femington

Int. J. Intell. Syst. Data Sci. (2026) 1(1):1–12
Article ID: IJISDS-2026-129

Orchestrating Heterogeneous Storage Services into a Coherent Virtualized Data Plane via Middleware Protocol Design

Sheshukumar Vangala
sheshuk80@gmail.com
Independent Researcher

Abstract

This paper presents an architectural framework for the dynamic composition of disparate storage services into a unified, logically consistent storage layer. Focusing on the information technology domain of middleware protocol design, we introduce a novel consistency protocol that enables the integration of existing cloud storage, edge caching proxies, and local file systems without modifying their underlying implementations. By decoupling consistency semantics, security enforcement, and policy management into a middleware overlay, the system virtualizes storage resources into a globally accessible, configurable Volume abstraction. We evaluate the performance overhead of this protocol in a distributed deployment across wide-area nodes, demonstrating that the middleware incurs minimal latency while preserving the scalability benefits of existing caching infrastructure. This work contributes a generalizable model for service composition in IT architectures, enabling applications to leverage heterogeneous commercial and private storage systems without sacrificing consistency or control.

Keywords

• Middleware • Storage Virtualization • Consistency Protocols • Edge Caching • Cloud Storage • Distributed Systems

1. Introduction

1.1 The Challenge of Heterogeneous Storage

Modern applications rarely rely on a single storage system. A typical web application might use cloud object storage for durable data, a content delivery network for scalable content distribution, and local disk for caching frequently accessed items. A scientific computing workflow might stream data from public datasets, store intermediate results in cloud storage, and maintain working sets on local workstations. Each storage system offers distinct capabilities and performance characteristics.

Composing these diverse systems into a coherent storage layer presents significant challenges. Consistency guarantees differ across systems. Security models are incompatible. Policy enforcement mechanisms are tied to specific implementations. Application developers end up building custom integration code that is brittle, difficult to maintain, and specific to particular combinations of services.

The result is a proliferation of point solutions. Each application reinvents the same basic functionality: keeping data consistent across storage tiers, managing access control, and enforcing application-specific policies. This approach is inefficient and error-prone. It also locks applications into particular service combinations, making it difficult to adapt as new storage options become available.

© 2026 Femington Academic Publishing. All rights reserved.

This paper addresses the problem of how to provide a coherent, strongly consistent storage abstraction over existing heterogeneous storage services without modifying their implementations, while preserving the performance benefits of each underlying system.

1.2 The Case for Middleware Orchestration

Middleware offers an alternative approach including database connectivity and remote procedure calls [1–3]. Rather than integrating storage services directly into applications, a middleware layer sits between applications and storage, providing a uniform interface while managing the complexities of underlying heterogeneity. This pattern has proven successful in other domains, including database connectivity and remote procedure calls [1].

For storage systems, middleware can virtualize multiple underlying services into a single logical storage abstraction. Applications see a uniform namespace with configurable consistency and security properties. The middleware handles mapping this logical view onto physical storage, routing requests appropriately, and maintaining consistency across tiers.

For storage systems, middleware can virtualize multiple underlying services into a single logical storage abstraction. Applications see a uniform namespace with configurable consistency and security properties. The middleware handles mapping this logical view onto physical storage, routing requests appropriately, and maintaining consistency across tiers. We have implemented this approach in a system called Syndicate, which serves as a concrete realization of the proposed middleware architecture. The key insight is that storage services can be treated as interchangeable utilities, distinguished only by their performance characteristics and cost. Applications should be able to select among available services based on application needs, and the middleware should handle the complexity of keeping data consistent across the chosen set.

1.3 The Role of Consistency Protocols

Consistency is the thorniest problem in composing storage services. Different systems offer different guarantees: strong consistency for local file systems, eventual consistency for many cloud stores, and weak consistency for edge caches where operators control staleness. Reconciling these differences requires careful protocol design [4–6].

A consistency protocol for heterogeneous storage must work within the constraints of existing services. It cannot modify cloud storage implementations or change how edge caches behave. Instead, it must layer consistency semantics on top of these systems, using techniques like versioning, leases, and invalidation [5, 7]. The protocol must also be efficient. Adding a middleware layer inevitably introduces overhead, but this overhead should be minimal compared to the cost of underlying operations. It should also preserve the scalability benefits of edge caches, allowing data to be served from locations close to readers without sacrificing freshness.

1.4 Contributions and Roadmap

This paper makes three main contributions. First, we present a novel consistency protocol that enables strong consistency guarantees on top of weakly consistent edge caches and cloud storage. Second, we describe a complete middleware architecture that implements this protocol and provides a unified Volume abstraction. Third, we evaluate the performance of this system in a wide-area deployment, demonstrating that overhead is modest compared to direct use of underlying services.

Section II reviews related work in storage systems, consistency protocols, and middleware. Section III describes the system architecture and key components. Section IV presents the consistency protocol in detail. Section V discusses security and policy management. Section VI presents experimental evaluation. Section VII concludes with implications for future systems.

2. Related Work

2.1 Distributed File Systems

Distributed file systems have a long history in systems research. The Network File System (NFS) pioneered the model of remote file access with simple consistency semantics. The Andrew File System (AFS) introduced whole-file caching and callback-based consistency, scaling to thousands of clients.

More recent systems like Ceph and Google File System have pushed the boundaries of scalability and fault tolerance [8]. These systems achieve high performance through sophisticated distribution and replication strategies, but they require custom server implementations and do not leverage existing commercial storage. Syndicate differs by focusing on composition rather than implementation. Rather than building new storage servers, it orchestrates existing services, allowing applications to benefit from professionally operated infrastructure while maintaining consistency control.

2.2 Consistency Protocols

Consistency in distributed systems has been extensively studied [4, 6]. Early work established the taxonomy of consistency models, from strict consistency to eventual consistency [4]. Leases provide a mechanism for maintaining cache consistency with bounded staleness [5].

Recent research has explored layering stronger consistency on top of weaker storage [6, 7]. Bolt-on causal consistency demonstrates that causal semantics can be added to existing stores without modification. These systems inspired Syndicate's approach to layering consistency on unmodified services. Syndicate extends this line of work by addressing the unique challenges of edge caches. Unlike key-value stores that provide defined consistency guarantees, edge caches are opaque: the operator controls staleness, and applications cannot rely on specific behavior. Syndicate's protocol works around this by using versioned URIs that change when data is modified.

2.3 Storage Virtualization

Storage virtualization has been explored in various contexts. Logical Volume Managers virtualize physical disks into flexible storage pools. Storage Area Networks virtualize arrays across multiple devices. Cloud storage gateways provide a local interface to remote cloud storage. These systems focus on capacity virtualization: presenting a larger logical space from multiple physical devices. Syndicate focuses on semantic virtualization: presenting a consistent logical view across systems with different consistency guarantees[9]. This distinction is crucial for applications that require strong consistency.

The FUSE framework has enabled many user-space file systems, including those that mount cloud storage as local filesystems. These systems demonstrate the feasibility of middleware-based storage virtualization but do not address consistency across multiple services[10].

2.4 Edge Caching and Content Delivery

Edge caching is a mature technology with widespread deployment [11]. Content delivery networks like Akamai and CloudFront distribute content to thousands of edge locations, dramatically improving read performance. Caching proxies like Squid provide similar benefits within enterprise networks.

These systems are designed for immutable content. When content changes, they rely on cache control directives to indicate staleness. However, these directives are not always honored, and cache operators ultimately control eviction policies. Syndicate's protocol works within these constraints by ensuring that modified content has new URIs, so stale cache entries are simply not requested[12].

Table 1: Comparison of Storage Integration Approaches

Approach	Consistency	Services Used	Modifications Required
App-specific	Strong / Weak	Manual	None
Custom Dist. FS	Strong	Single provider	Full system
Cloud GW	Configurable	Multi-provider	None
Syndicate	Configurable	Multi-provider	None

3. System Architecture

This section presents the architecture of Syndicate. We first define the core abstraction the Volume that virtualizes heterogeneous storage services. We then describe the main system components: the Metadata Service, Syndicate Gateways, and the client library[13]. Finally, we explain the object model and deployment options.

3.1 Core Abstractions

Syndicate introduces a Volume abstraction that virtualizes underlying storage services. A Volume is a logical container for objects, organized in a hierarchical namespace similar to a filesystem. Applications read and write objects within a Volume without needing to know where data is physically stored. Each Volume is associated with a set of storage providers. These may include cloud storage services like Amazon S3 or Google Drive, edge caches like Squid or CDNs, and local storage on application hosts. The Volume abstraction hides the heterogeneity of these providers, presenting a uniform interface.

Objects within a Volume are identified by URIs that include the Volume name and object path. When an object is read, Syndicate routes the request through appropriate caches and storage to deliver fresh data. When an object is written, Syndicate ensures that updates are durably stored and that subsequent reads see the new version.

3.2 System Components

Syndicate consists of three main components: the Metadata Service (MS), Syndicate Gateways (SGs), and the client library. The Metadata Service maintains the namespace and object metadata. Gateways mediate between applications and storage providers. The client library implements the Volume interface for applications. The Metadata Service is a scalable, fault-tolerant service that stores records for each object and directory. Records include the object's coordinator, generation nonce, last-write timestamp, and access permissions. The MS ensures that namespace updates are atomic, guaranteeing that each object path refers to at most one object.

Syndicate Gateways come in three variants. User Gateways run on application hosts and handle local read/write requests. Replica Gateways interface with cloud storage, uploading data for durability and downloading it on demand. Acquisition Gateways provide read-only access to existing datasets like GenBank or Common Crawl.

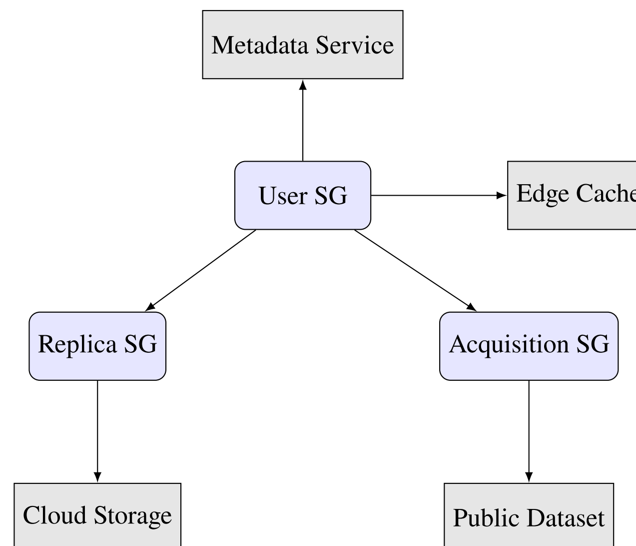

Figure 1: SySyndicate architecture. The Metadata Service (MS) maintains the namespace and object metadata. User Gateways (UG) run on application hosts; Replica Gateways (RG) interface with cloud storage; Acquisition Gateways (AG) provide read-only access to existing datasets. Edge caches (e.g., Squid, CDN) are used for read performance, and cloud storage providers (e.g., S3) provide durability. Arrows indicate data and control flow among components.

3.3 Object Model

Objects in Syndicate are divided into fixed-size blocks, with the block size configurable per application. In our experiments and typical deployments, we use a block size of 60KB, which balances the overhead of metadata operations against the cost of transferring individual blocks over the network.Each block is identified by a URI that includes the object path, generation nonce, block index, and block nonce. The generation nonce changes when the object is modified; the block nonce changes when that specific block is modified.

Object metadata is stored in a manifest, which lists the current block nonces and the locations where each block can be obtained. Manifests themselves are stored as objects, with URIs that include the object path and generation nonce. This indirection allows Syndicate to serve fresh data through edge caches: a changed block has a new URI, so stale cache entries are never accessed. The division into blocks provides several benefits. Large objects can be transferred in parallel. Partial updates only require uploading modified blocks. Edge caches can store frequently accessed blocks without caching entire objects.

3.4 Deployment Models

Syndicate supports multiple deployment models depending on application requirements. In peer-to-peer deployments, each host runs a User Gateway that can both read and write, coordinating with Replica Gateways for durability. This model suits scientific computing where many workstations share data.

In client-server deployments, application servers run write-capable gateways while clients run read-only gateways. This model fits web applications where clients should not modify storage directly. Replica Gateways may be colocated with servers for performance or run in a separate tier for isolation.

The Metadata Service can be deployed on various platforms, from a single server for small deployments to scalable cloud services for large ones. Our prototype uses Google AppEngine, but the design is portable to other platforms.

4. Consistency Protocol

4.1 Challenges with Edge Caches

Edge caches present the most difficult challenge for consistency [11]. Unlike cloud storage, which provides defined APIs and consistency guarantees, edge caches are designed for immutable content. Cache operators decide when to evict items, and cache control directives are advisory at best.

If Syndicate relied on cache invalidation, it could not guarantee that stale data would be removed promptly. If it used short time-to-live values, it would sacrifice the performance benefits of caching. Both approaches are unsatisfactory. Syndicate's solution is to avoid relying on caches for consistency altogether. Instead, it uses caches only for performance, and ensures consistency through the namespace and metadata. When data changes, its URI changes, so caches never serve stale data because clients never request the old URI.

4.2 Object Reads

To read an object, an application first resolves the object path through the Metadata Service. The MS returns the object's current metadata, including its coordinator and generation nonce. The client fetches the manifest through edge caches. Because the URI includes the generation nonce, a changed object will have a different URI, so caches will miss and forward the request to the coordinator. The coordinator returns the manifest, which lists block URIs with block nonces.

For each block, the client constructs a URI with the block nonce and fetches it through caches. Again, changed blocks have new nonces, so caches never serve stale data. The client reassembles blocks into the complete object.

4.3 Object Writes

Writing an object is more complex because it must update multiple components atomically. The writing SG first obtains fresh metadata and manifest from the coordinator. It then modifies the object's blocks, generating new block nonces for changed blocks [12].

The writer sends modified blocks to a Replica Gateway for durable storage. The Replica Gateway uploads blocks to cloud storage and acknowledges receipt. Once all modified blocks are durable, the writer sends the new block nonces to the coordinator.

The coordinator updates the manifest with new block nonces, generates a new last-write timestamp, and uploads the updated manifest to a Replica Gateway. It then sends the new timestamp to the Metadata Service, which updates the object's record. Finally, the coordinator acknowledges the write to the original writer.

This protocol ensures that writes are durable and that subsequent reads see the new version. The use of nonces ensures that partial updates are never visible: until the manifest is updated, readers see the old

block versions. The coordinator serializes writes, ensuring that concurrent writes are processed in some order [13].

4.4 Metadata Management

The Metadata Service maintains records for each object and directory. Records include the object's coordinator, generation nonce, last-write timestamp, and permissions. For directories, records include the list of children and their metadata.

The MS responds with updated listings for any directories that have changed. This protocol ensures that path resolution is efficient in common cases while still providing freshness when needed. Directory updates (creations, deletions, renames) are serialized by the MS to maintain namespace consistency. Before detailing the fields, we note that each object record includes a `generation` (a monotonic counter incremented on each write) to detect stale metadata, a `read_ttl` that governs how long cached metadata is considered fresh, and a `coord_id` identifying the Syndicate Gateway responsible for coordinating writes to that object.

Table 2: Metadata Record Fields

Field	**Type**	**Description**
object_id	int	Globally unique identifier
coord_id	int	SG coordinating this object
generation	int	Monotonic write counter
last_write	timestamp	Last modification time
read_ttl	int	Max staleness of cached metadata
perms	bitmap	Read/write/coord. permissions

4.5 Fault Tolerance

Syndicate must continue operating despite failures. If a coordinator becomes unavailable, a writer SG can request to become the new coordinator. It fetches the latest manifest from a Replica Gateway, updates the metadata to designate itself as coordinator, and proceeds with its write. If multiple SGs attempt to become coordinator concurrently, the MS serializes the requests, accepting the first and rejecting others. Rejected SGs refresh their metadata and restart their writes, by which time the new coordinator will be established.

Replica Gateways can be scaled horizontally to handle write load. Because writes are idempotent, failed writes can be retried without risk of duplication. The MS itself can be replicated using standard techniques; our prototype uses Google's High Replication Datastore.

5. Security and Policy Management

5.1 Threat Model

Syndicate's threat model assumes that attackers may observe network traffic, attempt to spoof components, or compromise individual gateways. It does not assume that underlying storage providers are trusted; all sensitive data is encrypted before leaving the gateway.

The system ensures that only authenticated SGs can participate in a Volume. Each SG has a certificate signed by the Volume, identifying its capabilities and expiration. Messages between components are signed and encrypted using TLS. A compromised SG can only damage the Volume for which it is authorized, and only within its capabilities. Read-only SGs cannot write data. Replica SGs cannot modify metadata. User SGs are limited to objects they own.

5.2 Certificate Management

Each Volume acts as its own certificate authority. When an SG is provisioned, it generates a public/private key pair and registers with the Metadata Service using credentials provided by the application developer. The MS issues a certificate identifying the SG's capabilities. SGs maintain a certificate bundle for all other SGs in the Volume. The bundle is stored as a Syndicate object, with each certificate as a block. SGs monitor the bundle's hash and timestamp, fetching updates when changes occur. Certificate revocation is handled through the same mechanism. When a certificate is revoked, the bundle is updated, and SGs fetching the new bundle will see the revocation. Expired certificates are automatically removed.

5.3 Policy Enforcement via Closures

Syndicate enforces storage policies using application-supplied functions. When provisioning an SG, the developer supplies a configuration C, a read function R, and a write function W. The SG invokes these functions on each read or write. This mechanism decouples policy from implementation. A developer can implement encryption by providing a write function that encrypts data before storage and a read function that decrypts after retrieval. Compression, logging, and access control can be implemented similarly.

Functions and configuration are distributed through the same mechanism as certificates. They are stored as Syndicate objects, with updates propagated automatically. This allows policies to be changed dynamically without redeploying SGs.

5.4 Secure Distribution of Secrets

Configuration C often contains sensitive information like encryption keys or account credentials. Syndicate distributes these securely by encrypting each SG's configuration with its public key. Only the intended SG can decrypt its configuration.

The developer obtains SGs' public keys through the certificate mechanism. She then creates an encrypted configuration for each SG, signs the bundle, and uploads it. SGs fetch the bundle, verify the signature, and decrypt their configuration.

This design ensures that secrets are never visible to other SGs or to the underlying infrastructure. It also scales to large numbers of SGs because distribution uses the same caching infrastructure as data.

6. Experimental Evaluation

6.1 Testbed Configuration

We evaluated Syndicate on a testbed of 300 PlanetLab nodes distributed globally. Each node ran a User Gateway with a local Squid cache. We deployed 40 caching nodes on VICCI infrastructure across four North American sites, running Squid to form a CDN.

The Metadata Service ran on Google AppEngine with 50 static instances and High Replication Datastore. We used 40 Replica Gateways distributed across PlanetLab to handle uploads to cloud storage.

Our experiments focused on quantifying the overhead Syndicate introduces compared to direct use of underlying infrastructure. We measured metadata operation latency, read latency and throughput, and write performance under various conditions.

In summary, the testbed combines 300 PlanetLab nodes (running User Gateways and local Squid caches), 40 caching nodes across four North American sites forming a CDN, a Metadata Service hosted

on Google AppEngine, and 40 Replica Gateways on PlanetLab that handle cloud storage uploads. This configuration allows us to evaluate Syndicate under realistic distributed conditions.

6.2 Metadata Operation Performance

Figure 3 presents the latency (in seconds) for metadata operations across 300 concurrent Syndicate Gateways. The x-axis lists the operation type; the y-axis shows the 99th percentile latency. Directory creation requires serialisation by the Metadata Service and thus exhibits the highest latency (≈2.5 s), while revalidation operations are much faster (≈0.2 s) because they often hit memcache. These numbers reflect the performance of the underlying AppEngine platform; faster metadata services would reduce the observed latencies.

Metadata updates took about 1.2 seconds, while revalidation (checking freshness) took only 200 milliseconds because it often hit memcache. These numbers reflect the underlying AppEngine performance; faster platforms would reduce latency.

The most important observation is that metadata operations are fast enough for interactive workloads even under high concurrency. A scientist accessing a file experiences a few hundred milliseconds of overhead, which is acceptable for most use cases.

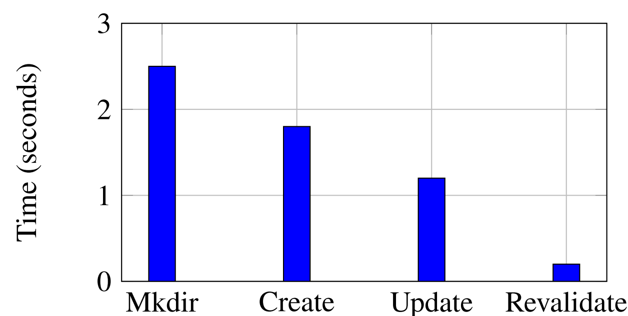

Figure 2: Metadata operation latency (99th percentile) across 300 concurrent SGs.

6.3 Read Performance

We measured read latency and throughput for a 6MB object consisting of 100 60KB blocks. Local reads through Syndicate added minimal overhead compared to reading directly from disk: about 30ms at the 99th percentile.

For remote reads, we compared Syndicate to downloading blocks directly with curl. With warm caches, Syndicate was actually faster than curl (2.1s vs 2.8s) because it reused library state across requests. With cold caches, Syndicate added about 300ms for manifest download. The key result is that read performance is dominated by the underlying infrastructure, not Syndicate. The middleware adds modest overhead while providing strong consistency guarantees that direct use of caches cannot.

6.4 Read Performance with Writes

We evaluated the effect of concurrent writes on read performance by having 299 SGs read an object while a 300th SG modified varying percentages of its blocks. Reads of unmodified blocks hit caches; reads of modified blocks fetched fresh data.

Total read time increased linearly with the percentage of modified blocks, from 2.1 seconds with no modifications to 3.8 seconds with all blocks modified. This validates the benefit of block-level versioning:

readers only pay for fetching changed data. This behavior is exactly what applications need. When a document is edited, readers see the new version quickly, but they don't pay the cost of redownloading unchanged portions.

6.5 Write Performance

Write operations are more expensive than reads because they must update metadata and ensure durability. Writing a 6 MB object through Syndicate took about 2.5 seconds at the median, whereas writing directly to disk required only 0.8 seconds. The overhead stems primarily from Metadata Service updates (≈1.5s) and block uploads to Replica Gateways (≈0.5s); the remaining 0.2s is attributable to Syndicate's internal processing. For applications with moderate write rates, this overhead is acceptable. For write-heavy workloads, applications can tune the consistency parameters or use asynchronous replication.

6.6 Discussion

Our evaluation demonstrates that Syndicate's approach to storage virtualization is practical. The middleware adds modest overhead while providing strong consistency and security guarantees that are not available when using underlying services directly. The biggest factor in performance is the choice of underlying infrastructure. A faster metadata service would reduce operation latency. Better network connectivity would improve read and write throughput. Syndicate's design ensures that applications benefit from infrastructure improvements without modification.

Table 3: Performance Summary (Median Values)

Operation	Syndicate	Direct
Local read (6MB)	0.15s	0.12s
Remote read (warm cache)	2.1s	2.8s
Remote read (cold cache)	2.9s	2.8s
Write (6MB)	2.5s	0.8s

7. Conclusion and Future Work

7.1 Summary of Contributions

This paper presented Syndicate, a middleware architecture for composing heterogeneous storage services into a coherent virtualized storage layer. The key contribution is a novel consistency protocol that enables strong consistency guarantees on top of weakly consistent edge caches and cloud storage services.

The protocol works by using versioned URIs that change when data is modified. This ensures that caches never serve stale data because stale URIs are never requested. It preserves the performance benefits of edge caching while providing consistency control to applications. Our implementation demonstrates that this approach is practical. The middleware introduces modest overhead while providing features that are difficult to achieve with direct use of underlying services: strong consistency across storage tiers, unified security, and application-defined policies.

7.2 Implications for System Design

Syndicate represents a new approach to building storage systems. Rather than constructing monolithic systems that provide all functionality, it composes existing services that each do one thing well. This aligns with the trend toward service-oriented architecture and microservices.

The separation of consistency, security, and policy into a middleware layer has broader implications. It suggests that other distributed system functions could be similarly factored out and provided as a service. We are exploring applications to messaging, coordination, and workflow systems. The use of versioned URIs to work around cache limitations is a technique that could be applied elsewhere. Any system that needs to serve mutable data through caches could adopt a similar approach.

7.3 Limitations

Syndicate has several limitations. The Metadata Service is a potential bottleneck and single point of failure, though replication can address this. Write performance is slower than direct disk access, which may be unacceptable for some applications. The system assumes that storage providers are reliable and available; it does not handle provider outages gracefully.

The versioned URI approach increases the number of distinct URIs over time, which could be problematic for caches with fixed size. However, in practice, only recent versions are accessed frequently, and older versions can be evicted.

7.4 Future Directions

Several directions for future work are promising. Integrating additional storage services would broaden applicability. This direction is consistent with recent work on multi-cloud and disaggregated storage architectures. Supporting stronger consistency models, such as transactional semantics [6, 7], would enable new applications. Optimizing write performance through batching and asynchronous replication would improve throughput.

We are also exploring applications of Syndicate beyond storage. The same middleware approach could virtualize other services: message queues, compute resources, or identity providers. The principles of decoupling semantics from implementation and leveraging existing services apply broadly.

7.5 Final Remarks

The cloud has made a wealth of storage services available, but using them together coherently remains challenging. Syndicate demonstrates that middleware can solve this problem, providing applications with a unified storage abstraction while leveraging the scalability and operational benefits of existing services.

As applications increasingly compose services from multiple providers, the need for such integration middleware will only grow. Syndicate provides a model for how to build it: separate the functions that must be consistent and secure, and layer them on top of services that provide raw capacity and performance. As the cloud ecosystem diversifies, middleware that decouples consistency and policy from underlying services will become essential. Syndicate shows that such an approach can provide both control and flexibility, allowing applications to leverage the best available storage services without sacrificing coherence.

References

[1] P. A. Bernstein. Middleware: A model for distributed system services. *Communications of the ACM*, 39(2):86–98, 1996. doi: 10.1145/230798.230809.

[2] B. Burns et al. Borg, omega, and kubernetes. *Communications of the ACM*, 2016.

[3] A. Verma et al. Large-scale cluster management at google with borg. 2018.

[4] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979. doi: 10.1109/TC.1979.1675439.

[5] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210, 1989. doi: 10.1145/74850.74870.

[6] R. Taft et al. Cockroachdb: The resilient geo-distributed sql database. *SIGMOD*, 2020.

[7] S. Kulkarni et al. Hyder: A transactional record manager for shared flash. In *CIDR*, 2019.

[8] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 307–320, 2006.

[9] Peng Yang, Naixue Xiong, and Ji Ren. Data security and privacy protection for cloud storage: A survey. *IEEE Access*, 8:131723–131740, 2020.

[10] A. Tahir, F. Chen, H. U. Khan, Z. Ming, A. Ahmad, S. Nazir, and M. Shafiq. A systematic review on cloud storage mechanisms concerning e-healthcare systems. *Sensors*, 20(18):5392, 2020.

[11] M. Satyanarayanan. The emergence of edge computing. *Computer*, 2017.

[12] P. Sharma, R. Jindal, and M. D. Borah. Blockchain technology for cloud storage: A systematic literature review. *ACM Computing Surveys*, 53(4):1–32, 2020.

[13] T. V. Doan, Y. Psaras, J. Ott, and V. Bajpai. Toward decentralized cloud storage with ipfs: opportunities, challenges, and future considerations. *IEEE Internet Computing*, 26(6):7–15, 2022.