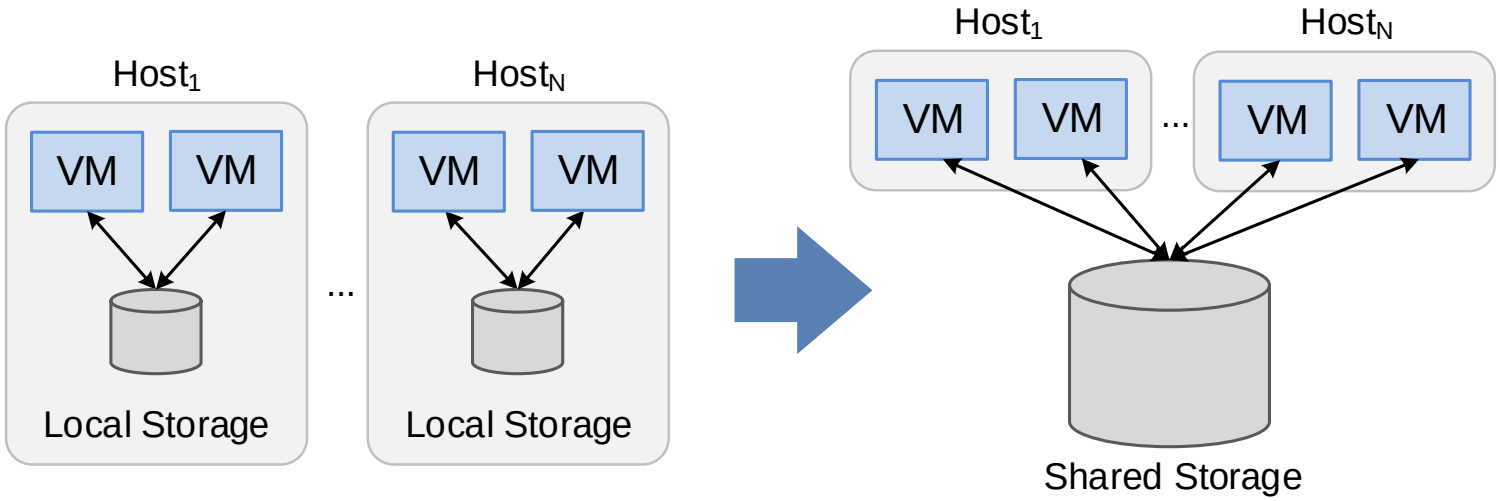
Virtualization Aware Access Control for Multitenant Filesystems

Giorgos Kappes, Andromachi Hatzieleftheriou, Stergios V. Anastasiadis

gkappes, ahatziel, stergios (at) cs.uoi.gr

Department of Computer Science and Engineering
University of Ioannina, Greece

Storage Consolidation



Benefits

- Efficient usage and management of storage resources
- Sharing support
- High capacity utilization
- Reduced cost

Storage Interfaces

Direct virtual disk access through block interface

+ Isolation, snapshoting, versioning, migration
+ Heterogeneous clients
– Semantic loss hardens consistency, sharing, manageability
– Reduced performance due to layering and FS nesting

Direct filesystem sharing through file interface

+ Isolation, snapshoting, versioning, migration
+ Semantic awareness: sharing, consistency, manageability
+ Elimination of layering and nesting: increased performance
– Complicates support for heterogeneous clients

Storage Multitenancy

Goal

- Storage infrastructure shared among different tenants

Requirements

- **Scalability:** Support enormous number of end users
- **Isolation:** Isolate the user identities and access control of different tenants
- **Sharing:** Flexible data sharing within or between tenants
- **Compatibility:** Compatibility with existing applications
- **Manageability:** Flexible resource management

Research focus

- Efficient and secure multitenancy in VM filesystems

Ceph Storage

A distributed and unified storage platform

- Everything is stored in the form of objects
- Scalable, no single point of failure, software-based, flexible

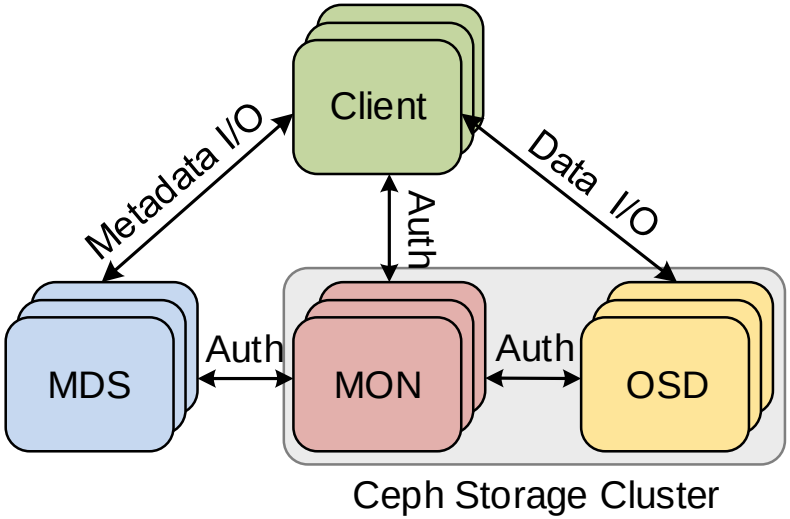
Ceph object store (RADOS)

- A reliable, distributed, autonomous object store

Storage interfaces:

- **Direct access** through librados
- **Object-level (RADOSGW):** HTTP REST gateway
- **Block-level (RBD):** A reliable and distributed block device
- **File-level (CEPHFS):** A POSIX-compliant distributed FS

Ceph FS



A distributed object-based filesystem

- Parallelization of file I/O
- Elimination of the potential metadata bottleneck

Separate management of file metadata and data

- Metadata managed by Metadata Servers (MDS)
- Data managed by Data Servers (OSD)

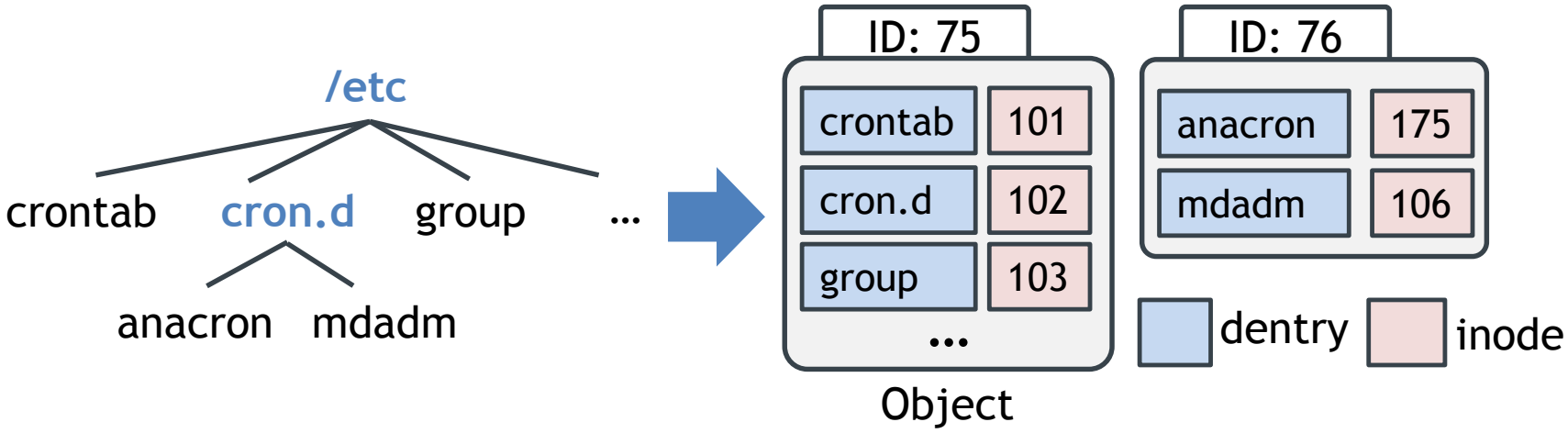
Metadata Server

Goal

- Create filesystem hierarchy on top of objects
- Manage metadata: hierarchy, permissions, timestamps, etc.

Metadata is stored in RADOS

- Inodes are embedded inside folders
- Folders are stored as single objects or as multiple fragments



Metadata Server

Metadata caching and failure recovery

- Recently updated metadata is cached
- The MDS journals metadata updates for failure recovery

Other features

- High availability: Multiple standby MDSs
- Scalability: Multiple active MDSs, subtree partitioning

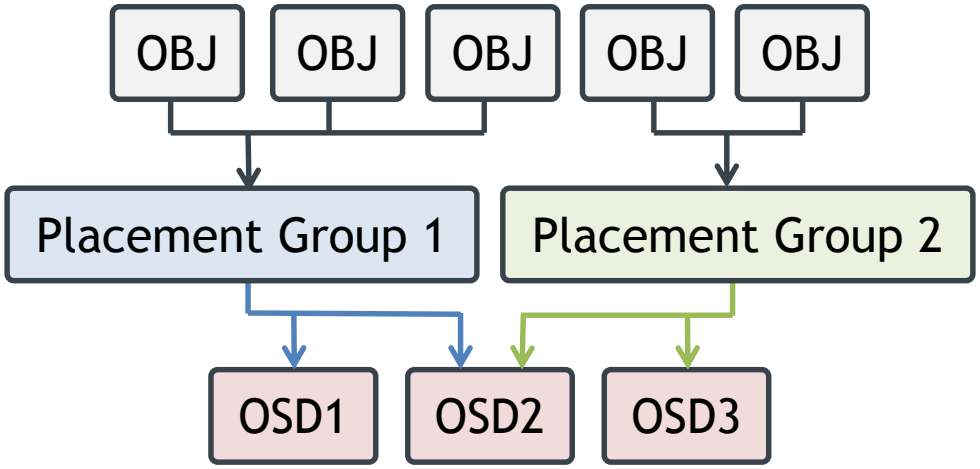
Data Server

Stores file data and metadata in object-form

- Objects: files with identifier, binary data, object metadata

Multiple active OSDs

- Scalability
- Fault tolerance



Data placement

- CRUSH maps objects to PGs and PGs to OSDs
- PGs are logically grouped into pools
- Clients run the CRUSH algorithm themselves

Monitor

Maintains the state of the cluster

- The state is maintained into the cluster map
- Clients check in periodically to obtain a fresh map copy

Cluster map

- Composition of: MON map, MDS map, OSD map, CRUSH map
- The maps track the state of the servers, PGs, storage, etc.
- The cluster map gets updated each time the state changes

Monitor quorum

- Consensus about the cluster map is established with Paxos
- An odd number of monitors is required

Client

Provides Access to the filesystem

- Linux kernel client
- FUSE client

Clients calculate the final position of an object

- **First step:** calculate object id (OID)

OID: INODE number, object fragment number

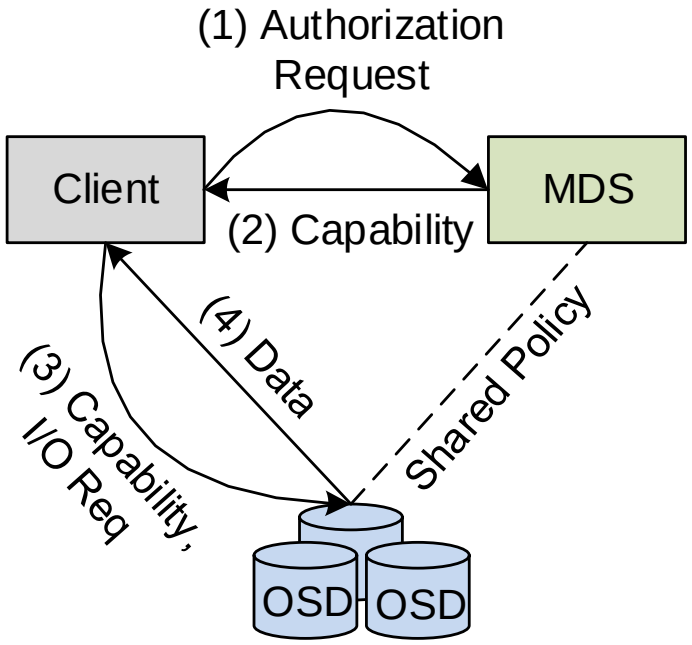
- **Second step:** choose a Placement Group inside a pool

PG: hash of OID, total number of PGs, pool number

- **Third step:** choose the OSDs to store the object

List of OSDs: Run CRUSH to map the PG to OSDs

Access control



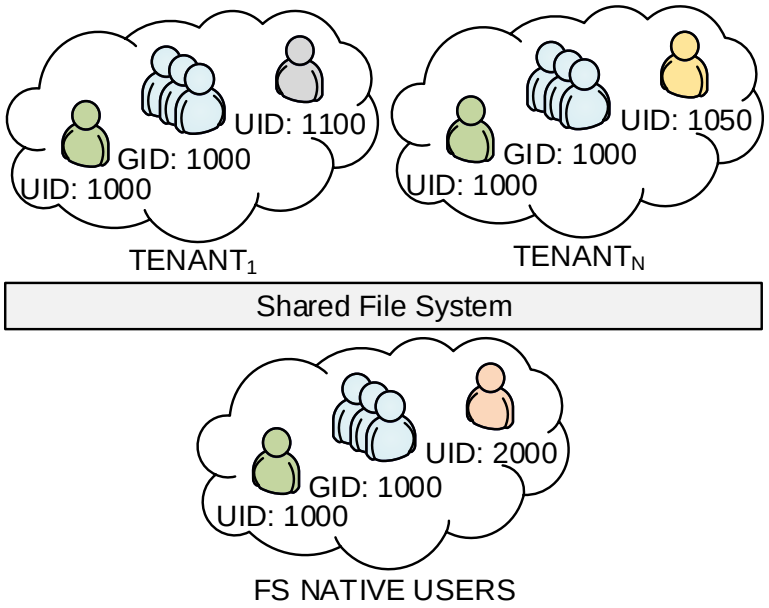
Access control decisions happen at the MDS

- The MDS authorizes a request and provides capabilities to clients
- The clients present the capabilities to OSDs
- The OSDs validate the capabilities

Motivation

Problem of multitenancy

- Shared FS namespace
- Crosstalk between tenants
- Complicated security



Native multitenancy at the filesystem level

- Clean way to isolate multiple tenants
- Shared hardware, operating system, fileservers
- Configurable isolation, sharing, performance, manageability

Prior approaches

Centralized

- The principals' identities of all tenants centrally maintained
  - Poor scalability, isolation and manageability

Peer-to-peer

- The principals of each tenant managed locally
- Tenants communicate to publicize their principals' identities
  - Overhead to periodically synchronize the tenants

Mapping

- Local principal IDs mapped to global unique IDs
  - Mapping overhead, sharing complications, security violations

The Dike Approach

Hierarchical identification and authentication

- The tenants manage their principals
- The provider manages the tenants

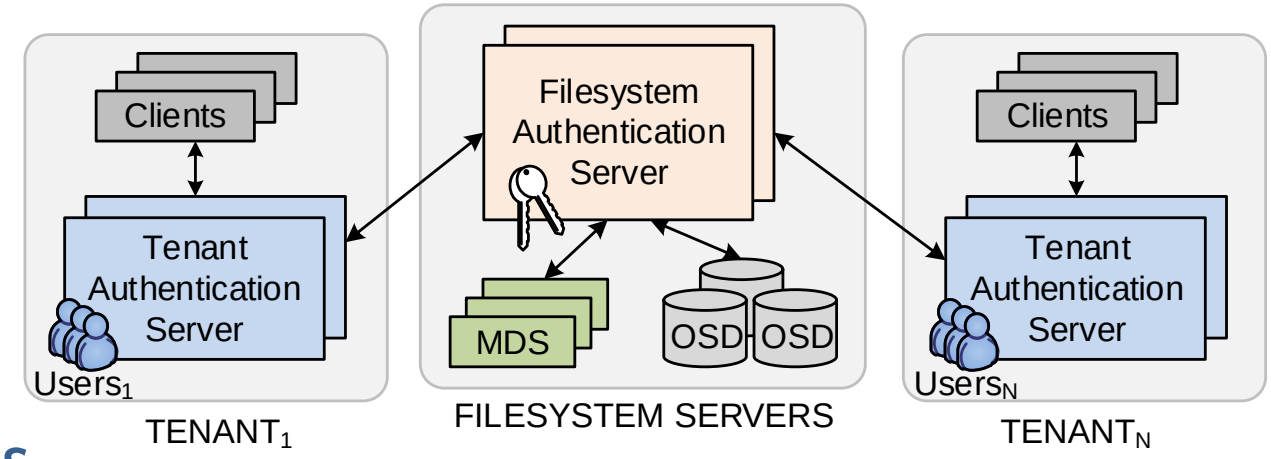
Native multitenant authorization

- Separate ACLs per tenant and provider
- Namespace isolation through filesystem views

Efficient permission management and storage

- Shared common permissions
- Inheritance of permissions

Identification



Principals

- **Tenant principals:** Use/manage tenant resources
- **Native FS principals:** Manage the FS

Tenant Authentication Server (TAS)

- Certifies local clients and principals

Filesystem Authentication Server (FAS)

- Certifies filesystem services, tenants, native principals

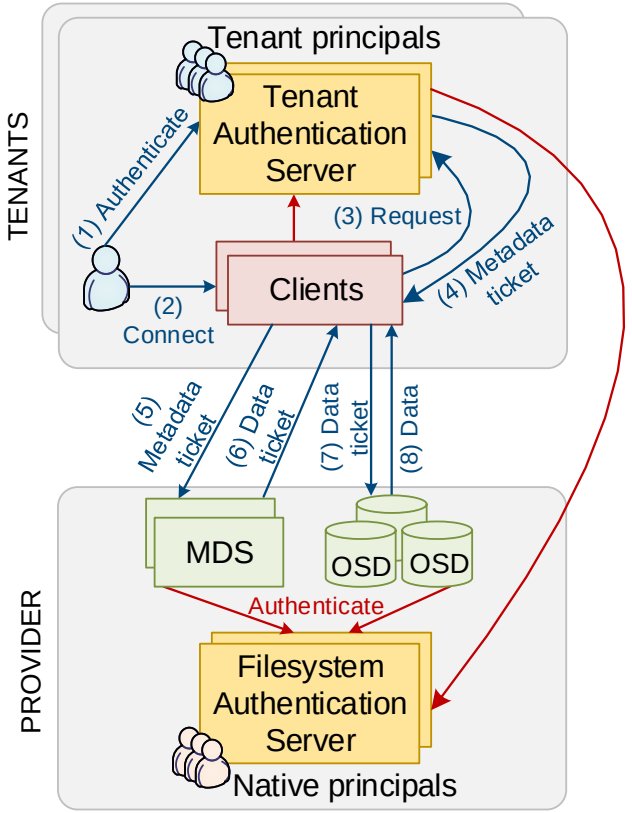
Authentication

Metadata ticket

- Securely specifies tenant principal
- Provides access to MDS

Data ticket

- Securely specifies tenant principal and permissions
- Provides access to OSDs



Steps

(1) Principal authenticated by TAS
(2) Principal requests FS access
(3) Client contacts TAS
(4) Client receives Metadata ticket
(5) Client contacts MDS
(6) MDS issues Data ticket
(7) Client contacts OSD
(8) Client accesses data

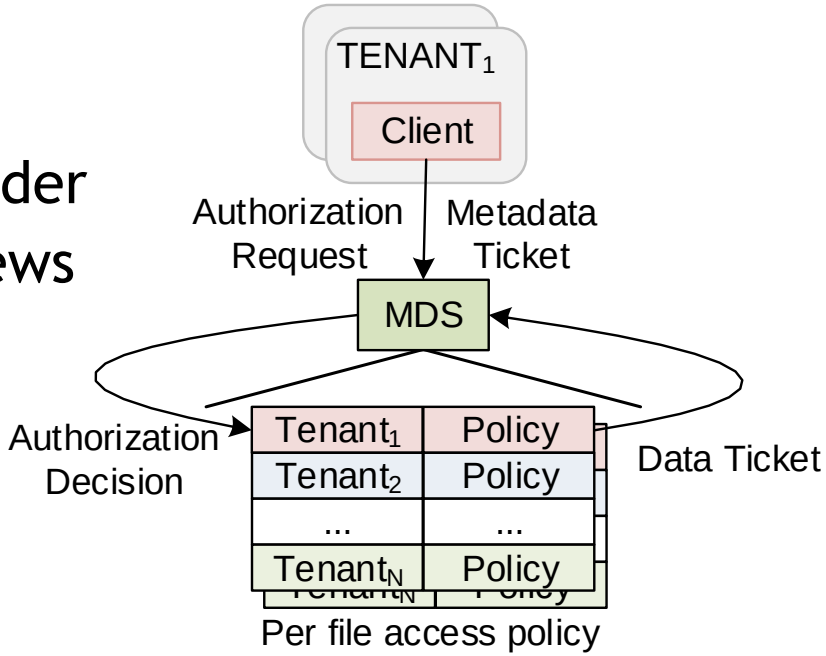
Authorization

Access control isolation

- Separate ACLs per tenant, provider
- Metadata accessible through views

Filesystem view

- Used by native principals to manage tenants



Tenant view

- Used by tenants to access or manage tenant resources

File sharing

- Private to a principal
- Shared across principals of one or more tenants

Common Permissions

Goal

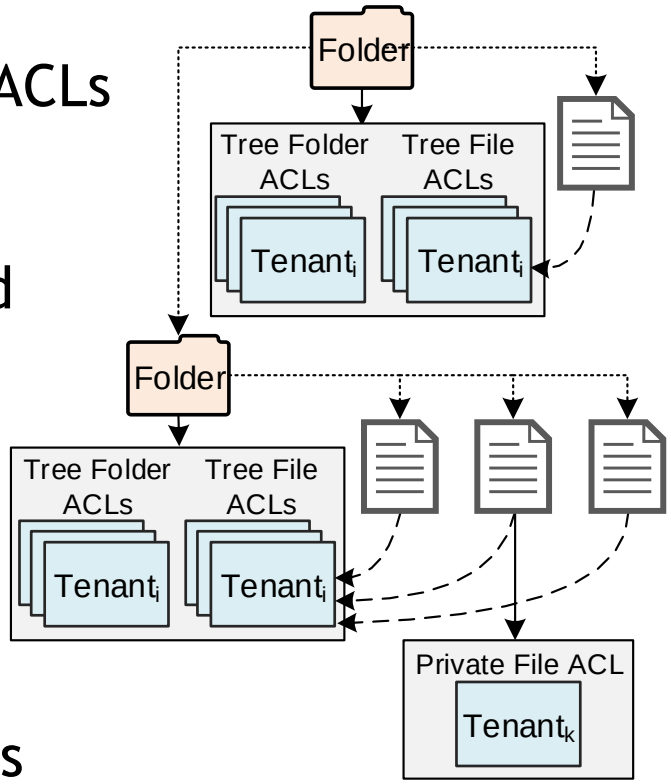
- Reduce filesystem load by reducing ACLs

Per tenant permission inheritance

- Permissions can be inherited to child files/folders

Per tenant common permissions

- Child files can share parent's ACL



ACLs

- **Tree folder ACLs:** Folder permissions
- **Tree file ACLs:** Shared child files permissions
- **Private file ACLs:** Child file permissions explicitly set by user

Security Analysis

Captured credential

- Fresh tamperproof credentials cannot be forged

Compromised tenant principal account

- Compromised tenant view is isolated
- Attack limited to principal's private or shared files
- Cross-tenant policy violation is prohibited

Attack by revoked tenant

- Restricted through deleted tenant view
- Tenant cannot access other views

Compromised provider administrator account

- Handle via good practices (e.g., restricted remote access)

Prototype

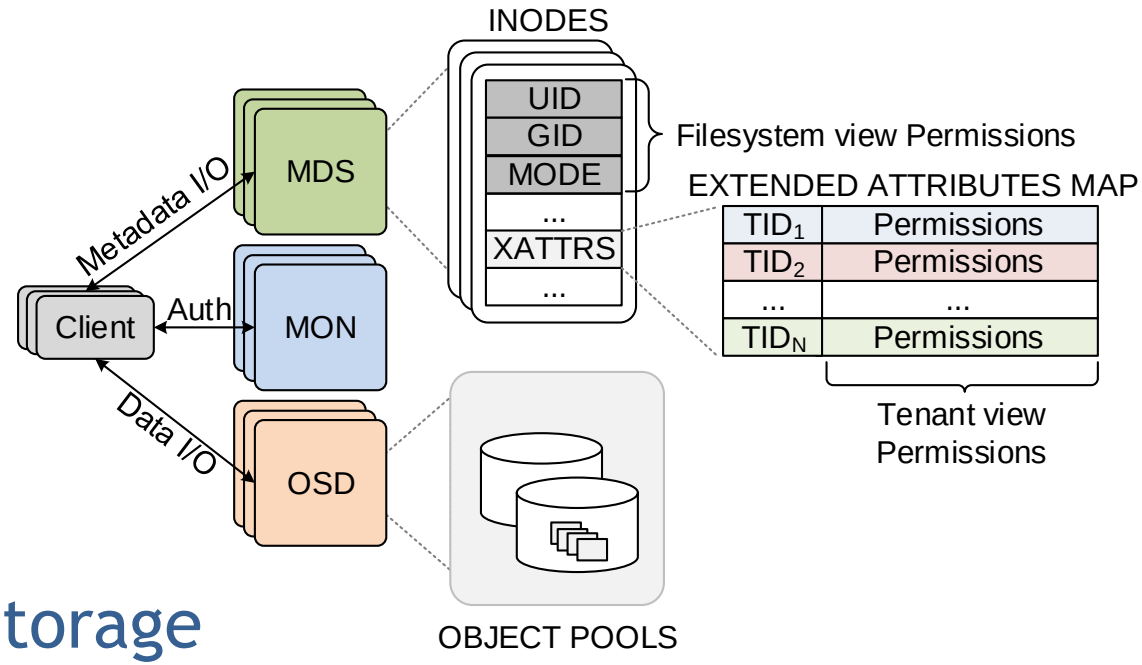
Identification

- Each tenant has its own public key pair
- The tenant ID is derived by applying a cryptographic hash function on the public key
- Clients know their tenant's ID

Session

- During FS mount clients initiate a new session with the MDS
- The create session request message contains the tenant ID
- The MDS tags the new session with the tenant ID
- The session can be used only by the principals of the identified tenant

Prototype



Permission storage

- **Tenant view:** Extended Attributes (EAs) indexed by Tenant ID
  - *Folders:* maintain per tenant tree file/folder permissions
  - *Files:* maintain per tenant private file permissions if specified
- **Filesystem view:** Regular fields
- EAs with tenant permissions not directly accessed by clients

Prototype

Capabilities

- Include principal and tenant identifiers
- Sent to clients with tenant file access

Management of tenant access policies

- grant_tenant_access, revoke_tenant_access
  - New client and MDS calls to grant/revoke tenant access to/from a file or a directory
- change_tenant_policy
  - Tool for administrate tenant access policies

Experimentation Environment

Configuration: AWS EC2 Instances

- **m1.xlarge:** x3, 4 VCPU, 15 GB RAM, Linux 3.9.3
- **t1.micro:** x32, 1 VCPU, 615 MB RAM, Linux 3.9.3

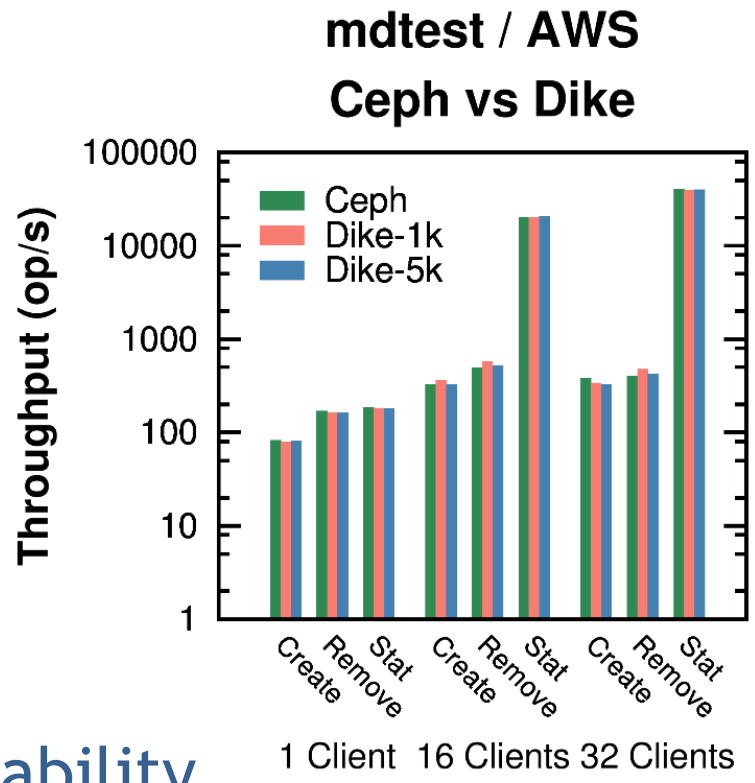
Filesystem configuration

- **Ceph/Dike:** m1.xlarge, 1xOSD+MON, 1xOSD+MDS, 1xOSD
- **Gluster/Heka:** m1.xlarge, 3 fileservers
- Replication factor 3

Microbenchmark

- mdtest
- 48000 created files and folders

Results



Dike Client scalability

- 1 → 32 clients: Similar to Ceph

Dike Tenant scalability

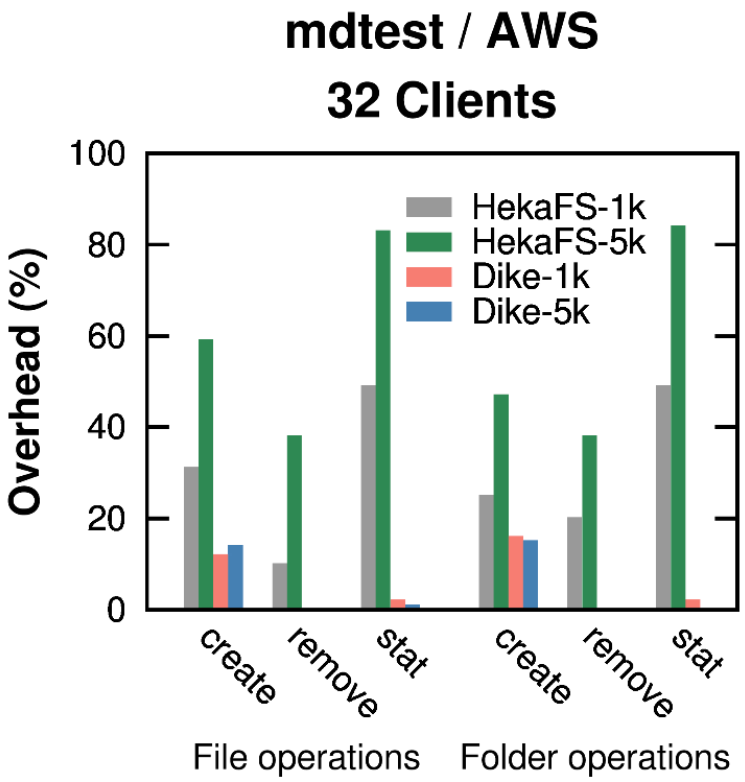
- 1k → 5k tenants: 2% extra overhead

Results

Dike native multitenancy

Limited overhead

Scalable to thousands tenants



Dike limited overhead

- 1k tenants overhead: up to 14%
- 5k tenants overhead: up to 16%

ID mapping multitenancy too costly

- 1k tenants overhead: up to 49%
- 5k tenants overhead: up to 84%

Conclusions

Native filesystem multitenancy with sharing support

- Hierarchical identification scheme
- Namespace isolation: Per tenant and provider ACLs
- Per tenant common permissions and inheritance

Performance and security analysis

- Limited multitenancy overhead up to 16%
- Dike scalable to several thousand tenants
- Tenant principals not able to violate cross-tenant policy

Future work

- I/O intensive application experimentation
- Weaker trust assumptions