POSTED ON OCTOBER 7, 2019 TO CORE INFRA, DATA INFRASTRUCTURE

Scribe: Transporting petabytes per hour via a distributed, buffered queueing system

By Manolis Karpathiotakis, Dino Wernli, Milos Stojanovic

Our hardware infrastructure comprises millions of machines, all of which generate logs that we need to process, store, and serve. The total size of these logs is several petabytes every hour. The outputs are generally processed somewhere other than where they were generated: They can be relevant to a variety of downstream processing pipelines and may each need to be accessed at different times. The task of collecting, aggregating, and delivering this volume of logs (with low latency and high throughput) requires a systematic approach. Our solution is Scribe, a distributed queueing system that encapsulates all the complexity behind moving service logs from point A to point B.

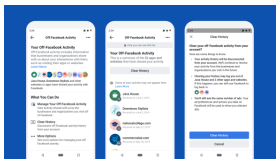
Scribe processes logs with an input rate that can exceed 2.5 terabytes per second and an output rate that can exceed 7 terabytes per second. To put this workload into perspective, the output of the Large Hadron Collider at CERN during its latest run was estimated to reach only 25 gigabytes per second. Scribe has recently undergone a major architectural revamp and simplification, and its new architecture is currently in production. We are sharing for the first time Scribe's current design, as well as the factors that led to this current architecture and how our scale and evolution have influenced it over the past decade.

Scribe: A general-purpose, buffered queueing system

Our ecosystem involves a diverse range of log generation scenarios. A typical example is that of web servers generating semi-structured logs indicating their health. In the most general case, developers want to transport the unstructured contents of a static file to a downstream system. That downstream system is generally one of our analysis tools. The typical expectation is that a developer should be able to perform analysis and exploration over a collection of logs. Depending on the use case, they might want to observe real-time trends or historical patterns. Logs are thus sent to the data warehouse for historical analysis, or to real-time stream processing systems, such as Puma, Stylus, and Scuba, for real-time exploration.

Related Posts

Sep 25, 2019
Networking @Scale 2019 recap



Aug 20, 2019
Redesigning our systems to provide more control over Off-Facebook activity

May 28, 2019
Extending DHCPLB: The path from load balancer to server

Related Positions

Software Engineer, Machine Learning
SINGAPORE

Software Engineer, Machine Learning
SUNNYVALE, US

Software Engineer, Machine Learning
REMOTE, US

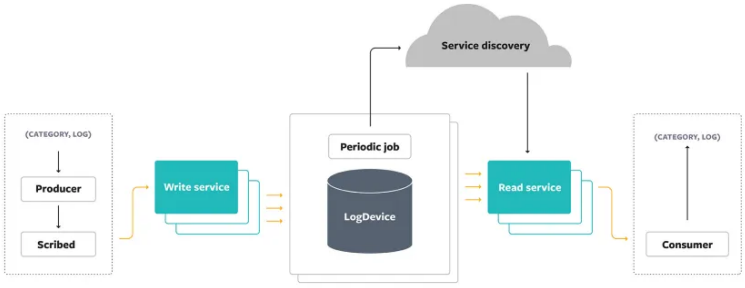
Software Engineer, Machine Learning
BELLEVUE, US

Software Engineer, Machine Learning
REDMOND, US

See All Jobs

Transporting a log

Scribe allows its users to write a payload to a named logical stream (called a category) from any of our machines. The payloads can vary widely in size, structure, and format, yet Scribe treats them in the same way. Each log is stored durably for a configurable retention period — typically a few days. During this period, consumers can read the stream of logs sequentially, an operation referred to as tailing.



A high-level view of Scribe's current architecture.

Any process can use a Producer library to write a log to Scribe. The Producer could belong to applications running in containers, such as one of the web servers executing Hack code (a dialect of PHP). An engineer using Scribe can instruct the Producer to take one of the following paths:

- Write to a local daemon (Scribed) in charge of eventually sending the log to storage back-ends
- Write to a remote tier of back-end servers (Write Service) directly

Finally, consumers can read logs in streaming fashion by contacting a Read Service.

Scribe's first incarnation was really an early version of Scribed — essentially a NetApp filer — that was responsible for persisting logs to network-attached storage drives. Although Scribe has been an always-available system without experiencing any downtime, its architecture has evolved over time into a more sophisticated one and now comprises more than 40 distinct components. The growing number of complex components made it difficult to retain an open source version stripped of our internal specifications. This complexity is the main reason that we archived the open source Scribe project. The current version of Scribe's architecture is detailed below, with a focus on the components that comprise the data plane.

Scribed

Most of our machines run a daemon process called Scribed. Customers can directly write logs to Scribed if their primary concern is to offload ownership of logs as quickly as possible. Scribed uses local disk (in addition to memory) as a buffer for situations in which the machine has lost connectivity or back-ends are otherwise unavailable. Writing logs directly to Scribed works well for the majority of Scribe users. But communication with Scribed can have some drawbacks:

- Storing logs on disk can have latency and consistency implications.
- If Scribed is unavailable on a single machine, writes can get lost or blocked for a non-trivial amount of time.
- Scribed is a shared resource on the machine and users that attempt to write great amounts of logs can monopolize Scribed resources, which affects the experience of other users on the same machine.

To solve the rare cases in which users experience these drawbacks, we built the ability for users to bypass Scribed and instead instruct their Producer to write logs directly to the Scribe Write Service. Doing so can decrease the end-to-end latency and increase write availability, because the Producer can direct the write operation to multiple back-end hosts rather than to a single local process. On the other hand, write resilience can decrease: For example, if a Producer is unable to offload logs to the Write Service quickly

enough, then the Producer's in-memory queue can fill up, resulting in denial of the excess write requests and message loss.

Write Service

Eventually, every log makes it to one of Scribe's back-end servers. Scribe delegates the choice of a server to an internal load-balancing service, based on locality and availability. The choice is dynamic, and the fleet of back-end servers acts as a single, resilient Write Service, without a single point of failure. Since Scribe collects logs from millions of machines, logs from different machines can arrive at the Write Service in any order. In addition, logs from a single machine can arrive at several different Write Service machines and eventually be stored at several different storage back-end instances. These back-ends do not share a notion of global time or causality, and therefore Scribe makes no attempt to preserve the relative order of the logs it receives.

Instead, the Write Service focuses on batching incoming logs by category and forwarding the batches to storage back-ends. Storage is organized in clusters: Each storage cluster is a set of machines that host Scribe's storage components. The Write Service selects which storage cluster to place each log batch in. The aim of the Write Service is to place related logs (e.g., the same category) in a subset of clusters, which strikes a balance between being large enough to ensure write availability and small enough to ensure read availability. Additionally, placement decisions take other factors into consideration, such as which geographic region we expect the logs to be read from.

Once the storage cluster has been selected, the Write Service forwards the log. Up until the point that a log reaches durable storage in a cluster, the log has generally been residing in the memory of a process and has thus been susceptible to a number of failure scenarios that can lead to its loss; tolerating a minimal amount of loss was one of the design choices we made in order to avoid overheads that would affect its high-throughput, low-latency offering. Users that require even more rigid delivery guarantees can opt for a configuration of Scribe that trades performance for further reduced loss.

Buffered storage in LogDevice

The storage back-end of Scribe is LogDevice, a distributed store for logs, which offers high durability and availability under a variety of workloads and failure scenarios. Scribe stores each log as a LogDevice record. By relying on LogDevice for durable storage, Scribe can operate as if there were a single (logical) copy for each record: The LogDevice layer hides the complexity of operations such as record replication and recovery.

LogDevice organizes records into sequences called partitions. A Scribe category is backed by multiple partitions spread across multiple LogDevice clusters. Once LogDevice persistently stores a record in a partition, the record becomes available for reading. Again, note that Scribe retains records in LogDevice for a limited amount of time — typically a few days. Scribe aims to buffer records for a period of time that is sufficient for customers to consume it or replay records from it in case of failures. Customers that require retention for longer than a few days typically push their logs to long-term storage.

Scribe had been relying on HDFS for its storage back-end before migrating to LogDevice. Scribe eventually reached the scalability limits of HDFS — around the same time that Facebook was deprecating its use of HDFS. The transition to a LogDevice-based back-end brought Scribe to its current form and allowed it to efficiently serve additional use cases.

Reading a log

Customers can read logs written to Scribe by contacting a Read Service, which provides read access to streams of logs. The implementation of the Read Service is based on streaming Thrift and follows the reactive streams specification. In addition to generic access to log streams, the Read Service also acts as a load-balancing layer, allowing Scribe to serve multitenant read use cases while sharing resources (e.g., connections to storage back-ends).

Reading logs from Scribe via the Read Service involves identifying the clusters that contain logs for the requested category and reading the corresponding LogDevice partitions. The contents from each partition are merged together and partially ordered

by storage timestamp. Given that a consumer is reading logs from multiple clusters and that logs generated in a single client host can end up in multiple Scribe clusters, the consumer avoids enforcing a strict output order. Instead, the consumer applies “rough” ordering, ensuring that output logs are within N minutes of one another (where N is typically 30) in terms of the time when the logs reached persistent storage; customers that require logs to be flushed in the precise order of their creation typically use a stream processing system such as Stylus to read logs from Scribe and order them further.

Design decisions

Scribe exposes itself as a Thrift service that runs on a host machine and collects logs streamed in real time from clients. Given its intended ubiquity, it has been designed based on a number of first-class requirements. Specifically, the Scribe service must be:

1. **Simple,** exposing a straightforward API for customers to write and read logs.
2. **(Write) Available,** tolerating failures at any part of the transport process, while acknowledging that small amounts of logs may get lost in the process.
3. **Scalable,** handling millions of producers, whose aggregate input rate can exceed 2.5 terabytes per second, and hundreds of thousands of consumers, whose aggregate output rate can exceed 7 terabytes per second.
4. **Multitenant,** ensuring that customers can multiplex over the shared Scribe medium without each customer affecting the service quality of others.

Simplicity

Scribe exposes a high-level API for users to write or read logs — a Producer and a Consumer API, respectively. The Producer API has bindings for multiple programming languages and consists of a single write method. The Consumer API exposes a consumer object, which can be used to read streams of logs from Scribe as well as to perform additional operations, such as obtaining checkpoints.

Besides the high-level APIs, Scribe offers convenience binaries that are available in our hosts. Customers that want to write logs in Scribe can use a command such as the following:

```
# scribe_cat $CATEGORY $PAYLOAD
```

Regarding reading logs from Scribe, customers can use the following command to dynamically create a read stream to tail a category’s log for a given time period:

```
# ptail --since $ts1 --till $ts2 $CATEGORY | consumerApp
```

Availability

The main concern of Scribe is ensuring that logs written to it manage to reach the persistent LogDevice storage, despite the presence of multiple failure types. Starting from the “edge” of Scribe, Producer instances buffer logs in memory to handle brief periods of Scribed unavailability (e.g., Scribed restarts for update purposes). In similar fashion, Scribed buffers logs on a local disk to handle periods of network outages that prevent it from successfully sending logs to the Write Service. LogDevice further reinforces write availability by persisting multiple copies of each log record it receives. The combination of these features enables Scribe to successfully serve the vast majority of write calls.

The read path of Scribe is designed to efficiently serve logs in the context of straggling storage hosts, racks, and clusters. Specifically, given the high fan-in of storage clusters in which Scribe logs end up, a read stream typically fetches logs from multiple clusters that are spread across the world. With logs spread across multiple clusters, Scribe needs to be able to handle the case where a cluster is unable to serve readers fast enough or at all.

Scribe handles situations of temporary cluster unavailability by allowing users to relax the delivery and ordering guarantees of output logs. In practice, this means readers can

choose whether they want to wait for unavailable logs or proceed without it.

Scalability

Scribe has to accommodate ever-growing amounts of logs written to it. At the same time, the writing patterns can vary significantly, even within a single day. The physical storage of a Scribe category is spread across multiple LogDevice partitions, each of which can sustain a maximum write throughput. Scribe dynamically scales the number of partitions for a category depending on the volume that the partitions have been receiving. The end result of this partition provisioning process is the automatic handling of changes in write traffic, which allows Scribe throughput to elastically scale horizontally.

Regarding log reads, it can be nontrivial for a single consumer process to process and flush the huge volume of logs generated for a given category: The network card of the consumer host might become saturated, or the process consuming the consumer's output might be unable to process logs at the speed it is generated due to CPU limitations. Scribe thus has introduced buckets — a feature that enables multiple independent consumer processes to collectively consume the logs from a single Scribe category. Buckets are a sharding/indexing mechanism: They enable users to specify the number of streams in which they want Scribe to shard the inputs, and allow users to retrieve a specific shard via a given consumer. The Producer can explicitly provide a bucket number or leave it unspecified to have logs randomly distributed among available buckets.

Multitenancy

Scribe's users treat it as a Facebook-scale service: At any given point, millions of Producers write logs to hundreds of thousands of Scribe categories; hundreds of thousands of consumers then consume these logs. Still, customers expect their experience to remain unaffected by the other customers of the Scribe service. Scribe thus continuously monitors usage patterns in order to detect unexpected user behavior that could compromise the overall health of the system.

Scribe enforces rate limits on write operations in order to protect its back-ends. Specifically, Scribe associates a write quota with each category: Quota excesses can then result in Scribe block

listing a category (i.e., drop its newly incoming traffic), accepting a fraction of its logs, or applying backpressure to the Producers. In addition, Scribe detects cases in which customers read the contents of a category multiple times and, as a result, place increased stress on Scribe and its storage backend and increase the usage of cross-region bandwidth — a resource that is consumed aggressively due to the "write-anywhere" nature of Scribe.

Besides controlling demand, Scribe attempts to minimize the interactions with the Scribe service that could potentially require manual intervention. For example, failovers and outages are transparent to the end customers, customer write quotas are auto-expanding to accommodate organic growth, increases in write traffic automatically result in provisioning storage resources, and customers can use consumer-provided "checkpoints" to restart the reading process from where they left off.

The future evolution of Scribe

Scribe has grown organically for more than a decade, and its design has evolved along the way. Among the opportunities for continued refinement of Scribe is further simplification of the overall system, which would involve coalescing the numerous distinct components, as well as encapsulating Facebook-specific infrastructure in the form of optional plugins. In addition to such simplification efforts, we are working to extend Scribe with new features to support the changing needs of our customers: For example, we are currently evaluating the explicit exposure of different delivery guarantees, such as at-least-once and exactly-once. In addition, we have been enriching the Read Service with caching and filtering capabilities in order to optimize use cases for customers who need to read the same logs multiple times and/or are interested in filtering a category's structured logs based on their content. Finally, we have been working to ensure that every aspect of Scribe — from our sharding mechanism and placement policies to our disaster-recovery readiness — keeps pace with the volume of logs we process daily.

An interesting pattern that arose during the multiple rewrites of Scribe is that its API and the architectural decisions behind it started resembling those of other large-scale messaging systems. The main motivation for continuing to evolve Scribe, rather than migrating our use cases to one of those other systems, is Scribe's ability to self-manage, self-scale, and self-heal, even in the case of large-scale catastrophic failures. These requirements are rigid, given the multitenant offering of Scribe. Hence, these requirements are built into its design, each of its components, and the components' synergy. Other systems typically rely on third-party solutions for features such as auto-scaling and disaster recovery.

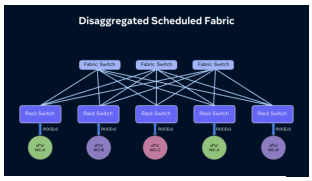
In summary, Scribe is a highly available system and service that has evolved on pace with our expanding tools and services over the past 12 years, becoming increasingly resilient. We are excited to see what the future holds for Scribe, and we are very eager to increase its scalability and fault tolerance even further, while we continue to simplify its implementation. We hope that these simplification efforts can pave the way toward open-sourcing Scribe again at some point in the future.

◂ Prev
Releasing a new benchmark and data set for evaluating neural code search models

Next ▸
Systems @Scale 2019 New York recap

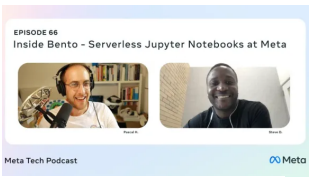

Read More in Data Infrastructure

View All ▸

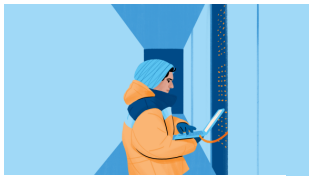


OCT 15, 2024
OCP Summit 2024: The open future of networking hardware for AI

OCT 15, 2024
Meta's open AI hardware vision

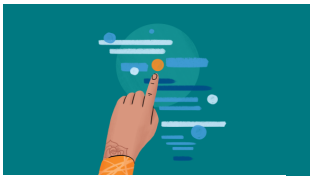


SEP 17, 2024
Inside Bento: Jupyter Notebooks at Meta



AUG 26, 2024
RETINAS: Real-Time Infrastructure Accounting for Sustainability

AUG 20, 2024
Aparna Ramani discusses the future of AI infrastructure



JUL 10, 2024
Meta's approach to machine learning prediction robustness

Software Engineer, Machine Learning

SINGAPORE

Software Engineer, Machine Learning

SUNNYVALE, US

Software Engineer, Machine Learning

REMOTE, US

Software Engineer, Machine Learning

BELLEVUE, US

Software Engineer, Machine Learning

REDMOND, US

See All Jobs

Engineering at Meta - X

Follow

AI at Meta

Read

Meta Quest Blog

Read

Meta for Developers

Read

Meta Bug Bounty

Learn more

RSS

Subscribe

Meta believes in building community through open source technology. Explore our latest projects in Artificial Intelligence, Data Infrastructure, Development Tools, Front End, Languages, Platforms, Security, Virtual Reality, and more.

ANDROID iOS WEB BACKEND HARDWARE

Learn More

Meta

Engineering at Meta is a technical news resource for engineers interested in how we solve large-scale technical challenges at Meta.

Home

Company Info

Careers