Syndicate: Building a Virtual Cloud Storage Service Through Service Composition

Jude Nelson
Princeton University

Larry Peterson
Princeton University

Abstract. Applications increasingly combine cloud storage, edge caches, and local storage to host and share data. They do so, however, in ad-hoc ways, producing a landscape of point solutions that solve similar problems. This paper describes an alternative approach that composes storage systems in a general, but configurable way. The resulting system, called Syndicate, factors out data consistency, system security, and storage policies in a coherent manner, allowing developers to control these functions independent of the underlying storage. This paper describes Syndicate's design, and in particular, its novel consistency protocol. It also evaluates a PlanetLab-based deployment of Syndicate, showing that it introduces a small amount of overhead relative to directly using underlying storage mechanisms.

1 Introduction

The cloud is changing how users store and access data. This is true for both legacy applications that are migrating local data into the cloud, and for emerging cloud-hosted applications that use a combination of content distribution networks (CDNs) and client-side local storage to achieve scalability. In both cases, the goal is to compose existing storage and caching services to implement scalable storage for the application. However, the challenges in doing so are to keep data consistent, keep the storage layer secure, and enforce storage policies across services. This paper presents Syndicate, a wide-area storage system that meets these challenges in a general, coherent manner.

There are two reasons to compose existing services to implement scalable storage. First, if we factor a scalable storage system by the capabilities of existing services, it is interesting to ask what extra functionality is needed, and how is it best provided? Our strategy is to use cloud storage for durability and scalable capacity, edge caches (CDNs and caching proxies) for scalable read bandwidth and reduced upstream load, and local storage for fast, possibly offline access. Once these capabilities are met, the remaining desirable functionality is wide-area consistency, storage layer security, and storage policy enforcement. Syndicate introduces a new Metadata Service and storage middleware that provides this functionality in a general, configurable way.

The second reason is that leveraging existing services is more than just taking advantage of someone having already implemented the corresponding functionality. More importantly, doing so also leverages a globally-deployed and professionally-operated instantiation of that functionality. Deploying and operating a global service, even with the availability of geographically distributed VMs, is an onerous task. Syndicate side-steps that problem by using existing storage and caches, and in the process, significantly lowers the barrier-to-entry to constructing a customized, global storage service.

The thorniest problem Syndicate faces is contending with the weak consistency offered by edge caches. Namely, the cache operator, not the application, ultimately decides what constitutes "stale" data for eviction purposes, making it difficult to rely on cache control directives. Rather than trying to avoid caches, we intentionally incorporate them into the solution due to the tangible benefits they offer. Already, enterprises deploy on-site caching proxies to accelerate Web access for users, content providers employ global CDNs to distribute content to millions of readers, and ISPs deploy CDNs in their access networks to avoid transferring frequently-requested data over expensive links [18, 23, 45].

Our key insight into composing existing storage and caching services is that we should *avoid* treating them as first-class components. Instead, we must treat them as interchangeable utilities, distinguishable only by how well they provide their unique capabilities. Then, applications select services arbitrarily, and layer consistency, security, and policies "on top" of them with Syndicate. By doing so, we provide them a virtual cloud storage service.

Our key contribution is a novel consistency protocol,

employed in both the data plane and control plane, that achieves this end. With this protocol, Syndicate overcomes the weak consistency of edge caches in the data plane while continuing to leverage the benefits they offer. It also lets Syndicate leverage caches in the control plane to scalably distribute certificates and signed executable code to middleware end-points, in order to secure the data plane and enforce data storage policies in a trusted manner.

2 Usage Scenarios

Syndicate is meant for applications that handle a scalable read/write workload in interactive settings, where local storage, cloud storage, and edge caches have complementary roles to play. To motivate its design, we explore three application domains that currently rely on point-solutions, and argue that an equivalent solution can be built from existing systems by layering the proper functionality over them. A concluding subsection summarizes the common themes in how this missing functionality is applied.

2.1 Scientific Datasets

Tens of thousands of computers work together to process multi-terabyte data sets; several genomic sequencing data sets being illustrative examples [24, 41, 1, 64]. 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 donated computers in the wide-area. At the same time, they share findings with collaborators, archive working data sets in cloud storage, and submit vetted results for integration into the original data set.

A hodge-podge of point-solutions have been developed to address various aspects of scientific data storage, but no coherent solution has emerged. In an ideal world, researchers would keep relevant data on their local workstations for fast access, and seamlessly stream data from the main data set on an as-needed basis (as in [47]). The data sets would support a scalable number of readers by employing edge caches (as in [10]), provided that readers are guaranteed to receive fresh data. Results from authorized grid computers would be sanitized and uploaded to researcher-chosen storage when it is cost-effective to do so (as in [35]). Cloud storage providers would be selected based on how expensive it is to transfer data to computing infrastructure later on.

2.2 Collaborative Document Sharing

Collaborative document sharing is widely used today, from version control systems like `git` [63] to online word-processing systems like Google Docs [27]. In all cases, a set of users read and write a set of documents, and the system provides an interface for user-driven conflict resolution.

Nevertheless, there are cases where the trustworthiness of their underlying storage systems limit their applicability. For example, hosting medical documents may require keeping a verifiable edit history and implementing certain privacy requirements. As another example, some organizations require online edits to be mirrored to private servers (as offered in [11]).

Ideally, a user could bring her own trusted storage system for hosting her writes. Then, the application writes her edits to it (with her permission), and makes them visible to a scalable number of readers by routing their requests to them through edge caches. However, a layer of abstraction will be necessary to separate user-facing features from user-given providers, and ensure readers do not receive stale data from edge caches.

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 fact that VM images do not change much between sessions [13] and have only one writer (the user) to achieve scalable VM deployment through local and on-site VM block caching.

While some VDI implementations [14, 42] use system-specific infrastructure, the ideal scenario is to let the corporation choose the cache and storage providers that best meet their business needs. The VDI infrastructure is not desirable if there already exists proven in-house infrastructure—not only for caching and storage, but also for VM security and identity management.

2.4 Observations

Our goal is for Syndicate to be flexible enough to serve as the storage layer for these and similar applications, allowing them to take advantage of existing services rather than having to create point solutions. The challenge is to layer the required functionality on top of the existing services, and to this end, we observe three common themes in how these applications use storage.

First, once consistency requirements are met, the value of edge caches is to improve data locality. In all three examples, the systems are able to scale in the number of reads because data naturally propagates to points in the Internet that are "closer" to readers, without the application having to do any extra work. This implies that

the storage layer should opportunistically exploit them for read performance, but handle consistency separately. It also suggests that the storage layer speaks HTTP and addresses object chunks and metadata with URIs.

Second, once write performance and storage policies are met, the value of cloud storage providers is to enhance data durability. As seen in these examples, the storage policies (including privacy) can be so specific that it is unrealistic to expect a provider to enforce them. This suggests that the virtual storage layer should do so itself, and treat a cloud storage provider as public stable storage with no functional guarantees. This requires the application to trust it to do so correctly, however.

Third, data consistency is application-specific, but metadata consistency has general requirements. Because reads and writes must be routed to the correct data objects for these applications to work, an object name (URI) must refer to at most one object at all times. This means updates to the object namespace must be atomic with respect to reads and writes. Since Syndicate will be used in interactive workloads, the application must be able to structure and search the namespace dynamically.

3 Design

Syndicate defines a cloud storage abstraction, called a *Volume*, that organizes application data across underlying storage. An object written to a Volume is stored in its cloud storage providers according to application storage policies, and later delivered to readers via its edge cache providers. Within a Volume, objects are organized into a filesystem-like directory hierarchy, with an additional control interface that lets administrators trade-off read performance and data consistency.

Syndicate's consistency protocol is central to implementing the Volume abstraction, since the protocol covers its data, metadata, and control plane information. We describe the protocol in two phases. First, we assume that the relevant components have fresh metadata, and show how they use it to overcome weak cache consistency without losing the benefits edge caches offer. Second, we show how they obtain fresh metadata in a scalable manner. We then show how Syndicate implements fault tolerance, storage layer security, and storage policies, using the consistency protocol to distribute the necessary control plane information.

3.1 Components

Syndicate consists of two components: a set of peer *Syndicate Gateways* (SG), and a scalable *Metadata Service* (MS). SGs are the middleware processes that interface between Syndicate and the existing storage elements, and work together to implement the Volume abstraction for applications by managing object data. The SGs coordinate to meet consistency, security, and storage policies via a shared MS, which itself manages object metadata and facilitates Volume administration.

The SG comes in three variants, depending on how it interfaces with the outside world:

User SG: Interfaces with user/application processes in order to implement the Volume abstraction. Responds to both locally generated read/write requests and read/write requests from peer SGs. Our prototype offers three application-facing interfaces: a FUSE [21] filesystem, a RESTful Web object store, and a Hadoop Filesystem [53] front-end.

Replica SG: Uploads written data to cloud storage providers on behalf of the Volume, and later downloads it and serves it to edge caches on demand. Responds to read/write requests from peer SGs, but does not generate any local requests. Our prototype currently supports Amazon S3 [5], Dropbox [19], Box.net [11], Google Drive [28], Amazon Glacier [5], and local disk as back-end storage.

Acquisition SG: Maps existing data sets into one or more Volumes as a read-only directory hierarchy. Responds to read (but not write) requests from peer SGs, but does not generate any local requests. Our prototype currently supports GenBank [24], Common Crawl [15], and M-Lab [40] data sets.

Although there are three variants of the SG, they all play the same role in the larger Syndicate design, and so we do not distinguish among them unless critical to understanding the system.

In a typical deployment (Figure 1), an application has one MS, and places its objects into one or more Volumes. Each host runs an SG locally for each Volume to which it has access, so processes only see the intended data. The application developer deploys additional SG instances to allow the application to store and later retrieve data from cloud storage, and to optionally read (but not write to) external existing data sets.

3.2 Object Data and Consistency

To manage a scalable number of objects, Syndicate divides responsibility for them among the SGs in the Volume. Each object in Syndicate is assigned one SG to act as its coordinator. A coordinator has two responsibilities: implementing data plane consistency with the help of the MS, and processing writes from other SGs.

It is difficult to download fresh data from HTTP-speaking edge caches because they do not always honor cache control directives, for example, due to bugs, misconfiguration, or intentional shunning by the edge cache

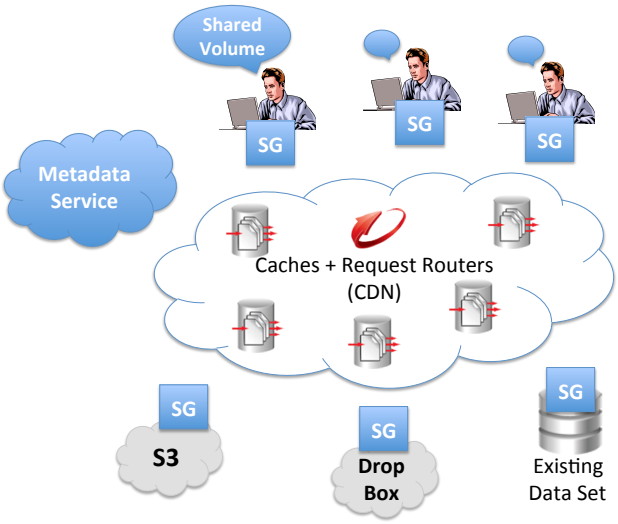

Figure 1: *Example deployment of SGs between edge caches, cloud storage, existing data sets, and application local storage.*

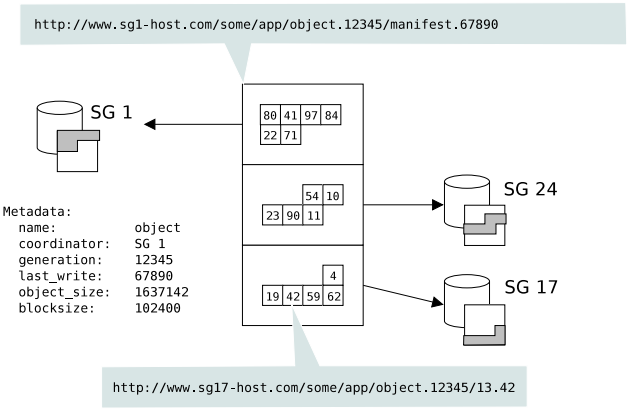

Figure 2: *Logical representation of a manifest for a 16-block object after three writes. Each SG (cylinders) hosts the latest copies of some of the object blocks (grey areas), which the manifest tracks. Generation nonces, last-write timestamps, and block nonces are kept short for brevity.*

operator. Thus to obtain fresh data, SGs use HTTP URIs that uniquely identify the latest snapshot of the data, obviating the need for them. The challenges are to use cache capacity efficiently, and ensure SGs discover the correct URIs before downloading.

To address the former, SGs serve object data as fixed-sized blocks (the size is application-defined). To address the latter, a coordinator SG stores two records for each block: a block nonce that uniquely identifies its current contents, and the set of which SG(s) can serve a copy. It additionally stores a unique object generation nonce assigned on creation, and a monotonically-increasing object last-write timestamp. It keeps track of this information using an object *manifest* (Figure 2), and uses this information to generate block URIs.

To read an object, an SG first obtains the object's latest generation nonce and last-write timestamp from the MS. Then, it generates an HTTP URI for the manifest using these two records, as well as the object's path. By including these records, the URI to the manifest will identify a snapshot as fresh as the records indicate. This allows the SG to download the manifest from the coordinator via the edge caches and receive fresh data.

Once obtained, the manifest lets the reader construct URIs for each of the object's blocks. A block URI includes the path to the object in the Volume, the generation nonce, the block ID, and the block nonce. Because the block nonce changes whenever the block is modified, the URI will change whenever the block changes. This way, the SG can distribute data with unmodified edge caches and guarantee that readers see data as fresh as the manifests indicate.

When an SG writes to an object, it first obtains fresh metadata and manifest data, and sends the modified blocks to a Replica SG in the Volume for archival to cloud storage. The writer then generates new nonces for the modified blocks and sends them to the coordinator. Upon receipt, the coordinator generates a new last-write timestamp, adds the block nonces in the manifest, and uploads the manifest to a Replica SG as well. It then sends the last-write timestamp to the MS for subsequent readers to fetch. It finally acknowledges the writer SG, completing the write operation.

From a consistency standpoint, Syndicate ensures readers receive data as up-to-date as the downloaded manifest, which itself is as up-to-date as the object metadata from the MS. By default, the SG offers last-write-wins semantics, which is adequate for our sample applications, and is comparable to what cloud storage providers offer today (although other semantics are possible). Reads and writes are not atomic with respect to one another, but synchronously fetching new metadata and manifests on each read yields per-object sequential consistency, with writes observed in the coordinator-observed order. By varying how stale cached metadata and manifests are allowed to become, the SG can trade stronger consistency for faster access (Section 3.3).

The SGs cache manifests and block data for locally-read objects, and continue to host recently-written data after it has been backed up. Once durable, hosted data is evicted once they become remotely overwritten, or if they are infrequently accessed. Manifests for the objects the SG coordinates are always kept resident. We allow an application to control this policy per-object, in order to guarantee that the object's data remains locally available for fast and/or off-line access.

3.3 Metadata and Consistency

While SGs handle reads and writes on object data, the MS helps them coordinate to implement data consistency. Specifically, SGs rely on the MS to announce their presence, to announce the objects they coordinate, to discover other SGs and their objects, and to help them read sufficiently recent object data from one another.

The MS maintains a metadata record for every object and directory in each Volume, as well as records for each SG. An object/directory record is functionally similar to an i-node, except that it does not track blocks (this is handled by the object manifest). A listing of relevant fields the MS tracks can be found in Table 1.

Name	Description
`type`	Object or Directory record.
`name`	App-chosen record name.
`object_id`	Volume-wide unique object ID.
`user_id`	App user ID that owns this object.
`coord_id`	ID of the coordinator SG.
`volume_id`	Which Volume contains this object.
`generation`	Object generation nonce.
`last_write`	Object last-write timestamp.
`perms`	Permissions for this object.
`read_ttl`	Cached lifetime on an SG.

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

The MS serves as the coordinator for all directories in a Volume, and processes object/directory creation and deletion requests on behalf of the Volume's SGs. As such, creations and deletions within a given directory are serialized, ensuring that a directory's immediate children have unique names. This ensures that an object URI refers to at most on object, as desired.

The freshness of an object's data is determined by how fresh the object's cached metadata is on the SG, which applications control to trade consistency (freshness) for read performance To do so, Syndicate gives each object and directory an application-set `read_ttl` field that tells the SG for how long metadata can be assumed fresh after it is downloaded. This is functionally equivalent to applying cache-control directives used today, and provides delta consistency [62] when `read_ttl` is non-zero (a value of zero provides sequential consistency). Syndicate additionally lets an application invalidate cached metadata, if it needs the next read operation to obtain fresh data.

Metadata Consistency Protocol. Since Syndicate is intended for read-heavy workloads, we assume that directories are rarely added or removed relative to the how often they or their children are accessed. For this reason, SGs locally cache and revalidate metadata grouped by directory listings, and only redownload listing when they discover that it has changed on the MS. This way, most of the time, the SG will have a locally cached (but potentially stale) listing for each directory along the paths to the objects it accesses.

When given a path to an object, the SG first walks down the cached directory listings along the path. It breaks the path into three disjoint segments of directory listings: the longest prefix along the path where each listing is both cached and fresh (P_{fresh}), the segment of listings after this prefix that starts with a locally cached but stale listing (P_{stale}), and the segment of listings after P_{stale} that are not locally cached (P_{absent}).

The SG does not need to revalidate the listings in P_{fresh}, but will need to revalidate every listing in P_{stale} since undiscovered changes in them can affect all subsequent listings. To do so, the SG sends the MS the cached `object_id` and `last_write` fields of each listing's directory. For each directory given, the MS will return the new listing if it has a different `last_write` value than indicated by the SG. Otherwise, it either responds with a "Not Modified" message if the directory has the same last-write timestamp, or with a "Not Found" message if the directory does not exist or the SG is no longer allowed to search it.

After processing the MS response, every directory listing in P_{stale} will be fresh. Then, for each listing in P_{absent}, the SG sequentially downloads and caches each listing in order to learn the `object_id` for the next listing (which is necessary to request it). Once all listings in P_{absent} have been downloaded, every listing along the path will be fresh, and the SG can proceed to access the object (whose fresh metadata is contained in its parent directory's listing).

The intuition behind this protocol is that it usually costs at most one RTT when objects exhibit access locality. When an object is accessed multiple times, directories that had been in P_{absent} the first time will likely be in either P_{fresh} or P_{stale} afterwards, meaning P_{absent} is usually empty. So, the cost of obtaining fresh object metadata, fresh path metadata, and checking search permissions is usually the cost of revalidating P_{stale}. The SG will not need to contact the MS at all if P_{fresh} includes the entire path, or if P_{stale} is empty and P_{absent} is not (in which case, the object can be inferred not to exist).

3.4 Provider Composition

Beyond consistency, applications have domain-specific, changing storage policies. To meet them, Syndicate automatically and securely distributes developer-supplied idempotent closures to SGs, which SGs call on reads and writes. They are used for achieving compatibility with specific providers, and enforcing storage policies.

When a user provisions an SG, she supplies the MS a read-only configuration C, a read function R, and a write function W. When it receives a read request for a manifest or block, the SG will run R with C to obtain the requested data. When it receives a write with new manifest or block data, it will run W with C to process it.

The functional differences between SGs lie mostly in how they implement R and W. For a User SG, W writes block data to local storage using last-write-wins write conflict resolution. R translates a read request and the object metadata into cache-specific HTTP GET requests, and downloads the data from them.

The Replica SG uses W and C to upload manifest and block data to underlying providers, and uses R and C to retrieve them later. It ensures W is idempotent and space-efficient by uploading a data record unique to the version of the data, and garbage-collecting stale data. An Acquisition SG has no W, but uses R to match an object path to a per-object function R_{object}. R_{object} will be run to generate the requested data from the underlying data set and perform internal caching.

Using closures in this way decouples the storage policies from the provider, creating a marketplace of reusable R and W implementations. Compatibility with a provider only needs to be addressed once for many applications, and enforcement of higher-level policies, such as data compression, encryption, erasure coding, logging, and so on, can be embedded into these closures. Additionally, per-application optimizations such as write-batching or asynchronous replication can be addressed with them.

C, R, and W may be modified on a live system—for example, to update credentials, distribute keys, or fix bugs. To address this, Syndicate treats them as special data objects. SGs will periodically refresh closure metadata and re-download new copies automatically, using the edge caches to scale read bandwidth.

In practice, C contains sensitive information, such as account credentials or encryption keys. To securely distribute C, the developer first obtains the SGs' public keys and encrypts each SG-specific record with the appropriate key. She then signs C, R, and W before uploading them. In this capacity, as long as the user can securely obtain the SG's public keys, she can rely on Syndicate to securely distribute her closures in a scalable manner. We address this in more detail in Section 3.6.

3.5 Scalability and Fault Tolerance

Our scalability and fault tolerance goals for Syndicate are to ensure that it is not a performance bottleneck for an application, and that it is available to process reads and writes despite SG failures. To achieve this, Syndicate scales up and distributes Replica SG instances to meet write load, and automatically transfers object coordinator responsibilities between SGs when they fail.

Syndicate cannot make progress if the MS is offline. To address this, we designed the MS to be portable across many cloud computing platforms and compatible with many highly-available NoSQL store implementations, allowing the developer to choose a sufficiently reliable cloud computing platform.

To scale application writes, Syndicate instantiates and runs Replica SGs across an application-chosen set of hosts, and distributes requests to them with an existing HTTP/DNS load balancer with failure detection. Because R and W are idempotent, a read or write on a failed Replica SG instance is simply restarted.

By default, an object's coordinator is simply the SG that created it. If the coordinator fails or becomes unavailable, a reader SG will simply fetch the object manifest and blocks from a Replica SG. A writer SG will attempt to become its new coordinator, so it can process its write (and subsequent writes). Note that although Replica SGs receive written object data, they are forbidden from serving as coordinators for security reasons.

If a coordinator SG is unresponsive to a writer, the writer requests the MS to set the object's `coordinator` field to itself. If the MS approves, the SG reads the latest copy of its manifest from a Replica SG, updates it to list the Replica SG as the host for all of the blocks, and then performs its write as normal. The other SGs, including the original coordinator, will detect the change when they next refresh its metadata. A read sent to the wrong coordinator SG will either be redirected (on cache miss) or will be served "stale" data that is less stale than the object's `read_ttl` allows (on cache hit).

If multiple SGs attempt to become the new coordinator, the MS picks the first SG as the winner, increments the object's last-write timestamp, and tells the remaining SGs to instead refresh the object's metadata and restart their writes. This means if a set of writer SGs are partitioned from one another, they will "take turns" becoming the coordinator as needed until they re-establish contact. Depending on application needs, intermittent writes will either fail fast (i.e. the application is instructed to try again) or be restarted internally if the wrong coordinator attempts to process them.

Any User SG with the right capabilities can become the coordinator of an object in the Volume, provided the object has the appropriate permission set in its `perms` field. The application can also explicitly move an object between allowed coordinators. This is because the application "knows best" how to to place its objects across SGs—for example, a VDI application would try to coordinate VM images with SGs close to its user, but an online document editor would try to distribute documents across its servers' SGs to balance load.

3.6 Security

To address security, Syndicate concerns itself with providing a trusted storage layer on top of an application's hosts, using the application to bootstrap trust between components. This lets the application developer avoid trusting existing services, and creates an environment for the application to securely implement data privacy and distribute sensitive control-plane information.

Our threat model assumes that eavesdroppers (Eve) can read data on the network, but not within Syndicate components or the MS's storage. We also assume that malicious agents (Mal) try to spoof the MS and SGs, and try to compromise SGs. As such, Syndicate ensures that only authorized SGs may participate in a Volume, and only if they are running on behalf of their user on a designated host/port. Even then, they may only interact with the Volume according to developer-given capabilities.

To enforce this, each Volume acts as a certificate authority for its SGs. The Volume maintains a signed certificate for each of its SGs, identifying the information listed in Table 2. Each SG keeps a certificate bundle for all other SGs in the same Volume.

Field	Description
`pubkey`	The SG's current public key.
`hostname`	The host on which the SG runs.
`port`	The port on which the SG listens.
`user_id`	App user ID running the SG.
`volume_ids`	IDs of this SG's Volumes.
`capabilities`	The SG's per-Volume capabilities.
`issued`	Certificate issue date.
`expires`	Certificate expiration date.
`revoked`	Revocation status of this certificate.

Table 2: *Summary of the fields kept in an SG certificate.*

To distribute and revoke certificates at scale, SGs treat their certificate bundle as a Syndicate object, where each certificate is a block. The MS and SGs gossip its cryptographic hash and last-write timestamp. When either value changes (due to the addition or revocation of a certificate), the SGs synchronize their certificate bundle with the MS by fetching the bundle's manifest, fetching new certificate data through edge caches, and deleting MS-revoked or expired certificates.

The application bootstraps trust between the SG and the MS. When the developer installs an SG on a host, she gives it a set of secret credentials to use to register itself with the MS upon instantiation. It submits these credentials (via TLS) to an OpenID [48] provider, which the MS is configured to trust (i.e. either a 3rd party one, or one built into the application logic). It then vouches for the authenticity of both components to each other.

Once authenticated, the SG generates a public/private key pair for this session (or uses one given by the application), sends the MS the public key, and receives the Volume's public key and root directory listing. The MS then generates and distributes a new certificate for it. The SG will periodically renew its certificate, but the MS will revoke it if it expires, if the SG shuts down normally, or if the developer stops trusting it.

With the exception of data served to the edge caches, Syndicate uses TLS to secure all messages between its components. Each message is signed by the sender, allowing other components to ignore unverifiable messages. This scheme prevents Eve from reading control-plane data, and prevents Mal from spoofing components.

A compromised SG is limited to damaging its Volume(s), and only as far as its capabilities allow. These capabilities are used to ensure that an Acquisition SG is only allowed to create a limited number of read-only objects and directories (the MS takes care of deleting them when its certificate is revoked). They also ensure that a Replica SG may never interact with Volume metadata beyond pulling new certificate data.

To limit the damage Mal can do with a compromised User SG, an application can specify for an entire Volume which SGs have the capability to write to objects, and which can coordinate objects (without either, a User SG is read-only, and does not participate in certificate distribution). In addition, the application compartmentalizes data by partitioning objects across multiple Volumes.

4 Implementation

Building Syndicate resulted in two important artifacts: a generic and scalable Metadata Service, and a "narrow waist" interface between applications and providers.

The first, which includes a client-side library, let an application use unchanging URIs (i.e. object paths) to download guaranteed-fresh mutable data from underlying caches, without relying on them for consistency. The client library manages manifests and caching, and provides an API for uploading new metadata.

To scale on top of a NoSQL store while providing filesystem-like metadata consistency, the MS divides a record into into a seldom-written "base" record, and a set of often-written "shard" records. Writes are distributed across shards, so the MS can batch outstanding writes regardless of how long an individual `put` operation in the store takes. They are recombined into a whole record by selecting the shard with the largest last-write timestamp, providing last-write-wins write semantics. Base records are transactionally created to prevent name conflicts.

The second artifact is an application-controlled cloud abstraction layer, which forms a "narrow waist" between applications and providers. This facilitates rapid storage service deployment by reducing the task of creating a

custom storage system to deploying the appropriate SGs. It also facilitates independent provider evolution by reducing the task of supporting legacy applications to providing the *R* and *W* that lets them continue to leverage the provider in the face of otherwise-incompatible changes.

Broadly, there two deployment models for Syndicate, depending on whether or not the application is peer-to-peer or client/server. In the first model, each peer runs a write- and coordinate-capable SG. This is the deployment model we use today on PlanetLab [46].

In the second model, application servers run write- and coordinate-capable SGs, and a scalable number of clients run read-only SGs and send writes directly to application servers. This model dovetails with Web application design, where Web page (client) does not write to the storage tier (server SG) directly, but instead sends data to the application tier to write it on its behalf.

In both cases, Replica SGs may be colocated with User SGs, or deployed in a separate cluster. The former horizontally scales writes to cloud storage, but consumes extra application resources. The latter provides the cost and security benefits of a dedicated, isolated storage tier, but at the expense of one more network hop per write.

5 Evaluation

Because Syndicate is storage middleware, it is on the critical path for reads and writes. We focus on quantifying the I/O overhead Syndicate introduces on top of directly leveraging the underlying infrastructure. The common theme in our evaluation is to measure the systemic performance for a particular set of operations, and then break down where time was spent—how much in Syndicate, and how much in the underlying infrastructure. We do so by examining the *relative* performance of Syndicate on top of controlled infrastructure, compared to directly accessing data. We show that a large fraction of I/O overhead can be reduced simply by leveraging better infrastructure, validating the need for our system.

Due to its immediate applicability to PlanetLab users, we focus on evaluating the peer-to-peer deployment model. This extends to understanding the performance of our example applications—the scientists' workstations, the document servers, and VDI VM servers all run both a User SGs and Replica SGs to process, store, and serve data to one another across the wide-area.

5.1 Testbed

We selected 300 PlanetLab nodes with the shortest `ssh` connection times, and provisioned a User SG and Replica SG on each. The User SG leveraged a local Squid [54] cache to store recently-read block and manifest data. We varied the behavior of the Replica SG depending on what we needed to measure.

We built a CDN from Squid [54] caches on top of 40 VICCI [66] nodes distributed evenly across the four North American sites. We used the Aura request router [8] (based on [67]) with a variant of consistent hashing to distribute a client request to one of three VICCI nodes, based on the client IP address.

The PlanetLab caches were configured to use 256MB of RAM and 1GB of disk. The VICCI caches were configured to use 1GB of RAM and 8GB of disk. These sizes were chosen to be big enough to store our working set, so as to consistently measure the performance of hitting partially-cached object data. In addition, PlanetLab nodes were limited to 10Mbps maximum bandwidth and 10GB maximum data sent per day.

We set up the MS with the configuration we plan to offer users (others are possible, at different price-points). Each MS instance in Google AppEngine had a 1GHz virtual CPU and 512MB RAM. We used the High Replication Datastore (HRD) for storing and replicating MS records (where individual record replication is coordinated via Paxos [34]). We allocated 50 static instances, and allowed Google to start more automatically.

5.2 Metadata Operations

While a scalable number of SGs read and write a scalable number objects, each SG contends with an interactive workload. Since each SG communicates with the MS as part of its control plane, the choice of MS platform will influence Syndicate's overhead.

To understand how, we measured the time taken by a metadata operation between the call into the MS client library and its return. This includes the added latency from the MS platform, which we measure indirectly by measuring the time taken by our MS code to process a request.

We simulated a set of peer SGs managing a peer-specific directory of peer-specific objects, which is a common access pattern for our sample applications. We first had each SG create a top-level directory in the Volume ("Mkdir"), forcing the MS to serialize many directory writes at once. We then had each SG create an object within this directory ("Create"), where no serialization is required. Afterwards, each SG wrote new metadata for the object ("Update") and then redownloaded all metadata for its path ("Revalidate"). Finally, each SG deleted the object ("Unlink") and then its parent directory ("Rmdir").

We started each SGs in a staggered fashion, so as to maximize churn in the MS. Each SG used a keep-alive HTTPS connection to issue the requests; the latency of the initial SSL negotiation is omitted (since it is noticed only when the SG registers with the MS). The runtimes

are summarized in Figure 3.

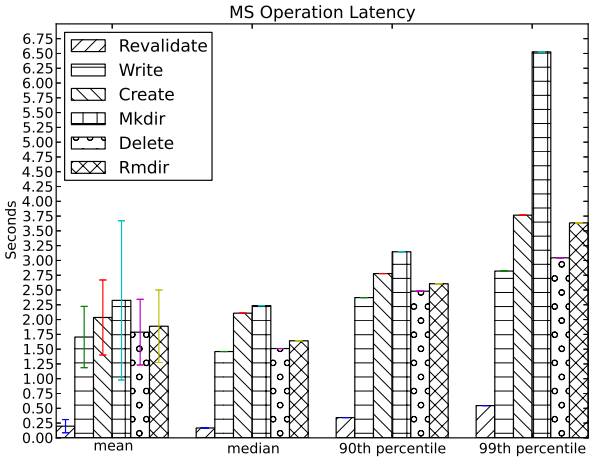

Figure 3: *Metadata operation times (in seconds) across 300 PlanetLab nodes. Error bars are one standard deviation.*

In addition, we recorded Google AppEngine's self-reported performance for the API calls we leverage [29]. Their performances, as well the number of times each MS operation uses them, are summarized in Table 3. Note that a datastore `get` is called whenever a memcache `get` fails.

Using the AppStats [7] middleware, we confirmed that each operation ran at least as slowly as the minimum times indicated by GAE's self-reporting. Additionally, we observed that most of the latency of each operation (particularly higher percentiles) was due to GAE buffering our requests internally—GAE holds a request for up to 10s before allowing a tenant to process it. The reason "Revalidate" operations are so short relative to the rest is because most of the time, it only interacts with `memcache`, making it a much faster operation.

While we can improve MS performance by using a lower-latency platform, making namespace updates atomic in "Create" is nevertheless a costly but necessary operation. Regardless, the MS is able to deliver metadata to readers quickly enough for interactive workloads even under degenerate conditions– -when the SG has not cached any metadata, when there is a high buffering time in the provider, and when there is a lot of churn in the directory structure. This is adequate for our sample applications.

5.3 Reads and Consistency

To evaluate Syndicate's overhead for reading consistent data, we had each VM create and then read a 6MB object comprised of 100 60KB blocks. We chose this object size to make it sufficiently large to merit distribution by CDN, and chose 100 blocks to generate a large number of GET requests at once (30,000 blocks and 300 manifest requests, or about 2GB of data, per experiment). In the context of our sample applications, a single object represents a piece of experimental data, a large document, or a VM's session information.

In these experiments, object data was stored on local disk on each VM and and read sequentially by the User SG. Due to its similarity to the Replica and Acquisition SGs in processing reads (beyond evaluating a specific R function), the insights into its overhead extend to understanding the the other two SGs' read overheads.

Read Latency. To understand Syndicate's overhead for accessing data, we measured read latency for reading the object sequentially, from both local and remote SGs. We defined latency as the wall-clock time between entering the call to `read()` and the receipt of the first byte of object data in the SG, in order to account for all the overhead of preparing to receive data (namely, revalidating the object metadata and downloading a fresh manifest). To ensure the SG did not falsely report zero latency on remote reads, we used cold caches and had each SG choose a remote object at random to read.

The local read latencies for the SGs are summarized in Figure 4, and the remote read latencies in Figure 5. In the local case, there is no appreciable latency difference between reading local data through Syndicate ("Read, local object") and reading 100 60KB local files directly ("Read, disk"). Almost all of the overhead comes from revalidating the object's metadata first ("Read, local object + Revalidate").

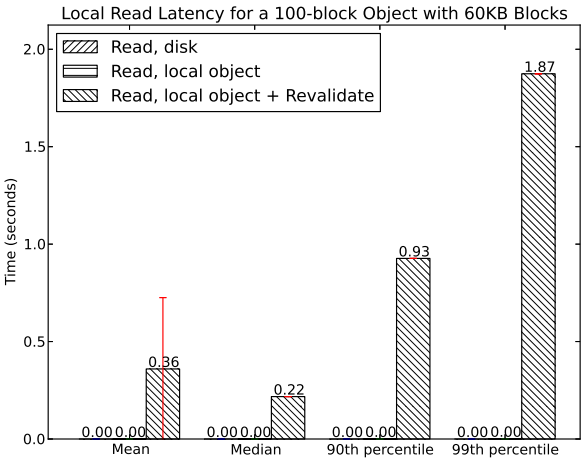

Figure 4: *Read latencies on a local object's data. Error bars are one standard deviation.*

In the remote case, the SG first downloaded the man-

Operation	Min (ms)	Max (ms)	Create/Mkdir	Revalidate	Update	Unlink/Rmdir
datastore `get`	35	65	0-2	0-1	0-1	0-2
datastore `put`	45	70	1	0	1	3
datastore `update`	50	120	1	0	0	0
datastore `query`	80	150	0	0-1	0	0-1
memcache `get`	1	10	4	2	2	3
memcache `set`	2	10	4	2	3	6
HTTPS `GET`	25	100	1	1	1	1

Table 3: *Approximate (self-reported) min/max latency from Google AppEngine on API calls, and the numbers of each API method synchronously called by each Syndicate metadata operation. The* `update` *operation is a transaction of one* `get` *and one* `put`.

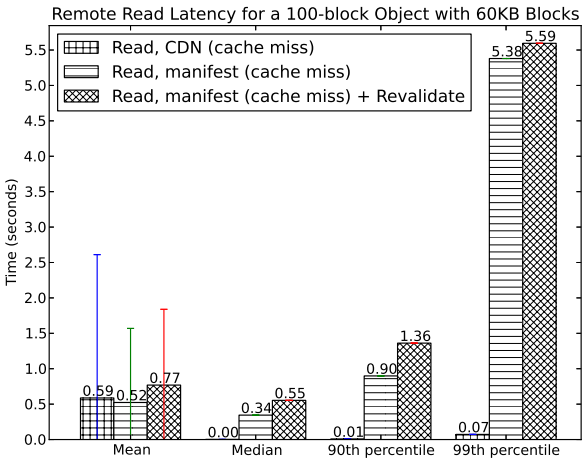

Figure 5: *Read latencies on a remote object's data. Error bars are one standard deviation.*

ifest before reading the object data. We compare the latency added by downloading the manifest only ("Read, manifest (cache miss)") to the latency added by downloading the manifest and also refreshing the object metadata ("Read, manifest (cache miss) + Revalidate"). As a baseline comparison, we additionally measured latency added just by contacting the CDN for object data ("Read, CDN (cache miss)").

We see that the latency added by the CDN is very small—70ms in the 99th percentile—and that most of the overall latency comes from fetching the manifest on cache miss. However, a local SG evaluation shows that it can generate and parse responses for manifest requests much more quickly than this test indicates, suggesting that the underlying infrastructure—the wide-area network and PlanetLab VMs—contributes most of the latency, and that replacing it with faster infrastructure would reduce it.

Read Times. We measured the read time for the object on each SG as the wall-clock time elapsed between receiving the first and last byte of data, after the manifest and metadata had been downloaded. We used both cold and warm caches on remote reads, to compare the worst-case performance (local and CDN cache miss) to the best-case (local and CDN cache hit). To serve as a baseline comparison for remote reads in both cases, we used the `curl` program to download the same 100 60KB blocks. With both `curl` and the SG, each block was fetched sequentially and as separate requests.

The results are summarized in Figure 6 (note the logarithmic time axis). Unsurprisingly, reading all 100 60KB blocks of a local object on the SG ("Read, local object") and directly reading them ("Read, disk") are comparable, with Syndicate adding 30ms extra time in the 99th percentile.

However, the SG was consistently faster than `curl` when reading locally-cached block data ("Read, CDN (cache hit)" versus "Read, Remote Object (cache hit)"). This was due to the slowdown added by running the `curl` program once for each block, whereas the SG used `libcurl` to sequentially fetch blocks and thus benefitted from preserving library state across requests. Specifically, most of the difference was from reusing cached DNS queries.

This time difference is also visible when reading block data through cold caches ("Read, CDN (cache miss)" versus "Read, remote object (cache miss)"), and is consistent with the difference observed when reading cached data. This suggests that read bandwidth of the SG versus `curl` is otherwise comparable, and indicates that the CDN was the limiting factor to read times.

Reads with Writes. To demonstrate the overall effect of reading data after it is written, we had 299 SGs concurrently read the 300th SG's object after a controlled percentage of its blocks were overwritten. After a write, each remote SG downloaded the new metadata and manifest, and sequentially read all of the object's blocks from warm caches. We warmed the caches prior to the first write by having each SG read the object's blocks. We also disabled cache control directives, requiring the local caches and CDN to keep data indefinitely.

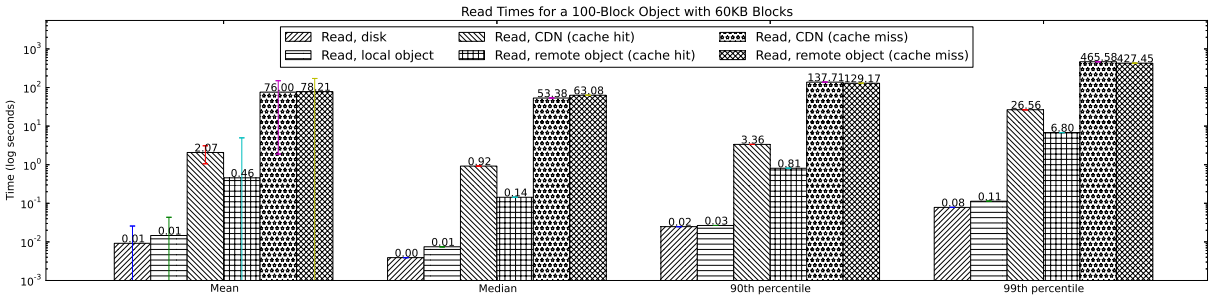

Figure 6: *Read times of Syndicate, as compared against reading blocks directly from disk and directly from the CDN (on both cache hit and miss). Note that the Y axis is in log-scale. Error bars are one standard deviation. Numbers on each bar are in seconds.*

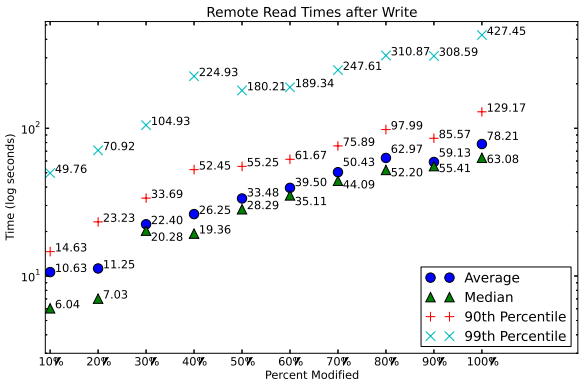

Figure 7: *Times taken to read a remote object after a given percent of its blocks were modified. Note that the Y axis is in log-scale.*

We overwrote the object in 10% increments, and measured the total time to read the object as the wall-clock time between the application calling `read()` and `read()` returning. The times are summarized in Figure 7.

Predictably, there is a roughly linear correlation between the fraction of modified blocks and the total read time, in all percentiles. This validates the benefit of breaking objects into chunks in the storage layer, and managing consistency separately from data locality—readers hit unmodified data in the cache and redownload modified data automatically, without needing cache control directives.

5.4 Writes and Consistency

To measure how much overhead writes introduce, we measured the time taken to write all the object data to local disk ("Write, local object"), as well as the time taken to additionally send a new last-write nonce to the MS ("Write + MS update"). We compared both scenarios to simply writing 100 60KB files to disk ("Write, disk"). The results are summarized in Figure 8.

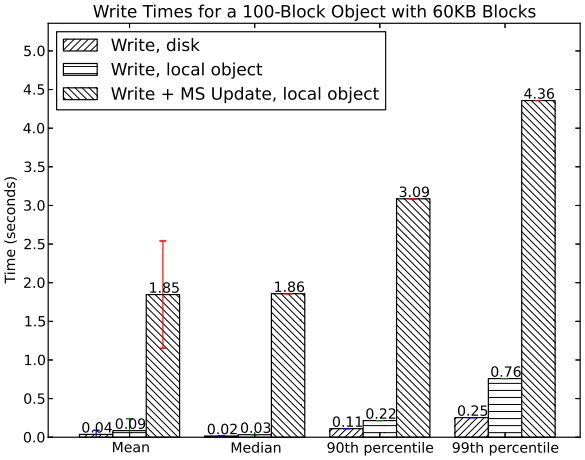

Figure 8: *Times taken to write a local object's data. Error bars are one standard deviation.*

In comparing writing directly to disk to writing to the SG, we observe that Syndicate adds a 3x write overhead in the 99th percentile, but a 1.5x overhead in the 50th percentile. This is due to two factors: keeping the manifest up-to-date with written data, and preserving some extra state on disk to track the last-write nonce. Both of these routines are not yet optimized in our SG prototype. Nevertheless, the cost of updating the last-write nonce on the MS dominates the cost to write this object.

5.5 Discussion

While this is a preliminary evaluation, we believe it provides enough information to demonstrate that Syndicate not only works, but also that our example applications

(and others) could benefit by using it in read-heavy but read-write interactive workloads.

One take-away is that the choice of infrastructure plays a big role in determining Syndicate's performance. This is by design, because the performance benefits a provider can offer should be passed on to the application whenever possible.

We are in the process of deploying Syndicate as a permanent service in PlanetLab, such that users will automatically receive one Volume per slice, and each VM will automatically install and mount our FUSE SG. Our full running system will leverage a private instance of the CoBlitz [45] CDN deployed on VICCI. We will maintain a set of Replica SGs for popular cloud storage providers, which users may choose when creating their Volumes.

6 Related Work

Syndicate adds to a growing body of work on leveraging existing storage systems, where the main contributions are in separating storage concerns into independent components, and applying the end-to-end principle [52] to move storage features to application end-points. Separating consistency has been exploited to provide causal consistency [9], ACID semantics [37], and key-group transactions [17, 25] on top of unmodified cloud storage. Separating access and admission control has been achieved with Kerberos [55], and is exploited today in Web applications with federated identity management [48, 59, 43] for single sign-on. Cloud storage customers enforce storage policies transparently using *cloud storage gateways*—on-site middleboxes that apply the policies locally (examples include [50, 3, 65]). Customers can also mount certain cloud storage as filesystems [19, 4], and then layer special-purpose filesystems on top of them that apply storage policies transparently (such as encryption [20]).

Unlike prior work, Syndicate separates all of these concerns in a coherent manner. By applying our consistency protocol to both data and control plane information. we leverage edge caches to distribute object data, process certificates, and install end-point functionality in a scalable, secure way.

Many peer-to-peer storage systems [49, 58, 16, 39, 56, 51, 61], caching systems [22, 33, 18, 12], and distributed filesystems [32, 68, 69, 57, 2, 36, 6, 44] have preceded Syndicate, and address consistency, security, and storage policy in different contexts while offering similar feature sets. Syndicate differs from these systems in two ways: its approach to systems building, and how it leverages caches. Whereas prior works focus on combining resource-constrained hosts to form scalable storage systems, Syndicate focuses on a general way of combining scalable storage systems into virtual storage systems for cloud-hosted applications. Whereas prior works relied on system-specific caches and used client/cache polling, object leases, and cache invalidation protocols (such as [60, 31, 38, 30]) to scalably deliver consistent data to readers (if they used them at all), Syndicate exploits already-deployed, unmodified Web caches to obtain a scalable data and control plane. In doing so, Syndicate complements prior and future caching and storage systems by letting applications use any of them transparently as underlying providers.

Web thin clients (such as [26]) combine local and cloud storage to reduce operating costs. However, they are vertically integrated, making it difficult to use infrastructure beyond what the vender supports.

7 Conclusion

This paper presents Syndicate, a wide-area storage service that is unique in how it composes existing cloud storage, edge caches, and local storage to take advantage of the unique capabilities each offers. By addressing consistency, storage security, and storage policies as a layer on top of them, Syndicate offers applications a virtual cloud storage service with the underlying capabilities of leveraged components. The key to doing so coherently is our novel consistency protocol, which leverages existing edge caches to deliver consistent data and control plane information to a scalable number of end-points. Our prototype implementation validates the design, and demonstrates that Syndicate passes on the benefits of underlying systems to applications, minus a small, configurable overhead.

By interpositioning Syndicate gateways between the key participants and manipulating the underlying transport protocol (HTTP) and naming scheme (URIs), Syndicate is able to transparently leverage those existing services without modification, and in doing so, gives users the opportunity to select among alternatives (based on their own criteria, including cost), and yet benefit from the wide-spread deployment and ongoing operational support they provide (freeing them from deploying and operating a global service on their own). In doing so, we democratize cloud storage by enabling user choice and lowering the barrier to entry.

Beyond the specifics of wide-area storage, we believe Syndicate represents an emerging style of building network system services through the composition of existing cloud services, and in doing so, has the potential to encourage a more efficient marketplace for them.

References

[1] 1000 Genomes: A Deep Catalog of Human Genetic Variation. `http://www.1000genomes.org/`.

[2] ADYA, A., BOLOSKY, W. J., CASTRO, M., CHAIKEN, R., CERMAK, G., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. P. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *PROCEEDINGS OF THE 5TH USENIX SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION* (2002), USENIX.

[3] Amazon Gateway. `http://aws.amazon.com/storagegateway/`.

[4] FUSE-based Filesystem backed by Amazon S3. `https://code.google.com/p/s3fs/`.

[5] Overview of Amazon Web Services. `https://d36cz9buwru1tt.cloudfront.net/AWS_Overview.pdf`.

[6] ANNAPUREDDY, S., FREEDMAN, M. J., AND MAZIERES, D. Shark: Scaling file servers via cooperative caching. In *Proc. 2nd NSDI* (Boston, MA, 2005).

[7] Appstats for Python. `https://developers.google.com/appengine/docs/python/tools/appstats`.

[8] Aura Lumen: Request Router. `http://www.akamai.com/dl/product_briefs/Aura_Lumen_Request_Router.pdf`.

[9] BAILIS, P., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, ACM, pp. 761–772.

[10] BLOMER, J., BUNCIC, P., AND FUHRMANN, T. Cernvm-fs: delivering scientific software to globally distributed computing resources. In *Proceedings of the first international workshop on Network-aware data management* (New York, NY, USA, 2011), NDM '11, ACM, pp. 49–56.

[11] Box.net. `http://www.box.net/`.

[12] BRESLAU, L., CAO, P., FAN, L., PHILLIPS, G., AND SHENKER, S. Web caching and zipf-like distributions: evidence and implications. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* (1999), vol. 1, pp. 126–134 vol.1.

[13] CHANDRA, R., ZELDOVICH, N., SAPUNTZAKIS, C., AND LAM, M. 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).

[14] Citrix VDI-in-a-Box. `http://www.citrix.com/products/vdi-in-a-box/overview.html`.

[15] Common Crawl. `https://drive.google.com/`.

[16] DAHLIN, M., GAO, L., NAYATE, A., VENKATARAMANI, A., YALAGANDULA, P., AND ZHENG, J. Practi replication for large-scale systems. Tech. rep., 2004.

[17] DAS, S., AGRAWAL, D., AND EL ABBADI, A. G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 163–174.

[18] DILLEY, J., MAGGS, B., PARIKH, J., PROKOP, H., AND WEIHL, B. Globally distributed content delivery. *IEEE Internet Computing 6* (2002), 50–58.

[19] Dropbox. `http://www.dropbox.com/`.

[20] EncFS Encrypted Filesystem. `http://www.arg0.net/encfs`.

[21] Filesystems in Userspace. `http://fuse.sourceforge.net`.

[22] FREEDMAN, M., AND MAZIRES, D. Sloppy hashing and self-organizing clusters. In *Peer-to-Peer Systems II*, M. Kaashoek and I. Stoica, Eds., vol. 2735 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 45–55.

[23] FREEDMAN, M. J. Experiences with CoralCDN: A Five-Year Operational View. In *Proc. 2nd NSDI* (Boston, MA, 2005).

[24] GenBank. `http://www.ncbi.nlm.nih.gov/genbank/`.

[25] GLENDENNING, L., BESCHASTNIKH, I., KRISHNAMURTHY, A., AND ANDERSON, T. Scalable consistency in scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 15–28.

[26] Google Chrome OS. `http://www.chromium.org/chromium-os`.

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

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

[29] Google AppEngine System Status. `https://code.google.com/status/appengine`.

[30] GRAY, C., AND CHERITON, D. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. *SIGOPS Oper. Syst. Rev. 23*, 5 (Nov. 1989), 202–210.

[31] GWERTZMAN, J., AND SELTZER, M. World-wide web cache consistency. In *PROCEEDINGS OF THE 1996 USENIX TECHNICAL CONFERENCE* (1996), pp. 141–151.

[32] HOWARD, J. H. An overview of the andrew file system. In *Proc. USENIX Winter Technical Conference '88* (Dallas, TX, 1988).

[33] KARGER, D., SHERMAN, A., BERKHEIMER, A., BOGSTAD, B., DHANIDINA, R., IWAMOTO, K., KIM, B., MATKINS, L., AND YERUSHALMI, Y. Web caching with consistent hashing. *Computer Networks 31*, 1116 (1999), 1203 – 1213.

[34] LAMPORT, L. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32*, 4 (Dec. 2001), 51–58.

[35] LARSON, S. M., SNOW, C. D., SHIRTS, M., P, V. S., AND PANDE, V. S. Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology.

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

[37] LEVANDOSKI, J. J., LOMET, D. B., MOKBEL, M. F., AND ZHAO, K. Deuteronomy: Transaction support for cloud data. In *CIDR* (2011), www.cidrdb.org, pp. 123–133.

[38] LIU, C., AND CAO, P. Maintaining strong cache consistency in the world-wide web. In *Distributed Computing Systems, 1997., Proceedings of the 17th International Conference on* (1997), pp. 12–21.

[39] MAYMOUNKOV, P., AND MAZIÈRES, D. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems* (London, UK, UK, 2002), IPTPS '01, Springer-Verlag, pp. 53–65.

[40] Measurement Lab. `http://www.measurementlab.net/`.

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

[42] Mokafive. `http://www.mokafive.com`.

[43] MORGAN, R. L., CANTOR, S., CARMODY, S., HOEHN, W., AND KLINGENSTEIN, K. Federated security: The shibboleth approach. *EDUCAUSE Quarterly 27*, 4 (2004), 12–17.

[44] MUTHITACHAROEN, A., MORRIS, R., GIL, T. M., AND CHEN, B. Ivy: A read/write peer-to-peer file system. In *PROCEEDINGS OF THE 5TH USENIX SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION* (2002), USENIX.

[45] PARK, K. S., AND PAI, V. Scale and Performance in the CoBlitz Large-File Distribution Service. In *Proc. 3rd NSDI* (San Jose, CA, 2006).

[46] PETERSON, L., BAVIER, A., FIUCZYNSKI, M., AND MUIR, S. Experiences Building PlanetLab. In *Proc. 7th Operating System Design and Implementation (OSDI)* (Seattle, Washington, Nov. 2006).

[47] RAJASEKAR, A., MOORE, R., HOU, C.-Y., LEE, C. A., MARCIANO, R., DE TORCY, A., WAN, M., SCHROEDER, W., CHEN, S.-Y., GILBERT, L., TOOBY, P., AND ZHU, B. *iRODS Primer: integrated Rule-Oriented Data System*. Morgan and Claypool Publishers, 2010.

[48] RECORDON, D., AND REED, D. Openid 2.0: a platform for user-centric identity management. In *Proceedings of the second ACM workshop on Digital identity management* (New York, NY, USA, 2006), DIM '06, ACM, pp. 11–16.

[49] RHEA, S., EATON, P., GEELS, D., WEATHERSPOON, H., ZHAO, B., AND KUBIATOWICZ, J. Pond: the oceanstore prototype. In *PROCEEDINGS OF THE 2ND USENIX CONFERENCE ON FILE AND STORAGE TECHNOLOGIES* (2003), USENIX.

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

[51] ROWSTRON, A. I. T., AND DRUSCHEL, P. 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* (London, UK, UK, 2001), Middleware '01, Springer-Verlag, pp. 329–350.

[52] SALTZER, J. H., REED, D. P., AND CLARK, D. D. End-to-end arguments in system design, 1984.

[53] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (Washington, DC, USA, 2010), MSST '10, IEEE Computer Society, pp. 1–10.

[54] Squid web proxy. `http://www.squid-cache.org/`.

[55] STEINER, J. G., NEUMAN, C., AND SCHILLER, J. I. Kerberos: An authentication service for open network systems. In *in Usenix Conference Proceedings* (1988), pp. 191–202.

[56] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2001), SIGCOMM '01, ACM, pp. 149–160.

[57] STRIBLING, J., SOVRAN, Y., ZHANG, I., PRETZER, X., LI, J., KAASHOEK, M. F., AND MORRIS, R. Flexible, wide-area storage for distributed systems with wheelfs. In *Proc. 6th NSDI* (Boston, MA, 2009).

[58] TERRY, D., THEIMER, M., PETERSEN, K., DEMERS, A., SPREITZER, M., AND HAUSER, C. Managing update conflicts in bayou, a weakly connected replicated storage system. In *In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (1995), pp. 172–183.

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

[60] THIRUMALE, R. T., TEWARI, R., NIRANJAN, T., AND RAMAMURTHY, S. Wcdp: A protocol for web cache consistency.

[61] TOLIA, N., KAMINSKY, M., ANDERSEN, D. G., AND PATIL, S. An architecture for internet data transfer. In *In Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI* (2006), pp. 253–266.

[62] TORRES-ROJAS, F. J., AHAMAD, M., AND RAYNAL, M. Timed consistency for shared distributed objects, 1999.

[63] TORVALDS, L. git. `http://git-scm.com/`.

[64] Tree of Life Web Project. `http://tolweb.org`.

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

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

[67] WANG, L., PAI, V., AND PETERSON, L. The effectiveness of request redirection on cdn robustness. *SIGOPS Oper. Syst. Rev. 36*, SI (Dec. 2002), 345–360.

[68] WANG, R., WANG, O. Y., AND ANDERSON, T. E. xfs: A wide area mass storage file system. In *In Fourth Workshop on Workstation Operating Systems* (1993), pp. 71–78.

[69] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proc. 7th OSDI* (Seattle, WA, 2006).