Syndicate: Virtual Cloud Storage through Provider Composition

Jude Nelson
Princeton University
35 Olden Street
Princeton, NJ 08540
jcnelson@cs.princeton.edu

Larry Peterson
Princeton University
35 Olden Street
Princeton, NJ 08540
llp@cs.princeton.edu

ABSTRACT

Syndicate is a storage service that builds a coherent storage abstraction from already-deployed commodity components, including cloud storage, edge caches, and dataset providers. It is unique in that it not only offers consistent semantics across multiple providers, but also offers a flexible programming model to applications so they can define their own provider-agnostic storage functionality. In doing so, Syndicate fully decouples applications from providers, allowing applications to choose them based on how well they enhance data locality and durability, instead of whether or not they provide requisite features.

This paper presents the motivation and design of Syndicate, and gives the results of a preliminary evaluation showing that separating storage functionality from provider implementation is feasible in practice.

Categories and Subject Descriptors

H.3.2 [**Information Storage and Retrieval**]: Information Storage; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*distributed systems*; D.4.3 [**Operating Systems**]: distributed file systems

Keywords

software-defined storage; service composition; storage gateway

1. INTRODUCTION

The cloud is changing how users interact with data. This is true for legacy applications that are migrating on-site data to cloud storage, and for emerging applications that are augmenting cloud storage and dataset repositories with edge caches. In both cases, leveraging multiple cloud storage systems, edge caches, and dataset repositories allows applications to harness their already-deployed infrastructure, instantly gaining a global footprint. However, doing so introduces several storage design challenges relating to functional requirements, data consistency, access control, and fault tolerance. This paper describes Syndicate, a wide-area storage system that addresses them in a coherent manner.

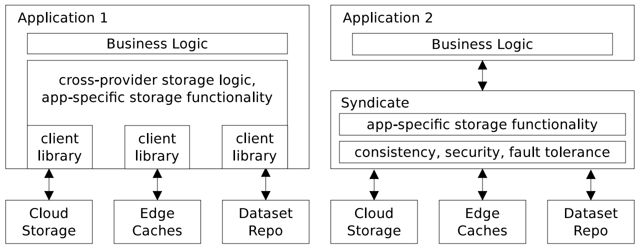


Figure 1: *Application design with and without Syndicate.*

Each type of component system offers well-understood *functional* benefits. Cloud storage offers an "always-on" repository for hosting a scalable amount of data, while keeping it consistent and enforcing access controls. Dataset repositories host and curate a scalable amount of read-only data on behalf of many applications. CDNs, caching Web proxies, and HTTP object caches ("edge caches") help under-provisioned origin servers scale up the volume of requests they can handle. In all cases, instances of these systems (*providers*) make their functional benefits available through instance-specific APIs.

In contrast, the providers' infrastructure offers two key *utility* benefits transparently —data durability and access locality. Cloud storage and dataset providers improve durability by replicating data to geographically distributed datacenters, and edge caching providers improve locality by placing temporary copies of data at sites closer to readers than the origin servers (lowering latency and increasing bandwidth).

Unlike functional benefits, utility benefits can be aggregated. Leveraging multiple providers yields more utility than leveraging any single one, and improvements to one provider improve overall utility. However, doing so is non-trivial because each provider has a different API, with different functional semantics. The developer must design the application around this, thereby coupling its storage logic to provider implementations (Figure 1, left).

Our key insight is that this coupling can be avoided by leveraging providers not for their functional benefits, but for the utility they offer—cloud storage and dataset providers offer durability, and edge caching providers offer locality. While this strategy ultimately makes the developers responsible for storage functionality, doing so lets them implement exactly the functionality they need while aggregating provider utility. Syndicate helps them implement and deploy this functionality at scale, while minimizing provider coupling and addressing common storage concerns on their behalf (Figure 1, right).

The key contribution of Syndicate is wide-area software-defined storage service that runs on top of unmodified providers. It provides an extensible interface for implementing domain-specific storage functionality in a provider-agnostic way, while addressing common cross-provider consistency, security, and fault-tolerance requirements automatically. Using Syndicate lets developers create a storage service for their applications that has the aggregate utility of multiple underlying providers, but without having to build and deploy a whole storage service from the ground up. That is, Syndicate creates virtual cloud storage through provider composition.

2. USAGE SCENARIOS

Syndicate helps applications handle scalable read/write wide-area workloads where cloud storage, edge caches, and external datasets have complementary roles to play. To motivate its design, we explore three application domains that today are dominated by vertically-integrated storage point-solutions, and argue that the ideal storage system in each domain would be flexible enough to integrate a changing set of disparate providers into a coherent whole. In doing so, it could allow applications to leverage providers for their utility benefits, independent of their functional benefits. A concluding subsection summarizes our observations about these examples, which influence our design.

2.1 Scientific Datasets

Tens of thousands of computers work together to process multi-terabyte datasets; genomic sequencing datasets being illustrative examples [16, 23, 1]. Researchers run small experiments in their labs on local workstations and clusters, but run large experiments on powerful cloud computing platforms and scalable grids of computers in the wide-area. At the same time, they share findings with collaborators, archive working datasets in cloud storage, and submit vetted results for integration into the original datasets.

However, each provider has different curation policies, APIs, performance profiles, consistency models, and access controls. Grids and clusters must be specifically programmed to access each, or data must be manually staged for them. This leads to point-solutions and manual procedures for sharing data.

Ideally, researchers would store relevant data on their local workstations for fast access, and seamlessly stream it from dataset providers on an as-needed basis (as in iRODS [27]), regardless of source. The data would be accessible to a scalable number of readers via edge caches (as in CERN VM-FS [8]), provided that readers always receive fresh data. Results from authorized computers would be sanitized and uploaded to researcher-chosen storage (as in Folding@Home [20]) for subsequent curation.

2.2 Collaborative Document Editing

Collaborative document editing systems include cloud-hosted systems like Google Docs [18], version control systems like `git` [40], and industry-specific form-processing systems such as Blackboard [7] and OpenEMR [37]. In all cases, the system lets a set of users read and write documents, subject to common format, consistency, and access control rules.

A key challenge is that users have different usage policies for their data. For example, a doctor editing medical records must keep a verifiable edit history and guarantee confidentiality. As another example, businesses sharing documents through a third party must require edits to be mirrored to their own servers for extra durability.

Ideally, users could bring their own storage system for hosting writes. The storage system would enforce the user's policies whenever another user accesses it through the application. At the same time, the application would allow a scalable number of users to access (fresh) data through edge caches, regardless of the underlying storage.

2.3 Virtual Desktop Infrastructure

An alternative approach to giving employees corporate computers is to let them bring their own devices, and have them run a corporate OS in a VM while they are at work. Employees download their VM images when they begin the day, and periodically save their sessions until they leave. VDI systems exploit the facts that VM images do not change much between sessions [10] and have only one writer (the user), in order to achieve scalable VM deployment through on-site VM block caching.

While some VDI implementations (such as Citrix [11]) use system-specific infrastructure, the ideal scenario is to let the corporation use the cache and storage providers that best meet their business needs, and impose their own user authentication requirements. The VDI infrastructure is not desirable if there already exists proven in-house equivalents that can be used for the purpose.

2.4 Observations

Despite being point-solutions, the real-world examples above address reoccurring storage-level concerns. These include stronger-than-eventual consistency, common fault tolerance strategies (such as automatic fail-over and integrity checks), and common security requirements (such as access control and authentication). This suggests that with the right refactoring, the mechanisms that address these concerns can be implemented once and reused across different application domains.

Our goal is for Syndicate to provide a storage service that addresses these concerns for these and similar applications, while allowing developers to dynamically extend it instead of rewrite it. In doing so, developers could re-use Syndicate as the storage layer to meet basic storage needs, and develop only the extra storage functionality as a code module for Syndicate to load and deploy. This saves the effort of creating multiple point-solutions, and creates an opportunity for re-using specialized storage functionality across Syndicate-powered applications. Once our goal is achieved, developers can use Syndicate to combine providers to gain the utility benefits they offer, and not concern themselves with integrating their functional benefits into the application design.

There are three design challenges to achieving this goal. First, Syndicate must provide storage abstractions that allow the application to interact with data without coupling its design to underlying providers. At the same time, the abstractions should couple Syndicate to providers as loosely as possible.

Second, to facilitate reuse, Syndicate must offer a storage programming model that distinguishes between logic that adds support for additional providers, and logic that adds provider-agnostic storage functionality (such as encryption or access logging) or extends Syndicate's built-in features.

Third, regardless of provider functional benefits and application extensions, Syndicate must always meet consistency, fault-tolerance, and security requirements. If needed, Syndicate must make up for shortcomings in the design of underlying providers. Once addressed, Syndicate can aggregate their utility benefits.

3. DESIGN

The data paths in our examples suggest a strategy to meet our design challenges. There are recurring logical "gateways" that connect the application and underlying components (Figure 2). Written data flows from the application to the cloud storage provider through a storage-facing gateway, external data flows from dataset providers to the application through a dataset-facing gateway, and data flows between application endpoints through cache-facing gateways.

Our strategy is to design these gateways as a wide-area storage service that address common storage concerns, while also allowing for application extensibility. In doing so, we let the developer focus on implementing domain-specific storage functionality, while internally handling consistency, fault-tolerance, and security at scale.

3.1 Abstractions and Components

Syndicate must distinguish between the application's data records in order to apply the domain-specific storage functionality to them. At the same time, it must allow the application to structure and search its records efficiently, in a way that is not coupled to an underlying provider's semantics.

To do so, Syndicate defines three data abstractions: *objects*, *directories*, and *Volumes*. Objects store record data and are similar to files, but contain extra metadata used to programmatically control the storage functionality applied to them. For example, developer-given code can extend Syndicate to impose internal structure on object data. Objects are organized hierarchically by directories, making them efficiently organizable and searchable by applications.

A Volume binds a rooted tree of directories to a set of providers and a set of principals, one of which is the Volume's owner. Within a Volume, an object's data can be distributed across one or more providers, and accessed by one or more principals.

To present these abstractions to applications, Syndicate provides *Syndicate gateways* (SGs), peer processes which coordinate via a scalable cloud-hosted *Metadata Service* (MS). SGs are the first-class analogues of the logical gateways discussed earlier.

The SG comes in three variants, based on how it interfaces with external providers. They all play the same role in the larger Syndicate design, and so we do not distinguish between them unless critical to understanding the system.

User SG: Interfaces with edge caches for an application endpoint. Responds to application read/write requests, read requests from edge caches, and read/write requests from peer SGs. Our prototype offers four instances: a FUSE [15] filesystem, a Web object store, a Python library, and a Hadoop Filesystem [32].

Replica SG: Interfaces with cloud storage providers. Responds to read/write requests from peer User SGs, but does not generate any requests of its own. Acts as an origin server for edge caches for cloud-hosted data. Our prototype currently supports Amazon S3 and Glacier [3], Dropbox [14], Box.net [9], Google Drive [19], and local disk.

Acquisition SG: Interfaces with dataset providers. Maps an existing dataset into a Volume as a read-only directory hierarchy. Acts as an origin server for edge caches on behalf of the dataset provider, and does not generate any local requests. Our prototype currently supports local filesystems and SQL-speaking RDBMSs, and can generate data dynamically using dataset-specific tools.

The MS helps SGs coordinate globally. It maintains the authoritative state of each Volume's metadata, and helps the system scale, tolerate faults, and keep data consistent. The MS binds each SG to one Volume, and helps SGs discover their peer SGs.

In a typical deployment, an application uses one MS, and places data in objects distributed across one or more Volumes. Application endpoints run User SGs locally to access Volume data, and the developer provisions other SGs to attach cloud storage and external datasets. The developer also has options for attaching an existing set of principals to a Volume (Section 3.7), facilitating integration with existing organizations.

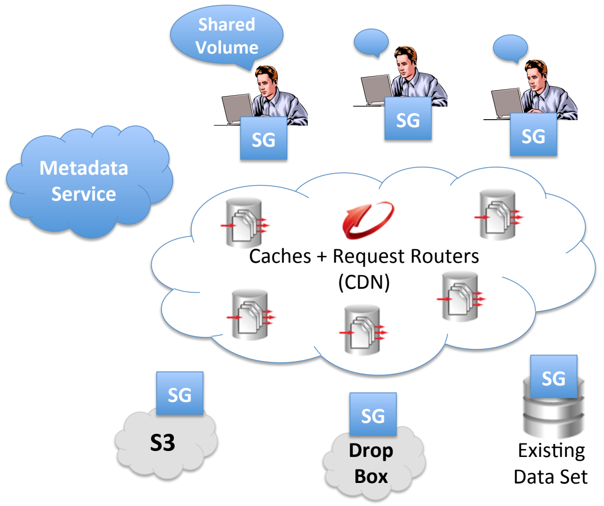


Figure 2: *Logical positioning of Syndicate Gateways (SGs) between edge caches, cloud storage (DropBox, S3), datasets, and application logic.*

3.2 Object Data

The SGs execute an extensible read and write protocol to handle application reads and writes in a way that is not coupled to the underlying providers. We present the protocols here, and discuss their extensibility in Section 3.4.

Each object is assigned an SG that acts as its *coordinator*, which handles reads and writes to it from other SGs and edge caches. Syndicate lets application control this assignment, and control post-read and pre-write processing (Section 3.4).

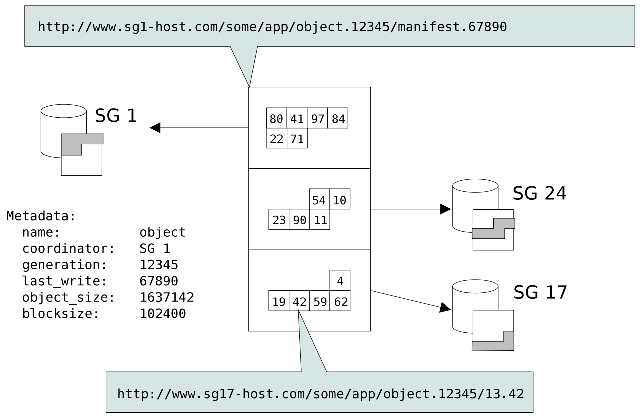
SGs read data from each other using their Volume's edge cache providers. This allows the application to transparently benefit from the data locality they offer, as well as scale up the number of reads it can handle. However, reading fresh data from them is difficult because each cache provider offers different functional semantics regarding cache control and consistency directives. For example, a cache provider might impose a minimum cached object lifetime to prevent clients from accidentally overwhelming an origin server.

Syndicate obviates the need for these directives by ensuring each URI uniquely identifies a snapshot of data. The challenges in doing so are to use cache capacity efficiently (to avoid thrashing), and ensure SGs discover the correct URIs before downloading.

To address the former, SGs serve object data as blocks. This lets edge caches evict data block by block, subject to their own eviction policies. The block size is chosen by the developer based on the application's workloads; Syndicate helps this choice by reporting internal fragmentation rates.

To address the latter challenge, an SG stores a generation nonce and last-modified nonce for each object it coordinates. The generation nonce is assigned on creation, and the last-modified nonce is assigned when the last successful write is processed. For each block in each object, the SG additionally stores a block nonce that identifies its current contents, as well as the set of SGs in the Volume that can serve a replica of it. It keeps track of this information using an object *manifest* (Figure 3), and uses this information to generate block URIs. The MS stores copies of each object's generation and last-modified nonces.

The SG reads an object's data by first obtaining a fresh object manifest, and then obtaining the desired blocks (Figure 4). To get a fresh manifest, the SG first obtains the object's generation and last-


http://www.sg1-host.com/some/app/object.12345/manifest.67890
SG 1
80 41 97 84
22 71
54 10
23 90 11
SG 24
4
19 42 59 62
SG 17
http://www.sg17-host.com/some/app/object.12345/13.42

```
Metadata:
  name:          object
  coordinator:   SG 1
  generation:    12345
  last_write:    67890
  object_size:   1637142
  blocksize:     102400
```

Figure 3: *Logical representation of a manifest for a 16-block object after three writes. Each SG (cylinders) hosts the latest copies of some of the object blocks (grey areas), which the manifest tracks. (Generation nonces, last-modified nonces, and block nonces are kept short in this figure for brevity.)*

modified nonces from the MS. It uses them, along with the object's path, to generate an URI for the manifest. By including them, the URI identifies a manifest snapshot as fresh as the nonces, allowing it to use edge caches to obtain fresh manifest data.

Once obtained, the SG uses the manifest to build URIs for each block. A block URI includes the object's path, generation nonce, the block offset, and the block nonce. This allows it to download block data from edge caches that is guaranteed to be as fresh as the manifest.

The write protocol (Figure 4) is designed to let Syndicate scale write bandwidth and tolerate SG failures while allowing the application to control data formatting and inter-object write ordering. When an SG writes to an object, it first obtains a fresh manifest, generates the new blocks and block nonces, and sends them to a Replica SG. It includes the object ID, generation nonce, and last-modified nonce, so the Replica SG can uniquely identify each block (so it can be garbage-collected when overwritten).

The writer sends the block nonces to the coordinator. Upon receipt, the coordinator generates a last-modified nonce, adds the block nonces to its manifest, and uploads the manifest to a Replica SG. It then sends the object last-modified nonce to the MS, so subsequent readers can discover it. It finally acknowledges the writer SG, and asynchronously garbage-collects overwritten blocks from the Replica SGs. Each User SG logs each step of the write to local stable storage, so it can roll back a write or garbage-collect data if it fails and restarts during the operation.

3.3 Object Metadata

Where data plane operations are concerned, the SGs rely on the MS to announce their presence and the objects they coordinate, to discover other SGs and objects, and to help them download fresh object data from one another.

The MS maintains a metadata record for every object/directory in each Volume, as well as records for each SG. A metadata record is functionally similar to an i-node; a listing of relevant fields the MS tracks can be found in Table 1. SGs download and cache these records whenever they access an object for an application.

The MS serves as coordinator for all directories, in order to serialize object/directory creation and deletion in a single directory. This ensures each path refers to one object, so a read does not combine blocks from different objects.

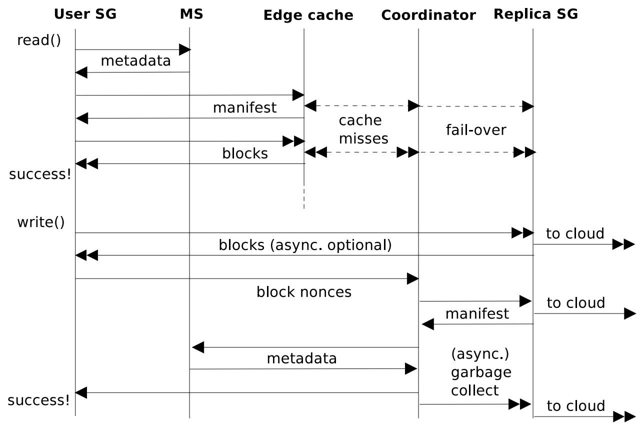


Figure 4: *Protocol overview for reads and writes. Some steps are performed asynchronously, as marked.*

Name	Description
`type`	Object or Directory record.
`name`	Object name.
`object_id`	Volume-wide unique object ID.
`principal_id`	ID of the app principal that owns this object.
`coord_id`	ID of the coordinator SG.
`volume_id`	Which Volume contains this object.
`generation`	Object generation nonce.
`last_mod`	Object last-modified nonce.
`perms`	Permissions for this object.
`read_ttl`	Cached metadata lifetime.
`write_ttl`	Metadata write-back delay.

Table 1: *Partial list of record fields in the Metadata Service.*

3.4 Programming Model

The goal of our protocol designs is to isolate provider-specific logic from provider-agnostic logic, while letting the developer control how Syndicate processes application data. To achieve this, each SG provides an event-driven programming model, where the developer implements a set of callbacks that operate on an application request (Table 2). The developer writes code and configuration for the application's SGs, and uses the MS to automatically distribute it to them.

An SG's configuration is split into built-in state, application-supplied state, secret state, and object attributes. Built-in state includes "inventory" information, such as its public key, its application-chosen name, its host and port, and its owner's ID.

Application-supplied state includes key/value pairs defined by the developer, which have extension-specific interpretations. Secret state includes key/value pairs that are sealed with the SG's public key before being pushed out, thereby ensuring that only the target SG can read them. For example, the developer's credentials to a cloud storage provider would be distributed as secret state to the appropriate Replica SG.

Object attributes are key/value pairs that are defined per-object. They are readable and writable only to authorized User SGs and the MS (discussed in Section 3.5).

When a method is called on an object, it has access to all of the above configuration, as well as internal methods within the SG (not described here). It also has access to local code libraries, local storage, and its own private RAM for storing soft state across calls.

User SG	Replica SG	Acquisition SG
`connect_cache(): socket`	`read(request), get(key): bytes`	`manifest(m,obj): int`
`write_preup(data): int`	`write(request,data), put(key,data): bool`	`read(block): int`
`read_postdown(data): int`	`erase(request), del(key): bool`	`publish(spec): int`
`chcoord_begin(obj): int`		
`chcoord_end(obj,status): int`		

Table 2: *Programmatic interface overview for each Syndicate Gateway. The Replica SG's interface has a provider-agnostic half (read(), write(), erase()) and a provider-specific half (get(), put(), del()).*

User SG API. User SG code opens connections to edge cache providers (`connect_cache()`), and applies application storage logic to data before uploading it (`write_preup()`) and after downloading it (`read_postdown()`). This gives the application a chance to apply end-to-end storage logic (such as encryption and formatting) on the data without having to deal with scalability and fault tolerance. These methods are called for both manifest and block data.

The User SG calls `chcoord_begin()` and `chcoord_end()` before and after becoming an object's coordinator. This lets application control initial object distribution, and how and when to change it (for example, an application may require popular objects to be pinned to highly-available SGs). The application can explicitly set an object's coordinator, and Syndicate can try to change it (subject to these methods) to tolerate faults and distribute load.

Replica SG API. The Replica SG serves manifest and block data to edge caches, and uses its storage logic to access blocks and manifests in storage providers. It divides its functionality into both provider-agnostic (`read()`,`write()`,`erase()`) and provider-specific (`get()`,`put()`,`del()`) logic, in order to make the former reusable while allowing the latter to be designed just once. Unlike the others, Replica SG methods must be idempotent, so Syndicate can automatically retry failed writes and deletes.

Acquisition SG API. The Acquisition SG serves blocks and manifests to edge caches automatically, but imposes no constraints on how to acquire data from the dataset provider. Its logic translates arbitrary datasets into a directory hierarchy according to a given specification, and uploads the directory tree to the MS (via `publish()`) when it initializes. The specification is provided by the data curators, so they can white-list the ways in which the data can be accessed by external parties (necessary when data is subject to external usage policies, such as privacy laws).

Once it has published the data, the Acquisition SG translates manifest and block requests into the appropriate calls to the dataset provider (`manifest()` and `block()`). Because data may be dynamically generated, it does not report object sizes immediately, nor does it reply immediately with data. Instead, it may send the User SG an EOF message for block offsets that are logically beyond the end of the object, and a "try again later" message for requests that will take a long time. The User SG handles both messages transparently to the application.

3.4.1 Point-Solutions, Revisited

To demonstrate the power of this programming model, we briefly sketch solutions to the three application domains we described earlier. A scientific data-processing application can implement the Replica SG's provider-agnostic interface to snapshot and stage gene sequences for a particular organism, such that continuous updates from grid computers and collaborators do not corrupt cloud-hosted data being accessed by compute nodes. It can implement the Acquisition SG's interface to export data in legacy on-site storage as a Volume for collaborators to access.

A collaborative document-sharing application can store documents as objects, and store a document schema as an object attribute. It can implement the User SG interface to impose the schema's structure and content requirements for the associated document objects. At the same time, it can implement end-to-end document encryption to keep fields private from providers and users, using an object attribute to hold the shared secret (which is protected by Syndicate's built-in access controls, as described in the next section). To efficiently ensure durability in the face of changing sets of providers, the Replica SG's provider-agnostic interface can be implemented to store erasure codes to multiple providers instead of full replicas.

A VDI application can implement the User SG interface to prompt for additional credentials, in order to authenticate to an in-house identity service prior to downloading the VM image. It can distinguish between system and user partitions in the disk image, so it can request the former from shared caching infrastructure (since many users share it) and cache data from the latter to local disk.

3.5 Default Consistency and Access Control

Unless explicitly overwritten by the developer in an SG's extension, Syndicate offers built-in default consistency and access controls. This lets developers rapidly deploy new storage functionality without repeatedly addressing common cases of these concerns.

Similar to edge cache control directives, the application controls both how long (in wall-clock time) cached object metadata is considered fresh in the User SG (via `read_ttl`), and how long to wait before uploading new object metadata on write (`write_ttl`). Values of zero make metadata access synchronous, and larger values reduce amortized access latency at the expense of data consistency. In addition, Syndicate offers last-write-wins semantics by default for both data and metadata, and the application controls whether or not data is "flushed" on write or via an explicit `fsync()` request.

In doing so, Syndicate offers four built-in consistency models. When both metadata access and writes are synchronous, objects have sequential consistency. But if writes are asynchronous, objects have close-to-open consistency. If metadata access is asynchronous and writes are synchronous, the object has delta consistency. If both metadata access and writes are asynchronous, the object's consistency is determined by the order in which the application flushes writes.

By default, Syndicate ensures that a User SG cannot discover objects that it cannot access. The MS uses the `perms` field to determine whether or not a User SG can read, write, or search an object or directory. Similar to POSIX permissions, these permissions apply based on whether or not the User SG is running on behalf of the principal that created the object (akin to "owner" permissions), whether or not it is the coordinator for the object (akin to "group" permissions), and whether or on it is in the same Volume as the coordinator (akin to "world" permission). The Volume owner also sets Volume-wide capabilities for User SGs—namely, to write data, to read or write metadata, and to coordinate writes. These preempt object permission bits, and apply to object attributes as well.

At a higher level, Volumes can be marked as "private," meaning that a User SG must prove that it is acting on behalf of an authorized principal (as opposed to "public," which allows anonymous read-only access). They may also be marked as serving as an "archive," meaning that they are read-only to every SG, except for Acquisition SGs owned by the Volume's owner. This lets external dataset providers export data for Syndicate's consumption, without clobbering objects.

These access controls determine how SGs can alter the state of the Volume, and are enforced on top of of application-supplied logic by default. However, they do not prevent direct access to object data via providers; the application should implement end-to-end encryption in the User SG if this is a concern.

3.6 Scalability and Fault Tolerance

Regardless of application-supplied storage functionality, Syndicate must transparently scale in the number of SGs and number of writes while tolerating failures in both. To manage SGs at scale, Syndicate treats code and configuration like objects, allowing SGs to use edge caches for scalable distribution. SGs exchange last-modified nonces for the set of code and configuration via an asynchronous gossip protocol, so new versions pushed to the MS are quickly discovered and pulled. They are signed with the Volume's private key to prove authenticity and integrity.

A logically-single Replica SG can be instantiated multiple times to scale write bandwidth, either alongside each User SG, or in a cluster behind a load-balancer. Because its storage methods are idempotent, User SGs can retry or roll back failed writes regardless of instantiation.

Syndicate scales aggregate write load by dynamically distributing objects across User SGs. At first, an object's coordinator is the SG that created it. If it becomes unavailable, or the application changes it, a candidate SG (i.e. a subsequent writer User SG or the application's new choice) will ask the MS to become the new coordinator. To do so, it submits the current object generation nonce with the request, and the MS atomically regenerates it on success. This forces other candidates to refresh the object metadata, causing them to learn the new coordinator before they try again. This also causes writer SGs that cannot contact each other to "take turns" coordinating, keeping objects available for writes in the face of such partitions (albeit with degraded performance).

Without being given extra functionality for doing so, an SG cannot make progress if the MS is inaccessible. This is equivalent to a non-Syndicate application losing access to its providers. However, the MS implementation is portable across multiple cloud platforms, so the application developer can choose the platform that offers the best availability/cost trade-off.

3.7 Security

At all times, Syndicate must provide a storage layer which the application can trust to enforce access control, authenticate principals, and distribute its extensions. Our threat model assumes that eavesdroppers (Eve) read data on the network and within providers, but not within Syndicate components, TLS CAs, or the MS's underlying storage. We also assume that malicious agents (Mal) try to spoof the MS and SGs, and try to compromise running SGs. Syndicate thwarts Eve with end-to-end encryption, and limits the damage Mal can do by enforcing built-in access controls and detecting attempted spoofs.

A Volume acts as a virtual certificate authority and key repository for its SGs. It maintains a signed certificate for each authorized, non-anonymous SG, each of which cryptographically binds the SG to its Volume and configuration (including its public key, capabilities, host, and port). If desired by the developer, it also hosts a copy of the encrypted private key for each SG (accessible only after the SG authenticates), which is automatically generated, sealed, and uploaded when the SG is provisioned.

Each SG maintains a certificate bundle for all other SGs in the same Volume. To distribute it at scale, Syndicate represents it as an object, where each certificate is a "block." The MS and SGs gossip its cryptographic hash and last-modified nonce, and automatically re-synchronize the bundle if it changes. Unlike a typical "object", the last-modified nonce of the certificate bundle increases monotonically, so SGs can detect stale bundles submitted by Mal.

To ensure the application can trust the storage layer created by the running SGs, Syndicate puts it in charge of bootstrapping the trust on a per-session basis. When the developer starts an SG, she gives it login credentials to use to register itself with the MS, as well as its private key decryption password. The SG submits the login credentials (via TLS) to a developer-chosen OpenID [28] provider, which the MS is configured to trust (i.e. either a 3rd party one, or one built into the application logic). Once the OpenID provider vouches for the authenticity of both components, the SG obtains its encrypted private key and decrypts it locally.

While running, Syndicate uses TLS to keep control-plane messages confidential and to prevent replay attacks from Mal. Each message is signed by the sender, allowing it to ignore Mal's unverifiable messages. To keep Mal from silently altering data in a provider, a coordinator signs each manifest; since the manifest contains the block's cryptographic hashes, a reader SG can detect tampering.

A compromised User or Acquisition SG is limited to damaging only its Volume, and only as far as Syndicate's built-in access controls allow (provided that an application-supplied extension does not weaken them). The MS always forbids a Replica SGs from altering Volume metadata, but a compromised Replica SG can destroy data. Application developers hedge against data loss by using multiple Replica SGs linked to different storage accounts in different providers.

4. PRELIMINARY EVALUATION

We have deployed our MS on Google AppEngine [17], and our Replica and Acquisition SGs on PlanetLab [26]. They use a CDN running on VICCI [42] based on CoBlitz [25], and use Amazon S3 to make data durable. We are in the process of gathering large-scale usage data.

Syndicate's performance is partially determined by the underlying providers. Our microbenchmarks indicate that User SG and Replica SG contribute a small constant-factor overhead on top of directly uploading data to cloud storage and downloading data from the CDN.

Syndicate's main performance bottlenecks are in reading consistent data in the face of writes, and in handling many writes on the same object. The costs for consistency come from re-synchronizing stale metadata and missing overwritten blocks in the cache. A preliminary test with PlanetLab nodes shows that half of metadata refreshes take less than 200 milliseconds, and 90% take less than 300 ($n = 300$).

Another early consistency evaluation shows that the time required to read an object is linear in the fraction of blocks overwritten, with $r^2 = 0.983$ for the median read times and $r^2 = 0.939$ for the 90% percentile read times ($n = 300$). The nodes read a 100-block object in increments of 10 blocks, with a 60KB block size (i.e. the size of a large gene sequence from GenBank). Blocks were downloaded sequentially, but we expect similar results if the blocks are downloaded in parallel. This represents the expected performance profile for reading with ongoing writes, as well as for distributing new certificates and SG code.

The performance costs in writing to the same object come from write serialization in the MS. In an experiment where 300 nodes wrote metadata to the same object, 50% of MS responses take less than 1500 milliseconds, and 90% take less than 2350 milliseconds. The MS operations are I/O bound in local tests, and according to Google AppEngine's internal measurements, at most 250 milliseconds are spent accessing the datastore and performing TLS negotiation. This suggests most of the latency comes from the platform's internal request buffering, including waiting for new front-end instances to spin up. As such, we believe this data is indicative of the MS's performance under load. We are working on preventing the platform from contributing so much overhead.

We are also in the process of evaluating the User SG's ability to handle many concurrent writes, as well as its ability to shed load to other User SGs. We anticipate that a loaded User SG will naturally distribute coordinator responsibility to heavy writers, since statistically their writes are most likely to fail first.

The take-away from our preliminary tests is that composing providers to reap their aggregate utility benefits is feasible in practice. They – not Syndicate – are the limiting factor in performance, so it is to the developer's advantage to select providers with the best cost/performance trade-off while using Syndicate.

5. RELATED WORK

Syndicate adds to a growing body of work on in separating storage aspects into independent composable functional units, which can be layered on top of existing providers. Separating consistency has been exploited to provide causal consistency [5], ACID semantics [22], and key-group transactions [13] on top of unmodified cloud storage. Separating access and admission control has been exploited to implement single sign-on across multiple providers [28, 36, 33]. Recently, cloud storage gateway middleboxes (such as [30, 41]) let developers separate some storage logic from providers, including encryption and de-duplication.

Unlike these systems, Syndicate abstracts storage aspects into a provider-agnostic programming model, allowing the application to control how the storage system addresses them. Syndicate provides default semantics for each in order to lower the barrier to entry for developing new functionality.

By providing default semantics, Syndicate is superficially similar to many peer-to-peer storage systems [29, 35, 12, 31, 39] and distributed filesystems [34, 2, 21, 4, 24], where the contributions are in orchestrating many wide-area computers to provide a logically-single provider. The key difference between these systems and ours is that we focus instead on orchestrating multiple providers (which have wildly different APIs, semantics, and behaviors compared to individual hosts) in order to offer consistent, application-defined functionality. Syndicate is a layer above these systems.

Syndicate contributes to the emerging field of software-defined storage (SDS). Previous work [38, 6] focuses on SDS in the datacenter, a problem domain concerned with enforcing end-to-end storage QoS, providing tenant isolation, and pooling storage. In contrast, Syndicate focuses on SDS for the wide-area, where the main concerns are in keeping data available in the face of Internet-scale read and write traffic while handling faults and mitigating security breaches. These separate domains have led to very different storage architectures, despite the complementary roles Syndicate and these systems play.

6. CONCLUSION

This paper presents Syndicate, a wide-area virtual cloud storage service with software-defined storage semantics. By interposing programmable gateways at the contact points between the application and underlying providers, Syndicate transparently leverages them without modification, and without coupling applications to them. In doing so, Syndicate gives applications the utility benefits various providers offer, as well as a programming interface for them to define their storage functionality in a provider-agnostic fashion.

Syndicate is under development, and is being deployed on PlanetLab. Its source code and documentation are available online at `http://github.com/jcnelson/syndicate`.

7. ACKNOWLEDGEMENTS

The authors would like to thank Wathsala Vithanage, Illyoung Choi, Muneeb Ali, John Whelchel, and Stephen Tredger for their contributions and feedback on the Syndicate prototype implementation. They would also like to thank Sapan Bhatia, Andy Bavier, John Hartman Matvey Arye, and Ariel Rabkin for their feedback on earlier versions of this paper. This work is funded by NSF grants CNS-0958340 and MRI-1040123.

8. REFERENCES

[1] 1000 Genomes: A Deep Catalog of Human Genetic Variation. `http://www.1000genomes.org/`.
[2] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *PROCEEDINGS OF THE 5TH USENIX SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION*. USENIX, 2002.
[3] Overview of Amazon Web Services. `https://d36cz9buwru1tt.cloudfront.net/AWS_Overview.pdf`.
[4] S. Annapureddy, M. J. Freedman, and D. Mazieres. Shark: Scaling file servers via cooperative caching. In *Proc. 2nd NSDI*, Boston, MA, 2005.
[5] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.
[6] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 242–253, New York, NY, USA, 2011. ACM.
[7] Blackboard learning system. `http://www.blackboard.com`.
[8] J. Blomer, P. Buncic, and T. Fuhrmann. Cernvm-fs: delivering scientific software to globally distributed computing resources. In *Proceedings of the first international workshop on Network-aware data management*, NDM '11, pages 49–56, New York, NY, USA, 2011. ACM.
[9] Box.net. `http://www.box.net/`.
[10] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. Lam. The collective: a cache-based system management architecture. In *Proc. 2nd Conference on Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.
[11] Citrix VDI-in-a-Box. `http://www.citrix.com/products/vdi-in-a-box/overview.html`.
[12] M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. Practi replication for large-scale systems. Technical report, 2004.
[13] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 163–174, New York, NY, USA, 2010. ACM.
[14] Dropbox. `http://www.dropbox.com/`.
[15] Filesystems in Userspace. `http://fuse.sourceforge.net`.
[16] GenBank. `http://www.ncbi.nlm.nih.gov/genbank/`.

[17] Google App Engine. `https://developers.google.com/appengine/`.

[18] Google documents. `https://docs.google.com/`.

[19] Google Drive. `http://www.measurementlab.net/`.

[20] S. M. Larson, C. D. Snow, M. Shirts, V. S. P, and V. S. Pande. Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology.

[21] Tahoe Least-Authority File System. `https://tahoe-lafs.org/trac/tahoe-lafs`.

[22] J. J. Levandoski, D. B. Lomet, M. F. Mokbel, and K. Zhao. Deuteronomy: Transaction support for cloud data. In *CIDR*, pages 123–133. www.cidrdb.org, 2011.

[23] Metagenomics MG-RAST. `http://metagenomics.anl.gov/`.

[24] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *PROCEEDINGS OF THE 5TH USENIX SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION*. USENIX, 2002.

[25] K. S. Park and V. Pai. Scale and Performance in the CoBlitz Large-File Distribution Service. In *Proc. 3rd NSDI*, San Jose, CA, 2006.

[26] L. Peterson, A. Bavier, M. Fiuczynski, and S. Muir. Experiences Building PlanetLab. In *Proc. 7th Operating System Design and Implementation (OSDI)*, Seattle, Washington, Nov. 2006.

[27] A. Rajasekar, R. Moore, C.-y. Hou, C. A. Lee, R. Marciano, A. de Torcy, M. Wan, W. Schroeder, S.-Y. Chen, L. Gilbert, P. Tooby, and B. Zhu. *iRODS Primer: integrated Rule-Oriented Data System*. Morgan and Claypool Publishers, 2010.

[28] D. Recordon and D. Reed. Openid 2.0: a platform for user-centric identity management. In *Proceedings of the second ACM workshop on Digital identity management*, DIM '06, pages 11–16, New York, NY, USA, 2006. ACM.

[29] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the oceanstore prototype. In *PROCEEDINGS OF THE 2ND USENIX CONFERENCE ON FILE AND STORAGE TECHNOLOGIES*. USENIX, 2003.

[30] Riverbed Gateway. `http://www.riverbed.com/us/company/news/press_releases/2011/press_060711.php`.

[31] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK, 2001. Springer-Verlag.

[32] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[33] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *in Usenix Conference Proceedings*, pages 191–202, 1988.

[34] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris. Flexible, wide-area storage for distributed systems with wheelfs. In *Proc. 6th NSDI*, Boston, MA, 2009.

[35] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 172–183, 1995.

[36] The OAuth 2.0 Authorization Framework. `http://tools.ietf.org/html/rfc6749`.

[37] The Open Electronic Medical Records Project. `http://www.open-emr.org`.

[38] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 182–196, New York, NY, USA, 2013. ACM.

[39] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for internet data transfer. In *In Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI*, pages 253–266, 2006.

[40] L. Torvalds. git. `http://git-scm.com/`.

[41] Twinstrata Gateway. `http://www.twinstrata.com/`.

[42] VICCI: VIrtual Cloud Computing Infrastructure. `http://www.vicci.org/`.