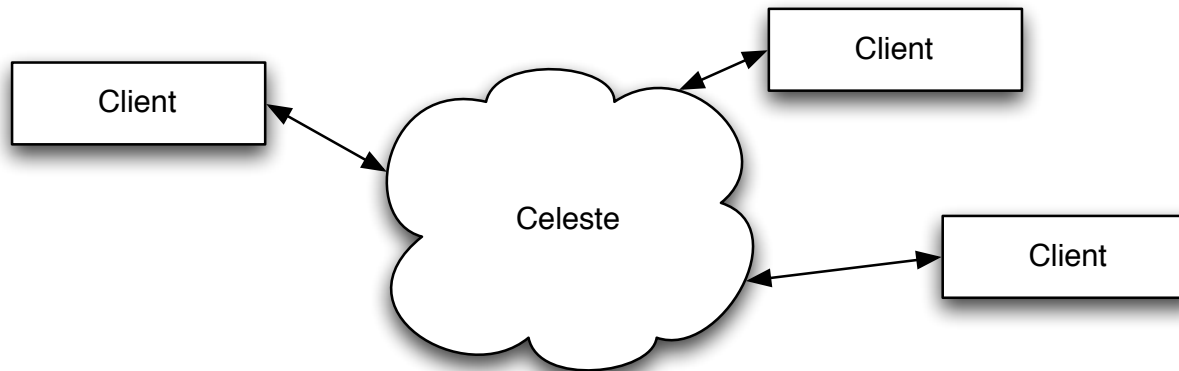
# Why a File System?

- Usability
  - > “4A3F6C8B9854D8291C60F7E93EAF7D5697D61F5C5ED3646039903324C4514D00” is a hopelessly bad file name.
  - > Don’t want to have to jump through hoops to use Celeste.
- Celeste test bed
  - > Has exposed numerous functional and performance bugs.
  - > E.g., extending truncates, unnecessary round trips, improper create after create behavior, ...
- A probe for deeper issues
  - > Is the Celeste foundation adequate for other uses we might wish to make of it?

# Implementing a File System

- Some things are straightforward.
  - > Translating from the Celeste API to a POSIX-like API.
  - > Tracking positions within files.
  - > Implementing directories.
  - > And a laundry list of other features.
- But threaded through these features are implementation challenges.
  - > The rest of the talk focuses on the nastiest of them (more or less in the order I tripped over them).

# Concurrency Management



- Cannot assume that code running at any given node is the sole accessor of underlying Celeste resources.
  - > Consider:

```
file.lock(); file.munge(); file.unlock();
```
  - > This sequence is broken.
    - > `file` refers to the client's local representation of the file.
    - > Access from other clients is not excluded.

# Concurrency Management: How to Fix?

- Lock is too far away from its file, so how about moving it closer?
  - > I.e., add locking primitives to Celeste interface.
  - > Would require a distributed locking protocol.
    - > To fit with Celeste, must be byzantine fault tolerant as well.
    - > Would also have to cope with failures in lock-holding clients.
- Other possibilities
  - > leases, fine-grained access control effectively amounting to locks
- Or, something we already had...

# Predicated Update

- Basic idea:
  - > An update operation must name the version it's based on.
  - > The update succeeds only if that version is still current.
- Operation outline:

```
do {  
    version = file.getCurrentVersion();  
    compose operation based on this version;  
    invoke the operation, predicating it on version;  
    if (success) return;  
} while (not ready to give up);
```
- In essence, a hand-built transaction.
- Can optimize out some round trips.

# Celeste Feature Wish List, Part I

- Transactions!
  - > Predicated update is acceptable for single object operations, but becomes difficult to use with multiple objects.
    - > A file rename can involve changing up to four objects; nearly impossible to avoid exposing intermediate state.
  - > Versioned objects provide a good foundation.
    - > Allow provisional object versions to be visible within a transaction, but don't make them official until it commits.
  - > Under investigation.
    - > May fall victim to the “and magic occurs here” problem.

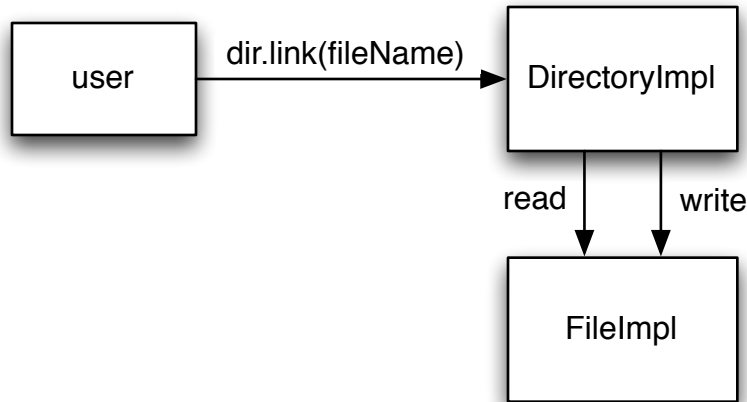
# Owner and Group Identities

- The system originally conflated file naming with file ownership.
  - > “Profiles” named users, but each one also acted as a name space encompassing all that user’s files.
  - > Thus, ownership was implicit in a file’s name.
- So how to support explicit owners and groups?
  - > First try: record them in file system-maintained file metadata.
    - > Ostensibly worked, but ran afoul of the bypass problem (more on that later).
  - > Current design: record them as Celeste-maintained file metadata.
    - > Implies adding Celeste operations to manipulate (and check) them.

# Access Control

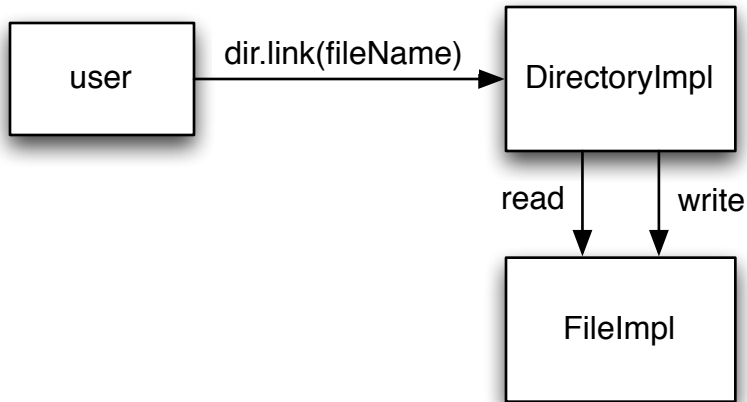
- Semantic mismatch: everything in Celeste is readable.
  - > So enforcing permissions that forbid reading is problematic.
- Celeste operations don't directly match traditional file system operations.
  - > When defining access control lists (ACLs), Celeste operations need one set of permissions and file system operations need another.
  - > The file system implementation must translate between these ACL types.
  - > Worse yet, operations at an upper level can decompose into multiple lower-level operations.

# Access Control Example



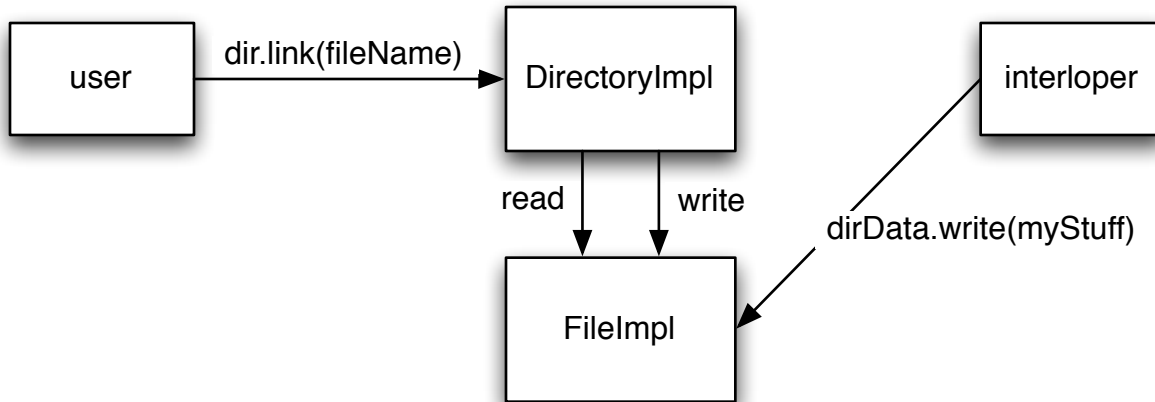
- A user invokes a link operation.
  - > user's credentials checked against an ACL requiring permission for the link.
  - > The directory implementation reads the directory contents file, modifies the contents, and writes them back.
  - > It must have credentials that pass ACL checks for read and write permission on its backing file.
    - > What credentials might those be?

# The Bypass Problem



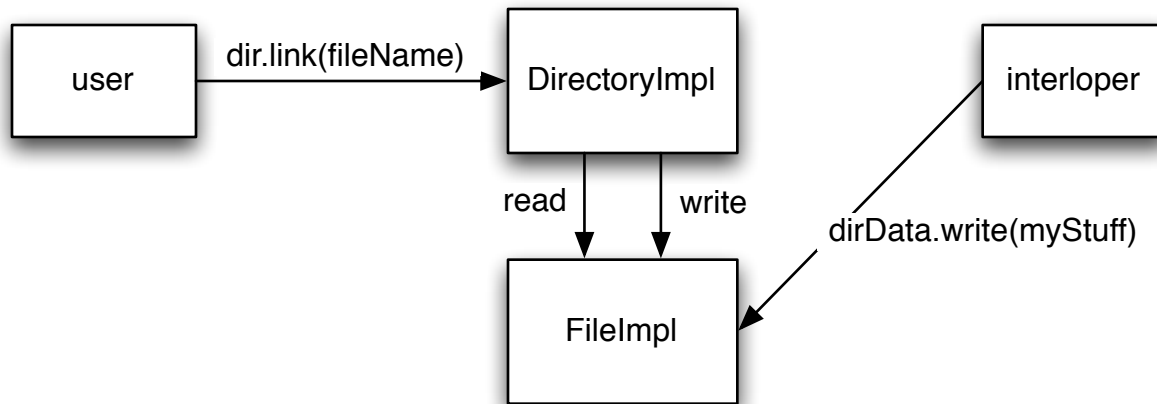
- More complications: Start with the previous example.

# The Bypass Problem



- More complications: Start with the previous example.
- What prevents interloper from subverting the directory's contents?

# The Bypass Problem



- More complications: Start with the previous example.
- What prevents interloper from subverting the directory's contents?
  - > A bad answer: security through obscurity.
  - > A better answer: An ACL on the directory contents file that allows access only to the directory implementation.
    - > But what identity should the ACL let through?

# Celeste Feature Wish List, Part II

- Autonomous objects!
  - > The directory implementation should have an identity (and life) of its own.
    - > It could then set an ACL on its directory contents objects that forbid access to others.
  - > To be determined:
    - > How are autonomous objects started?
    - > How do they gain access to their identity and acquire their credentials?

# Conclusions

- File stores, such as Celeste, need to provide more file system support than you might think.
  - > Have to bake owner, group, and ACLs into the file store if the file system is to support them properly.
  - > In fact, any feature requiring trustworthy behavior must be built in at or below the layer providing security.
- Thus, much of the file system is layered too high and ought to be pushed down into the Celeste infrastructure.
  - > Depends on extensions to that infrastructure.
- Work in process; we're still tinkering and doing the occasional wholesale re-implementation.

# Questions?

- Fire away!



**Glenn Skinner**  
[glenn.skinner@sun.com](mailto:glenn.skinner@sun.com)



**2008  
Sun Labs  
Open House**

