DistributedLog: A high performance replicated log service

Sijie Guo
sijieg@twitter.com
Twitter

Robin Dhamankar
robin.dhamankar@gmail.com
Twitter

Leigh Stewart
lstewart@twitter.com
Twitter

***Abstract*—DistributedLog is a high performance, strictly ordered, durably replicated log. It is multi-tenant, designed with a layered architecture that allows reads and writes to be scaled independently and supports OLTP, stream processing and batch workloads. It also supports a globally synchronous consistent replicated log spanning multiple geographically separated regions. This paper describes how DistributedLog is structured, its components and the rationale underlying various design decisions. We have been using DistributedLog in production for several years, supporting applications ranging from transactional database journaling, real-time data ingestion, and analytics to general publish-subscribe messaging.**

I. Introduction

The publish-subscribe paradigm appears in large-scale distributed applications in many forms - queues, transaction logs for distributed data stores, change capture and propagation, and data movement in ETL (extract, transform, load) pipelines to name a few. Based on the characteristics of the workload, these use cases can be broadly classified into three categories.

Online transaction processing (OLTP) constitutes workflows and systems that implement the core business operations of the product. In a service-oriented architecture, these workflows are implemented by several micro services that communicate using queues. Workflows that modify state may use publish-subscribe systems to record the state changes and replay atomically. As they impact the core user experience or business metrics, these use cases expect very high availability and strict bounds on latency (for example, less than 30 ms at the 99.9th percentile).

Near real-time processing includes real-time analytics and asynchronous processing triggered by core workflows. For instance updating a search index as objects are created or tracking user impressions. These applications can tolerate higher latencies (500 milliseconds to several seconds), but the volume of data is large.

Thirdly, publish-subscribe systems are used for **collecting and aggregating** large amounts of log or telemetry data for **batch processing**. These systems are optimized for throughput and cost efficiency as opposed to latency. Latencies of several minutes are acceptable.

These use cases have very different requirements from the underlying publish-subscribe system, and as such, it is not surprising that in many environments different solutions are used for each of these categories. Our systems architecture in early 2012 was no exception to this.

Kestrel[3] - a queuing service - satisfied the OLTP use cases. Kestrel was simple to provision and scale, provided predictable latencies when the queues were drained efficiently. However, it suffered from many limitations. Kestrel did not have replication forcing us to rely on RAID and survive single disk failures. Every consumer in Kestrel had a separate copy of the queue. The persistent state of Kestrel was stored in journal files, with a separate file for every copy of a queue. Reading from these journal files was almost always guaranteed to incur random I/O. Therefore it was critical that all the major consumers had enough memory allocated to them so the tail of the queue can be served from memory and enough partitions so that when the consumer was in a read-behind mode, the readers could catch up. Adding more serving capacity by increasing the memory or network bandwidth meant adding more expensive machines with hardware redundancy to the cluster with their disks severely underutilized most of the time.

Limitations in Kestrel led several teams to explore other solutions. Kafka[8] 0.7 was gaining a foothold as Storm[10] integrated well with Kafka for real-time analytics. Kafka then did not demonstrate the predictable low latency required of a Kestrel replacement. Lack of ordering guarantees or strict durability in Kestrel led to use of append-only tables in MySQL as a replicated, ordered publish-subscribe system.

Each of these systems came with their maintenance overhead, including *software components* (the backend, various clients and integrations with other systems), *manageability and supportability* (deployment,

2375-026X/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDE.2017.163

upgrades, hardware maintenance and optimization), and *technical know-how*.

Consolidating different technologies and satisfying diverse use cases meant we needed a system with the following characteristics:

- **Unified stack** that allows trade-offs between latency, throughput and efficient resource usage using runtime configuration.
- **Guaranteed delivery and high availability** through durable writes, intra-cluster, and geo-replication.
- **Strict ordering** within a log.
- **Multi-tenancy** facilitating collocation of applications with different requirements for latency, throughput, and retention within the same cluster.
- **Scale** serving capacity **independent** of the storage.
- **Ease of manageability** by separating stateless serving and caching from persistent storage. The systems that maintain the persistent state provide simple primitives thereby requiring less on-going development. The stateless systems can be iterated upon more rapidly and deployed more frequently with less impact to availability.

We built *DistributedLog* to incorporate all of the above requirements. In this paper, we describe DistributedLog, its features and unique design choices that helped us meet the requirements we set out to achieve.

Through this paper we make the following contributions:

1) To our knowledge this is the first large-scale replicated log service that was built from the ground up to support transactional systems, stream processing, and batch data delivery.
2) A layered architecture that separates serving from persistence, allowing us to scale resources independently, and isolates reads from writes - allowing large read fanouts.
3) The system has been in production for three years, and one of our largest clusters delivers *1 trillion* records per day amounting to 17PB of data.
4) Enabling the same architecture to seamlessly support synchronous geo-replication for globally consistent operations.

The rest of the paper is organized as follows. Section II provides an overview of the system. In section III, we dive deeper into consistency and availability of data. In section IV, we present how we implement high performance reads. In section V, we describe geo-replication of logs for global consistency. In section VI & VII, we present applications and performance evaluation. In section VIII, we review related work. In section IX, we offer conclusions.

II. Overview

A. Capabilities

As different applications have different requirements, we've carefully considered the capabilities that should be included in DistributedLog leaving the rest up to the applications that build on the distributed log:

Consistency and Ordering Applications require two different types of ordering guarantees for building replicated systems - *Write Ordering* and *Read Ordering*. Write ordering requires that all writes issued by the writer be written in a strict order to the log. While read ordering only requires that any reader that reads the log stream should see the same record at any given position. The log records however may not appear in the same order that they were applied by the writer. DistributedLog is designed to support both use cases.

Partitioning Partitioning facilitates horizontal scale. The partitioning scheme is closely related to the ordering guarantees the application requires. For example, distributed key/value store requires modifications within each unit of consistency to be strictly ordered. On the other hand, real-time counting of impressions doesn't require strict order, can use *round robin partitioning* to evenly distribute records across all partitions. We provide applications the flexibility to choose a suitable partitioning scheme.

Processing Semantics Applications typically choose between *at-least-once* and *exactly-once* processing semantics. At-least-once processing guarantees to process all log records. However, after a restart, previously processed records may be re-processed if they have not been acknowledged. Exactly-once processing is a stricter guarantee where applications must see the effect of processing each record exactly once. It can be achieved by maintaining read positions together with the application state and atomically updating both the read position and the effects of the corresponding log records. For instance, for strongly consistent updates in a distributed key/value store the read position must be persisted atomically with the changes applied from the corresponding log records. Upon restart from a failure, the reader resumes from the last persisted position thereby guaranteeing that each change is applied only once.

B. Data Model

a) Log Streams: DistributedLog exposes the *log stream* as the unit of operation. A *log stream* is a totally ordered, append-only sequence of log records. A *log*

record is a sequence of bytes. Log records are batched into *entries* and written into *log segments*. Figure 1 depicts the structure of a log stream.

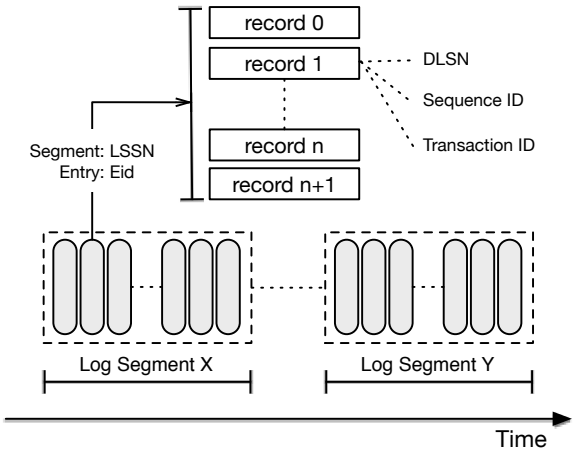

Fig. 1. Structure of a Log Stream

b) Log Segments: Although the application views the log stream as a continuous sequence of log records, it is physically stored as a set of *log segments*. All records in a log segment have the same replication configuration. The log segments are allocated, distributed and stored in a *log segment store*, where data is distributed across multiple failure domains (e.g. racks).

A log stream starts with an *in-progress* log segment. The current *in-progress* log segment will be completed, and a new one will be created when either following criteria are meet. It can because the log segment has been written for more than a configured interval (aka *time-based rolling*), the size of the log segment has reached a configured threshold (aka *size-based rolling*), or whenever the ownership of a log stream is changed.

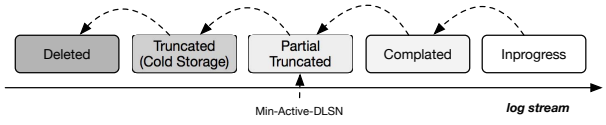

Fig. 2. The lifecycle of log segments

Applications can choose to truncate log segments *explicitly* or by specifying *TTL based expiration*. Truncated log segments are moved to cold storage for a configured period before being deleted permanently. The lifecycle of a log segment is summarized in Figure 2.

c) Log Sequence Numbers: Log records are written sequentially to a log stream, and assigned a unique sequence number called *DLSN* (DistributedLog Sequence Number). A DLSN has three components: a Log Segment Sequence Number, an Entry ID (*EID*), and a Slot ID (*SID*). Records are ordered by DLSN.

In addition to the DLSN, applications can assign a *Transaction ID*, a non-decreasing positive 64-bit integer, to each log record. This mechanism facilitates application-specific sequencing of records and positioning of the reader. For example, a common use of the transaction ID is to store the timestamp of when the log record was added to the log stream. This transaction ID can then be used to rewind to a particular time.

C. Architecture

The main components of DistributedLog are shown in Figure 3.

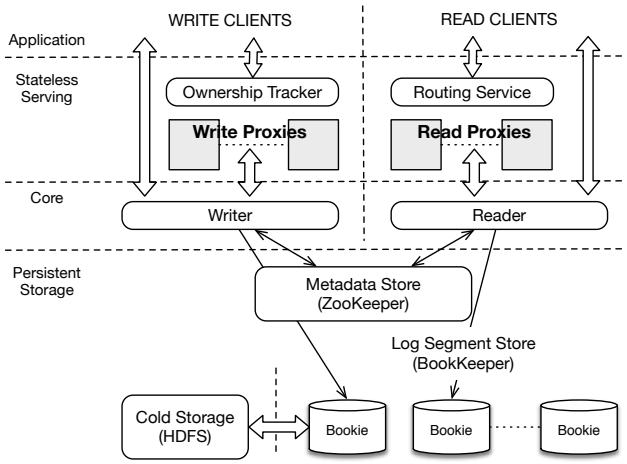

Fig. 3. DistributedLog Components

1) Persistent Storage: The *Log Segment Store*, *Cold Storage* and *Metadata Store* manage the persistent state for DistributedLog. These components are responsible for durability, availability and consistency.

a) Log Segment Store (Apache BookKeeper): The Log segment store is responsible for storing the log segments as they are created and ensure they are durable replicated. We use Apache BookKeeper[7] as the log segment store. BookKeeper helps us achieve low latencies as it is optimized for the following:

- Interleaved storage format in the Bookie (BookKeeper storage node) supports a large number of small writes efficiently. This mechanism allows supporting a large number of active log streams (10K per node).
- We enhanced BookKeeper's readers to support long polling (notifications) to deliver the records to the readers as soon as they are persisted.

- BookKeeper natively provides a fencing [1] mechanism which DistributedLogbuilds on to provide consistency & strict ordering.
- BookKeeper's storage architecture provides isolation between appends to the end of the log (writes), tail reads (most common) and catch-up reads (infrequent), thereby preventing slow readers from slowing down other readers and writers.

b) Cold Storage (HDFS): The data in the log segment store is eventually moved to cold storage. Applications may want to have access to old data for application error recovery or debugging. Cold storage allows cost-efficient storage of log segments for an extended period. As log segments are completed, they are proactively copied over to cold storage, thereby providing a backup for disaster recovery or an operations error.

c) Metadata Store: The metadata in DistributedLog consists of the mapping from log streams to their constituent log segments and attributes of each log segment. We use ZooKeeper[6] as the metadata store because it is a strongly consistent data store that provides versioned updates, reliable ordering, and change notifications using watches.

2) DistributedLog Core Library: DistributedLog core library implements the core data model and provides a single-writer-multiple-reader semantic.

Writers sequence log records written to the log streams. Therefore there is only one active log segment for a given log stream at a time.

Reading from a log stream starts by positioning a reader on a log record using either a DLSN or a Transaction Id. Once a reader has been positioned, it receives all the log records in increasing order of the sequence numbers, and each record is delivered exactly once. Based on their desired processing semantics, individual applications can choose an appropriate mechanism to record readers' positions and provide this as the start position when a new reader session begins (e.g. restart from a failure).

3) Serving: A stateless serving layer is built on top of the storage layer to support a large number of writers and readers, comprised of *Write Proxy* and *Read Proxy*. *Write Proxy* embeds the core library and sequences writes from many clients (aka *Fan-in*). *Read Proxy* caches log records for multiple readers consuming the same log stream (*Fan-out*).

Ownership Tracker At a given time, one write proxy is designated as the owner of a log stream. The *ownership tracker* tracks owners of log streams and handles failovers when write proxies fail.

Routing Readers Since readers are strict followers and read committed data from persistent storage, the read proxies do not have to maintain a single owner for every log stream. We use consistent hashing as a routing mechanism to route the readers to corresponding read proxies.

Applications can either use a thin client that talks to the serving tier to access DistributedLog or embed the core library to talk to the storage directly when they require strict write ordering. Applications choose partitioning strategies and track their read positions based on their specific requirements.

D. Lifecycle of records

Figure 4 illustrates the lifecycle of a log record in DistributedLog as it flows from writers to readers.

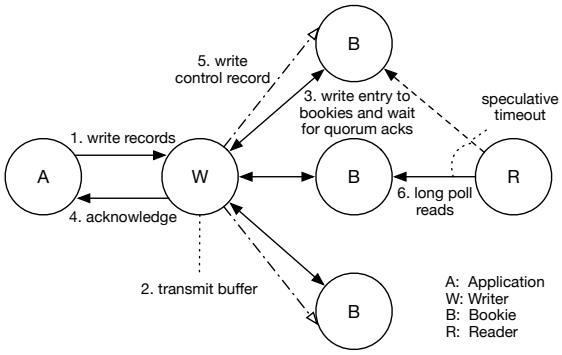

Fig. 4. Lifecycle of records.

The application constructs the log records and initiates write requests (step 1). The write requests will be forwarded to the write proxy that is the owner of the log stream. The owner write proxy will write the records to the transmit buffer for the log stream. Based on the configured flush policy, records in the transmit buffer will be sent as a batched entry to log segment store (step 2). Application can trade latency for throughput by transmitting *immediately* (lowest latency), or *periodically* (grouping records that appear within the flush period). The batched entry is transmitted to multiple bookies in parallel (step 3). The bookies will respond back to the writer once the entry is persisted durably. Once a quorum of writes succeed, the write is acknowledged (step 4). The writer records a '*commit*' to make this record visible to all the readers. This 'commit' can piggyback on the next batch of records from the application or be sent as a special control log record (step 5).

The reader that is (long) polling for new data will now receive the recently committed log records (step

6). Speculative long poll read requests would be sent to other replicas, to guarantee the response returns within SLA. The log records will be cached in read proxies for fan-out readers.

III. Consistency and Availability

DistributedLog achieves strong consistency, using the fencing mechanism provided in *Apache BookKeeper* to guarantee data consistency and versioned updates in *Apache ZooKeeper* to guarantee metadata consistency.

A. LastAddConfirmed

DistributedLog leverages bookkeeper's *LAC* (LastAddConfirmed) protocol [7]. Figure 5 illustrates the basic concepts of this protocol.

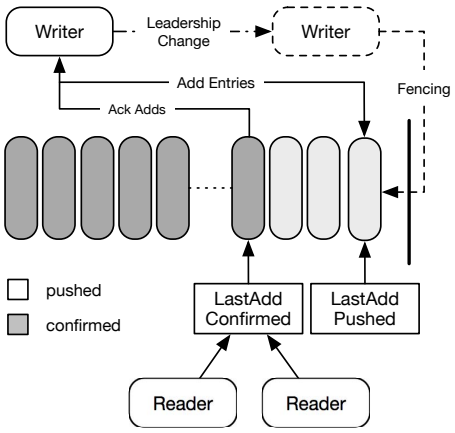

Fig. 5. Consistency in Log Segment Store

Each batched entry appended to a log segment will be assigned a *monotonically increasing entry id* by the log segment writer before being dispatched to BookKeeper. The log segment writer then updates **LastAddPushed** as the entry id of the last batched entry pushed to the log segment store by the writer. The entries can be written *out of order* but are always acknowledged *in entry id order*. Once an entry is successfully acknowledged the log segment writer updates the LAC (LastAddConfirmed) to this entry id.

The readers can read entries up to LAC as those entries are known to be durable replicated, and can be safely read without the risk of violating read ordering. All the entries written between LAC and LAP are unacknowledged data and must not be read. The writer includes the current LAC in each entry that it sends to BookKeeper. Consequently, each subsequent entry makes the records in the previous entry visible to the readers.

If no application records arrive within the specified SLA, the writer will generate a system record called control record to advance the LAC of the log stream. The control record is added either *immediately* after receiving acknowledgement or *periodically* if no application records are added. This behavior is configured as part of the writer's *flush policy*. While control log records are present in the physical log stream, they are not delivered by the log readers to the application. Since readers are strictly *followers*, they can leverage LAC to read durable data from any of the replicas without communicating or coordinating with the single writer.

B. Fencing

LAC protocol is not enough to guarantee correctness. When the ownership of a log stream has changed, there may be multiple writers active at the same time in the event of a network partition. Fencing is used to guarantee that only one of the writers can make forward progress. Fencing is a built-in mechanism in bookkeeper - when a client wants to fence a ledger[2], it will send a special fence request to all the replicas of that ledger; the bookies (storage nodes) that host that ledger will change the state of that ledger to *Fenced* after that all the write attempts to it would fail. The client considers the fence operation a success when it receives successful fence responses from a quorum of replicas.

Figure 6 illustrates how DistributedLog builds upon BookKeeper fencing to handle ownership changes.

Whenever the ownership is changed from one writer to the other (step 0), the new owner of the log stream will first retrieve the list of log segments along with their versions. The new owner will find the current in-progress log segment and recover the log segment in the following sequence:

1) It would first fence the log segment in BookKeeper (step 2.1). Fencing successfully makes the segment non-updateable.
2) If the previous owner is network partitioned, it may still try to add records to that log segment. Since the log segment has been fenced, writes by the previous owner will be rejected (step 2.2). The previous owner will discover that it has lost ownership.
3) Once the log segment is fenced, the new owner will initiate recovery of the log segment. Once the log segment has been recovered, it will issue a versioned set operation to the metadata store to convert the log segment status from '*in-progress*' to '*completed*' (step 2.3). A new in-progress log segment will be created by the new writer to continue writing to this log stream (step 3).

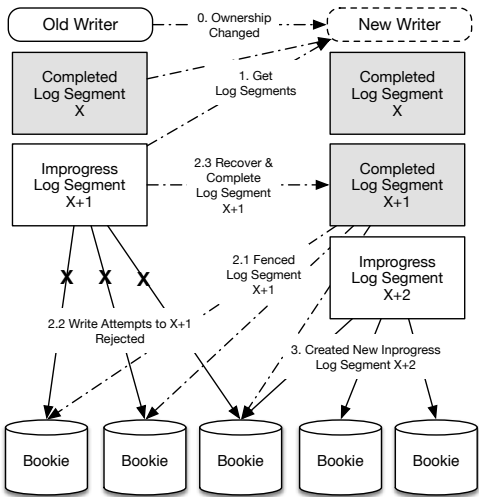

Fig. 6. Fencing & Consistency

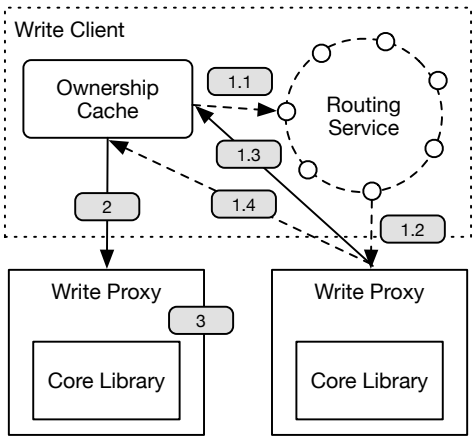

Fig. 7. Requests are routed from write clients to write proxies

Completing an in-progress log segment and creating a new log segment can be executed in parallel for performance.

C. Ownership Tracking

With a built-in fencing mechanism in the storage layer and conditional metadata updates, DistributedLog doesn't require strict *leader election* to guarantee correctness. Therefore we use '*ownership tracking*' as opposed to 'leader election' for log stream ownership management.

DistributedLog uses ZooKeeper ephemeral znodes for tracking the ownership of log streams. In production environments, we tune zookeeper settings to ensure failures can be detected within one second. An aggressive bound on failure detection increases the possibility of false positives. If ownership hops back and forth between write proxies, delays will result from writes blocked by log stream recovery. *Deterministic routing* allows multiple clients to choose the same write proxy to fail over when the current owner proxy is unavailable. The details are described in Figure 7.

The write client will first lookup the owner in the ownership cache, a local cache that caches a mapping between log streams and their owners. On cache miss, the client will use a consistent hashing based routing service to compute a candidate write proxy (step 1.1) and then send the write request to this proxy (step 1.2). If it already owns the log stream or it can successfully claim the ownership, it will satisfy the write request and respond back to the client (step 1.3). If it cannot claim the ownership, it then sends the response back to the client to ask it redirect to the right owner (1.4). All successful write requests will keep the local ownership cache up-to-date, which helps to avoid redirection of subsequent requests. Failed write requests will invalidate the cached ownership from the local cache, forcing subsequent requests to consult the routing service for a new write proxy to try.

IV. Streaming Reads

After the readers have caught up to the current tail of the log, DistributedLog provides readers with the ability to read new log records as they are published - a mechanism commonly known as *tailing* the log. Readers start out by **positioning** at a record in the log stream based on either *DLSN* or *Transaction ID*. The reader starts **reading** records until it reaches the tail of the log stream. Once it has caught up with the writer, the reader waits to be **notified** about new log records or new log segments.

a) Positioning: As mentioned above, there are three types of sequence numbers associated with a log record. Both *DLSN* (implicit) and *Transaction ID* (explicit) are attached to log records in writing time, and *Sequence ID* is computed at reading time. Applications can use any of them for positioning. DLSN is the best sequence number for positioning because it consists of the precise coordinates of a record in the log - a log segment number, entry id, and slot id. No additional search operations are required. To position a reader by transaction id, DistributedLog will first look up the list of log segments to find which log segment contains the given transaction id, and then look up the records in the found log segment to figure out the position of the record. Both log segment list lookup and lookup within

a log segment use binary search. Transaction id is a kind of secondary index, and it is useful for logical lookups depending on its meaning (for example lookup by some time in the past if it is a timestamp).

b) Reading: Figure 8 illustrates reading *batched entries* From the log segment store. There are two basic read operations: *read a given entry by entry id* (a) and *read LAC* (b).

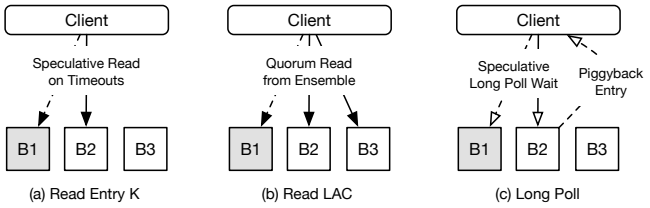

Fig. 8. Read entries from log segment store

Since an entry is immutable after it is appended to a log segment, only one replica needs to be read to satisfy a read operation.. We deployed a *speculative read* mechanism to achieve predictable low latency even during bookie failures: A read request will be sent to the first replica. If the client doesn't receive a response within a speculative timeout, it will send another request to a second replica and wait for the responses of both the first and second replicas. It will keep retrying the replicas until it times out or receives a valid response.

Reading LAC is an operation for readers to catch up with the writer. It is typically a quorum-read operation to guarantee freshness: the client sends the read requests to all replicas in the log segment and waits for responses from a majority of them. It could be optimized to be a best-effort quorum-read operation for tailing reads, in which case it does not have to wait for a quorum response from the replicas and can return whenever it sees an advanced LAC.

Figure 8(c) illustrates the third type of read request, which is called '*Long Poll Read*'. It is a combination of (a) and (b), serving the purpose of *reading the next available entry* in the log segment. The client sends a long poll read request along with the next read entry id to the log segment store. If the log segment store already saw the entry and it is committed (the entry id is not greater than LAC), it responds to the request immediately with the latest LAC and the requested entry. Otherwise, it would wait for the LAC to advance to the provided entry id and respond back with the requested entry. A *speculative* read mechanism is also deployed in long polling to achieve predictable low latency.

c) Notifications: Once the reader is caught up with the writer, it would turn itself into '*notification*' mode. In this mode, it would wait for notification of new records with *long polling reads* (described above) and *notification* of state changes of log segments. The notification of log segments changes is provided by a ZooKeeper watcher.

d) ReadAhead: The reader will read ahead to proactively bring new data into a cache, for applications to consume. This helps to reduce read latency as it proactively brings newer records into the cache before applications consume them. DistributedLog uses LAC as an indicator to detect if a reader is still catching up or has already caught up, and adjusts the read ahead pace based on the reader's state and how fast it is consuming data.

V. Geo-Replicated Log

In a multi-datacenter setup, a geo-replicated distributed database must provide a globally consistent view of critical user and business application data which in turn requires a globally consistent order of log records in DistributedLog.

We introduce *region* as an additional level in the network hierarchy. *Regions* are treated as independent failure domains in that we can mark complete regions as *unavailable* or *offline*. A region need not imply a separate geographic location; it can be a separate cluster within the same location if it is an independent failure domain (e.g. separate power source and network topology).

We need a globally consistent metadata store as a building block for global consistency. In order to keep the changes to the rest of the stack to a minimum, the knowledge of the network topology is contained within the module that accesses the *Log Segment Store*. A placement policy is used to determine the storage nodes that data for a given log stream should be placed on, and this policy is also used to route reads to the appropriate nodes. We introduce a special *region aware placement policy* that takes into account distribution of data across multiple regions for fault tolerance and read locality.

Figure 9 illustrates a *Geo-Replicated Log* setup. The '*Global*' metadata store is a global setup of ZooKeeper. Write clients talk to the write proxies in their local region to bootstrap the ownership cache and are redirected to write proxies in other regions through direct TCP connections. Readers will try reading from the local region whenever possible and speculatively try remote regions for fault tolerance.

A. Region Aware Data Placement Policy

Region aware placement policy uses *hierarchical* allocation so that data is spread uniformly across the available regions and within each region uniformly across the available racks.

Region aware placement policy is governed by a parameter that ensures that the ack-quorum covers at least $minRegionsForDurability$ distinct regions. This parameter allows the system to survive the failure of $(totalRegions - minRegionsForDurability)$ regions without loss of availability.

The placement algorithm maintains the following invariant:

> There is no combination of nodes that would satisfy the ack quorum with less than $minRegionsForDurability$

This invariant ensures that enforcing ack-quorum is sufficient to enforce that the entry has been made durable in $minRegionsForDurability$ regions.

B. Cross Region Speculative Reads

As discussed before, read requests can be satisfied by any replica of the data, however, for high availability, speculative requests are sent to multiple copies to ensure that at least one of the requests returns within the time specified by the SLA. The reader consults the data placement policy to get the list of replicas that can satisfy the request in the order of preference. This order is decided using the following principle:

- The first node in the list is always the node that is closest to the client - if more than one such node exists, one is chosen at random.
- The second node is usually the closest node in a different failure domain. In the case of a two-level hierarchy that would be a node in a different rack.
- The third node is chosen from a different region

The delay between successive speculative read requests ensures that the probability of sending the *n-th* speculative read request decays exponentially with *n*. This mechanism ensures that the number of requests that go to more distant nodes is kept to a minimum. However making sure that we cross failure domains in the first few speculative requests improves fault-tolerance of the reader. The reader transparently handles temporary node failures via this straightforward and generalizable speculative read policy. This policy can be thought of as the most granular form of failover where each request essentially fails over to an alternate node if the primary node it attempted to access is unavailable. In practice, we have found this mechanism is also better able to handle network congestion where routes between specific pairs of nodes may become unavailable without necessarily making the nodes completely inaccessible.

The use of distance and failure domains to decide the preferred order of read requests extends beyond the static location of the nodes and can be utilized for effective decisions based on observed latency or failure rates from specific nodes. Routing decisions based on such metrics are better able to capture partial and transient network outages that affect specific routes within the network.

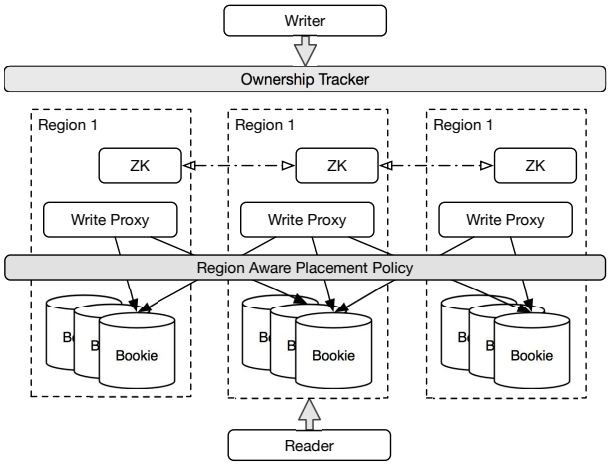

Fig. 9. Global DistributedLog

VI. Applications

In this section, we will describe a few DistributedLog applications and how they use DL to meet their requirements.

A. PUB/SUB

DistributedLog is used for building different message consuming systems. A common usage is to use the log streams for building a partitioned pub/sub system. Each topic is partitioned into multiple partitions, and each partition is backed by one log stream. The publisher writes the data according to the configured partitioning scheme, and the subscribers that form a subscriber group coordinate to reading those log streams. The subscribers' read positions are periodically updated in an offset store for subscribers recovering from failures, to achieve at-least-once processing semantics. The data is kept for a configurable period and truncated after TTL expiration.

B. Transaction Log

Replicated storage uses DistributedLog to order updates to all its replicas, to support strongly consistent operations such as compare-and-set (CAS). The storage

replica stores its read position along with its persistent state (for example, in SSTables of an LSM-tree based storage backend). This way it can resume reading from the persisted read position when recovering from failures and achieve exactly-once processing. The data in the log stream is only allowed to be truncated when all the replicas have successfully applied the records. The truncation position is determined by finding the smallest known position of all replicas' read positions. The coordination can be achieved by each replica publishing its read position into the same log stream.

C. Real-time Change Propagation

Real-time changes are propagated from Twitter's core product into DistributedLog and used for rebuilding states in other services. Search indexing is one example. Indexed objects are partitioned into multiple sets. The indexer of each partition writes the propagated changes in a strict order and leverages fencing to guarantee correctness on failure (i.e. no two indexers are propagating changes at the same time for a given partition). The readers consume the changes by rewinding to a given time and catching up.

VII. Performance

A. DistributedLog in Production

In production, we deploy DistributedLog in one global cluster among multiple data centers, and a few local clusters in each data center for supporting different types of workloads - transaction log, pub/sub, and change propagation. Currently, there are around 8,000 global streams and 30,000 local log streams, comprised of 1,700,000 live log segments. These active log segments are spread over several hundred bookie nodes with retention ranging from hours to years. One of our largest systems delivers 1 trillion records per day, roughly amounting to 17.5 PB of data.

Our evaluation of DistributedLog focuses on our production deployment and a few synthetic benchmarks that illustrate specific performance characteristics of DistributedLog.

Hardware

Bookies: The bookie node has 24 CPU cores, 70GB of memory, and a 1Gb NIC. There is one HDD drive and 5 SSD drives mounted; each drive has 420GB. The HDD drive is used as the journal directory while the others are used for ledger directories. Unless specified we use a bookkeeper cluster that contains 20 nodes.

Dedicated Instance: A proxy that has 22 CPU cores, 58GB memory, and a 10Gb NIC.

Shared Instance: A proxy that is scheduled in a shared mesos [5] pool. It is a micro-instance, which has 4 CPU cores, 6GB memory, and only supports 37.5MB/sec outbound network bandwidth.

Metrics

Write Latency: the time from when a writer writes a log record to when it receives the acknowledgement.

Delivery Latency: the time from when a writer writes a log record to when a reader reads it.

Percentiles: we measure p50, p99 and p999 values for latency metrics.

B. Latency

In this section, we evaluate the latency and performance of DistributedLog using different flushing policies. These flushing policy settings are production settings for different real-world workloads. They are:

Immediate Commit: User records are transmitted immediately, and a control record is committed immediately once the additions are acknowledged.

Periodic Flush: User records are buffered and transmitted periodically.

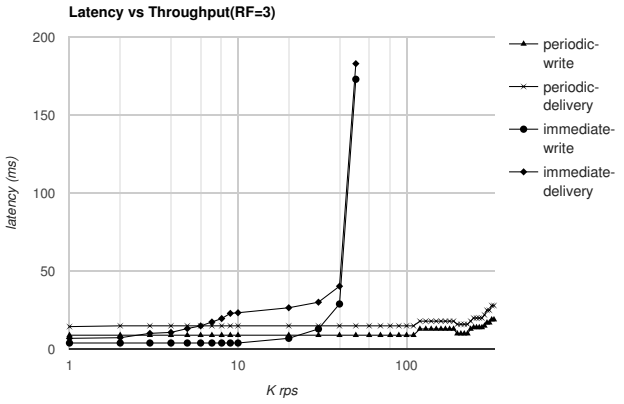

Fig. 10. Latency and throughput under different flush policies.

Methodology. We use two dedicated instances - one for writing records and one for reading records. The write instance writes 128-byte log records increasing RPS from 10K records per second. Figure 10 illustrates the test result.

Result. DistributedLog can sustain very low latency on immediate-commit when throughput is less than 30k. After 30k, it starts hitting the IOPS bandwidth limitation on the journal disk. However, by trading off latency for throughput, DistributedLog could sustain 330k rps with 99% percentile latency still under 10 ms.

C. Scalability

In this section, we evaluate the scalability of DistributedLog with an increasing number of streams.

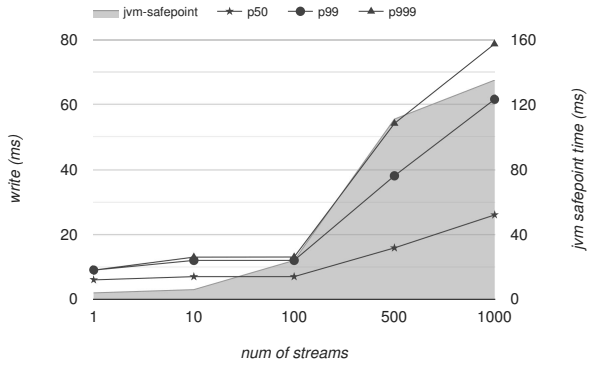

Fig. 11. Latency increased with increased number of stream on a single hybrid proxy. The latency is affected by the cpu usage and jvm safepoint time.

1) Scaling the number of streams on a proxy: **Methodology**. We use two dedicated instances, one for writing records and one for reading records. The write instance is writing 10K records per second to multiple streams, with 10ms periodical flush. Each record is 128 bytes. The number of streams is increased from 1 stream to 1000 streams. Figure 11 illustrates the test result.

Result. DistributedLog can sustain low latency with high throughput at 100,000 records per second with roughly a hundred streams on a single proxy. If we try to add more streams, the writer starts seeing high latency. The high latency is introduced by CPU context switching and elevated JVM safepoint time.

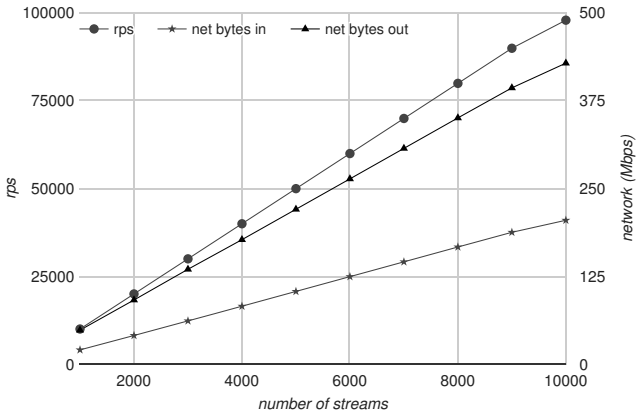

Fig. 12. Throughput increases linearly with the number of streams.

2) Scaling the number of streams on a cluster: **Methodology**. We setup a writer job in shared mesos. Each instance of the writer job writes 100 records per second to 100 distinct streams, with 10ms periodic flush. Each record is 1024 bytes. We increase the number of instances by ten every 30 minutes. We also set up a reader job in shared mesos at the beginning to tail read all of the 10k streams thateventually exist. Figure 12 illustrates the test result.

Result. DistributedLog can linearly scale the aggregated throughput with increasing number of streams when there is sufficient capacity.

D. Efficiency for high read fanout

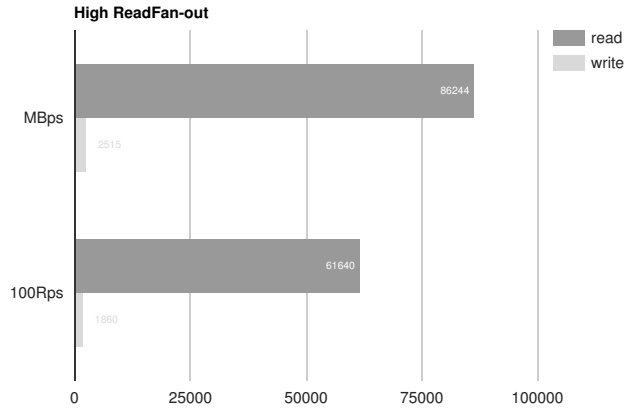

Fig. 13. An application writes more than 2GB per second, while the data has been fanned-out 40x to readers.

Methodology. This workload is from one of our analytics applications. The analytics application writes around more than 2GB per second. The data has been fanned-out to multiple readers by 40x. It uses dozens of bookies (1Gb NIC) and almost a hundred read proxies (hybrid instance - 10Gb NIC) to support such traffic. Figure 13 illustrates the test result.

Results. DistributedLog can scale serving writes independently of scaling serving reads. Adding more read proxies to cache and serve data enables scaling of reads.

E. Geo-Replicated Log

Methodology. We evaluate write and read latency with increases In throughput in a 2-region global cluster. Each region has five bookie nodes - one region is on the west coast of the United States while the other region is on the east coast and the latency between these two regions is around 80ms. We setup a writer job and a reader job in mesos - the writer job has 100 instances, and keeps writing 1024-byte records to 10k

log streams, while the reader job has 100 instances that tail read from those 10k log streams. Each log stream is configured to replicate 10 ways. We use immediate flush on write proxies to make data immediately available to the readers. We plot the result of write latency in Figure 14 and delivery latency in Figure 15.

Result. DistributedLog can sustain a high throughput of over 38,000 immediate-flushed records per second, without a latency penalty. Beyond 38,000, both the publishers and subscribers start to see elevated latency, as we are approaching the limit of a single bookie.

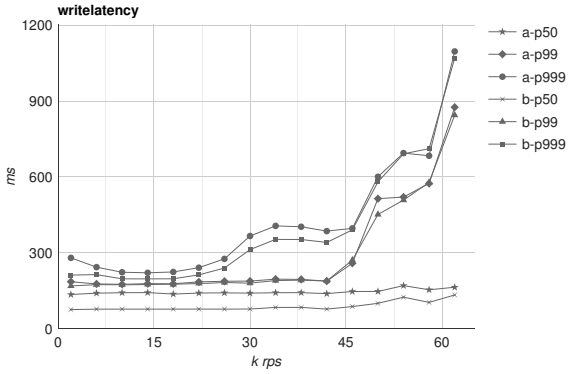

Fig. 14. write latency vs RPS (2k - 62k)

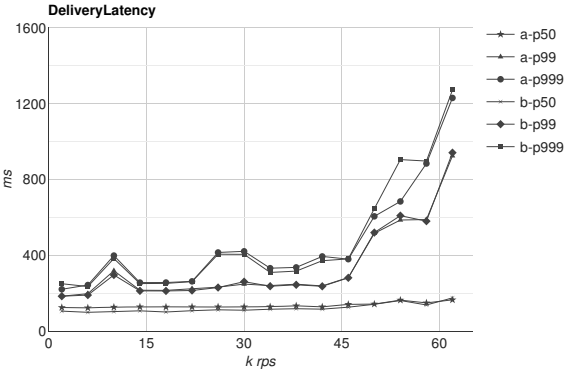

Fig. 15. delivery latency vs RPS (2k - 62k)

VIII. Related Work

Below we consider some of the best-known systems from consensus, pub/sub, and messaging area.

Raft Raft[9] is a distributed consensus algorithm that is designed for easy understanding. The algorithm is decomposed into two major independent problems: Leader election and Log replication. To some extents, DistributedLog is similar to Raft - log segment acts as the role of 'term' in Raft, data are replicated to multiple storage nodes by the leader (log segment writer) and the unique log segment id is the term number in Raft. However, DistributedLog doesn't enforce leader election like Raft. It provides a fencing mechanism to guarantee correctness when a writer moves from one log segment to the other one. This is for two reasons: 1) Leader election usually involves lease based failure detection. It can be impacted by different types of problems such as long-pause garbage collection and network interruptions. Mixing leader election with log replication together makes things more complicated and less reusable. Instead, separating out leader election allow us to leverage existing common infrastructure such as ZooKeeper to do failure detection. 2) Applications usually need an additional fencing mechanism to guarantee correctness when using external consensus or coordination systems for leader election. Providing a built-in fencing mechanism in DistributedLog simplifies the applications using it.

Kafka Kafka[8] is LinkedIn's message bus, now open source and maintained by Apache. It has a topic-based pub/sub API, providing features like partitioning, subscriber groups and offset tracking. The writes, reads, and storage are served from a single role called a Broker. Each partition of a topic is stored as multiple files on the local disks on a broker. Kafka may suffer bad performance with a large number of partitions or when durable writes are enabled, and any slow reader can potentially affect other writers and readers in the same system due to the occurrence of random I/O in the system. In DistributedLog, the data is stored in an interleaved structure and optimized for I/O isolation for multi-tenancy. Also, DistributedLog uses a parallel replication scheme for writing multiple copies of data, which allows it to tolerate slow replicas without impacting tail latency; whereas Kafka could suffer performance degradation when a master broker becomes slow as it uses a master-slave replication scheme. Kafka is mainly used for collecting data for analytics where most users run it in nondurable mode.

Thialfi Thialfi[4] is Google's cloud notification service namely the invalidation of cached objects. Thialfi is geographically distributed and highly reliable, even in the face of long disconnections. However, its clients are applications running in browsers on end-users' mobile phones, laptops, and desktops, not applications within Google itself, such as caching and indexing services. The workloads on Thialfi are, accordingly, very different from DistributedLog. In particular, Thialfi is not concerned about I/O efficiency. Thialfi also sends only

version number for the data to subscribers and coalesces many updates into the most recent one. DistributedLog, on the other hand, sends all records to readers, and each record contains all information needed by applications, not just a version number.

IX. Conclusion

This paper describes DistributedLog, a high performance replicated log service. DistributedLog introduces a layered architecture, separating stateless serving from persistent storage to allow to independent scaling of reads and writes writes which facilitates multi-tenancy and large read fan-out. We've demonstrated that DistributedLog scales to support various workloads, from transactional online serving, to real-time stream analytics and reliable data delivery. One of our largest systems delivers 1 trillion records per day which roughly amounts to 17.5 PB of data in the steady state with low latency.

X. Acknowledgements

DistributedLog would not have been possible without the hard work of Aniruddha Laud, Franck Cuny, Mike Lindsey, David Helder, Vishnu Challam, Amit Shukla, and Rob Benson. We would also like to thank the Apache BookKeeper community for teaching us numerous lessons and for moving the state of distributed logging forward.

References

[1] Bookkeeper fencing. https://cwiki.apache.org/confluence/display/BOOKKEEPER/Fencing.

[2] Bookkeeper overview. http://bookkeeper.apache.org/docs/r4.3.2/bookkeeperOverview.html.

[3] Kestrel: A simple, distributed message queue. https://twitter.github.io/kestrel.

[4] A. Adya, G. Cooper, D. Myers, and M. Piatek. Thialfi: a client notification service for internet-scale applications. In *ACM SOSP*, 2011.

[5] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[6] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.

[7] F. P. Junqueira, I. Kelly, and B. Reed. Durability with bookkeeper. *ACM SIGOPS*, 2013.

[8] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *NetDB*, 2011.

[9] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, 2014.

[10] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. In *SIGMOD*, 2014.