Building a Replicated Logging System with Apache Kafka

Guozhang Wang[1], Joel Koshy[1], Sriram Subramanian[1], Kartik Paramasivam[1]
Mammad Zadeh[1], Neha Narkhede[2], Jun Rao[2], Jay Kreps[2], Joe Stein[3]
[1]LinkedIn Corporation, [2]Confluent Inc., [3]Big Data Open Source Security LLC

ABSTRACT

Apache Kafka is a scalable publish-subscribe messaging system with its core architecture as a distributed commit log. It was originally built at LinkedIn as its centralized event pipelining platform for online data integration tasks. Over the past years developing and operating Kafka, we extend its log-structured architecture as a replicated logging backbone for much wider application scopes in the distributed environment. In this abstract, we will talk about our design and engineering experience to replicate Kafka logs for various distributed data-driven systems at LinkedIn, including source-of-truth data storage and stream processing.

1. INTRODUCTION

Kafka was first developed to solve a general problem of delivering extreme high volume event data to diverse subscribers. As the largest professional social network on the Internet, today LinkedIn generates billions of log entries and events that capture its over-300 million members' activities every day and feeds them to its products. Examples of these products include derived information and news feeds such as "Who's Viewed My Profile?"; machine learning backed social and content recommendation such as "People You May Know"; and many other backend monitoring and reporting platforms such as site health auditing and account fraud detection. It is hence very critical to be able to deliver this high-volume activity data in real time to both our user-facing products as well as these backend systems.

With this goal in mind, in 2010 we implemented Kafka as LinkedIn's centralized online data pipelining system. Kafka organizes messages as a partitioned write-ahead commit log on persistent storage and provides a pull-based messaging abstraction to allow both real-time subscribers such as online services and offline subscribers such as Hadoop and data warehouse to read these messages at arbitrary pace. Since Oct. 2012, Kafka has become a top-level Apache open source software and be widely adopted outside LinkedIn as well [1].

Recently, there has been much renewed interest in using log-centric architectures to build distributed systems that provides efficient durability and availability [3, 5, 6, 7]. In this approach, a collection of distributed servers can maintain a consistent system state via a replicated log that records state changes in sequential order. When some of the servers fail and come back, their states can be deterministically reconstructed by replaying this log upon recovery. At LinkedIn, the idea of using a reliable and highly available log structure as the underlying data flow to scale distributed systems has also been well envisioned and exercised over the past years. Given its log-structured architecture, we realize Kafka could be a good fit for this vision and have extended its design as a replicated logging system. In this abstract, we present our experience in building such a system with Kafka, and describe a few use cases at LinkedIn of this replicated log.

2. BUILDING A REPLICATED LOG

We identify two major requirements while extending Kafka into a replicated logging system and present our solutions to them as follows:

(1) A Log Consensus Protocol is required to keep the replicated log consistent among servers, i.e. we need to ensure that every replica of the log eventually contains the same entries in the same order even when some servers fail. There are two common strategies for keeping replicas in sync: primary-backup replication and quorum-based replication.In both cases, one replica is designated as the leader and the rest are followers. All log appends go through the leader and then propagate to the followers. In primary-backup replication, the leader waits until the append is completed on all the replicas before acknowledging the client, whereas in quorum-based replication it only waits until the append is completed on a majority of the replicas. As a result, the primary-backup approach could employ fewer replicas $(F+1)$ to tolerate the same number of failures (F) than the quorum approach $(2F+1)$, while the quorum approach usually achieves better latency as it only requires any $F+1$ replicas out of $2F+1$ for committing log appends.

Since Kafka is typically deployed in the same data center where the variance of network latency is low, the benefit of using less replicas outweighs the potential slight increase in latency. We designed and implemented Kafka message log replication in Kafka 0.8.0, with the key idea of separating the key elements of a consensus protocol such as leader election and membership changes from log replication itself. Each log partition can be replicated on different servers, with one replica as the leader and others as followers. Each server can act as the leader for some of the partitions and follower for some other partitions at the same time. We rely on the quorum-based Apache Zookeeper service [4] for making consensus decisions such as leader election and storing critical partition metadata such as replica lists, while using a primary-backup approach for replicating logs from leader to followers. The log format is much simpler with such separation since it does not need to maintain any leader election related information, and the replication factor for the log is decoupled from the number of parties required for the quorum to proceed (e.g., in Kafka we can choose a replication factor of two for the log while using an ensemble of five Zookeeper nodes). Producer clients can specify different replication criterion for acknowledging their sent messages; for example, they can choose to wait for acknowledgement until the messages have been replicated to all in-sync replicas, or only the leader. On the other hand, consumer clients only fetch messages that have been replicated to all in-sync replicas. We will present more details about committing, in-sync replica membership management, partition assignment and leader elections, etc in the talk.

(2) A Log Truncation Mechanism is required to avoid the log from growing indefinitely.Kafka originally supports only window-based data retention policies where the window can be defined based on time or space: for example, users can indicate servers to clean up oldest log segments after some time has passed or some amount of bytes have cumulated on the log. Such retention policies do not suit for cases where log entries are organized by keys, and the system only cares about the most recent data value of each key but not necessarily every change to it.

We introduced a key-based log compaction mechanism in Kafka 0.8.1 where stale log information can be cleaned up to reduce log size.Each message is formulated as a key-value pair, and applications can use the message key to organize log entries whose precedence is then naturally defined as append ordering. When log compaction is enabled, Kafka servers asynchronously scan log segments of each partition, purge old events if there is a newer message with the same key in order to construct compacted log segments. The compacted logs can then be atomically swapped in with the old logs, with the wrapped out logs deleted afterwards. This process is done without interfering concurrent client requests to append or fetch log entries at the same time.

3. USE CASES

We describe two use cases at LinkedIn that make use of this replicated logging system:

(1) Change Log Replication. One example usage of replicated Kafka logs is to store data changes of a distributed data store. For instance, Espresso is a scalable document store built at LinkedIn to serve as its online data storage platform [8]. It depends on the underlying storage engine for its data replication (e.g. MySQL storage node replication), which results in high operating and maintenance costs. To resolve this problem, we are replacing Espresso's replication layer by using Kafka to capture its data change updates as a commit log. This log will then be applied to support both intra and inter data center replication. To be more concrete, we create a Kafka topic for each Espresso table, which is partitioned accordingly to the shards of the table. Each Espresso instance embeds a Kafka producer to publish its change logs to the local Kafka cluster. Other Espresso instances who host the slave shards of the corresponding partitions can then consume these Kafka streams and apply update events. By doing this, we migrate Espresso into a partition-level replication mechanism which largely increased load balance and system operability.

(2) Resilient Stream Processing. Historically, the term "stream processing" refers to the scenario where data tuples are streaming in continuously but can only be queried once. However, when data streams are backed by persistent storage, you can (re-)process these streams from long time ago as long as it is still retained in the source. In this case, stream processing can be treated just as a generalization of batch-oriented processing that produces low-latency outputs from a sequence of input data without blocking on the input data set to be fully complete. To achieve this goal, we have developed Samza, an Apache open source distributed stream processing framework which uses Kafka as its underlying streaming layer [2]. Each Samza job can read one or more upstream logs generated by other jobs or some data stream firehose, and write output to downstream logs or other systems (e.g. materialized views in databases); a group of such jobs connected by their input and output streams can then form a dynamically modifiable stream processing topology with much simpler processor failure and back-pressure handling thanks to the underlying durable and highly available Kafka stream. Samza has been used at LinkedIn for various low-latency data processing tasks such as service monitoring, real-time data standardization, online machine learning, etc.

4. CONCLUSION

Log structures have been the heart of database and distributed system design for many years. At LinkedIn we demonstrate via Apache Kafka that a replicated log can also be a very powerful abstraction in practice for scaling a wide scope of distributed systems.

5. REFERENCES

[1] Apache Kafka. `http://kafka.apache.org`.

[2] Apache Samza. `http://samza.apache.org`.

[3] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber. CORFU: A distributed shared log. *ACM Trans. Comput. Syst.*, 31(4):10, 2013.

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

[5] W. Lin, M. Yang, L. Zhang, and L. Zhou. PacificA: Replication in log-based distributed storage systems. Technical Report MSR-TR-2008-25, Microsoft Research.

[6] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, pages 305–319, 2014.

[7] J. K. Ousterhout et al. The case for ramclouds: scalable high-performance storage entirely in DRAM. *Operating Systems Review*, 43(4):92–105, 2009.

[8] L. Qiao et al. On brewing fresh Espresso: Linkedin's distributed data serving platform. In *SIGMOD*, pages 1135–1146, 2013.