Kafka: a Distributed Messaging System for Log Processing

Jay Kreps
LinkedIn Corp.
jkreps@linkedin.com

Neha Narkhede
LinkedIn Corp.
nnarkhede@linkedin.com

Jun Rao
LinkedIn Corp.
jrao@linkedin.com

ABSTRACT

Log processing has become a critical component of the data pipeline for consumer internet companies. We introduce Kafka, a distributed messaging system that we developed for collecting and delivering high volumes of log data with low latency. Our system incorporates ideas from existing log aggregators and messaging systems, and is suitable for both offline and online message consumption. We made quite a few unconventional yet practical design choices in Kafka to make our system efficient and scalable. Our experimental results show that Kafka has superior performance when compared to two popular messaging systems. We have been using Kafka in production for some time and it is processing hundreds of gigabytes of new data each day.

General Terms

Management, Performance, Design, Experimentation.

Keywords

messaging, distributed, log processing, throughput, online.

1. Introduction

There is a large amount of "log" data generated at any sizable internet company. This data typically includes (1) user activity events corresponding to logins, pageviews, clicks, "likes", sharing, comments, and search queries; (2) operational metrics such as service call stack, call latency, errors, and system metrics such as CPU, memory, network, or disk utilization on each machine. Log data has long been a component of analytics used to track user engagement, system utilization, and other metrics. However recent trends in internet applications have made activity data a part of the production data pipeline used directly in site features. These uses include (1) search relevance, (2) recommendations which may be driven by item popularity or co-occurrence in the activity stream, (3) ad targeting and reporting, and (4) security applications that protect against abusive behaviors such as spam or unauthorized data scraping, and (5) newsfeed features that aggregate user status updates or actions for their "friends" or "connections" to read.

This production, real-time usage of log data creates new challenges for data systems because its volume is orders of magnitude larger than the "real" data. For example, search, recommendations, and advertising often require computing granular click-through rates, which generate log records not only for every user click, but also for dozens of items on each page that are not clicked. Every day, China Mobile collects 5–8TB of phone call records [11] and Facebook gathers almost 6TB of various user activity events [12].

Many early systems for processing this kind of data relied on physically scraping log files off production servers for analysis. In recent years, several specialized distributed log aggregators have been built, including Facebook's Scribe [6], Yahoo's Data Highway [4], and Cloudera's Flume [3]. Those systems are primarily designed for collecting and loading the log data into a data warehouse or Hadoop [8] for offline consumption. At LinkedIn (a social network site), we found that in addition to traditional offline analytics, we needed to support most of the real-time applications mentioned above with delays of no more than a few seconds.

We have built a novel messaging system for log processing called Kafka [18] that combines the benefits of traditional log aggregators and messaging systems. On the one hand, Kafka is distributed and scalable, and offers high throughput. On the other hand, Kafka provides an API similar to a messaging system and allows applications to consume log events in real time. Kafka has been open sourced and used successfully in production at LinkedIn for more than 6 months. It greatly simplifies our infrastructure, since we can exploit a single piece of software for both online and offline consumption of the log data of all types. The rest of the paper is organized as follows. We revisit traditional messaging systems and log aggregators in Section 2. In Section 3, we describe the architecture of Kafka and its key design principles. We describe our deployment of Kafka at LinkedIn in Section 4 and the performance results of Kafka in Section 5. We discuss future work and conclude in Section 6.

2. Related Work

Traditional enterprise messaging systems [1][7][15][17] have existed for a long time and often play a critical role as an event bus for processing asynchronous data flows. However, there are a few reasons why they tend not to be a good fit for log processing. First, there is a mismatch in features offered by enterprise systems. Those systems often focus on offering a rich set of delivery guarantees. For example, IBM Websphere MQ [7] has transactional supports that allow an application to insert messages into multiple queues atomically. The JMS [14] specification allows each individual message to be acknowledged after consumption, potentially out of order. Such delivery guarantees are often overkill for collecting log data. For instance, losing a few pageview events occasionally is certainly not the end of the world. Those unneeded features tend to increase the complexity of both the API and the underlying implementation of those systems. Second, many systems do not focus as strongly on throughput as their primary design constraint. For example, JMS has no API to allow the producer to explicitly batch multiple messages into a

NetDB'11, Jun. 12, 2011, Athens, Greece.

single request. This means each message requires a full TCP/IP roundtrip, which is not feasible for the throughput requirements of our domain. Third, those systems are weak in distributed support. There is no easy way to partition and store messages on multiple machines. Finally, many messaging systems assume near immediate consumption of messages, so the queue of unconsumed messages is always fairly small. Their performance degrades significantly if messages are allowed to accumulate, as is the case for offline consumers such as data warehousing applications that do periodic large loads rather than continuous consumption.

A number of specialized log aggregators have been built over the last few years. Facebook uses a system called Scribe. Each front-end machine can send log data to a set of Scribe machines over sockets. Each Scribe machine aggregates the log entries and periodically dumps them to HDFS [9] or an NFS device. Yahoo's data highway project has a similar dataflow. A set of machines aggregate events from the clients and roll out "minute" files, which are then added to HDFS. Flume is a relatively new log aggregator developed by Cloudera. It supports extensible "pipes" and "sinks", and makes streaming log data very flexible. It also has more integrated distributed support. However, most of those systems are built for consuming the log data offline, and often expose implementation details unnecessarily (e.g. "minute files") to the consumer. Additionally, most of them use a "push" model in which the broker forwards data to consumers. At LinkedIn, we find the "pull" model more suitable for our applications since each consumer can retrieve the messages at the maximum rate it can sustain and avoid being flooded by messages pushed faster than it can handle. The pull model also makes it easy to rewind a consumer and we discuss this benefit at the end of Section 3.2.

More recently, Yahoo! Research developed a new distributed pub/sub system called HedWig [13]. HedWig is highly scalable and available, and offers strong durability guarantees. However, it is mainly intended for storing the commit log of a data store.

3. Kafka Architecture and Design Principles

Because of limitations in existing systems, we developed a new messaging-based log aggregator Kafka. We first introduce the basic concepts in Kafka. A stream of messages of a particular type is defined by a *topic*. A producer can publish messages to a topic. The published messages are then stored at a set of servers called *brokers*. A consumer can subscribe to one or more topics from the brokers, and consume the subscribed messages by pulling data from the brokers.

Messaging is conceptually simple, and we have tried to make the Kafka API equally simple to reflect this. Instead of showing the exact API, we present some sample code to show how the API is used. The sample code of the producer is given below. A message is defined to contain just a payload of bytes. A user can choose her favorite serialization method to encode a message. For efficiency, the producer can send a set of messages in a single publish request.

Sample producer code:

```
producer = new Producer(…);
message = new Message("test message str".getBytes());
set = new MessageSet(message);
producer.send("topic1", set);
```

To subscribe to a topic, a consumer first creates one or more message streams for the topic. The messages published to that topic will be evenly distributed into these sub-streams. The details about how Kafka distributes the messages are described later in Section 3.2. Each message stream provides an iterator interface over the continual stream of messages being produced. The consumer then iterates over every message in the stream and processes the payload of the message. Unlike traditional iterators, the message stream iterator never terminates. If there are currently no more messages to consume, the iterator blocks until new messages are published to the topic. We support both the point-to-point delivery model in which multiple consumers jointly consume a single copy of all messages in a topic, as well as the publish/subscribe model in which multiple consumers each retrieve its own copy of a topic.

Sample consumer code:

```
streams[] = Consumer.createMessageStreams("topic1", 1)
for (message : streams[0]) {
  bytes = message.payload();
  // do something with the bytes
}
```

The overall architecture of Kafka is shown in Figure 1. Since Kafka is distributed in nature, an Kafka cluster typically consists of multiple brokers. To balance load, a topic is divided into multiple partitions and each broker stores one or more of those partitions. Multiple producers and consumers can publish and retrieve messages at the same time. In Section 3.1, we describe the layout of a single partition on a broker and a few design choices that we selected to make accessing a partition efficient. In Section 3.2, we describe how the producer and the consumer interact with multiple brokers in a distributed setting. We discuss the delivery guarantees of Kafka in Section 3.3.

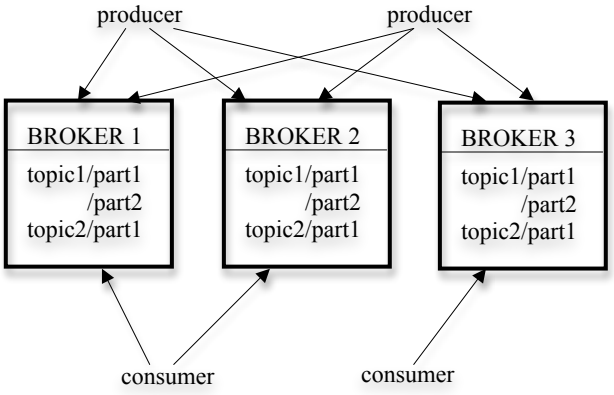

Figure 1. Kafka Architecture

3.1 Efficiency on a Single Partition

We made a few decisions in Kafka to make the system efficient.

Simple storage: Kafka has a very simple storage layout. Each partition of a topic corresponds to a logical log. Physically, a log is implemented as a set of segment files of approximately the same size (e.g., 1GB). Every time a producer publishes a message to a partition, the broker simply appends the message to the last segment file. For better performance, we flush the segment files to disk only after a configurable number of messages have been published or a certain amount of time has elapsed. A message is only exposed to the consumers after it is flushed.

Unlike typical messaging systems, a message stored in Kafka doesn't have an explicit message id. Instead, each message is addressed by its logical offset in the log. This avoids the overhead of maintaining auxiliary, seek-intensive random-access index structures that map the message ids to the actual message locations. Note that our message ids are increasing but not consecutive. To compute the id of the next message, we have to add the length of the current message to its id. From now on, we will use message ids and offsets interchangeably.

A consumer always consumes messages from a particular partition sequentially. If the consumer acknowledges a particular message offset, it implies that the consumer has received all messages prior to that offset in the partition. Under the covers, the consumer is issuing asynchronous pull requests to the broker to have a buffer of data ready for the application to consume. Each pull request contains the offset of the message from which the consumption begins and an acceptable number of bytes to fetch. Each broker keeps in memory a sorted list of offsets, including the offset of the first message in every segment file. The broker locates the segment file where the requested message resides by searching the offset list, and sends the data back to the consumer. After a consumer receives a message, it computes the offset of the next message to consume and uses it in the next pull request. The layout of an Kafka log and the in-memory index is depicted in Figure 2. Each box shows the offset of a message.

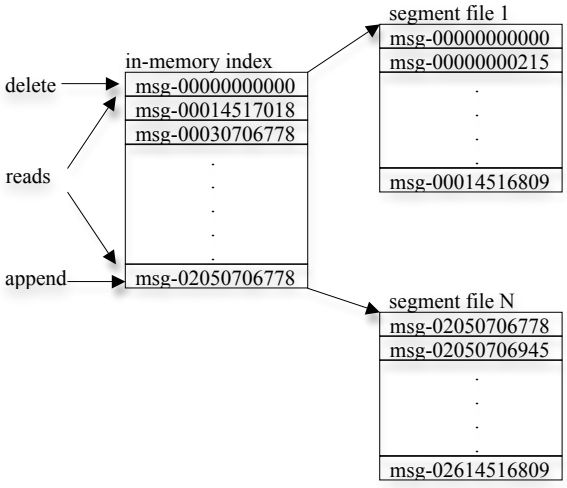

Figure 2. Kafka log

Efficient transfer: We are very careful about transferring data in and out of Kafka. Earlier, we have shown that the producer can submit a set of messages in a single send request. Although the end consumer API iterates one message at a time, under the covers, each pull request from a consumer also retrieves multiple messages up to a certain size, typically hundreds of kilobytes.

Another unconventional choice that we made is to avoid explicitly caching messages in memory at the Kafka layer. Instead, we rely on the underlying file system page cache. This has the main benefit of avoiding double buffering---messages are only cached in the page cache. This has the additional benefit of retaining warm cache even when a broker process is restarted. Since Kafka doesn't cache messages in process at all, it has very little overhead in garbage collecting its memory, making efficient implementation in a VM-based language feasible. Finally, since both the producer and the consumer access the segment files sequentially, with the consumer often lagging the producer by a small amount, normal operating system caching heuristics are very effective (specifically write-through caching and read-ahead). We have found that both the production and the consumption have consistent performance linear to the data size, up to many terabytes of data.

In addition we optimize the network access for consumers. Kafka is a multi-subscriber system and a single message may be consumed multiple times by different consumer applications. A typical approach to sending bytes from a local file to a remote socket involves the following steps: (1) read data from the storage media to the page cache in an OS, (2) copy data in the page cache to an application buffer, (3) copy application buffer to another kernel buffer, (4) send the kernel buffer to the socket. This includes 4 data copying and 2 system calls. On Linux and other Unix operating systems, there exists a sendfile API [5] that can directly transfer bytes from a file channel to a socket channel. This typically avoids 2 of the copies and 1 system call introduced in steps (2) and (3). Kafka exploits the sendfile API to efficiently deliver bytes in a log segment file from a broker to a consumer.

Stateless broker: Unlike most other messaging systems, in Kafka, the information about how much each consumer has consumed is not maintained by the broker, but by the consumer itself. Such a design reduces a lot of the complexity and the overhead on the broker. However, this makes it tricky to delete a message, since a broker doesn't know whether all subscribers have consumed the message. Kafka solves this problem by using a simple time-based SLA for the retention policy. A message is automatically deleted if it has been retained in the broker longer than a certain period, typically 7 days. This solution works well in practice. Most consumers, including the offline ones, finish consuming either daily, hourly, or in real-time. The fact that the performance of Kafka doesn't degrade with a larger data size makes this long retention feasible.

There is an important side benefit of this design. A consumer can deliberately *rewind* back to an old offset and re-consume data. This violates the common contract of a queue, but proves to be an essential feature for many consumers. For example, when there is an error in application logic in the consumer, the application can re-play certain messages after the error is fixed. This is particularly important to ETL data loads into our data warehouse or Hadoop system. As another example, the consumed data may be flushed to a persistent store only periodically (e.g, a full-text indexer). If the consumer crashes, the unflushed data is lost. In this case, the consumer can checkpoint the smallest offset of the unflushed messages and re-consume from that offset when it's restarted. We note that rewinding a consumer is much easier to support in the pull model than the push model.

3.2 Distributed Coordination

We now describe how the producers and the consumers behave in a distributed setting. Each producer can publish a message to either a randomly selected partition or a partition semantically determined by a partitioning key and a partitioning function. We will focus on how the consumers interact with the brokers.

Kafka has the concept of *consumer groups*. Each consumer group consists of one or more consumers that jointly consume a set of subscribed topics, i.e., each message is delivered to only one of the consumers within the group. Different consumer groups each independently consume the full set of subscribed messages and no coordination is needed across consumer groups. The consumers

within the same group can be in different processes or on different machines. Our goal is to divide the messages stored in the brokers evenly among the consumers, without introducing too much coordination overhead.

Our first decision is to make a partition within a topic the smallest unit of parallelism. This means that at any given time, all messages from one partition are consumed only by a single consumer within each consumer group. Had we allowed multiple consumers to simultaneously consume a single partition, they would have to coordinate who consumes what messages, which necessitates locking and state maintenance overhead. In contrast, in our design consuming processes only need co-ordinate when the consumers rebalance the load, an infrequent event. In order for the load to be truly balanced, we require many more partitions in a topic than the consumers in each group. We can easily achieve this by over partitioning a topic.

The second decision that we made is to not have a central "master" node, but instead let consumers coordinate among themselves in a decentralized fashion. Adding a master can complicate the system since we have to further worry about master failures. To facilitate the coordination, we employ a highly available consensus service Zookeeper [10]. Zookeeper has a very simple, file system like API. One can create a path, set the value of a path, read the value of a path, delete a path, and list the children of a path. It does a few more interesting things: (a) one can register a watcher on a path and get notified when the children of a path or the value of a path has changed; (b) a path can be created as *ephemeral* (as oppose to *persistent*), which means that if the creating client is gone, the path is automatically removed by the Zookeeper server; (c) zookeeper replicates its data to multiple servers, which makes the data highly reliable and available.

Kafka uses Zookeeper for the following tasks: (1) detecting the addition and the removal of brokers and consumers, (2) triggering a rebalance process in each consumer when the above events happen, and (3) maintaining the consumption relationship and keeping track of the consumed offset of each partition. Specifically, when each broker or consumer starts up, it stores its information in a broker or consumer registry in Zookeeper. The broker registry contains the broker's host name and port, and the set of topics and partitions stored on it. The consumer registry includes the consumer group to which a consumer belongs and the set of topics that it subscribes to. Each consumer group is associated with an ownership registry and an offset registry in Zookeeper. The ownership registry has one path for every subscribed partition and the path value is the id of the consumer currently consuming from this partition (we use the terminology that the consumer *owns* this partition). The offset registry stores for each subscribed partition, the offset of the last consumed message in the partition.

The paths created in Zookeeper are ephemeral for the broker registry, the consumer registry and the ownership registry, and persistent for the offset registry. If a broker fails, all partitions on it are automatically removed from the broker registry. The failure of a consumer causes it to lose its entry in the consumer registry and all partitions that it owns in the ownership registry. Each consumer registers a Zookeeper watcher on both the broker registry and the consumer registry, and will be notified whenever a change in the broker set or the consumer group occurs.

During the initial startup of a consumer or when the consumer is notified about a broker/consumer change through the watcher, the consumer initiates a rebalance process to determine the new subset of partitions that it should consume from. The process is described in Algorithm 1. By reading the broker and the consumer registry from Zookeeper, the consumer first computes the set (P_T) of partitions available for each subscribed topic T and the set (C_T) of consumers subscribing to T. It then range-partitions P_T into $|C_T|$ chunks and deterministically picks one chunk to own. For each partition the consumer picks, it writes itself as the new owner of the partition in the ownership registry. Finally, the consumer begins a thread to pull data from each owned partition, starting from the offset stored in the offset registry. As messages get pulled from a partition, the consumer periodically updates the latest consumed offset in the offset registry.

Algorithm 1: rebalance process for consumer C_i in group G

```
For each topic T that C_i subscribes to {
  remove partitions owned by C_i from the ownership registry
  read the broker and the consumer registries from Zookeeper
  compute P_T = partitions available in all brokers under topic T
  compute C_T = all consumers in G that subscribe to topic T
  sort P_T and C_T
  let j be the index position of C_i in C_T and let N = |P_T|/|C_T|
  assign partitions from j*N to (j+1)*N - 1 in P_T to consumer C_i
  for each assigned partition p {
    set the owner of p to C_i in the ownership registry
    let O_p = the offset of partition p stored in the offset registry
    invoke a thread to pull data in partition p from offset O_p
  }
}
```

When there are multiple consumers within a group, each of them will be notified of a broker or a consumer change. However, the notification may come at slightly different times at the consumers. So, it is possible that one consumer tries to take ownership of a partition still owned by another consumer. When this happens, the first consumer simply releases all the partitions that it currently owns, waits a bit and retries the rebalance process. In practice, the rebalance process often stabilizes after only a few retries.

When a new consumer group is created, no offsets are available in the offset registry. In this case, the consumers will begin with either the smallest or the largest offset (depending on a configuration) available on each subscribed partition, using an API that we provide on the brokers.

3.3 Delivery Guarantees

In general, Kafka only guarantees at-least-once delivery. Exactly-once delivery typically requires two-phase commits and is not necessary for our applications. Most of the time, a message is delivered exactly once to each consumer group. However, in the case when a consumer process crashes without a clean shutdown, the consumer process that takes over those partitions owned by the failed consumer may get some duplicate messages that are after the last offset successfully committed to zookeeper. If an application cares about duplicates, it must add its own de-duplication logic, either using the offsets that we return to the consumer or some unique key within the message. This is usually a more cost-effective approach than using two-phase commits.

Kafka guarantees that messages from a single partition are delivered to a consumer in order. However, there is no guarantee on the ordering of messages coming from different partitions.

To avoid log corruption, Kafka stores a CRC for each message in the log. If there is any I/O error on the broker, Kafka runs a recovery process to remove those messages with inconsistent CRCs. Having the CRC at the message level also allows us to check network errors after a message is produced or consumed.

If a broker goes down, any message stored on it not yet consumed becomes unavailable. If the storage system on a broker is permanently damaged, any unconsumed message is lost forever. In the future, we plan to add built-in replication in Kafka to redundantly store each message on multiple brokers.

4. Kafka Usage at LinkedIn

In this section, we describe how we use Kafka at LinkedIn. Figure 3 shows a simplified version of our deployment. We have one Kafka cluster co-located with each datacenter where our user-facing services run. The frontend services generate various kinds of log data and publish it to the local Kafka brokers in batches. We rely on a hardware load-balancer to distribute the publish requests to the set of Kafka brokers evenly. The online consumers of Kafka run in services within the same datacenter.

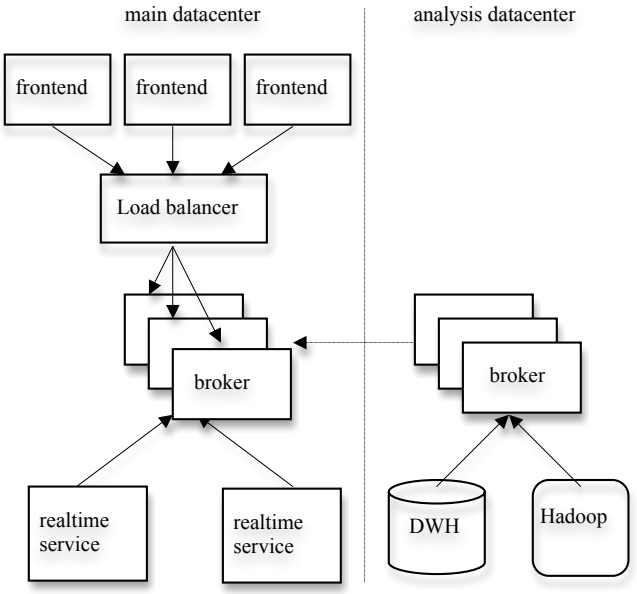

Figure 3. Kafka Deployment

We also deploy a cluster of Kafka in a separate datacenter for offline analysis, located geographically close to our Hadoop cluster and other data warehouse infrastructure. This instance of Kafka runs a set of embedded consumers to pull data from the Kafka instances in the live datacenters. We then run data load jobs to pull data from this replica cluster of Kafka into Hadoop and our data warehouse, where we run various reporting jobs and analytical process on the data. We also use this Kafka cluster for prototyping and have the ability to run simple scripts against the raw event streams for ad hoc querying. Without too much tuning, the end-to-end latency for the complete pipeline is about 10 seconds on average, good enough for our requirements.

Currently, Kafka accumulates hundreds of gigabytes of data and close to a billion messages per day, which we expect will grow significantly as we finish converting legacy systems to take advantage of Kafka. More types of messages will be added in the future. The rebalance process is able to automatically redirect the consumption when the operation staffs start or stop brokers for software or hardware maintenance.

Our tracking also includes an auditing system to verify that there is no data loss along the whole pipeline. To facilitate that, each message carries the timestamp and the server name when they are generated. We instrument each producer such that it periodically generates a monitoring event, which records the number of messages published by that producer for each topic within a fixed time window. The producer publishes the monitoring events to Kafka in a separate topic. The consumers can then count the number of messages that they have received from a given topic and validate those counts with the monitoring events to validate the correctness of data.

Loading into the Hadoop cluster is accomplished by implementing a special Kafka input format that allows MapReduce jobs to directly read data from Kafka. A MapReduce job loads the raw data and then groups and compresses it for efficient processing in the future. The stateless broker and client-side storage of message offsets again come into play here, allowing the MapReduce task management (which allows tasks to fail and be restarted) to handle the data load in a natural way without duplicating or losing messages in the event of a task restart. Both data and offsets are stored in HDFS only on the successful completion of the job.

We chose to use Avro [2] as our serialization protocol since it is efficient and supports schema evolution. For each message, we store the id of its Avro schema and the serialized bytes in the payload. This schema allows us to enforce a contract to ensure compatibility between data producers and consumers. We use a lightweight schema registry service to map the schema id to the actual schema. When a consumer gets a message, it looks up in the schema registry to retrieve the schema, which is used to decode the bytes into an object (this lookup need only be done once per schema, since the values are immutable).

5. Experimental Results

We conducted an experimental study, comparing the performance of Kafka with Apache ActiveMQ v5.4 [1], a popular open-source implementation of JMS, and RabbitMQ v2.4 [16], a message system known for its performance. We used ActiveMQ's default persistent message store KahaDB. Although not presented here, we also tested an alternative AMQ message store and found its performance very similar to that of KahaDB. Whenever possible, we tried to use comparable settings in all systems.

We ran our experiments on 2 Linux machines, each with 8 2GHz cores, 16GB of memory, 6 disks with RAID 10. The two machines are connected with a 1Gb network link. One of the machines was used as the broker and the other machine was used as the producer or the consumer.

Producer Test: We configured the broker in all systems to asynchronously flush messages to its persistence store. For each system, we ran a single producer to publish a total of 10 million messages, each of 200 bytes. We configured the Kafka producer to send messages in batches of size 1 and 50. ActiveMQ and RabbitMQ don't seem to have an easy way to batch messages and we assume that it used a batch size of 1. The results are shown in Figure 4. The x-axis represents the amount of data sent to the broker over time in MB, and the y-axis corresponds to the producer throughput in messages per second. On average, Kafka can publish messages at the rate of 50,000 and 400,000 messages per second for batch size of 1 and 50, respectively. These numbers

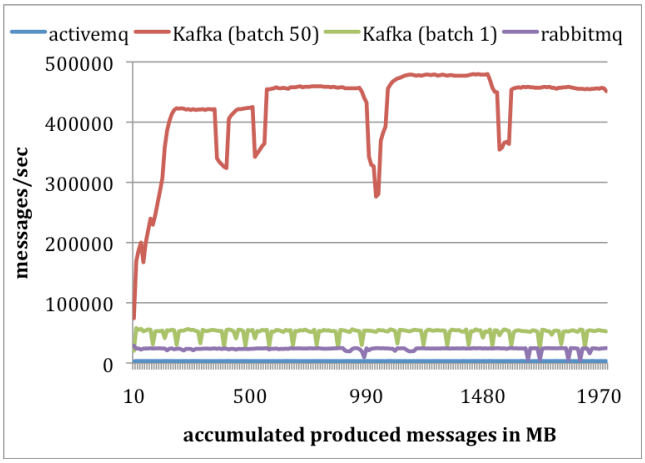

Figure 4. Producer Performance

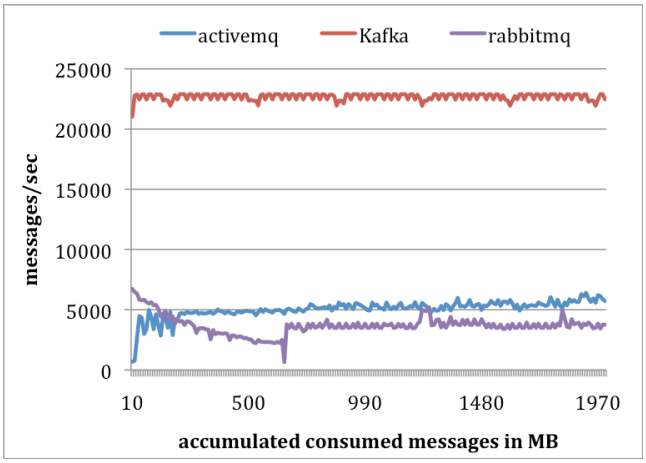

Figure 5. Consumer Performance

are orders of magnitude higher than that of ActiveMQ, and at least 2 times higher than RabbitMQ.

There are a few reasons why Kafka performed much better. First, the Kafka producer currently doesn't wait for acknowledgements from the broker and sends messages as faster as the broker can handle. This significantly increased the throughput of the publisher. With a batch size of 50, a single Kafka producer almost saturated the 1Gb link between the producer and the broker. This is a valid optimization for the log aggregation case, as data must be sent asynchronously to avoid introducing any latency into the live serving of traffic. We note that without acknowledging the producer, there is no guarantee that every published message is actually received by the broker. For many types of log data, it is desirable to trade durability for throughput, as long as the number of dropped messages is relatively small. However, we do plan to address the durability issue for more critical data in the future.

Second, Kafka has a more efficient storage format. On average, each message had an overhead of 9 bytes in Kafka, versus 144 bytes in ActiveMQ. This means that ActiveMQ was using 70% more space than Kafka to store the same set of 10 million messages. One overhead in ActiveMQ came from the heavy message header, required by JMS. Another overhead was the cost of maintaining various indexing structures. We observed that one of the busiest threads in ActiveMQ spent most of its time accessing a B-Tree to maintain message metadata and state. Finally, batching greatly improved the throughput by amortizing the RPC overhead. In Kafka, a batch size of 50 messages improved the throughput by almost an order of magnitude.

Consumer Test: In the second experiment, we tested the performance of the consumer. Again, for all systems, we used a single consumer to retrieve a total of 10 millions messages. We configured all systems so that each pull request should prefetch approximately the same amount data---up to 1000 messages or about 200KB. For both ActiveMQ and RabbitMQ, we set the consumer acknowledge mode to be automatic. Since all messages fit in memory, all systems were serving data from the page cache of the underlying file system or some in-memory buffers. The results are presented in Figure 5.

On average, Kafka consumed 22,000 messages per second, more than 4 times that of ActiveMQ and RabbitMQ. We can think of several reasons. First, since Kafka has a more efficient storage format, fewer bytes were transferred from the broker to the consumer in Kafka. Second, the broker in both ActiveMQ and RabbitMQ had to maintain the delivery state of every message. We observed that one of the ActiveMQ threads was busy writing KahaDB pages to disks during this test. In contrast, there were no disk write activities on the Kafka broker. Finally, by using the sendfile API, Kafka reduces the transmission overhead.

We close the section by noting that the purpose of the experiment is not to show that other messaging systems are inferior to Kafka. After all, both ActiveMQ and RabbitMQ have more features than Kafka. The main point is to illustrate the potential performance gain that can be achieved by a specialized system.

6. Conclusion and Future Works

We present a novel system called Kafka for processing huge volume of log data streams. Like a messaging system, Kafka employs a pull-based consumption model that allows an application to consume data at its own rate and rewind the consumption whenever needed. By focusing on log processing applications, Kafka achieves much higher throughput than conventional messaging systems. It also provides integrated distributed support and can scale out. We have been using Kafka successfully at LinkedIn for both offline and online applications.

There are a number of directions that we'd like to pursue in the future. First, we plan to add built-in replication of messages across multiple brokers to allow durability and data availability guarantees even in the case of unrecoverable machine failures. We'd like to support both asynchronous and synchronous replication models to allow some tradeoff between producer latency and the strength of the guarantees provided. An application can choose the right level of redundancy based on its requirement on durability, availability and throughput. Second, we want to add some stream processing capability in Kafka. After retrieving messages from Kafka, real time applications often perform similar operations such as window-based counting and joining each message with records in a secondary store or with messages in another stream. At the lowest level this is supported by semantically partitioning messages on the join key during publishing so that all messages sent with a particular key go to the same partition and hence arrive at a single consumer process. This provides the foundation for processing distributed streams across a cluster of consumer machines. On top of this we feel a library of helpful stream utilities, such as different windowing functions or join techniques will be beneficial to this kind of applications.

7. REFERENCES

[1] http://activemq.apache.org/

[2] http://avro.apache.org/

[3] Cloudera's Flume, https://github.com/cloudera/flume

[4] http://developer.yahoo.com/blogs/hadoop/posts/2010/06/enabling_hadoop_batch_processi_1/

[5] Efficient data transfer through zero copy: https://www.ibm.com/developerworks/linux/library/j-zerocopy/

[6] Facebook's Scribe, http://www.facebook.com/note.php?note_id=32008268919

[7] IBM Websphere MQ: http://www-01.ibm.com/software/integration/wmq/

[8] http://hadoop.apache.org/

[9] http://hadoop.apache.org/hdfs/

[10] http://hadoop.apache.org/zookeeper/

[11] http://www.slideshare.net/cloudera/hw09-hadoop-based-data-mining-platform-for-the-telecom-industry

[12] http://www.slideshare.net/prasadc/hive-percona-2009

[13] https://issues.apache.org/jira/browse/ZOOKEEPER-775

[14] JAVA Message Service: http://download.oracle.com/javaee/1.3/jms/tutorial/1_3_1-fcs/doc/jms_tutorialTOC.html.

[15] Oracle Enterprise Messaging Service: http://www.oracle.com/technetwork/middleware/ias/index-093455.html

[16] http://www.rabbitmq.com/

[17] TIBCO Enterprise Message Service: http://www.tibco.com/products/soa/messaging/

[18] Kafka, http://sna-projects.com/kafka/