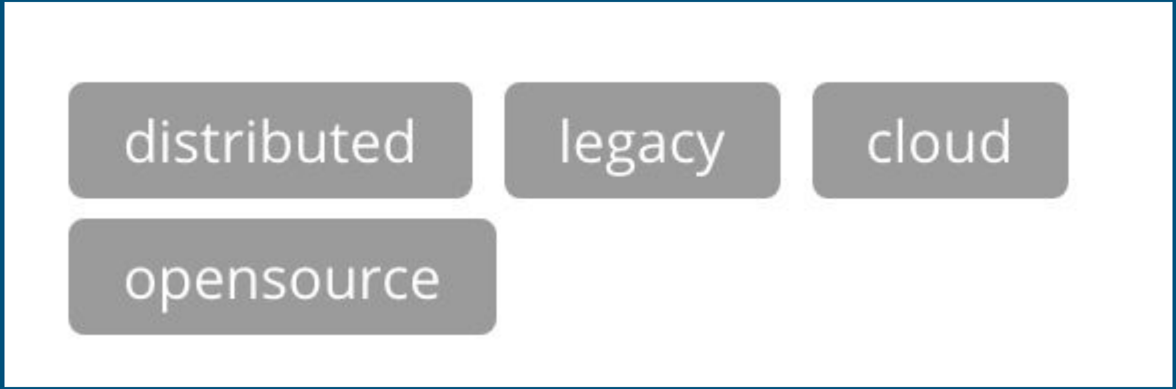
Designing Features for Mature Systems: Lessons Learned from Manta

Jordan Paige Hendricks
@itsajordansystm
GOTO Chicago
April 25, 2018

Legacy?



Manta 101!

Manta

Manta 101: Features

- Highly scalable, distributed object store
- HTTP-based
- Compute as a first class citizen!
  - Manta “jobs”
- Interface feels like a Unix filesystem

```
/jhendricks/stor/myDir/myObj.txt
```

Manta 101: API operations

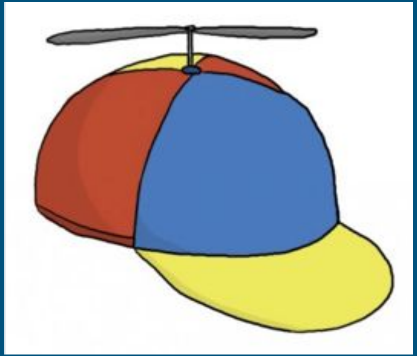
- `mput`: upload an object
- `mget`: download an object
- `mrm`: remove an object
- `mmkdir`: create a directory
- `mrmdir`: remove a directory
- `mls`: list a directory
- `mln`: make a ‘snaplink’ to an object
- `mlogin`: login to a container with your object

Manta 101: System Primitives

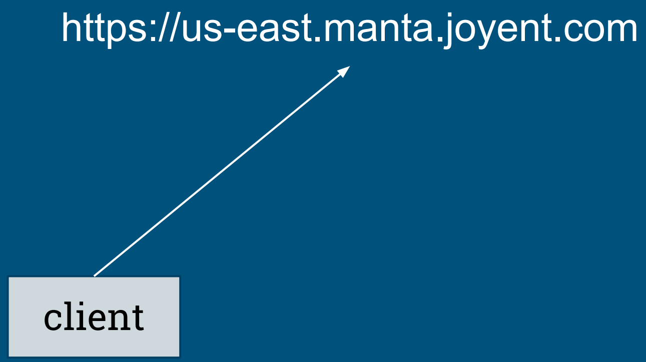
- **Objects**
  - Stored as flat files
  - Immutable
- **Directories**
  - Are listable!

Manta 101: Design Constraints

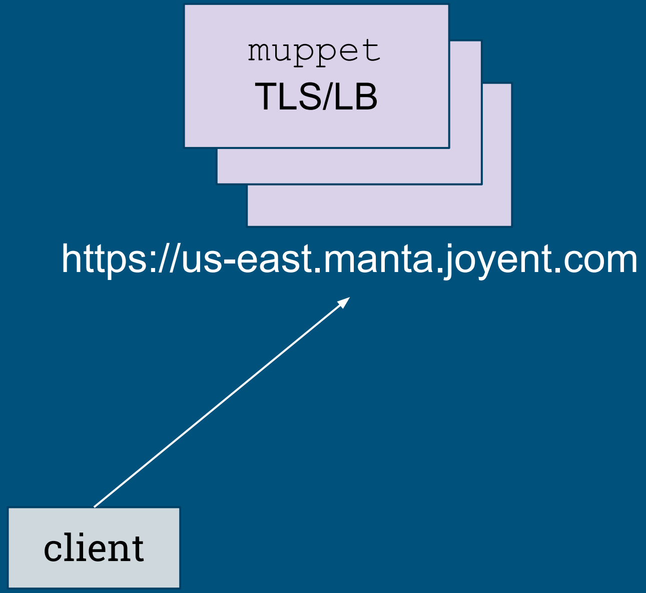
- CP system
  - When faced with a network partition, Manta chooses consistency over availability.
- Horizontally scalable where possible



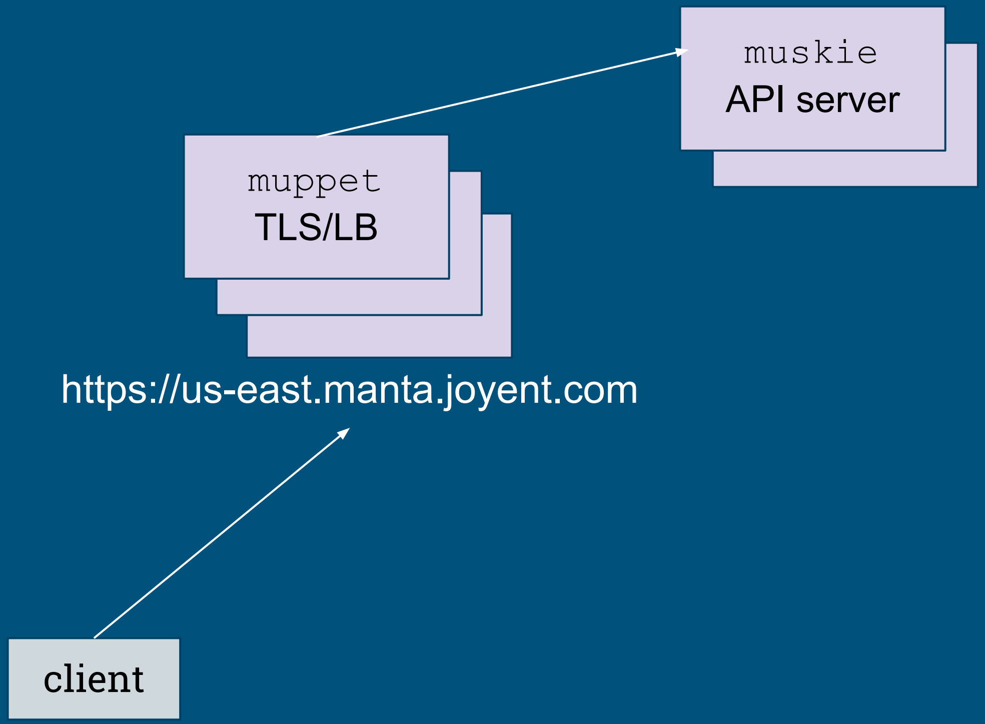
GET /jhendricks/stor/foo.txt



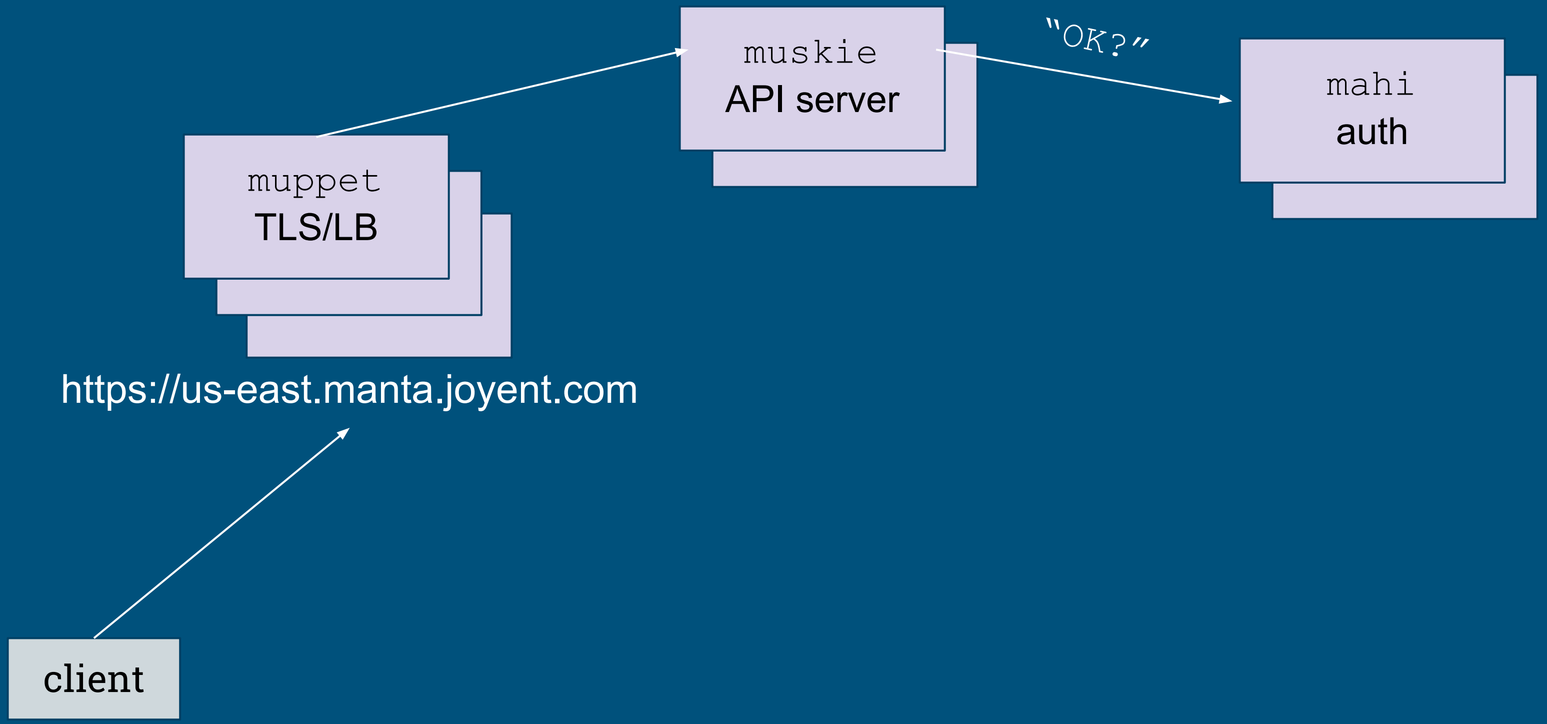
GET /jhendricks/stor/foo.txt



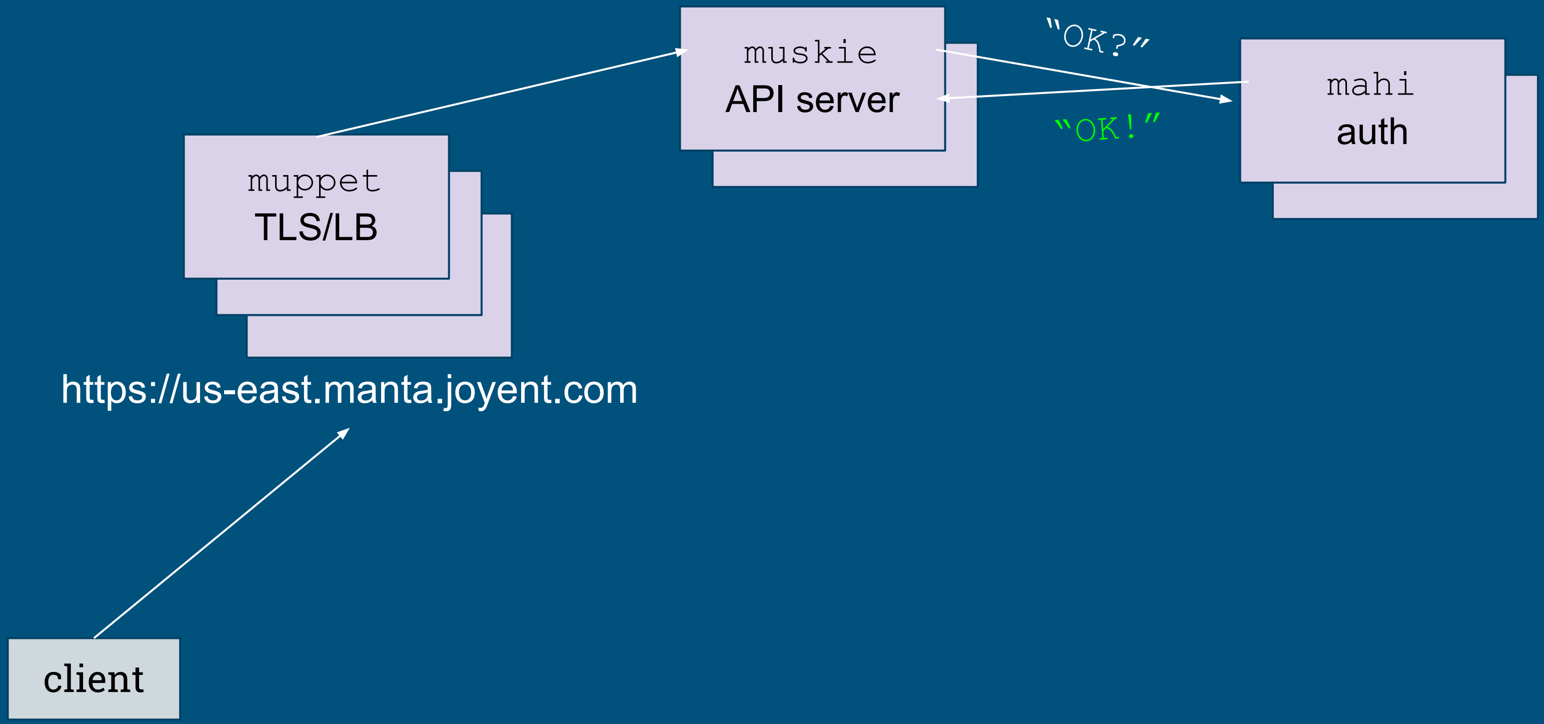
GET /jhendricks/stor/foo.txt



GET /jhendricks/stor/foo.txt



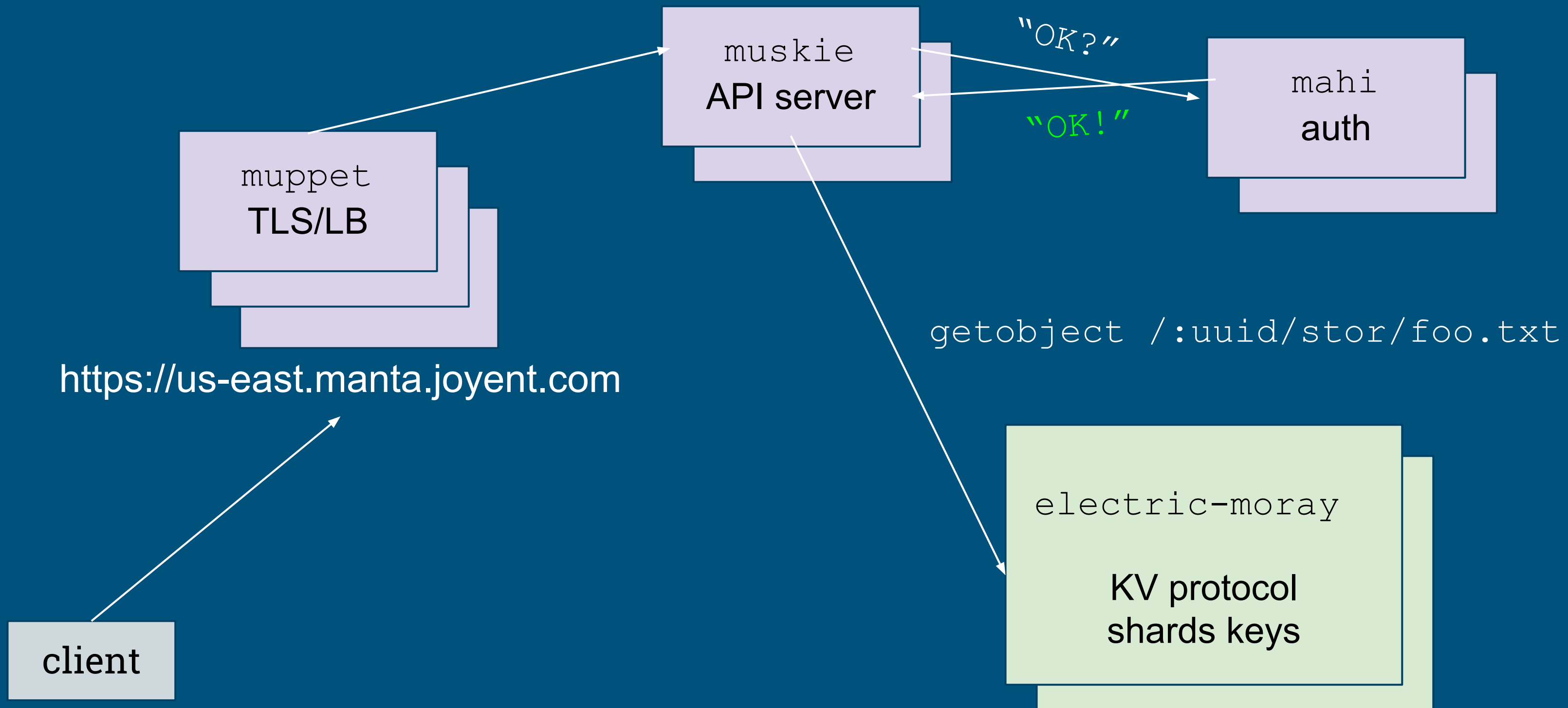
GET /jhendricks/stor/foo.txt



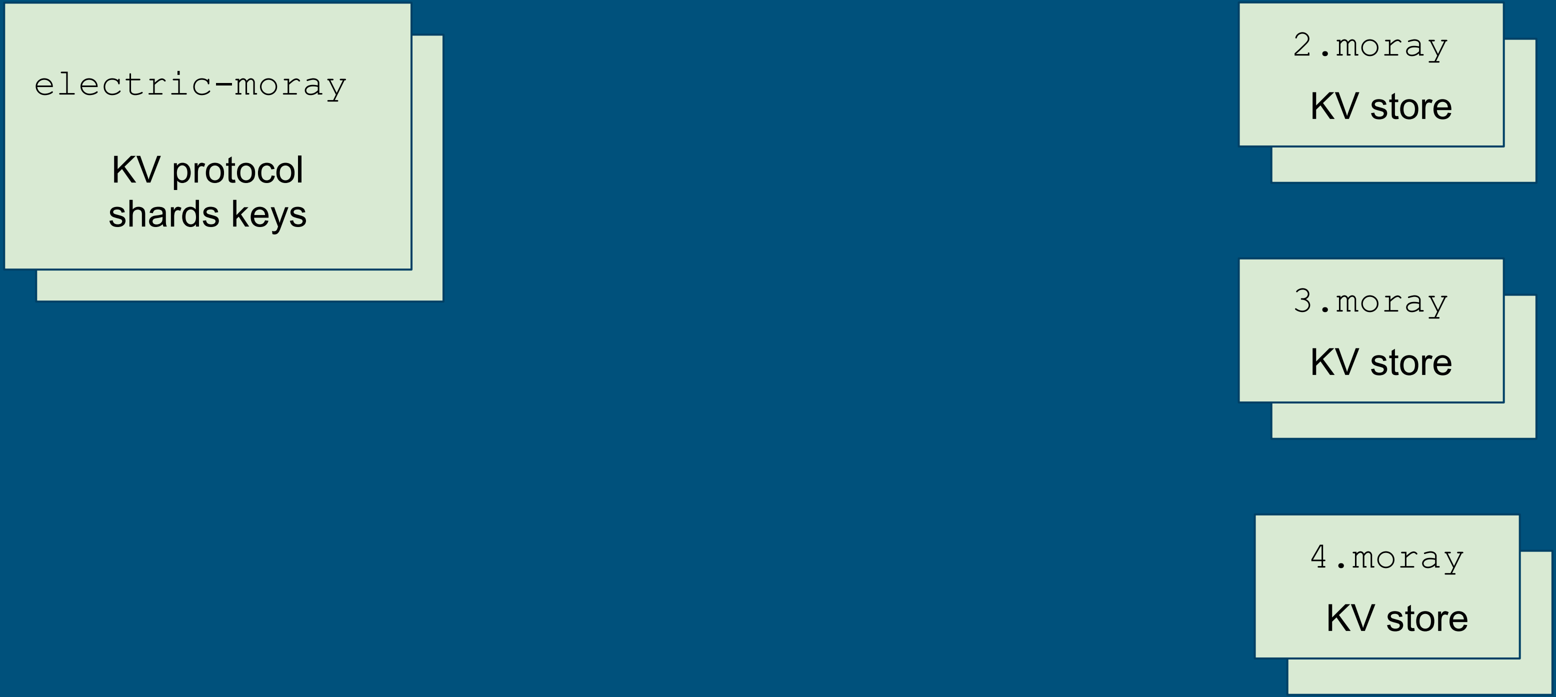
Manta Design: Data Path

- Architectural separation of *metadata* and *storage*
- Metadata tier responsible for information about the objects: its name, size, content MD5, who owns it, permissions, where it is stored, etc.
- Storage tier responsible for storing the object
  - Compute jobs also run directly on data on storage CNs

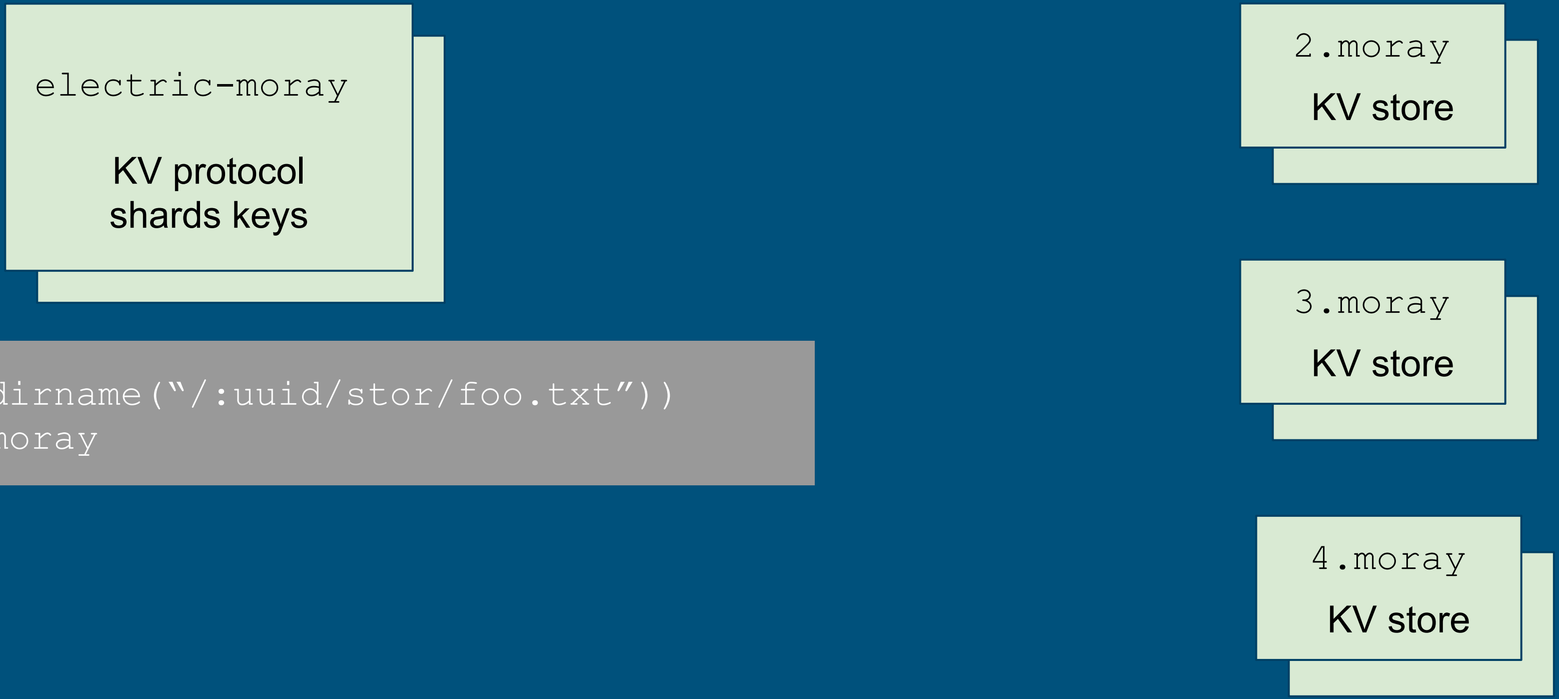
GET /jhendricks/stor/foo.txt



GET /jhendricks/stor/foo.txt

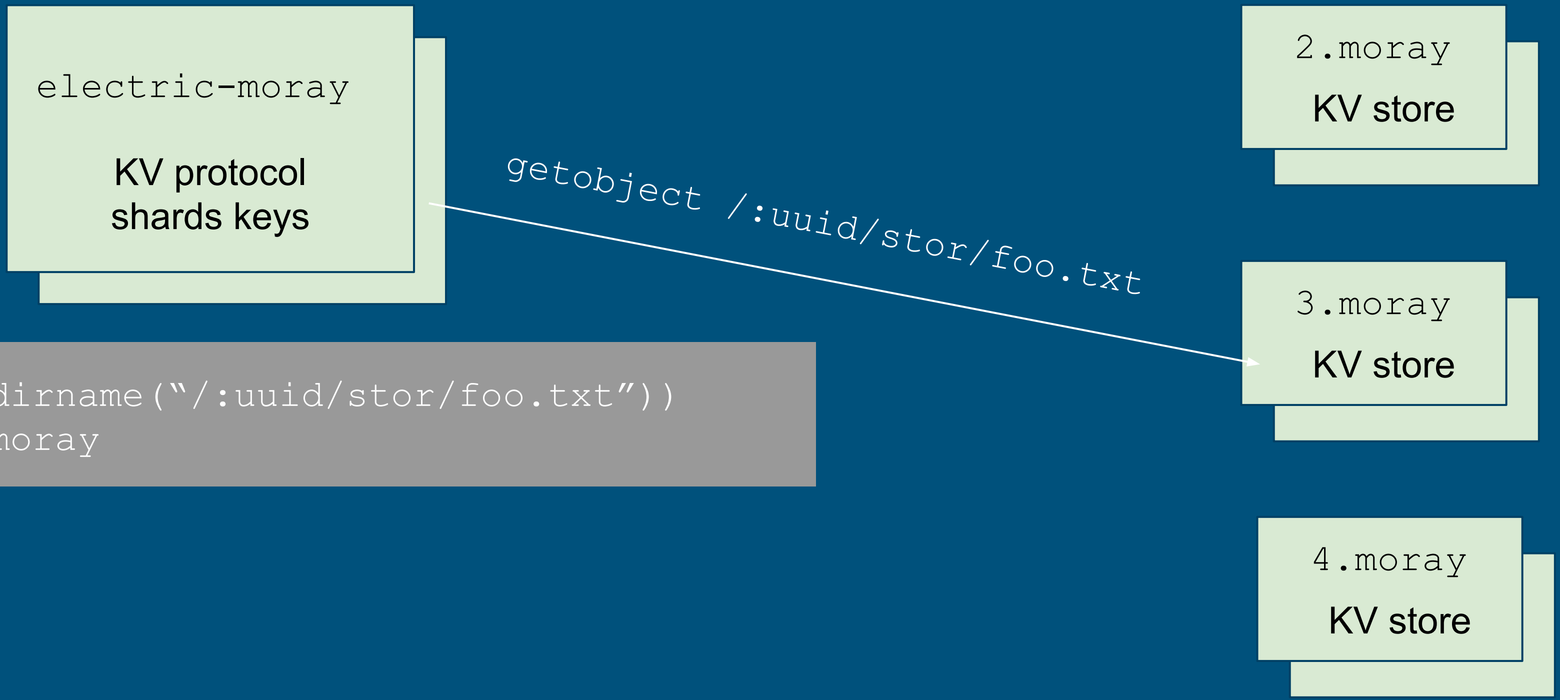


GET /jhendricks/stor/foo.txt



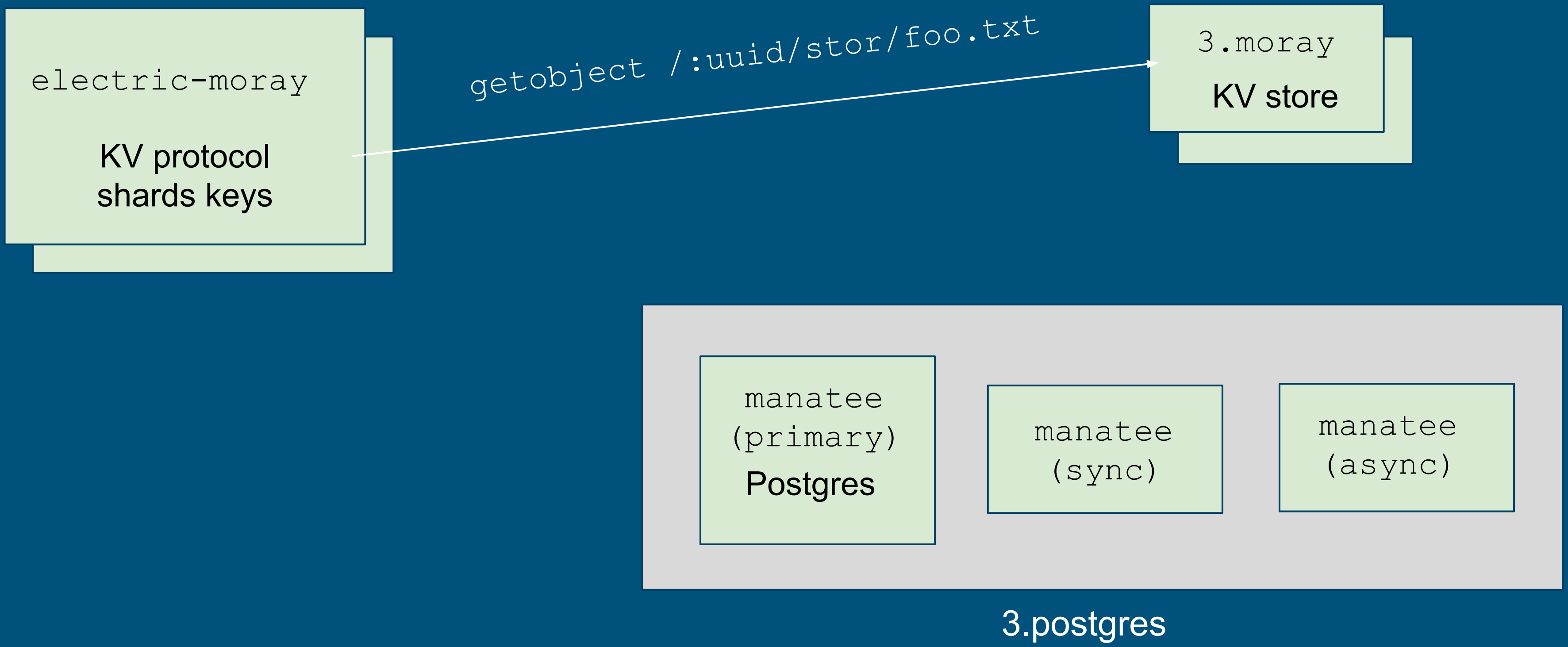
```
hash(dirname("/:uuid/stor/foo.txt"))
=> 3.moray
```

GET /jhendricks/stor/foo.txt

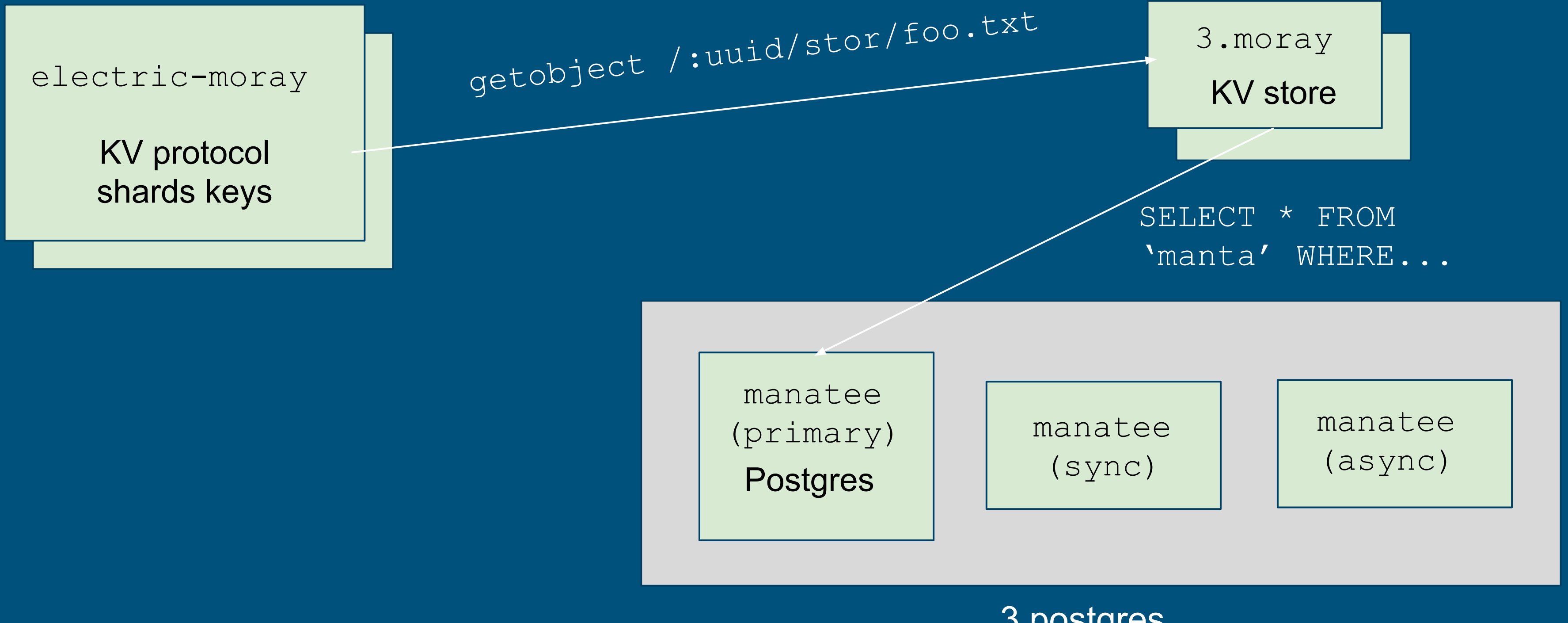


```
hash(dirname("/:uuid/stor/foo.txt"))
=> 3.moray
```

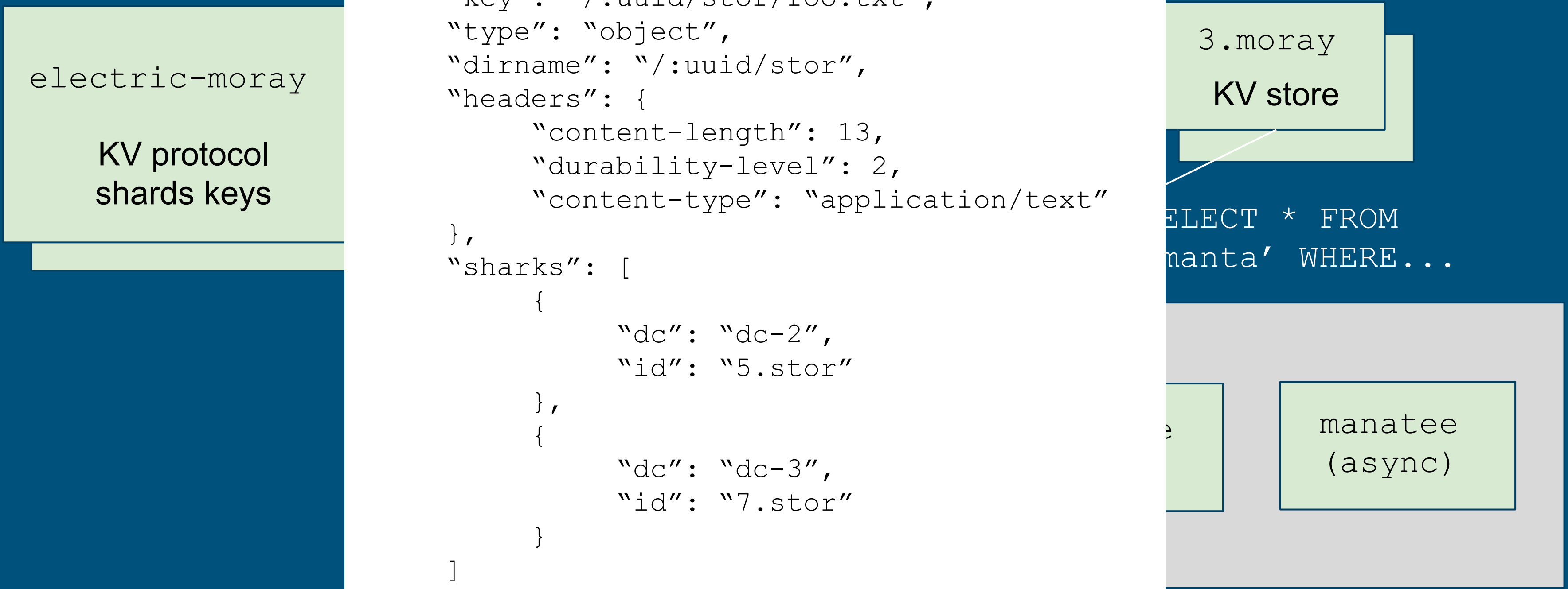
GET /jhendricks/stor/foo.txt



GET /jhendricks/stor/foo.txt

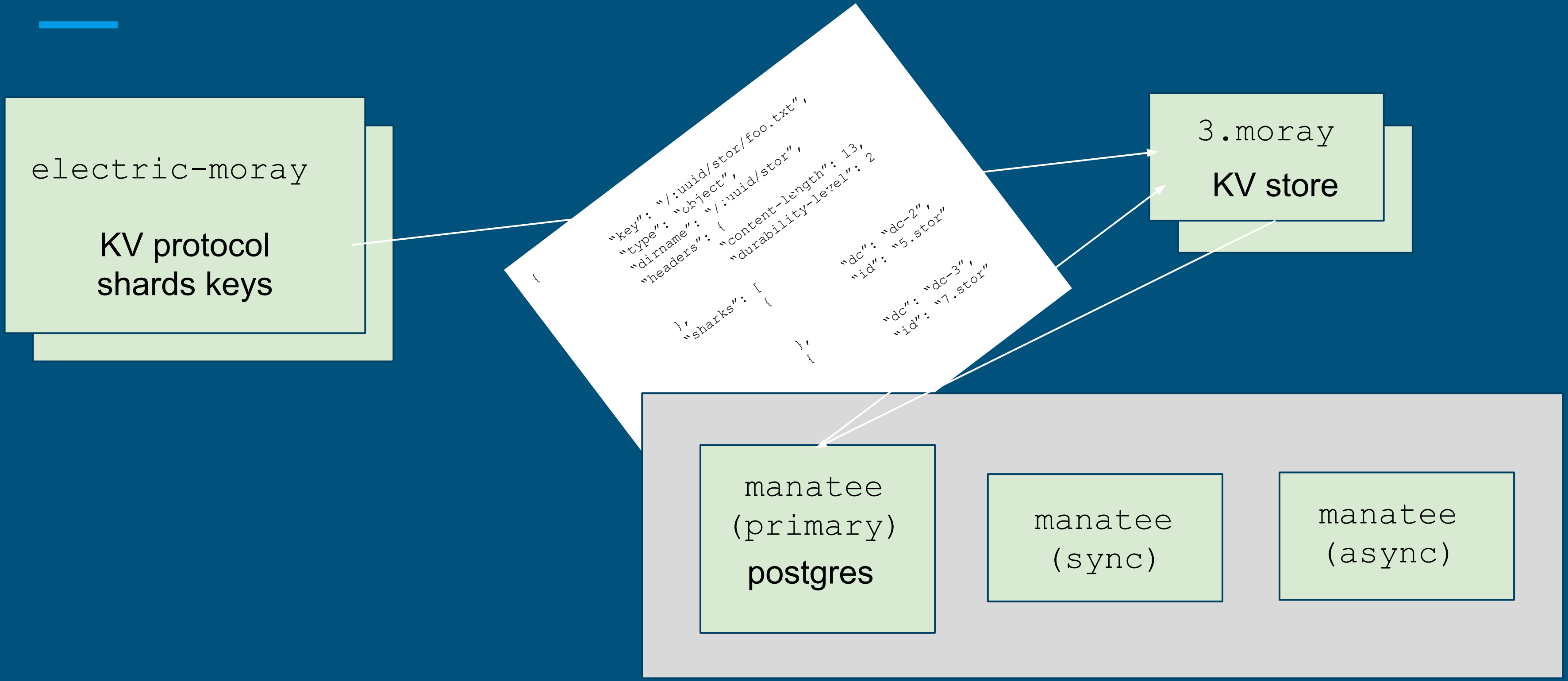


GET /jhendricks/stor/foo.txt



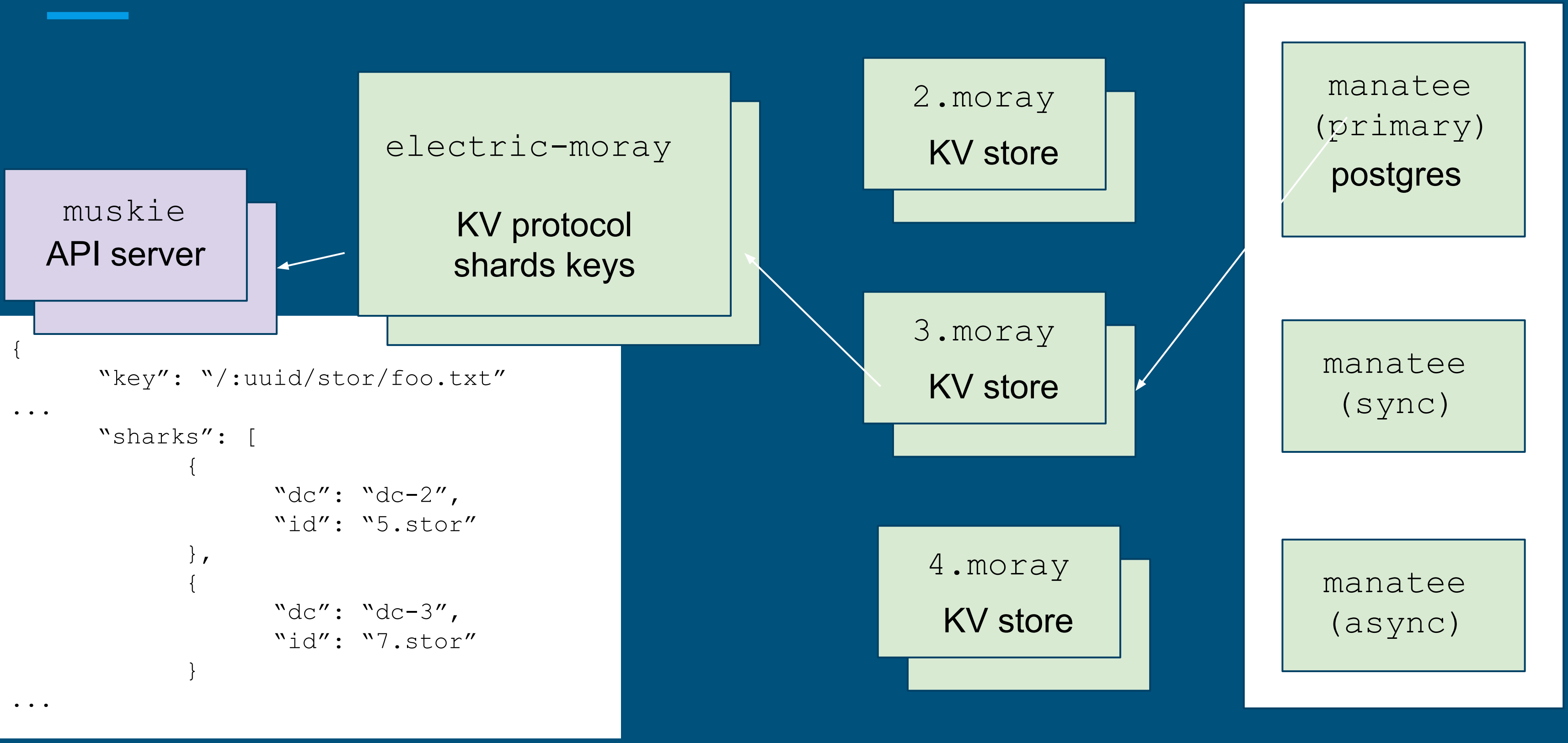
```
{
        "key": "/:uuid/stor/foo.txt",
        "type": "object",
        "dirname": "/:uuid/stor",
        "headers": {
                "content-length": 13,
                "durability-level": 2,
                "content-type": "application/text"
        },
        "sharks": [
                {
                        "dc": "dc-2",
                        "id": "5.stor"
                },
                {
                        "dc": "dc-3",
                        "id": "7.stor"
                }
        ]
}
```

GET /jhendricks/stor/foo.txt



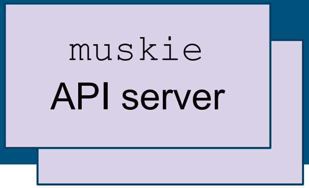
```
{
    "key": "/:uuid/stor/foo.txt",
    "type": "object",
    "dirname": "/:uuid/stor",
    "headers": {
        "content-length": 13,
        "durability-level": 2
    },
    "sharks": [
        {
            "dc": "dc-2",
            "id": "5.stor"
        },
        {
            "dc": "dc-3",
            "id": "7.stor"
```

GET /jhendricks/stor/foo.txt

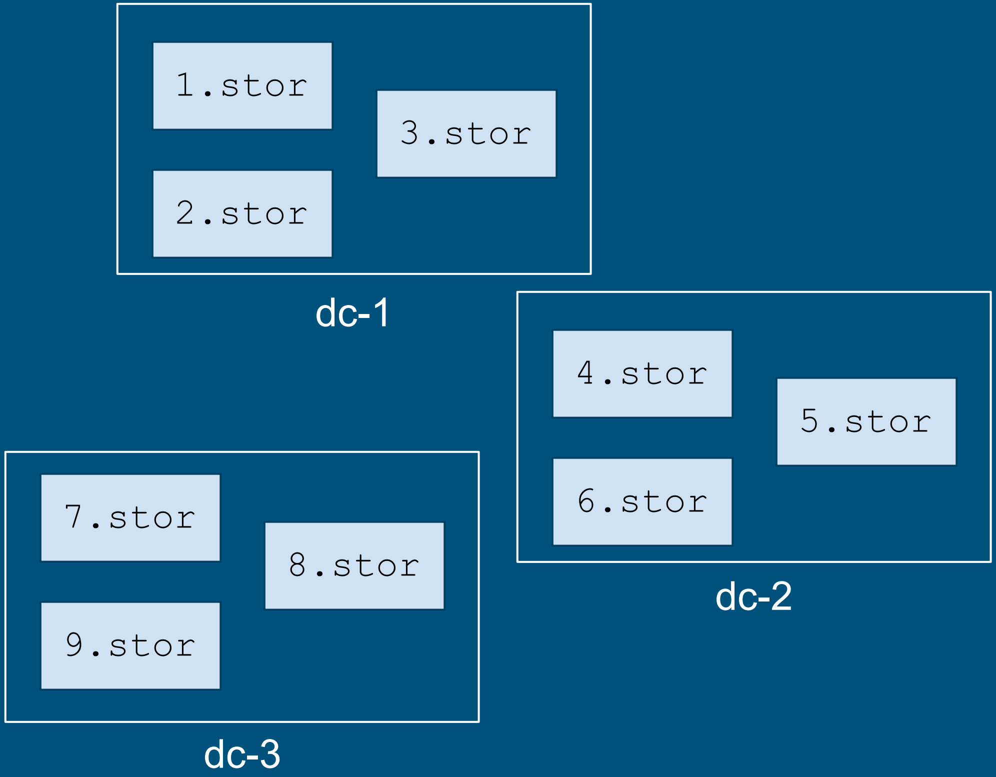


```
{
        "key": "/:uuid/stor/foo.txt"
...
        "sharks": [
                {
                        "dc": "dc-2",
                        "id": "5.stor"
                },
                {
                        "dc": "dc-3",
                        "id": "7.stor"
                }
...
```

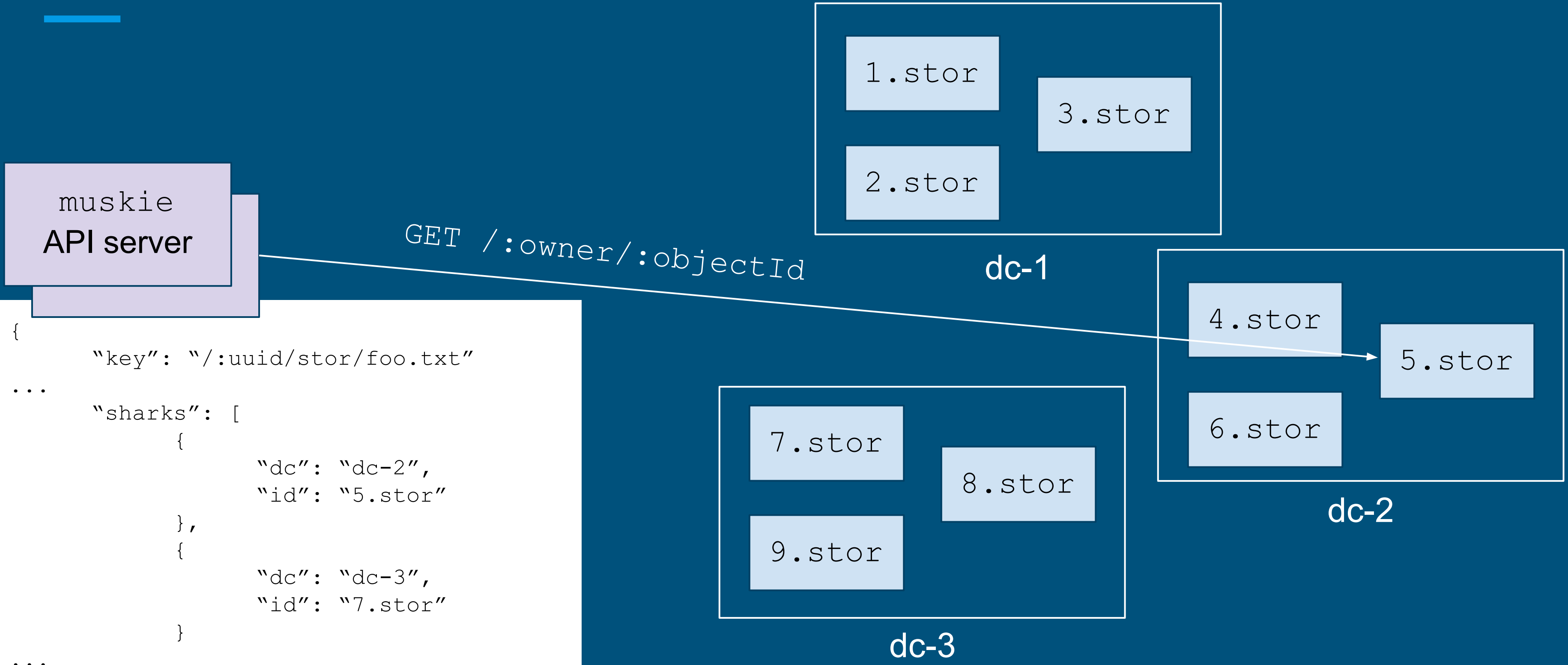
GET /jhendricks/stor/foo.txt



```
{
        "key": "/:uuid/stor/foo.txt"
...
        "sharks": [
                {
                        "dc": "dc-2",
                        "id": "5.stor"
                },
                {
                        "dc": "dc-3",
                        "id": "7.stor"
                }
...
```

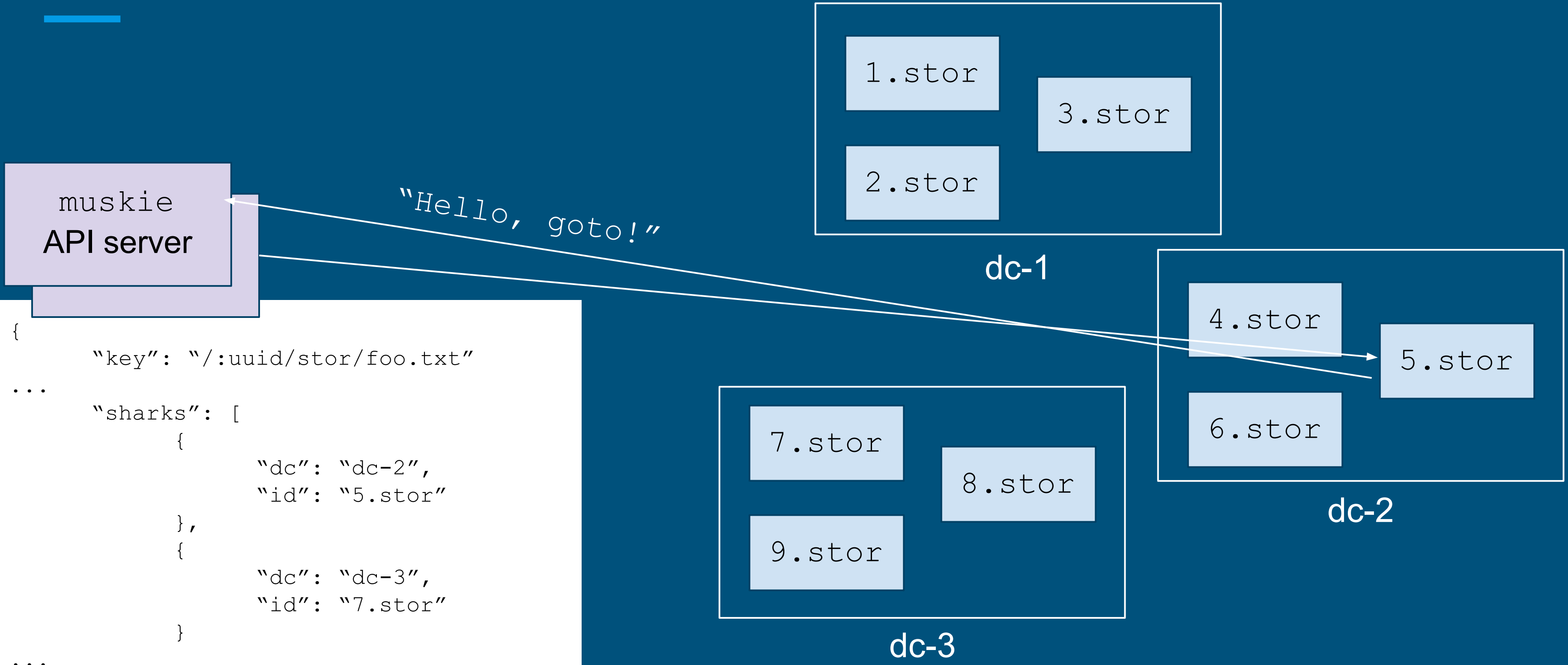


GET /jhendricks/stor/foo.txt



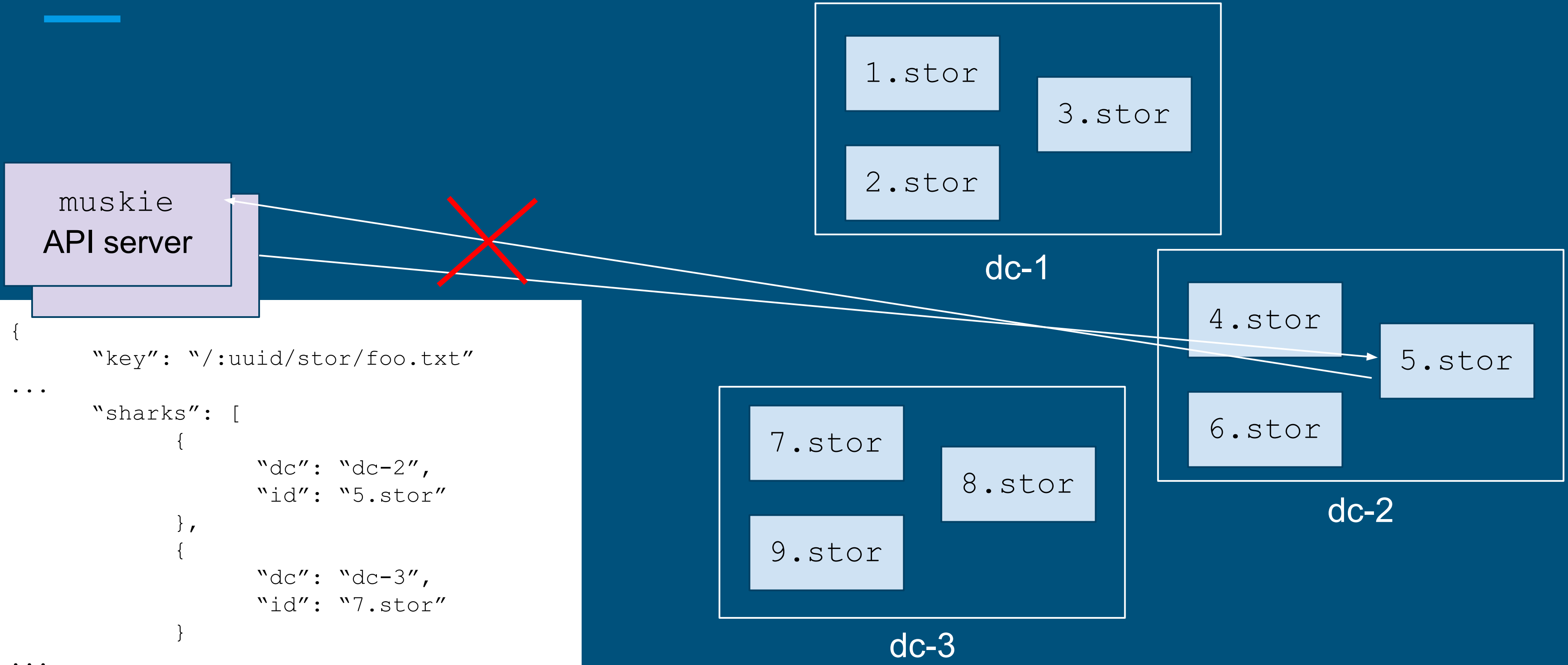
```
{
        "key": "/:uuid/stor/foo.txt"
...
        "sharks": [
                {
                        "dc": "dc-2",
                        "id": "5.stor"
                },
                {
                        "dc": "dc-3",
                        "id": "7.stor"
                }
...
```

GET /jhendricks/stor/foo.txt



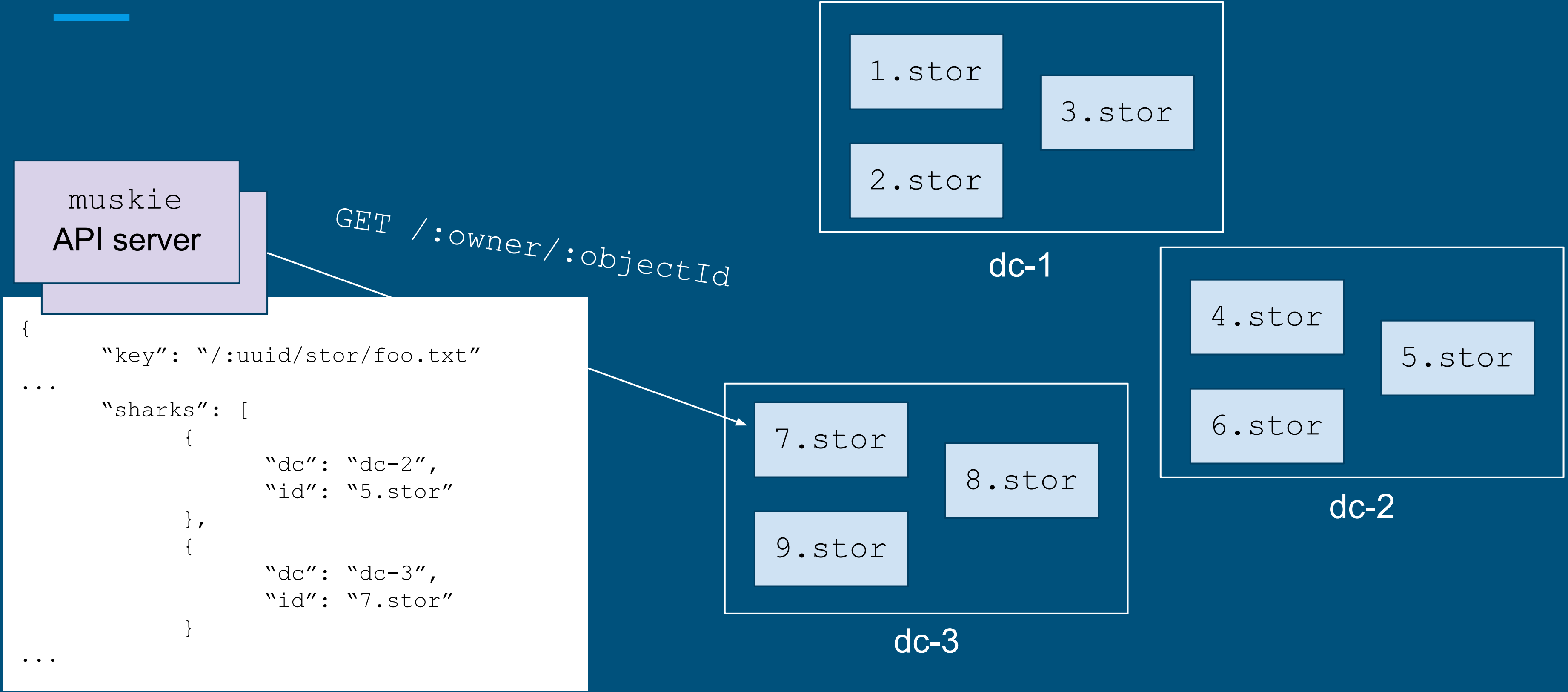
```
{
        "key": "/:uuid/stor/foo.txt"
...
        "sharks": [
                {
                        "dc": "dc-2",
                        "id": "5.stor"
                },
                {
                        "dc": "dc-3",
                        "id": "7.stor"
                }
...
```

GET /jhendricks/stor/foo.txt



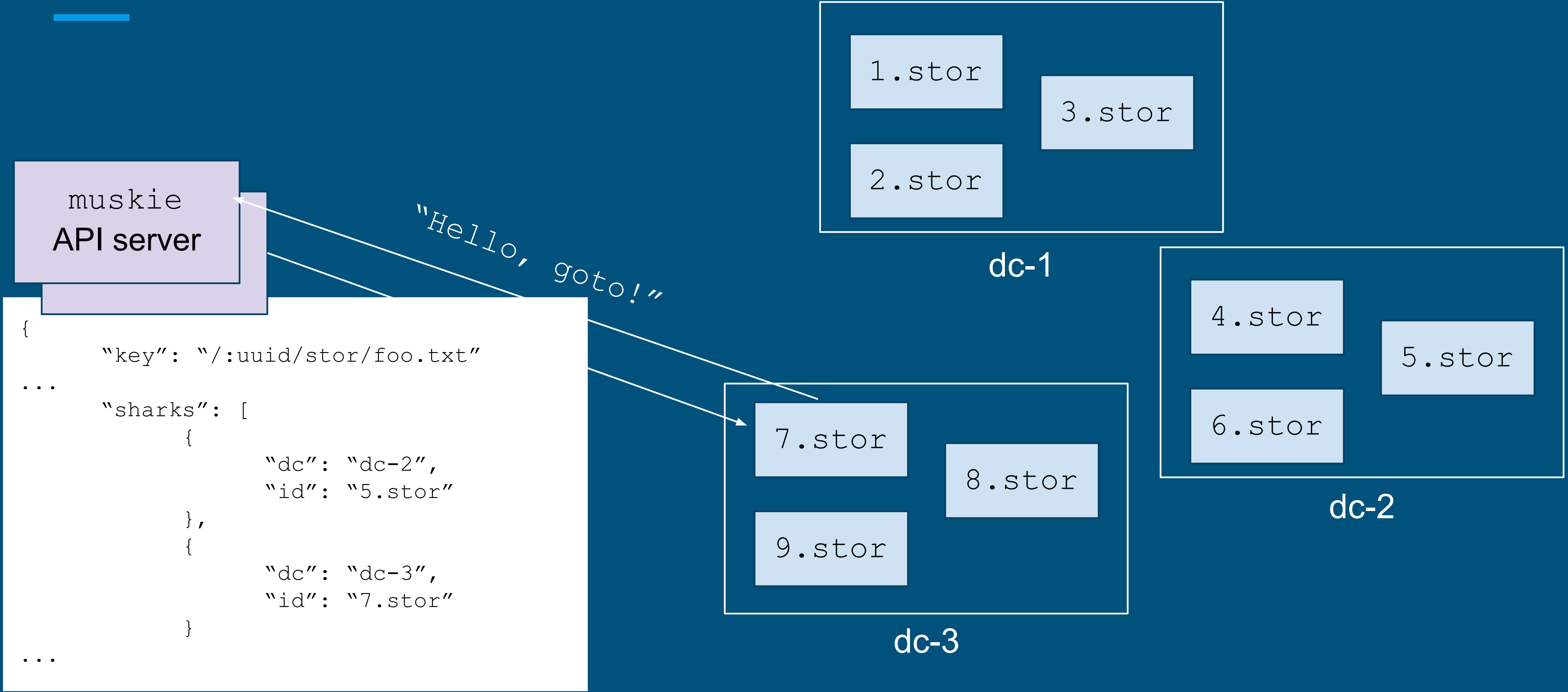
```
{
        "key": "/:uuid/stor/foo.txt"
...
        "sharks": [
                {
                        "dc": "dc-2",
                        "id": "5.stor"
                },
                {
                        "dc": "dc-3",
                        "id": "7.stor"
                }
...
```

GET /jhendricks/stor/foo.txt



```
{
        “key”: “/:uuid/stor/foo.txt”
...
        “sharks”: [
                {
                        “dc”: “dc-2”,
                        “id”: “5.stor”
                },
                {
                        “dc”: “dc-3”,
                        “id”: “7.stor”
                }
...
```

GET /jhendricks/stor/foo.txt



```
{
        "key": "/:uuid/stor/foo.txt"
...
        "sharks": [
                {
                        "dc": "dc-2",
                        "id": "5.stor"
                },
                {
                        "dc": "dc-3",
                        "id": "7.stor"
                }
...
```

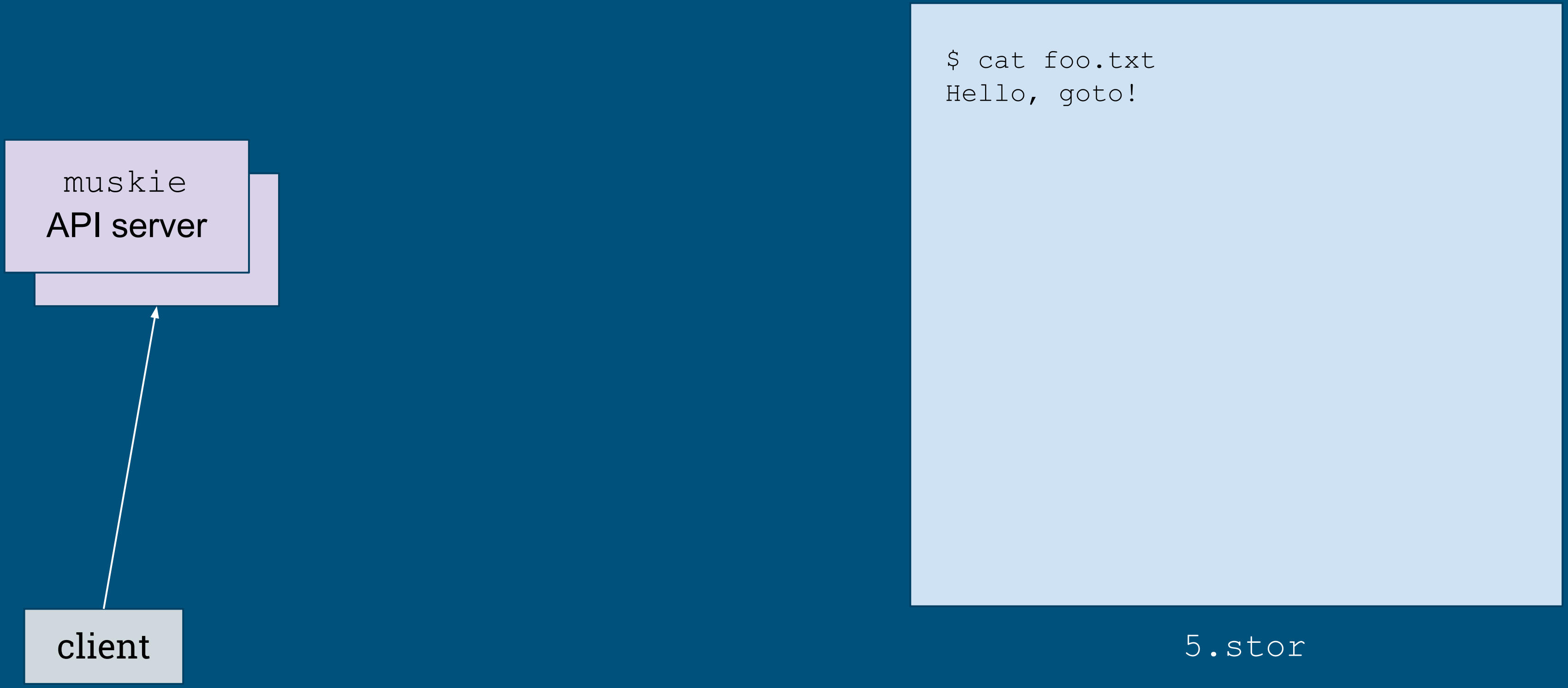
PUT /jhendricks/stor/newObj.txt

- Authorization (mahi)
- Muskie picks sharks to store the object on (spread across at least 2 DCs)
- Muskie streams the data to the sharks
- Muskie updates electric-moray with the new metadata record

Manta Design: Compute

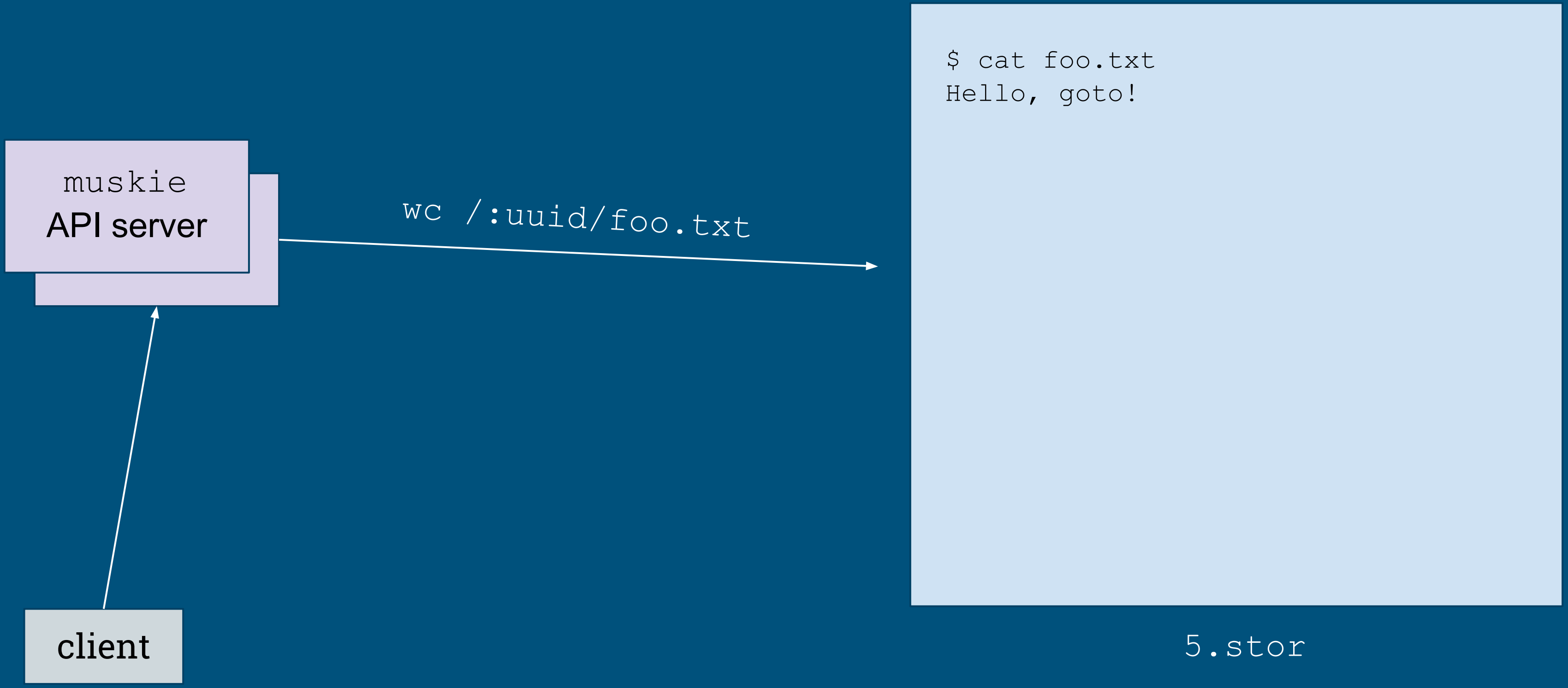
- Objects can be *large*
- Want to avoid copying data between servers
- To make objects possible to compute on with existing software, Manta's abstraction for an object is a flat file on its storage node.
    - Can run programs directly on these files
- Instead of copying data to run jobs on, move jobs to where the data lives!

$ wc /jhendricks/stor/foo.txt



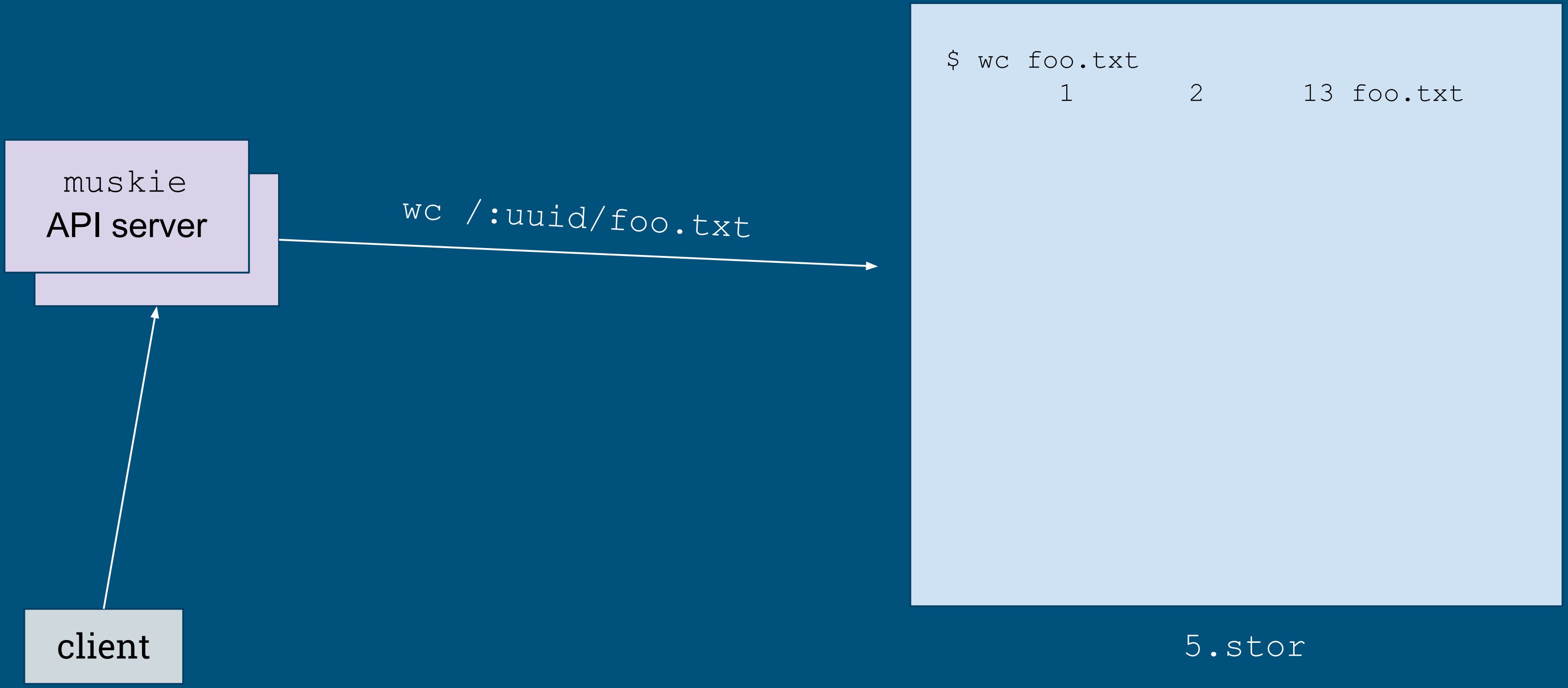
```
$ cat foo.txt
Hello, goto!
```

$ wc /jhendricks/stor/foo.txt



```
$ cat foo.txt
Hello, goto!
```

$ wc /jhendricks/stor/foo.txt



```
$ wc foo.txt
       1       2      13 foo.txt
```

Multipart Uploads!

Multipart Uploads: First Principles

- Upload an object in “parts”
- “Commit” the object when all parts are uploaded
    - Creates a new object, **indistinguishable from normal objects** created via PUT path
- Use cases?
    - Large files
    - Intermittent network connections
    - Streaming data from a source

Multipart Uploads: Design Questions

- How to identify multipart uploads?
- Where to store parts?
- What does a “commit” of an MPU look like?

Multipart Uploads: Design Considerations

- How can we ensure parts are listed easily?
- How can we list all multipart uploads?
- What happens if multiple clients operate on the same MPU?
- What happens during failures of Manta components?
- How will parts be cleaned up after commit?
- ...Can I cancel MPUs, too?

Multipart Upload Design

- How to identify MPUs?
  - UUID per MPU
- How to store parts?
  - Obvious answer: Use Manta objects!
  - Store parts as objects in a Manta directory
  - One directory per MPU: max of 10,000 parts << number allowed dirents (1 million)
  - Allows clients to list parts easily

Multipart Upload Design

- How to store part directories?
  - Under a new top-level directory, `/:account/uploads` (analogous to `/:account/jobs`)
  - But: don't want to limit the number of ongoing MPUs to the number of allowed dirents
  - Solution: one-level nested "prefix" directories, in which all uploads starting with the same characters have the same parent
  - Allows all MPUs to be listed in as many requests as it takes to list all prefix directories

Example MPU Structure

- Upload ID: `eaff0760-9b17-4fb7-b7c4-f2de818681f3`
- Parts directory

```
/jordan/uploads/eaf/eaff0760-9b17-4fb7-b7c4-f2de818681f3
```

- Example parts

```
/jordan/uploads/eaf/eaff0760-9b17-4fb7-b7c4-f2de818681f3/0
/jordan/uploads/eaf/eaff0760-9b17-4fb7-b7c4-f2de818681f3/1
/jordan/uploads/eaf/eaff0760-9b17-4fb7-b7c4-f2de818681f3/2
```

Multipart Upload Design: Commits

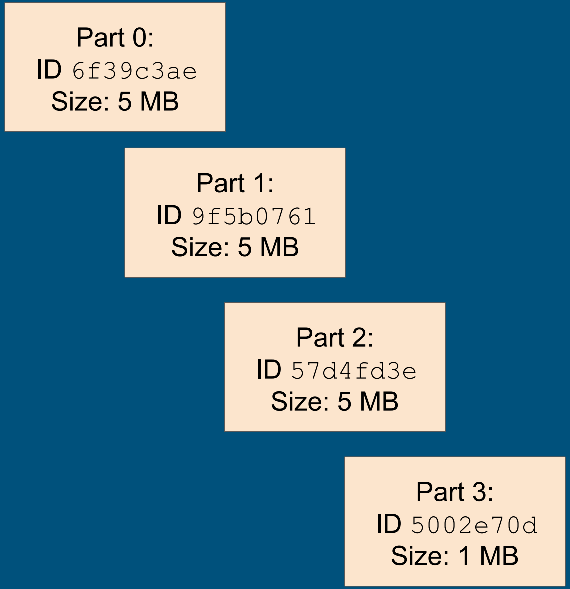
- Design constraints for `mpu-commit` operation?
  - Must be idempotent
  - Must be atomic
- What steps need to happen in Manta architecture?
  - Metadata layer: Insert an object record for the target object
  - Storage layer: Create the object on disk from its parts, on the appropriate storage nodes

Commit: Storage Layer Implementation

- **Constraints**
  - Cannot append or mutate parts on an existing object
  - Would like to avoid copying data over network
- Design
  - *Co-location* of parts on the storage nodes the final target object will live on
  - Create object on disk from parts *locally*
  - New operation on mako (storage node service): `mako-finalize`: requires array of part etags

mako-finalize

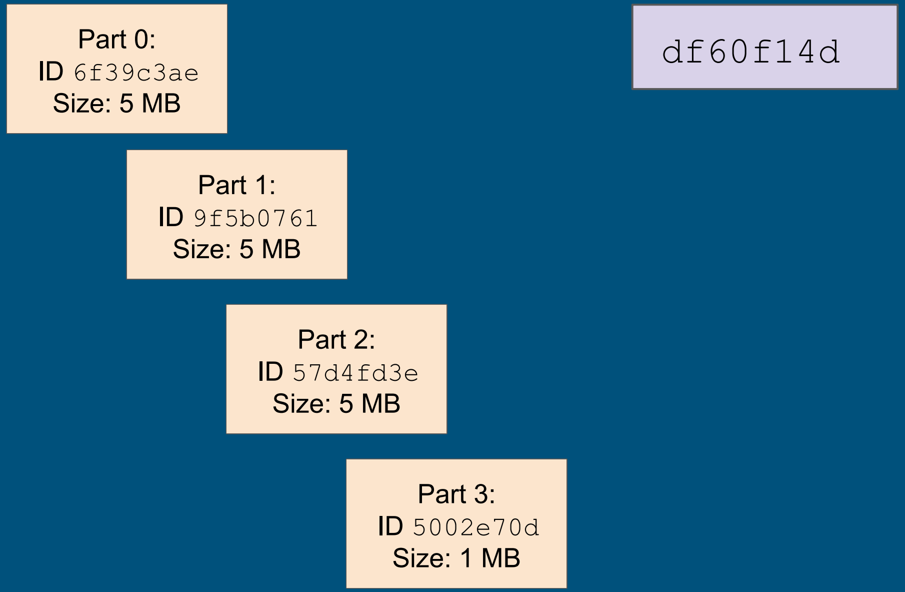
```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
targetFile: df60f14d
```



3.stor.us-east.joyent.us

mako-finalize

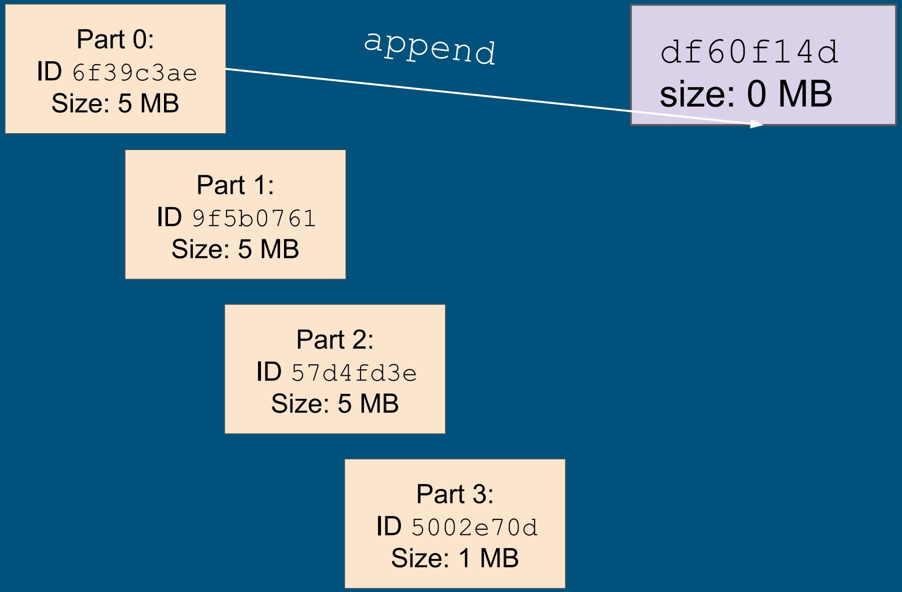
```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
targetFile: df60f14d
```



`3.stor.us-east.joyent.us`

mako-finalize

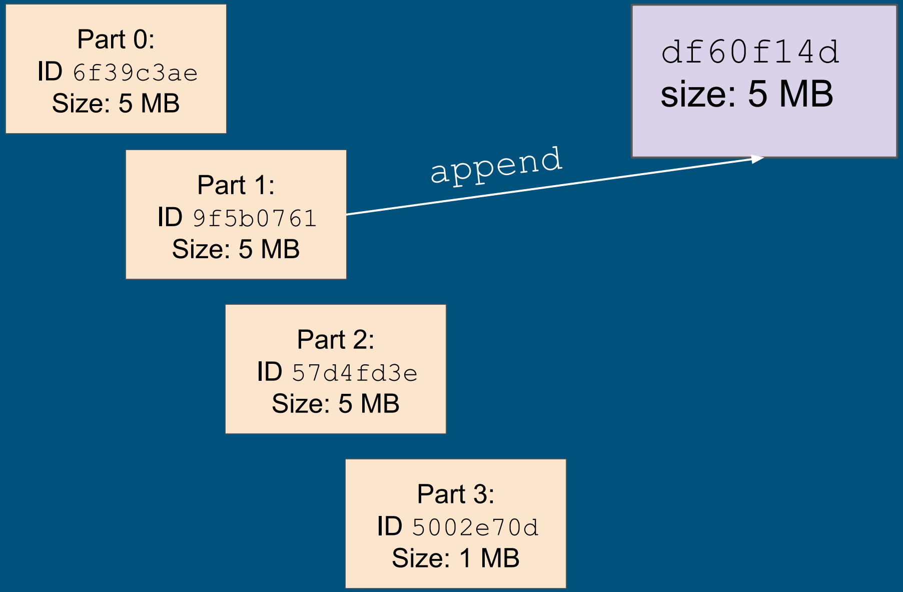
```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
targetFile: df60f14d
```



3.stor.us-east.joyent.us

mako-finalize

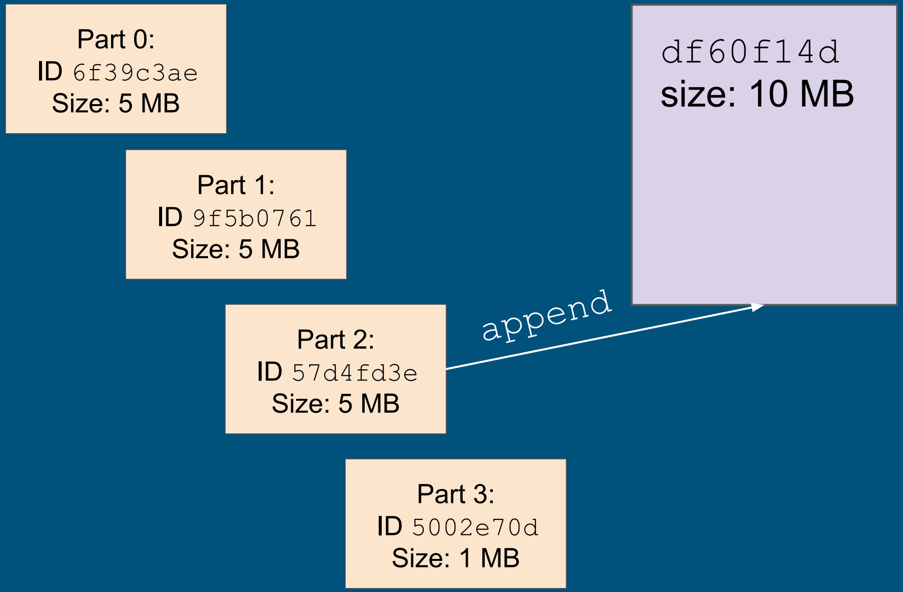
```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
targetFile: df60f14d
```



3.stor.us-east.joyent.us

mako-finalize

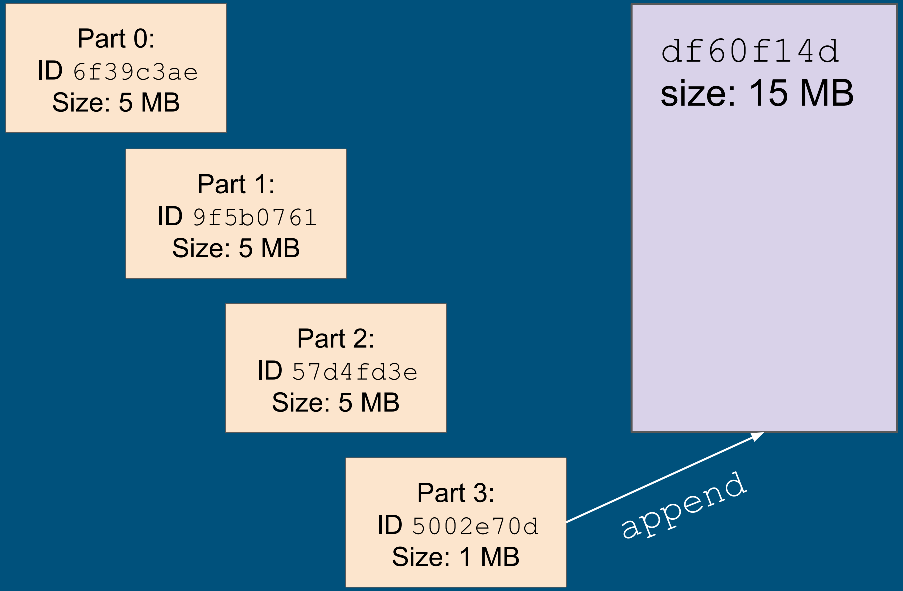
```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
targetFile: df60f14d
```



3.stor.us-east.joyent.us

mako-finalize

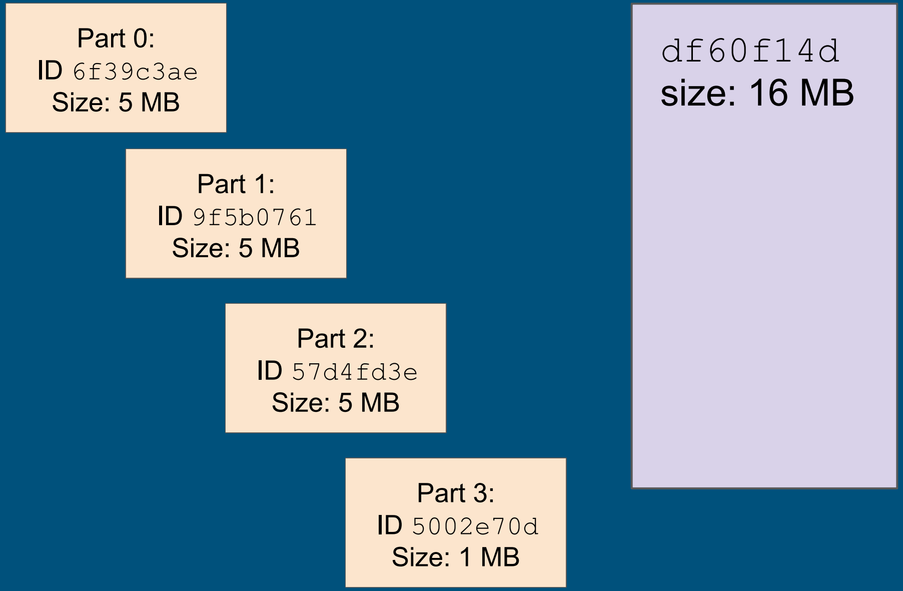
```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
targetFile: df60f14d
```



3.stor.us-east.joyent.us

mako-finalize

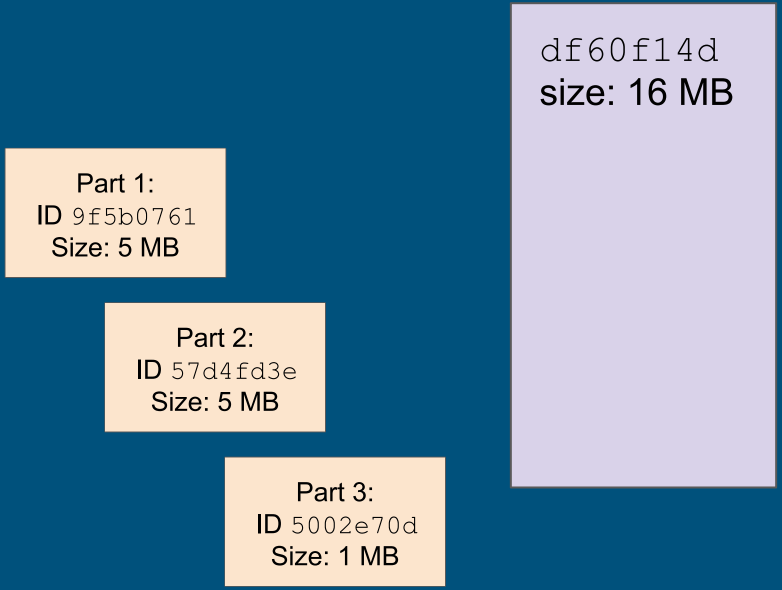
```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
targetFile: df60f14d
```



3.stor.us-east.joyent.us

mako-finalize

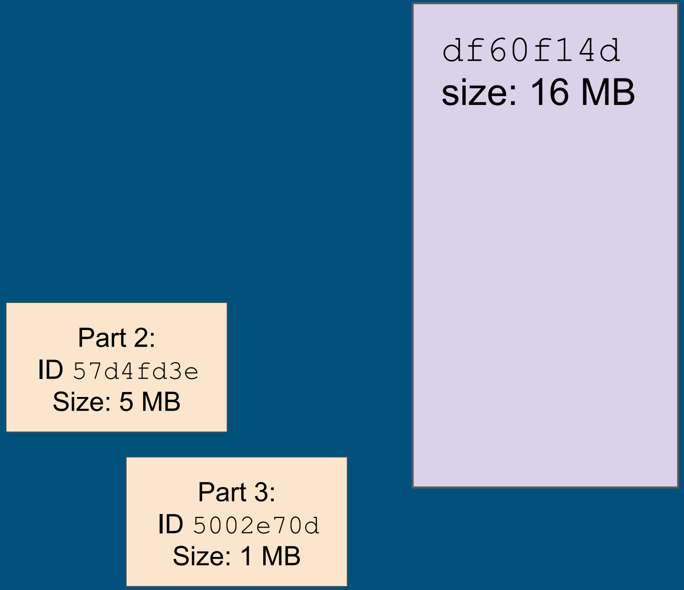
```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
targetFile: df60f14d
```



`3.stor.us-east.joyent.us`

mako-finalize

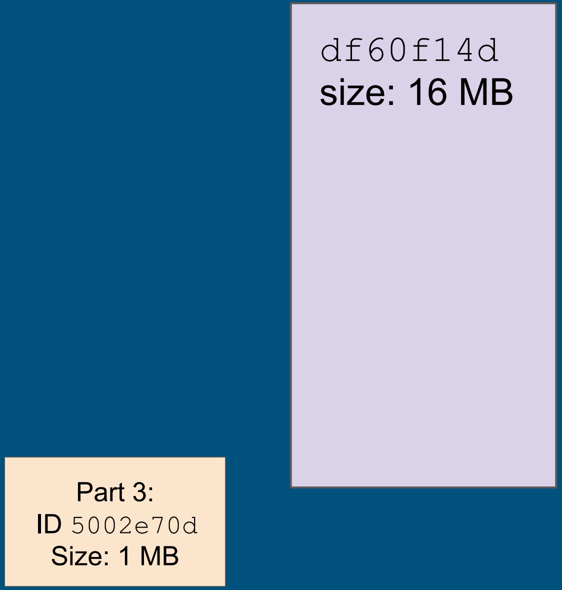
```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
targetFile: 67115618
```



3.stor.us-east.joyent.us

mako-finalize

```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
targetFile: df60f14d
```



3.stor.us-east.joyent.us

mako-finalize

```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
targetFile: df60f14d
```

df60f14d
size: 16 MB

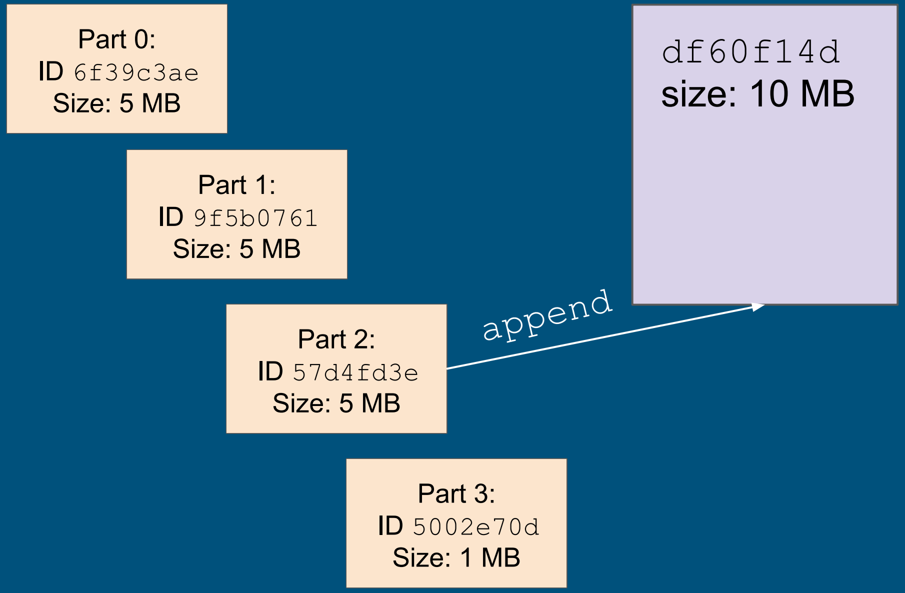
3.stor.us-east.joyent.us

`mako-finalize`: atomicity & idempotency

- What happens if mako crashes…
  - While writing to the target object file?

mako-finalize

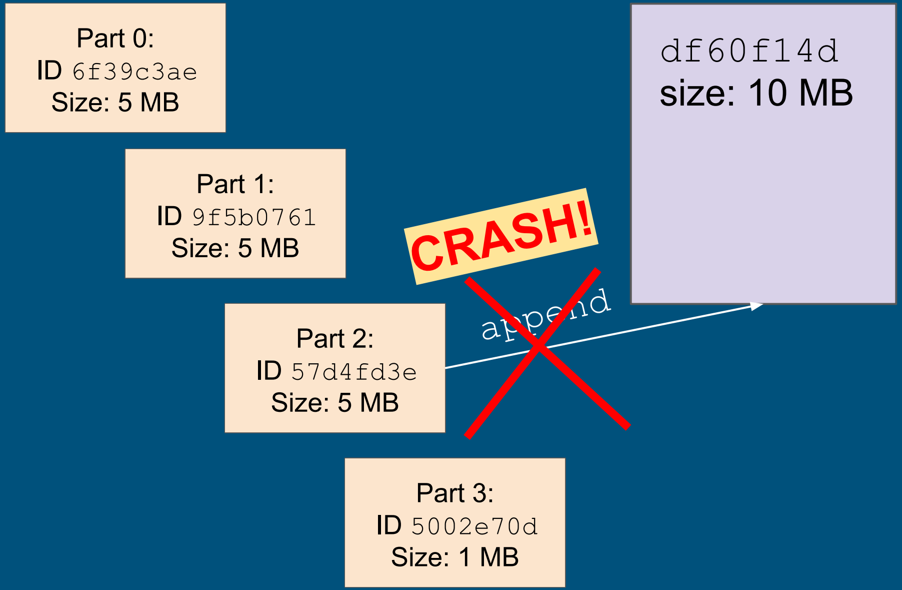
```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
outputFile: df60f14d
```



`3.stor.us-east.joyent.us`

mako-finalize

```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
outputFile: df60f14d
```



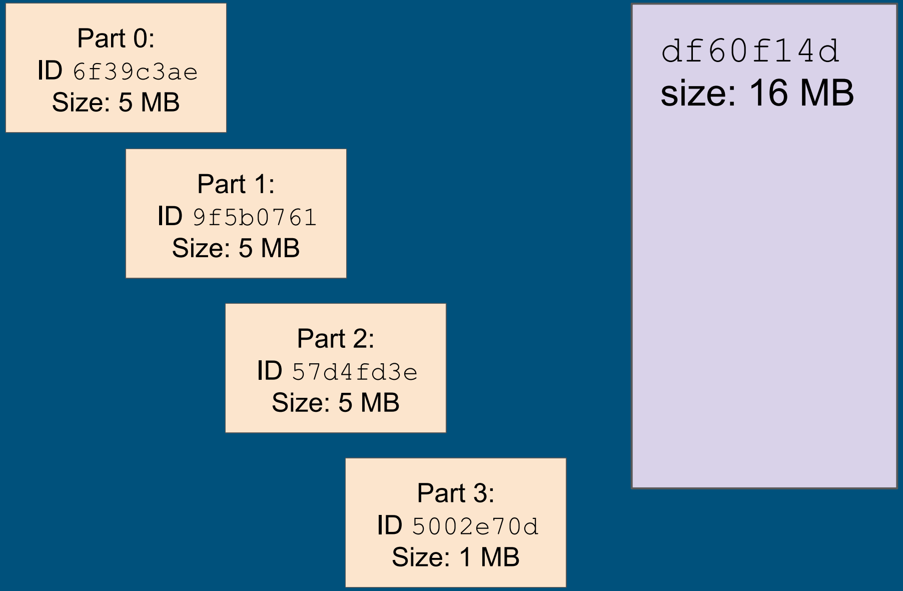
3.stor.us-east.joyent.us

`mako-finalize`: atomicity & idempotency

- What happens if mako crashes…
  - While writing the target object file?
    - `mako-finalize` can safely retry later, because it hasn’t removed the parts yet
  - While removing parts?

mako-finalize

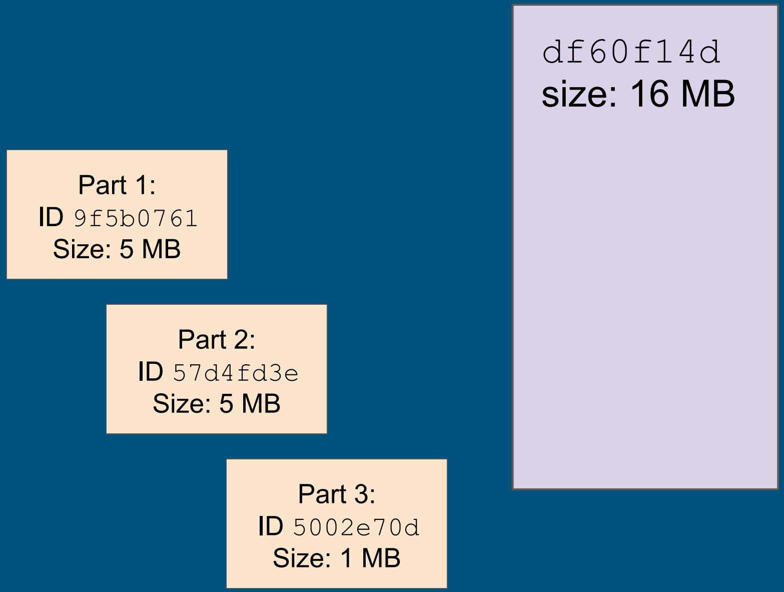
```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
targetFile: df60f14d
```



3.stor.us-east.joyent.us

mako-finalize

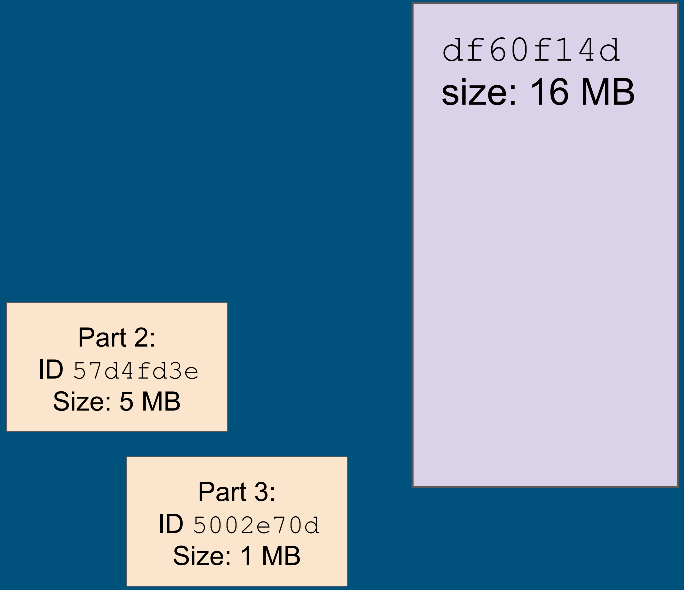
```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
targetFile: df60f14d
```



3.stor.us-east.joyent.us

mako-finalize

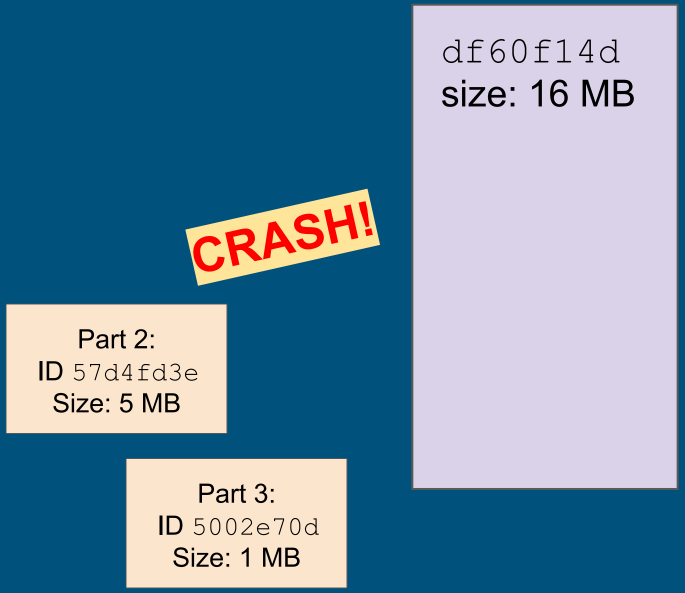
```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
targetFile: df60f14d
```



`3.stor.us-east.joyent.us`

mako-finalize

```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
targetFile: df60f14d
```



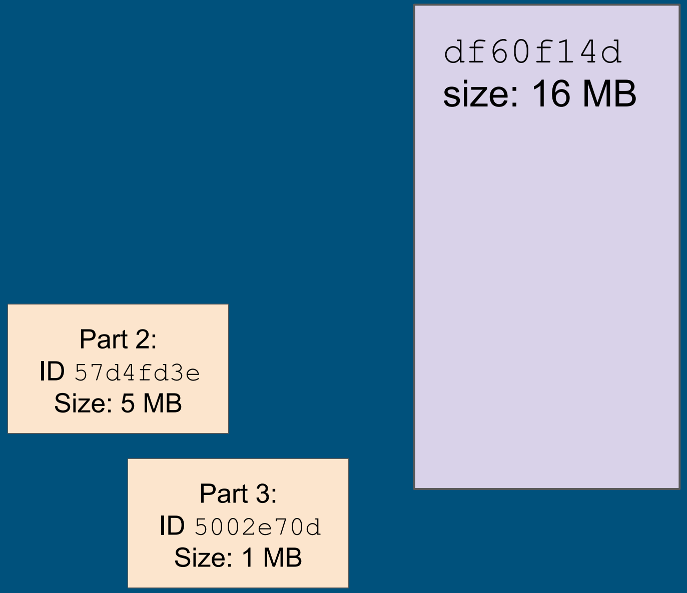
3.stor.us-east.joyent.us

`mako-finalize`: atomicity & idempotency

- What happens if mako crashes…
  - While writing the target object file?
    - Can safely retry later, because it hasn’t removed the parts yet
  - While removing parts?
    - Can still retry later…
    - Need a way to check that the target object file exists and is correct

mako-finalize

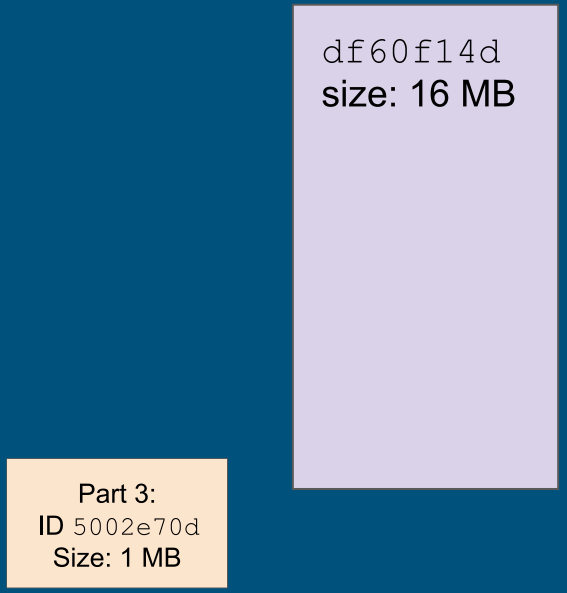
```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
targetFile: df60f14d
nbytes:16 MB
```



3.stor.us-east.joyent.us

mako-finalize

```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
targetFile: df60f14d
nbytes:16 MB
```



`3.stor.us-east.joyent.us`

mako-finalize

```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
targetFile: df60f14d
nbytes:16 MB
```

df60f14d
size: 16 MB

3.stor.us-east.joyent.us

mako-finalize $\longrightarrow$ **SUCCESS!**

```
parts: [
    6f39c3ae,
    9f5b0761,
    57d4fd3e,
    5002e70d
]
targetFile: df60f14d
nbytes:16 MB
```

df60f14d
size: 16 MB

3.stor.us-east.joyent.us

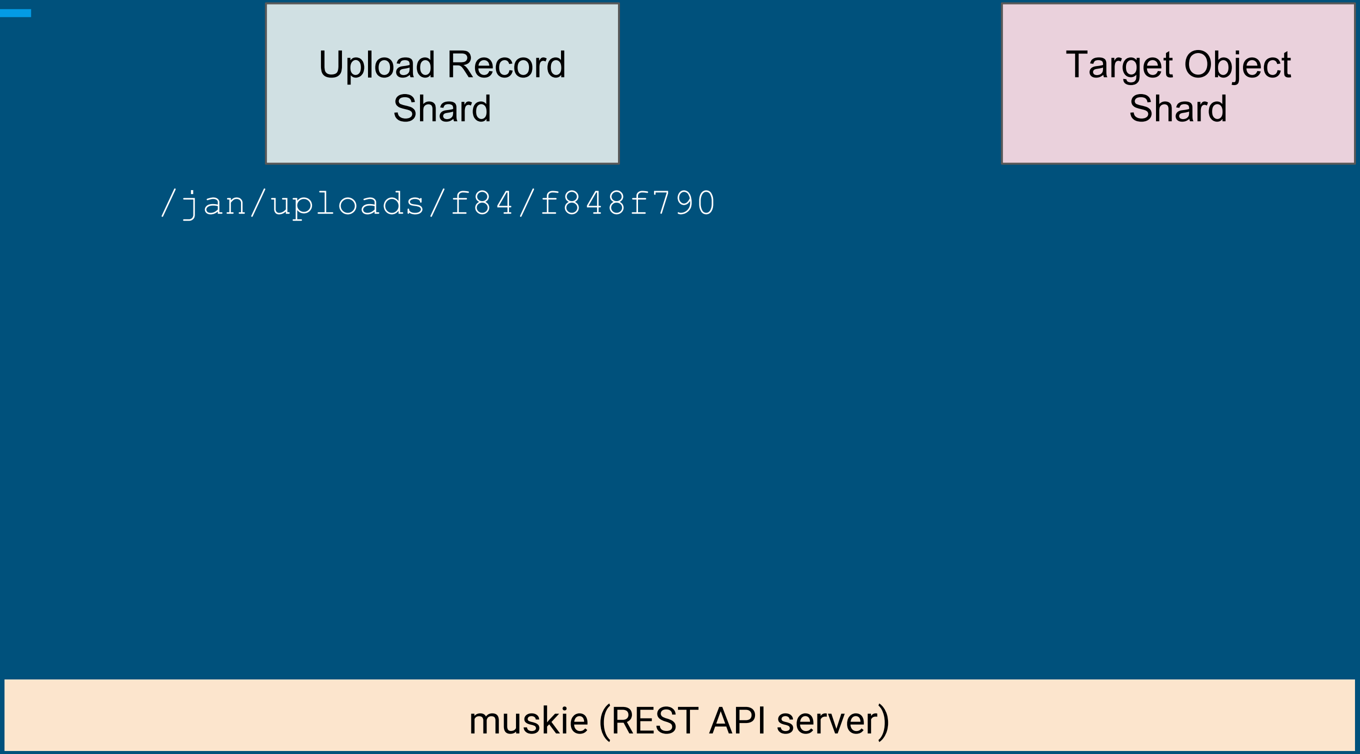
`mako-finalize`: atomicity & idempotency

- Protections presented thus far prevent problems from the same MPU (exact same set of parts)
- Additional constraints needed at the muskie (REST API) layer
  - Don't want to allow two clients to commit a different set of parts for the same MPU
    - Check for conflicts *before* invoking `mako-finalize`
  - Other conflicts:
    - Aborts and commits conflict with each other
      - If one client tries to commit an MPU and another tries to abort it, we only want one to win (atomicity)
  - **Need to store additional state in the metadata tier**

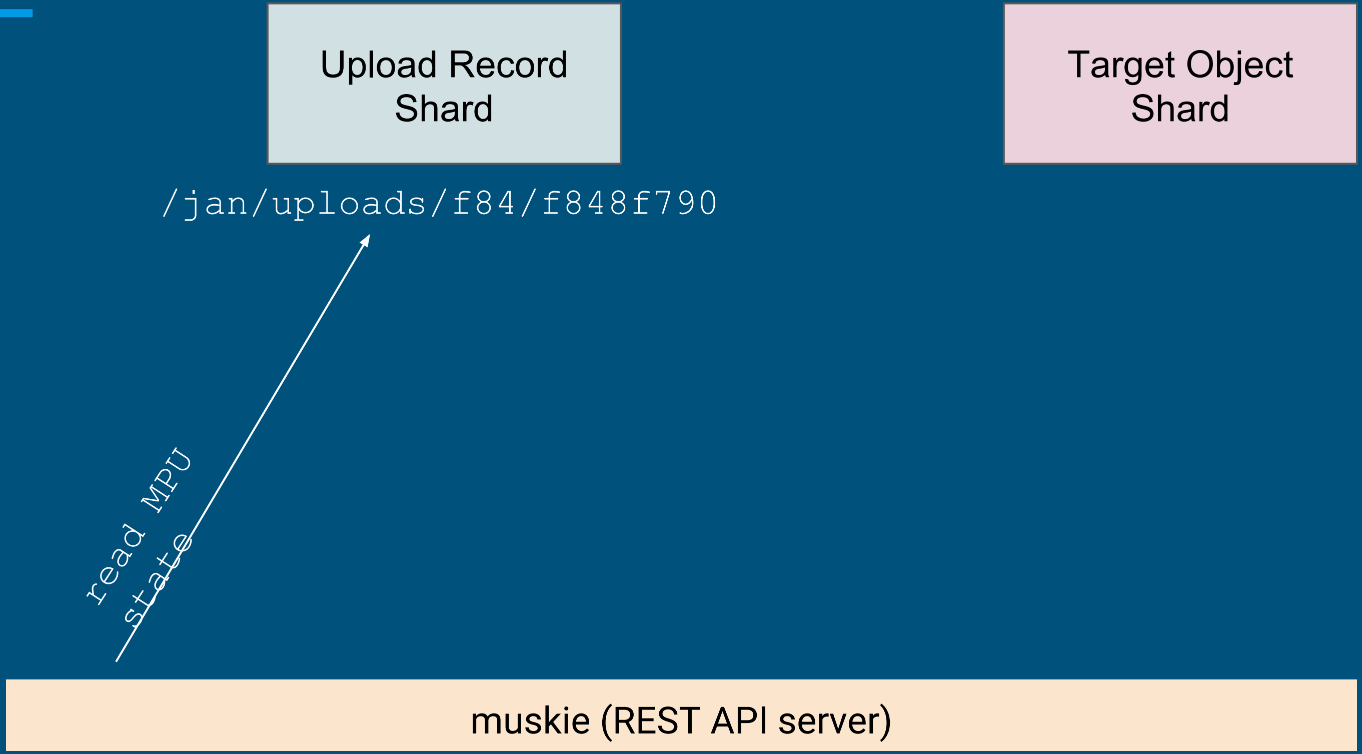
Commits: atomicity & idempotency

- Where do we store the multipart upload state?
- Considerations:
    - Need the state change and target object’s visibility in Manta to be an *atomic* operation
- Suppose state about an MPU was stored in only the metadata record of the parts directory…
    - Is this atomic?

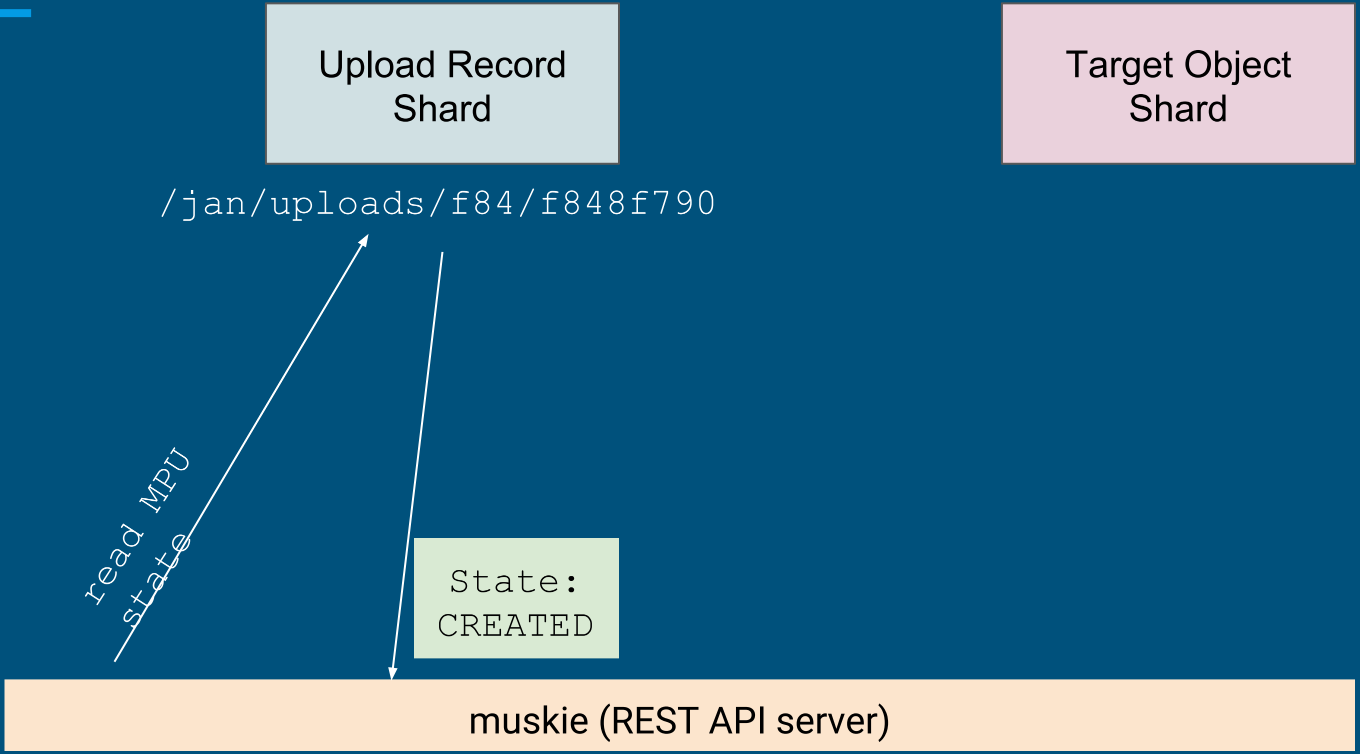
Multipart Upload Commits: Idempotency



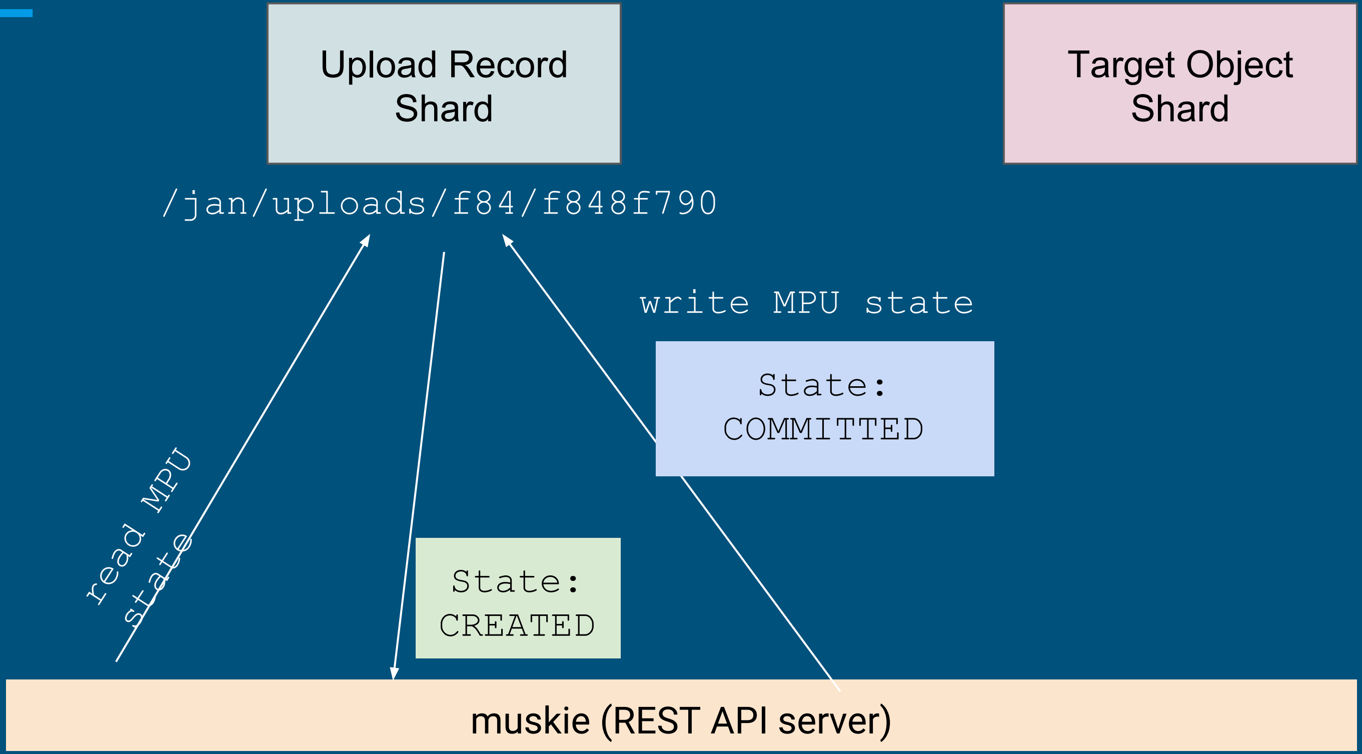
Multipart Upload Commits: Idempotency



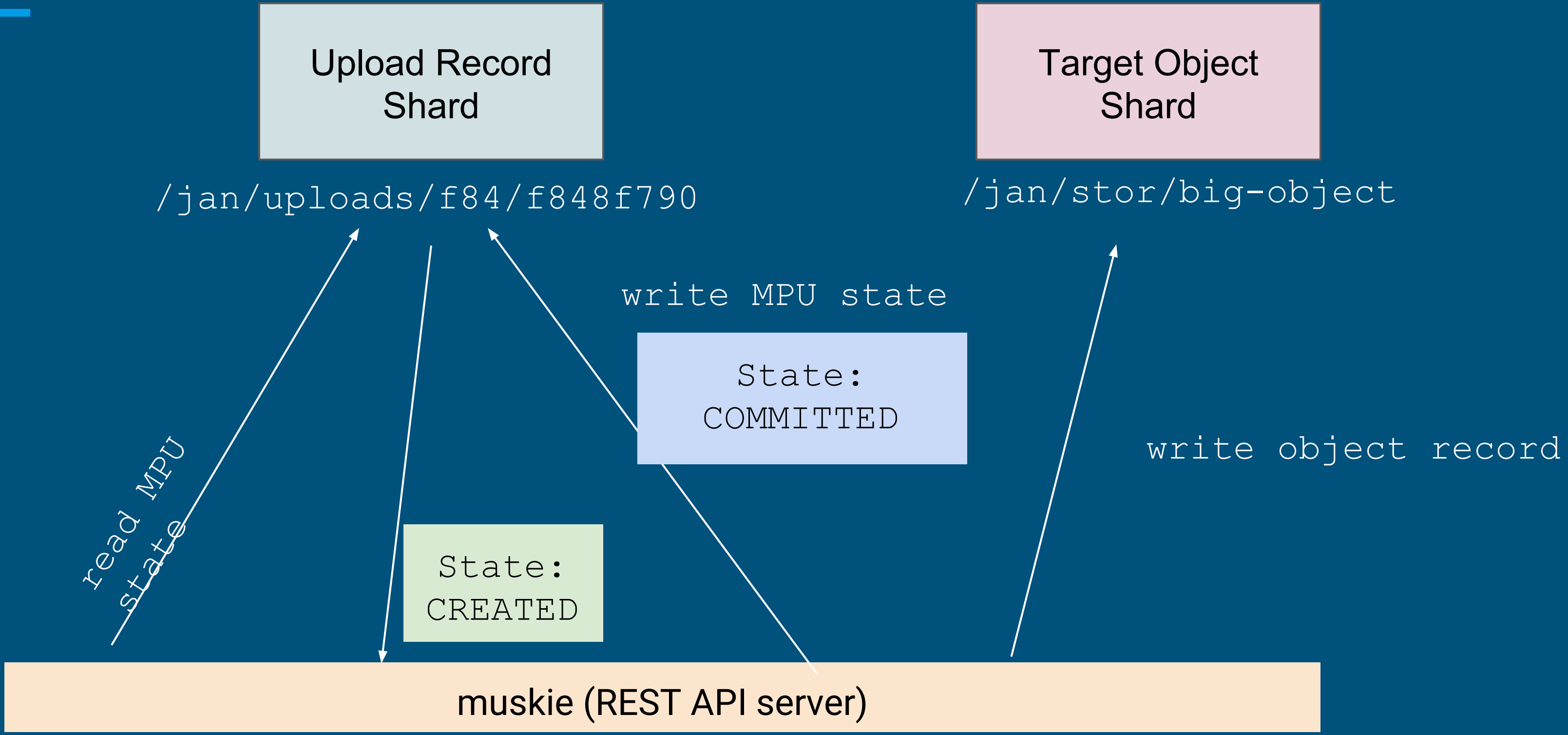
Multipart Upload Commits: Idempotency



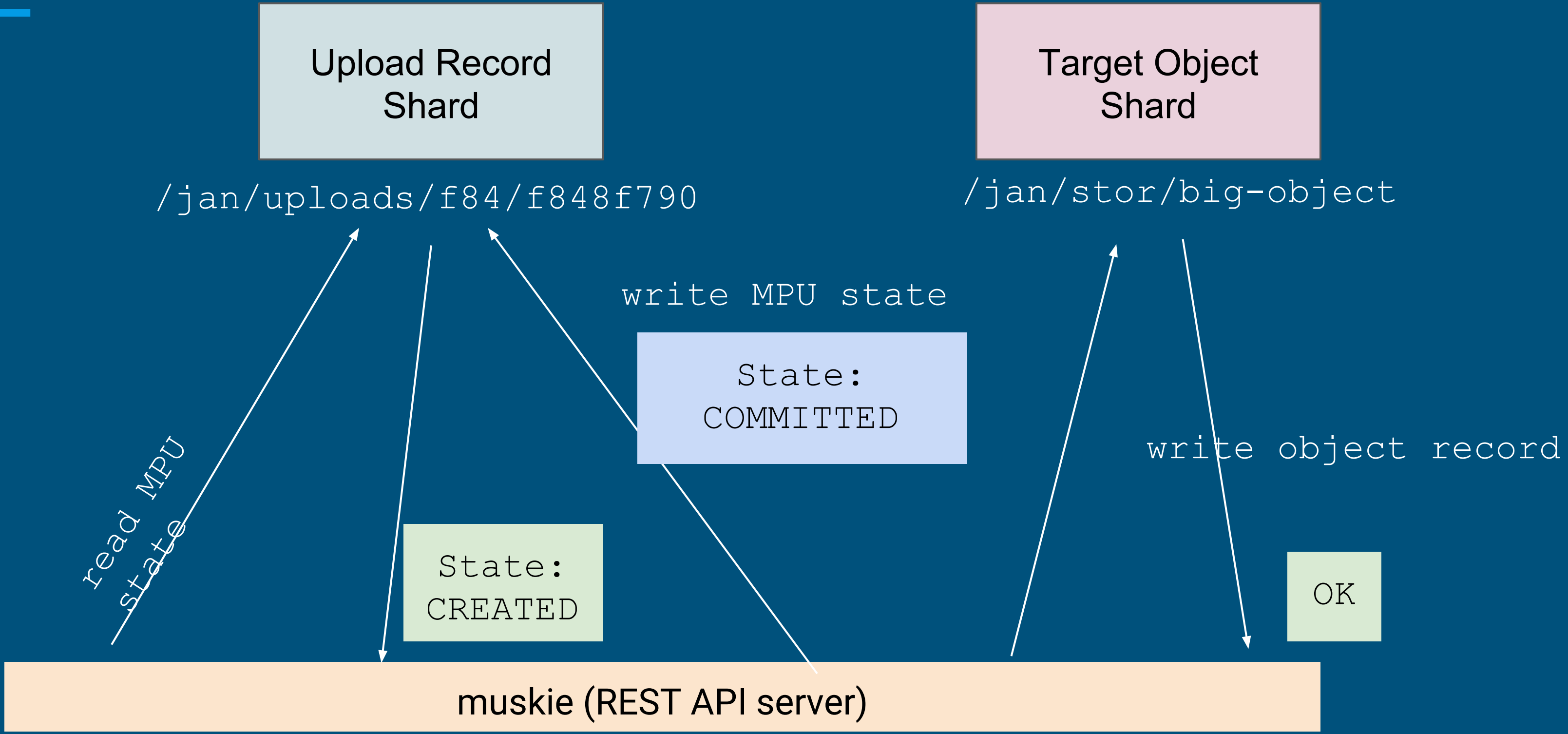
Multipart Upload Commits: Idempotency



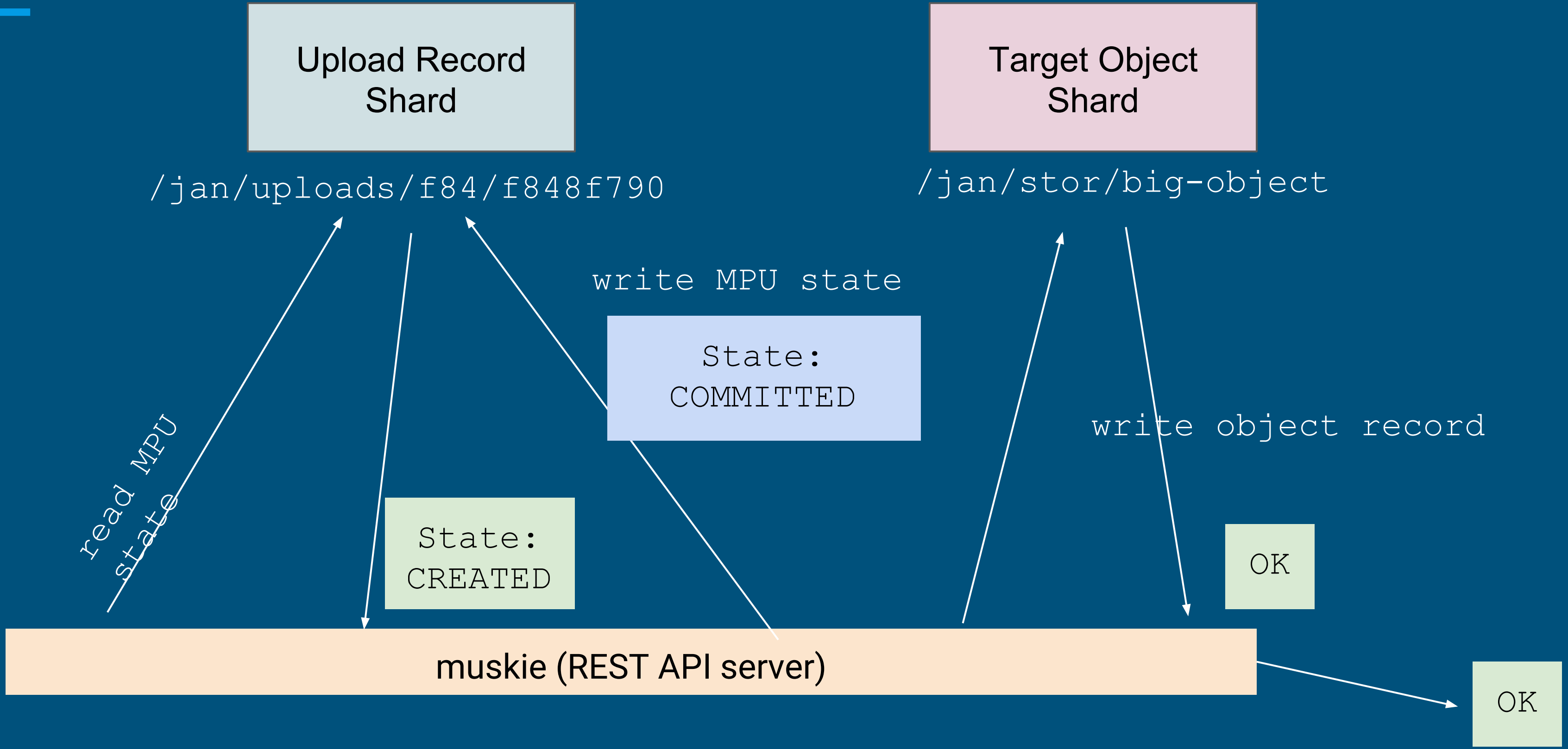
Multipart Upload Commits: Idempotency



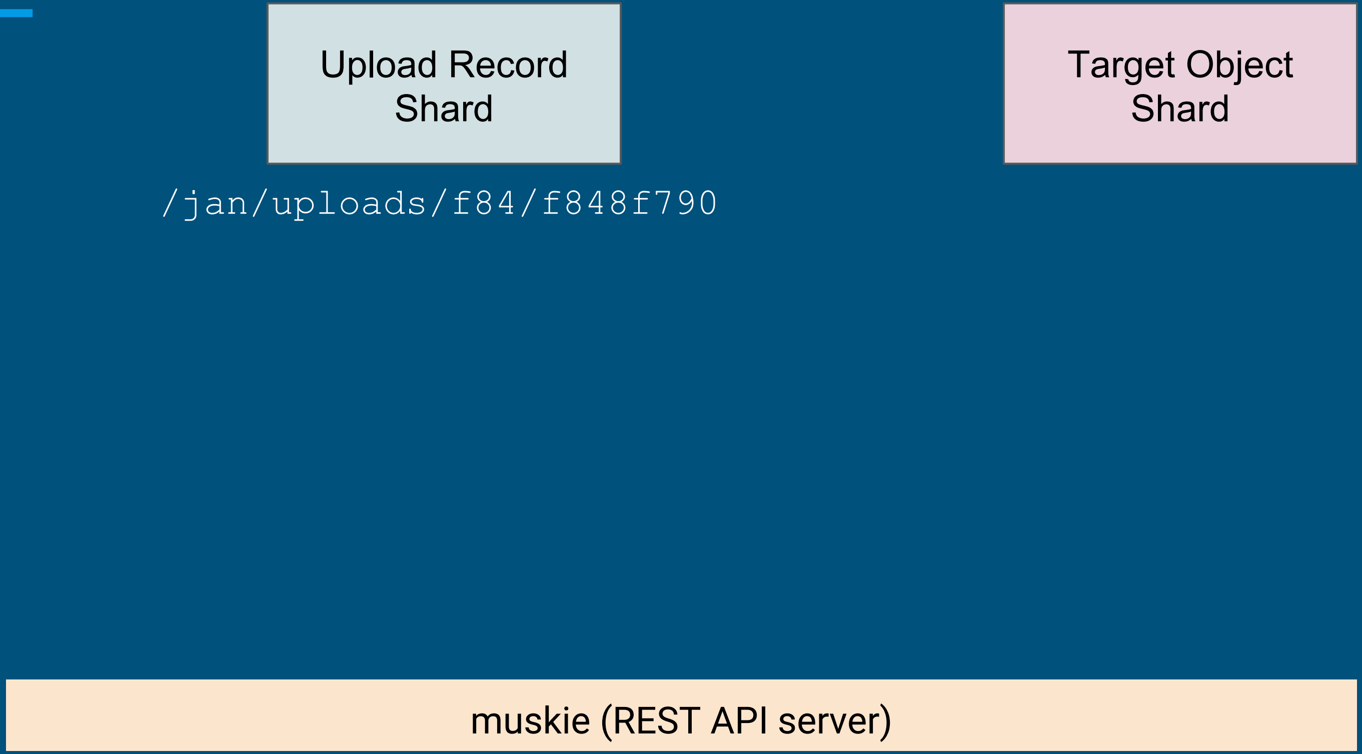
Multipart Upload Commits: Idempotency



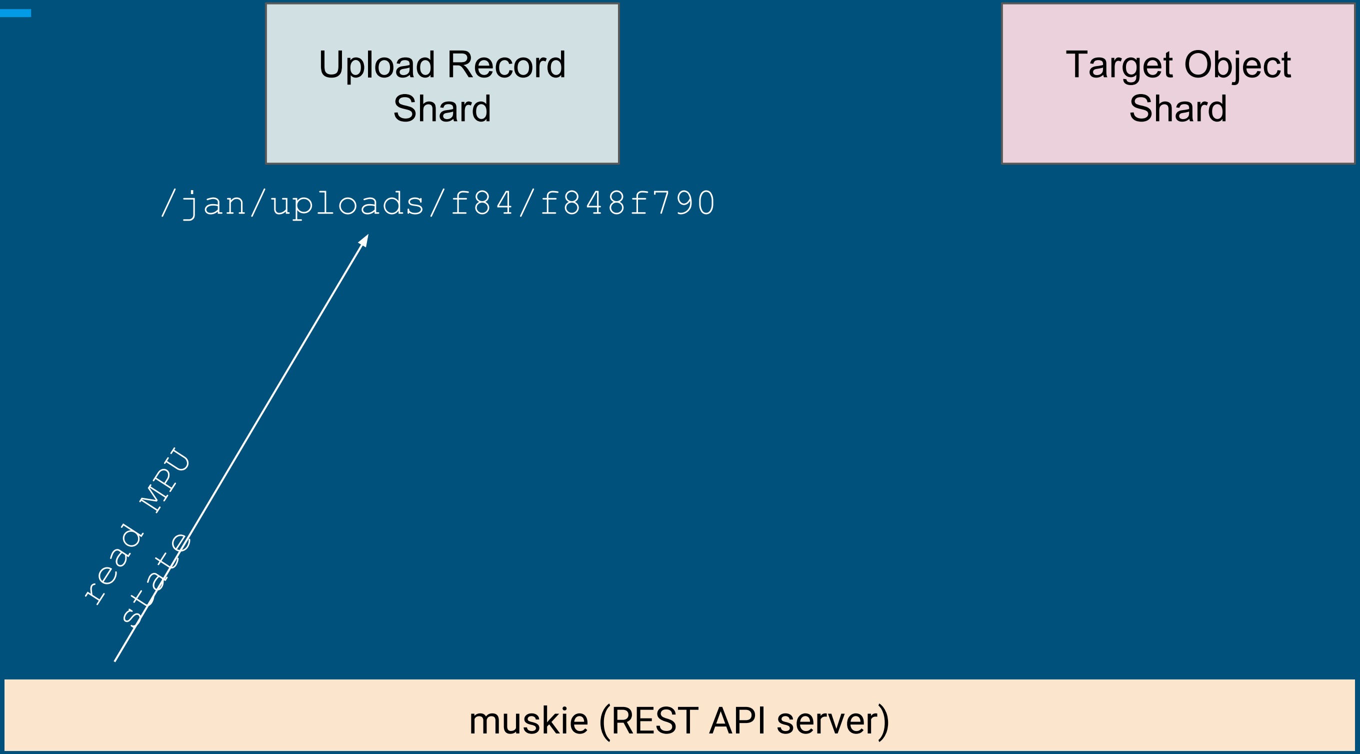
Multipart Upload Commits: Idempotency



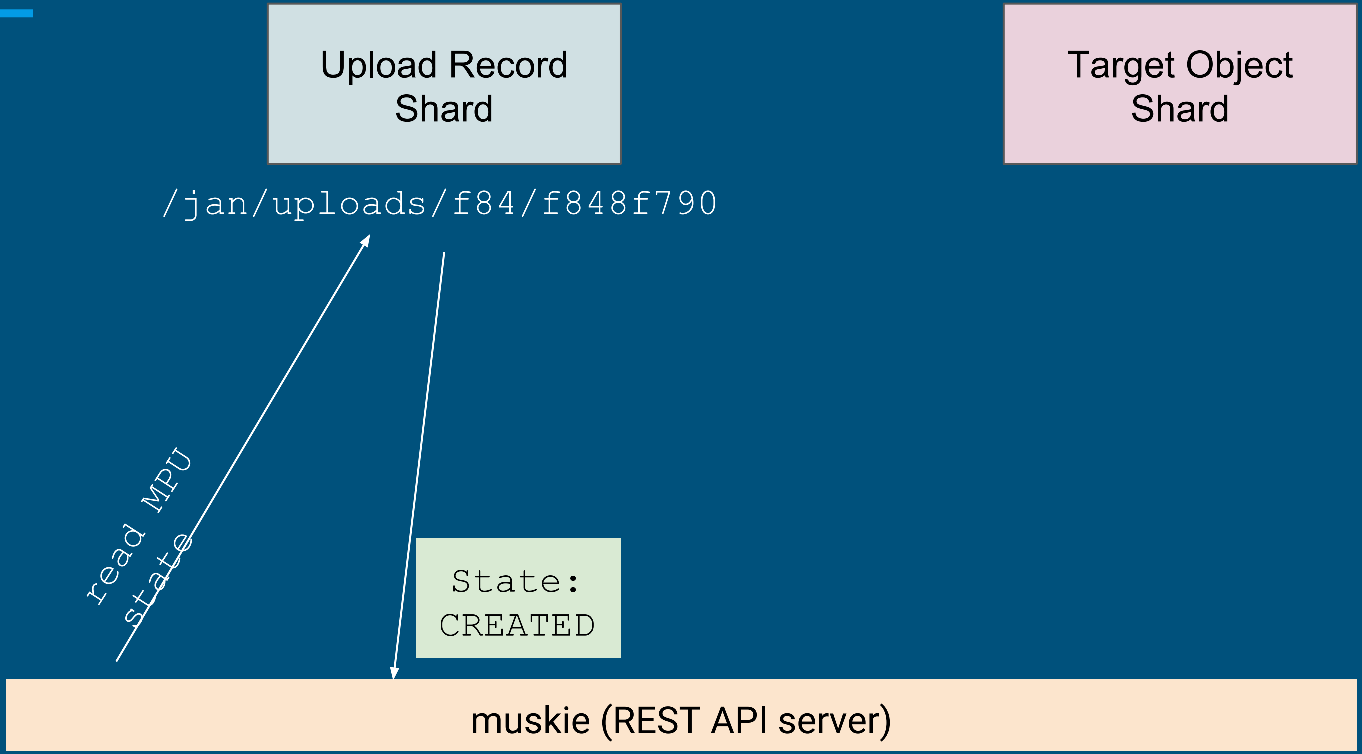
Multipart Upload Commits: Idempotency



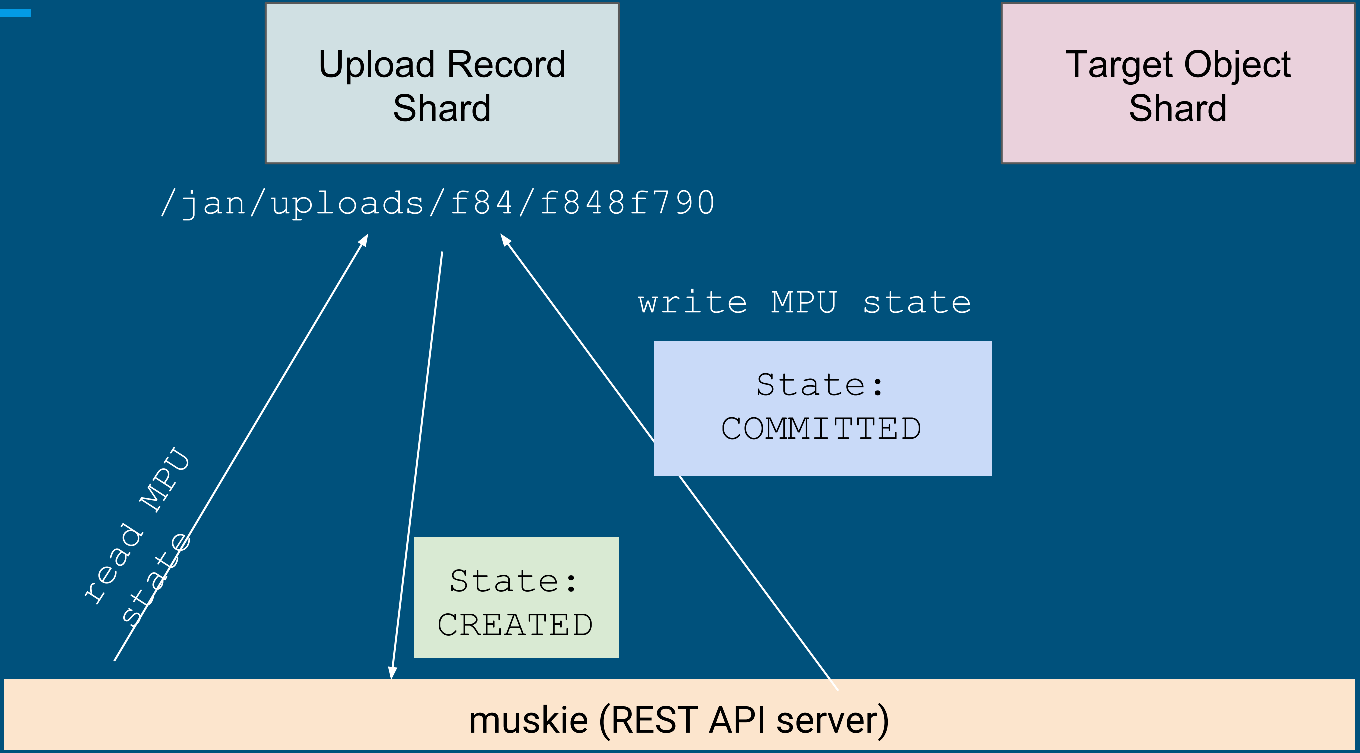
Multipart Upload Commits: Idempotency



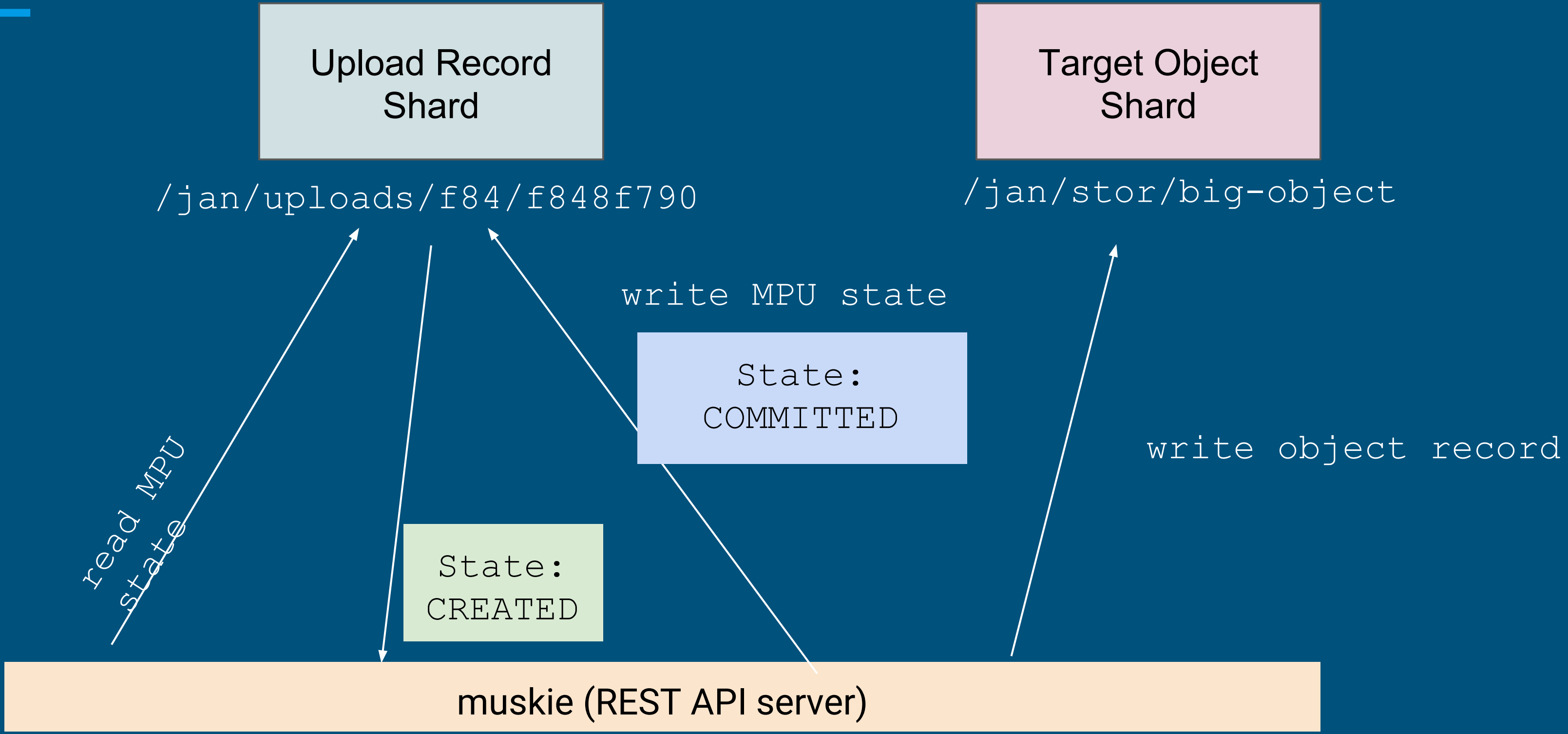
Multipart Upload Commits: Idempotency



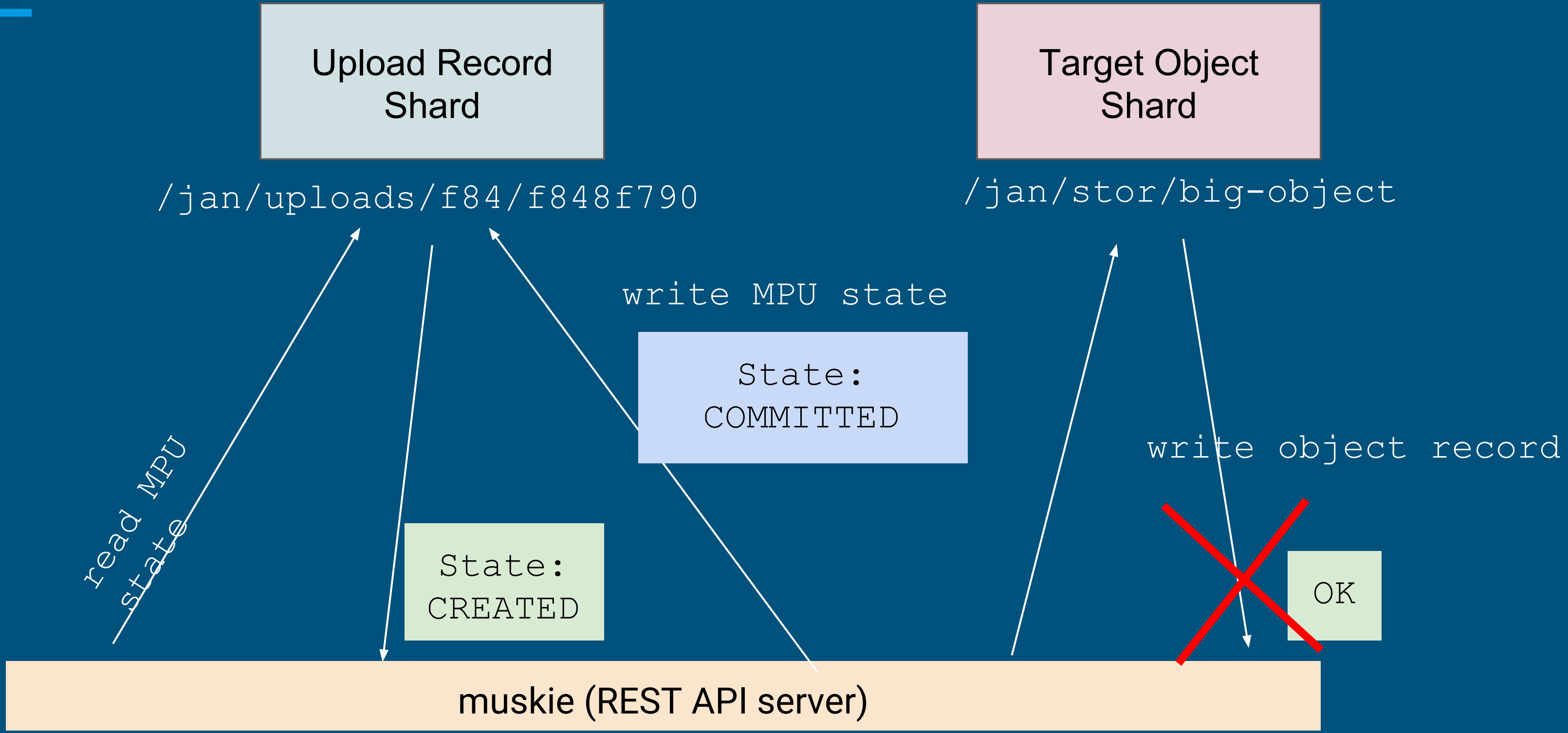
Multipart Upload Commits: Idempotency



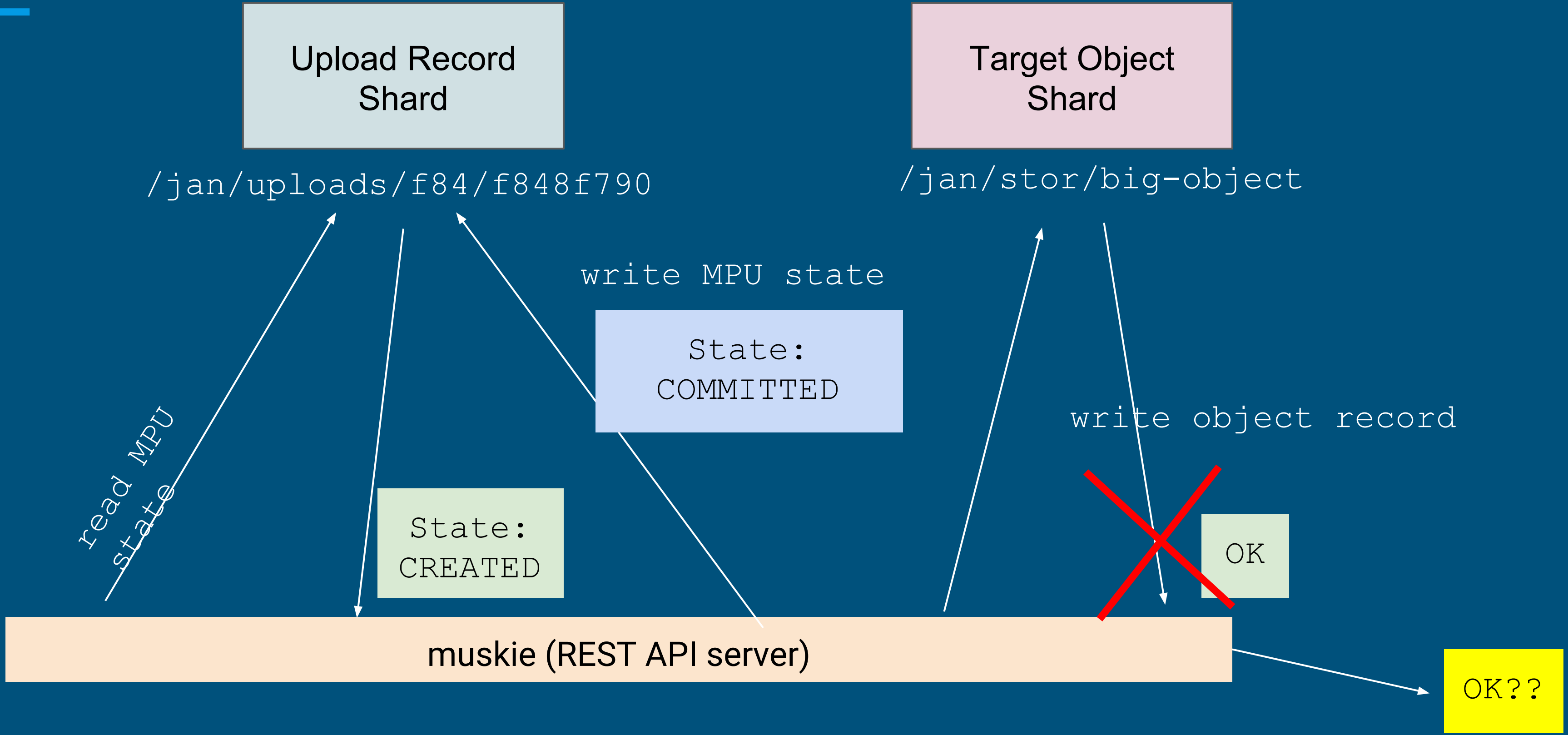
Multipart Upload Commits: Idempotency



Multipart Upload Commits: Idempotency



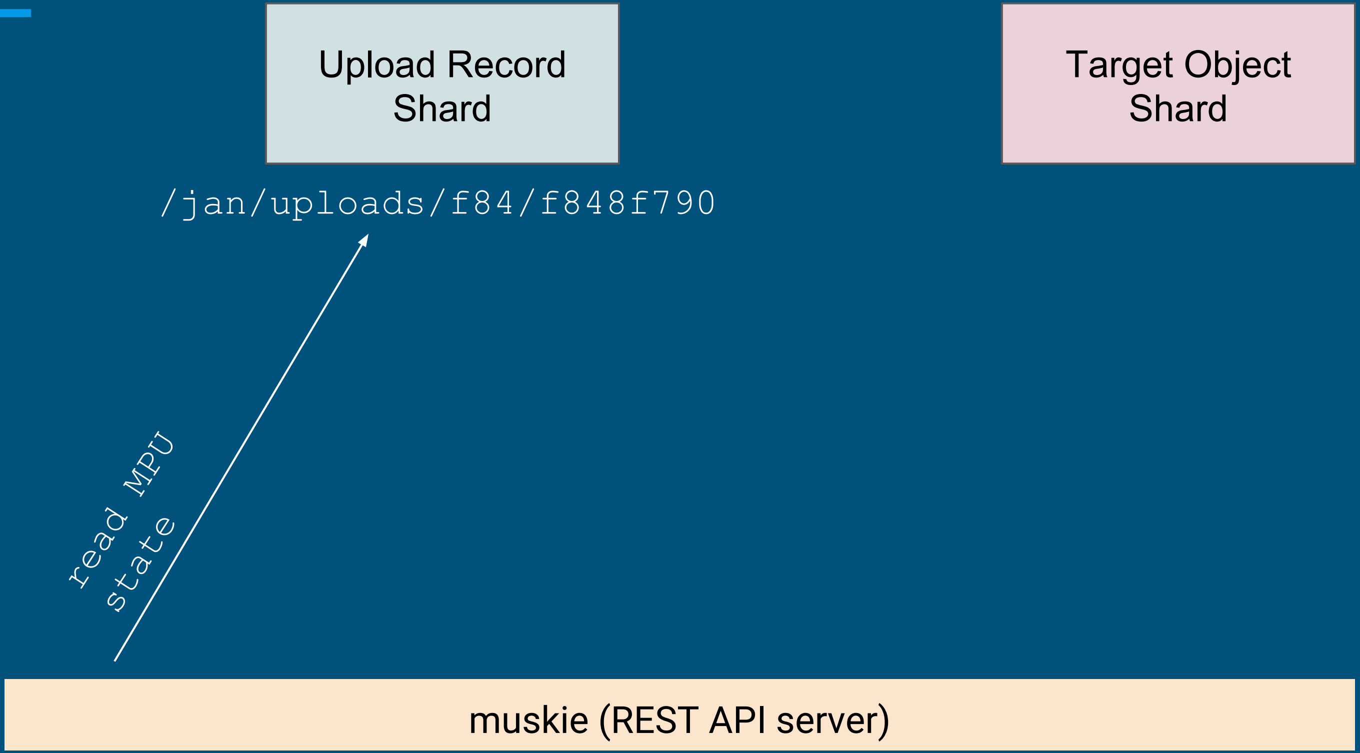
Multipart Upload Commits: Idempotency



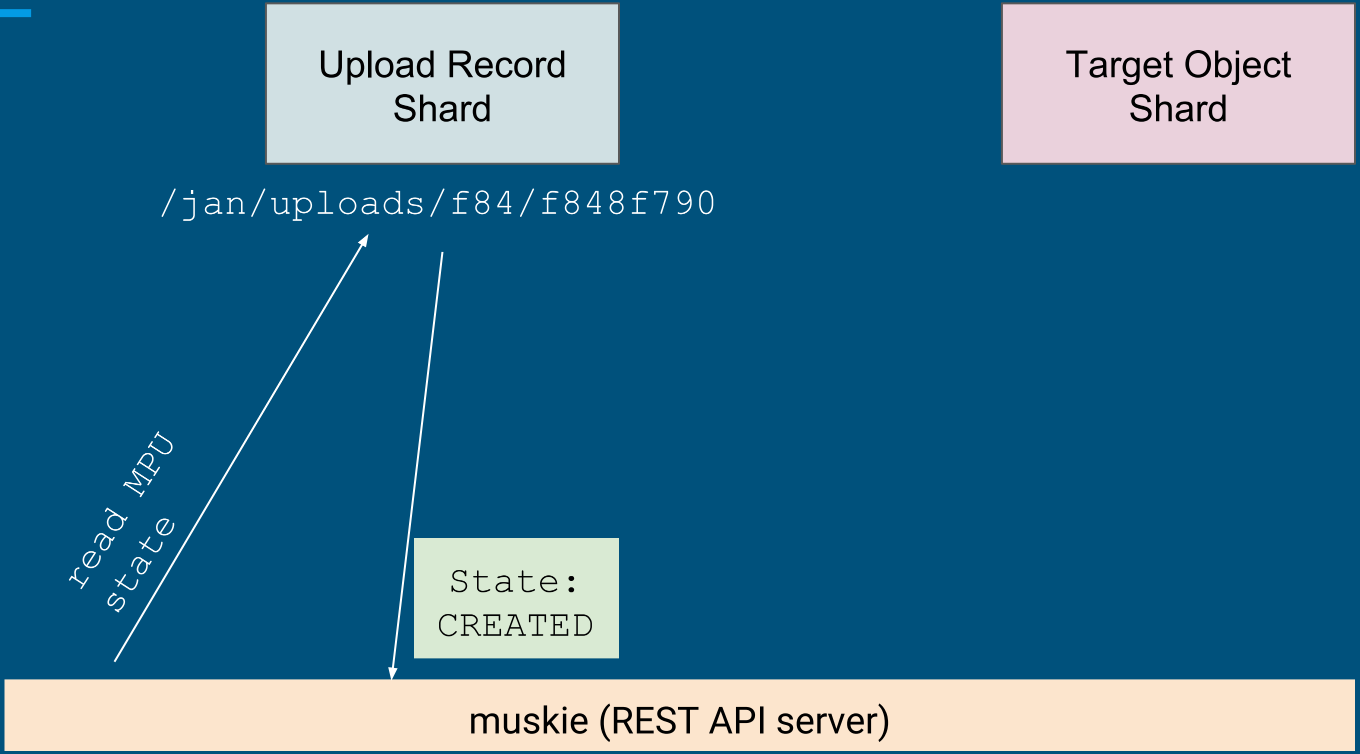
Commits: atomicity & idempotency

- Storing official state of MPU on parts directory metadata record is not atomic!
- Need visibility of object and state change of multipart upload to occur together, or not at all
- Solution?
  - Store official state (whether MPU is done) on the same shard as object!
  - Can still store some state on parts directory record as an optimization

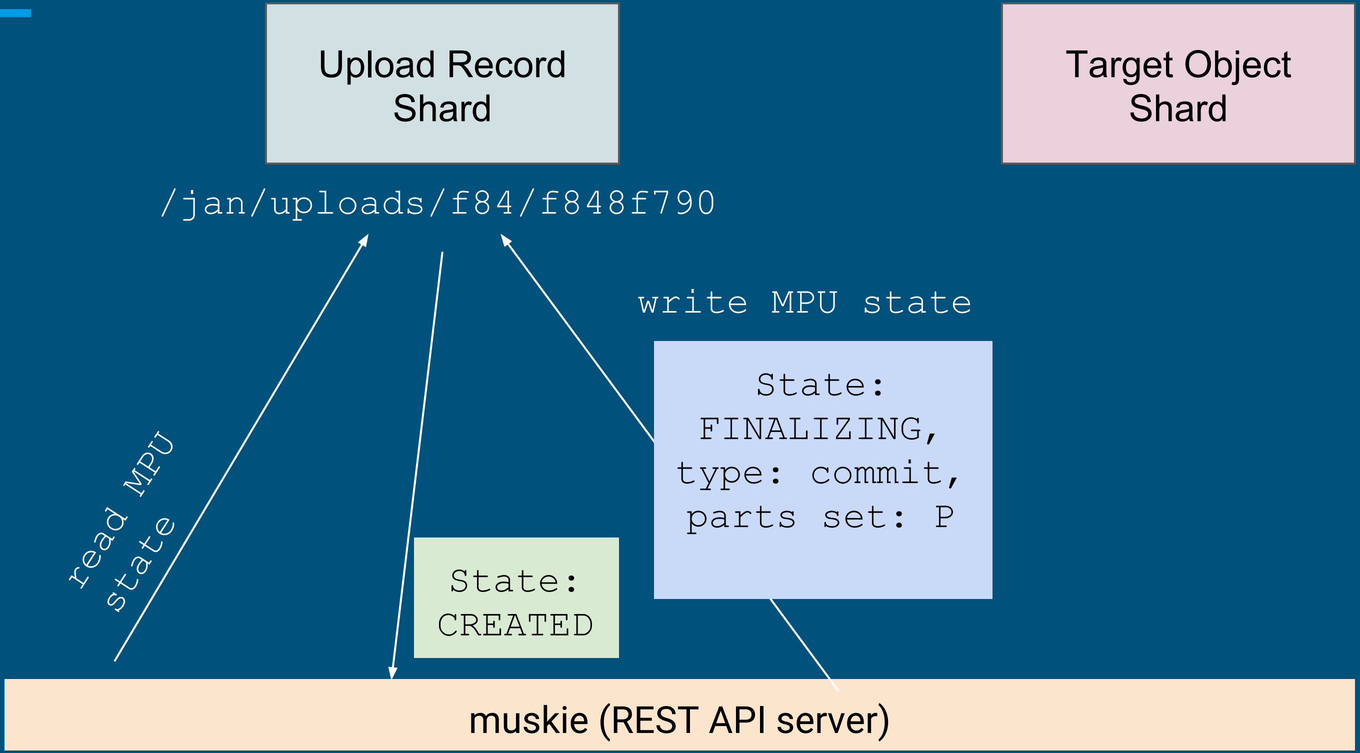
Multipart Upload Commits: Idempotency



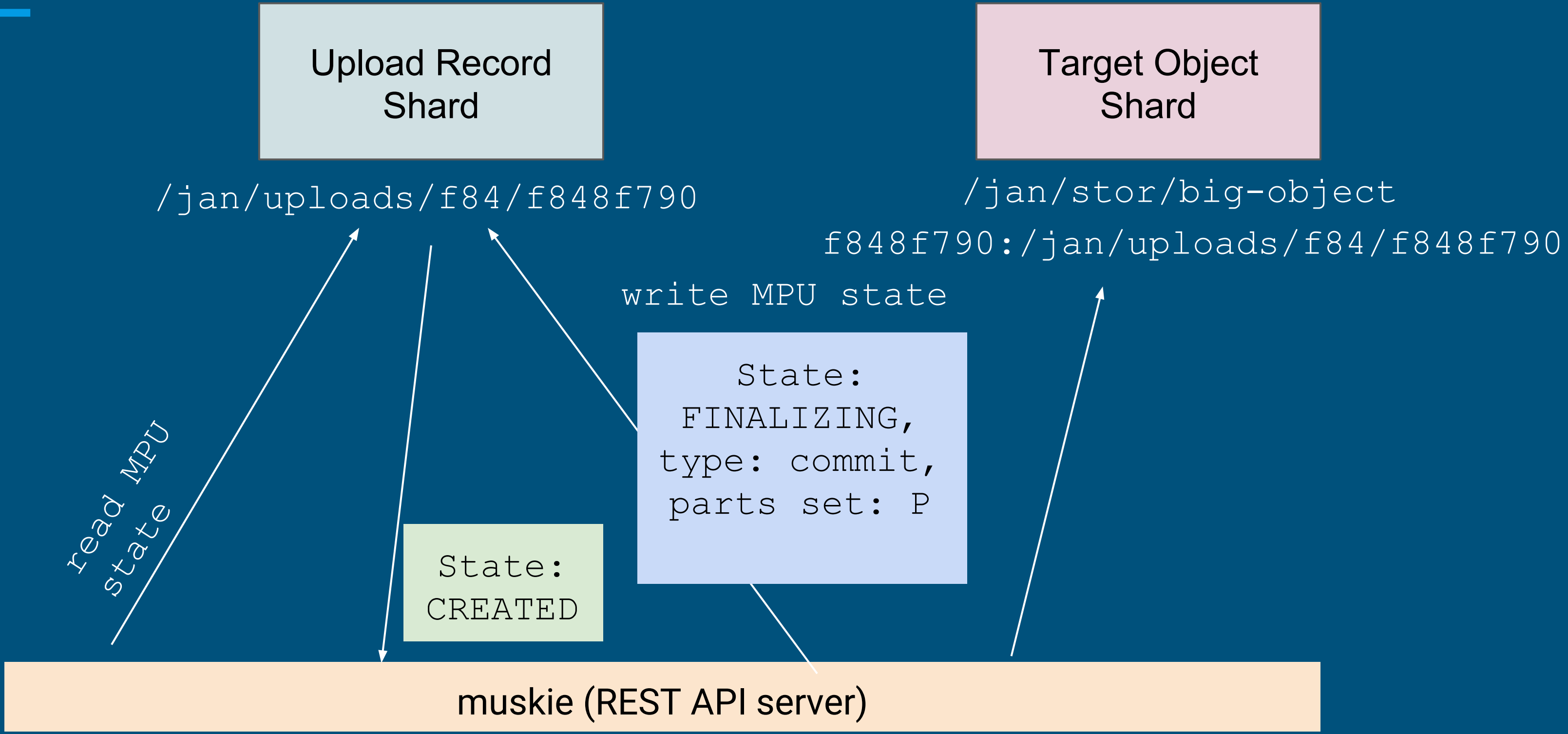
Multipart Upload Commits: Idempotency



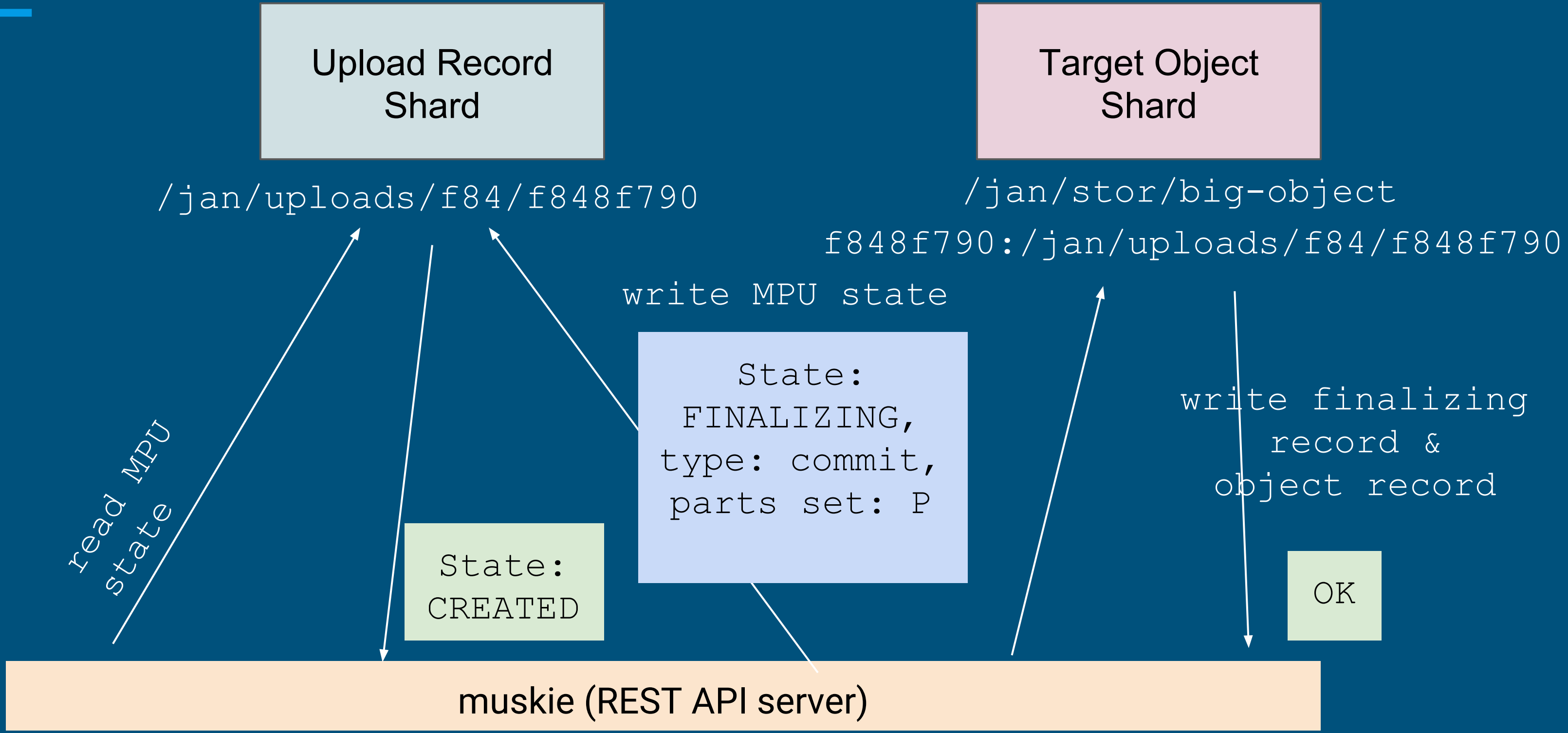
Multipart Upload Commits: Idempotency



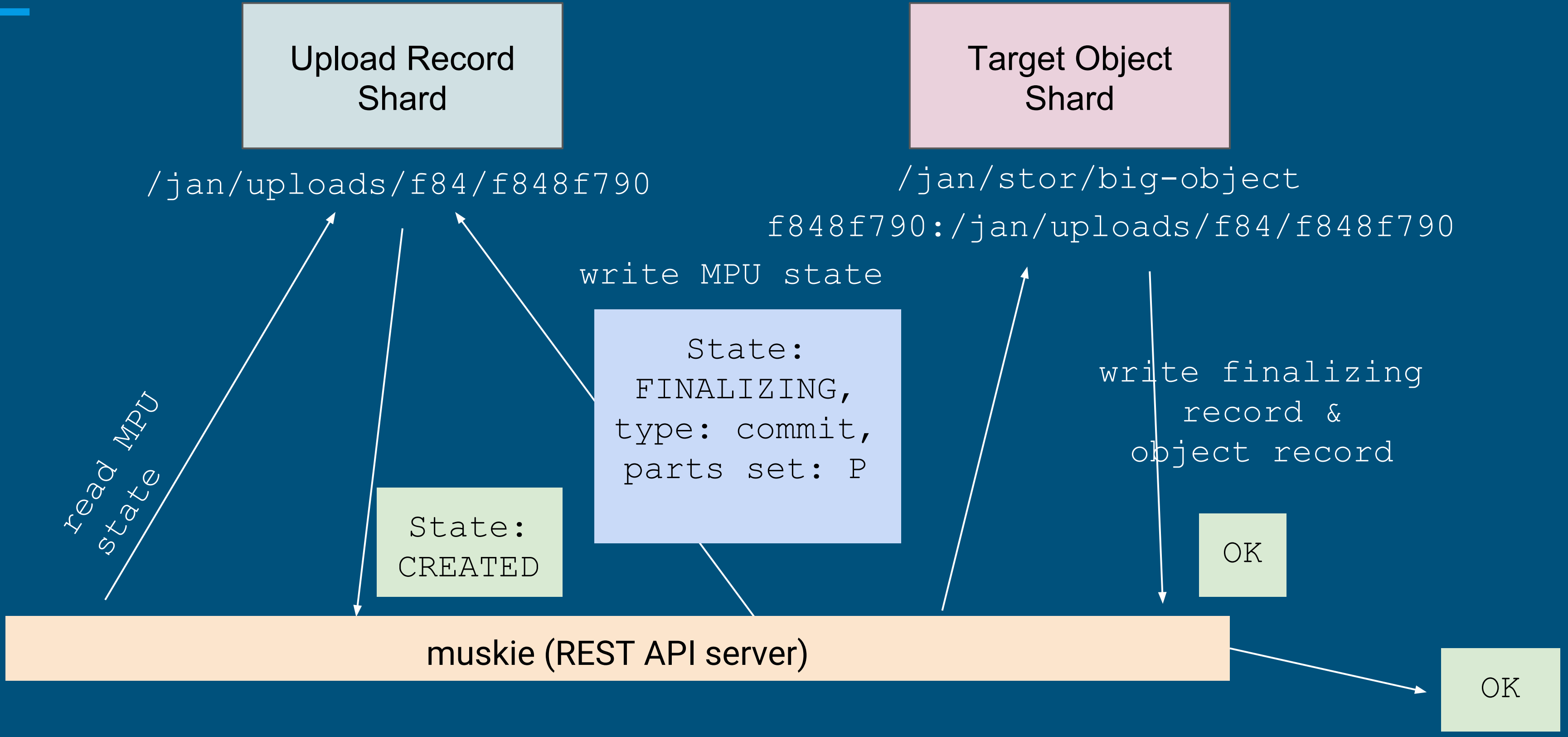
Multipart Upload Commits: Idempotency



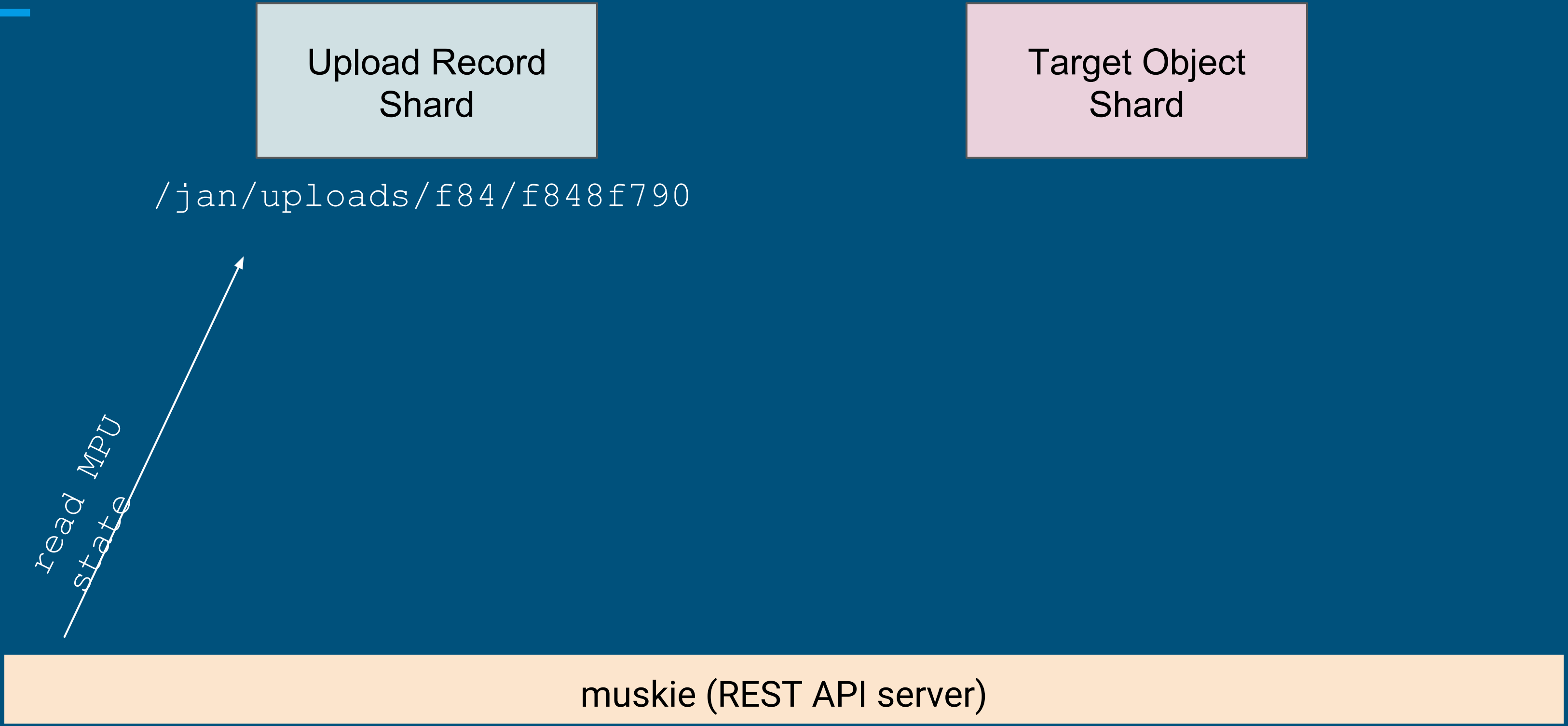
Multipart Upload Commits: Idempotency


Upload Record Shard
Target Object Shard
/jan/uploads/f84/f848f790
/jan/stor/big-object
f848f790:/jan/uploads/f84/f848f790
read MPU state
State: CREATED
write MPU state
State: FINALIZING, type: commit, parts set: P
write finalizing record & object record
OK
muskie (REST API server)

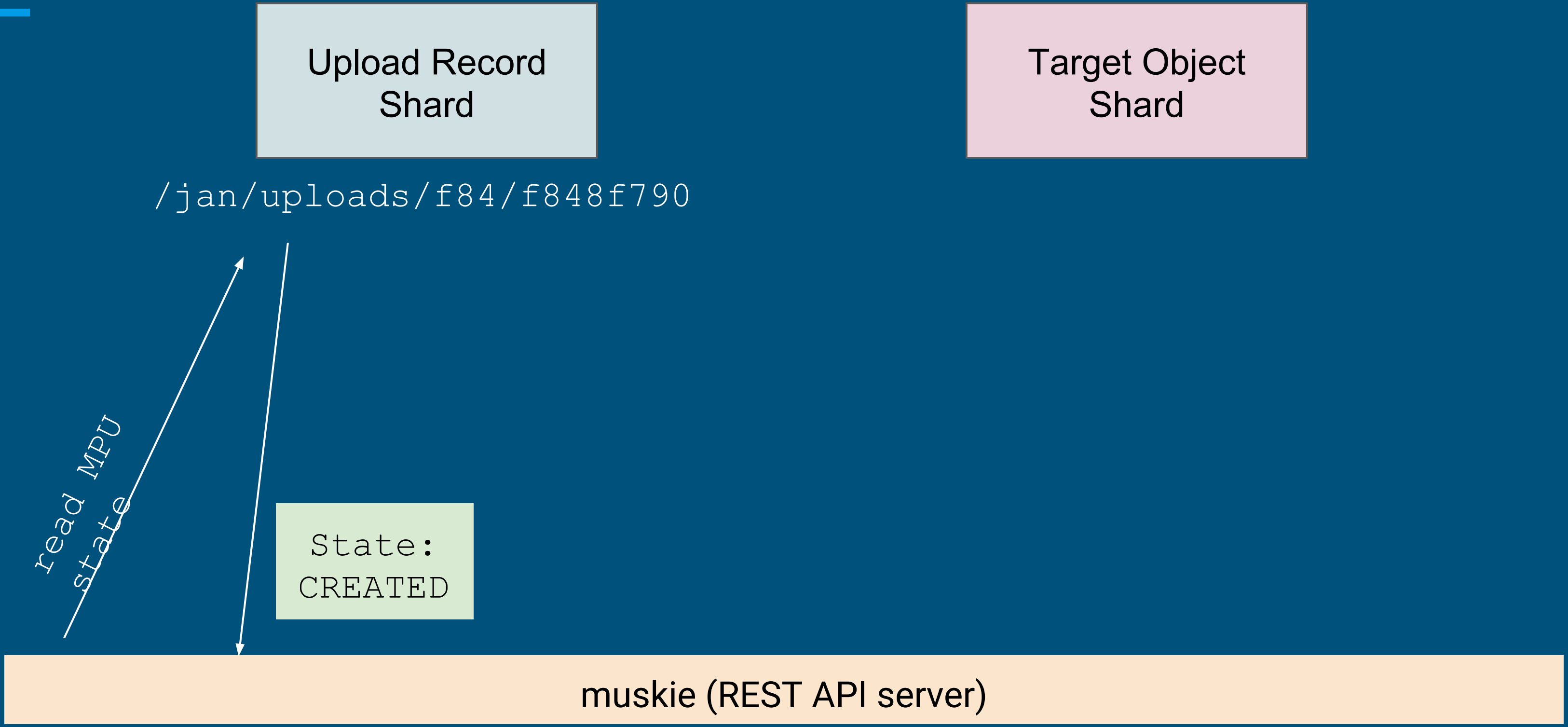
Multipart Upload Commits: Idempotency


Upload Record Shard
Target Object Shard
/jan/uploads/f84/f848f790
/jan/stor/big-object
f848f790:/jan/uploads/f84/f848f790
read MPU state
State: CREATED
write MPU state
State: FINALIZING, type: commit, parts set: P
write finalizing record & object record
OK
muskie (REST API server)
OK

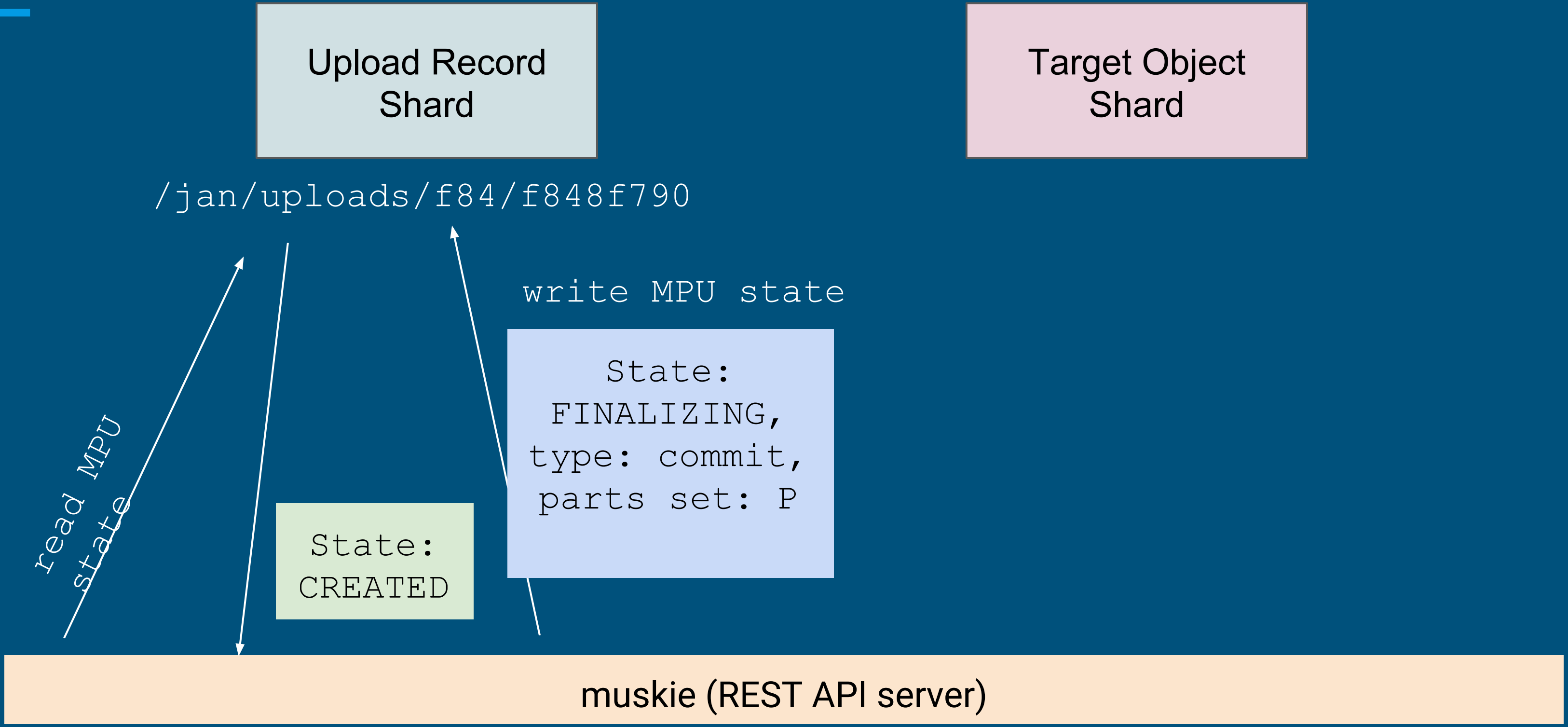
Multipart Upload Commit



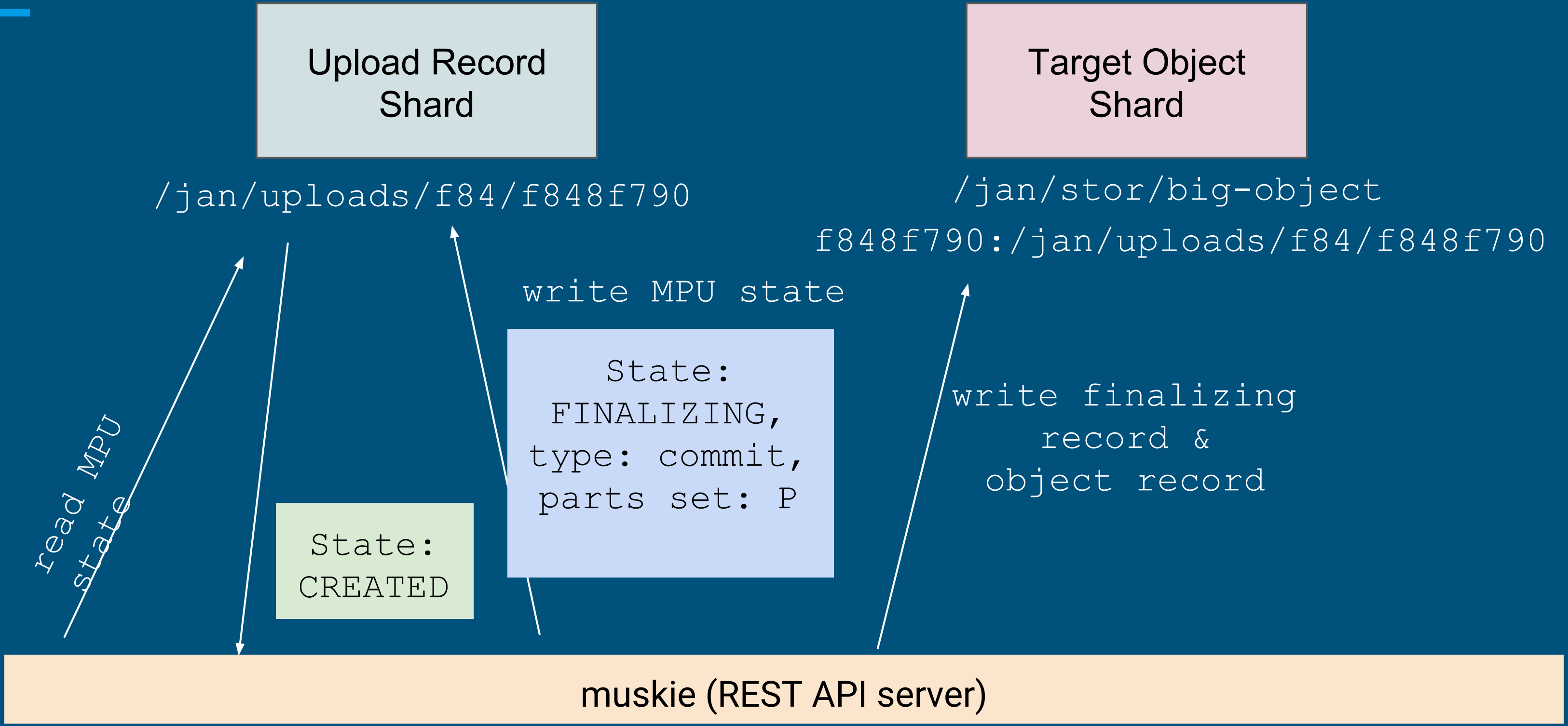
Multipart Upload Commit



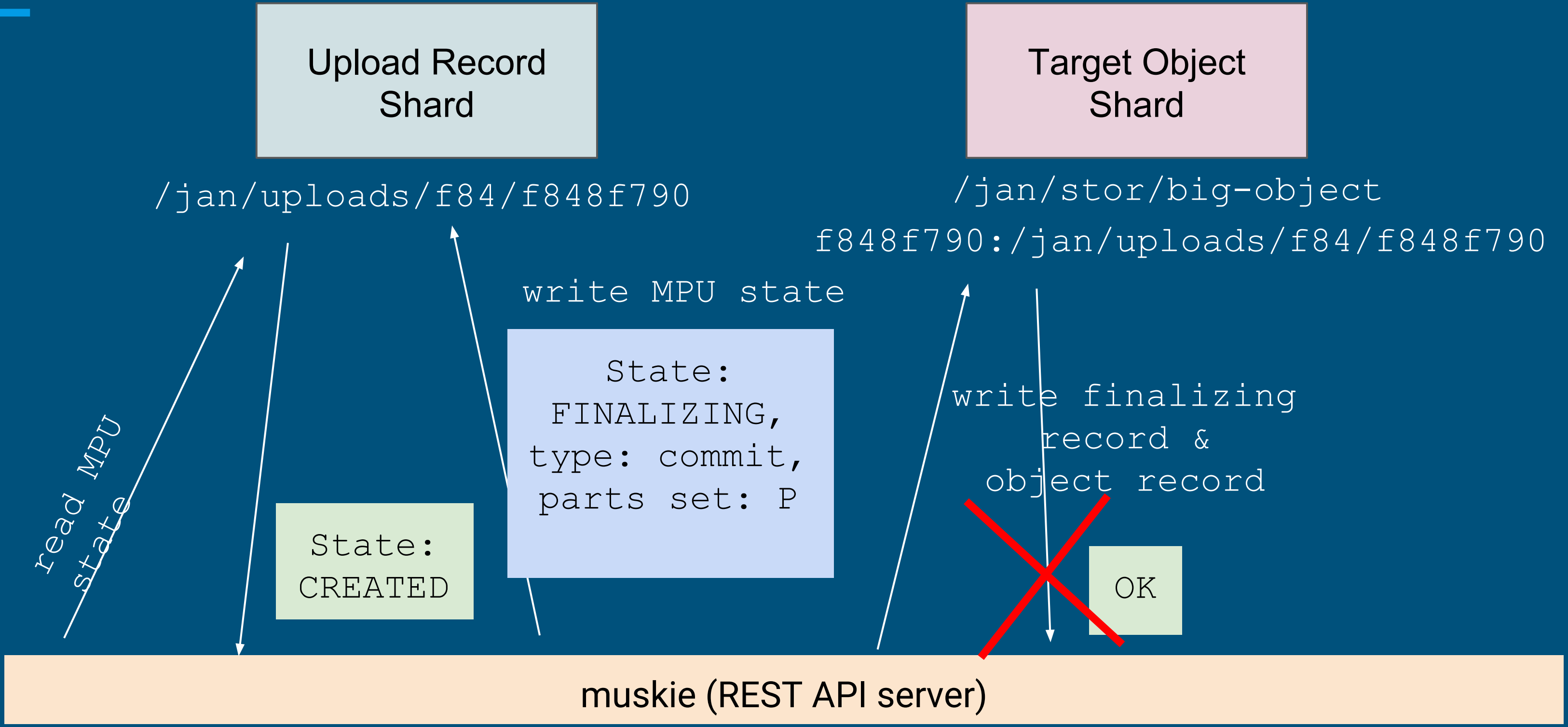
Multipart Upload Commit



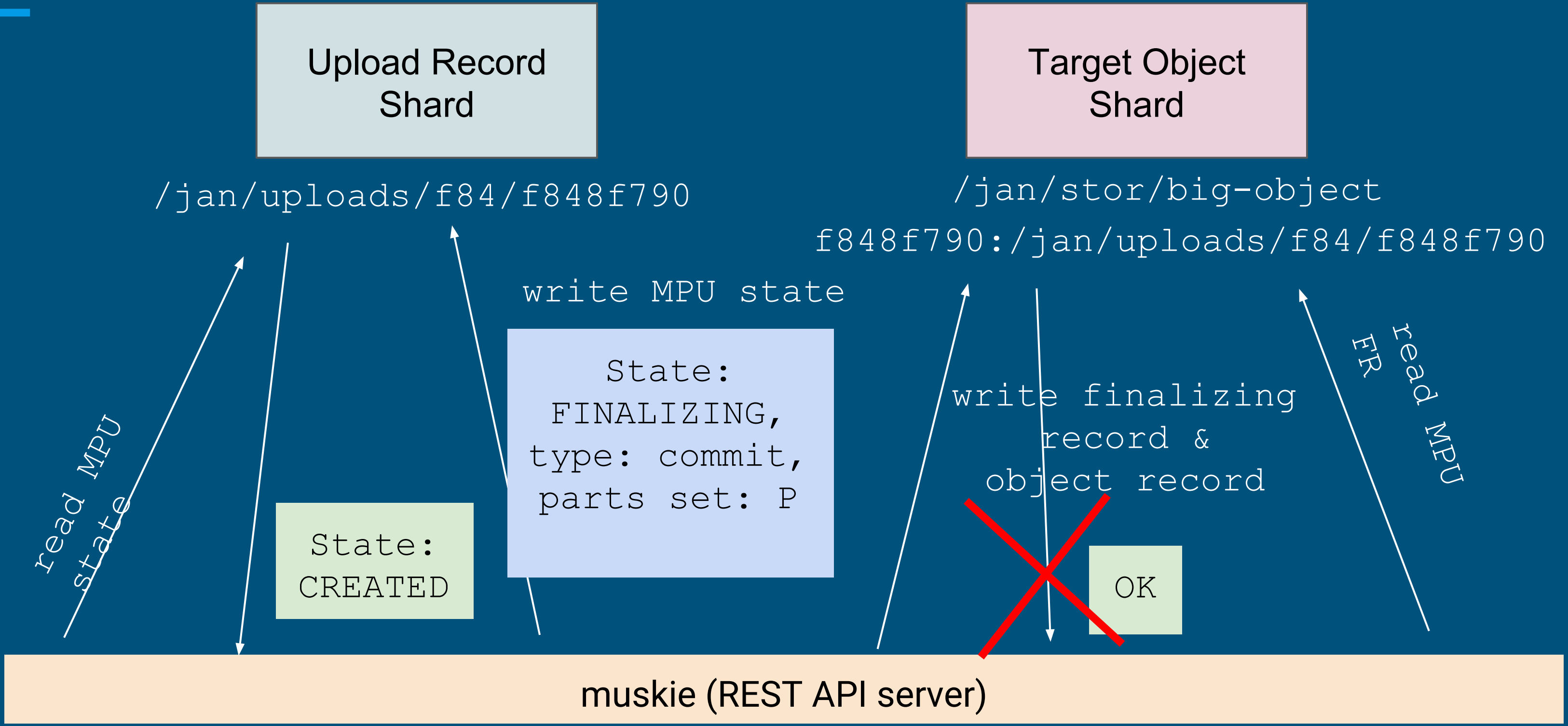
Multipart Upload Commit


Upload Record Shard
Target Object Shard
/jan/uploads/f84/f848f790
/jan/stor/big-object
f848f790:/jan/uploads/f84/f848f790
read MPU state
write MPU state
State: CREATED
State: FINALIZING, type: commit, parts set: P
write finalizing record & object record
muskie (REST API server)

Multipart Upload Commit

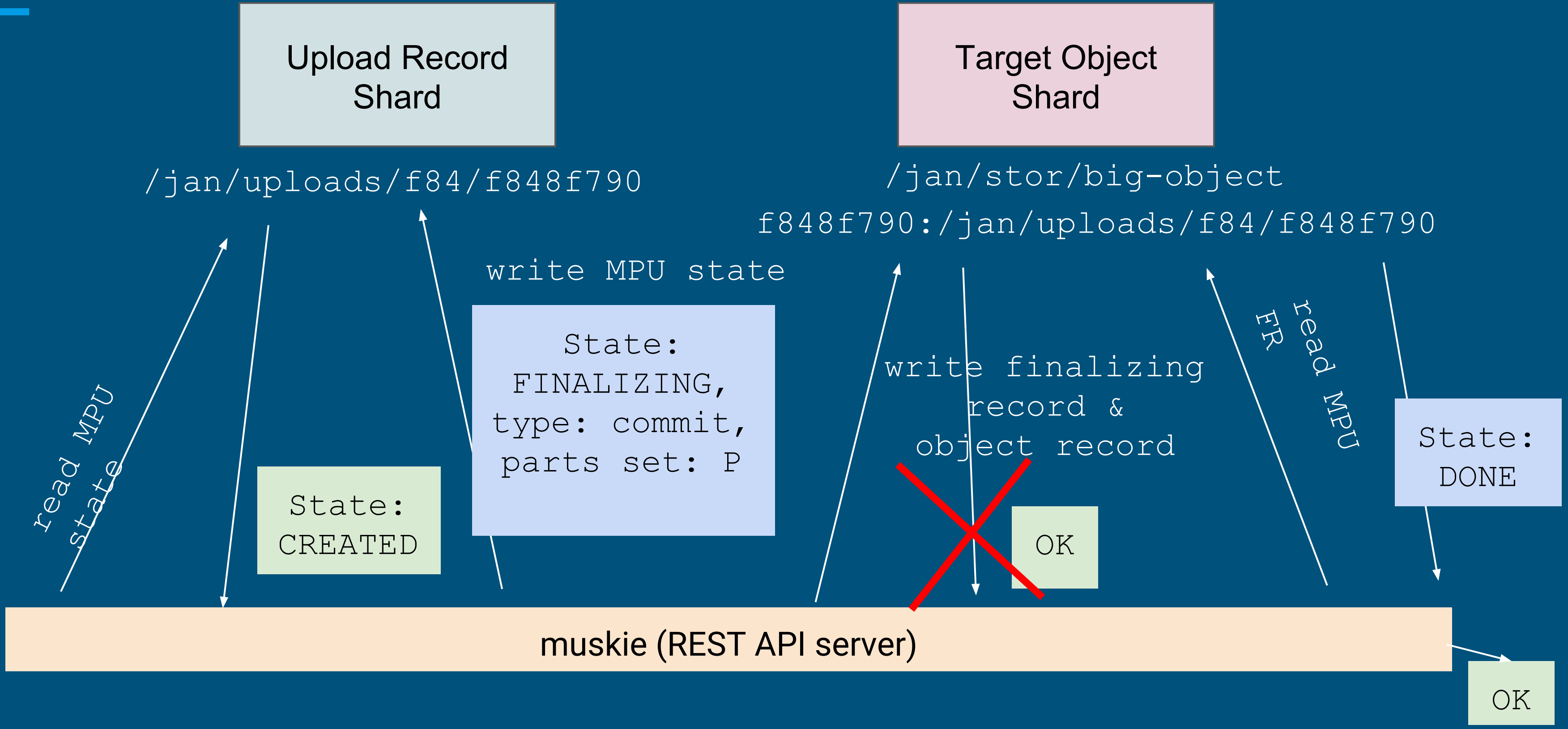

Upload Record Shard
Target Object Shard
/jan/uploads/f84/f848f790
/jan/stor/big-object
f848f790:/jan/uploads/f84/f848f790
read MPU state
write MPU state
State: CREATED
State: FINALIZING, type: commit, parts set: P
write finalizing record & object record
OK
muskie (REST API server)

Multipart Upload Commit

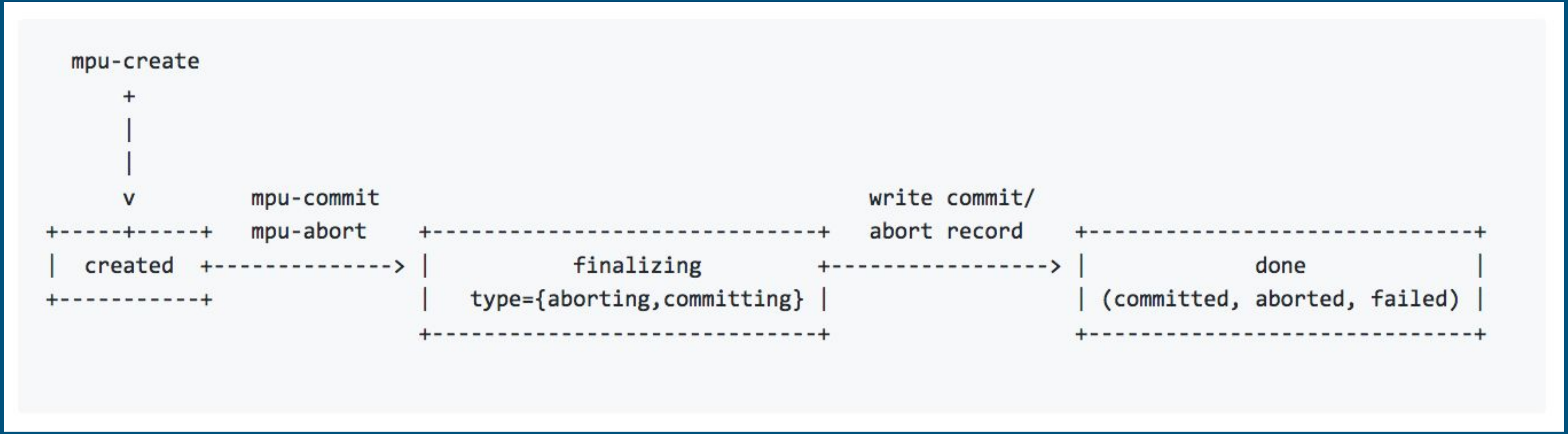


Upload Record Shard
Target Object Shard
/jan/uploads/f84/f848f790
/jan/stor/big-object
f848f790:/jan/uploads/f84/f848f790
read MPU state
State: CREATED
write MPU state
State: FINALIZING, type: commit, parts set: P
write finalizing record & object record
OK
read MPU FR
muskie (REST API server)

Multipart Upload Commit



MPU State Machine



```
  mpu-create
      +
      |
      |
      v         mpu-commit                                      write commit/
+-----+-----+   mpu-abort    +------------------------------+   abort record    +------------------------------+
|  created  +---------------> |          finalizing          +-----------------> |             done             |
+-----------+                |   type={aborting,committing} |                   | (committed, aborted, failed) |
                             +------------------------------+                   +------------------------------+
```

Commits: Final Steps

- Read upload directory record and check MPU’s state.
  - If state is CREATED, verify etags of all parts.
    - If the etags are valid, update the state on the upload record to FINALIZING, type commit.
    - If the etags aren’t valid, return an error.
  - If state is FINALIZING, type commit, then verify the part etags match the MD5 etag summary stored on the upload directory record. If they don’t, return an error.
  - If state is FINALIZING, type abort, return an error.
- Invoke mako-finalize on the storage node set.
- Atomically insert a finalizing record and the target object record on the shard of the target object.
- Return a response indicating success to the client.

Concurrency in other MPU operations

- `mpu-create`: creating a multipart upload
  - Returns a unique handle for an MPU (no contention on target object path)
- `mpu-get`: get the state of an MPU
  - Metadata read
- `upload-part`: upload a part to a given MPU
  - Can overwrite parts as often as you want (consistent with Manta PUT behavior)
  - Race between updating a upload record state to “finalizing” and inserting a new part record
    - Doesn’t lead to any server-size inconsistency, but likely indicates a buggy use of the API
- `abort-mpu`: abort an upload
  - Same concurrency protections as `commit-mpu` (`mako-finalize` is not invoked)

Revisiting Design Constraints

- Multipart upload as a feature started with its own set of constraints
  - Atomic commits/aborts
  - Sane listing of parts & uploads
- … but was constrained to invariants of the system it was designed for
  - Immutable objects
  - Separation between metadata and storage
  - No support for cross-shard transactions
  - Composed of distributed services that can fail unexpectedly

Working with Design Constraints

- **Immutable objects:** mutating objects was not a possible solution
- **Separation between metadata and storage:** separate mechanisms of maintaining correctness of metadata layer and storage layer
- **No support for cross-shard transactions:** use only one shard as the final source of truth for the state of an MPU
- **Composed of distributed services that can fail unexpectedly**: consider atomicity & idempotency of all operations

Tradeoffs in Design

- `mako-finalize`: a complex, variable-latency operation added to mako (previously only a thin shim on top of nginx)
  - Tradeoff: Variable-length latency hit for `mpu-commit`
  - Alternative: Copying data across the network, which would probably be slower in most cases.
- muskie (REST API service) chooses storage nodes for an MPU’s target object when it is created
  - Tradeoff: Storage nodes selected may not be available when parts are uploaded or the object is committed.

Final Thoughts

- For a complete discussion of MPU design, see RFD 65
- When adding new functionality to a system, consider how it will maintain the invariants of the system (and if it doesn’t, at what cost?)
- Even *legacy* systems can have new and innovative features the original authors never imagined :)
  - Can’t wait for a similar presentation on Manta’s tenth birthday!

Questions?