Better Cloud Storage Usability Through Name Space Virtualization

Ernest Artiaga*, Jonathan Martí*, and Toni Cortes*†

*Barcelona Supercomputing Center, 08034 Barcelona, Spain
†Technical University of Catalonia - BarcelonaTech, Dept. of Computer Architecture, 08034 Barcelona, Spain
Email: {ernest.artiaga, jonathan.marti}@bsc.es, toni@ac.upc.edu

***Abstract*—Cloud-based storage synchronization and backup services are widely available. Nevertheless, their usability is often rigid and coarse-grained: for example, most services synchronize specific whole directories, but are not able to synchronize a single file in arbitrary locations of the file system hierarchy. One of the reasons is that these services focus on the challenges of data transfer between the local system and the cloud; consequently, they try to simplify the interactions with the local file system and the users' ways.**

We propose using file system name space virtualization to improve the usability of existing cloud-based synchronization and backup services. Our system introduces a layer that decouples the name space view from the actual organization of the local file system. This way, the user sees a fully-functional view of the file system hierarchy with complete, fine-grained control over the cloudified files and their location. On the other side, the cloud service application sees a view specifically adapted to its needs (e.g. with all cloud-related files concentrated in a single directory).

In this paper, we discuss the requirements and architecture of the virtualization layer. Then, we show the mechanisms used to implement prototypes in two widely deployed operating systems (MS Windows and Linux).

I. Introduction

Cloud-based storage services are widely available. Nowadays, these services are provided to final users by means of a number of commercial products (Box [1], Dropbox [2], JustCloud [3] or SugarSync [4] are some examples). Some popular features of such products are data synchronization among devices, data sharing among users, and backup.

From a technical perspective, some of the challenges that these applications face are keeping the integrity and consistency of data, and optimizing the data transfer between the client system and the cloud. Nevertheless, for the final user, such technical aspects are things that happen behind the scenes and, once a certain quality in the service is achieved, there is another aspect that becomes important: usability.

In terms of usability, cloud-based storage services show a lack of flexibility. The selection of files to consign to the cloud is usually rigid and coarse-grained. For example, it is typically not possible to select a single file in an arbitrary location of the file system for synchronization or sharing: most systems force the synchronization of the whole directory where the file resides. The result is that the user is forced to select a limited number of directories to put in the files that will be under the cloud control and, at most, select some specific files in them. Moving a file outside one of such directories may cause the file to fall out the cloud.

There is a reason behind the rigidity of file organization for cloud-based storage services. The interaction between a local system and the cloud-based storage service is handled by local components: these components are programs or daemons that run in the local computer and act as local agents that communicate with the remote cloud service. These local components usually need to determine if a file has been modified and changes should be synchronized; for that, the local components have to monitor and track changes in the local file system. Modern file systems offer event-driven interfaces to obtain this information without incurring in the high overhead that a continuous polling from applications would cause. However, these interfaces vary in terms of granularity and the details provided: the minimum common service just indicates if changes have occurred in a directory, but it does not necessarily inform about which files have been modified[1], and it is up to the application to collect and store the necessary state to identify the changes. As a consequence, cloud-based services use directories as the basic unit of operation, and try to limit their number as a way to minimize the state required for effectively tracking changes. Unfortunately, this also reduces the ability of the user to organize the files for the cloud in a flexible way, having to abide by the constraints imposed by the cloud-service.

In order to increase the flexibility in the organization of files to be incorporated to the cloud, we propose decoupling the view of the file system name space from the actual underlying directory hierarchy. This virtualization of the name space allows offering different simultaneous views to different users or applications. A direct application of this technique permits a local component of a cloud-based service to keep relevant files concentrated in a small, easy to track set of directories while, at the same time, the user sees files organized according to her preferences and needs.

[1]For example, the `FSEvents` API in Mac OS [5] sends a notification when files in a selected directory subtree have changed, but it does not indicate which ones; the Windows API function `ReadDirectoryChangesW` informs about specific changes in a given directory subtree, though information may be dropped if the notification buffer overflows [6]; finally, the Linux `inotify` interface provides details about changes related to an arbitrary file or directory (but not to subdirectories) [7].

One of the advantages of name space virtualization is that it does not require changes to either the file system change tracking interfaces or the local components of the cloud-based storage services. Yet, it provides the user with increased flexibility.

In this paper, we show how to improve the usability of cloud-based storage systems by using a name space virtualization layer, offering different simultaneous views of the file system for the cloud service components and for the rest of applications. The contributions of this work are as follows: first, we introduce the use of a name space virtualization layer to seamlessly provide different, simultaneous file system views adapted to specific application needs; and second, we show how this layer can be implemented in both Windows-based and Linux-based systems.

In the next section, we describe the principles of the name space virtualization mechanism. In Section III, we show details about proof-of-concept implementations for Windows-based and Linux-based systems. Section IV discusses additional functionality extensions that can be implemented using name space virtualization. We cover related work in Section V, and conclude in Section VI.

II. The Name Space Virtualization Model

In this section, we present some concepts introduced by the virtualization of the name space, and then we discuss the details of the structure of the virtualization layer, as well as some of the design decisions.

A. Decoupling virtual name space from physical layout

Cloud-based storage systems rely on interfaces provided by the underlying file system to track relevant changes in monitored local directories; however, as mentioned in Section I, some of the available mechanisms make tracking the whole directory tree impractical; as a consequence, relevant files are usually confined into a specific set of folders.

The name space virtualization model allows to overcome these limitations without changing the way cloud components work. Particularly, we can let the local components of the cloud-based service interact directly with the native file system, organizing the relevant files in the usual way (i.e. concentrating them in a set of selected folders). We will refer to these subtrees as *cloudified* directories. For the sake of clarity, in the rest of the paper we will refer to the directory hierarchy of the native file system as the *physical name space*, which will be composed of *physical* files and directories. In order to simplify the examples, the cases discussed in this paper will use a single *cloudified* directory; nevertheless, all techniques described are equally valid for a set of them.

On the other hand, the user should be able to organize the files without the restrictions imposed by cloud-based services. This means that we need to provide an alternate view of the file system layout which may differ, at least in some points, from the physical name space (for example, synchronized files will appear anywhere in the directory hierarchy instead of being concentrated in the *cloudified* physical directory). We will call this alternate view a *virtual name space*.

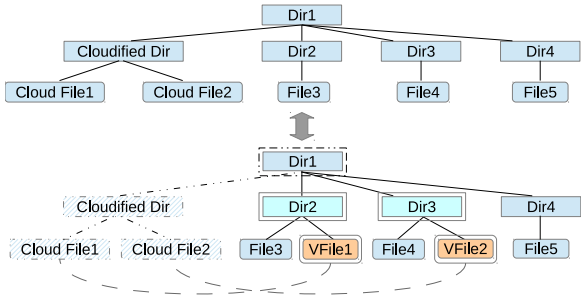

Figure 1. Example of relation between a physical and a virtual name space

In our model, the *virtual name space* will expose part of the *physical name space* enriched with *virtual files*. In this paper, we use the term *virtual file* to indicate an entry in the virtual name space representing a file with a different path in the physical name space: for example, a file to be synchronized in the cloudified physical directory, but being visible and accessible outside that directory through the virtual name space. Similarly, we use the term *anchor directory* to indicate a directory in the virtual name space exposing a modified view of the contents of the physical directory with the same path, adding virtual files associated to that directory and hiding physical files that should not be available in the virtual view. The expression *virtual object* generically refers to objects related to a virtual name space (either a virtual file or an anchor directory), and *virtual entry* indicates the *name* of that object (i.e. the last component of its path).

Fig. 1 shows the relationship between the physical name space (at the top) and a virtual name space (at the bottom). The physical name space has a dedicated cloudified directory containing two files to be managed by a cloud-based service; this is the name space that the local components of the cloud service will see. On the other hand, the rest of applications will see the virtual name space at the bottom of the figure. Directories `Dir1` and `Dir2` act as *anchor directories* and they expose the contents of the corresponding physical directories plus two virtual files (`VFile1` and `VFile2`) which correspond to physical files `CloudFile1` and `CloudFile2`. Regarding `Dir4`, there are no differences between the physical and virtual name spaces.

The cloudified directory in Fig. 1 could be hidden from the virtual name space view, preventing applications from directly accessing and manipulating the cloud-based storage service data. This adds an extra level of protection to the cloudified directory and allows using it to store administrative information for the name space virtualization mechanism (for example, the cloud service itself could be used to synchronize information about the mapping of the cloudified physical files into the virtual name space for a particular device).

The name space virtualization model borrows notions from layered file systems [8]: the construction of the virtual name space is similar to the result obtained by stacking two layered file systems. The main difference is that, for name space virtualization, the top layer is not a whole file system but just a collection of name space entries referring to the actual files in the physical name space (the actual file system): this makes the virtualization layer lighter and easier to implement than a fully-fledged stackable file system composition layer.

The binding between the virtual entries and the physical files is similar to hard links in POSIX file systems, allowing multiple entries to reference the same data file; nevertheless, their semantics differ in two aspects. First, when a user removes a file in the virtual name space, the corresponding entry is also to be deleted from the cloudified directory in the physical name space - and vice versa; so, the removal of an entry in a name space requires the removal of the entries referring to the same file from the other name spaces (conventional hard links must be removed individually, and finding all hard links to a given file is a costly operation). Second, the attributes associated to entries in different name spaces may be different, even when they refer to the same data file: for example, the file access permissions could allow only read-write access for the owner in the virtual name space and permit read-only access for a synchronization application accessing the physical name space (oppositely, hard link entries would share all attributes).

The relationship between entries in the physical and virtual name spaces is transparent to applications: the virtualization layer is responsible for keeping information to track the use of name spaces and maintaining their consistency.

B. Structure of the virtualization layer

Applications perceive the file system name space through path resolution (e.g. when checking if a file exists in a given path and can be open) and directory listings. Inside the file system, the path resolution mechanism converts a given path into a reference to the actual piece of information, which is then used in subsequent operations to access and manipulate the data. Additionally, directory listings also collaborate to expose a picture of the name space by providing the names of the objects located in a particular node of the directory hierarchy. Consequently, our mechanism to provide a coherent virtual name space to applications focuses on providing support for those two operations, while relying on the underlying file system for the rest of storage management aspects (such as storing and retrieving file contents).

Fig. 2 outlines the architecture of our name space virtualization layer. Applications interact with the storage system using the standard file system interface. This way, they are unaware of the presence of the name space virtualization layer: usual access operations such as opening, reading, writing and closing files are available, no matter if the target is a physical or a virtual object. Likewise, operations more

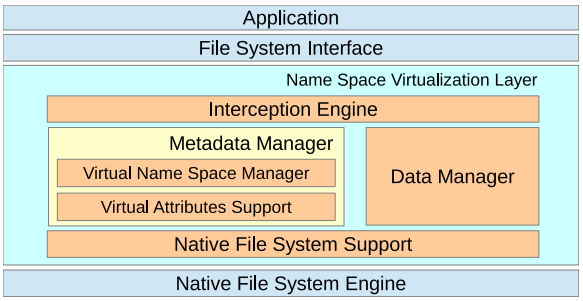

Figure 2. Overall architecture of the name space virtualization layer

closely related to the name space organization (such as creation, renaming or deletion) are identical on both anchor and physical directories.

When a request from an application crosses the file system interface, it is captured by the *Interception Engine* of the name space virtualization layer; then, the *Metadata Manager* decides what to do with the request. Essentially, the *virtual name space manager* detects if the request is related to a virtual entry; if so, the system uses information stored as *virtual attributes* to fulfill the petition (these *virtual attributes* include pieces of data such as the corresponding path in the physical name space or the permissions to be applied when accessing the target object through a particular virtual name space). On the contrary, if the original target is a physical object, the request is forwarded to the underlying file system.

The *Data Manager* is responsible for organizing the storage of data in the underlying file system (for instance, when a new virtual file is created, this module determines where the corresponding physical file should be located according to its cloud-related properties - e.g. if it has to be synchronized or shared). Additionally, the *Data Manager* may monitor the cloudified physical directory in order to detect new files and, if necessary, to incorporate them to the virtual name space (this situation occurs, for example, when a synchronized file is created in a remote device).

The interaction of the name space virtualization layer with the underlying file system is done through the *Native File System support* module. The mission of this module is to deal with the intrinsics of a particular file system and maintain the necessary housekeeping information to allow a correct and consistent access to the actual data.

C. Design decisions

One of the design choices was making the interception mechanism kernel-based. The reason for this was that all the information necessary to process the file system requests, and the execution environment information to take the appropriate decisions, are readily available at the kernel level. Moreover, hiding the interception engine behind the operating system interface also makes the use of the virtualization layer more transparent to the applications: they just have to

issue system calls in the usual way and do not see anything different from the standard operation mode. Additionally, the kernel provides a global view of the status of the system, making it easier to keep the coherence in case of concurrent operations; and also protects the interception mechanism, preventing applications from bypassing it.

On the contrary, the *Metadata* and *Data Managers* are implemented in user space. Here, the main reason was the ease of implementation: standard libraries and tools that can be used at user space are not always available at the kernel level. Moreover, encapsulating the complexities of the name space virtualization logic in a standard process also improves the stability and security of the underlying operating system.

The attributes of virtual objects are stored in database tables. Each object has a unique identifier which is independent of its path. The tables use that identifier as a key, and store relevant information about the object. In particular, for a virtual file, one of the tables stores the path of the corresponding physical file. Ancillary tables contain other pieces of information depending on the functionalities required (for example, access permissions information, a hard link counter for POSIX-like file systems, etc).

The virtual name space manager also uses a table to store virtual directory entries. A virtual entry record contains the parent directory identifier and the name inside the directory as the key, as well as the identifier of the virtual object itself and its type as values. This allows the representation of virtual directory hierarchies.

Another relevant design decision involves defining the semantics of cross-namespace operations, such as the behavior of a *rename* request when the source and the target paths may belong to different name spaces. For the implementation of the prototypes, we tried to take a "least surprise approach" from the user's expectations.

First, we consider that the target of a virtual file rename is always a virtual file. This fulfills one of the main purposes of the name space virtualization mechanism: to improve the flexibility of cloud-based storage services. In our context, a virtual file represents a file under the control of the cloud (e.g. a synchronized file); therefore, even if we move that file around, we still want that file to be in the cloud (which in practice means that it has to be a virtual file with its physical counterpart stored in the cloudified physical directory). In the simple case, both the parent directories of the source and the target entries are anchor directories: then, rename just consists of adjusting the virtual name space information in the *Metadata Manager* (the mapping to the physical file in the cloudified directory remains unmodified). When the target's parent directory is a physical directory, it is automatically transformed into an anchor directory before the actual move takes place (as virtual files are always associated to anchor directories).

When renaming a physical file, the target will be converted to a virtual file if the target's parent directory has been configured to do so (i.e. if the user has selected the whole directory to be synchronized or backed up in the cloud). Also, it will be virtualized if the virtual name space is being used and a virtual file with the target path already exists (in this case, renaming can be understood as replacing the target file's contents with the source file - and if the target were in the cloud, it would be expected to remain there).

III. Implementation

In this section, we explain some details about the implementation of prototypes of name space virtualization layers. We have chosen two different targets for the prototypes: the first is a Windows-based system (Microsoft Windows 7 Enterprise Edition, using NTFS as file system); the second is a Linux-based system (Ubuntu-Server 12.04 LTS distribution, using a POSIX file system - namely ext4). With this target selection, we demonstrate that virtual name spaces are implementable in widely different platforms.

A. Common implementation aspects

Despite the semantics of POSIX and NTFS being different, we have tried to make both implementations as similar as possible in order to facilitate code reuse across platforms. Naturally, the kernel-level code (including interception and the low-level file system support) must be system-specific but, as mentioned in subsection II-C, the majority of the name space virtualization logic is implemented in a user-level service, and most of it can share a common structure across platforms. From now on, we will refer to the set of components of the virtualization layer running in the user space as the User-level Service (or ULS, for short).

The ULS is divided into two layers: a platform-dependent layer transforms the requests captured by the system-dependent interception mechanism into system-independent requests; then, a platform-agnostic layer handles the high level logic and the data structures needed to provide the file system functionality.

The data structures representing the virtual name space and the objects in it are the same in the Windows and Linux prototypes. Each virtual object is designated by an unique identifier, which is used as a key to access database tables with the necessary information to manage the object. The stored data includes file attributes (owner, permissions, etc.), internal management information (such as the actual path of the physical file corresponding to a virtual file), and specific support data for the cloud-based service (e.g. flags indicating if a virtual file or an anchor directory is being synchronized, shared or backed-up, as well as additional data related to this functionality). The virtual objects are linked together to form the virtual name space by means of an entry table, where the key is the composition of the parent directory's identifier and the name of the object inside that directory.

The differences in the file system interfaces are addressed by decomposing the native requests into simpler operations

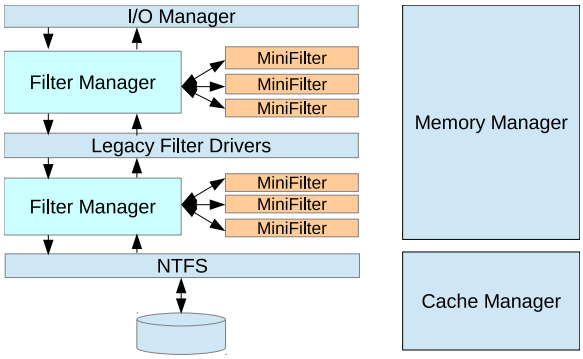

Figure 3. Windows I/O subsystem components

with common functionality or, alternatively, by coalescing multiple low-level requests into a single ULS operation. For example, Windows calls taking a full path argument are transformed into a series of individual path component lookups to determine the internal target object, followed by the invocation of the platform-independent operation on such object; this makes it compatible with the Linux VFS interface, which separates the path resolution from the operation itself. Oppositely, a single conceptual operation such as a file removal consists of a sequence of three low-level requests in Windows: opening the file, setting a deletion flag, and then closing the file and eventually deleting the entry; in this case, the platform-dependent layer in the ULS keeps information to track the request sequence and invokes a single platform-independent removal operation when necessary.

B. Windows-specific implementation details

Fig. 3 represents the Windows I/O stack. The Windows I/O subsystem is packet-based: a request from an application to the I/O service is translated into a series of I/O Requests Packets (IRP) by the I/O Manager [9]. IRPs are sent down the I/O stack for a particular device; such stack usually consists of a series of filter drivers, which may alter the information contained in the IRP (and even generate new IRPs). The stack ends up in the file system driver, which communicates with the physical device driver and responds to the IRPs. The components in the I/O stack also interact closely with the Memory Manager and the Cache Manager [10].

Recent versions incorporate a special type of filter driver called 'Filter Manager'. The Filter Manager acts as a place-holder to plug *minifilters* [11]. The main difference between a minifilter and a filter driver is that a minifilter does not need to implement all IRP operations: it may register to receive a subset of the IRPs, and the missing functionality is provided by the Filter Manager itself. Several instances of the Filter Manager may appear at different depths of the I/O stack.

A minifilter may register both a 'pre-operation' and a 'post-operation' callbacks. The 'pre-operation' callback is invoked when the IRP is going 'down' the I/O stack; the 'post-operation' callback is invoked when the response to the IRP is traveling 'up' to the application, once the request has been processed by the lower levels. Additionally, a minifilter can declare the processing of an IRP completed (effectively preventing it from reaching the lower levels and the file system) or, on the contrary, it may generate additional IRPs.

The Windows prototype of our name space virtualization layer uses a minifilter to intercept IRPs related to file system operations. Due to the fact that a 'high level' operation can consist of a series of IRPs, this minifilter is able to aggregate information and keep the necessary kernel state to build-up a service request for the ULS. The minifilter may also cache data from the ULS to be reused for several IRPs.

One of the most important IRPs for the name space virtualization is `IRP_MJ_CREATE`. This IRP either creates a new file or directory, or opens it if the target already exists. It receives the path of the target object in the arguments; so, handling it correctly is critical in order to generate a consistent virtual view of the file system.

The interception minifilter collects the `IRP_MJ_CREATE` arguments and related information (such as the issuer process, security context, etc.) and sends them to the ULS during the 'pre-operation' step. The ULS then discriminates the name space under which the request must be processed. The simple case corresponds to the physical name space: the ULS does no additional processing, and the minifilter just forwards the IRP to the lower levels without alteration.

The use of the physical name space is activated when a request comes from the ULS itself or a process explicitly entitled to use it (e.g. the local components of a cloud-based storage service). For convenience, when a new process is created, it inherits the name space settings of its parent. The rest of processes also fall back to the physical name space when there is no virtual object in the requested path (that would be the case, for example, of a process accessing `Dir1/Dir4/File5` in the virtual name space of Fig. 1).

If the target of `IRP_MJ_CREATE` is a virtual entry, then some additional processing is required. Upon reception of the request, the ULS resolves the path against the virtual name space. If the target does not exist, the *Data Manager* in the ULS decides the physical path where the file should reside and issues a create request using the standard, user-level file system interface; then, the virtual name space is updated with the new entry, and the link between the new virtual object and the actual physical path is stored by the ULS *Metadata Manager*. On the contrary, if the target already existed, the mapping to an actual physical path will be retrieved by the *Metadata Manager* and fed into the *Data Manager* to open the underlying file.

In both cases, the ULS *Data Manager* actions result in issuing a standard file system operation which, in turn, will generate a new IRP going down the I/O stack. As mentioned before, the virtualization minifilter will let pass this IRP unaltered, and the ULS will get an open handle for the physical file (namely a user-level file descriptor).

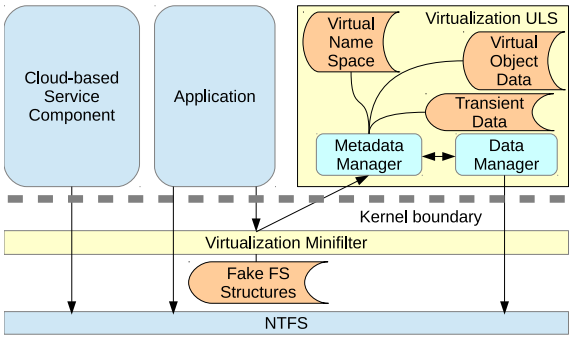

Figure 4. Virtualization path in the Windows-based prototype

The last step consists in binding the obtained handle on the physical file with the original IRP still pending in the virtualization minifilter. For that, the ULS *Metadata Manager* stores the handle and other relevant information in a transient data area, and sends a reference to that information back to the minifilter. Then, the minifilter creates fake structures to represent the virtual file and associates the ULS reference to them, so that the virtualization layer can retrieve the proper information when further operations are performed on the virtual object.

A representation of the IRP processing can be seen in Fig. 4. It is important to remark that, when dealing with an IRP related to a virtual file, the processing ends in the virtualization minifilter, and the IRP is not forwarded to the lower levels of the I/O stack (and, in particular, it never reaches the underlying file system). This adds two responsibilities to the minifilter: first, it must create the kernel structures that are usually created by the file system driver to represent a file system object (and fill them with consistent values); second, it must detect all references to these structures in the I/O stack and divert them to the virtualization ULS, effectively preventing them from reaching the lower levels of the I/O stack and the underlying file system itself - failure to do so could result in file system inconsistencies.

Once the `IRP_MJ_CREATE` has been successfully processed, further IRPs related to the same file will provide a reference to the kernel structures representing the file. The minifilter can easily distinguish native file system structures from fake ones (corresponding to virtual objects) by checking the associated ULS information that was set up during the `IRP_MJ_CREATE` processing. In general, if the IRP corresponds to a virtual file, it will be diverted to the ULS for processing; nevertheless, it may be possible to fulfill the request completely at the kernel level if all the necessary information is available (a particular case are read and write operations: the minifilter may interact with the cache manager to complete the request if the required data is being cached).

Another essential operation for name space virtualization is directory listing. This operation is translated into the I/O stack as an initial `IRP_MJ_CREATE` followed by a series of IRPs of type `IRP_MN_QUERY_DIRECTORY`, which provide buffers to fill-in with directory entry information. Similarly to the `IRP_MJ_CREATE` request, IRPs processed under the physical name space are just forwarded to the lower I/O stack levels, and those corresponding to a virtual name space are forwarded to the ULS. In the latter case, the ULS will access the underlying physical directory, combine its entries with the entries from the corresponding directory in the virtual name space, and eventually hide any entries not to be available through the virtual name space.

C. Linux-specific implementation details

The implementation of the name space virtualization layer for Linux follows the same principles as the implementation based on Windows: the interception of file system request is based on kernel grounds, while the bulk of processing (the *Metadata* and *Data Managers* from Fig. 2) is implemented at the user level. In this section, we will focus on the differences with respect to the Windows-based implementation.

The VFS (Virtual Filesystem Switch [12]) mechanism inside the Linux kernel was initially designed to facilitate the addition of file systems [8]; it provides an interface for registering the code that should be executed to fulfill each low level file system operation (such as name lookups, getting and setting file attributes, reading and writing data, etc). This set of callbacks also provides the necessary support to capture requests related to the file system name space.

The interception mechanism of our name space virtualization layer is based on the Linux VFS callbacks, but we use them via FUSE [13] instead of registering our code directly. FUSE (Filesystem in USErspace) provides a kernel module that hooks into Linux VFS and exports the callbacks to a user space application. FUSE is a standard component of current Linux kernels and provides a stable and widely used platform for implementing file system features in user space. The decision to use FUSE was driven by the ease of development as well as its proven robustness. On the other hand, it hides some kernel-level information and has limited ability to interact with other kernel components, though this does not substantially hinder the implementation of the prototype.

One of the main differences between the interception mechanisms in Windows and Linux is that the combination of VFS and FUSE does not allow file system requests to 'pass through' the virtualization layer and reach the 'underlying file system': FUSE diverts all requests to the user-level service (ULS). In particular, this includes both requests coming from the cloud-based service components (which should access the physical name space) and from the rest of the applications (which should access the virtual name space). Therefore, it is the ULS who must explicitly forward to the underlying file system the operations related to the physical name space.

Fig. 5 illustrates the behavior of the virtualization layer in Linux. The virtualization layer attaches to VFS via the

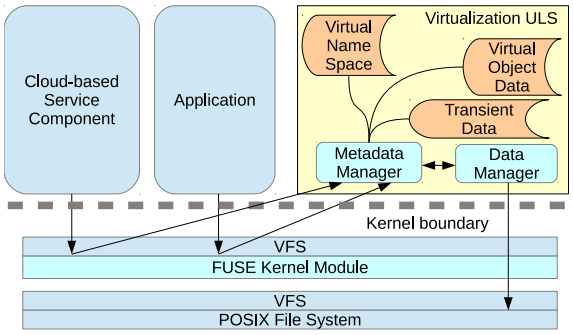

Figure 5. Virtualization path in the Linux-based prototype

FUSE kernel module and appears as a conventional file system mounted on top of the physical name space (i.e. the underlying POSIX file system). File system operations from applications are converted into one or more VFS requests, which are intercepted by the FUSE module and diverted to the ULS; if the request is to be processed under the physical name space, the ULS generates the corresponding series of operations for the underlying file system.

In order to bypass the virtualization layer and perform operations directly on the underlying file system, the ULS acquires a handle of the root directory *before* the virtualization layer is mounted on top of it. Then, whenever a physical object is to be accessed, the physical path is resolved after this handle, instead of using the conventional path resolution mechanism; in the prototype, this is enabled by using the `openat` system call and its relatives.

As discussed in subsection II-B, path resolution is one of the key elements to provide the perception of a virtual name space to applications. In the case of Windows, the requirement to resolve a path was embedded as part of the IRPs representing the different operations. In the Linux implementation, path resolution is performed separately from the actual operation via explicit VFS+FUSE `lookup` callbacks; the path resolution procedure returns an internal identifier which is then used to invoke the actual operation (`create`, `mkdir`, `unlink`, etc).

Apart from the formal difference, the treatment of `lookup` in the Linux-based prototype is equivalent to the Windows-based prototype: if the request refers to the physical name space, the ULS generates a new request for the underlying file system; otherwise, the ULS tries to resolve it against the virtual name space data, and may fall back to check the underlying file system if the entry is not found.

The processing of directory listings is equivalent to the procedure followed in the Windows prototype, with the sequence of `IRP_MJ_CREATE` and `IRP_MN_QUERY_DIRECTORY` IRPs translated into an `opendir` followed by a series of `readdir` callbacks, with essentially the same semantics as their Windows counterparts.

User notification of file system changes in the Linux prototype deserves a specific comment. In the Linux kernel, the change notification mechanism (usually *inotify* [7]) is implemented inside the VFS and reacts to requests from the user space, without requiring any action from the underlying file system. Specifically, FUSE delegates all change notification handling to the Linux VFS. Unfortunately, this means that FUSE does not provide any means to notify the user applications that changes have occurred behind the scenes.

This limitation would affect a component of the cloud-based service that wished to track the changes in the cloudified directory through the virtualization layer. When the user application acts on an arbitrary directory in the virtual name space, the ULS could directly update the cloudified directory in the underlying file system; but then, the VFS associated to the virtualization layer could not relate the virtual directory with the physical cloudified directory and the cloud-service component would not be notified.

The initial solution used in the Linux-based prototype was making the *Data Manager* of the ULS redirect updates to the cloudified directory through the virtualization layer, instead of writing directly to the underlying file system; that makes the VFS aware of the changes, so that notifications work properly. An alternative solution consists in modifying FUSE to allow informing the VFS about the changes made by the ULS. FUSE already provides a mechanism for sending requests from the user-space to the kernel module (e.g. to invalidate data cached by the kernel); the same feature can be used to push change notification events into the kernel and, upon reception, the FUSE kernel module can invoke the proper change notification kernel interface.

IV. Extensions

The model of having a physical and a virtual name space can be easily extended to multiple, simultaneous, virtual name spaces. In fact, the prototypes described in the paper permit such multiplicity, and the only addition needed is the logic to select the appropriate name space to resolve a given path.

Using multiple virtual name spaces enables presenting specialized views of the file system to a particular application or group of applications. Following the case described in the paper, we could have, for example, multiple cloud-based storage services using incompatible directory layouts coexisting in the same system. Additional features can also be added, such as hiding part of the files to certain services, quarantining the changes from certain applications or selectively encrypting sensitive data.

V. Related work

The proposal in this paper is related to previous work in many areas, including cloud storage services, file systems and operating systems.

There are multiple commercial products providing cloud-based storage with added value services, such as data synchronization across devices or collaborative sharing. Many

of these services are integrated in the form of one or more selected directories in the existing file system (like Box [1], Dropbox [2], JustCloud [3] or SugarSync [4], to name just a few). In all these products, the use of the cloud is activated by moving the files into specific places (either drives or directories); on the contrary, our proposal enables the cloud services support independently of a particular file's location.

Name space virtualization has been used to provide a unified virtual view of multiple file systems: well-known examples of this technique are union file systems [14] [15]. Fan-out stackable file systems [8] are one of the mechanisms allowing the construction of unified virtual name spaces; for example, RAIF [16] uses this technique to combine several file systems with the same directory hierarchy under a single virtual name space, with the goal of diverting files to the most appropriate underlying file system according to each file's characteristics. Our approach uses a similar technology with the opposite goal: instead of unifying several file systems under a single virtual name space, we provide multiple virtual name spaces for a single underlying file system.

Regarding technical aspects, the implementation of the minifilter-based interception mechanism for the Windows prototype borrows ideas from the "isolation driver" concept from OSR [17] [18]. The Linux prototype relies on FUSE [13] to interface with the underlying file system; it also takes advantage of the experience of COFS [19], which provides a virtualization layer to alter the layout of a distributed file system in order to minimize consistency conflicts.

VI. Conclusion

Cloud-based storage services provide means to replicate local data and keep it synchronized across different platforms. To keep this data up to date, local components of the cloud services need to track changes in the local file systems. However, current mechanisms to perform this tracking require rigid file organizations in order to be effective.

Our proposal consists of improving the flexibility of cloud-based directory hierarchies by providing multiple simultaneous views of the file system organization: a name space virtualization layer allows user applications to access a view of the file system without organizational restrictions, while cloud components maintain a view of the file system structured according their needs.

We have described prototype implementations of such name space virtualization layer instantiated in both Windows and Linux platforms, demonstrating the feasibility of the approach in actual systems.

Acknowledgments

We would like to thank Juan González de Benito, Thanos Makatos and Jan Wiberg for their contributions to the design and implementation of the Windows-based prototype of the name space virtualization layer. This work was partially supported by Spanish MECD under grant TIN2012-34557, and the Catalan Goverment under the 2009-SGR-980 grant.

References

[1] "Box," Web site: http://www.box.com.

[2] "Dropbox," Web site: http://www.dropbox.com.

[3] "JustCloud," Web site: http://www.justcloud.com.

[4] "SugarSync," Web site: http://www.sugarsync.com.

[5] "File system events programming guide," Online document: http://developer.apple.com/documentation/Darwin/Conceptual/FSEvents_{}ProgGuide/, 2012.

[6] "MSDN: Desktop app development documentation: Directory management functions," Online document: http://msdn.microsoft.com/en-us/library/windows/desktop/aa365465(v=vs.85).aspx, 2013.

[7] R. Love, "Intro to inotify," *Linux Journal*, September 2005.

[8] E. Zadok, R. Iyer, N. Joukov, G. Sivathanu, and C. P. Wright, "On incremental file system development," *Trans. Storage*, vol. 2, no. 2, pp. 161–196, 2006.

[9] M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals, Fifth Edition*. MS Press, 2009, ch. 7, I/O System.

[10] R. Nagar, *Windows NT File System Internals*. OSR, 2006.

[11] "MSDN: File system minifilter drivers," Online document: http://msdn.microsoft.com/en-us/library/windows/hardware/ff540402(v=vs.85).aspx, 2012.

[12] D. P. Bovet and M. Cesati, *Understanding the Linux kernel, Third Edition*. O'Reilly, 2005, ch. 12, The Virtual Filesystem.

[13] M. Szeredi, "FUSE: Filesystem in USErspace," Web site: http://www.fuse.org, 2005.

[14] J.-S. Pendry and M. K. McKusick, "Union mounts in 4.4BSD-lite," in *TCON'95: Proceedings of the USENIX 1995 Technical Conference*. Berkeley, CA, USA: USENIX Association, 1995.

[15] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair, "Versatility and unix semantics in namespace unification," *Trans. Storage*, vol. 2, no. 1, 2006.

[16] N. Joukov, A. M. Krishnakumar, C. Patti, A. Rai, S. Satnur, A. Traeger, and E. Zadok, "RAIF: Redundant array of independent filesystems," in *MSST '07*, Sept. 2007, pp. 199–214.

[17] "Getting away from it all: The isolation driver (part i)," *The NT Insider*, vol. 17, no. 2, Aug. 2010.

[18] "Getting away from it all: The isolation driver (part ii)," *The NT Insider*, vol. 18, no. 1, Jan. 2011.

[19] E. Artiaga and T. Cortes, "Using file system virtualization to avoid metadata bottlenecks," in *DATE'2010: Design, Automation and Test in Europe*, March 2010.