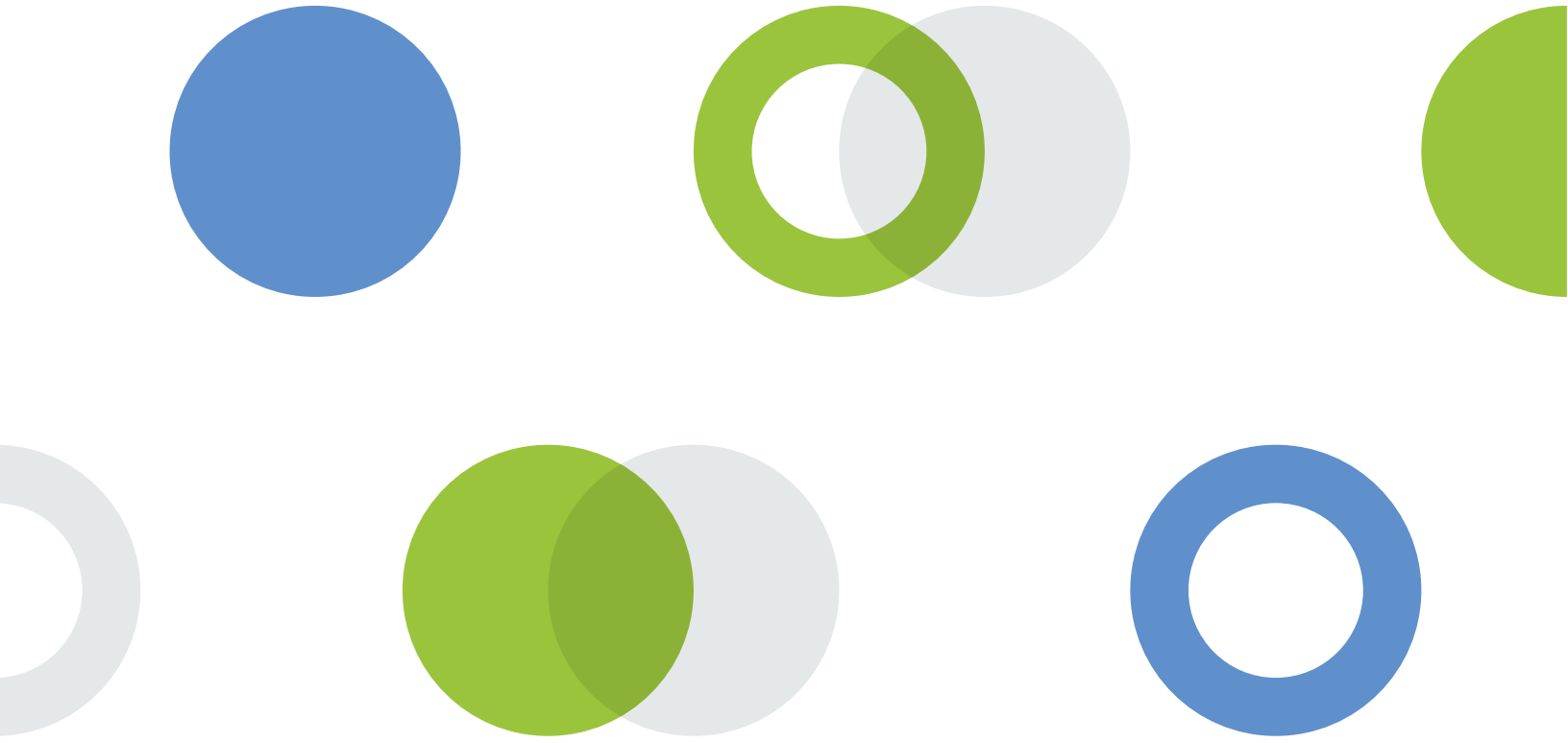
Cloud-Native File Service

ARCHITECTURAL OVERVIEW

HIGH-PERFORMANCE SERVICE FOR DISTRIBUTED WORKLOADS



LucidLink

Table of Contents

Our Vision

Cloud-native file services will continue to evolve, eventually allowing the bulk of the data currently stored on-premise to be moved to the cloud. This transition has so far been hampered by a combination of inadequate performance, security and privacy guarantees, and features.

In recent years, a confluence of several trends has unlocked new opportunities for businesses to migrate file service workloads to the cloud.

The first trend is the advancements in the speed and reliability of the Internet infrastructure. While only ten years ago the average connection speeds were around 10Mbps, now 1 Gigabit connections are common and rival those of LAN environments. Such two orders of magnitude boost in performance fundamentally change what's possible to do over the Internet. The trend will continue towards 10Gbps with direct-to-cloud connections already available and 5G adoption in the coming years. Latencies have also gone down through various infrastructure advancements and all these improvements have literally brought the cloud closer.

The second trend is the maturing of commercial cloud storage. The current level of elasticity, durability, and availability of the cloud storage platforms make it suitable for even the most risk-averse organizations. The most demanding and mission-critical data workloads can now be stored and run in the cloud.

The industry has so far responded with mostly bifurcated offerings for either on-premise or cloud storage solutions. The choice has been to either keep your data and associated compute workloads on-premise or move both off-premise, typically to the public cloud. At LucidLink, we believe a better approach is possible, one that unifies the two by seamlessly transforming cloud storage into another tier of local storage, allowing data to seamlessly flow between tiers so it can be migrated, replicated, and shared securely.

The rapid increase in distributed workforces and workflows now requires instant access to data regardless of where it's stored. Moving previously siloed local data off-premise would answer this need by making file data accessible from anywhere. The concept of data gravity is real, but software and infrastructure advances will reduce its effects, allowing a lot more flexibility in where applications are run relative to the data while maintaining the existing tried-and-true file semantics.

The evolution in file system design will naturally move beyond local storage to incorporate access to repositories over distance, whether in public or private clouds. LucidLink's unique technology positions it at the forefront of this natural evolution in the world of enterprise file sharing.

LucidLink Filespaces is a new generation, advanced, general-purpose distributed file system architected for the cloud that uses any off or on-premise object storage as a back-end repository. It leverages the outstanding properties of modern object storage systems such as durability, elasticity, and scalability as a foundation to offer an infinitely scalable, elastic, and durable file service. Data is streamed on demand directly to and from the object store.

Filespaces is designed to address the business needs around storing large data sets on and off premise and accessing them over distance while offering best-in-class security and privacy guarantees. It further allows businesses complete control over where the data is hosted to respond to varying economic, performance, and data sovereignty requirements. Additionally, it offers advanced file capabilities like snapshots, with future extensions like deduplication and multi-cloud support. LucidLink Filespaces is a no-compromise solution for leveraging file services for private or public clouds.

Existing Technologies

NFS/SMB

Both NFS and SMB protocols were designed in the early 80s for the Unix and Windows worlds respectively. They were and still are the most widely used standards for network file access. The trouble is that they were designed for entirely different environments, i.e., low-latency, high-bandwidth LANs. As such, these protocols tend to be very chatty with many server round trips for even basic operations. This approach breaks down over distance where this chattiness is exacerbated by high-latency, which leads to dramatic performance degradation. Their aging architecture becomes further evident when used over the Internet, for which they lack built-in security, require complex network configurations with firewalls/VPNs, and cannot leverage Single Sign-On (SSO) services. Additionally, they don't play well across both Windows and Unix and lack both mobile support and modern collaboration features. Despite their numerous shortcomings, they are still nonetheless the go-to solution for direct remote access.

Filespaces has been designed from the ground up to address the limitations of both the NFS and SMB protocols. It is a modern file system that works equally well over LAN and WAN, has a high-performance concurrent streaming architecture, reduced network chatter and a very robust security model, and is compatible with all operating systems. This document explores the specific design choices in further detail below.

File Sync and Share

The first successful implementations of distributed file services over the Internet were based on file synchronization. File synchronization is another term for automatic file copying, which was a smart way to overcome the unreliable and intermittent connections of the past. By virtue of keeping all file replicas synchronized, file-syncing forms a distributed file system of sorts, one that is eventually consistent and requires complete replicas on all participating nodes. It has been wildly successful, becoming the bedrock for all content collaboration technologies. However, due to its requirement to maintain full replicas across all nodes with its associated high demand for network bandwidth and storage, it has also limited its applicability to smaller data sets typically in the GB to low TB range.

In contrast, LucidLink does away with synchronization and streams data on demand directly from the cloud as needed by the application. The single source of truth is kept in the cloud, and frequently accessed data is cached locally to deliver the best of both worlds in terms of performance and flexibility when accessing TB to PB-range volumes.

Object Storage With File System Proxies

Object storage was invented to address the scalability limits of the prevailing storage systems of that era. It moved some of the intelligence of the file system down to the object storage system while using simpler access semantics than the file interface. In practice, this meant separating the data path (reads, writes) from the metadata operations of a typical file system and delegating the data storage responsibilities to the object storage devices. Interestingly, the original invention was not meant to replace file systems but to enhance their capabilities. However, object storage really took off with

the advent of Amazon's Simple Storage Service (S3) offering, where it was sold as a distinct cloud service. The simpler object storage semantics fit the HTTP web protocols particularly well, and over time S3 built on top of HTTP became the de facto market standard for object storage access.

Some applications born in the cloud have enthusiastically adopted S3, while the rest of the industry has primarily used it for backup and archive purposes. The tremendous success of Amazon S3 object storage has clearly validated the approach for storing unstructured data in the cloud, but despite its adoption, object storage is not a file system and cannot be used as such.

Attempts at bolting on simple file system proxies in front of object storage fail for several reasons. First, object storage has a flat namespace; therefore a hierarchical file system must be projected onto this flat namespace. This leads to inefficient file metadata operations, e.g. a simple folder rename can potentially involve copying the entire folder sub-tree with all its objects. Second, as objects are updated in their entirety, the random read/write capabilities of file systems are lost. Third, Amazon and some other vendors have traded consistency for performance, employing the so-called eventual consistency data model, in which S3 GET operations are allowed to return stale data for some indeterminate amount of time. This inconsistency window has no predefined upper boundary, so it is impossible to provide a regular file system interface, which needs stronger forms of consistency, e.g. read-your-write consistency. All in all, file system proxies do not amount to an actual file system, and the user experience suffers as a result.

To combat these challenges, LucidLink has taken a completely different approach from the simple proxies defined above. It implements its own data layout model, based on a log-structured design on top of the object store. Files are split into multiple objects similar to the way local file systems use underlying block devices. A separate metadata database is maintained to offer a complete file system functionality. Incidentally, this was how the original authors of object storage envisioned its usage, but LucidLink applies it in the cloud framework.

Cloud Storage Gateways

Storage gateways were the first attempt to bridge the gap between on-premise and cloud storage. They are either virtual or physical on-premise NAS filers that can tier into the cloud. A great way to bridge branch offices, gateways address the collaboration needs of distributed teams to a certain degree. They are particularly useful for environments where workloads and teams are co-located so they can all benefit from the large cache of a typical gateway device. Where they fall short are cases where workloads/teams are dispersed across multiple locations, prohibiting the deployment of a single gateway in each location. Such appliances also come with additional CapEx and operational overhead. Last but not least, gateways are scale-up server devices, which by definition funnel all traffic, thus inhibiting scalability.

LucidLink obviates the need for gateways altogether as it streams data directly to the underlying object store. It untethers the computing devices from their location and allows the same uniform experience, whether in or outside the office, on-premise or in the cloud. LucidLink's software-only, cloud-first approach significantly expands the use cases for cloud file systems, provides much better integration with the host operating system, and eliminates CapEx.

A Modern Approach to File System Design

Basic Architecture

LucidLink Filespaces provides a uniform shared, global file system namespace across any device connected to the Internet. It is a software-only solution, which delivers the intelligence to make any simple object storage act as a high-performance, file system volume shared, and accessible from anywhere.

More importantly, the technology has been specifically developed to offer a near-local user experience, even when accessing data over distance. From a high-level perspective, it functions like a network attached storage or file server, but both the clients and data repository can be located anywhere. All authorized client devices can see and access the same shared data set, irrespective of their operating system or location, as long as they have network access to the object store.

The file system is comprised of 3 main functional components:

1. Client software
2. Metadata service (a.k.a. the hub)
3. Object store

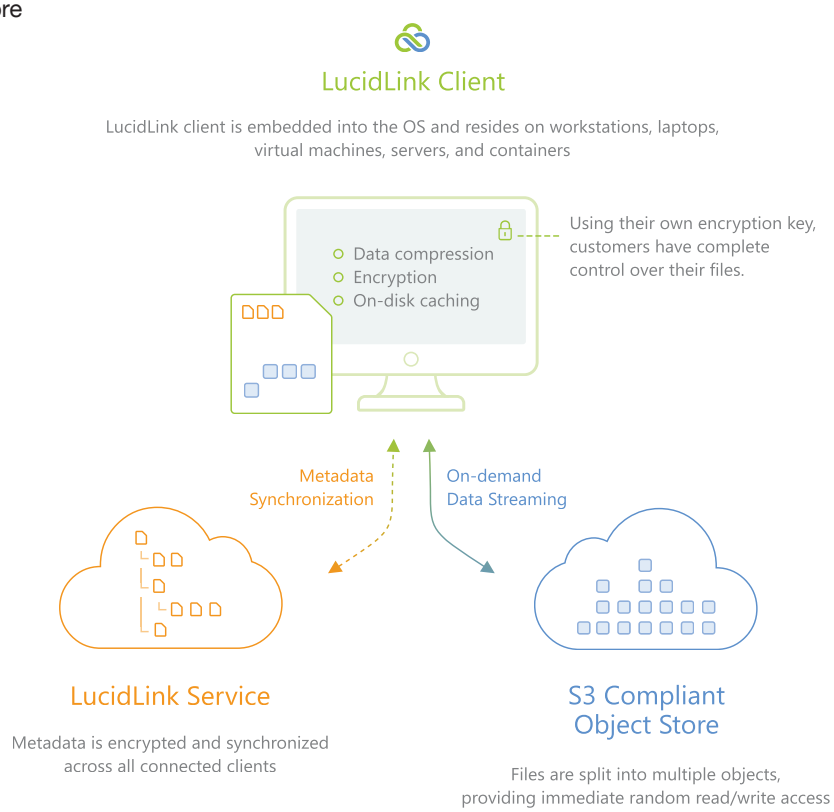


Figure 1

First, each client device runs a software agent which integrates with the local operating system to present a file system mount point that appears as a local file system volume. In contrast to other solutions, which may expose their file system through NFS or SMB protocols, LucidLink's file system client

talks directly to the metadata service and the object store on the backend. There are no intermediary gateways or file servers.

The second component, the metadata service, is a discrete process typically offered by LucidLink as a service, which is entirely transparent to the end-user. Its primary function is to provide file system metadata information to all client nodes—while keeping the metadata separate from the object store. This is not performance critical because it's not on the data path, and except for file locking, is not generally part of any IO processing.

The third component, the object storage itself, is the data repository where all file content is stored. File data is first compressed and encrypted on the client device before being stored in the object store.

Metadata is continuously synchronized between devices, while the data is kept in place and only fetched as needed (Figure 2). Most of the intelligence is built into the client software, which assembles the metadata and the data to present a complete, POSIX-compliant, random-access file system to its host operating environment.

Unlike other services, data is not synchronized, but instead streamed on-demand to and from each client device. Data streaming is based on the requests, which local applications running on each device make to the file system. LucidLink's approach offers instant access to extensive data sets that are kept securely in the object store but are immediately available when needed.

Performance is greatly enhanced through a number of software techniques ranging from metadata synchronization, local on-disk caching, intelligent prefetching, inline compression, to multiple parallel streams (see below).

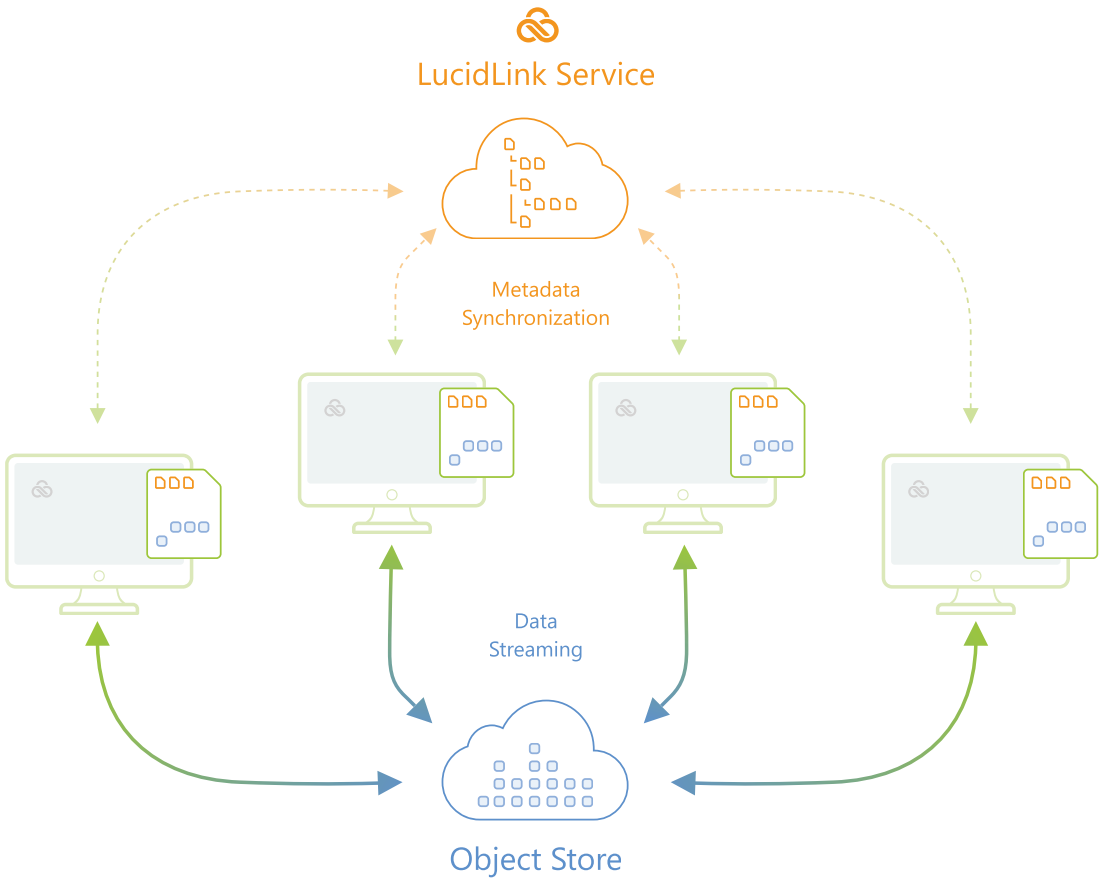


Figure 2

A significant limiting factor of the S3 interface is the fact that objects represent atomic units of data, which can only be manipulated as a whole. This goes against the random read/write semantics of file systems where any section of a file can be updated at will. To address this limitation, LucidLink implements its own data layout where each file is constructed from multiple objects, and each object only represents a segment of the file. This is analogous to how local file systems layout data over equally-sized blocks in block storage devices. The approach allows objects to be viewed as a rough equivalent to blocks in traditional file system implementations.

Design Requirements

Lucidlink is architected around the following requirements:

- Provide regular file system semantics
- Work well over distance, i.e., in high-latency environments
- Provide a shared, unified namespace
- Support all modern desktop and mobile platforms
- Offer the highest degree of security and privacy
- Host large data volumes in the PB range
- Support any object store
- Maintain a low footprint on the client devices
- Scale linearly with the underlying object storage
- Support off-line (disconnected) mode of operation
- Provide high-level collaboration features

The design also incorporates support for advanced storage features including instant snapshots, client-side deduplication, and zero-copy file clones.

Future advancements will deliver multi-cloud capabilities in the form of striping, tiering, and replication across multiple S3 buckets. Striping addresses limitations around the number of objects that can be stored in a single bucket for some S3 providers. Tiering allows moving infrequently accessed data to lower-cost buckets or providers. Finally, replication to multiple regions or providers offers an even higher degree of durability, reduces the distance to data by allowing each node to access its closest location.

Design Principles

Universality

LucidLink provides the intelligence or the 'glue' between any client devices distributed across the Internet and any object storage. To achieve that, LucidLink designed the system to be both universal on the front as well as on the backend:

- Ability to support all major platforms, i.e. Windows workstations & servers both 32 & 64-bit, macOS, all Linux variants both 32 and 64-bit, as well as Android and iOS

- Ability to utilize any S3-compliant object storage provider and only use the minimum set of operations, providing compatibility with almost all on-premise and cloud vendors.

Such universality allows for very flexible deployments to accommodate different use case scenarios, including on-premise, hybrid, and in-cloud workloads.

Split-Plane Architecture

While classical client-server models work great, they also impose scalability limits due to all traffic flowing through the server. To handle the scalability requirement above, LucidLink has opted to eliminate the server component so that every client node can stream directly to and from the object store, which is itself based on a scale-out architecture. With this method, hundreds of client nodes accessing the same namespace could be talking to hundreds of servers on the storage backend, to achieve horizontal scalability.

The lack of server creates the need to maintain the metadata separately, which leads to a split-plane architecture with separate data and metadata planes. The metadata plane is controlled through a metadata service called a hub, typically provided by LucidLink and entirely transparent for the end-user. The metadata service acts as a hub for all client nodes. It coordinates all metadata synchronization operations between them, perform garbage collection (see below), and provides other services like global file locking, snapshot management, etc.

End-To-End Security

Transferring business data and hence intellectual property outside a company's organizational perimeter is fraught with security risks, and as a result, still represents a significant impediment to adoption. Most modern cloud storage systems provide some encryption to satisfy data privacy requirements. Unfortunately, this is typically a weak form of security known as server-side encryption, in which data is encrypted in transit and on the servers using provider generated keys. As a result, the storage provider has complete access to their customers data.

LucidLink believes that the need to protect businesses' most valuable assets should go beyond this simple model to offer a much higher degree of confidentiality—one which guarantees that only the customer is able to unlock and see the data.

To that end, the security in LucidLink's system is achieved through end-to-end encryption. With the use of AES-256 in GCM mode by default, both the metadata and the data are encrypted with customer keys. Each file has its own encryption key, and its parent folder's encryption key wraps each file's key. The system administrator grants individual users access to certain keys in the file system hierarchy, which allows them to see only the content which they have been given access to in the subtree. This provides fine-grained user access control, without a server acting as a gate-keeper. On top of this encryption, all network communication between the hub, the object store, and the client devices is additionally encrypted through TLS.

Most importantly, this client-side encryption guarantees that only the client devices can decrypt the metadata or data. It also prevents even LucidLink and the object storage provider from seeing the content or any other meaningful information about the data being transferred or stored.

Bimodal Consistency

The success of NoSQL systems has transformed our understanding of distributed database design. It has focused on the simplicity of the data model in order to unlock scalability. The CAP theorem has dominated this system design with most database implementations favoring availability over consistency and dispensing with ACID transactions.

LucidLink applies NoSQL design principles to file systems, so similarly, the model is relaxed to attain significant performance gains. Valuable insight is that there are many file use cases for which durable consistency is not required. In those cases, LucidLink relaxes the strong consistency guarantees and instead employ an eventually consistent model, a form of weak consistency. Simply stated, a file update on one node may not be immediately visible on other nodes. This model however should not be confused with the AWS S3 eventual consistency—this model employs a robust client-centric consistency. From the perspective of each device the file system behaves consistently, although different devices may see external updates in any order.

This eventual consistency model leads to possible semantic conflicts and the need for subsequent conflict resolution. Fortunately, studies have shown that in practice, even in large environments, conflicts occur rarely. LucidLink currently uses a last-write-wins approach. Given its log-structured design, which preserves all updates from all nodes, it can easily implement other more sophisticated conflict resolution modes.

Collaborative applications designed to use shared data models, e.g., in architecture, engineering, and construction cannot work with an eventually consistent file system as defined above. Such systems, however, use file locking primitives to serialize their access to the shared project. LucidLink has implemented global file locking to accommodate these scenarios. It dynamically switches between eventual consistency and strong consistency modes so in the presence of file locks it falls back to a strong consistency mode. This means that a node updating a file within a lock is guaranteed to flush its updates to the object store before releasing the lock and another node requesting the same file lock will update its view of the file before taking the lock.

In summary, instead of hard-coding which two of the three properties, consistency, availability, or partition tolerance is favored, the best two are selected depending on the needs of the application.

Concurrency

In order to achieve high-performance, on each device, the client software must be designed to be highly parallel and asynchronous. As an implementation technique, an entire actor-based framework for C++ was developed. It allows writing efficient and reliable code, which is asynchronous and parallel yet entirely lock-free. The principle of concurrency is applied throughout the system from concurrent IOs on the front end, all the way to the network connections on the backend where multiple parallel streams are used to communicate with the object store. Parallel TCP streams increase the IO queue depth to account for the following two performance factors:

- Networks with a large bandwidth-delay product
- Bandwidth-optimized object stores

Multiple HTTPS connections to the object store are opened and then GET/PUT requests are performed in parallel across these multiple connections. Conceptually there is a single object store, but in reality, these multiple connections likely end up connecting to different servers/load balancers comprising the cloud provider infrastructure. In the case of a local on-premise object store, they might be different nodes forming a cluster.

As stated earlier, files are split up into multiple segments (blocks), and each segment is a separate S3 object. When the user reads from a file for instance, the prefetcher reads simultaneously numerous parts of the file by performing GETs on various segments using the opened connections. When writing, again, many objects are simultaneously uploaded across these multiple connections.

The default maximum is 64 back-end IOs per client device, but as the network conditions change it's necessary to adjust the IO queue depth. For that reason, there is a flow control mechanism that dynamically regulates the number of back-end IOs sent to the object store.

Loose Coupling

In a highly dynamic, decentralized environment with many devices constantly in flux, it is challenging to elicit any global state. Therefore, LucidLink adopted a general design principle that every node should aim to make its own local decisions with as little direct knowledge of other nodes as possible, while not directly altering or interfering with the state and behavior of others. In broad terms, this is the principle of loose coupling. It is evident in several design choices, from the eventual consistency model to pull-based metadata reconciliation and data replication, to flexible communication schemas.

State-Based Synchronization

Distributed systems could be also classified as state-based or log-based depending on what information they exchange in their pursuit of equilibrium. Log-based systems exchange actions instead of static state and are simple to understand but suffer from some significant deficiencies, i.e., they are chatty and eventually lead to the difficult problem of distributed garbage collection. In systems with high rates of change, it is preferable to seek reconciliation based on a state, which is more frugal in terms of bandwidth.

Data Model

As stated earlier, the file system is decoupled into separate metadata and data layers:

- A distributed key-value store for the file system metadata
- Object storage for file replicas

The metadata layer contains the file system specific information (file system hierarchy, file and folder names, attributes, permissions, etc.) while the data layer hosts file replicas. This separation allows them to reside separately and facilitates their scaling and optimization. It also gives rise to different data hosting models, e.g., local, cloud, or hybrid. In the context of file systems, the metadata is a relatively small percentage of the overall data volume so the initial implementation performs full replication across all nodes. Studies show that actual data transfer requests (reads and writes) account for a relatively small percentage of all file IO operations (file stat, directory reads, open, close, etc.). As a result, the overall system responsiveness increases significantly when all metadata related operations are serviced locally.

Representing the file system metadata on top of a generic key-value store provides an additional degree of flexibility. First, we can store user metadata like tags or system metadata like data placement and snapshot policies, etc. More broadly we can model a file system that is not only based on a hierarchical namespace but other namespaces as well, for instance a tag-based semantic file system. Second, breaking down logical objects, e.g., files, into their constituent attributes as key-value pairs and reconciling them independently, allows us to resolve non-semantic conflicts automatically, e.g., one node performs a file rename while another concurrently updates the same file.

While it might seem counterintuitive at first, object storage represents an ideal backend for a distributed cloud-based file system. The S3 interface is straightforward and utilizes HTTPS connections, which are typically allowed through corporate firewalls, making it ubiquitously accessible. Also, the elastic, durable, and cost-effective properties of modern object stores are directly transferrable to the file system built on top. Commercial cloud storage offers a very high degree of 11 nines (99.999999999%) of durability and beyond. This is usually much higher than traditional storage solutions. Elasticity allows the file system to expand as needed, which eliminates the need for costly and disruptive storage upgrades. Finally, the object stores are generally built as scale-out systems that do not funnel all traffic through a single choke point, unlike regular file servers and NAS devices. In our architecture, where each node directly communicates with the object store, the aggregate performance of the entire file system thus scales linearly with the performance of the underlying object store.

Distributed Key-Value Store

Each node maintains its own view of the key-value store and continuously updates it with the hub service to bring its state up-to-date. To update the metadata, we employ version vectors, specifically version vector pairs to reconcile differences. The scheme is very efficient in terms of network bandwidth consumption with a typical overhead of only two version vectors for each unidirectional synchronization (which is further optimized). It also allows the implementation of a disconnected mode of operation where changes accumulated while offline can be efficiently disributed to all nodes.

Log-Structured Data Layout

Each file is split into smaller segments of equal size. Each of them represents an atomic unit with respect to data operations. The segment size is a configurable parameter at file system initialization time. In our design, each segment is represented by a single object in the object store.

File systems can be roughly classified into in-place vs. log-structured (out-of-place) based on their data placement policy. In-place updates will always write the same chunk to the same location. Log-structured systems, in contrast, write every subsequent update of the same file chunk to a different location. On local storage, which has a finite capacity, it leads to the difficult problem of scrubbing the old data. In turn, this involves moving data around the disk, leading to a significant IO amplification. Despite that, there are successful commercial implementations inspired by the log-structured design, specifically copy-on-write COW file systems like btfs, zfs and NetApp's WAFL. Even the firmware of solid state devices (SSDs) use it internally for wear-leveling.

LucidLink utilizes a log-structured file layout in which each new chunk update is stored as a unique object. It's interesting to observe that object storage is a particularly good fit for such a model because the two main performance problems plaguing local implementations, i.e., slow random reads and IO amplification due to garbage collection aren't applicable.

Our log-structured design achieves two objectives:

- Overcomes the limitations of the eventual consistency model of some object stores like AWS S3
- Supports file history/snapshots with minimal performance impact

Garbage collection is still necessary and is performed by the hub. In cloud-based deployments where the object store size is elastic, garbage collection isn't performance critical as it doesn't need to keep up with normal front-end writes.

Data Integrity

One of the primary objectives of a modern file system is to offer protection against data degradation. All data blocks are encrypted using a form of authenticated encryption. Besides privacy, this form of encryption also ensures authenticity. When data is read, the authentication tag is decrypted and verified before data is returned to the application. If the verification fails, then the read request also fails. This guarantees that all data being read is indeed accurate. Any loss of data integrity, whether malicious or inadvertent, e.g., bit rot or logical errors will be immediately detected upon access.

Snapshots

Snapshots preserve the state of the entire filespace. This allows users to view and even restore the whole filespace at a particular point in time. The initial implementation supports read-only snapshots, but the model can be extended to writable ones as well. Snapshots are managed by the administrators and can be created either manually or through customizable policies.

Creating a snapshot does not impact performance. The client simply instructs the hub to mark the time point. The hub will then use a copy-on-write technique to preserve all key-value pairs that are part of a snapshot. Snapshots are implemented at the key-value store level, so everything stored in them can be kept in its entirety, e.g., the entire file system with all its system and custom metadata. Due to the log-structured design, the data is also written using copy-on-write, offering native support. Garbage collection takes into account all snapshots to preserve the data at the correct time-points. Finally, deleting a snapshot makes all newly unreferenced data segments eligible for collection.

System Decomposition

LucidLink mounts as a file system on each client host. Applications accessing this mount point send calls to the operating system, which redirects them to the client software (see Figure 3). Metadata requests are typically serviced immediately by executing transactions against the local metadata database. Data requests are split into segments that are processed through the pipeline described below.

The software is based on a modular design, allowing various components to be turned on and off. This includes compression, encryption, caching, and prefetching, among others. Different configuration settings can also be fine-tuned during file system initialization or at runtime, e.g., file segment size, network settings, etc. All this gives customers a maximum degree of flexibility for different production environments.

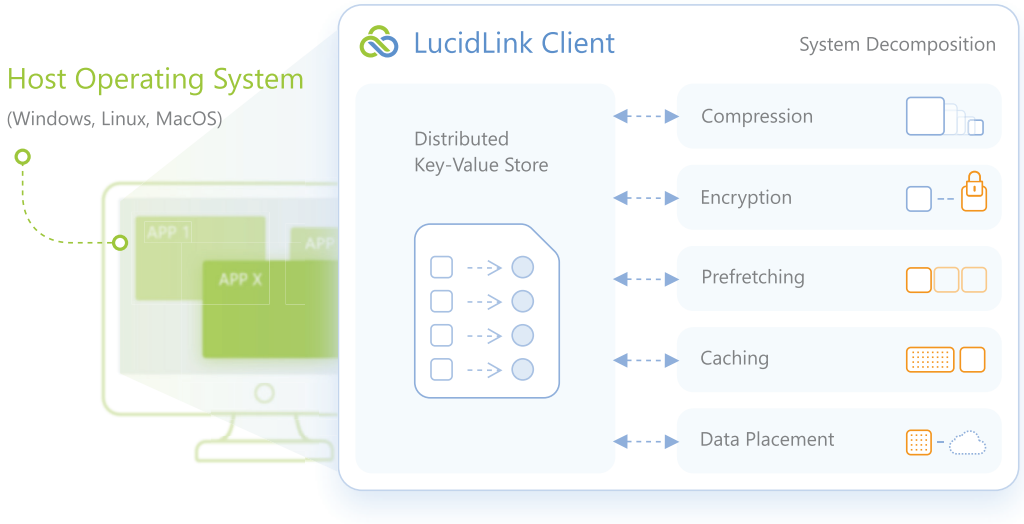


Figure 3

Compression

The first stage in the pipeline attempts to compress each block on the fly. LZ4, a light-weight compression algorithm, optimized for speed is used by default. It can be fine-tuned further with various compression levels trading performance for increased compression ratio. Depending on the data content, compression can significantly reduce the network traffic, improving the overall system performance while reducing the storage footprint.

Encryption

Each data block being updated is encrypted with its corresponding file encryption key, using AES-256 GCM and a unique initialization vector. On retrieval, the block is decrypted before being passed back to the requesting application. Note that the encryption takes place before caching, which means that all data cached on the local disk is already encrypted. This has an important implication, namely as soon as the user logs out, the residual cached blocks become inaccessible.

Prefetching

Prefetching, a.k.a. read-ahead, means reading more blocks than requested by the application in order to warm up the cache. This way future reads can be serviced directly without incurring any network traffic with the associated response latency. The algorithm looks at the IO flow to determine patterns in the read requests. It distinguishes between two types of IO flows, reads within the same file and within the same directory. Multiple streams are identified and analyzed to improve the prefetch accuracy. For instance, if an application issues multiple streams of sequential or semi-sequential reads within the same file or multiple files, each stream will trigger prefetching. The prefetcher eliminates cache misses, which can dramatically improve the user experience. Future work will incorporate machine learning to further enhance its predictive capabilities.

Caching

Persistent write-back caching is employed and all cached data is stored on the local disk. The write requests are acknowledged as soon as the data is safely written to the disk cache. From there, it's pushed to the object store asynchronously, and in parallel. As each block is uploaded, the metadata is updated, in lockstep so all other clients can see the appropriate state as soon as the data is available to them. Employing write-back caching significantly improves write performance in two ways. First, the speed of writes becomes mostly a function of the local disk rather than the network speed. Second, frequent writes to the same file might get absorbed before being uploaded, resulting in less extraneous network traffic.

The cache expands on demand up to a predefined and configurable limit. Its size can vary from a few GBs on smaller devices like laptops up to multiple TBs on servers. Once the cache fills up with dirty data, all new writes then get processed at the speed at which old ones are flushed to the cloud.

Data Placement

Data blocks are saved as individual S3 objects. Based on the log-structured model, the data layout describing which objects form which files, is stored alongside the data content within the objects themselves. The data layout for each file is a type of B-tree, which has a large number of children at each level, keeping the tree depth very shallow. Each leaf points to the physical object where the file segment is stored. Updating a file segment leads to the creation of a new branch from the root to the leaf, which is stored with the data for the update itself.

The part of the metadata that represents the data layout is thus not part of the regular metadata database that all nodes synchronize. As a file is updated, only a so-called root pointer is stored in the database, which points to the most current layout stored in the object store. Essentially, the size of each file does not affect the size of the metadata database. This allows the file system to scale into the PB range.

Summary

Cloud object storage has fundamentally changed how enterprises store and access data. Despite these advances, modern workflows require a different approach, one that allows seamless and instant access to data from geo-distributed locations.

LucidLink has reimagined the file system built for the cloud to provide a truly innovative alternative to the prevailing architectures of today. It is engineered for enterprises seeking a scalable, reliable file service with best-in-class security and zero operational overhead. It revolutionizes the use of object storage for modern cloud-computing environments by transforming the cloud into a local storage tier.

— *George Dochev, CTO*