Syndicate: A Scalable, Read/Write File Store for Edge Applications

Jude Nelson (student)
Princeton University
jcnelson@cs.princeton.edu

Larry Peterson
Princeton University
llp@cs.princeton.edu

Applications in middle and last mile networks (the "edge") that make use of local storage, network caches, and datacenter storage must consider availability, durability, performance, cost, and consistency requirements in their design. We present Syndicate, a wide-area read/write file store that addresses these concerns.

To illustrate them, suppose a physicist PI and her collaborators host experimental data on her university's file server. As research progresses, she discovers she needs higher data availability and durability, so she uploads the important datasets to cloud storage. While her collaborators may still read and edit them, she now pays for hosting and data transfer.

Later, the group starts to use off-site grid computers to regularly download and process the data. To decrease latency and transfer costs and increase bandwidth and availability, the PI employs network caches to scale up the number of concurrent downloads. However, depending on caching policy, remote readers may get stale data well after a modification, causing the collaborators to suffer invalid results.

This example offers four key insights. First, using network caches for remote reads improves availability and amortized performance regardless of where the data is hosted. By using network caches, the collaborators may store their data wherever is best for them; only cache misses affect read performance.

Second, durability only needs to be considered on writes. When a collaborator commits new experimental data, he chooses how many replicas to make, and where to put them, to achieve a desired durability. This choice is specific to the dataset, and is a matter of policy (e.g. durability, cost, etc.), not implementation.

Third, remote readers cannot rely on caches for consistency. Even though many caches today offer support for object TTLs and refresh requests (i.e. via HTTP directives), the cache has its own cost and performance objectives, and may choose to ignore directives to meet them. For example, a cache could keep an object resident beyond its TTL to reduce the cost and performance penalties of frequent revalidation. Moreover, because caches can be transparent, the collaborators and grid computers can neither reliably control nor predict caching policy.

Fourth, widely-deployed storage and caching infrastructure are almost good enough. Rather than modifying the infrastructure, the collaborators address these concerns out-of-band (e.g. storage conventions on a wiki).

From these insights, we derive Syndicate. Syndicate organizes data into a filesystem (a *Volume*) and addresses consistency by treating each version of each block of each file as a read-only cacheable object with a version-specific URL. Applications control how often to check for new versions on reads, and how often to publish new versions on writes. Readers and writers contact a scalable Metadata Service (MS) in the cloud to synchronize Volume metadata and version records. In doing so, Syndicate decouples caching from consistency, whereby applications, not caches, pay for freshness.

Applications access data through local User Gateways (UGs). The UG performs the aforementioned metadata synchronization, and reads remote data for applications via network caches. It hosts written data on local storage, and synchronously replicates it to an application-defined quorum of Replica Gateways (RGs), which mediate access to cloud storage. This decouples performance and read availability from durability. Volume administrators (i.e. the PI) enforce replication and access control policy by binding UGs and RGs to Volumes, thereby decoupling storage policy from implementation.

With sufficient permission, any application may read or write any file. The UGs and MS coordinate to enforce access control and ensure that any reads after a write return the latest data after an application-given deadline.

We implemented the UG as a FUSE filesystem, the RG as a cloud-hosted process, and the MS as a Google AppEngine service. We will present a demo, where the RG leverages Amazon S3 for cloud storage and the UG leverages the CoBlitz CDN and Squid for network caches.