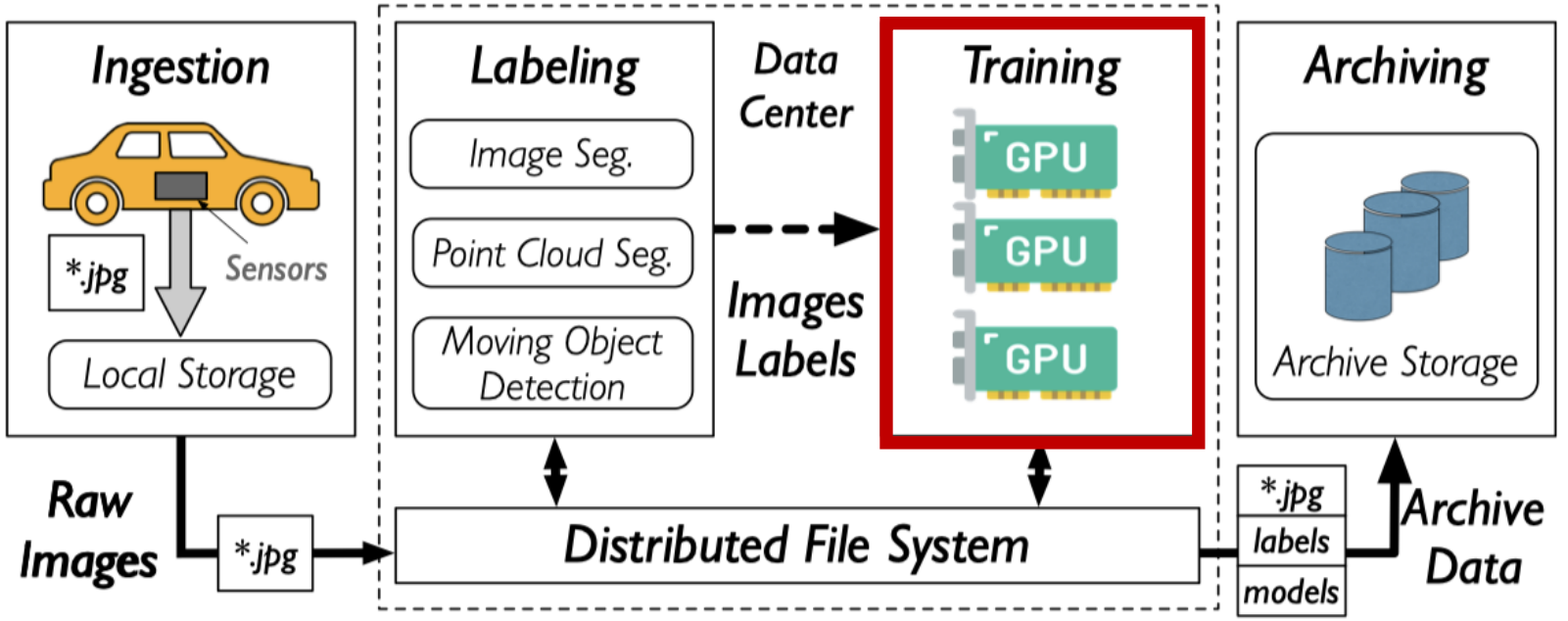
FalconFS: Distributed File System for Large-Scale Deep Learning Pipeline

Jingwei Xu, Junbin Kang, Mingkai Dong, Mingyu Liu, Lu Zhang, Shaohong Guo, Ziyan Qiu, Mingzhen You, Ziyi Tian, Anqi Yu, Tianhong Ding, Xinwei Hu, and Haibo Chen

Institute of Parallel and Distributed Systems (IPADS),
Shanghai Jiao Tong University
Huawei Technologies

Metadata in Deep Learning Training Pipelines



Numerous Small Files

300 billion files, 1 billion directories

~300KB average file size

Poor Locality

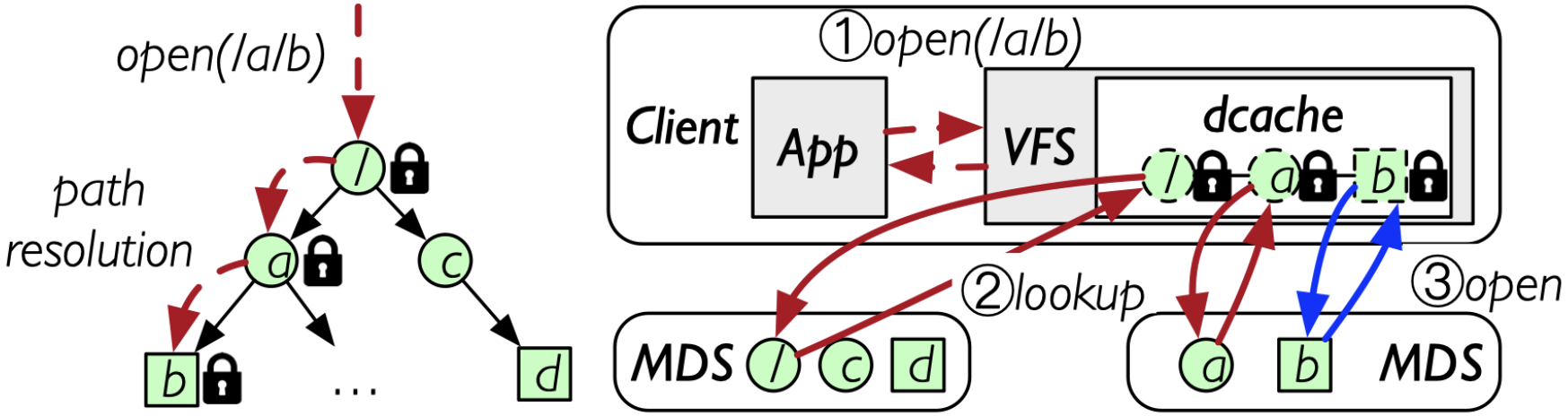
random file access

Challenges to the Underlying Distributed Filesystem

Numerous Small Files ⇒ **Significant Metadata Overhead**

~300KB average file size

~2.4 TB/s throughput ⟶ 8 million files/s



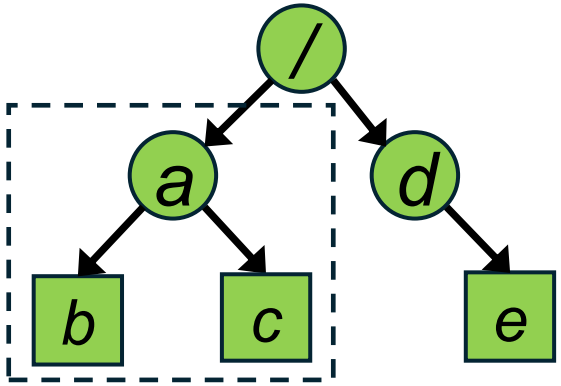
Main Source of Overhead

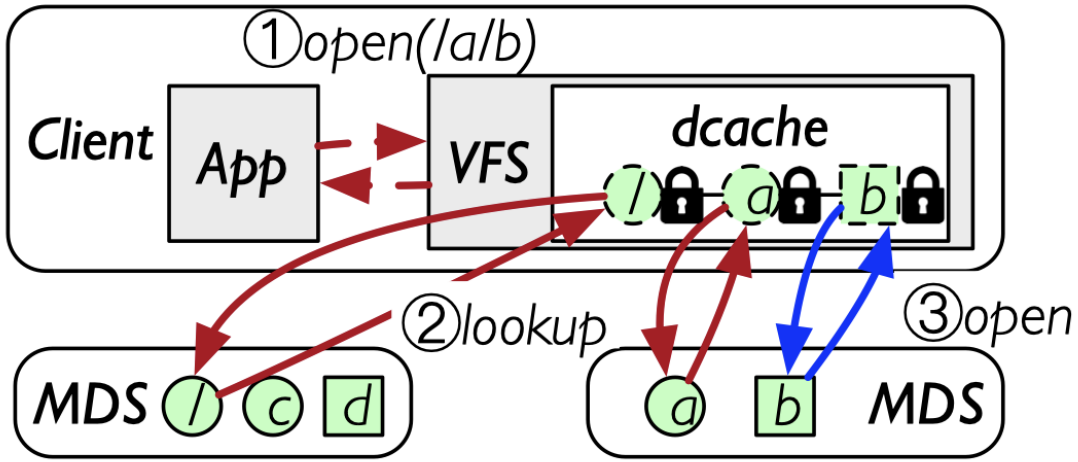
open(/a/b/c) incurs three lookup requests to */*, */a*, and */a/b*

Challenges to the Underlying Distributed Filesystem

Traditional Approach: Client Caching

Every cache hit eliminates one remote lookup





Assumption: Locality of Access

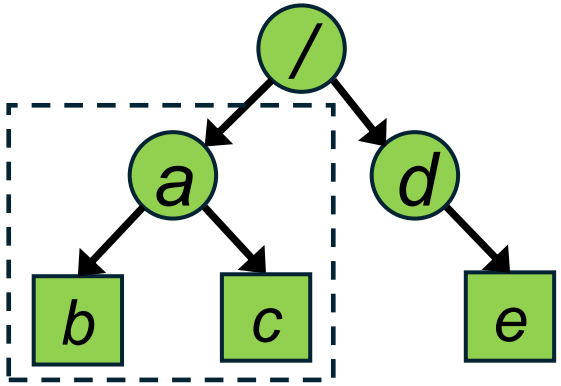
Clients access a small number of directories

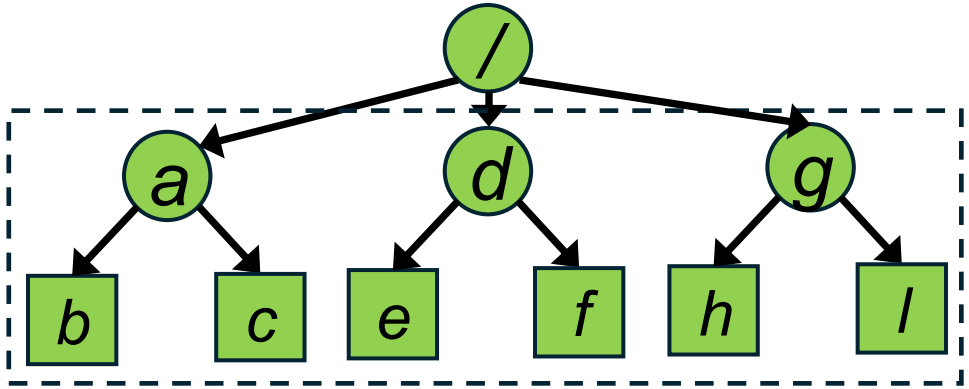
High hit rate with low memory consumption

Challenges to the Underlying Distributed Filesystem

Traditional Approach: Client Caching

Every cache hit eliminates one remote lookup





Assumption: Locality of Access

Clients access a small number of directories
High hit rate with low memory consumption

DL Workload: *Poor Locality*

Training tasks access files in random order
Low hit rate and high memory demand
Huawei ADS: > 100B files, 1B directories

Challenges to the Underlying Distributed Filesystem

Inherent Trade-Off: *Memory Consumption* & *Performance*

Memory Consumption

Cache billions of directories

⇓

Consume TBs of memory

Performance

Random accesses

⇓

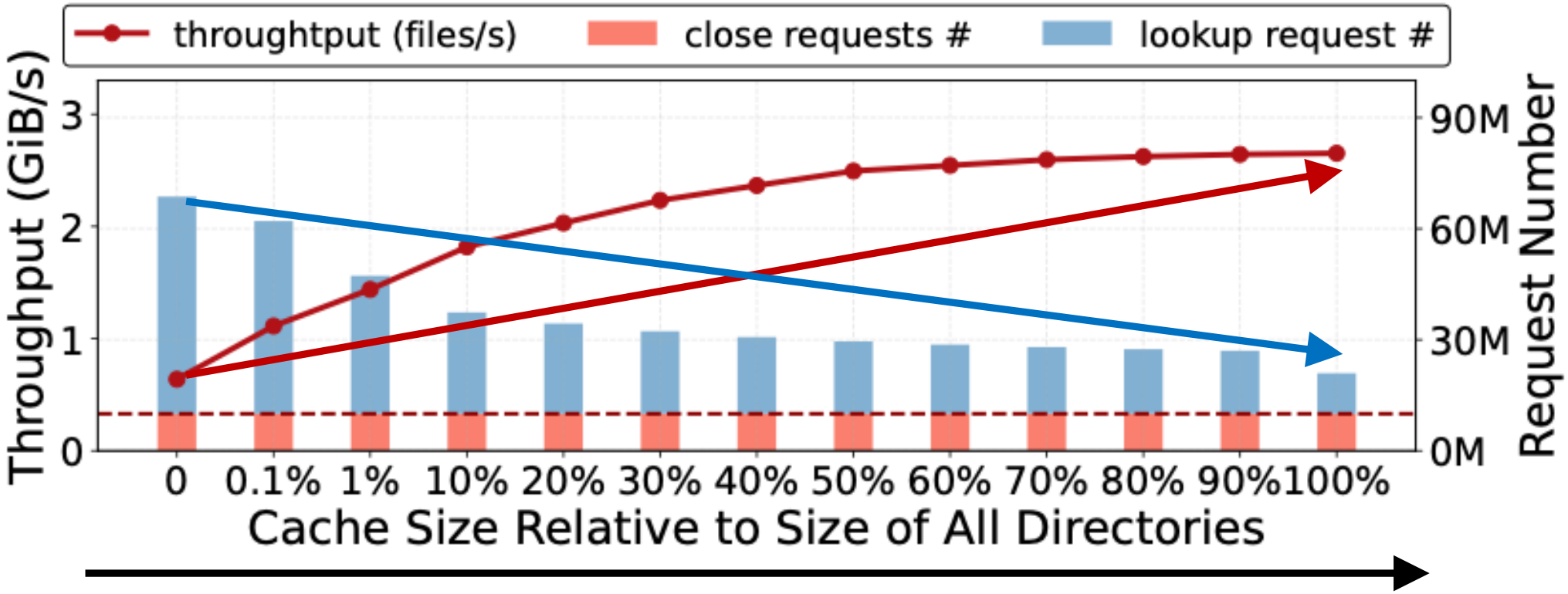
Cache miss rate is proportional to the number of not-cached directories

Client caching cannot solve the path resolution bottleneck

Challenges to the Underlying Distributed Filesystem

Inherent Trade-Off: *Memory Consumption* & *Performance*

Randomly read 10 million 64KB files in 1 million directories



Throughput Increases

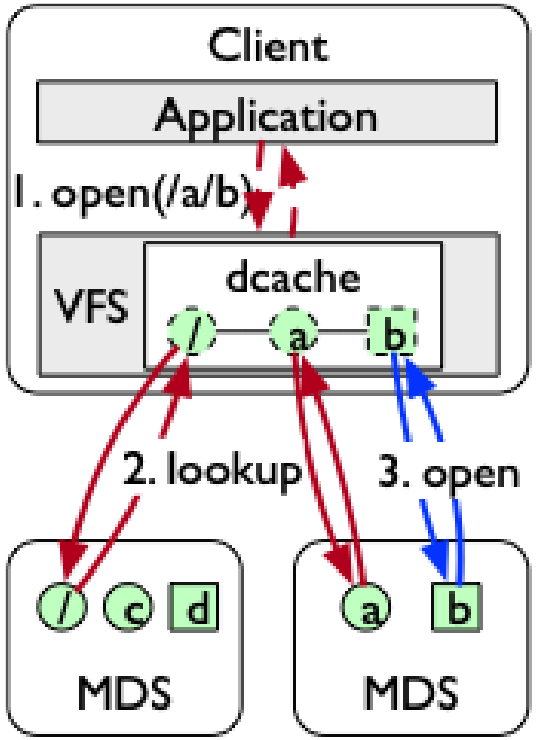
Request # Reduces

Cache Size Increases

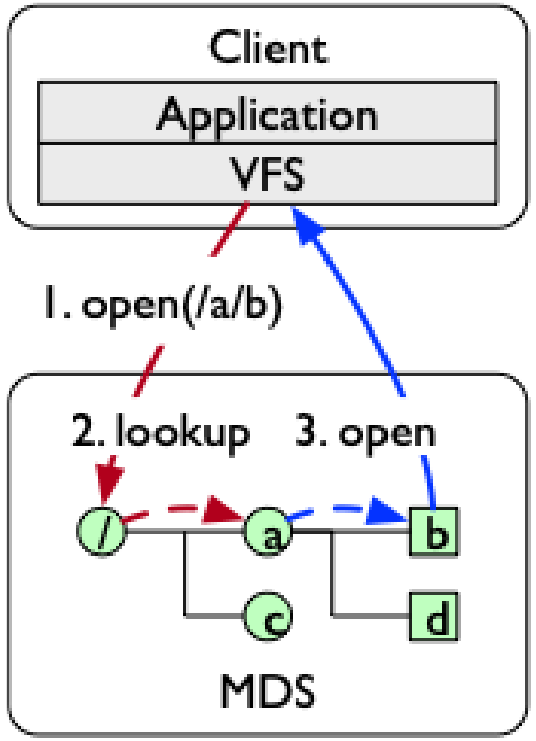
Our Solution

Key idea: make the client ***Stateless*** and offload path resolution to the servers

Client-Side Path Resolution



Server-Side Path Resolution



Why Stateless Client Architecture is Helpful

Client-Side Path Resolution

- Thousands of client nodes
- Kernel objects in VFS
- Colocate with training tasks

⇒

- Highly duplicated cache content
- Large object (800 bytes)
- High memory contention

Server-Side Path Resolution

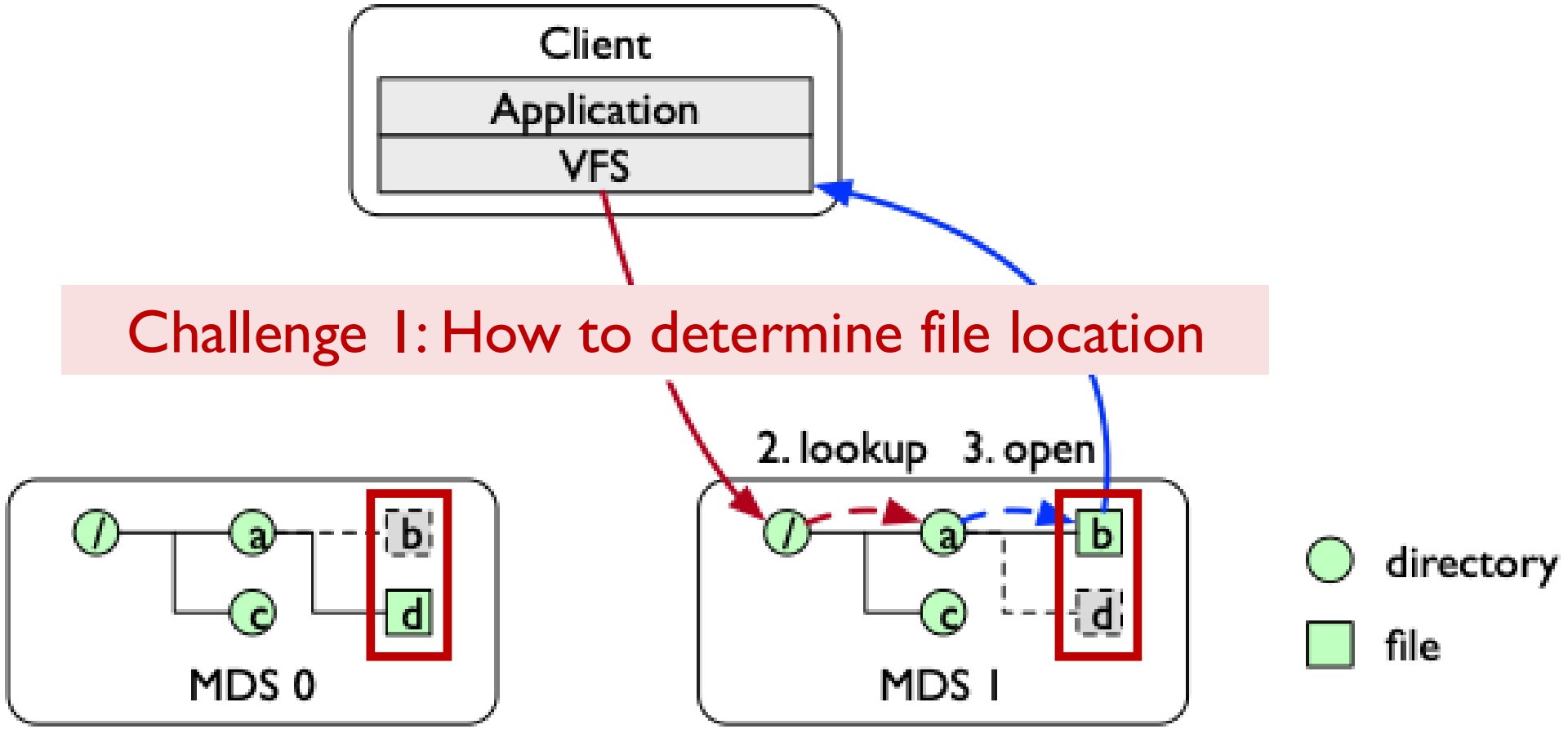
- Typically < 64 server nodes
- Customized metadata objects
- Standalone server processes

⇒

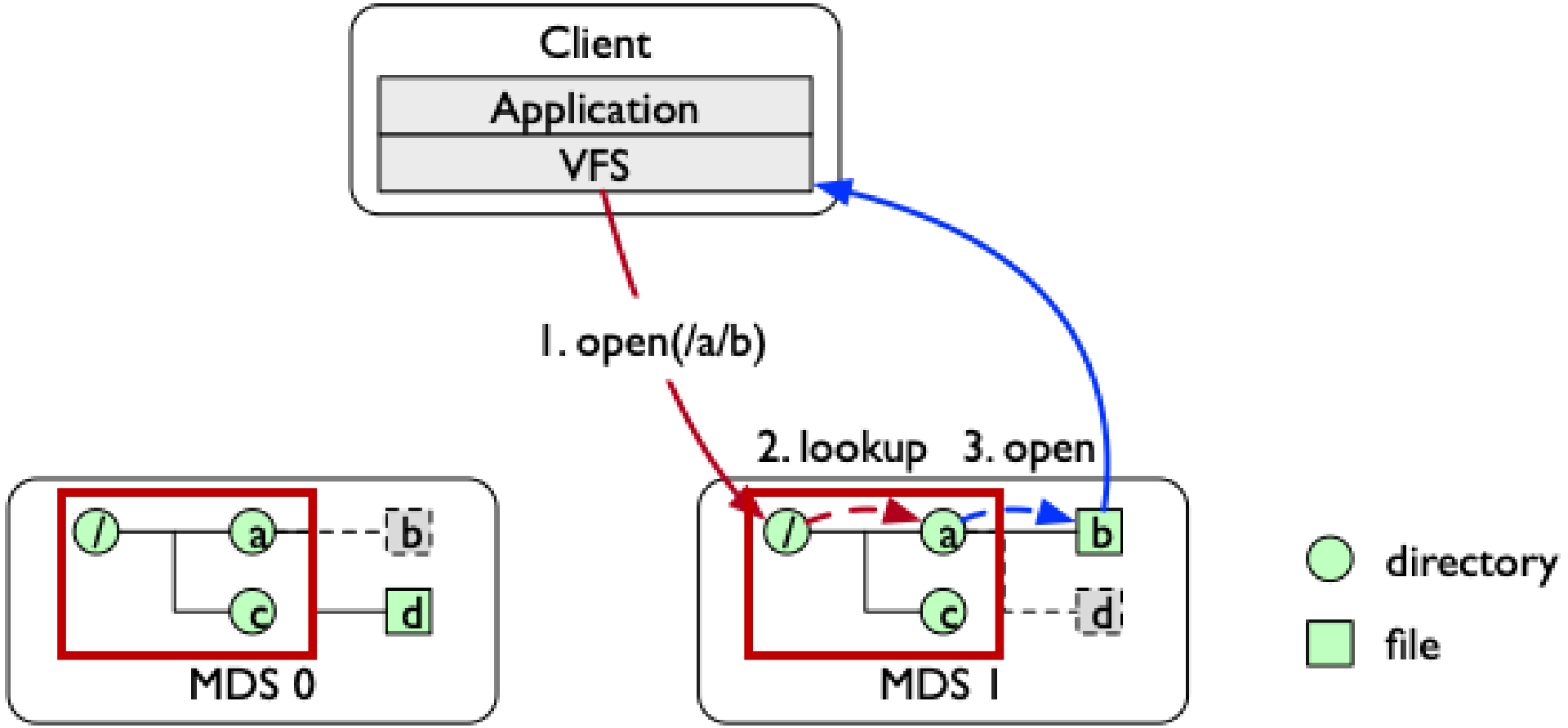
- Smaller total cache size
- Smaller objects (~100 bytes)
- Relatively abundant memory resources

Memory efficiency is higher on the server side

Challenges in Building the Stateless Client



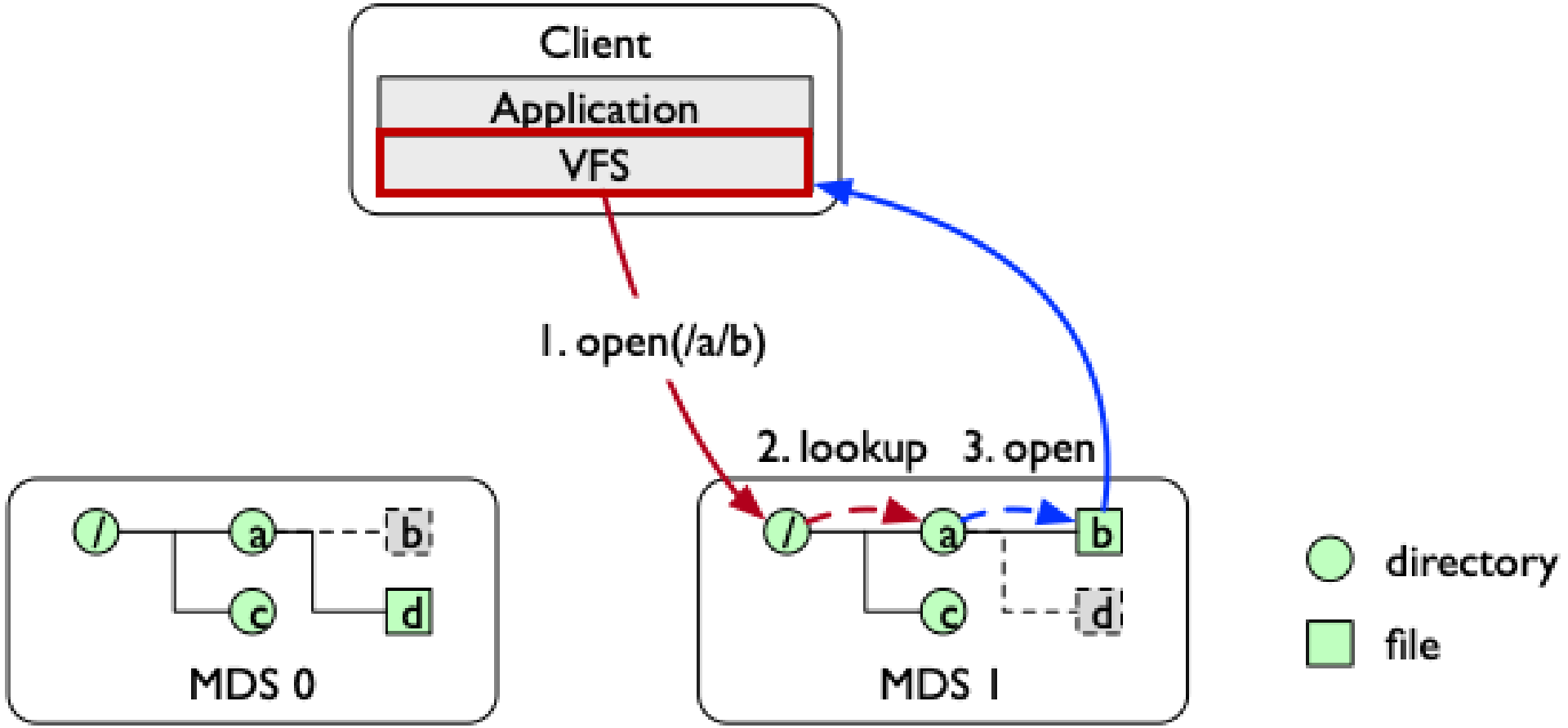
Challenges in Building the Stateless Client



Challenge 2: How to efficiently replicate the namespace

Challenges in Building the Stateless Client

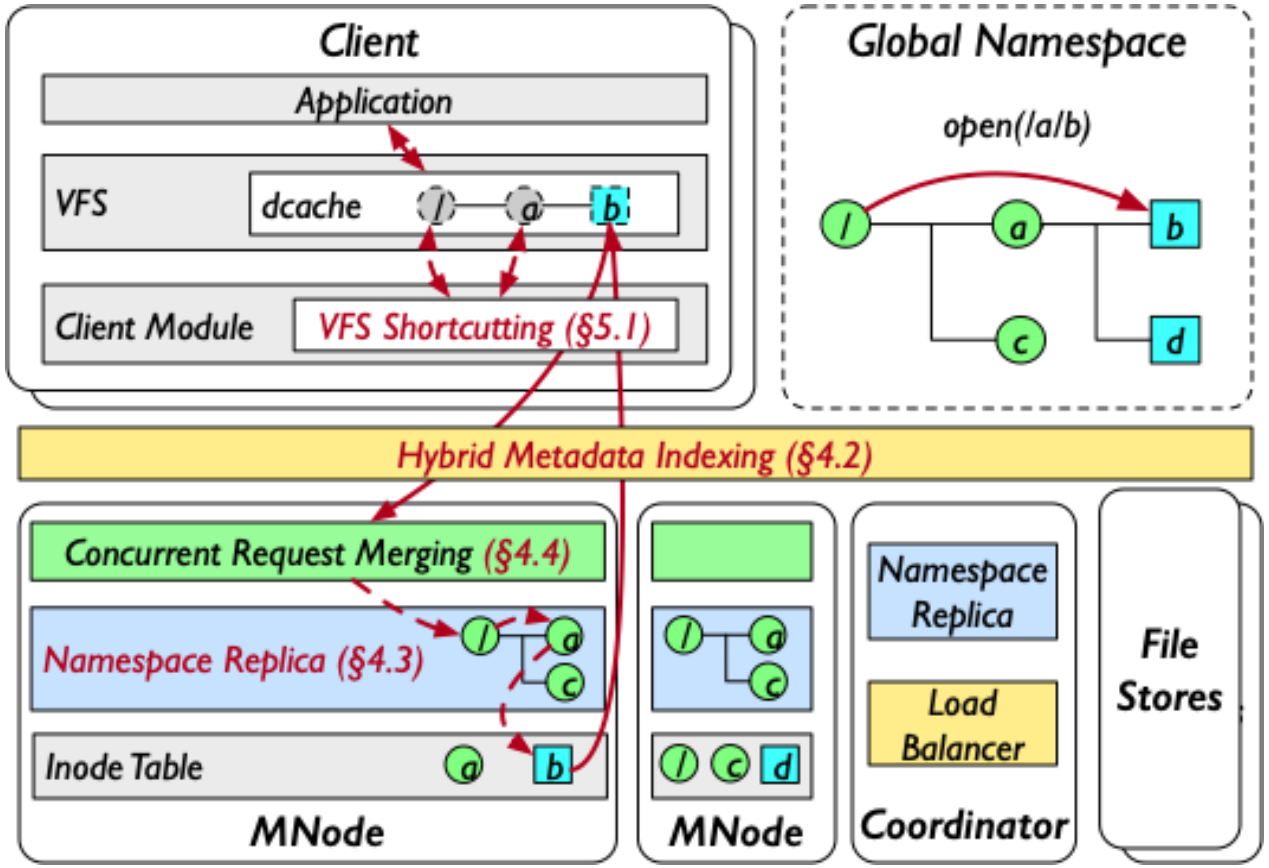
Challenge 3: How to preserve VFS compatibility



FalconFS Overview

Four Components:

- Client + MNode + Coordinator + File Stores



Metadata Schema:

	Key	Value	Organization
dentry	pid, name	id, perm	replicate
inode	pid, name	id, attr	partition

Namespace (dentry) replicated

Attributes (inode) partitioned

How to Locate File Inode?

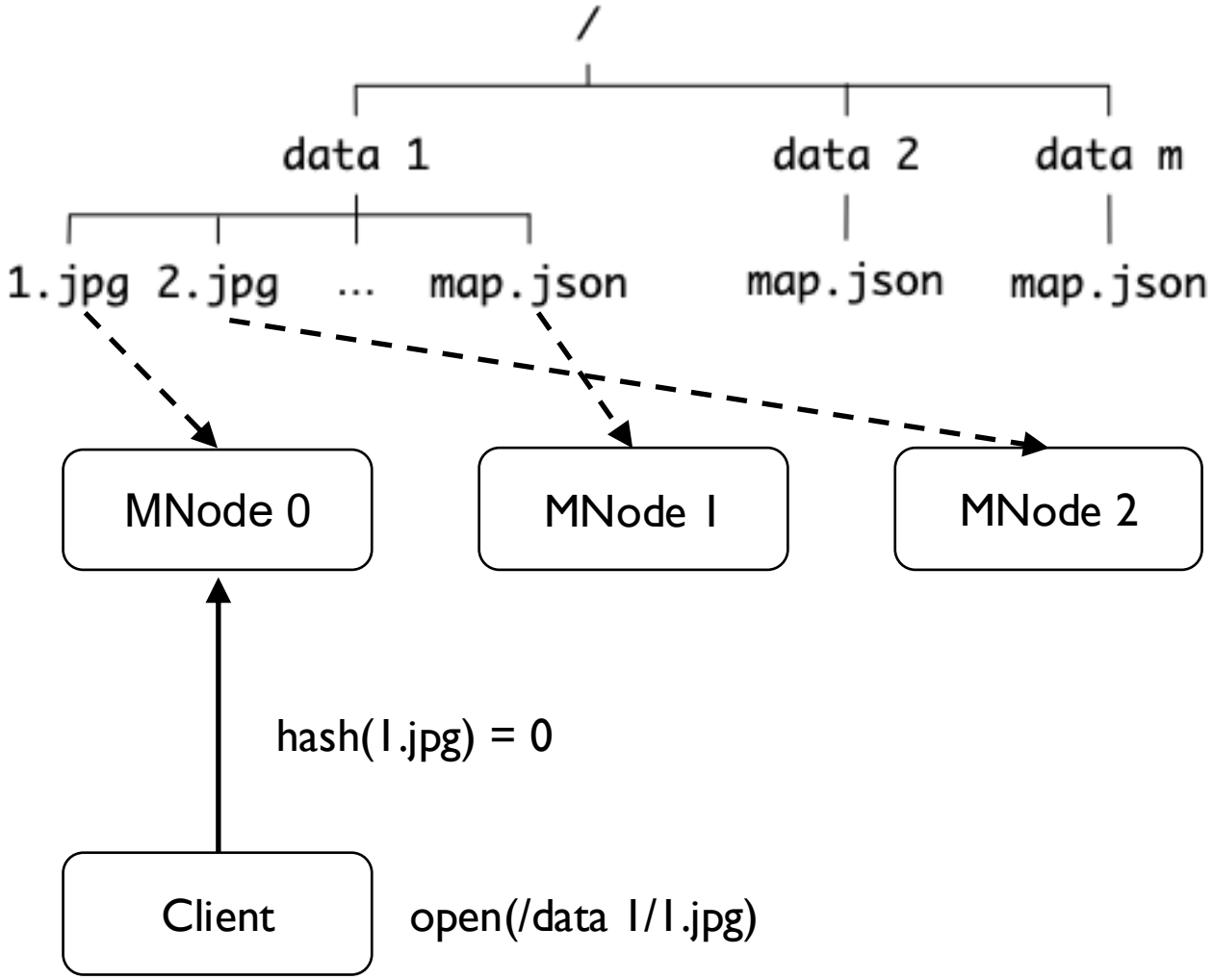
Common path: hash by filename

+

Directory size is typically far larger than the number of servers

⇓

Most files are distributed evenly 😄



How to Locate File Inode?

Common path: hash by filename

+

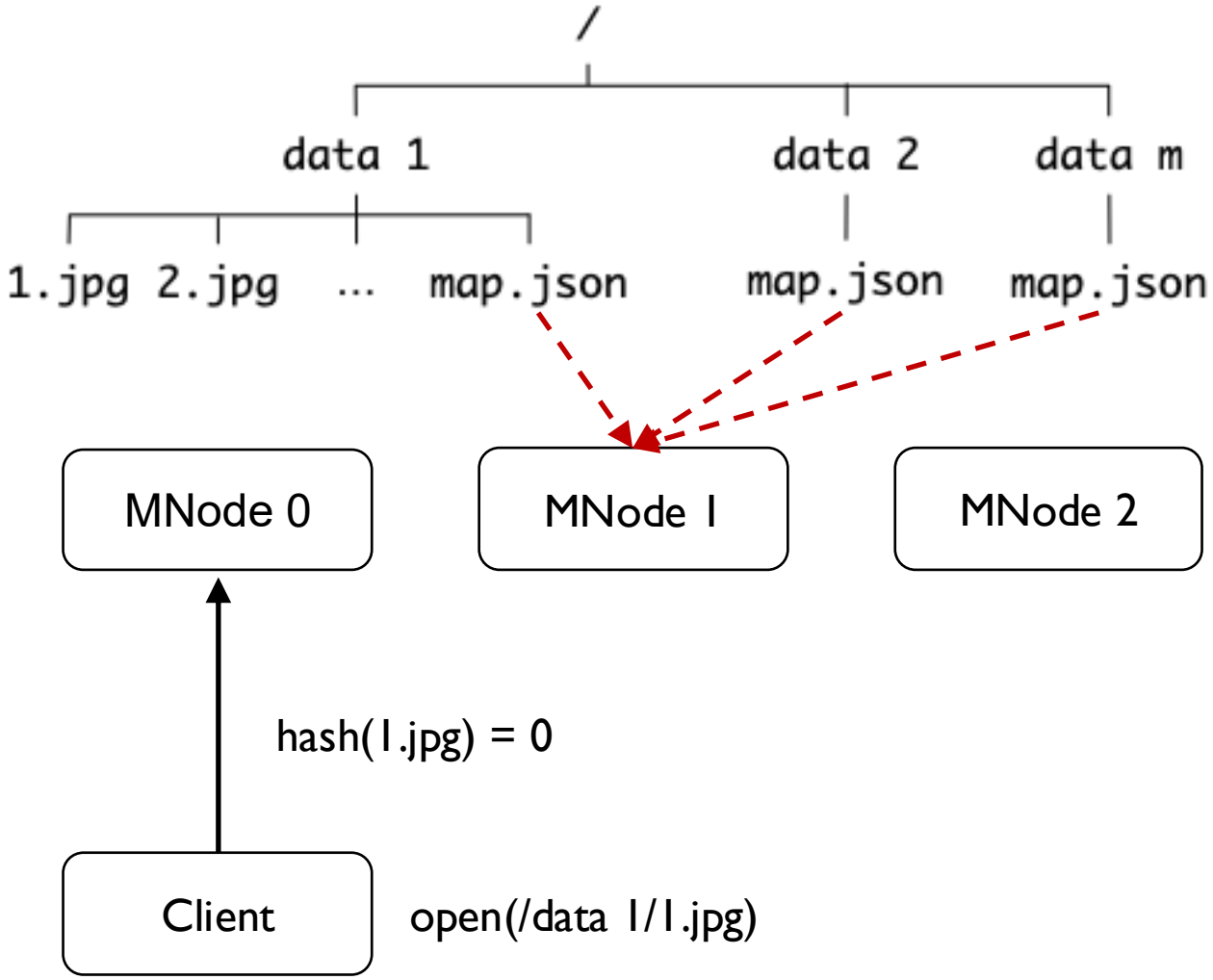
Directory size is typically far larger than the number of servers

⇓

Most files are distributed evenly 😄

But

Same-name files can become hotspots ☹

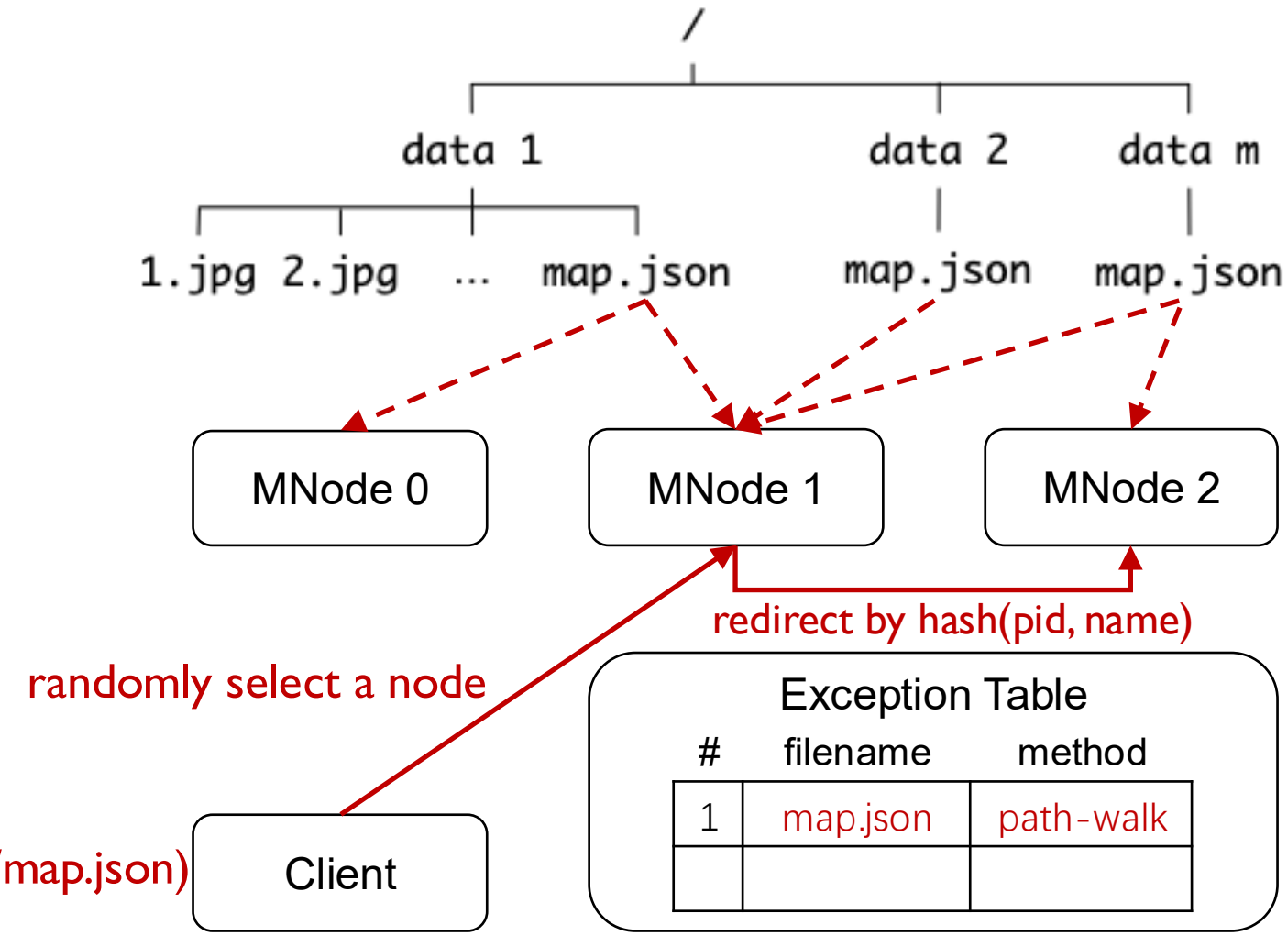


How to Locate File Inode?

Corner-case: selective redirection

Rule 1: path-walk redirection

- place inode by hash(pid, name)



How to Locate File Inode?

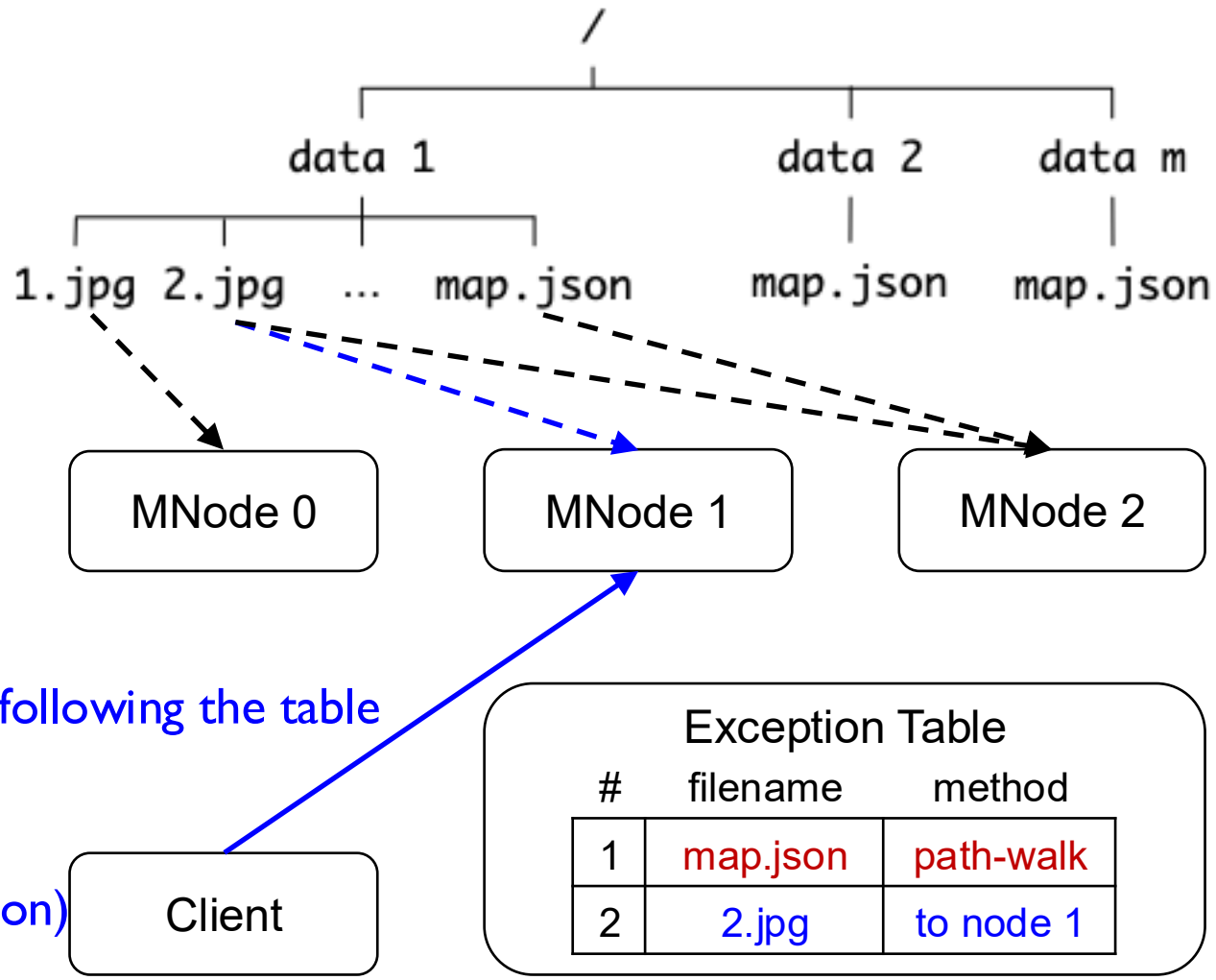
Corner-case: selective redirection

Rule 1: path-work redirection

- place inode by (pid, name)

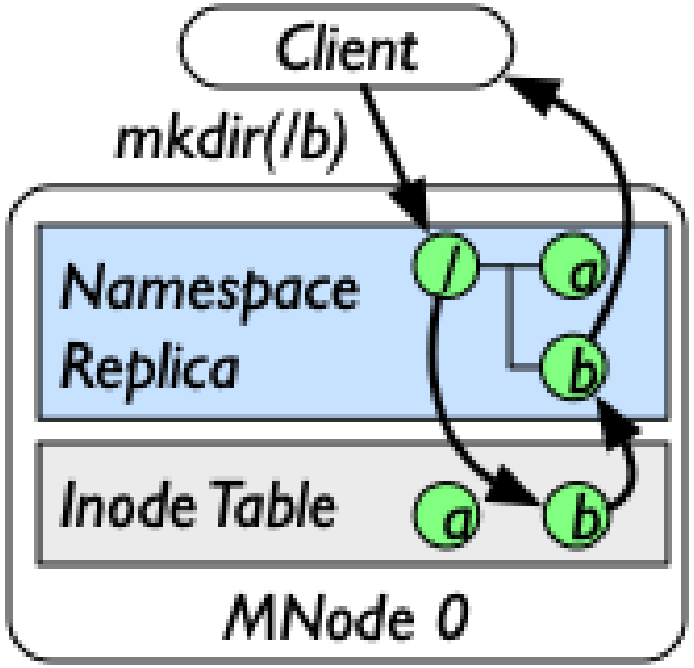
Rule 2: overriding redirection

- place inode according to table entry

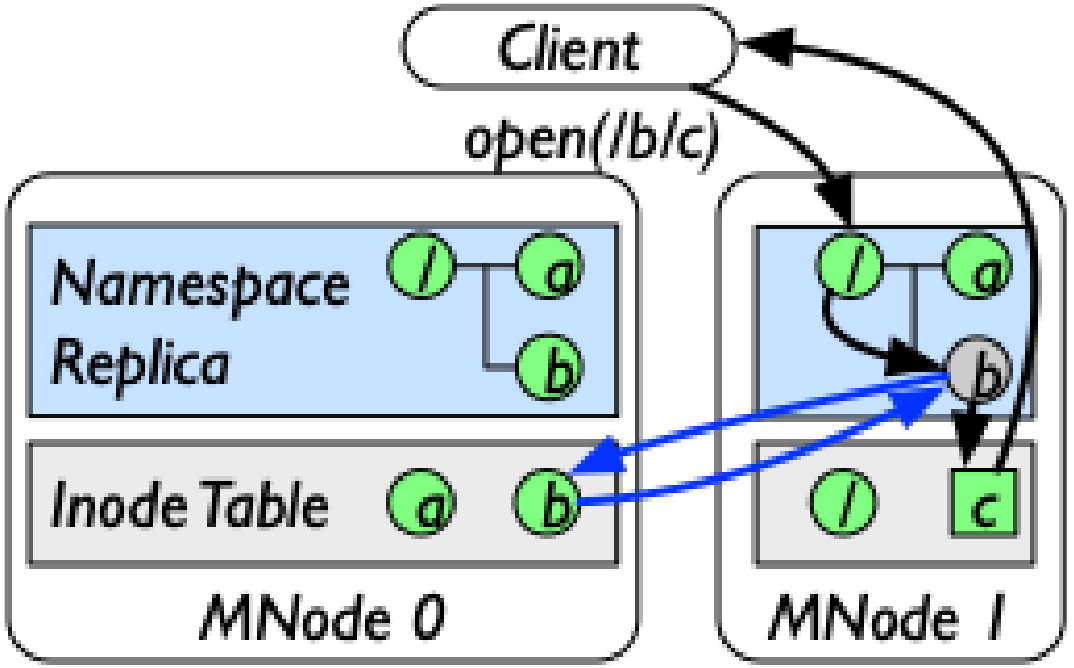


How to Manage Server-Side Namespace Replicas?

Create on a single node

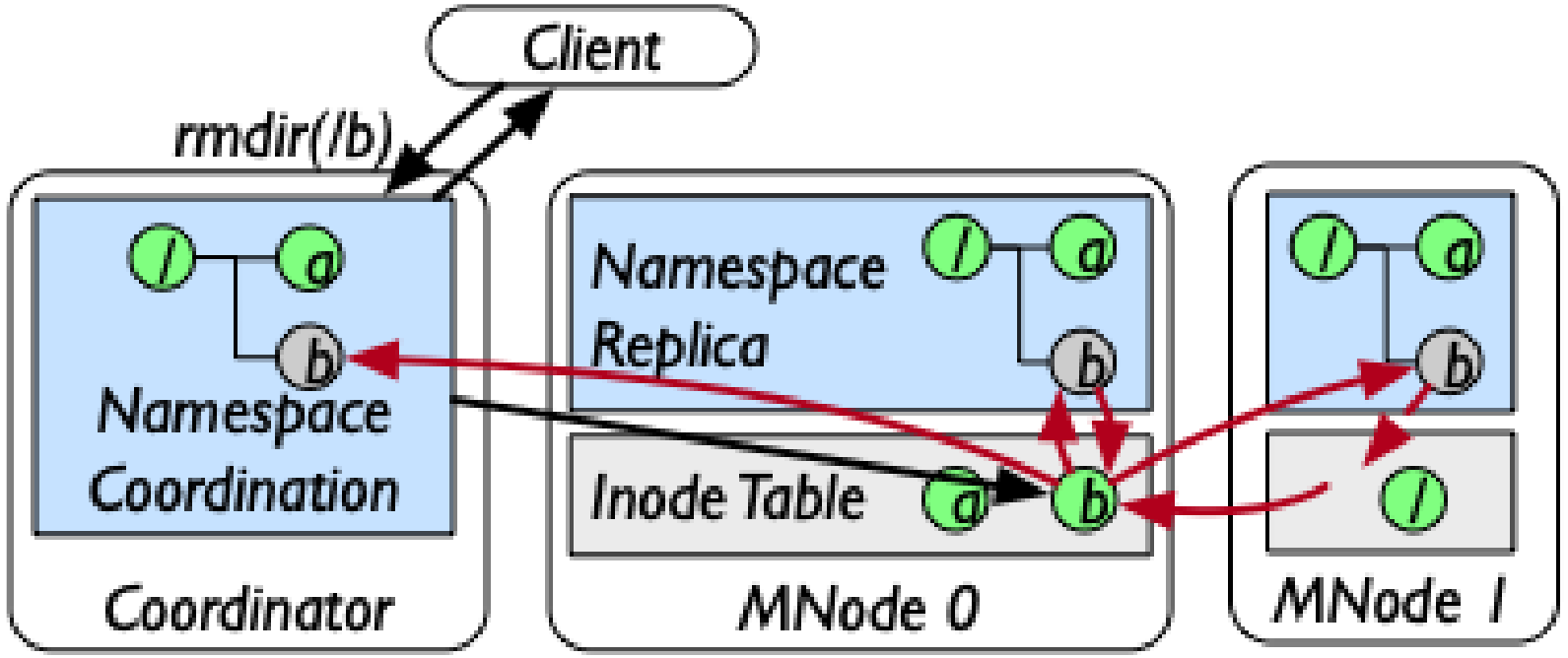


Replicate on lookup



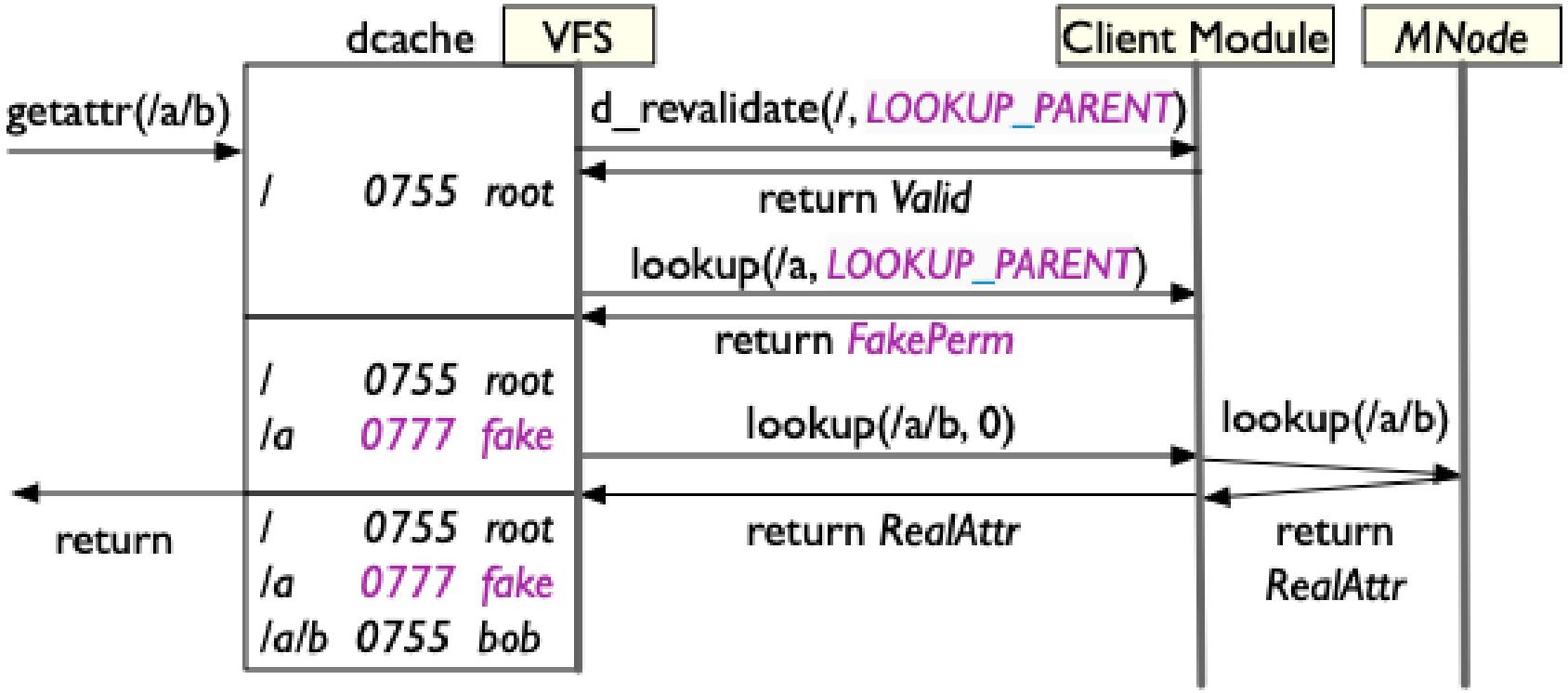
How to Manage Server-Side Namespace Replicas?

Invalidate replicas when updating the namespace



How to Implement VFS Compatibility?

- Distinguish lookups to the *intermediate* and the *last* components by leveraging *an existing flag*
- Return *FakePerm* for the intermediate lookups
- Construct the full path and perform remote lookup for the last lookups



Evaluation Setup

Hardware Platform

- 16 server nodes; 10 client nodes
- Each server node uses 4 cores

CPU	Intel Xeon Gold 5317 3.00GHz, 12 cores
Memory	8 × DDR4 2933MHz 16GB
Storage	Intel Optane Persistent Memory
Network	ConnectX-5 100GbE

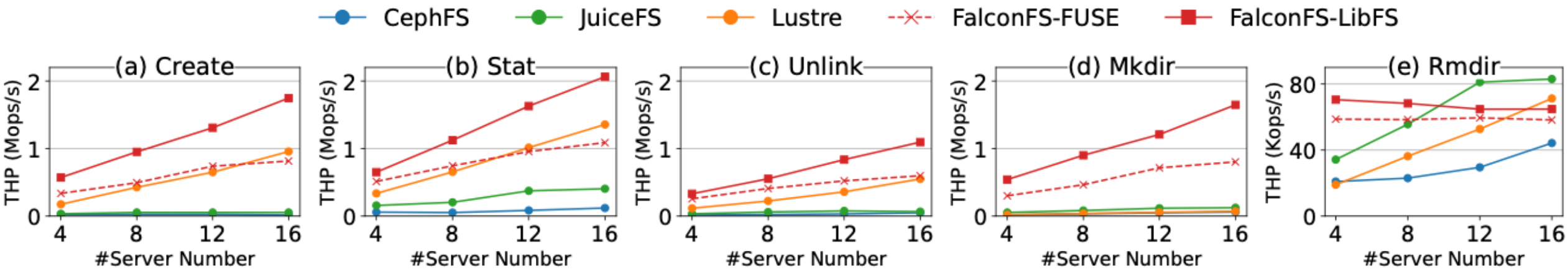
Compared Systems

- CephFS 12.2.13, JuiceFS 1.2.1, and Lustre 2.15.6
- Data and metadata replication disabled

Metadata Throughput

- Each client accesses its own private directories
- All directory lookups hit the (client-side) cache
- Saturate the performance by increasing the client thread number

Better

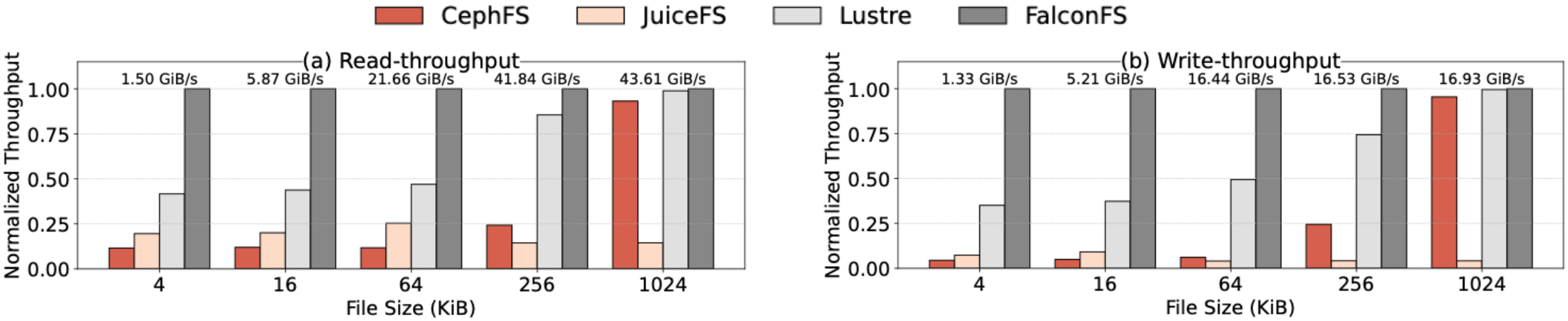


FalconFS has good basic metadata performance

Data Throughput

- Each client accesses 1024 files in its own private directories
- Saturate the performance with up to 2560 client threads
- Bypass page cache with O_DIRECT flag

Better



FalconFS's metadata advantage lead to better small file access performance

Load Balance in Real Workloads

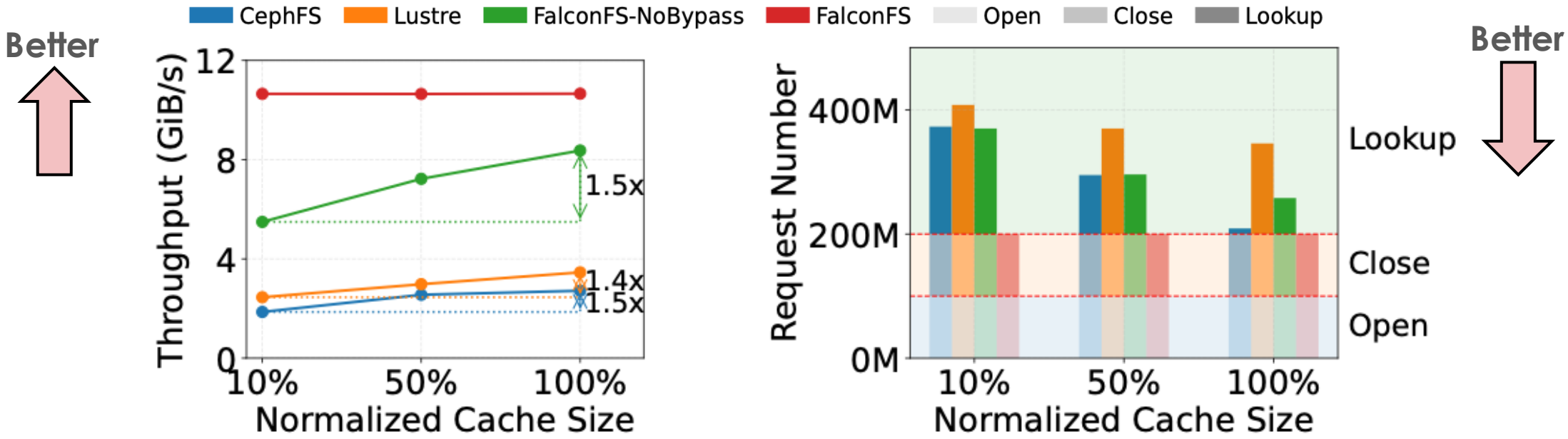
- Distribute the directory tree over 16 nodes
- In Linux-6.8 code, "Makefile" and "Kconfig" are path-walk redirected, accounting for 5.55% of all files in total
- In FSL homes, the most frequent filename accounts for 1.24% of all files

	inode #	inode distribution max	inode distribution min	exception entry # path-walk	exception entry # overriding
Labeling task	33320	6.99%	5.30%	0	0
ImageNet [17]	2027728	6.29%	6.21%	0	0
KITTI [13]	15003	7.01%	5.47%	0	0
Cityscapes [11]	20022	6.30%	6.22%	0	0
CelebA [27]	202599	6.54%	6.95%	0	0
SVHN [33]	33404	6.77%	5.76%	0	0
CUB-200-2011 [45]	12003	6.68%	5.95%	0	0
Linux-6.8 code	88936	6.49%	5.96%	2	0
FSL homes [41]	655177	6.83%	5.45%	1	0

In real workloads, FalconFS usually enjoys balanced load without redirection

Performance with Constrained Memory

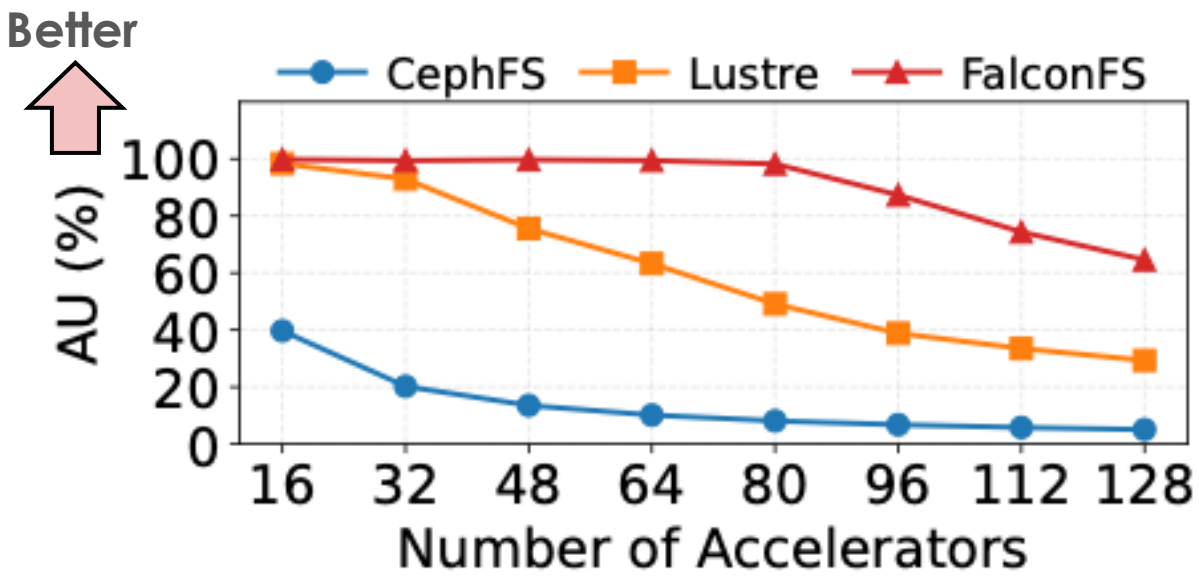
- Limit memory available to DFS client via cgroup v2 / CephFS config
- Randomly access 100 million files in 11.1 million directories

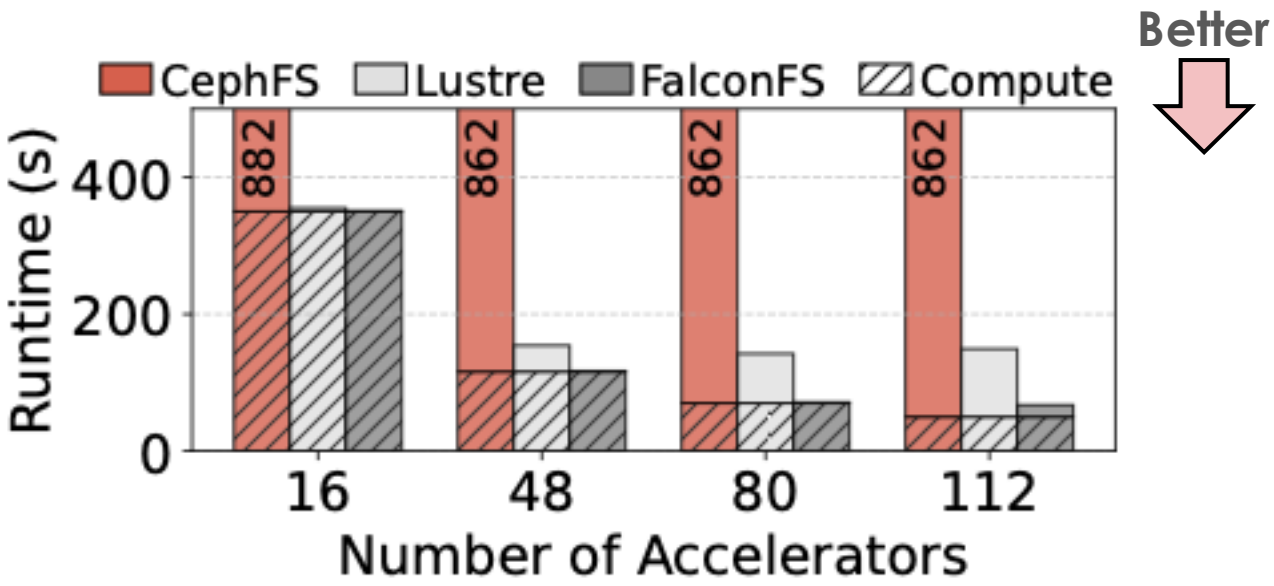


FalconFS has stably high performance when memory resources are constrained

End-to-End Evaluation

- Simulate training a ResNet-50 model
- Training set: 10 million 112 KiB files, in 1 million directories





FalconFS has stably high performance when memory resources are constrained

Conclusion

Problem

- Traditional path resolution architecture is a bottleneck for random file access

Key Techniques of FalconFS

- Hybrid metadata indexing
- Lazy namespace replication
- VFS shortcutting

Results

- Achieves stably high performance in numerous file random access workloads
- Deployed in Huawei ADS, reducing storage cost by >50%

Thanks & QA

FalconFS: Distributed File System
for Large-Scale Deep Learning Pipeline