ΚΛΙΜΑΚΩΣΙΜΟΣ ΕΛΕΓΧΟΣ ΠΡΟΣΒΑΣΗΣ ΓΙΑ ΑΣΦΑΛΗ ΠΟΛΥΜΙΣΘΩΤΙΚΑ ΣΥΣΤΗΜΑΤΑ ΑΡΧΕΙΩΝ

Η ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης

του Τμήματος Μηχανικών Η/Υ και Πληροφορικής
Εξεταστική Επιτροπή

από τον

Γεώργιο Καππέ

ως μέρος των Υποχρεώσεων για τη λήψη του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ

ΣΤΑ ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ

Οκτώβριος 2013

Dedication

To my parents Lefteris and Areti,
and my beloved sister Antonia.

Acknowledgements

I would like to express my deepest gratitude to all those people that I have been fortunate to be with, and offered me advice, support, encouragement, and friendship.

First and foremost, I would like to express my sincere gratitude to my advisor Prof. Stergios Anastasiadis for his invaluable assistance and guidance at every stage of my graduate career. From the early stages of this thesis to the last ones, he was always willing to help with any problem I was facing. With his deep knowledge on the field of computer systems, he not only guided me through this project, but taught me the invaluable process of conducting computer systems research.

I am also grateful to all my friends who each in his way helped me to complete this thesis. Especially, Andromachi Hatzieleftheriou, through endless hours of brainstorming sessions, provided me with precious feedback at several points of my thesis. Not to mention the delicious meals she offered us. Giorgos Margaritis provided tea, bread-sticks, and jokes. The multiple laps we had together around the Zosimades stadium helped me to mentally relax after a full day of work. Nikolaos Papanikos was always coming at the office on the right moment to break the routine. Finally, Eirini Micheli suggested me the “LyX” text-editor in which I wrote this thesis. But more importantly, she provided me with warmness, help, and encouragement throughout my studies at the University of Ioannina. She was listening with patience every kind of problem I was facing, and it was her who encouraged me to begin my graduate studies. Eirini was always there for me.

Above all, however, I would like to express my deepest love and gratitude to my family for the support they provided me through my entire life. Their love and warmness give me courage to pursue my interests and satisfy my curiosities.

Finally, it should be noted that this work was in part supported by an “AWS in Education Research grant” award.

Table of Contents

List of Figures

List of Tables

Abstract

Georgios E. Kappes, MSc, Computer Science and Engineering Department, University of Ioannina, Greece. October, 2013. Scalable Access Control for Secure Multi-Tenant Filesystems.

Thesis Supervisor: Stergios V. Anastasiadis.

In a virtualization environment that serves multiple customers (or tenants), storage consolidation at the filesystem level is desirable because it enables data sharing, administration efficiency, and performance improvements. However, accessing storage at the file level leads to a reconsideration of the access control techniques used to isolate different tenants. Existing solutions require intermediate translation layers for purposes of networked file access or identity management. Nevertheless, such translations hinder the file sharing between different tenants, complicate manageability, and degrade performance.

In the present study we emphasize the need for a new access control architecture in collaborative multitenant virtualization environments to achieve (i) fine-granularity access control, (ii) storage efficiency, (iii) data sharing, and (iv) administration flexibility. In this context, we analyze the security requirements of multitenant filesystems. Then we introduce a system architecture that is backwards compatible to object-based filesystems, and combines native access control with namespace isolation. Our architecture securely isolates different tenants, and enables flexible file sharing both within and among tenants. It also offers more manageability opportunities with respect to the existing solutions.

Based on our design, we developed a system prototype over a mature distributed filesystem. We experimentally evaluate our software implementation with synthetic benchmarks and application-level workloads using a local cluster and the Amazon public cloud. Thus, we show that our approach incurs limited performance overhead in comparison to traditional single-tenant filesystems, achieves better performance than existing solutions based on intermediate translation layers, and also we provide better scalability for a large number of tenants.

Εκτεταμενη Περιληψη στα Ελληνικα

Γεώργιος Καππές του Ελευθερίου και της Αρετής. MSc, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Οκτώβριος, 2013. Κλιμακώσιμος έλεγχος πρόσβασης για ασφαλή πολυμισθωτικά συστήματα αρχείων.

Επιβλέποντας: Στέργιος Β. Αναστασιάδης.

Σε ένα περιβάλλον εικονικοποίησης που εξυπηρετεί πολλαπλούς πελάτες (ή μισθωτές), η ενοποίηση των αποθηκευτικών πόρων ανεξάρτητων μισθωτών στο επίπεδο του συστήματος αρχείων μπορεί να αποτελέσει τη βάση για την ανάπτυξη ενός αποδοτικού και ασφαλούς περιβάλλοντος συνεργασίας. Μια τέτοια ενοποίηση προϋποθέτει τη χρήση μιας διεπαφής πρόσβασης σε επίπεδο αρχείων, η οποία καθιστά εφικτή την κοινή χρήση αρχείων, επιτρέπει την αποτελεσματικότερη διαχείριση του συνολικού συστήματος και βελτιώνει την απόδοση. Ωστόσο, η πρόσβαση στους αποθηκευτικούς πόρους με χρήση μιας διεπαφής επιπέδου αρχείων καθιστά αναγκαία την επανεξέταση των τεχνικών ελέγχου πρόσβασης που χρησιμοποιούνται από το σύστημα αποθήκευσης ώστε να παρέχει ασφάλεια και απομόνωση στους μισθωτές. Οι υπάρχουσες λύσεις απαιτούν ενδιάμεσα επίπεδα μετάφρασης για να παρέχουν ασφαλή πρόσβαση σε ένα δικτυακό σύστημα αρχείων και να διαχειρίζονται τις ταυτότητες των χρηστών που έχουν πρόσβαση στο σύστημα. Εντούτοις, η χρήση πολλαπλών επιπέδων μετάφρασης δυσχεραίνει την κοινή χρήση αρχείων μεταξύ χρηστών που ανήκουν σε διαφορετικούς μισθωτές, δυσκολεύει τη διαχείριση του συστήματος και μειώνει τη συνολική του απόδοση.

Στην παρούσα εργασία τονίζουμε την ανάγκη για μια νέα αρχιτεκτονική ελέγχου πρόσβασης σε συνεργατικά πολυμισθωτικά περιβάλλοντα για λόγους (1) ελέγχου πρόσβασης με υψηλότερο βαθμό ανδρομέρειας, (2) αποδοτικότερης αποθήκευσης, (3) κοινής χρήσης αρχείων, (4) καλύτερης και ευκολότερης διαχείρισης. Στο πλαίσιο αυτό, αναλύουμε τις απαιτήσεις σε ασφάλεια των πολυμισθωτικών συστημάτων αρχείων και εισάγουμε μια νέα αρχιτεκτονική ελέγχου πρόσβασης. Η αρχιτεκτονική που προτείνουμε συνδυάζει τον εγγενή έλεγχο πρόσβασης με την ασφαλή απομόνωση του χώρου ονομάτων κάθε μισθωτή και είναι συμβατή με οποιοδήποτε σύστημα αρχείων που βασίζεται στα αντικείμενα. Επιπλέον, διαχωρίζει αποτελεσματικά τους χώρους ονομάτων διαφορετικών μισθωτών, και ταυτόχρονα καθιστά εφικτή την κοινή χρήση αρχείων μεταξύ χρηστών που ανήκουν στον ίδιο ή σε διαφορετικούς μισθωτές. Τέλος, παρέχει ευκολότερη και αποτελεσματικότερη διαχείριση του συστήματος.

Με αναλυτικά αποτελέσματα και πειράματα σε πρωτότυπη υλοποίηση δείχνουμε ότι η λύση μας εισάγει περιορισμένη επιβάρυνση σε σχέση με παραδοσιακά συστήματα αποθήκευσης ενός μισθωτή. Επιπλέον, δείχνουμε ότι η λύση μας εισάγει χαμηλότερη επιβάρυνση σε σχέση με υπάρχουσες λύσεις που απαιτούν ενδιάμεσα επίπεδα μετάφρασης, και παρέχει καλύτερη κλιμακωσιμότητα για μεγάλο αριθμό από μισθωτές.

Chapter 1

Introduction

In recent years, the cloud computing paradigm has enabled enterprises to dramatically improve how they organize their infrastructure and operate their business, taking advantage of the scalability and flexibility of a cloud environment. The increasing popularity of cloud environments poses ever greater demands on the scalability, and security of the underlying storage systems.

Whether providing services to the public or serving internal customers, cloud platforms typically allow multiple customers to share the same physical server and network infrastructure, as well as to use common platform services. Cloud customers could be independent organizations or business groups and they are known as tenants [6, 9]. The consolidation of resources into a shared resource pool is a prominent feature of cloud computing in order to improve efficiency, scalability, and reduce costs.

While multitenancy on cloud environments provides seemingly limitless scalability, it raises new security and privacy issues, because it hands the processing and storage tasks over to third parties and involves an enormous number of tenants that share the same resources. In fact, access control over the resources of a multitenant environment is a challenging problem due to the enormous number of end users involved and the required isolation of the security administration across different organizations. Distributed authorization has already been extensively studied in the context of networked services, e.g., distributed filesystems [40]. However, a cloud environment introduces unique characteristics that warrant reconsideration of the assumptions and solution properties.

1.1 Motivation

In the present study we are particularly interested to take advantage of service co-location in the datacenter to better consolidate the storage infrastructure used by common data files at the application (e.g. collaboration documents) or system level (e.g. root images). Secure storage consolidation at the filesystem level is increasingly advocated as the preferred multitenancy paradigm for cloud environments [43, 36, 21, 11, 64]. Although virtual disks are attractive for their versioning, isolation, and migration properties, a file-based interface can additionally support fine-grained controlled sharing, easy resource administration, and file-level performance optimizations. Below, we examine scenaria of virtualization environments in which file based storage consolidation makes sense for reasons of (i) fine-granularity access control, (ii) storage efficiency, (iii) data sharing, and (iv) administration flexibility.

Scientific data: Collaborative research groups require to share scientific data across teams that span multiple institutions. Data owners should be able to easily share their data with users that belong to different institutions without requiring them to have accounts on the storage servers where the data resides. In addition, a tenant's identity should be verified before making shared data available and only users that belong to this tenant should be able to access the data. Data providers must have full control over both the data that may be shared and the permissions that may be granted to external users.

Virtual Desktops: An enterprise stores the desktop filesystems of personal thin clients. Each desktop root filesystem is stored as a separate folder with access limited to a single client. As an optimization, there is a shared folder that is branched into the private folder of each client. Hence, clients can use the shared folder to collaborate on a project. A similar approach can also be applied to manage the home folders of users. In this scenario, the root folder of each client is branched into a shared but read-only folder. In addition, each user is given its own private home folder, where she can store private files.

Software-as-a-service: A software-as-a-service provider supports different business customers with separate end users. The filesystem treats each business customer as a tenant with separate application files in writable mode (e.g. databases), but possibly shared system files in read-only mode (e.g. configuration scripts).

Software Repository: A public provider offers a shared software repository that different groups of developers can fork into separate branches. The members of a group obtain writable access to their own branch, and read-only access to the branches of other groups. A simpler scheme without branches could be used for sharing scientific datasets.

1.2 Research objectives

Accessing shared storage through a block-level interface completely hides file-level access control. Read from or writing to storage devices happens at the granularity of blocks and hence file semantics are completely hidden. On the other hand, when a file-level interface is employed to access shared storage, the fileserver is ultimately in charge of access control. The adoption of a file-based solution in a multitenant environment, where multiple customers share a single filesystem namespace, raises the need to reconsider the access control techniques used in order to effectively isolate the principals of different tenants.

Existing file-based solutions face scalability limitations because they either lack support for multiple guest tenants, rely on global-to-local identity mapping to manage the users of different tenants [11], or have the guests and a centralized filesystem (or proxy) running at the same host [43, 21, 12]. In addition, they hinder support for file sharing among principals that belong to different tenants and complicate administrative tasks.

In the present study we set as our primary goal to securely manage the shared filesystem namespace, in order to provide each tenant with an isolated private view. However, in contrast to previous approaches, our solution should permit principals of the same or different tenants to share files and collaborate on a shared project. In addition, it shall provide system administrators with more manageability opportunities, and finally, it shall maintain high performance and scalability by natively supporting multitenancy.

1.3 Contributions

Secure access control is a challenging problem that organizations face in collaborative virtual environments, which has prevented many of them from migrating critical data or applications into such environments. In our research we examine approaches for efficient and effective support of multitenancy in filesystems used by virtual machines. We require that each client directly mounts the filesystem instead of having the filesystem mounted by an intermediate proxy. Trusted computing techniques are used to certify the integrity of tenants that wish to access the shared filesystem. Tenants are then responsible for authenticating and authorizing principals operating on their behalf to provide access to the filesystem. The filesystem natively manages the access control metadata of each tenant, and ensures that each tenant can only access its own namespace. Controlled file sharing is relatively straightforward as a result of the file-level access to a common filesystem with file-granularity access control.

We provide prototype implementation of the above approach in the Ceph production-grade, distributed filesystem. With microbenchmarks and application-level experiments we quantitatively demonstrate the limited performance overhead of our design.

We can summarize our contributions as follows:

- Analysis of access control requirements in file-level consolidated storage for virtualization.
- Architectural design of native access control in a multitenant filesystem with backwards compatibility to object-based storage.
- Prototype implementation over a production-grade distributed filesystem.
- Experimental performance evaluation of multitenancy overheads.

1.4 Roadmap

In chapter 2 initially we present the basic features of cloud environments and virtualization, and the core security primitives for securing data in large-scale distributed storage systems. Then, we delve deeper into storage management in virtualized environments and we summarize the different multitenancy architectures for filesystem storage clouds that have been proposed until now. Finally, we discuss why a file-level storage interface makes multitenancy challenging.

In chapter 3 we first analyze the security requirements in multitenant filesystems. Then, we list the goals that we have set for our proposed access control architecture. Furthermore, we provide details about our trust and threat model.

In chapter 4 we introduce a new access control architecture for multitenant shared storage at the file level. Our architecture combines tenant isolation with native access control and is backwards compatible to object-based filesystems.

In chapter 5 we describe our implementation of the proposed multitenant access control over a distributed, object-based filesystem. In addition, we explain important implementation decisions.

In chapter 6 we experimentally evaluate our prototype implementation and give reasons for the limited added performance overhead of our solution. Furthermore, we compare it with existing techniques that aim to enable multitenancy.

In chapter 7 we review the state-of-the-art multitenancy architectures for filesystem storage clouds, and we outline recent works that aim to provide trusted cloud storage. Furthermore, we present an overview of the related literature in the field of access control in distributed filesystems.

Finally, in chapter 8 we summarize the conclusions of our work and highlight opportunities for future research.

Chapter 2

Background

In this chapter we first present an introduction to the basic concepts of cloud environments and virtualization. We also present the core security primitives for securing data in large-scale distributed storage systems. Then, we concentrate on storage management and we briefly introduce the architecture of object-based distributed filesystems and its advantages over traditional distributed filesystems. Furthermore, we compare the block-level interface with the file-level interface in terms of sharing and manageability efficiency, as well as performance. Finally, we summarize the different multitenancy architectures for filesystem storage clouds that have been proposed until now and we highlight why a file-level storage interface makes multitenancy challenging.

2.1 Cloud environments

Cloud infrastructures are increasingly used for a broad range of computational needs in private and public organizations. Cloud computing aims at allowing access to large amounts of computing power in a fully virtualized manner, by aggregating resources and offering a single system view. The deployment of cloud infrastructures can be performed in different ways, depending on the organizational structure and the provisioning location [35].

Four deployment models are usually distinguished, public, private, community, and hybrid. The deployment of a public cloud infrastructure is characterized by the public availability of the cloud service offering. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them and is offered to the public through a public network. On the other hand, in a private cloud deployment, the cloud infrastructure is provisioned for exclusive use by a single organization comprising of multiple customers. It is owned, managed, and operated by the organization, a trusted third party, or a combination of them. The main advantage of this kind of deployment is that the organization retains full control over corporate data, security guidelines and system performance. While a private cloud is only accessible by a single organization, a variant of this deployment, which is known as a community cloud, enables organizations with similar requirements (projects, security requirements, policies) to share a cloud infrastructure in order to collaborate. The infrastructure could be managed and hosted by one or more of the organizations in the community, or by a third-party. Finally, in a hybrid cloud deployment the cloud infrastructure is a composition of two or more distinct cloud infrastructures (public, private, or community) that remain unique entities. A hybrid deployment allows an organization to maintain sensitive data behind its firewall, while taking advantage of the lower cost and flexibility of a public cloud.

The main idea behind cloud computing is to deliver a huge amount of computing resources as services through a public network such as the Internet. Cloud services can be divided into three categories according to the abstraction level of the resource provided, namely: (1) Software as a Service, (2) Platform as a Service, and (3) Infrastructure as a Service. In the first model, Software as a Service, one or more applications and the computational resources to run them are provided for use on demand as a service. On the other hand, Platform as a Service is a model of service delivery whereby the computing platform (typically including operating system, programming language, execution environment, database, and web server) is provided as a service to software developers. Finally, Infrastructure as a Service is a service model where the cloud provider offers virtualized resources (computation, storage, and network) on demand. To deploy their applications and services, customers install operating system images and their applications on the cloud infrastructure. The focus of this study lies in this final model.

Cloud computing services are usually backed by large-scale datacenters. Modern datacenters are heavily virtualized, thereby, computing, storage, and network resources of each physical server are multiplexed across a large number of different applications and tenants. Thus, cloud platforms allow multiple tenants to share the same resources. This leads to multiple benefits. On the one hand, higher resource utilizations are achieved and on the other, resource sharing can lead to a great reduction of energy consumption and cut costs. In fact, most datacenters often utilize virtualization and distributed services to manage resources and provide a scalable computing platform [31], making virtualization a fundamental component of cloud computing.

2.2 Virtualization

Virtualization is a broad term of computer systems that refers to an abstraction mechanism which hides the physical characteristics of certain computational resources in order to simplify the way in which other systems, applications or end users interact with them. Thus, virtualization enables sharing the resources of a computer system in multiple execution environments.

The concept of virtualization has its roots in the mid 1960's, when it was used by IBM as a method for logical partitioning of large centralized systems (mainframes) into separate virtual machines. The virtual computers were distributed to users of the system, allowing each user to work in an isolated environment without affecting other users. To make this sharing possible, IBM introduced a new feature called Virtual Machine Monitor (VMM).

The Virtual Machine Monitor [46] is a software layer that is placed on top of the hardware layer and has direct access to hardware resources. Its main objective is to manage and allocate system resources to one or more virtual machines in order to make virtualization possible. Virtualization follows various approaches that are directly related to the architecture of the Virtual Machine Monitor. In the hosted architecture the VMM runs as an application on the host operating system and relies on it for resource management, system memory, and device drivers. In the autonomous architecture the VMM is placed directly above the hardware. Thus, it is responsible for managing system resources and their allocation to different virtual machines. This last architecture is more efficient because the VMM has direct access to system resources.

Guest operating systems run with limited privileges and they don't have direct access to hardware. Thus, it is difficult to virtualize some critical operating system instructions because their implementation requires higher privileges. Two approaches were followed to solve this problem: full virtualization and paravirtualization.

Full virtualization provides a virtual environment that simulates the real hardware. Specifically, each virtual machine is provided with all the services of the real system, such as full command set of the real processor, virtual appliances, and virtual memory management. The major difference from other virtualization techniques lies in operating system's awareness that it runs under a virtualized environment. Thus, any software that is capable to run in the real system can run without changes in the virtualized environment. In order to make the execution of critical instructions possible, a technique known as binary translation was introduced. According to this approach, the software is patched at runtime. The critical instructions that cannot run in the virtual environment are replaced by different instructions that can run safely. However, the continuous scanning and emulation of critical instructions reduces performance. VMware's Workstation [63] follows the above approach.

On the other hand, paravirtualization provides to the virtual machines a software interface that is similar but not identical to that of the real system. The main purpose of paravirtualization is to reduce the proportion of time spent in performing critical patches on the guest's unsafe instructions. Instead of using the binary translation technique, the client software is modified and communicates directly with the VMM, when it needs to execute a critical instruction. Then, the VMM undertakes responsibility to execute the instruction. As a consequence, the guest operating system must be altered slightly in order to run in a paravirtualized environment. A system that follows the paravirtualization approach is Xen [7].

As the benefits of virtualization are tremendous, manufacturers of processors have reviewed the instruction set of their products by making them virtualization-friendly. Thus, virtualization of critical operating system instructions can be solved directly using the new instruction set.

2.3 Core security mechanisms

In a distributed filesystem, client is typically a process that provides local filesystem access to a node and servers the processes that implement filesystem action across the network. Principal is an entity that accesses the filesystem through the client. This entity can be a process, a person, or a role. A principal can also be a compound of other principals, for example a group of users [40].

Reliability and security in a large-scale storage system can be enforced with a combination of four different techniques, including *Encryption*, *Identification*, *Authentication*, and *Authorization*.

Encryption is the process of encoding information in such a way that eavesdroppers cannot read it, while authorized parties can. A secure cloud storage system requires two kinds of encryption: For data that is being transferred over the network and for data "at rest" on disk. Usually, when cloud tenants don't entrust the cloud provider with their data, they can provide confidentiality to themselves by encrypting the data they store on the cloud.

Identification is the process in which an entity supplies information to identify itself to an authentication service. Some examples of identification mechanisms are usernames, memory cards, and public keys.

Authentication is the activity of verification of an entity's identity. It can be performed using passphrases, passwords, cryptographic keys, and tokens. It confirms the identity of an individual, but says nothing about its access rights. Authentication often involves verifying the validity of at least one form of identification.

Authorization is the process of determining access rights: What an identified entity can actually access and what operations it can carry on this information. Authorization is normally preceded by authentication for user identity verification.

Most access control systems need also to limit the actions of application processes. In particular, they must prevent a process from reading or overwriting memory that it may not access. One solution to this problem is to use sandboxing techniques. A *sandbox* is a security mechanism for separating running programs (e.g. SELinux [30]). However, sandboxing techniques are often too restrictive for general computing environments. Another solution to the above problem is to use mechanisms like *segment addressing*, which integrate hardware access control with the memory management functions [5].

A secure environment must also ensure the integrity of computing platforms. In fact, users must be sure that a given program runs on a machine with a given specification; that is, the software has not been modified and the hardware configuration has not been changed. A typical mechanism that provides such assurances is the Trusted Platform Module or TPM. The TPM is actually a secure co-processor which monitors a system at boot time and reports its state to the operating system. In fact, it generates a cryptographic key that depends on the current system's state, as well as a fingerprint (hash) of the software stack that booted on the system and provides them to the operating system. Thus, if a modification is made to the system's configuration, the TPM chip will generate a different cryptographic key and the previously encrypted material will not be made available. A system can also use the TPM to certify the identity of a remote system. This process is called *remote attestation*. Furthermore, TPM can also be used for disk encryption. It offers two primitives, *seal* and *unseal* to encrypt and decrypt information respectively. Seal encrypts data and specifies a state in which the TPM must be in order for the data to be decrypted (unsealed) [5, 52].

2.4 Storage management

File and storage systems used in virtualization environments have proved critical to the overall performance of an exceedingly broad class of applications. Storage systems can be distinguished in three different categories depending on how data is stored on the underlying system: block-based storage, file-based storage, and object-based storage. The storage in the block-based approach is conceptually modeled as a long stream of bytes divided into equally sized blocks. All accesses to the storage devices are performed by reading or writing blocks. Examples of this type of storage include Amazon Elastic Block Store (EBS) [1], Ceph RBD [20], and iSCSI [53]. In the second form of storage a filesystem is layered on top of a block storage device. The filesystem is a a higher-level logical structure that maps higher-level objects, which are typically called files (such as documents, pictures, and videos), onto disk blocks. Some examples of file-based storage systems include NFS [55] and CIFS [38]. In the latter form of storage, which is known as object storage, the storage system uses objects to store information. Object-based distributed storage systems emphasize the scalability of secure data and metadata management. Some typical examples of object-based storage systems include Amazon S3 [3], Rackspace Cloud files [49], and Ceph Storage [67].

2.4.1 Object-based distributed filesystems

An object-based distributed file system separates the management of file metadata[1] from file data. File metadata is managed by metadata servers, while a different form of servers, which are known as object storage servers, manage file data. Both data and metadata are split into objects which are stored on object storage servers. The filesystem client employs metadata and object storage servers to present a full filesystem abstraction to the users [67].

A significant advantage of the object-based file system architecture is the elimination of the potential bottleneck of the metadata server and the parallelization of all file I/O. In fact, a client needs to contact the metadata server only once, for example when it opens a new file. Another benefit of this architecture is that by grouping data into objects allows the object storage server to optimize access to related blocks, because data that resides in the same object is related and potentially different from data in a different object. What is more, data can be split to multiple objects in order to keep the size of an object under a specific limit. These objects are then stored to different object storage servers (*data striping*). This allows the stripe width to be adapted to the access properties of an individual file.

[1]Such as the filename, the file size, and access control information.

2.4.2 An outline of Ceph

Ceph is a distributed object-based filesystem developed by Weil et al [67]. It consists of four components: The clients provide access to the filesystem, the metadata servers (MDSs) manage the namespace hierarchy, the object-storage devices (OSDs) reliably store data in the form of objects, and the monitors (MONs) manage the server cluster map. Both data and metadata are stored on OSDs, but they are separately managed for greater scalability. The metadata is dynamically partitioned across the MDSs to preserve locality and achieve load balancing.

Metadata management

The MDS is responsible to manage metadata for files and folders. A Ceph folder is stored as a single object, or as a collection of fragments, with each fragment on a different object. When a folder is divided into multiple fragments, the Ceph client is responsible to request the correct fragments from the MDS. If the client needs the whole folder, it iteratively requests the next folder fragment from the MDS, until it forms up the whole folder. A folder entry includes the name, the inode, and the extended attributes of a file. Every MDS maintains a journal [47] of recently-updated metadata. Incoming metadata updates are labeled as *projected* while written to the journal but not yet to the in-memory cache, *committing* while queued to disk, and *committed* when written in stable storage. Metadata journaling allows the MDS to serve recent metadata back to clients. In addition, the journal is also useful for failure recovery.

Data and metadata storage

Ceph stores file data and metadata as objects. Each object is stored as a file in the underlying filesystem of an OSD. An object has an identifier, binary data, and object metadata consisting of a set of key/value pairs. Note that the actual file metadata (the file inode) is stored in a different object.

Objects are mapped into Placement Groups (PGs). Grouping objects to PGs helps ensure performance and scalability, as tracking metadata for each individual object would be too costly. The PGs are then mapped to one or more OSDs. Replication is done at the PG layer. However, the degree of replication is specified higher, at the pool level, and all

PGs in a pool will replicate stored objects into multiple OSDs. A pool is a collection of PGs and thus a collection of objects. Objects are mapped to PGs and PGs to OSDs with the help of a pseudo-random data distribution function [68]. This function allows Ceph to re-balance dynamically when the OSD map changes. Furthermore, it ensures that object replicas do not end up on the same disk or host.

2.5 Storage interfaces

Storage systems can be accessed through different types of storage interfaces which can be distinguished in three different categories: a block-level interface, an object-level interface and a file-level interface. A block-level interfaces exposes a block device to the user and allows the writing and reading of fixed-size blocks. On the other hand, with an object-level interface users can access objects typically through a REST API [13]. Finally, a file-level interface exposes the file and folder structure to clients. Then, clients ask the server to read or write a consecutive range of bytes within a particular file in each request and the server maps this request to the storage devices.

Existing virtualization environments primarily apply storage consolidation at the block level [37, 18, 54, 62]. Guests access virtual disk images which are typically stored in a central location. They are offered to guests as direct attached disks through a block I/O interface, or as volumes through a storage area network mounted by the host. This approach incurs important benefits, such as high availability of data, easier administration, and optimal usage of storage capacity. The block-level interface provides a narrow interface to storage and yields an agnostic and simple implementation. Furthermore, it offers the system compatibility of a standalone machine. Moreover, a block-level interface is useful for supporting heterogeneous clients and client applications.

An alternative approach is the design of a virtualization-aware distributed filesystem [43, 21, 12]. This architectural design goes one step further beyond virtual disks and attempts to provide storage virtualization at the file level. In fact, it combines the sharing opportunities offered from distributed systems with the intrinsic features of virtual disks such as isolation, versioning, and mobility.

2.5.1 Sharing and manageability

The semantic gap introduced by virtualizing the system at a low level obscures higher-level information that could aid in identifying opportunities for sharing among different VMs, complicating the efficiency of storage management, and making collaboration tasks impossible. Furthermore, it hides the storage structure and thus complicates administrative tasks.

File level access of consolidated storage offers more manageability opportunities than block level access. Stored data is directly accessible through a standard file-level interface at the server without requiring shutting down the virtual machines which access the storage.

A block-level interface offers no opportunities to share read and write access between multiple parties, which complicates virtual machine management. Concurrent access can only be enabled with the help of a secondary protocol, or the usage of a traditional network or distributed filesystem to export specific parts of the filesystem. However, such solutions incur extra manageability effort because they involve either the design of extra protocols, or the maintenance of multiple administrative domains.

In addition, filesystem administrative tasks, such as data searching and software installations or updates, can be performed more efficiently and globally through a file-level interface. With a block-level interface instead, the administrator would be forced to shutdown all the affected virtual machines in order to mount their images and perform the required task in one image at a time. In fact, a file-level interface increases administration flexibility and efficiency because it enforces a granularity of individual files rather than entire disk or blocks.

Another potential manageability benefit from using a file-level interface lies to its ability to provide an ephemeral and highly composable storage. A filesystem can be synthesized from a set of file trees, each of which contains related files. In addition, a file tree can be shared among multiple users or can be private. For example, there might be a collection of file trees, each of which may contain the root filesystem of a different operating system. Another collection may contain file trees that hold each user's home folder. A last one would have file trees that have specialized applications installed. Let's suppose that Alice is a developer who uses the Eclipse Integrated Development

Environment (IDE) and prefers the Debian Linux distribution. With composable storage enabled she has the flexibility to synthesize her filesystem by choosing the Debian file tree from the operating systems collection and combine it with probably a shared file tree that contains the Eclipse IDE and a private file tree for storing personal files.

The above discussion makes it clear that a file-level interface offers significant manageability benefits in comparison to a block-level interface. A recent study [4] tries to mitigate these limitations of block-level storage by storing images in a format that indexes their storage structure, instead of as opaque disk images. Thus, by providing a file-aware format this approach allows administrative operations such as searching, patching, and virus scanning to execute online.

2.5.2 Performance

File-level access of consolidated storage may improve performance because of its potential to reduce the number of levels to a storage stack traversed by an I/O request. On the other hand, when a VM guest operating system accesses storage through a virtualized block device, file access traverses the guest operating system's filesystem and its block device stack, and then it traverses a similar block device stack in the VMM. Even if the guest operating system accesses storage through a pass-through block device, the file I/O request needs to be translated to a block request and then back to a file request. However, such translations can reduce performance [21].

Hildebrand et al. [19] study the effects of having multiple layers in the storage stack of a virtualization environment. They are particularly interested to investigate the effects of layering a virtual disk on top of a NAS store. In this scenario, a VM's file request is translated to a block request by the hypervisor's storage controller emulator, which in turn translates the block request back to a file request and sends it to the disk image via the NAS client. This large number of layers in the storage stack increases the amount of processing that each request needs and hence increases the I/O latency. In addition, the caching of entire blocks by the block-layer causes read-modify-write operations over the NAS protocol which degrades performance. Having these observations in mind and conducting several experiments the authors state that the layering of the guest block-layer on top of a file-level layer can dramatically reduce performance. As a possible solution, they suggest the guest VMs to consolidate storage directly at the file level by mounting a network-based filesystem.

The performance implications of nested filesystems in a virtualized environment are investigated by Le et al. [27]. They focus on the scenario in which a guest VM accesses a local virtual disk image. Their main observation is that the choice of nested filesystems on both hypervisor and guest levels has a significant performance impact on guest's I/O performance. In addition, they realize that system administrators should carefully choose a combination of guest-hypervisor filesystems according to the type of the anticipated workload. What is more, they demonstrate that there are cases where nested filesystems should be completely avoided.

2.6 Secure storage multitenancy

Secure multitenancy in cloud storage supports multiple customers at low cost [25]. However, maintaining security and access control in cloud environments is a challenging problem and has prevented many organizations from moving critical data or applications to such environments.

Cloud storage systems must address challenges that are not addressed by traditional distributed filesystems. These challenges mostly revolve around isolation, identity management, and privacy. Cloud tenants do not trust each other, and in the case of a public cloud they even do not trust the cloud provider. A cloud storage system must ensure that tenants are isolated from each other, while it offers them opportunities for a secure and collaborative file sharing. Tenant isolation in a filesystem storage cloud can generally be performed in four different levels: hardware level, hypervisor level, operating-system level, and application level.

Tenant isolation can be performed at the hardware level by using a dedicated server per tenant. However, this solution does not scale well, wastes hardware resources, and dramatically increases operating costs.

A second approach is to isolate a tenant at the hypervisor level by using a shared hypervisor and separate virtual machines to host each tenant's fileservers [41, 44, 25]. In this case, the hypervisor enforces isolation by ensuring that the data from one tenant is not propagated to untrusted locations outside the tenant. Although this architecture securely isolates tenants, it hinders the benefits of a shared filesystem, such as data sharing, group collaboration, and data processing scalability.

In contrast to the above approach, the operating-system multitenancy architecture uses shared server hardware and operating system, and relies on the fileserver kernel to isolate the resources of different tenants leading to lower execution overhead [48, 25]. However, this architecture shares the same disadvantages with the previous one regarding the inability for file sharing and collaboration, as well as the poor scalability, because each client has its own dedicated file-service.

Finally, tenant isolation can be performed at the application level, by using shared server hardware, operating system, and fileserver among tenants. This form of multitenancy is also referred as native multitenancy and is considered as the cleanest way to isolate multiple tenants [25]. Despite the fact that achieving multitenancy at this level is a challenging task, this architecture enables all the benefits of the deployment of a shared filesystem, including data sharing, group collaboration, and high scalability.

2.7 Access control on multitenant storage systems

When a storage system is shared across multiple customers, it is critical to control how the access is differentiated so that only the permitted principals to be able to access the data of each tenant. Below, we initially discuss how access control is handled in block-level multitenant storage systems. Then, we highlight the multitenancy challenges that are introduced by storage consolidation at the file level.

2.7.1 Access control on block-level storage systems

Existing cloud environments primarily apply storage consolidation at the block level. Guests access virtual disk images either directly as volumes of a storage-area network (SAN), or indirectly as files of network-attached storage (NAS) mounted by the host. In fact, virtual disk images provide the same block-level interface as physical disks and they have no access control responsibility. Therefore, if a tenant accesses its own collection of virtual disk images, its namespace is strongly isolated from others. While file-level access control is completely hidden by the block-level interface, tenants instead of sharing individual files with each other can share the whole virtual disk image.

The secure sharing of virtual machine images in a cloud environment has been researched by Wei et al. [66]. They propose a virtual disk image management system that controls secure access to images by different tenants, tracks the provenance of images, and provides tenants and administrators with efficient image management tools that detect and prevent security violations. However, a finer-grained sharing at the level of files is more desirable, but a block-level interface can not support it.

2.7.2 Multitenancy challenges of a file-level storage interface

File-based access of consolidated storage has been advocated to improve data sharing, manageability, and performance. Unfortunately, the access control model that is used when a virtual machine accesses storage through a block-level interface cannot be used when the filesystem must be deployed as a shared service and tenants access it through a file-level interface. The main problem in such a shared deployment is that the namespaces of different tenants are no longer isolated from each other. Thus, the filesystem needs to securely support and isolate different administrative domains.

An interesting example is how the isolation of principals is affected in such a deployment. In fact, each tenant contains its own pool of users (see Figure 2.1). Each user is represented by an identity which is called the User ID (UID). The UID is a projection of an actual individual or service into the system. Establishing a unique UID for each individual who will access resources in a shared deployment is critical for security. However, the use of a shared filesystem introduces a possibility of conflict involving the use of the same UID by users belonging to different tenants. As a result, a user who belongs to a particular tenant can access the files of a user with the same UID who belongs to a different tenant. To make the situation worse, the storage server contains its own identity space. As a consequence, a tenant user can gain extra privileges on the fileserver with catastrophic results.

At the same time, other problems related to file permissions and special files are arising when a shared filesystem deployment is used. File permissions assigned to a file by a tenant's user not only affect other users of the same tenant, but they also mistakenly

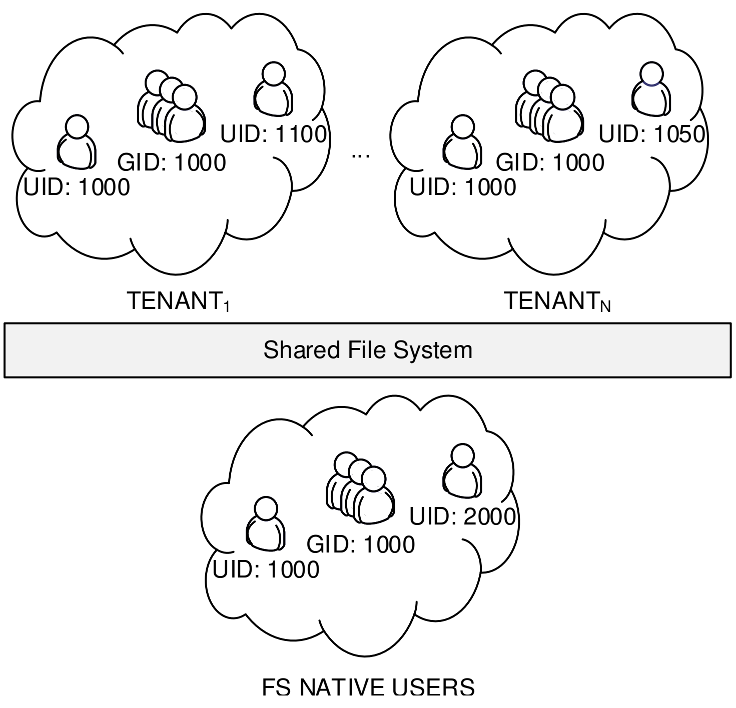


Figure 2.1: The ID collision problem when a single namesapace is shared between different tenants and the provider.

affect the users of different tenants. This situation is unacceptable and is driven by the fact that the different namespaces are not properly isolated from each other. A similar situation arises when a user creates a new special file (e.g. symlink or device file). This file is also presented as a special file on the fileserver. However, a special file has a specific meaning only in the space where it is created. When such files are presented as special files on outer spaces, they may impose a serious impact on system's security. For example, an intruder can use them as backdoor to read or even modify kernel memory, files, disk drives, and other critical devices. Thus, it is critical for a multitenant filesystem to prevent identity collisions and isolate the different tenant namespaces.

2.8 Summary

Cloud computing is a new computing paradigm that provides software, platform, and infrastructure services on demand to customers around the world. A cloud environment may be public, private, community, or hybrid, each with its own distinct constraints. In order to efficiently support an enormous number of customers at low cost, cloud environments rely on sharing of computing resources. A key technology that enables resource sharing is virtualization. The Virtual Machine Monitor is a critical component of virtualization. It may run directly as an application on the host, or it may be placed directly above hardware. A challenge that virtualization faces is how to virtualize critical operating system instructions. This challenge can be solved either with full virtualization or with paravirtualization.

File and storage systems used in virtualization environments are a critical component for the overall performance of hosted applications. Storage systems are distinguished into different categories depending on how data is stored and accessed. Existing virtualization environments apply storage consolidation at the block level. Although the block-level access provides many benefits, such as support for versioning, isolation, and migration, it precludes file sharing, hinders manageability, complicates resource administration, and reduces performance. For these reasons, a file-level interface is more desirable in environments that target collaboration, easy resource administration, and high performance. However, a file-level interface leads to a reconsideration of the access control mechanisms used to securely isolate different customers.

Chapter 3

Design requirements

In this chapter we explain the security requirements in multitenant storage systems and we list the goals that we have set for our proposed access control architecture. We also give details about our trust and threat model. In the next chapter, we propose a design to meet the specified requirements and goals.

3.1 Security requirements of multitenant storage systems

The idea of multitenancy is fundamental to cloud computing. Especially in a storage cloud, service providers are able to build storage architectures that are very efficient and highly scalable to serve the needs of the large numbers of customers that share them. However, in a multitenant storage system, data of different tenants is stored in the same underlying storage devices. Thus, the primary requirement for multitenant storage is to ensure the security of tenant data.

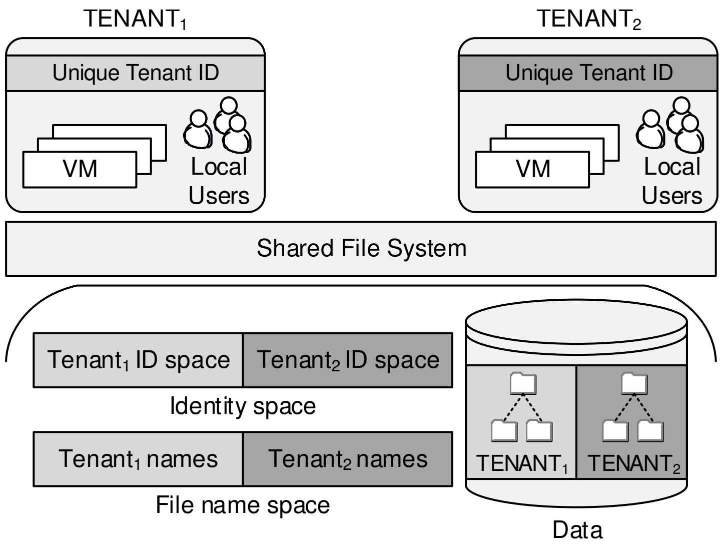


Figure 3.1: Attributes of a shared multitenant filesystem.

When the storage system offers a block-level interface, each tenant accesses its own virtual disks and hence it is hard for a particular tenant to access the data of another tenant. On the other hand, when tenants access the shared storage with a file-level interface, they share a single filesystem namespace. In this case, the risks of multitenancy become more severe.

As shown in Figure 3.1, the filesystem itself must securely separate, protect, and isolate a tenant's data from other tenants. This separation must be complete and secure. However, it must not affect the management, sharing, and flexibility benefits of a shared filesystem. As we have explained earlier in subsection 2.7.2, the access control mechanism must take into account the fact that a single namespace is shared between multiple parties and properly prevent namespace collisions. All in all, if an attacker manage to gain access to a tenant's local account, then his attack must be confined within that tenant and he should be unable to access data of another tenant.

In a multitenant storage system, the tenant ID is what distinguishes one tenant from the others. Authentication mechanisms must ensure that no other tenant can assume a tenant's identity to gain data access. Furthermore, the filesystem must take into account how the access is authorized and differentiated, so that only the right principals can view and manage a tenant's data.

In addition, care must be taken to protect tenant data at rest and obstruct deletion or corruption (accidental or malicious) of it. In the present work we assume that the storage provider and the filesystem servers are trusted. However, there are known techniques, like encryption of data at rest, that can provide an option to meet the security concerns of the most sensitive tenants.

Finally, tenant data access must not be disrupted by denial of service attacks against the filesystem servers and by the normal or abnormal activities of other tenants. However, in this thesis we do not attempt to provide solutions for these kinds of attacks.

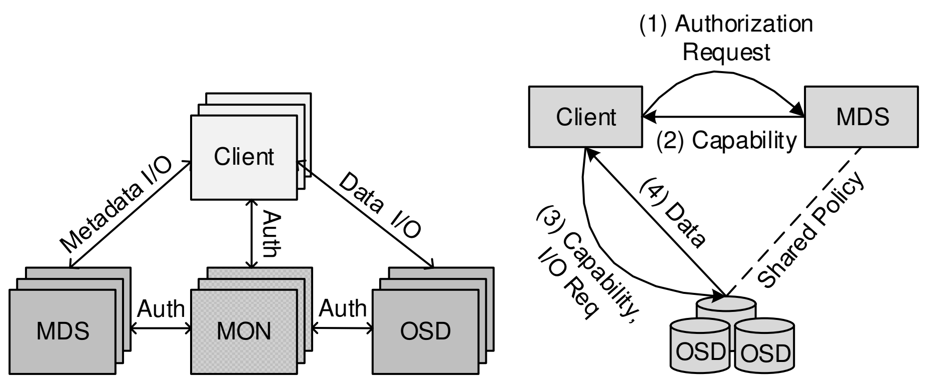
3.2 Architectural goals

The following goals guided the design of the proposed scheme of filesystem access control:

- **Isolation:** Securely isolate different tenants and prevent namespace collisions.
- **Sharing:** Enable collaboration by providing flexible file sharing among the principals of the same and different tenants.
- **Efficiency:** Provide fast data access with native support of multitenant access control for filesystem performance and scalability.
- **Compatibility:** Ensure architectural compatibility with existing scalable and reliable filesystems.
- **Manageability:** Provide more manageability opportunities to facilitate administrative tasks.

3.2.1 Isolation

We assume that each tenant has its own identity space and operates a group of virtual machines with an identical pool of principals and with identical access rights to a set of files. We further assume that two identity spaces of different tenants can collide. In fact, a tenant should not be aware of how other tenants manage their identities. The filesystem should properly isolate the identity space and access control of principals from different tenants.



(a) Architecture (b) Access control

Figure 3.2: The architecture of an object-based, distributed filesystem and its access control mechanism.

3.2.2 Sharing

In addition to tenant isolation, our architecture must provide opportunities for fine granularity intra-tenant and inter-tenant file sharing. For this reason, we use a file-level storage interface that enables sharing, in contrast to a block-level interface. Furthermore, we rely on access control lists and tenant-issued credentials in order to natively authorize file access and we avoid techniques that complicate file sharing.

3.2.3 Efficiency

Our architecture must also be scalable and maintain high performance standards. Thus, we rely on an object-based distributed filesystem to handle the storage requirements of clients (e.g. virtual machines) belonging to different tenants.

As shown in Figure 3.2(a), a collection of object storage servers (OSDs) are responsible to redundantly store the data and metadata in object form. In order to provide scalability to metadata operations, metadata management is separated from the storage of data. Multiple metadata servers (MDSs) manage metadata, and achieve locality and load balancing by partitioning over the object servers the name, data index, and access permissions of different files. Each metadata server can manage a different portion of the namespace for better scalability. Namespace portions can also overlap to the same MDS for better redundancy. Finally, multiple monitors (MONs) are used to manage the whole system, identify component failures, and authenticate the different system components. The system can flexible manage secure access to stored objects with help of the operating system at each object server [67, 70].

Access control decisions happen at the MDS. Object storage servers have no implicit knowledge of access privileges or authorizations. Thus, the MDS authorizes a client request and provides the client with a *capability* [29]. A capability is a token of authority that specifies the access rights that a particular principal has over a particular system resource (e.g. a file). The client presents the capability to the OSD, which according to the policy that is specified on the received capability replies to the client with the appropriate data (see Figure 3.2(b)).

Existing security solutions that rely on capabilities for access control have been criticized for their limited scalability: the number of security operations is strongly tied to the number of users, files, and requests. However, a recent work from Leung et al. [28] solves this problem with the Maat protocol. More specifically, for a single tenant, the extended capability of the Maat protocol authorizes I/O for any number of users and files in petabyte filesystems, is cryptographically secure, and maintains fixed size capabilities through Merkle hash trees. Our proposed architecture is compatible with such extended capabilities.

3.2.4 Compatibility

Our architecture must be compatible with existing scalable and reliable filesystems. For this reason, we use traditional structures and mechanisms that are made available in the most of the current widely used distributed filesystems.

3.2.5 Manageability

Finally, our architecture should permit system administrators to efficiently manage the filesystem in terms of performing administrative tasks, managing tenants, and specifying access control policies for different tenants. Thus, we rely on a file-level interface, which provides more manageability opportunities in comparison to a block-level interface. In addition, we avoid techniques that complicate manageability tasks.

3.3 Trust and threat model

The clients and servers of the filesystem all run in one datacenter that is physically protected and operated by an independent provider. A secure co-processor certifies the software stack on each physical host[1]. A central monitor establishes the trust of the infrastructure from the integrity of the participating nodes. Public keys (or hashes thereof) uniquely identify tenants, principals, and services. The nodes securely communicate over temporary symmetric keys dynamically agreed upon via public-key cryptography. The private keys of principals and services are permanently stored in encrypted form and only appear in clear-text form at the volatile memory of authorized nodes. Before the reallocation of host memory across different nodes, the memory contents are scrubbed to prevent information leakage.

The filesystem protects the confidentiality and integrity of stored data and metadata by restricting access to authorized principals. We assume that the provider has no malicious intent to compromise the system security. However, there may be other reasons (e.g. poor security practices) for which the provider is not trusted for particular applications. In that case, the tenant may externally apply known techniques of encryption, hashing, and auditing to achieve end-to-end confidentiality, integrity, and freshness [42]. Our present study targets filesystem access control without any explicit attempt to provide solutions for public-key distribution, denial of service, and traffic analysis. Finally, we do not address general distributed processing, which involves multitenant sharing of resources other than storage (e.g. computation).

3.4 Summary

In the present study we are particularly interested to design a more efficient access control architecture for multitenant shared storage at the file level. On the one hand, our proposed architecture must securely isolate the identity space and access control of principals from different tenants, while on the other should provide opportunities for flexible file sharing, more efficient manageability, and high scalability. In addition, our proposed architecture

[1]For example, hash chain generated by a Trusted Platform Module [42].

must be compatible with existing scalable filesystems. For performance reasons we rely on an object-based distributed filesystem. In the next chapter we introduce an access control architecture for multitenant filesystems that meets the specified criteria.

Chapter 4

System design

In this chapter we introduce a more efficient architecture for multitenant shared storage at the file level that combines tenant isolation with native access control. Our architecture is backwards compatible to object-based filesystems and meets the goals that we set in the previous chapter.

4.1 Secure multitenancy

A primary objective in a multitenant environment is to securely isolate the namespaces of different tenants. Tenant isolation is explicitly associated with the mechanisms used to

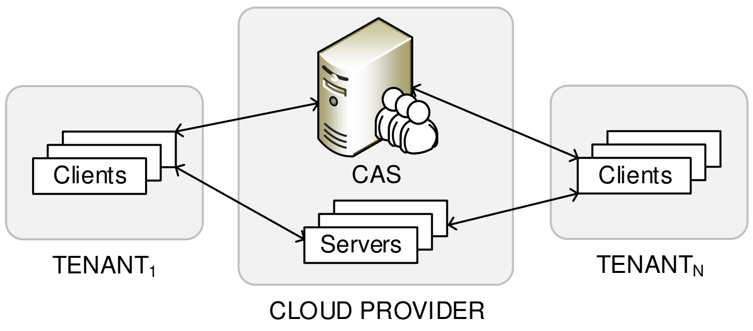


Figure 4.1: The centralized approach: All principals are registered to a central directory.

identify tenants and principals. Identification is the process in which an entity establishes its identity and is securely identified by an authentication server. Identification namespaces can be local or global in scope and each identity has a valid meaning only in the namespace where it is defined. In addition, depending on their scope, identities must be locally or globally unique. In traditional decentralized, distributed filesystems principals are identified by their corresponding public keys. However, a principal's public key needs to be certified[1]. A common approach to certifying a principal's public key is for a *certification authority* (CA) to issue a certificate that contains the principal's name, its public key, and other attributes, such as the certificate's starting date and time, a signature verification key, and the issuer. The CA might be run by a local system administrator; or it might be a remote trusted service.

First, in the following three subsections, we describe possible approaches to add support for secure multitenancy in a file-level storage system. Then, in subsection 4.1.4 we present an outline of our approach.

4.1.1 Tenant isolation with centralized identification

Identity collisions that described in subsection 2.7.2 can be prevented by delegating the identity management to a shared service like Kerberos [59], establishing a new centrally administered ID space which can be shared between clients and services. Thus, instead of relying to their local identification services, tenants are required to register their principals to the provider's identity service, as shown in Figure 4.1.

Inter-tenant file sharing is straightforward when a central directory is used. The

[1]That is a key must be securely binded to a particular principal.

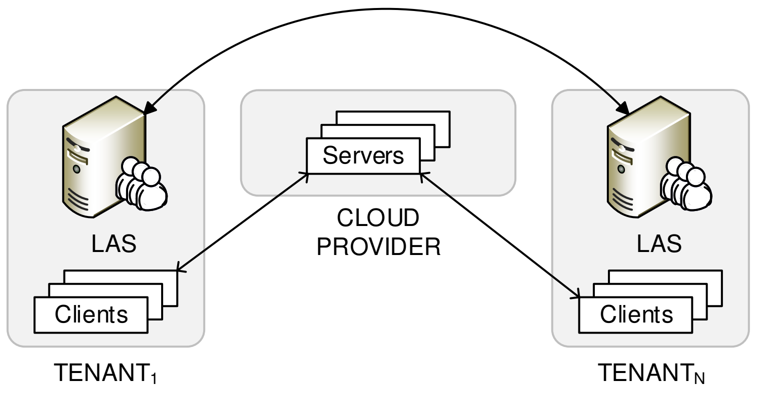


Figure 4.2: Decentralized authentication with local authentication servers (LAS).

central directory is trusted by all the involved parties, while the user and group identifier assignments are kept consistent, because the task of identity management is outsourced to the central directory. Thus, a tenant's principal could use existing techniques to share files with principals of different tenants.

However, such an approach is unrealistic for the tenants of a cloud provider due to scalability and security challenges incurred by the enormous number of users involved. Furthermore, tenants may prefer to run their own identity management systems and would thus be forced to support two such systems simultaneously. What is more, this solution is unrealistic for a cloud environment, where tenants do not trust the cloud provider and other tenants.

4.1.2 Tenant isolation with public-key identification

Another possible approach to prevent identity collisions is to rely on a Public Key Infrastructure (PKI) for principal identification and authentication [40, 24]. In this approach principals could be identified directly by their public keys and a trusted authority can associate a public key with a particular principal.

As shown in Figure 4.2, each administrative domain could operate a local authentication server and trust remote authentication servers of different domains [40, 22]. Each local authentication server may establish a list of identities for local users and groups, and upon request, might return them to the fileserver as credentials. Then, the fileserver can issue these credentials for access control decisions. Inter-tenant file sharing is straightforward. Users can directly list remote users on each file in order to grant them access permissions. The user's local authentication server might prefetch and cache users and group definitions of remote authentication servers belonging to different domains. Thus, during file access, the authentication server can establish identities for principals based on local information.

The above approach enables data sharing across organizational boundaries. However, a cloud environment introduces unique characteristics that make this solution inapplicable to such an environment. First, in order to properly isolate each tenant, there must be a second mechanism that associates each principal's public key with a specific tenant. Second, for manageability reasons, tenants might prefer to use their own identity schemes. Third, this approach alone does not take into account that a single namespace is shared between different tenants. Hence, it does not deal with namespace limitation. Fourth, a tenant's local authentication server needs to trust the remote authentication servers of other tenants. This is unrealistic for an environment such as the cloud where tenants possibly don't trust each other.

4.1.3 Tenant isolation with identity mapping techniques

Identity mapping, a well-known technique from the area of grid computing [14, 58, 65] can be used to solve the identity collision problem that arises when a namespace is shared between multiple parties. Identity mapping allows a fileserver to map incoming UIDs or GIDs from any tenant to the server's own known UIDs and GIDs. In addition, different ranges of server IDs can be assigned to different tenants, in order to provide tenant isolation.

Figure 4.3 shows an example of several local-to-global mappings. First, tenant's A UIDs 100–500 are mapped to fileserver UIDs 2000–2500. Second, tenant's B UIDs 100–200 are mapped to fileserver UIDs 5000–5100 preventing a possible collision with tenant's A identities. Third, tenant's C UIDs 0–100 are mapped to fileserver UIDs 6000–6100. Note that identity mapping enables root-squashing: Any incoming UID (or GID) 0 is mapped to another number that does not have superuser privileges. In our example, the UID 0 of tenant C is mapped to the fileserver UID 6000.

Identity mapping is performed bidirectionally. Forward mapping is performed when a client sends a request to the fileserver. In this case, the fileserver maps the user's local

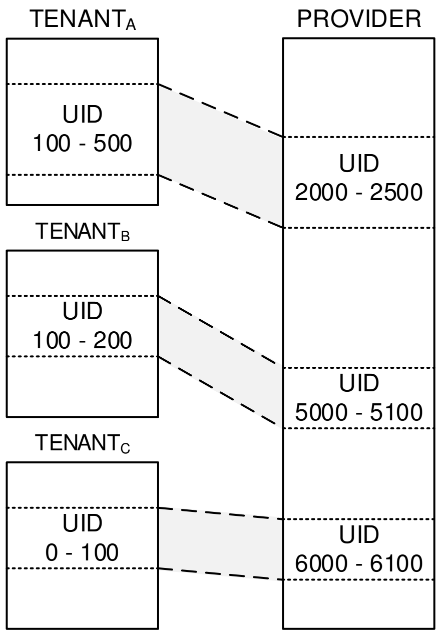


Figure 4.3: The identity mapping technique: The local identity space of each tenant is mapped to a different range of the provider's identity space.

UID and GID to the assigned global UID and GID. Reverse mapping is performed when the fileserver replies to the client. The server maps the user's global UID and GID to the corresponding local UID and GID.

Identity mapping solutions successfully isolate the identity spaces of different tenants and thus they have been recently applied to cloud filesystems [11]. However, such identity mapping techniques have been recently criticized as cause for limited scalability [32, 16]. In addition, the mapping of remote users to existing local user classes also poses the threat of implicit rights amplification, where users requiring only limited rights are given stronger than necessary. Moreover, the specification and the enforcement of the access control policies could become a cumbersome task, because each server maintains its own separate mappings. Even if the mappings are coordinated across different servers, access control policies could not be specified in the granularity of users, because each tenant maintains a variable list of users. In fact, users can not express access control policies that refer to identities that the fileservers have not yet encountered [60]. Thus, identity mapping techniques complicate or disable inter-tenant file sharing completely.

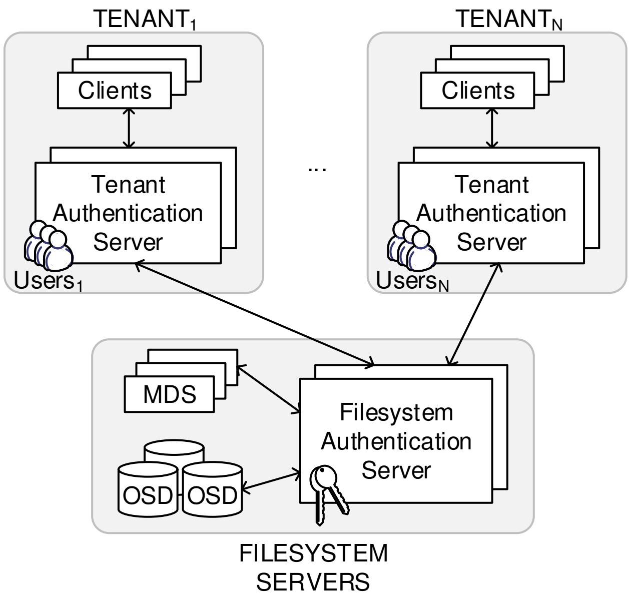
Furthermore, the common practice of mapping remote users to existing local user accounts in order to isolate different namespaces poses extra manageability difficulties. The global-to-local mappings are either created manually by administrators [65], or the administrators only specify the remote lower and higher bounds of ID values. In this case, the mappings themselves are updated at runtime [11]. In the first approach, when thousands of tenant users must be mapped, manually creating the mappings can be a daunting task. In the latter, users can not express access control that refers to identities that the fileserver has not yet encountered and as a result the global-to-local mappings for these identities do not exist. In both of the above approaches it is not possible to maintain automatically common user accounts and global-to-local mappings between multiple fileservers. This leaves maintenance and synchronization of global-to-local mappings as a manual process, or leads to the development of new tools to automate this task. Finally, the UID space of the fileservers can possibly run out of numbers, because a cloud filesystem involves an enormous number of end users.

4.1.4 An outline of our approach

Instead of registering tenant principals into a centralized directory service, or using identity mapping, which faces scalability issues and complicates file sharing as well as manageability tasks, we rely on local authentication servers where each tenant certifies local clients and principals. The local authentication server of each tenant in turn is certified by a global authentication service. Tenants can use their own identity mechanism to name principals locally. However, local identities can be associated with global identities in order to permit inter-tenant file sharing. In addition, we differentiate our solution from previous approaches in that we combine local authentication servers with native access control by carefully storing identities and access control information directly on the filesystem. In order to preserve tenant isolation, identities that refer to principals that belong to different tenants are stored on separate places. Then, each of these places is pinned to a distinct tenant and thus the namespace of each tenant is properly isolated.

4.2 Architectural overview

Our proposed architecture consists of five core components, as shown in Figure 4.4: the Clients, the Tenant Authentication Servers (TAS), the Metadata Servers (MDS), the Object Storage Servers (OSD), and the Filesystem Authentication Servers (FAS). Next,



TENANT$_1$
TENANT$_N$
Clients
Clients
Tenant Authentication Server
Tenant Authentication Server
...
Users$_1$
Users$_N$
MDS
Filesystem Authentication Server
OSD
OSD
FILESYSTEM SERVERS

Figure 4.4: An architectural overview of our proposed system.

we provide details for each of the above components.

4.2.1 Tenant Authentication Server

Every tenant certifies the identity of local clients and principals with its own authentication server, that is securely registered to the filesystem authentication servers. Tenants, as well as principals and groups maintain their own public/private key pairs. The tenant authentication server securely maintains the tenant's private key, as well as the private keys of each principal. Tenants, as well as clients and principals are uniquely identified by their public ID. This public ID might be a hash of their public key.

Tenant administrators, for reasons of privacy or administration complexity, have the flexibility to use their own identification mechanism to locally identify principals. However, in order for cross-domain file sharing to be possible, there must be a mechanism which certifies that a particular principal belongs to a particular tenant. For this reason, the tenant authentication server issues a credential to prove that a public principal ID belongs to a particular tenant. Specifically, this credential binds the principal's public ID with the tenant's ID and other local identity attributes, such as the principal's local ID. Credentials contain only signed identity attributes of the principals rather than policy statements with permitted actions over the requested file resources.

The TAS is a critical component of the overall system to maintain security, operation, and scalability. Hence, it needs to be distributed and replicated. Having multiple authentication servers per tenant, not only guarantees redundancy of the stored information in case one of them suffers an outage, but also guarantees availability and scalability to a large number of users.

For the above reasons, it is possible to build the TAS on top of a distributed key-value-store (e.g. Cassandra), which supports replication for fault-tolerance, is decentralized (no single point of failure), and scalable. Each stored entry is a key/value pair and corresponds to a tenant, client, principal, or group. The key is the public ID of the particular entry, while the value is the quintet:

$$< type, publickey, privatekey, localidentity, metadata >$$

The "*type*" element specifies the type of the entity and can be one of: tenant, client, user, or group. The "*publickey*" and the "*privatekey*" elements correspond to the public and the private keys of the entity respectively. The "*localidentity*" entry is used only for user and group entities, and corresponds to a local identity that is assigned to that entity by the tenant's local identification mechanism. Finally, the "*metadata*" entry can be used by the tenant to store additional information for an entity.

4.2.2 Client

The client component represents the interface between user processes and the filesystem, and provides a POSIX-like interface to the users. In addition to the POSIX-like interface, the client provides tools for managing the tenant's namespace and granting or revoking access to other tenants. Each client has a public/private key pair and is registered to a tenant authentication server of a single tenant. A trusted monitor at the datacenter certifies the integrity of the software stack running at the client, in order to harden a potential compromise of the client component.

4.2.3 Object Storage Server

Object storage servers (OSDs) are responsible for storing file data and metadata. The content of a single file is represented by one or more objects. Object storage servers are responsible to perform the mapping of these objects to blocks on their local filesytems.

For security reasons, each OSD maintains its own public/private key pair and is identified by a hash of its public key. In addition, object storage servers are securely registered to the filesystem authentication servers.

Object storage servers are also responsible for data migration, replication, failure detection, and failure recovery. Every object is written to the primary OSD first, and then the primary replicates it to one or several replicas to ensure redundancy. This replication can be synchronous in order to guarantee the availability of a new or updated object, before the client is notified that a write operation has completed. An object storage server writes the new or updated object to its local journal before replicating it to a replica object storage server.

4.2.4 Metadata Server

The Metadata server (MDS) is responsible to manage the filesystem namespace and provide POSIX semantics to clients. For availability and performance reasons, there may be multiple metadata servers running on different hosts. Each MDS stores metadata on object storage servers in the form of objects. In fact, the MDS itself does not provide metadata storage, but works as an intelligent metadata cache.

Each MDS maintains its own public/private key pair and is identified by a hash of its public key. In addition, it is securely registered to the filesystem authentication servers. Metadata servers manage the location of metadata and also decide where to store new data. Furthermore, the filesystem namespace is split into different portions. For scalability reasons it is possible to assign each portion to a different MDS. Namespace portions can also overlap for redundancy reasons. This mapping of namespace portions to metadata servers can be performed using dynamic subtree partitioning algorithms [69].

4.2.5 Filesystem Authentication Server

The Filesystem Authentication Server (FAS) certifies the identity of metadata servers, object storage servers, and tenant authentication servers. It is also responsible to manage the operation of the whole system and identify component failures. For this task, the FAS keeps information in the form of maps. For instance, to manage the cluster of object storage servers it maintains an OSD map that stores information about the location of the object storage servers and their current state.

Having multiple filesystem authentication servers is essential to guarantee redundancy of the stored information and high availability. However, this means that the stored information must be kept consistent across all of them. For this reason, it is possible for each server to use a distributed consensus algorithm, like Paxos [26], in combination with a local key-value-store (like the architecture of Ceph's monitor[2]). Each time a map is modified, a new version is created and run through a quorum of servers. Only when a majority acknowledge the change, the primary server will store the new version to its local key-value-store and the new version will be considered committed.

4.3 Authentication

Authentication is the process of verification that an individual or an entity is who it claims to be. In a traditional distributed filesystem, all principals are registered to a central directory service by utilizing an existing security infrastructure, such as Kerberos [59]. If a principal is securely identified by the directory, it receives a *ticket* to contact the filesystem. A ticket is a cryptographically secure, time stamped data structure that contains authentication and other information about a specific proposed interaction between a client and a server. On the other hand, decentralized distributed filesystems avoid the requirement of a central directory that knows all principals and group definitions. Different administrative domains maintain their own principals and policies in a local directory. Hence, local directories issue credentials for registered principals.

As mentioned in subsection 4.2.1, tenants certify the identity of local clients and principals with their own authentication servers, which are securely registered to the filesystem authentication servers. When a TAS authenticates to the filesystem authentication server, it receives a ticket that grants access to the metadata servers.

A client talks with the TAS and receives a secret key to decrypt its private key. Then, it uses public-key cryptography to establish secure connection with the TAS. A principal connects to a particular client and provides a secret password for authentication by a tenant authentication server that also stores the password in encrypted form. After the

[2]http://ceph.com/docs/master/rados/configuration/mon-config-ref/. Accessed: 2013-08-19.

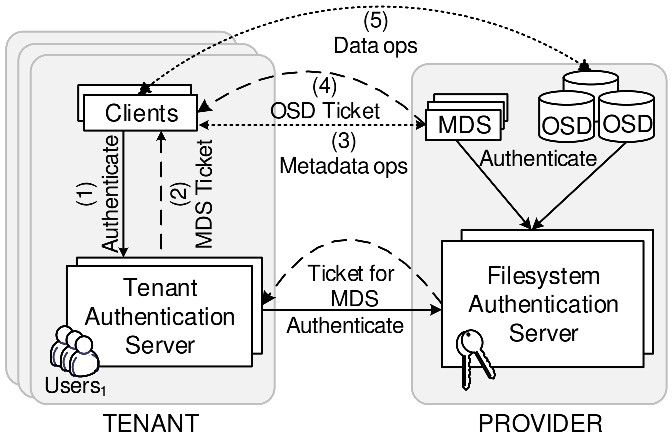


Figure 4.5: The authentication architecture.

successful authentication, the principal receives a secret key to decrypt a respective private key that is made accessible at the client.

Upon authentication, the TAS delegates to the principal the ticket that grants access to the metadata servers (steps 1 and 2 in Figure 4.5). Then, the principal has everything it needs to perform metadata operations. For example, lets assume that a principal needs to access a particular file. Using the client, the principal issues a metadata request to a metadata server (step 3 in Figure 4.5). The request carries the MDS ticket, as well as a credential that proves the identity of the principal. The MDS verifies the ticket and the principal's credential and upon correct verification responds with a map that contains the object storage servers and the specific locations of the file's fragments. In addition, the MDS embeds to its reply the necessary OSD tickets. Finally, the principal receives the metadata server's reply and issues file operations to the OSDs using the received OSD tickets (steps 4 and 5 in Figure 4.5). Request freshness is ensured with a client-provided nonce that the server returns modified according to a known function (e.g. increment by one).

4.4 Authorization

Authorization is a security mechanism used to determine principal's access rights related to system resources. Access rights are organized as a large matrix called *Access Control Matrix*. Each row of the matrix refers to a subject (e.g. a user or a group) and each column

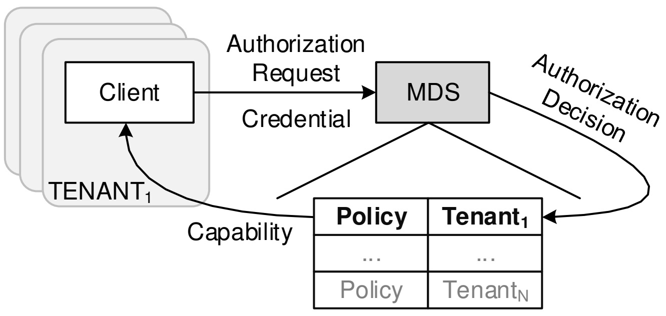


Figure 4.6: The authorization architecture.

refers to an object (a resource, e.g. a file). Each cell lists the rights that a particular subject has over a particular object. A column of this matrix is a list containing all the subjects that can access the object, and how. This list is called the *Access Control List* or *ACL*. In fact, an ACL is associated with each file and lists all principals authorized to access it along with their permissions. A principal's identity must be known before access rights can be looked up in the ACL. Thus, authorization depends on prior authentication. On the other hand, a row of this matrix is a list that associates with each subject a list of objects that may be accessed, along with the permitted permissions on each object. This list is called a capability. In contrast to ACLs, capabilities do not require explicit authentication.

The filesystem grants to a principal a permitted file access according to the tenant-issued credential. The authorization policy is specified in ACLs maintained by the filesystem. The rules of principals that belong to different tenants and the provider are respectively maintained across separate ACLs (Figure 4.6). Thus, our architecture successfully isolates the namespaces of different tenants and the provider's without the need of identity mapping tables. Each file is associated with a list of ACLs, one for each tenant that can access the file. The ACL of a tenant for a particular file is a list of entries; each entry consists of a principal's identity and a representation of the permitted actions. There is a separate ACL where the filesystem maintains the permissions of its native principals (the provider's principals). A file can be configured as private or shared across the principals of a single or multiple tenants.

For administration purposes the system provides selective access to metadata in the form of views. We call this technique *namespace filtering.* Namespace filtering allows each tenant to access a filtered view of the shared namespace. The filesystem administrator

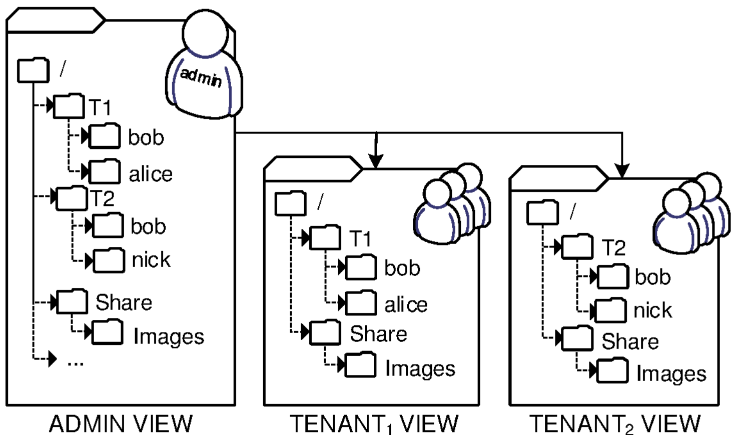


Figure 4.7: Namespace fitlering: admin and tenant view of the filesystem metadata.

has access to the *admin view*, which allows specification of permissions at the granularity of entire tenants or principals. In fact, the administrator can use policies to provide namespace limits to tenants. Instead, the *tenant view* allows a tenant administrator to configure metadata made accessible to the tenant by the provider's administrator. Depending on whether it belongs to the provider or tenant, respectively, a principal can only access a subset of the admin or tenant view filtered according to the applicable permissions. Thus, by preventing a principal to name an object through namespace filtering, the system can prevent access to the object (Figure 4.7).

4.5 Optimizations

The number of files that large-scale storage systems need to store is increasing rapidly due to the growing number of end-users involved. Associating an ACL to each file leads to an enormous number of ACLs that the system needs to store and manage, and can cost considerable storage space and performance overheads.

According to a recent study from Smetters et al. [57] users rarely change the access rights of single files. They prefer to add new files to an existing folder with its permissions already set. Hence, new added files inherit the permissions of the parent folder. The authors also state that permission inheritance is consistent with "best practice" recommendations for using access control settings, which recommends setting permissions rarely and rely on inheritance to manage most controls. We exploit this observation in

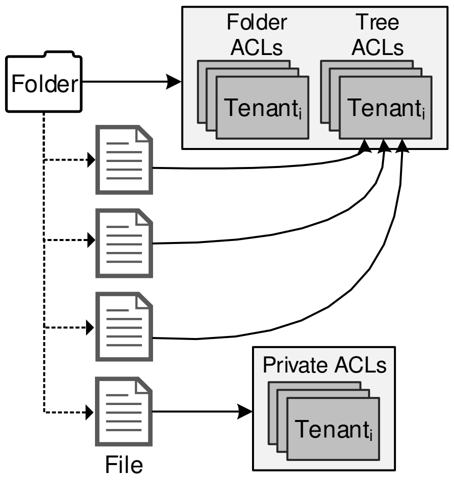


Figure 4.8: ACL sharing with tree ACLs.

order to reduce the size of each file's metadata stored on the object storage servers and managed by metadata servers.

More specifically, we associate with each folder two ACLs per tenant, a *folder ACL* and a *tree ACL*. The folder ACL controls access to the folder as before. On the other hand, the tree ACL controls access to the folder's contents. Newly created files share their parent's tree ACL, as shown in Figure 4.8. However, when a user explicitly sets permissions on a particular file, then a new *private ACL* is created for this file, and the file no longer shares the tree ACL with its parent folder (for example see the last file in Figure 4.8). A child folder inherits its parent's tree ACL.

The tree ACLs can be set by users with the help of a special tool, or they can be set automatically. In the second case, they can be updated either statically or dynamically. In the static updating scheme, when a new file is created in an empty folder its private ACL is promoted to a tree ACL. From that point, the tree ACL does not change, unless a user explicitly change it. However, if the majority of files in the same folder contain private ACLs, then the benefits of ACL sharing are being lost. To mitigate this problem, in the dynamic updating scheme it is possible to update the tree ACL taking into account the majority of private ACLs. In this manner, the most frequent occurring private ACL is first promoted to a tree ACL and then is removed from all the files that contain it. Our current prototype implementation only supports the static updating scheme, however, it is straightforward to implement the dynamic scheme in a future version.

Associating a tree ACL with each folder allows us to take advantage of the collocation of file metadata from a range of files that reside under the same folder. When the MDS needs to authorize access to a given file, it fetches the whole object that contains the metadata of all the parent folder's contents (if there isn't already in its cache). This object also contains the tree ACL that controls access to the file. Thus, the MDS does not need to fetch extra objects from the object storage servers.

4.6 Security analysis

In this section we review the security model presented in this work. Below, we list the security players of our architecture along with their permitted actions:

a) **Principals.** They can be distinguished to native filesystem principals or tenant principals. We consider native principals as trusted administrators who can create and destroy data, and specify access policies by delegating read and write access to other principals, or revoke another principal's privilege to access data. On the other hand, tenant principals are untrusted users (or groups of them therefore) who belong to a particular tenant. They can read and write data once they have the appropriate permissions. They can also delegate access to other users of the same tenant.

b) **Clients.** They are trusted entities used by principals to access the filesystem.

c) **Storage servers.** They are trusted storage devices which store and return data and metadata upon request.

d) **Metadata servers.** They are trusted servers which manage filesystem metadata and access control policies. They also allow traversal of the filesystem namespace. Metadata servers are responsible to securely separate the namespaces of different tenants from each other, as well as from the native filesystem namespace.

e) **Authentication servers.** They are trusted servers which certify other players. There are two kinds of authentication servers: Tenant authentication servers which certify local clients and principals and a global filesystem authentication service which certifies tenant authentication servers, as well as the filesystem's storage and metadata servers.

f) **Wire.** It transfers data between players.

We define an attacker to be an entity who tries to perform operations other than those that is authorized to. The security model presented in this work considers two types of attacks: intra-tenant and cross-tenant attacks. Below, we name a number of possible attacks that may be mounted on the data or the metadata:

Attack on the wire. An attacker may manage to intercept a ticket that allows access to a metadata or storage server. However, tickets are encrypted and therefore cannot be forged. An attacker may also manage to capture an authorization credential. Credentials are signed bindings of public keys with specific identity attributes. If such a credential is intercepted, the only information that may be obtained is that a particular principal with public key A belongs to the tenant with public key B. In addition, the attacker can not forge the credential because it is signed. Thus, the specification of the authorized tenant and principal in a signed credential along with a secure exchange among nodes prevent an attacker principal from getting unapproved access to the data and metadata of other principals from the same or different tenant. The attacker might also tamper with network traffic and launch a denial-of-service attack. Freshness of network communications to protect against replay attacks or injection of non-authentic data is achieved through message nonces and timestamps.

Attack on a client or tenant principal. An attacker may manage to penetrate a client and guess the password of a tenant's principal. He may also mount a man-in-the-middle attack against a principal in order to learn her password. In both cases, the attacker can access the principal's data. Yet, the attack is either limited to the principal's data, or if the principal's account has local administrative rights, it will affect all principals of the victim's tenant. However, the attack is confined within the tenant. In fact, the attacker is still unable to modify the system wide access policy, which affects the native principals of the filesystem or the principals of other tenants.

Attack by a revoked tenant. When a filesystem administrator revokes access of a tenant to a collection of files, the tenant's principals are not able to access these files anymore. In fact, the filesystem keeps a separate ACL for each tenant, and when a tenant's access to a particular file is revoked, the corresponding ACL is removed from this file. Hence, future access to this file by principals belonging to the revoked tenant can not be authorized.

Attack on a native filesystem principal. While native filesystem principals are considered trusted, an attacker may manage to gain access to a native principal's account. In this case, the attacker will be able to gain complete access to the data of all tenants. Special protection measures make harder to forge the identity of administrators, for example by disabling access to the respective accounts from outside the datacenter.

Attack on tenants' data. Data is stored as cleartext on the storage servers, which implies that tenants trust the servers and their administrators. However, there may be reasons, such as poor administration practices or poor disposal policies of defective storage devices, for which the provider is not trusted for critical data. In this case, a tenant may externally apply data-protection techniques, such as encryption, to strengthen the confidentiality of their data.

4.7 Summary

In the present study we design a more efficient access control architecture for multitenant shared storage at the file level. On the one hand, our proposed architecture must securely isolate the identity space and access control of principals from different tenants, while on the other should provide opportunities for flexible file sharing, more efficient manageability, and high performance. For performance reasons we rely on an object-based distributed filesystem.

Instead of relying on a centralized authentication service where each tenant user is registered, or using mapping techniques for identity translation which face scalability issues, in the proposed architecture each tenant has a local tenant authentication server which certifies local clients and principals. The tenant authentication server in turn is certified by the global filesystem authentication service. For authorization purposes each file maintains different ACLs for each tenant. By combining per-tenant ACLs and namespace management, we avoid extensive explicit access control infrastructure and mapping layers, because each tenant sees a masked view of the shared namespace through namespace filtering.

Finally, as an optimization instead of assigning a separate ACL to each file, we assign a tree ACL to the parent folder which controls access to all the folder's contents. However, it is still possible to explicitly set permissions on a particular file. In this case, the file's private ACL controls access to this file which in turn is different from the tree ACL.

Chapter 5

Implementation details

In the present chapter we describe our implementation of the proposed multitenant access control architecture over a distributed, object-based filesystem. The prototype implementation is based on Ceph, a flexible prototyping platform with scalable management of metadata and extended attributes.

5.1 Implementation overview

We base our implementation on Ceph, an open source distributed object-based filesystem written in C++ and C. We call our prototype Dike[1]. We developed two versions of Dike: One that does not support ACL sharing and another one that supports it. For

[1]In ancient Greek culture, Dike was the spirit of moral order and fair judgment based on immemorial custom, in the sense of socially enforced norms and conventional rules [72].

	MDS	**Client**	**Messages**	**Other**
Dike without ACL sharing				
Comments	191	128	0	139
Code	534	274	24	415
Total	725	402	24	554
Dike with ACL sharing				
Comments	239	128	0	139
Code	803	274	24	416
Total	1402	402	24	555

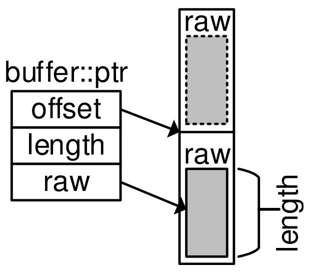
Table 5.1: Number of added and modified lines of source code in different parts of Ceph.

the needs of our implementation, we mainly modified the MDS component, as well as the user space client which utilizes Filesystems in User SpacE (FUSE) [15] in order to provide filesystem access to users. We also modified some additional parts of Ceph like the message structures. Finally, we developed administrative tools for creating tenants and assigning tenant permissions.

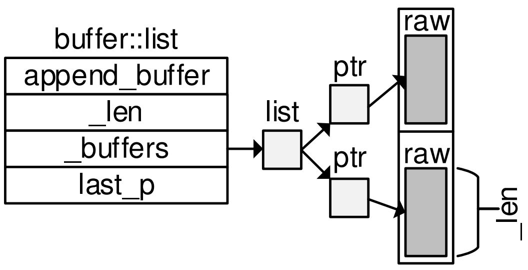
As the Table 5.1 shows, the source code size of the Dike prototype without ACL sharing is roughly 1705 lines (C++ code and comments), from which the added or modified lines in the MDS component are 725, in the client component are 402, in the message structures are 24, and in other parts of Ceph, including newly developed administrative tools, are 554. On the other hand, the source code size of the Dike prototype with ACL sharing is roughly 2023 lines, from which the added or modified lines in the MDS component are 1042, in the client component are 402, and in the message structures are 24. Finally, we added or modified 555 lines of code in other parts of Ceph, including newly created administrative tools.

5.2 Key structures of Ceph

A key structure of Ceph is called *buffer* and is used to process data in memory. The actual data is stored in `buffer::raw` opaque objects. They are allocated with malloc, new, or reusing a pointer provided by the caller. A variant of the malloc constructor provides an



(a) buffer::ptr



(b) buffer::list

Figure 5.1: Key structures of Ceph that are used to process data in memory.

area that is page aligned on `CEPH_PAGE_SIZE`, which is equal to the system's page size.

The `buffer::raw` area can only be accessed through a `buffer::ptr` pointer. As the Figure 5.1(a) shows, it addresses the `buffer::raw` bytes in the range $[offset, offset + length]$. Bytes can be copied in or out within the $[offset, offset + length]$ range. In case the underlying `buffer::raw` extends beyond $offset + length$, more bytes can be appended.

The `buffer::ptr` methods are very flexible and can be used to implement more complex data structures such as lists (see Figure 5.1(b)). In particular, the `buffer::list` structure that Ceph provides is a list of `buffer::ptr` pointers.

The extended attributes are managed as key/value pairs stored in a C++ map structure (red-black tree). Each entry of the map corresponds to a key/value pair, where the key is a name and the value is a `buffer::ptr` data structure which keeps the data.

Another important structure of Ceph is the map structure (implemented as a C++ STL map) that is used to maintain information about the fragments of each cached folder inode. Each entry of the map is a set of key/value pairs, where the key corresponds to the fragment ID and the value corresponds to the actual fragment data. In addition, the folder inode contains a folder fragment tree which is always consistent with the folder fragment map. This tree represents an entire namespace and its partitions. It essentially informs the MDS where fragments are split into other fragments and by how much. The goal is to use a binary split strategy to partition the namespace. The MDS caches a pre-configured number of inodes. This cache size provides a limit on how many files can be in use simultaneously with good performance, but not on total number of files in the system. Furthermore, whenever a folder is read off disk, the MDS needs to be able to hold the whole folder in memory, and if the folder holds more entries than the MDS cache can hold, then the overall performance degrades.

5.3 Multitenant access control

Into Ceph we added native support for multitenant access control according to the proposed design. We deliberately avoid global-to-local identity translations because they introduce performance bottlenecks, replica inconsistencies, and impersonation risks.

A registered client shares a secret key with the monitor. When a user requests from the client to mount a filesystem, the client authenticates to the monitor and receives a session key encrypted with the secret key. The session key is used by the client to securely request from the monitor a ticket that authenticates the client to the MDSs and OSDs. The ticket is encrypted with a secret key that the monitor shares with the MDSs and OSDs. The client uses this ticket to initiate a new session with the MDS. The MDS receives from the client a message of type *MClientSession* and sends back the capability (i.e. ticket) that enables access to the root folder at the OSDs. The returned capability contains the inode number, the permitted operations, the replication factor, and the striping method of a file. From the capability the client derives an object identifier, which is hashed to the placement group of OSDs that contain the object replicas.

The session between a client and the filesystem is limited to only serve the permitted actions of the requesting principal. In a filesystem mount request to an MDS, a client has to securely identify the respective tenant. We derive a unique tenant identifier (TID) by applying the RIPEMD-160 cryptographic hash function on the public key of the tenant. Then, we include the TID into an expanded *MClientSession* request and send it to the MDS over a secure session. For authorization purposes the request should additionally carry a tenant-issued credential that we do not yet support in our prototype.

The MDS extracts the TID from the *MClientSession* message and stores it as a field of the session state. Our current implementation only supports Unix-like permissions of individual users and groups, but makes it straightforward to add access-control lists in a future version. We facilitate the system administration with the support of multiple filesystem views. Based on the supplied TID, a client obtains tenant view of the filesystem

Method	**Description**
`bool check_tenant_perm()`	Check tenant permission
`void grant_tenant_perm()`	Grant tenant permission
`void set_unix_uid()`	Set user ID
`void set_unix_gid()`	Set group ID
`void set_unix_mode()`	Set file permissions
`uid_t get_unix_uid()`	Return user ID
`gid_t get_unix_gid()`	Return group ID
`mode_t get_unix_mode()`	Return file permissions

Table 5.2: The methods that we added into the class *CInode* to manage the tenant permissions of an Inode.

for access by a principal of the tenant.

For global configuration needs, we also provide the admin view that enables full access permissions to the filesystem. We extended the *CInode* class of Ceph with eight new operations to set and retrieve the permissions of tenants and individual principals as shown in Table 5.2. When the tenant view is used, the permission attributes are stored in the extended attributes of the filesystem; otherwise the regular fields of the inode are used (see Figure 5.2). We use as key the string "$TID \| permtype$" where TID is the tenant identifier and `permtype` is set to `"UNIX"` for Unix permissions or `"ACL"` for the ACL model. In the Unix model the value of the pair can be set to `"UID:GID:mode"`: UID and GID refer to the user and group ID, while mode represents the Unix file permissions. We modified all the filesystem functions of the original Ceph related to permissions handling, including the constructor of a new inode. If the client uses the admin view, then we directly update the regular inode of the filesystem. Otherwise we save the user/group IDs and the file permissions into extended attributes keyed under TID; we also update the regular inode of the filesystem according to the user/group IDs and file mode of the parent inode. Thus, special files (e.g. block device files) are stored as regular files at the filesever and they are presented as special files only in the particular tenant view.

A capability is only sent to a client whose tenant has access to the file. In order to allow or deny a file access to a client, we modified the returned capability to include

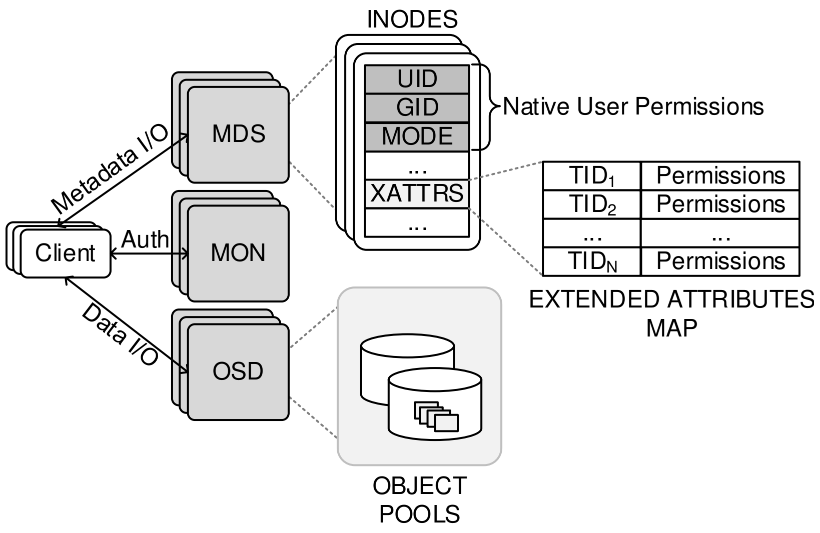


Figure 5.2: Prototype implementation of the proposed multitenant access control architecture.

the tenant identifier and the respective file ownership metadata. A client cannot directly access the extended attributes that contain access control information; instead only the filesystem is allowed to read and update extended attributes on behalf of authorized client requests.

5.4 Optimizations

Instead of assigning tenant permissions to each file separately we permit a collection of files to inherit the access control information stored in their parent folder. As we explained in section 4.5, we store a tree ACL to each folder which controls access to the folder's contents, in order to reduce the number of ACLs that the system needs to store and manage.

Hence, a folder's extended attributes contain two types of permission attributes for each tenant: (a) the folder permission attributes which control access to the folder, and (b) the tree permission attributes which control access to the folder's contents. We differentiate between folder and tree permissions by extending the value of each attribute in order to contain the attribute type. For example, in the Unix model the value of an attribute can be set to `"TYPE:UID:GID:mode"`, where `TYPE` can be either `"folder"` or `"tree"`.

Access to every newly created file is controlled by the tree permissions stored in its parent folder. However, if a user explicitly set permissions on a particular file, then a new entry will be stored in the file's extended attributes for the user's tenant.

In order to authorize a request to a particular file, the MDS initially checks the file's local extended attributes to find a permission attribute. If a private permission attribute exists, then the MDS authorizes the request according to this attribute. Otherwise, it checks the tree permissions stored in its parent folder's extended attributes.

5.5 Summary

We base our prototype of the proposed architecture on Ceph, an open source, object-based, distributed filesystem. For the needs of our prototype we mainly modify the client and the MDS components of Ceph.

The MDS stores per tenant ACLs in the extended attributes of each file and folder. In a filesystem mount request to an MDS, a client securely identifies the respective tenant. Then, every time that the MDS needs to authorize access to files or folders, it checks the permission attributes of the given tenant. The filesystem's native users, however, are handled separately. For these users, the user/group IDs and the access permissions are stored directly in the regular inode fields. Thus, the filesystem's native users have a complete view of the shared namespace (i.e. the admin view), in contrast to tenant users who have a filtered view (i.e. the tenant view).

As an optimization, we permit files to share a single ACL per tenant which is stored in their parent's extended attributes. This optimization decreases the number of ACLs that the system needs to store and manage.

Chapter 6

Experimental evaluation

We experimentally evaluate our prototype implementation with microbenchmarks and application-level benchmarks to answer the following questions: (a) how much overhead does our multitenant prototype introduce to a single-tenant filesystem, (b) how well does our prototype compare with a cloud filesystem which uses identity mapping techniques, and (c) how much do long ACLs affect system performance and how ACL sharing comes up with this problem.

6.1 Experimentation environment

We developed a prototype of the proposed architecture (which we call Dike) over Ceph version 0.61.4 (Cuttlefish) and we evaluated it on two environments: (a) a local cluster and (b) a cloud computing platform. In each figure we state whether we use the local or the cloud environment for experimentation.

#	CPU	RAM	Disk	Kernel	NET	VMM
MDS						
1	Intel E5345	6 GB	2x250 GB, 7200 RPM	Linux 3.9.3	1 Gbps	-
OSD						
3	Intel E5345	3 GB	2x250 GB, 7200 RPM	Linux 3.9.3	1 Gbps	-
MON						
1	Intel E5345	3 GB	2x250 GB, 7200 RPM	Linux 3.9.3	1 Gbps	-
HOST DOM0						
6	Intel E5345	4 GB	2x500 GB, 7200 RPM	Linux 3.5.5	1 Gbps	Xen 4.2.1
CLIENT DOMU						
36	1 VCPU	512 MB	15 GB root, 2 GB swap	Linux 3.9.3	bridged	-

Table 6.1: Local experimentation environment.

6.1.1 Local testbed

Table 6.1 summarizes the local experimentation environment. It consists of HP ProLiant DL140 G3 server nodes running Debian 6.0 GNU-Linux. We used up to five nodes as filesystem servers and up to six nodes as client hosts. Each filesystem server node is equipped with one quad-core 64-bit Intel Xeon E5345 processor at 2.33 GHz, 3-6 GB RAM, two SATA 250 GB 7.2 KRPM HDs, and runs Linux kernel 3.9.3. The server with the 6 GB of RAM is used as MDS, while the other three are used as OSDs with 2-way replication, and the last one as the cluster monitor. The OSD nodes have their second hard disk formatted with the XFS filesystem and use it to store objects. For journaling purposes they have a 1 GB journal file stored on the first hard disk. On the other hand, client host nodes are equipped with two quad-core 64-bit Intel Xeon E5345 processors at 2.33 GHz, 4 GB RAM, two SATA 500 GB 7.2 KRPM HDs, and run Xen hypervisor (version 4.2.1) and Linux kernel 3.5.5. Each node has one activated gigabit network link.

We use paravirtualized VMs as clients each one set up with a single dedicated CPU core and 512 MB of RAM. The guest OS is Debian 6.0 GNU-Linux with Linux kernel 3.9.3. Each VM has two virtual disks connected to the host through a blktap-2 device: one with a root filesystem (15 GB) and another used as swap space (2 GB). The machines are connected using bridged networking on each host.

Type	CPU	RAM	Disk	Kernel	NET
FILESYSTEM SERVERS					
m1.xlarge	4 VCPU	15 GB	4x420 GB, Root on EBS	Linux 3.9.3	high
MICROBENCHMARK CLIENTS					
t1.micro	1 VCPU	615 MB	Root on EBS	Linux 3.9.3	Very low
APPLICATION LEVEL CLIENTS					
c1.medium	2 VCPU	1.7 GB	1 x 350 GB, Root on EBS	Linux 3.9.3	Moderate

Table 6.2: Cloud computing environment (Amazon Web Services).

6.1.2 Cloud computing platform

We used resources from Amazon EC2 [2] in order to compare our prototype with existing solutions. Our EC2 experiments use "m1.xlarge" virtual machine instances for fileservers (having four 64-bit cores, 15 GB of memory, 4x420 GB ephemeral storage, and high network performance). For clients, we use two types of instances. For microbenchmarks, we use "t1.micro" virtual machine instances with one 64-bit core and 615 MB of memory. On the other hand, for application-level benchmarks, which are more computationally intensive, we use "c1.medium" instances (having two 64-bit cores, 1.7 GB of memory, and moderate network performance). All instances run the Red Hat Enterprise Linux Server release 6.4 (Santiago) with the Linux kernel 3.9.3. In addition, all instances run in the US East region (Table 6.2).

Each time, we have one of GlusterFS[1] (version 3.2.7), HekaFS[2] (version 0.7), Ceph (version 0.61.4), or Dike installed on three fileserver VMs. As the Table 6.3 shows, in the case of Ceph (or Dike) all VMs are OSDs, however, one of the VMs is also the MDS, while another one is both an OSD and the cluster monitor. In the case of GlusterFS (or HekaFS) all VMs are fileservers who manage both data and metadata. At this point, it should be noted that a more fair configuration for Ceph and Dike would be to use three active MDSs, one per VM, because the fileservers of GlusterFS manage both data and metadata. However, the feature of having multiple active MDSs in Ceph is considered

[1]GlusterFS is an open source, distributed file system developed by RedHat. It consists of layers, where features (also known as translators) can be added or removed as per the requirement [10].

[2]HekaFS provides a set of translators to make GlusterFS more suitable as a cloud filesystem. It uses identity mapping techniques to isolate the namespaces of different tenants [11].

Filesystem	Server	#	Type
Ceph/Dike	OSD+MDS	1	m1.xlarge
	OSD+MON	1	m1.xlarge
	OSD	1	m1.xlarge
GlusterFS/HekaFS	File server	3	m1.xlarge

Table 6.3: Different filesystem configurations on AWS.

unstable[3] in the version we are using. Hence, we use only one active MDS in the cases of Ceph and Dike. Finally, all VMs have their first local disk (ephemeral) formatted with the XFS filesystem and use it to store files or objects. In the case of Ceph, the OSD journal is stored on the second local disk of each VM. In all configurations we use a replication factor of three.

6.2 Methodology

Here we explain the experimental methodology used to evaluate Dike. Our analysis concentrates on metadata performance and is performed in three steps. First, we evaluate the overhead imposed by the Dike prototype of multitenant access control in comparison to the single-tenant Ceph. Second, we analyze both the overhead of Dike in comparison to Ceph, as well as the overhead of HekaFS in comparison to GlusterFS on which it is based. Then, we compare these overheads with each other in order to understand which multitenant access control architecture introduces the lowest performance overhead. Finally, in the last step, we measure the impact of long ACLs. For this reason, we use an administrative tool that we developed in order to assign permissions for multiple tenants in a collection of files and folders. We also evaluate the performance improvements of ACL sharing.

We first conduct microbenchmark experiments to measure the performance of basic metadata operations. In a next step, we experiment with application-oriented benchmarks for applications in distributed environments in order to explore the performance of the proposed architecture in real world applications.

[3]As discussed here: http://ceph.com/dev-notes/cephfs-mds-status-discussion/. Accessed: 2013-09-10.

We repeat each experiment to constrain the 95% confidence-interval half-length within 5% of a selected parameter. Before each repetition of every experiment we flush the buffer cache of all clients and servers. We also format the storage device used for experimentation of every fileserver. Furthermore, we set the size of the internal MDS cache to a large enough value in order to make sure that entries were not flushed from the caches by the time they were needed again.

6.3 Microbenchmarks

First, we measure the system performance with the mdtest v1.9.1 [34] from LLNL. This is a microbenchmark running in the MPI environment over a parallel filesystem. Each spawned MPI task iteratively creates, stats, and removes a specified number of files and folders. We repeat each experiment to constrain the 95% confidence-interval half-length within 5% of the average file-stat throughput.

6.3.1 Optimal number of processes per client

Theoretically, with the given hardware resources we could launch a large number of client processes to emulate the behavior of a medium size supercomputing environment, but the excessive loading of local client resources could result in much lower than expected performance. In this manner, we measure the performance of a single client while we change the number of client processes. The goal of this experiment is to find the optimal number of processes for forthcoming experiments. We have Dike installed on the fileservers of the local testbed. Dike is configured to support a single tenant. A total number of 31104 created files/folders are equally divided among the tasks of the experiment. We notice that 12 processes per client give the highest throughput for the majority of the examined operations. For example, increasing the number of processes from 1 to 12 leads to higher throughput for file create by about a factor of 4. The only exception is the folder stat operation, where a single client process gives 1237 ops/s and lies 0.9% higher that 12 client processes. Finally, as shown in Figure 6.1 the overall throughput of all the examined operations drops slightly, when we increase the number of processes to 24.

Thus, with our existing setup, experiments with 12 processes per client yield the

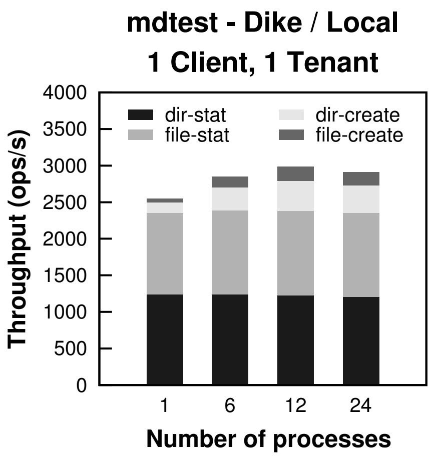


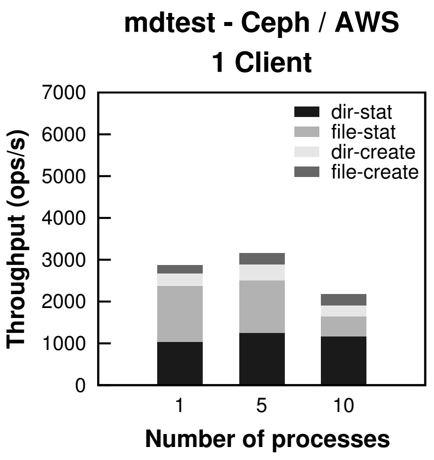
Figure 6.1: Finding the optimal number of processes per client for the mdtest microbenchmark on the local tesdbed.

optimal client side IOPS rate. Therefore, in the subsequent experiments with the mdtest microbenchmark on the local testbed all results are presented with 12 client processes.

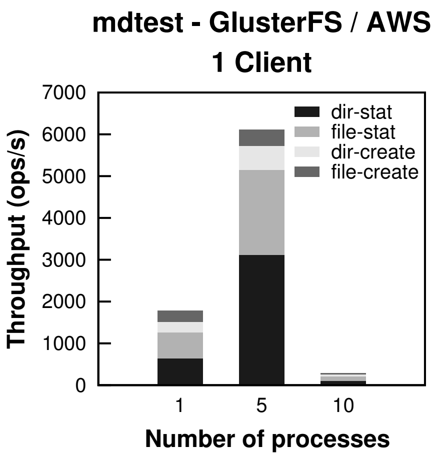
In addition, we conduct the same type of experiment on the cloud testbed (see Figure 6.2). Again, we measure the performance of a single client while we change the number of processes. We repeat the experiment for two cases: Firstly, we have Ceph installed on the fileservers, while secondly we have GlusterFS installed. We would like to find the optimal number of client processes for both baseline filesystems. A total number of 1000 created files/folders are equally divided among the tasks of the experiment.

In the case of Ceph (Figure 6.2(a)) we notice that 5 processes per client give the highest throughput for the majority of the examined operations. For example, increasing the number of processes from 1 to 5 leads to 38% higher throughput for the file create operation. However, when we further increase the number of processes from 5 to 10, the throughput of file create drops nearly 4.5%. The only exception is the file stat operation, where 1 client process gives 1332 ops/s and lies 6% higher than the throughput of 1257 ops/s which is achieved with 5 client processes.

In the case of GlusterFS (Figure 6.2(b)) we reach the same conclusion: 5 processes per client give the highest throughput for all the examined operations. However, this time the benefits of 5 client processes are more clear. When we increase the number of processes from 1 to 5, the throughput of the folder stat operation increases by about a factor of 5 and reaches 3115 ops/s. However, when we further increase the number of processes from 5 to 10, it drops to 104 ops/s.



(a) Ceph.



(b) Gluster.

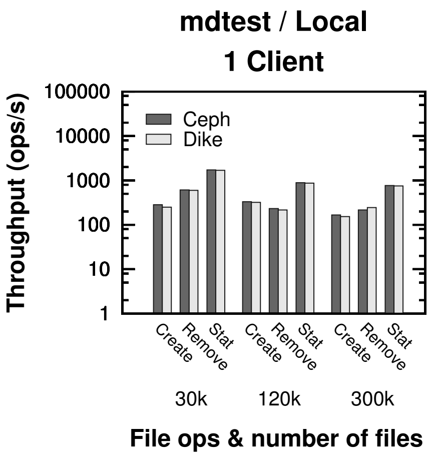
Figure 6.2: Finding the optimal number of processes per client for the mdtest microbenchmark on AWS.

In conclusion, with 5 processes per client we get the best client side IOPS rate for both Ceph and GlusterFS. Thus, in the subsequent experiments with the mdtest microbenchmark on the cloud testbed all results are presented with 5 client processes.

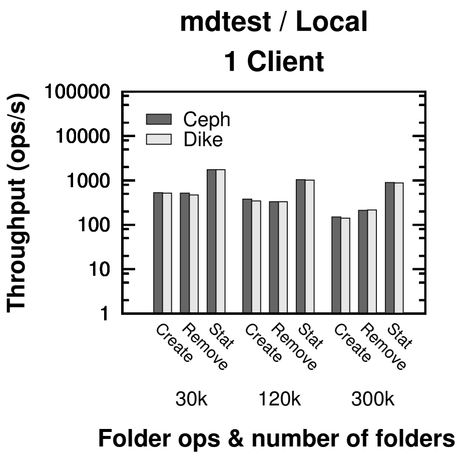
6.3.2 Scalability with number of files

In Figures 6.3(a) and 6.3(b) we measure the metadata performance of Ceph and Dike for different numbers of total files and folders on the local testbed. We create a file tree with depth 1 and 10 folder leafs. In each case, we equally divide the total number of files/folders to leafs. We use one client with 12 client processes. Dike is configured to support 36 tenants.

The measured performance is comparable between Ceph and Dike. The overhead of Dike on all the examined operations lies between 0.4-11%. The file and folder stat operations seem to be less affected by Dike in comparison to the respective creations. For example, the throughput of file stat with 30000 files reaches 1683 ops/s when Dike is used. It lies only 2.2% lower than the maximum throughput of 1721 ops/s which is reached when Ceph is used. On the other hand, the throughput of file create, when 30000 files are created, reaches 250 ops/s with Dike and lies 11% lower than Ceph with 282



(a) File operations performance with Ceph and Dike.



(b) Folder operations performance with Ceph and Dike.

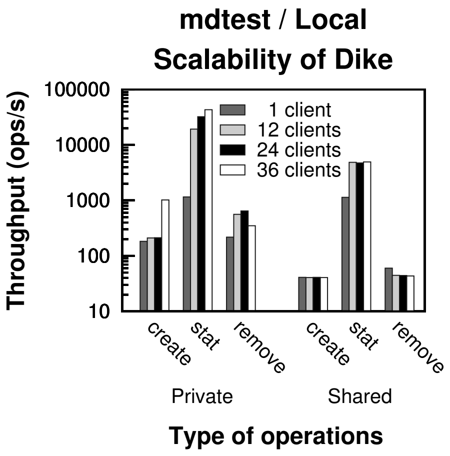
Figure 6.3: Ceph vs Dike with different number of total files and folders. Dike supports 36 tenants.

ops/s. We also notice that in both Ceph and Dike the throughput of file/folder stat and folder create operations drops as the number of total files and folders increases. However, the throughput of the file create operation increases slightly as the total number of files increases from 30000 to 120000 and finally drops when the total number of files reaches 300000.

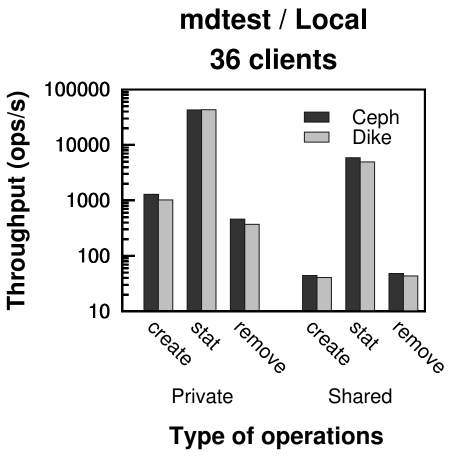
6.3.3 Scalability with number of clients

In Figure 6.4 we measure the scaling of metadata operations with the number of MPI processes that are launched on available clients. We use the local testbed for experimentation. We examine the cases that either every client creates files in a private folder of the filesystem or all clients use a shared folder. Each time, a total number of 31104 created files and folders are equally divided among the tasks of the experiment. Dike supports 36 tenants and has each client accessing the filesystem through a dedicated tenant.

Figure 6.4(a) shows the performance scaling of Dike with number of clients. In the case of the private folder tests, the throughput of the majority of the examined operations continues to increase as we increase the number of clients from 1 to 36. In particular,



(a) Scalability of Dike.



(b) Ceph vs Dike.

Figure 6.4: Performance comparison of Ceph and Dike across different number of clients. Dike supports 36 tenants.

the throughput of the create operation increases by a factor of 5.5, while the throughput of stat increases by a factor of 37. However, the throughput of remove drops beyond 24 clients. This behavior is reasonable given the different intensity of contention caused by shared (e.g. stat) or exclusive (e.g. remove) locks respectively involved in the operations. On the other hand, in the case of the shared folder tests, only the throughput of stat increases from 1134 ops/s to 4948 ops/s, as we increase the number of clients from 1 to 36. The throughput of create remains stable at around 40 ops/s, while the throughput of remove drops from 59 ops/s to 43 ops/s. Again, this behavior is reasonable because the stat operation involves shared read locks, while the create and remove operations need exclusive write locks. Overall, the throughput of all the examined operations on the shared folder tests is lower than the respective throughput on the private folder tests because of higher lock contention.

In Figure 6.4(b) we compare the throughput of mdtest running on 36 clients. The measured performance is comparable between Ceph and Dike. The overhead of Dike lies between 0-20%. Dike has no negative effect on file stat operation. Instead, file stat is improved by 1% when Dike is used. The operation that is mostly affected by Dike is file create over a private folder, where Dike with 1022 ops/s lies 20% lower than Ceph with

1287 ops/s. This is likely the result of the added code within the inode creation process, which updates both the inode and its extended attributes.

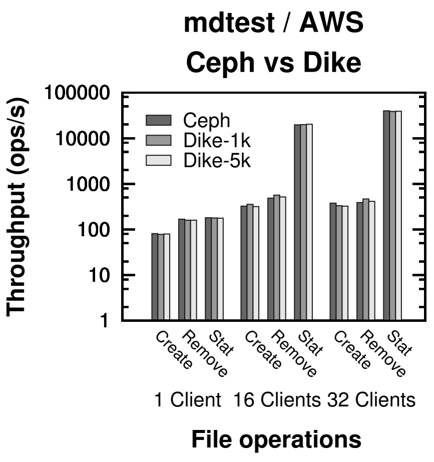
6.3.4 Comparison with other multitenancy solutions

In Figure 6.5 we measure the multitenancy overheads incurred by Dike and HekaFS over their baseline equivalents. The goal of this experiment is to understand how existing solutions that use identity mapping techniques to support multitenancy scale to a large number of tenants, and how they compare with our prototype of the proposed access control architecture. This time we use the Amazon Web Services for experimentation.

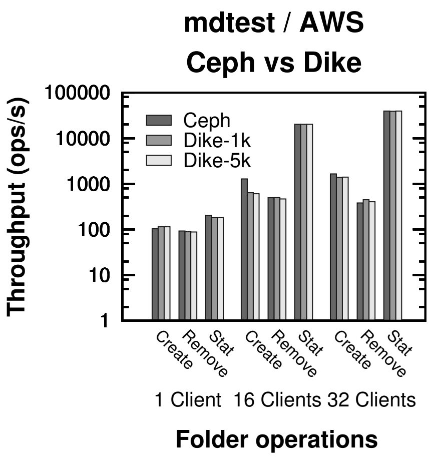
Before presenting the results, we mention some of the key features of HekaFS. HekaFS is a translation layer that adds multitenancy functionality to GlusterFS. In order to isolate the identity space of each tenant it uses identity mapping to map local tenant IDs to globally-unique IDs. It stores these mappings in a map file. During the initialization phase, the translator reads the ID mappings from the map file and loads them into an in-memory table structure. Each time that the translator needs to perform a mapping from a local tenant ID to a global ID or the opposite, it locks the whole in-memory mapping table, and then performs a linear search to find the requested entry on the table. When the translator encounters a new local tenant ID, it first adds a new mapping to the in-memory table, and in a next step it writes the whole table to the mapping file in order to make the change persistent.

In Figures 6.5(a) and 6.5(b) we compare Ceph and Dike with mdtest across different number of clients[4]. In the case of Dike, a client accesses the filesystem through a dedicated tenant. Each time, a total number of 48000 created files and folders are equally divided among the tasks of the experiment. We configure Dike to support either 1000 tenants (denoted as Dike-1k) or 5000 tenants (denoted as Dike-5k). The scalability of both file and folder metadata operations is comparable between Ceph and Dike. The throughput of all the examined operations continues to increase as we increase the number of clients from 1 to 32. With 32 clients we get the best IOPS rate. The only exception is the remove operation whose maximum throughput drops slightly beyond 16 clients. However, this is reasonable due to lock contention, since the remove operation involves exclusive locks.

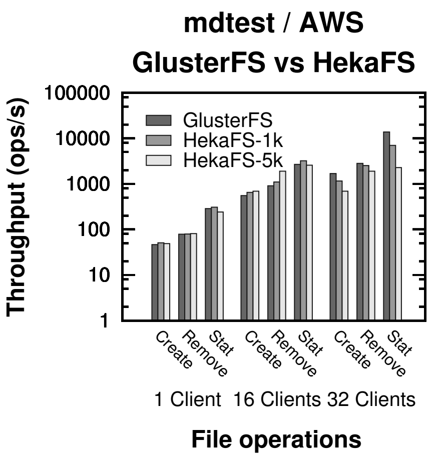
[4]We use t1.micro EC2 instances for clients.



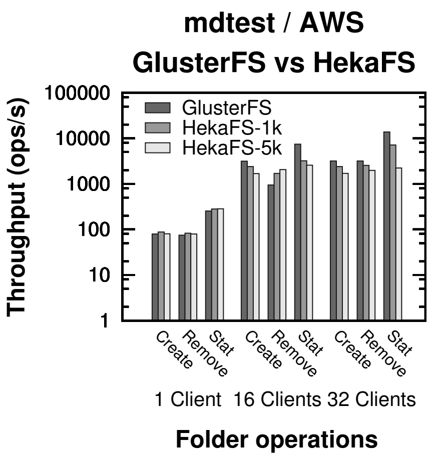
(a) File operations performance with Ceph and Dike.



(b) Folder operations performance with Ceph and Dike.



(c) File operations performance with GlsuterFS and HekaFS.



(d) Folder operations performance with GlsuterFS and HekaFS.

Figure 6.5: Performance comparison of Dike and HekaFS across different number of clients and supported tenants with mdtest.

Figure 6.5(a) compares the performance of different file metadata operations between Ceph and Dike. With 1 client, the throughput of the file create operation is 81 ops/s with Ceph, while it reaches 78 ops/sec with Dike and 1000 supported tenants. Increasing the number of supported tenants to 5000 seems not to affect throughput, which reaches 80 ops/s and remains slightly below 81 ops/s. Similarly, Dike incurs a limited overhead on the file remove and file stat operations. The overhead of Dike on file remove lies between 4-5%, while its overhead on file stat lies between 0-2%. With 16 clients, the throughput is nearly identical between Ceph and Dike. Only a limited overhead of 1% on the file stat operation is incurred by Dike when it supports 5000 tenants. The most interesting case, however, is when 32 clients run the mdtest microbenchmark in parallel. In this case we get the best client-side IOPS rate for both Ceph and Dike. The overhead incurred by Dike when it is configured to support 1000 tenants lies between 0-12%. When we increase the number of supported tenants to 5000, the performance is comparable with the previous case with only a 2% overhead on the file create operation.

Figure 6.5(b) compares the performance of different folder metadata operations between Ceph and Dike. When 1 client runs the mdtest microbenchmark, Dike does not affect the throughput of folder create operation. In addition, the throughput of folder remove is only reduced by 4%, when Dike with 5000 tenants is used. Finally, Dike incurs an overhead of 11% on the file stat operation when it either supports 1000 or 5000 tenants. With 16 clients, we observe a noticeable decrease of 50% in throughput of the folder create operation when we use Dike with either 1000 or 5000 tenants. On the other hand, the throughput of folder remove decreases slightly by 5% with Dike, while the throughput of folder stat is nearly identical between Ceph and Dike. As in the case of file metadata operations, the best client-side IOPS rate is achieved with 32 clients for both Ceph and Dike. The overhead incurred by Dike when it is configured to support 1000 tenants, lies between 0-16%. The folder create is the only operation that is affected by Dike. In particular, its throughput reaches 1655 ops/s when we use Ceph. However, when we use Dike with 1000 tenants, it reaches 1390 ops/s, which implies a reduction of 16%. Increasing the number of supported tenants on Dike to 5000 seems not to affect throughput which reaches 1401 ops/s. This indicates that Dike scales well to a large number of tenants.

In Figures 6.5(c) and 6.5(d) we compare HekaFS and GlusterFS with mdtest across different number of clients. In the case of HekaFS, a client accesses the filesystem through a dedicated tenant. Again, a total number of 48000 created files and folders are equally divided among the tasks of the experiment. We configure HekaFS to support either 1000 tenants (denoted as HekaFS-1k) or 5000 tenants (denoted as HekaFS-5k). When we use GlusterFS, the throughput of all the examined operations continues to increase as we increase the number of clients from 1 to 32. With 32 clients we get the best IOPS rate.

In HekaFS with 1000 tenants the throughput of all the examined operations, except the folder stat, continues to increase as we increase the number of clients from 1 to 32, but with a lower rate in comparison to GlusterFS. However, the scalability of HekaFS is dramatically affected when the number of supported tenants reaches 5000. For example, the throughput of the file stat operation reaches 2576 ops/s when 16 clients run the mdtest microbenchmark. Nevertheless, when we increase the number of clients to 32, the throughput of file stat drops to 2290 ops/s.

Figure 6.5(c) compares the performance of different file metadata operations between GlusterFS and HekaFS. With 1 client, the throughput of the file create and file remove operations is not affected by HekaFS. However, the throughput of file stat is reduced by 15% when we use HekaFS with 5000 tenants. Similarly, when 16 clients run the mdtest microbenchmark, the performance of all the examined operations is nearly identical between GlusterFS and HekaFS. Only the throughput of file stat is reduced by 3% when we use HekaFS with 5000 tenants. However, in the case of 32 clients, the overhead incurred by HekaFS when it is configured to support 1000 tenants is considerable and lies between 10-49%. The most affected operation is the file stat whose throughput is reduced by 49% when we use HekaFS. Additionally, the throughput of file create reaches 1690 ops/s with GlusterFS, but with HekaFS it only reaches 1158 ops/s. This implies a 31% reduction of its throughput. On the other hand, file remove is less affected and its throughput is reduced by nearly 10% with HekaFS. When we increase the number of supported tenants to 5000, the system performance degrades even further and the incurred overhead of HekaFS over GlusterFS lies between 38-83%. Again, the file stat is the operation that is mostly affected by HekaFS. Its throughput reaches 689 ops/s and lies 83% below its corresponding throughput when GlusterFS is used. The other two examined metadata operations are also affected by HekaFS. In particular, the throughput of file create is reduced by 59%, while the throughput of file remove is reduced by 38%.

Figure 6.5(d) compares the performance of different folder metadata operations between GlusterFS and HekaFS. When 1 client runs the mdtest microbenchmark, the performance of all the examined operations is nearly identical between GlusterFS and HekaFS. However, with 16 clients HekaFS begins to adversely affect their throughput. In particular, the throughput of folder create is reduced by 24% when HekaFS with 1000 tenants is used, and when HekaFS supports 5000 tenants, it is reduced by 47%. The throughput

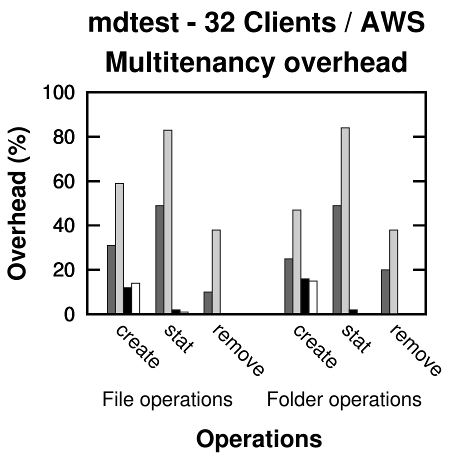


Figure 6.6: Multitenancy overhead comparison between HekaFS and Dike.

of folder stat, however, is more seriously affected by HekaFS. When 1000 tenants are supported, its throughput reaches 3224 ops/s and is 56% below the throughput of 7428 ops/s, which is achieved with GlusterFS. Adding 5000 tenants to HekaFS impacts the performance of folder stat further. In this case, its throughput only reaches 2574 ops/s and lies 65% below the baseline throughput performance. With 32 clients, the overhead incurred by HekaFS is more considerable. As in the previous case of 16 clients, the folder stat is the operation that is mostly affected by HekaFS. Its throughput reaches 7114 ops/s when HekaFS with 1000 tenants is used, and lies 49% below 13858 ops/s which is achieved with GlusterFS. Increasing the number of tenants to 5000 in HekaFS leads to a further reduction of the folder stat throughput. In this case, its throughput only reaches 2245 ops/s and is 84% below the baseline throughput performance.

Finally, in Figure 6.6 we summarize the overheads incurred by Dike and HekaFS over the filesystems that they are based in the case of 32 clients. As the figure shows, Dike with 1000 supported tenants incurs an overhead of up to 12% to the file metadata operations. This overhead is comparable with the maximum incurred overhead of 14% when the system supports 5000 tenants. In addition, Dike with 1000 tenants incurs a maximum overhead of 16% to the folder metadata operations, while with 5000 tenants the overhead is up to 15%. On the other hand, HekaFS with 1000 supported tenants incurs an overhead of up to 49% to the file and folder metadata operations. However, the maximum incurred overhead reaches 84% when the number of supported tenants is further increased to 5000.

In conclusion, Dike incurs a limited overhead and scales well to a large number of tenants. The operation that is mostly affected by Dike is file/folder create. This is in par with the experiments conducted in the local cluster (see subsection 6.3.3). On the other hand, the mapping layer of HekaFS can be a performance bottleneck for scalability when the number of tenants increases. The operation that is mostly affected by HekaFS is file and folder stat. This is reasonable because on each stat operation HekaFS needs to perform a reverse identity mapping in order to map a global ID to the corresponding tenant local ID. Thus, when the mapping table gets too large, the time needed to search the table or to write the table to disk increases, and as a consequence the overall system performance is being reduced.

6.4 Application-oriented benchmarks

We conduct application-level experiments in order to evaluate the performance of Dike in real life collaborative use cases.

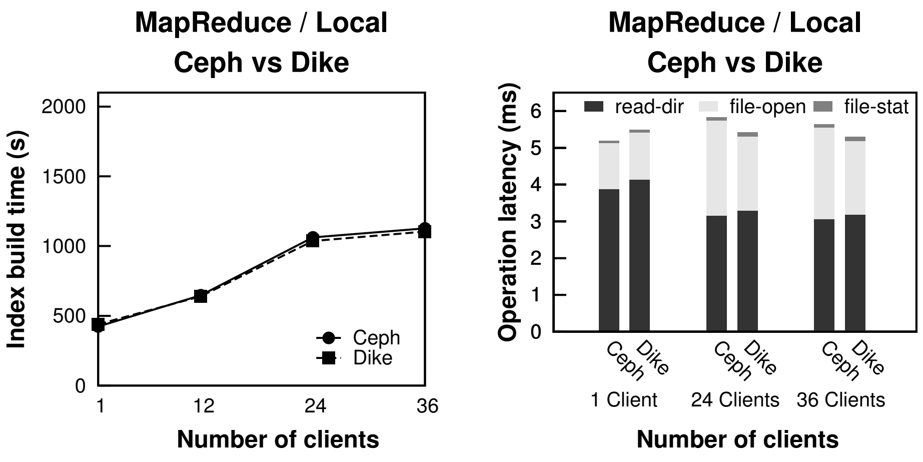
6.4.1 MapReduce application

The first application that we use is Stanford's Phoenix version 2 [50] shared-memory implementation of Google's MapReduce. Our MapReduce application is called reverse index: it receives a collection of HTML files and generates the text index with links to the files. Our dataset[5] contains 78,355 files in 14,025 folders and occupies 1.01 GB. We measure the total running time, as well as the latency breakdown of several metadata operations during the index building. We repeated each experiment to constrain the 95% confidence-interval half-length within 5% of the average total running time.

6.4.2 Comparison of Ceph and Dike with MapReduce

In Figure 6.7 we compare Ceph and Dike with MapReduce across different number of clients on the local testbed. Dike is configured with 36 tenants. Each client on Dike accesses the filesystem through a dedicated tenant. The total running time (Figure 6.7(a))

[5]Stanford's reverse index dataset: http://mapreduce.stanford.edu/datafiles/reverse_index.tar.gz. Accessed: 2013-08-19.



(a) Total time to build the index. (b) Latency of metadata operations.

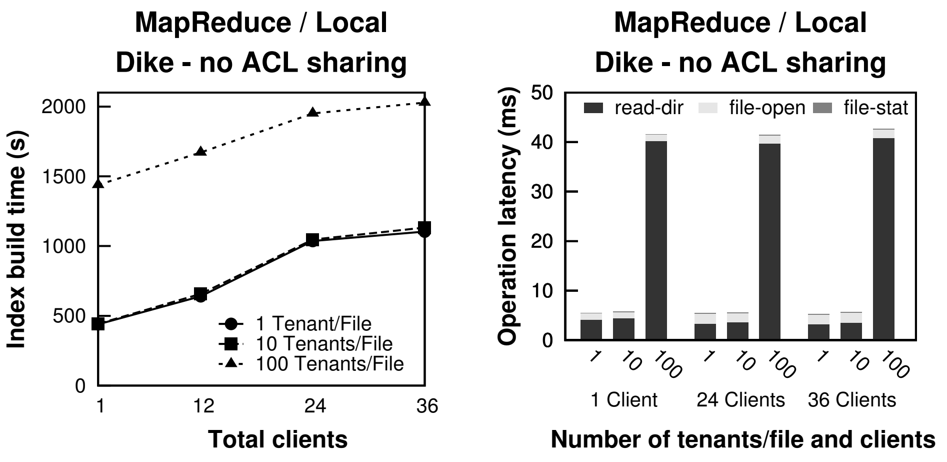
Figure 6.7: Comparison of Ceph and Dike with MapReduce across different number of clients. Dike supports 36 tenants.

increases as we increase the number of clients that run the reverse index in parallel. However, it is comparable between Ceph and Dike. The overhead imposed by Dike is negligible and it lies between 0-3.8%. The 3.8% overhead occurs when a single client runs the reverse index application. In this case, when we use the original Ceph, the total time spent to build the index reaches 423.61 seconds, which is the lowest measured time, while it reaches 439.75 seconds when we use Dike.

Figure 6.7(b) shows the latency breakdown of metadata operations. In the case of Dike, the most of them are completed in latency comparable to that of the original Ceph. One exception is readdir whose latency lies 7% higher when Dike is used in comparison to Ceph. This is due to the extra access control information that is stored in the extended attributes of each file/folder.

6.4.3 Impact of ACL size

In real-life collaborative environments where an enormous number of end users is involved, situations where files are shared by a large number of users are common. For this reason, we emulate a real-life file sharing scenario in order to understand how file sharing and in particular the size of ACLs impacts the overall system performance.



(a) Total time to build the index. (b) Latency of metadata operations.

Figure 6.8: Impact of long ACLs on the overall system performance. Dike supports 100 tenants.

In Figure 6.8 we measure the system performance impact when the size of the file and folder ACLs increases. In order to increase the size of ACLs, we permit multiple tenants to access the dataset. To accomplish this we developed a tool which grants a specific tenant access to files and folders. Thus, the size of the extended attributes of each file and folder is being increased, because an extra entry is being added for each tenant. We consider three different sharing scenarios: (a) only 1 tenant can access the dataset, (b) 10 tenants have read access to the shared dataset, and (c) 100 tenants have read access to the shared dataset. We use the local testbed for experimentation. Each time, Dike is configured with 100 tenants and a client on Dike accesses the filesystem through a dedicated tenant.

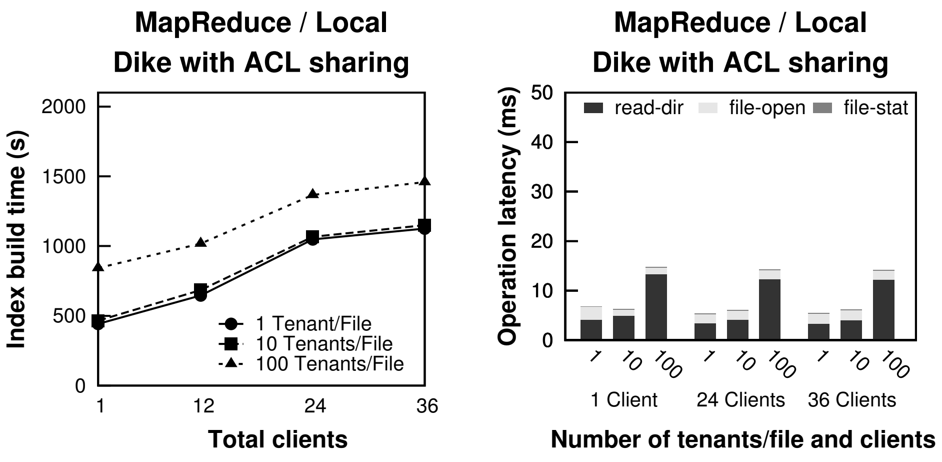
In all the examined cases, the total running time (Figure 6.8(a)) increases as we increase the number of clients that run the reverse index application. The running time of the application is comparable in both cases where the dataset is shared by 1 and 10 tenants. Only a slight 2.6% increase of the application's total running time is observed when 36 clients run the reverse index in parallel. However, long ACLs (100 tenants/file) impose a significant increase on the total time spent to build the index. When a single client runs the reverse index application and the dataset is shared by 100 tenants, the total running time reaches 1437 seconds. This implies an increase of about a factor of 3 when compared to the scenario where only one tenant has access to the dataset.

In Figure 6.8(b) we measure the latency breakdown of different metadata operations in order to better understand which operation is mostly affected by long ACLs and is responsible for the largest proportion of time spent to build the index. We observe that the latency of the majority of operations is comparable. However, when ACLs become too long (100 tenants/file), then the latency of the readdir operation increases by a factor of 10. In particular, when a single client runs the reverse index application, the latency of the readdir operation reaches 4 ms in the case where the index is shared by only one tenant. Instead, when the index is shared by 100 tenants, the measured latency of the readdir operation reaches 40 ms. This latency increase is high because in a readdir operation the MDS fetches the entire directory from the OSDs (if it isn't already in its cache), including inode contents. As we explained in subsection 2.4.2, Ceph (and Dike therefore) stores the contents of a folder (including the extended attributes of each file) in a single object. If this object exceeds an upper limit, it is also possible to split the folder contents into multiple objects. However, in the conducted experiments we have disabled fragmentation, because it is still considered an unstable feature[6] of the Ceph version we are using. Hence, when the size of extended attributes of each file gets large, the MDS has to fetch a lot of information from the OSDs which leads to higher latencies.

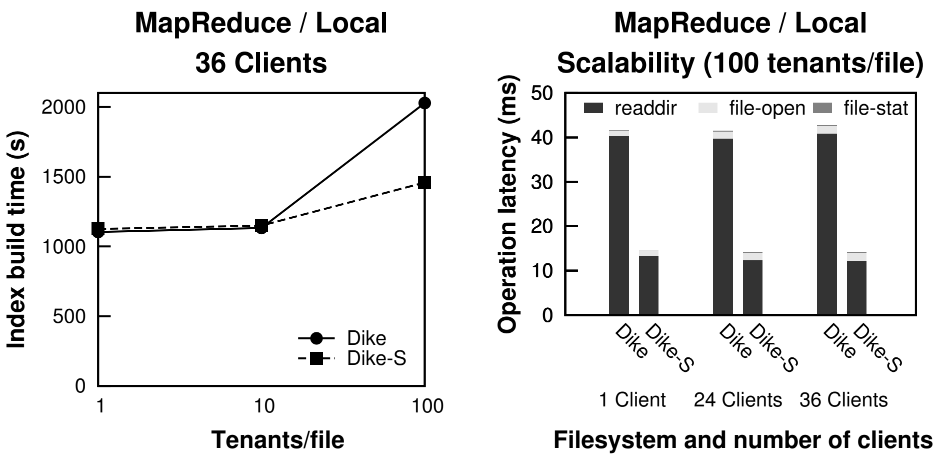
When ACLs tend to become too long, associating an ACL to each file leads to an enormous number of long ACLs that the system needs to store and manage, and can cost considerable performance overheads. As we discussed in section 4.5, users prefer to add new files to an existing folder with its permissions already set instead of setting the access rights on newly created files. Thus, it is common for files under the same folder to have identical access rights. Hence, we permit files with identical access rights to share their folder's global ACL, which we call tree ACL, in order to improve scalability and performance in the cases where ACLs tend to increase in number and size.

In Figure 6.9 we measure the system performance impact of Dike with ACL sharing (denoted as Dike-S) in the local testbed and compare it with the case where we have ACL sharing turned off. Again, the total running time (Figure 6.9(a)) increases as we increase the number of clients that run the reverse index. However, when ACLs are long (100 tenants/file) and a single client runs the reverse index application, we notice a 91%

[6]As discussed here: http://ceph.com/dev-notes/cephfs-mds-status-discussion/. Accessed: 2013-09-10.



(a) Total time to build the index. (b) Latency of metadata operations.

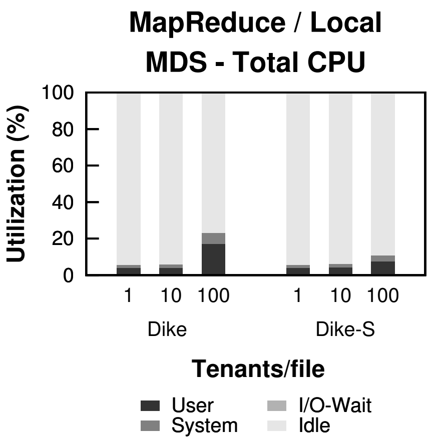


(c) Total time to build the index. (d) Latency of metadata operations.

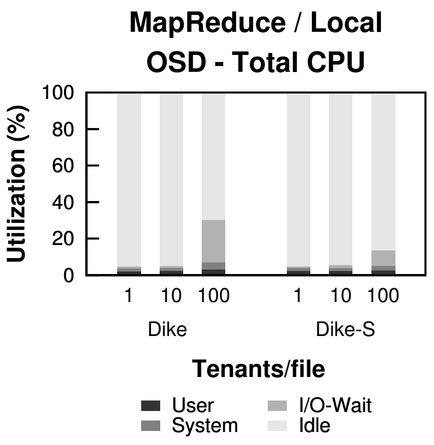
Figure 6.9: The benefits of ACL sharing when ACLs tend to become very long. Dike supports 100 tenants.

increase to the total running time of the application which implies an improvement of 39% when compared with the case where ACL sharing is disabled (see Figure 6.9(c)).

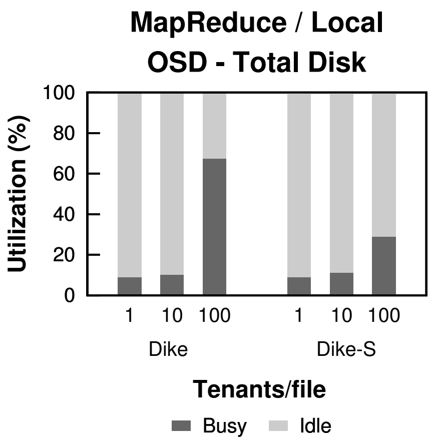
In Figure 6.9(b) we measure the latency breakdown of metadata operations during the experiment. Again, the latency of the majority of operations is comparable. Only when ACLs become long, the latency of readdir increases by a factor of 3, when a single client runs the application. In particular, when the index is shared by only one tenant, the latency of the readdir operation reaches 4 ms. Instead, when the index is shared by



(a) MDS CPU utilization.



(b) OSD CPU utilization.



(c) OSD Disk utilization.

Figure 6.10: CPU and disk utilization of fileserver nodes.

100 tenants, the measured latency of the readdir operation reaches 13 ms. However, it is 3 times lower than the measured readdir latency of 40 ms when ACL sharing is turned off and the dataset is shared by 100 tenants (see Figure 6.9(d)).

A possible overhead of multiple long ACLs is the CPU cost that is needed so that the MDS to be able to manage them in order to enforce access control. In addition, multiple long ACLs may impose high I/O loads on the OSDs.

In Figure 6.10 we evaluate the impact of multiple long ACLs to the total CPU utilization of the MDS and OSDs, as well as its impact to the disk utilization of the OSDs. In the case where ACL sharing is disabled, we observe that the total CPU utilization of the MDS remains below 6%, when the ACLs are small. However, it reaches 23%, when the ACLs become long. Instead, as Figure 6.10(a) shows, when we enable ACL sharing, the average CPU utilization only reaches 11%.

In addition, Figure 6.10(b) shows that in both cases where ACL sharing is turned off or on, the CPU of the OSDs remains idle, whether doing nothing or waiting for the I/O operations to finish. However, when ACLs are long and the ACL sharing mechanism is disabled, the disk utilization reaches 67% in comparison to the 9% average utilization when the ACLs are small. Instead, as Figure 6.10(c) shows, ACL sharing reduces disk utilization in the case of long ACLs to 28%.

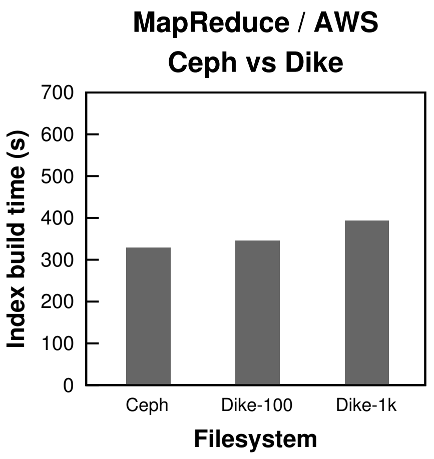
6.4.4 Comparison with other multitenancy solutions

In this subsection we study the comparative multitenancy overhead incurred by Dike over Ceph with the corresponding overhead incurred by HekaFS over GlusterFS using an application-oriented experiment (Figure 6.11). For this purpose we use the reverse index application on AWS with one client[7].

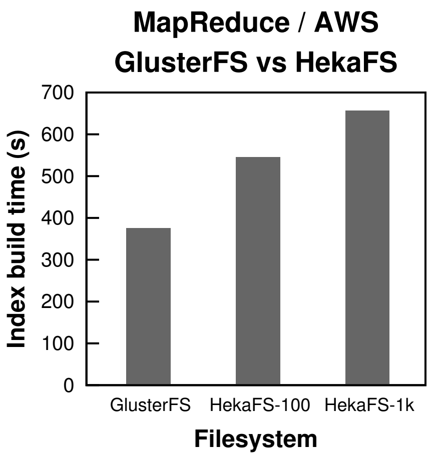
In Figure 6.11(a) we compare Ceph and Dike with the reverse index application. We configure Dike to support either 100 tenants (denoted as Dike-100) or 1000 tenants (denoted as Dike-1k). We notice that the total running time of the reverse index application is 328 seconds when we use Ceph. On the other hand, it reaches 346 seconds when we use Dike with 100 tenants. Thus, Dike with 100 tenants adds an extra latency of 5% to the total application's running time. Then, we increase the number of supported tenants on Dike to 1000 and repeat the same experiment. This time, the application's total running time reaches 394 seconds, which implies a 20% of extra latency.

In Figure 6.11(b) we repeat the same experiments as above, but this time we compare GlusterFS and HekaFS. We configure HekaFS to support either 100 tenants (denoted as HekaFS-100) or 1000 tenants (denoted as HekaFS-1k). The total running time of the reverse index is 375 seconds when we use GlusterFS. However, it reaches 545 seconds when we use HekaFS with 100 tenants, which implies an increase of 31%. Increasing the number of supported tenants in HekaFS leads to a higher added latency. In particular, when we

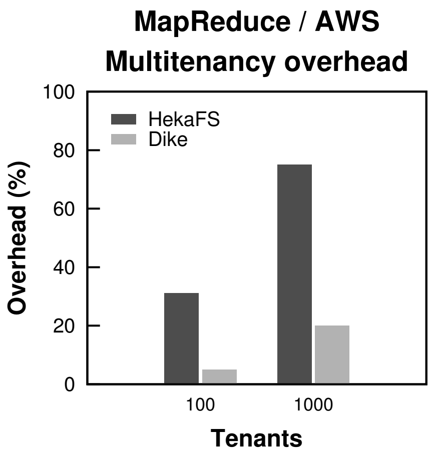
[7]We use a c1.medium instance for the client.



(a) Comparison of Ceph and Dike across different number of supported tenants.



(b) Comparison of GlusterFS and HekaFS across different number of supported tenants.



(c) Multitenancy overhead comparison between HekaFS and Dike.

Figure 6.11: Performance comparison of Dike and HekaFS across different number of supported tenants with MapReduce. We use a single client (c1.medium EC2 instance).

use HekaFS with 1000 tenants, the total running time of the reverse index reaches 656 seconds. This implies a 75% of extra latency.

In Figure 6.11(c) we can see the added overheads of HekaFS and Dike. The overhead incurred by Dike lies between 5-20% and is lower than the corresponding overhead of

HekaFS, which lies between 31-75%. These results verify our previous conclusion that multitenancy solutions which perform identity mappings can adversely affect the overall system performance when the number of tenants increases.

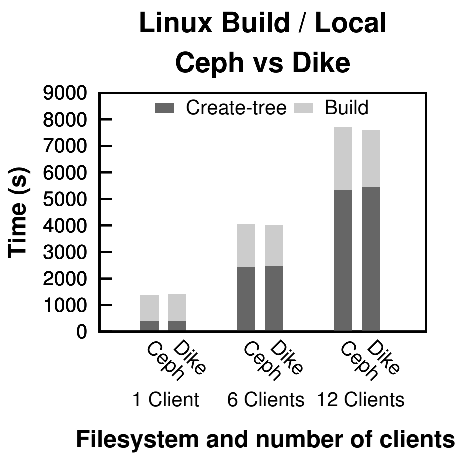
6.4.5 Linux compilation

In a different application-oriented experiment we store the source of the Linux kernel (version 3.5.5) in a shared folder of the filesystem. Then we make the code accessible to private folders of the tenants through soft links. We measure the average times to create the soft links and to build the system image. We repeated the experiment to constrain the 95% confidence-interval half-length within 5% of the average time to build the system image.

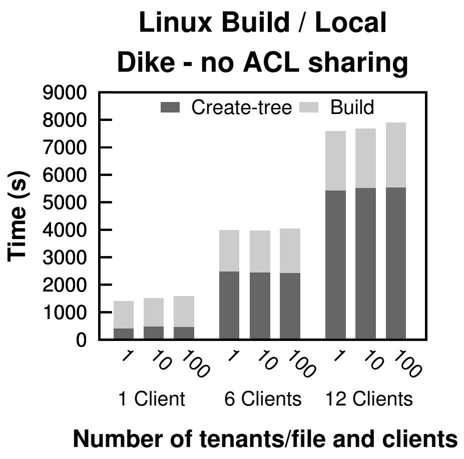
In Figure 6.12 we compare Ceph and Dike with Linux compilation across different number of clients on the local testbed. We have ACL sharing disabled on Dike. We measure the average time to create soft links on the shared Linux tree and the average time to build the system image by up to 12 clients assuming dedicated tenant per client in the Dike case. The extra latency added by Dike to soft link creation time is 4.5% with one client and 2% with 12 clients. In addition, the image building times are comparable between Ceph and Dike. The overhead imposed by Dike is negligible and it lies between 0-0.7%. The 0.7% overhead occurs when a single client runs the experiment. In this case, the image building time is 983 seconds in the case of Dike, where in the case of Ceph it lies 0.7% lower at 976 seconds.

Figure 6.12(b) evaluates the impact of long ACLs on the Linux kernel compilation experiment. For this experiment we have turned off ACL sharing. The extra latency added to soft link creation time when ACLs are long (100 tenants/file) is 14.5% with 1 client and 1.7% with 12 clients. In addition, the image building time for a single client lies 13% higher when ACLs are long, and for 12 clients lies 9% higher.

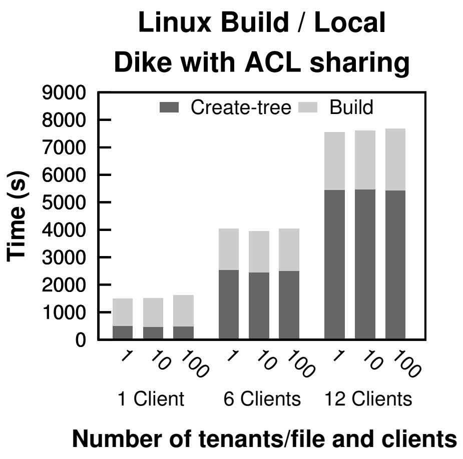
We also repeat the same experiment with ACL sharing turned on. Figure 6.12(c) shows the results. For long ACLs (100 tenants/file), the link creation time is nearly similar to the respective time when only a single tenant has access to the Linux kernel source. Regarding the image building time, with a single client the results are comparative with the case in which ACL sharing is turned off. However, with 12 clients, ACL sharing



(a) Ceph vs Dike.



(b) Dike with ACL sharing turned off.



(c) Dike with ACL sharing turned on.

Figure 6.12: Linux compilation. Dike supports 100 tenants.

improves the total time by 5% in comparison to the case where Dike does not use ACL sharing.

6.5 Summary

We experimentally evaluate a prototype implementation of the proposed architecture using microbenchmarks and application-level benchmarks. For experimentation we use two environments: (a) a local cluster and (b) a cloud platform. In summary, we demonstrate that our prototype adds a limited performance overhead, while it enables secure multitenancy. Additionally, our prototype scales well to a large number of tenants without affecting the overall system performance. Furthermore, we notice that long ACLs, which are common in real life collaborative environments can adversely affect system performance. However, the technique of ACL sharing that we introduced mitigates this problem.

In conclusion, our prototype adds a limited performance overhead and scales well to a large number of clients and tenants, in contrast to existing solutions that require an extra layer to map a local tenant ID to a globally-unique ID. This identity mapping layer can be a performance bottleneck for scalability and the proposed architecture eliminates it.

Chapter 7

Related research

In this chapter we review comparative studies that attempt to enable secure multitenancy in filesystem storage clouds. We also outline the most important studies that aim to provide trusted cloud storage. Finally, we survey previous works for access control in distributed filesystems.

7.1 Multitenancy in filesystem storage clouds

In this section we present recent studies that attempt to provide secure multitenancy in filesystem storage clouds.

7.1.1 Hypervisor-level multitenancy

In hypervisor-level multitenancy, the hypervisor itself is responsible to track information flow between virtual machines and enforce access control. A system that follows this approach is presented by Mundata et al. [41]. Their system, which is called SilverLine, implements two types of isolation at the hypervisor level: (1) data isolation and (2) network isolation. To enforce data isolation, SilverLine uses labels to control information flow between files and processes within a single machine or across the network. In fact, tenants are allowed to label data with security labels; trusted enforcers at the hypervisor level then use these labels to ensure that data from one tenant is not propagated to untrusted server instances belonging to other tenants, or to locations outside the cloud.

A similar study is presented by Popa et al. [44]. CloudPolice implements access control at the end-hosts within hypervisors. It provides various access control policies, such as complete tenant isolation, selective inter-tenant communication, fair-sharing among tenants, rate-limiting tenants, and allowance of locally initiated connections. Hypervisors know the policies of their hosted virtual machines and communicate with other hypervisors at runtime in order to learn external policies and control information flows. When a new information flow is being initiated by a virtual machine, the source hypervisor communicates with the destination hypervisor and the latter checks the policy for the destination virtual machine. If the policy forbids the traffic, then the destination hypervisor blocks it and appropriately informs the source hypervisor. Otherwise, if the traffic is allowed, the destination hypervisor initiates the state for this flow.

Kurmus et al. [25] implement a virtualization-based multitenancy architecture using KVM by running multiple virtual interface nodes as guests on the same physical node. Virtual machine guests that belong to the same tenant maintain a distributed filesystem with the tenant's data. Each virtual machine runs one instance of the file-service and exports the filesystem through a network filesystem protocol such as NFS. Tenant isolation is generally applied at the hypervisor who is in charge to block inter-tenant traffic according to tenant-specific policies.

All the above studies successfully isolate tenants at the hypervisor level. However, this approach is not suitable for a collaborative filesystem storage cloud because it hinders group collaboration and leads to performance scalability problems. The main reason behind this, is the fact that the filesystem is not deployed as a shared service but a separate file-service instance runs for each tenant in a tenant-dedicated virtual machine. Furthermore, it is observed [25] that the addition of multiple isolation layers and policy enforcers at the hypervisors incurs a significant performance overhead.

7.1.2 Operating system-level multitenancy

Isolating tenants at the operating system-level can lead to lower execution overheads. Tenant isolation is performed by mechanisms which are called containers [23, 48]. Containers create isolated namespaces for resources such as filesystems, network interfaces, and processes inside the same operating system. Each tenant gets its own namespace which is isolated from namespaces of different tenants.

Kurmus et al. [25] present an implementation of this approach. Their implementation uses SELinux multi-category-security (MCS) policies for isolating the fileserver processes that serve a particular tenant. Fileserver processes belong to different categories according to tenant-specific policies. This ensures that a tenant can not access the resources of a different tenant because they belong to a different category. In fact, multiple domains (or containers) exist on the same operating system and each domain consists of a chroot folder in the root filesystem of the physical host.

This approach leads to lower execution overhead in comparison with the hypervisor-level access control approach. However, both approaches share the same disadvantages regarding the inability for file sharing and collaboration. This is due to the fact that each tenant runs its own file service which is completely isolated from the file services of different tenants.

7.1.3 Trusted multitenant storage

The two biggest concerns about storage systems used in virtualization and cloud environments, beyond high performance and scalability, are reliability and security. Secure cloud storage involves four desirable properties, including data and metadata confidentiality, integrity, write-serializability, and read freshness [45]. Organizations will not entrust their data to an external storage system without a guarantee that they'll be able to access the latest version of their data whenever they want and no one else will be able to access or modify it without their permission.

In recent years, there has been considerable work on trusted cloud storage. Popa et al. introduce CloudProof, a system that allows customers of cloud storage to securely detect and prove violations of integrity, write-serializability, and freshness [45]. Assuming that the cloud infrastructure is entirely untrusted, access control over read and write requests is enforced through data encryption with secret keys and update verification with public-key signatures.

The work of Santos et al. [52] is motivated by the observation that current trusted computer technology can not be used on the cloud as it exposes internal details of the cloud infrastructure, hinders performance and scalability, and has several manageability limitations. Their system, which is called Excalibur uses a trusted computing abstraction (policy-sealead data) to encrypt and decrypt data according to a specified node policy. Excalibur combines current trusted computing technology, such as TPMs, with a set of associated protocols and attribute-based encryption to offer developers two new primitives, seal and unseal, for constructing more trusted cloud services.

In contrast to the above works, in the current study we target secure storage access within the datacenter and aim to provide native multitenancy support at the file level by directly storing access control metadata in trusted object-based fileservers.

7.2 Access control in multitenant filesystems

Next, we review comparative studies for secure access control in filesystems that need to support multiple tenants, such as cloud, grid, and virtualization-aware filesystems.

7.2.1 Access control in filesystems for cloud and grid environments

HekaFS [11] enables a tenant to assign identities to local principals through hierarchical delegation. A user's identity consists of the user ID plus the ID of the tenant, to which the user belongs. Tenants have complete freedom to manage their own identity space on their own machines. However, a tenant user identity needs to be mapped to a global server identity. This mapping is done by the cloud translator which sits at the top of each per-tenant translator stack on the server. In fact, each server keeps a mapping table which maps a tenant ID plus a user ID to a unique server ID. The server adds a new mapping to the mapping table every time it encounters a unique combination of a tenant ID plus a user ID. Fortunately, this mapping is not coordinated across servers. Each server uses its own separate mapping table. However, file sharing between users that belong to different tenants becomes a cumbersome task because each server maintains its own separate mapping [60]. What is more, users may wish to express access control that refers to identities that a given system has not yet encountered [61]. Finally, as discussed in subsection 4.1.3, such identity translations may lead to scalability limitations and introduce security risks.

Support of storage access from different institutions requires consistent ownership and permission data across multiple client mounts [65]. Lustre, a parallel distributed filesystem designed to provide storage to high performance computing systems uses a similar approach based on credential mapping to solve the identity collision problem. Because each Lustre client contains its own UID space, it is necessary that the Lustre metadata server be given the ability to map from client UIDs and GIDs to an authoritative list of server UIDs and GIDs. Furthermore, Lustre can organize client sets as clusters in order to make mapping of nodes that share the same UID/ GID namespace easier. In this manner, the UID maps that are maintained by the MDS are indexed by ranges. When a client connects to a server, part of the process categorizes her into a cluster, and hence gives her a pointer into the maps for forward and reverse UID/ GID mapping. To facilitate fast lookups, the mapping module is implemented as a forest of binary trees. The UID map itself can be created either manually or with the help of a map creation tool. The approach of Lustre for solving the identity collision problem shares the same drawbacks with HekaFS. In the present work, we aim to natively support multitenancy by directly storing access-control metadata at the fileserver without the need for identity translations from one tenant to another.

7.2.2 Access control in filesystems for virtualization environments

Pfaff et al. [43] propose and design a virtualization-aware filesystem. Their system, which is called Ventana, resembles a conventional distributed filesystem in that it provides centralized storage for a collection of file trees, allowing transparency and collaborative sharing among users. Ventana's distinction from other distributed filesystems is its versioning, isolation, and mobility features to support virtualization. Ventana [43] offers a secure access control across multiple client guests through a combination of multiple ACLs and branching. More specifically, client guests can use private branches to isolate their files, or shared branches to share their files with other clients. Furthermore, they can use branch ACLs to control access to all of the files in a particular branch. In addition, some other types of ACLs are provided, such as file ACLs which control access to particular files, or version ACLs which control access to a particular version of a file. However, deep chains of branches along with multiple ACLs can adversely affect system performance.

A slightly different proposal is presented by Jujjury et al. [21]. VirtFS introduces a paravirtualized filesystem driver based on the VirtI/O framework. Their paravirtualized filesystem can be used to connect a host-based fileserver to multiple guests. Furthermore, it can also be used to provide guest-to-guest filesystem access. The mixing of different namespaces in VirtFS triggers some serious security issues that need to be resolved. To resolve these issues it offers two types of security models: the mapped security model and the passthrough security model. The mapped security completely isolates the guest user domain from that of the host's. In particular, the VirtFS server intercepts and maps the file create operations and all the get/set attribute requests. Files on the fileserver are created with VirtFS server's credentials, while the guests' user credentials are stored in extended attributes. When a guest performs a file or folder stat operation, the server extracts the guest's user credentials from extended attributes and sends them to the client. In contrast, the passthrough security model shares the host's and guest's user domains. In this model, the VirtFS server passes down all requests to the underlying filesystem. Files on the fileserver are created with guests' user credentials. Both security models have some limitations. Specifically, the mapped security model successfully isolates the guests' principals from the host's, however, it fails to isolate the principals from different guests. On the other hand, the passthrough security model passes all the requests to the fileserver and does not guarantee any isolation.

HumFS [12] is a similar approach to VirtFS in that it provides access to filesystems on the host for UML guests. However, it is a virtual filesystem, in the sense that it is not stored within the UML block device. In many cases, the data is simply stored in kernel structures. HumFS is conceptually similar to a network filesystem such as NFS. HumFS separates the guest's identity space from that of the host's by associating a metadata file to each file and folder. The metadata file keeps identity and permissions related information and lies in a parallel folder hierarchy with the exported hierarchy. The main issue with this approach is that it induces additional disk seeks during file stat and create operations. These additional seeks can severely hurt system performance. In addition, although HumFS successfully separates the guest's namespace from the host's namespace, no measure is taken in order to isolate the namespaces of different guests.

7.2.3 Cloud collaboration and data sharing

Storage consolidation offers new opportunities for data sharing and collaboration in the cloud. Geambasu et al. present CloudViews [17], a system that attempts to enable data sharing among different services in a single cloud. CloudViews attempts to enable flexible sharing between different services at any granularity and to design an efficient and scalable access control mechanism that protects private data. To achieve its first goal, the system allows different services that run in the same cloud to create and share shared and restrictive views of their data with other co-located services. To come up with its second goal, CloudViews uses cryptographic signatures to sign the different offered views. It combines the signed views with resource allocation and update notification mechanisms to enable high performance data sharing.

The S4 framework presented by Walfield et al. extends Amazon's S3 cloud storage to provide data sharing across different web services [64]. It supports access delegation over the objects of different users via hierarchical, filtered views of the applicable policy. The S4 framework is similar with CloudViews in the sense that both of them are distributed filesystems that use the view-based access control technique to enable data sharing among services. However, it extends CloudViews by enabling interaction-free modification of existing views and by considering cross-cloud sharing scenarios. The S4 framework allows users to create new principals and to associate views with each of them. A view is a subset of the namespace and consists of access rights and a set of filters (e.g. regular expressions, UNIX permissions). Access control is based on hierarchical evaluation of each view's access rights.

Both solutions presented above target scenarios where a user authorizes different Web services to access their data which is centralized with a storage provider. Instead, we target secure collocation of multiple tenants and their data on a shared filesystem and aim to isolate each tenant from other tenants, while permitting their users to collaborate by sharing files.

7.3 Access control in distributed filesystems

Access control has been comprehensively examined across known distributed filesystems. Next, we highlight some of the previous work on this area.

7.3.1 Centralized access control

Steiner et al. present Kerberos [59], a centralized authentication system based on symmetric key cryptography which allows for strong authentication in distributed environments. An administrative domain in Kerberos is known as a *realm*. A realm contains nodes that use *tickets* to communicate over a non-secure network and to prove their identity to one another in a secure manner. A main service of Kerberos is the Key Distribution Center (KDC). The KDC maintains a database of local principals and their secret keys. When a user logins to the system the KDC issues a time stamped Ticket Granting Ticket (TGT), encrypts it using the user's password, and returns the encrypted result to the user's workstation. When the user wishes to communicate with another node, he sends the TGT to the Ticket Granting Service (TGS). The TGS verifies the validity of the received TGT. If the user is permitted to access the requested service, the TGS grants him a ticket encrypted with the TGS's secret key and a session key encrypted with the user's secret key. The ticket enables access to specific services on specific nodes in the realm. Thus, the user can present the ticket to the desired service along with its service request. The communication with the service is secured with the obtained session key. Cross-realm authentication is not straightforward, because Kerberos relies on symmetric key cryptography. Realm administrators have to set up trust relationships and exchange keys for principals to access services in a different realm. Efforts have been made to extend the Kerberos protocol with public-key cryptography support [56] and with cross-realm authentication [71].

Centralized access control solutions that have the users registered on a central location are inapplicable to a cloud environment. A centralized solution can not adapt and scale to a large number of users. What is more, tenant administrators prefer to manage their users locally.

7.3.2 Decentralized access control

The work of Reiher et al. [51] is one of the first studies that recognize the need for cross-domain authorization and secure file sharing. Their system, which is called Truffles, is build on the replication services provided by the Ficus filesystem and adds a mechanism for secure file sharing between different administrative domains. Sharing in Truffles happens at the granularity of volumes. User authentication in Truffles relies on a hierarchy of certification authorities and users are identified by public keys bound to X.500 distinguished names in X.509 certificates. Authorization in Truffles is based on standard UNIX and Ficus access control mechanisms. The authors recognize the problem of identity collisions between principals of different administrative domains. Their solution is to map a principal's ID to a globally unique identifier and to store this global identifier as one of the file's attributes. Furthermore, Truffles does not permit a local root user to be mapped to the root user on a remote domain. However, as discussed in subsection 4.1.3, identity mapping techniques can be the source of scalability problems and can complicate file sharing and manageability.

Belani et al. introduce CRISIS [8] as a wide-area authentication and access control system which forms the security subsystem of WebOS. Authentication in CRISIS is based on X.509 certificates which are signed by a trusted CA. CAs are organized hierarchically, thus principals in different administrative domains need to have a common root CA to securely share files. Authorization in CRISIS uses a hybrid model of ACLs and capabilities and relies on certificates to specify group memberships. However, the reliance to certificates for cross-domain access control has been criticized for high complexity [22].

Miltchev et al. [39] follow a different approach and present the Distributed Credential FileSystem (DisCFS) which uses trust management credentials to identify files, principals, and access rights. Principals in DisCFS are identified by public keys. These public keys are directly binded to any set of authorizations. When a principal wishes to access a remote file, a trust-management credential is being generated which contains the principal's public key, as well as the authorizer's public key which is trusted by the remote domain. Access control policies are specified by the administrators and either accept or reject actions. Actions are also specified by the administrators as a set of name-value pairs. Polices can be distributed to principals as credentials. It is clear that this credential-based design incurs a high performance overhead when the chains of credentials get very long, or when the number of active users is high. This is because credentials can become long as they contain both signed identity attributes and the policy statements. Credential caching and hardware acceleration for cryptographic operations in conjunction with data replication across servers mitigate this problem. However, this solution is unrealistic for a cloud environment, where an enormous number of users is involved.

Secure file sharing between principals in different administrative domains is enabled by GSFS [22], a further development of SFS [33]. Authentication in GSFS is based on public keys. Each principal and server have a public/private key pair. The administration server of an administrative domain prefetches and caches users and group definitions of remote authentication servers belonging to different domains. Thus, during file access, the authentication server can establish identities for users based on local information. Users name remote authentication servers, users, and groups using self-certifying hostnames. Authorization in GSFS relies on access control lists (ACLs) that contain local and remote users. Remote users can only be listed on the ACL with their public keys. However remote groups can not be listed directly on the ACL, but they can be listed indirectly as members of local groups. GSFS has been recently criticized for limited autonomous delegation support [40]. Moreover, a GSFS authentication server needs to trust the remote authentication servers of any remote domains. This is unrealistic for an environment such as the cloud where tenants don't trust each other.

Margaritis et al. propose Nephele, an architecture for hierarchical access control in federated file services across different administrative domains [32]. Their goal is to improve application portability and identity management, and to reduce transfer costs in collaborative environments that require data sharing among principals of different domains. In such a large federated environment there are several large groups consisting of multiple subgroup layers across the different domains. The authors introduce the *hypergroup* as an heterogeneous two-layer construct. The upper layer contains administrative domains of a federation and the lower layer contains user groups from each participating domain. Domains and principals are both identified by public keys and they are binded to hypergroups with credentials. Access control is applied without central management of the principals or their access rights. Each domain manages its users, groups, and their access rights locally. Access rights over local storage resources are specified using access control lists. As a result, the network traffic that is needed to propagate group memberships specified in terms of users is avoided. However, Nephele does not deal with the namespace collision problem that arises in a multitenant environment.

The above discussion makes it clear that existing access control solutions can not be used in a cloud environment without a reconsideration of their security model and mechanisms. In contrast to the above works, in the present work we study the problem of storage multitenancy over virtualization environments, which introduces new challenges as a result of the system consolidation involved in the same datacenter.

7.4 Summary

Secure multitenancy in cloud storage supports multiple customers at low cost. The hypervisor-level multitenancy architecture runs separate virtual machines for each customer over a distributed filesystem. Instead, the OS-level multitenancy architecture relies on the fileserver kernel to isolate the resources of different customers leading to lower execution overhead. Considerable work has also been done on the field of trusted cloud storage in order for the cloud providers to be able to provide security guarantees to their customers about their data.

Several works in the field of cloud and virtualization-aware filesystems have identified the namespace collision problem, however they focus on the separation of the host's namespace from that of the guests', without isolating the principals of different guests. Other works depend on identity mapping techniques to solve the identity collision problem that stems from the file-based access. However, the addition of layers that perform identity mappings introduce manageability inefficiencies, performance degradation, and complicate file sharing.

Traditional file-based access presumes that principals are registered into a central authentication service. Due to identity management challenges from the large number of the involved users, this is unrealistic for the tenants of a cloud provider. Other solutions rely on trust management certificates for direct authorization, or presume that each administrative domain has its principals registered to a local authentication server. Then, the local authentication server trusts remote authentication servers in order to support cross-domain file sharing. In a cloud environment, however, it is unrealistic for a tenant to trust other tenants.

Chapter 8

Conclusions and future work

In this chapter we conclude this work by summarizing our contributions and discussing opportunities for future research.

8.1 Conclusions

Cloud collaboration is a newly emerging way of file sharing and coworking on shared projects, whereby collaboration documents, shared source code, or scientific data are uploaded to a central shared storage, and can be accessed by multiple parties. Consolidating storage at the filesystem level enables such sharing scenarios. Furthermore, it offers significant manageability benefits to system administrators. The file-level interface exposes the file structure of a filesystem, while it enables shared read/write access. Furthermore, it can provide an ephemeral and highly composable storage. However, it does not properly isolate the namespaces of different customers who access the shared storage. Thus, it is important to reconsider the access control techniques used in order to effectively isolate the principals of different tenants.

We have pointed out that a solution which depends on an identity mapping mechanism should be avoided because it complicates file sharing and manageability, and reduces performance. Furthermore, traditional solutions that have the principals registered on a centralized directory face scalability limitations, because they are not designed for an environment with an enormous number of end users. Moreover, their trust assumptions do not apply to a cloud environment.

Accordingly, we have proposed an architecture that eliminates the need of a global directory service which maintains all filesystem principals, by allowing tenants to operate their own tenant authentication servers. Tenant authentication servers are securely registered to a globally trusted filesystem authentication server and certify local principals. In addition, our architecture natively supports multitenancy in virtualization environments that use file-based storage consolidation. We have achieved this by carefully storing access control metadata directly at the fileservers without the need for identity translations. In fact, the filesystem maintains per-tenant dedicated ACLs, where it stores tenants' local principals and access control policies. Thus, it successfully isolates each tenant's namespace. Furthermore, we permit files with identical access rights to share their parent's global ACL in order to keep the number and size of ACLs small.

With a prototype implementation of the proposed access control architecture over a production-grade filesystem we have experimentally demonstrated a limited performance overhead using microbenchmarks and application-level benchmarks. Furthermore, we have compared our solution with existing approaches that use the identity mapping technique and shown that our prototype scales well to a large number of tenants without affecting the overall system performance. In contrast, multitenancy solutions that perform identity mappings can adversely affect performance when the number of tenants tends to be high. Also, by emulating situations of real collaborative environments where long ACLs are common, we have demonstrated that numerous of long ACLs can adversely affect system performance. However, the ACL sharing technique that we introduced can mitigate this problem.

8.2 Future work

There are several directions of future work related to this thesis. In this section we list a number of interesting topics that need further research.

In this thesis we have proposed an access control architecture to enable secure multitenancy in a private cloud environment. However, the consideration of weaker trust assumptions would be an interesting future research topic which could provide a way for strengthen the security of the proposed solution in order to make it applicable to a public cloud.

Our experimental results indicated that multiple long ACLs can adversely affect system performance and the ACL sharing technique could be an initial solution to this problem. However, this area needs further research in order to improve the structure and the scalability of ACLs. Direct authorization through trust management certificates [39] has been suggested to better meet the requirements for autonomous delegation across organization boundaries. Furthermore, a method was proposed recently for hierarchical access control in federated file services across different administrative domains [32].

An implementation of the complete proposed architecture is also a plan for future work, as well as its integration into a trusted virtualization platform in the datacenter.

Finally, it is necessary for further experimentation with I/O-intensive applications at large scale over different object-based filesystems.

Bibliography

[1] Amazon. Amazon Elastic Block Store (EBS). `http://aws.amazon.com/ebs/`. Accessed: 2013-07-23.

[2] Amazon. Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2/`. Accessed: 2013-08-19.

[3] Amazon. Amazon Simple Storage Service (Amazon S3). `http://aws.amazon.com/s3/`. Accessed: 2013-07-23.

[4] Glenn Ammons, Vasanth Bala, Todd Mummert, Darrell Reimer, and Xiaolan Zhang. Virtual machine images as structured data: the mirage image library. In *Hot Cloud '11: Proceedings of the 3rd USENIX Workshop on Hot Topics in Cloud Computing*, Portland, OR, June 2011.

[5] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, second edition, 2008.

[6] Lee Badger, Timothy Grance, Robert Patt-Corner, and Jeff Voas. Cloud computing synopsis and recommendations. Technical Report NIST SP - 800-146, National Institute of Standards and Technology, May 2012.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, Bolton Landing, NY, USA, October 2003.

[8] Eshwar Belani, Amin Vahdat, Thomas Anderson, and Michael Dahlin. The CRISIS wide area security architecture. In *SSYM '98: Proceedings of the 7th conference on USENIX Security Symposium*, pages 15–30, January 1998.

[9] Sudip Chahal, Jay Hahn-Steichen, Das Kamhout, Rick Kraemer, Hong Li, and Chris Peters. An enterprise private cloud architecture and implementation roadmap. Technical Report IT@Intel White Paper, Intel Information Technology, June 2010.

[10] Gluster Community. GlusterFS Documentation. `http://www.gluster.org/community/documentation/index.php/Main_Page`. Accessed: 2013-09-10.

[11] Jeff Darcy. Building a cloud file system. *USENIX ;login:*, 36(3):14–21, June 2011.

[12] Jeff Dike. *User Mode Linux*. Prentice Hall Computer, 2006.

[13] Roy T. Fielding and Richard N. Taylor. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, May 2002.

[14] Ian Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. A security architecture for computational grids. In *CCS '98: Proceedings of the 5th ACM conference on Computer and communications security*, pages 83–92, San Francisco, CA, USA, November 1998.

[15] FUSE. Filesystem in Userspace. `http://fuse.sourceforge.net/`. Accessed: 2013-09-04.

[16] Vasile Gaburici, Pete Keleher, and Bobby Bhattacharjee. File system support for collaboration in the wide area. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, Lisboa, Portugal, July 2006.

[17] Roxana Geambasu, Steven D. Gribble, and Henry M. Levy. CloudViews: Communal Data Sharing in Public Clouds. In *HotCloud'09: Proceedings of the 2009 conference on Hot topics in cloud computing*, San Diego, CA, USA, June 2009.

[18] Jacob G. Hansen and Eric Jul. Lithium: Virtual machine storage for the cloud. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, pages 15–26, Indianapolis, IN, USA, June 2008.

[19] Dean Hildebrand, Anna Povzner, Renu Tewari, and Vasily Tarasov. Revisiting the storage stack in virtualized NAS environments. In *WIOV '11: Proceedings of the 3rd conference on I/O virtualization*, Portland, OR, USA, June 2011.

[20] Inktank. Ceph Block Device. `http://ceph.com/docs/next/rbd/rbd/`. Accessed: 2013-07-23.

[21] Venkateswararao Jujjuri, Eric V. Hensbergen, Anthony Liguori, and Badari Pulavarty. VirtFS – a virtualization aware file system pass-through. In *OLS '10: Proceedings of the 2010 Ottawa Linux Symposium*, pages 109–120, Ottawa, Canada, May 2010.

[22] Michael Kaminsky, George Savvides, David Mazières, and M. Frans Kaashoek. Decentralized user authentication in a global file system. In *SOSP '03: Proceedings of the 19th ACM symposium on Operating systems principles*, pages 60–73, New York, NY, USA, October 2003.

[23] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *SANE '00: Proceedings of the 2nd International System Administration and Network Engineering conference*, Maastricht, The Netherlands, May 2000.

[24] Angelos D. Keromytis and Jonathan M. Smith. Requirements for scalable access control and security management architectures. *Communications of the ACM*, 7(2), May 2007.

[25] Anil Kurmus, Moitrayee Gupta, Roman Pletka, and Christian Cachin. A comparison of secure multi-tenancy architectures for filesystem storage clouds. In *Middleware '11: Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware*, pages 471–490, Lisboa, Portugal, December 2012.

[26] Leslie Lamport. Paxos made simple. *SIGACT News*, 32(4), Decemper 2001.

[27] Duy Le, Hai Huang, and Haining Wang. Understanding performance implications of nested file systems in a virtualized environment. In *FAST '12: Proceedings of the 10th USENIX conference on File and Storage Technologies*, San Jose, CA, USA, February 2012.

[28] Andrew W. Leung, Ethan L. Miller, and Stephanie Jones. Scalable security for petascale parallel file systems. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 16:1–16:12, Reno, NV, USA, November 2007.

[29] Henry M. Levy. *Capability-Based Computer Systems.* Butterworth-Heinemann Newton, MA, USA, 1984.

[30] Peter A. Loscocco and Stephen D. Smalley. Meeting critical security objectives with security-enhanced linux. In *OLS '01: Proceedings of the 2001 Ottawa Linux Symposium*, Ottawa, Canada, July 2001.

[31] Zhiqiang Ma, Zhonghua Sheng, Gu Lin, Liufei Wen, and Gong Zhang. DVM: towards a datacenter-scale virtual machine. In *VEE '12: Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 105–120, London, UK, March 2005.

[32] Giorgos Margaritis, Andromachi Hatzieleftheriou, and Stergios Anastasiadis. Nephele: Scalable access control for federated file services. *Journal of Grid Computing*, 1(1):83–102, March 2013.

[33] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *SOSP '99: Proceedings of the 17th ACM symposium on Operating systems principles*, pages 124–139, Kiawah Island Resort, SC, USA, December 1999.

[34] MDTEST. mdtest: HPC benchmark for metadata performance. `http://sourceforge.net/projects/mdtest/`. Accessed: 2013-08-21.

[35] Peter Mell and Timothy Grance. The NIST definition of cloud computing. Technical Report NIST SP - 800-145, National Institute of Standards and Technology, September 2011.

[36] Dutch Meyer, Jake Wires, Norman Hutchinson, and Andrew Warfield. Namespace management in virtual desktops. *USENIX ;login:*, 36(1):6–11, February 2011.

[37] Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Mike J. Feeley, Norman C. Hutchinson, and Andrew Warfield. Parallax: Virtual disks for virtual

machines. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 41–54, Glasgow, Scotland, March 2008.

[38] Microsoft. Common Internet File System (CIFS) Protocol. `http://msdn.microsoft.com/en-us/library/ee442092.aspx`. Accessed: 2013-07-30.

[39] Stefan Miltchev, Vassilis Prevelakis, Sotiris Ioannidis, John Ioannidis, Angelos D. Keromytis, and Jonathan M. Smith. Secure and flexible global file sharing. In *ATC '03: Proceedings of the USENIX 2003 Annual Technical Conference, Freenix Track*, pages 165–178, San Antonio, Texas, USA, June 2003.

[40] Stefan Miltchev, Jonathan M. Smith, Vassilis Prevelakis, Angelos Keromytis, and Sotiris Ioannidis. Decentralized access control in distributed file systems. *ACM Computing Surveys*, 40(3):1–30, August 2008.

[41] Yogesh Mundada, Anirudh Ramachandran, and Nick Feamster. SilverLine: Data and network isolation for cloud services. In *HotCloud '11: Proceedings of the 2011 USENIX HotCloud Workshop*, Portland, OR, USA, June 2011.

[42] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. Bootstrapping trust in commodity computers. In *SP '10: Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 414–429, Oakland, CA, USA, May 2010.

[43] Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *NSDI '06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, pages 353–366, San Jose, CA, USA, May 2006.

[44] Lucian Popa, Yu Minlan, Steven Y. Ko, Sylvia Ratnasamy, and Ion Stoica. CloudPolice: Taking access control out of the network. In *Hotnets-IX: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Monterey, CA, USA, October 2010.

[45] Raluca A. Popa, Jacob R. Lorch, David Molnar, Helen J. Wang, and Li Zhuang. Enabling security in cloud storage SLAs with CloudProof. In *ATC '11: Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, Portland, OR, USA, June 2011.

[46] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[47] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conferenc*, Anaheim, CA, USA, April 2005.

[48] Daniel Price and Andrew Tucker. Solaris Zones: Operating system support for consolidating commercial workloads. In *LISA '04: Proceedings of the 18th Conference on Systems Administration*, pages 241–254, Atlanta, USA, November 2004.

[49] Rackspace. Rackspace Cloud Files. `http://www.rackspace.com/cloud/files/`. Accessed: 2013-07-23.

[50] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Phoenix, AZ, USA, February 2007.

[51] Peter Reiher, Thomas Page, Jeff Cook, Stephen Crocker, and Gerald Popek. Truffles - a secure service for widespread file sharing. In *Proceedings of the Privacy and Security Research Group Workshop on Network and Distributed Systems Security*, San Diego, CA, USA, February 1993.

[52] Nuno Santos, Rodrigo Rodrigues, Krishna P. Gummadi, and Stefan Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *Security '12: Proceedings of the 21st USENIX conference on Security symposium*, pages 175–188, Bellevue, WA, USA, August 2012.

[53] Julian Satran, Kalman Meth, Constantine Sapuntzakis, Mallikarjun Chadalapaka, and Efri Zeidner. Internet small computer systems interface (iSCSI). *IETF Request for Comments (RFC), RFC 3720*, April 2004.

[54] Mohammad Shamma, Dutch T. Meyer, Jake Wires, Maria Ivanova, Norman C. Hutchinson, and Andrew Warfield. Capo: Recapitulating storage for virtual desk-

tops. In *FAST '11: Proceedings of the 9th USENIX conference on File and stroage technologies*, pages 31–45, San Jose, CA, USA, February 2011.

[55] Spencer Shepler, Brent Callagan, David Robinson, Robert Thurlow, Carl Beame, Mike Eisler, and Dave Noveck. Network file system (NFS) version 4 protocol. *IETF Request for Comments (RFC), RFC 3530*, April 2003.

[56] Marvin A. Sirbu and John Chung-I Chuang. Distributed authentication in Kerberos using public key cryptography. In *SNDSS '97: Proceedings of the 1997 Symposium on Network and Distributed System Security*, pages 134–141, San Diego, CA, USA, February 1997.

[57] Diana K. Smetters and Nathan Good. How users use access control. In *SOUPS '09: Proceedings of the 5th Symposium on Usable Privacy and Security*, Mount View, CA, USA, July 2009.

[58] Joseph Spadavecchia and Erez Zadok. Enhancing NFS cross-administrative domain access. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 181–194, Monterey, CA, USA, June 2002.

[59] Jennifer G. Steiner, Clifford B. Neuman, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter 1988 Technical Conference*, pages 191–202, Dallas, TX, USA, January 1988.

[60] Douglas Thain. Identity boxing: A new technique for consistent global identity. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Seattle, WA, USA, November 2005.

[61] Douglas Thain, Christopher Moretti, Paul Madrid, Philip Snowberger, and Jeffrey Hemmes. The consequences of decentralized security in a cooperative storage system. In *SISW '05: Proceedings of the Third IEEE International Security in Storage Workshop*, pages 71–82, San Francisco, CA, USA, December 2005.

[62] Satyam B. Vaghani. Virtual machine file system. *ACM SIGOPS: Operating Systems Review*, 44(4):57–70, December 2010.

[63] VMWare. VMWare Workstation. `http://www.vmware.com/products/workstation/`. Accessed: 2013-07-23.

[64] Neal H. Walfield, Paul T. Stanton, John L. Griffin, and Randal Burns. Practical protection for personal storage in the cloud. In *EUROSEC '10: Proceedings of the Third European Workshop on System Security*, pages 8–14, Paris, France, April 2010.

[65] Joshua Walgenbach, Stephen C. Simms, Justin P. Miller, and Kit Westneat. Enabling Lustre WAN for production use on the TeraGrid: A lightweight UID mapping scheme. In *OTG '10: Proceedings of the 2010 TeraGrid Conference*, pages 1–6, Pittsburgh, PA, USA, August 2010.

[66] Jinpeng Wei, Xiaolan Zhang, Glenn Ammons, Vasanth Bala, and Peng Ning. Managing security of virtual machine images in a cloud environment. In *CCSW '09: Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 91–96, Chicago, IL, USA, November 2009.

[67] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *FAST '08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 307–320, Seattle, WA, USA, November 2006.

[68] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: controlled, scalable, decentralized placement of replicated data. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, Tampa, FL, USA, November 2006.

[69] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic metadata management for petabyte-scale file systems. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputingy*, Pittsburgh, PA, USA, November 2004.

[70] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *FAST '08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, San Jose, CA, USA, February 2008.

[71] Assar Westerlund and Johan Danielsson. Heimdal and Windows 2000 Kerberos - How to Get Them to Play Together. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 267–272, Boston, MA, USA, June 2001.

[72] Wikipedia. Dike (mythology), 2013. Accessed: 2013-09-10.

Author's Publications

Giorgos Kappes, Andromachi Hatzieleftherou, Stergios V. Anastasiadis, Dike: Virtualization-aware Access Control for Multitenant Filesystems, Technical Report DCS2013-1, Department of Computer Science, University of Ioannina, February 2013.

Short Vita

Georgios E. Kappes was born in Ioannina, Greece in 1987. He graduated the 2nd High School of Ioannina in 2004 and obtained his B.Sc degree from the Department of Computer Science, of the University of Ioannina in 2011. His B.Sc. thesis was entitled “Logging file access patterns for a more efficient file search on file systems”. Currently, he is a postgraduate student at the same department and a member of Systems Research Group (SRG) of the University of Ioannina. His research interests lie in the fields of virtualization, file and storage systems, as well as security and privacy.