Apache Kafka

//TODO Insert Logo

About Me

David Arthur

http://mumrah.github.io/

- Software Engineer at LucidWorks
- Open source contributor
- Gardener
- Dad

TOC

- Project overview
- Architecture
- Implementation
- Miscellany

Project info

- http://kafka.apache.org
- Written in Scala
- Open sourced by LinkedIn SNA in 2011
- Soon after entered Apache Incubator
- Not just an idle "open source donation"
- Active development, 0.8 out now
- Apache TLP late 2012, 13 committers
- Still no logo

12 word overview

Apache Kafka is publish-subscribe messaging rethought as a distributed commit log

Ok...

Kafka is a persistent, distributed, replicated pub/sub messaging system. Publishers send messages to a cluster of brokers. The brokers persist the messages to disk.

Consumers then request a range of messages using an (offset, length) style API. Use of NIO `FileChannel` allows for very fast transfer of data in and out of the system.

Highlights

- ZooKeeper for Broker coordination
- Configurable Producer acks
- Consumer Groups
- TTL persistence
- Sync/Async producer API
- Durable
- Scalable
- Fast
- <Additional buzzwords>

Motivation

- Activity stream processing (user interactions)
- Batch latency was too high
- Existing queues handle large volumes of (unconsumed) data poorly - durability is expensive
- Need something fast and durable

Key design choices

- Pub/sub messaging pattern
- Messages are persistent
- Everything is distributed - producers, brokers, consumers, the queue itself
- Consumers maintain their own state (i.e., "dumb" brokers)
- Throughput is key

TOC

- Project overview
- Architecture
- Implementation
- Miscellany

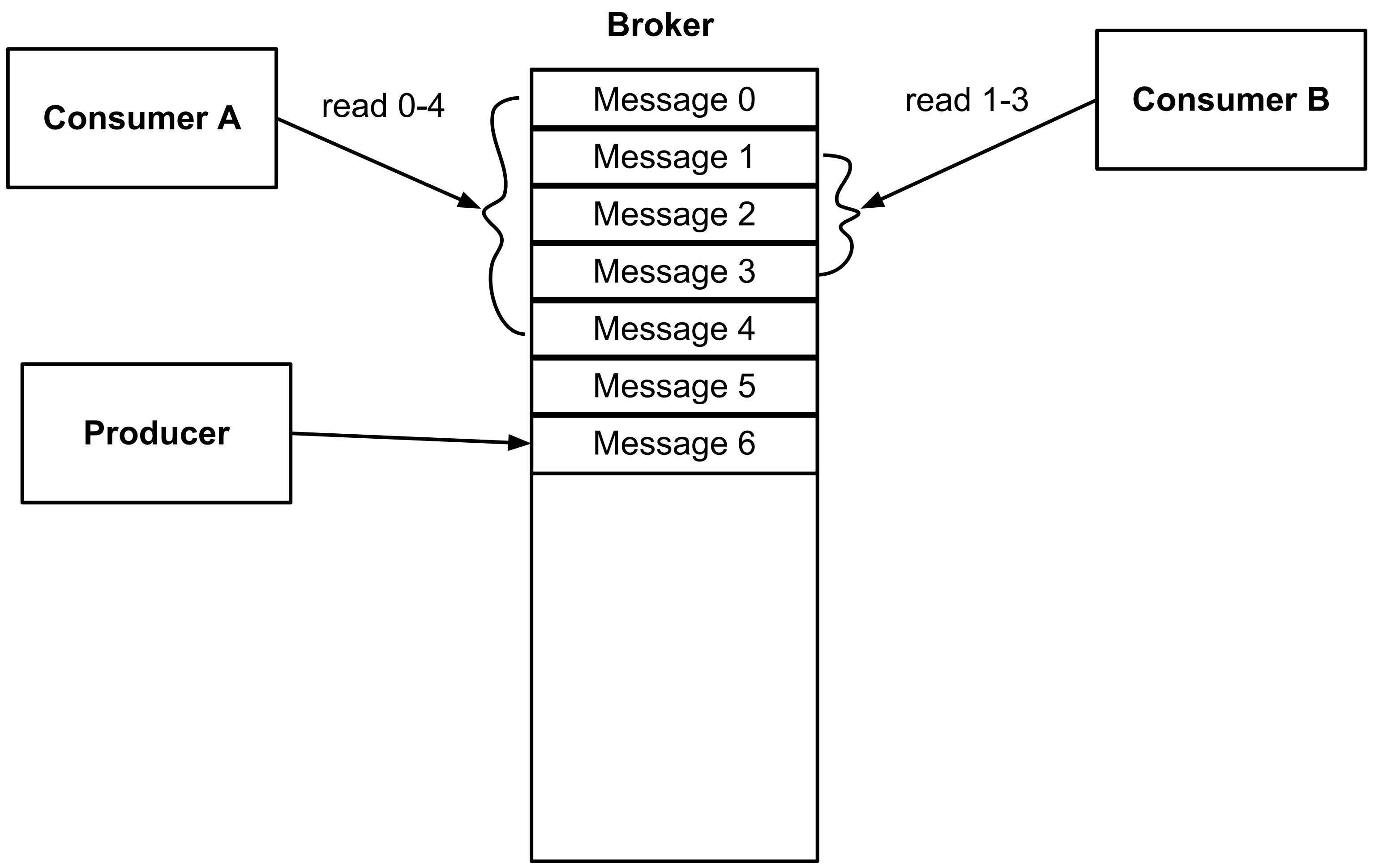
Brokers

- Receive messages from Producers (push), deliver messages to Consumers (pull)
- Responsible for persisting the messages for some time
- Relatively lightweight - mostly just handling TCP connections and keeping open file handles to the queue files

Log-based queue

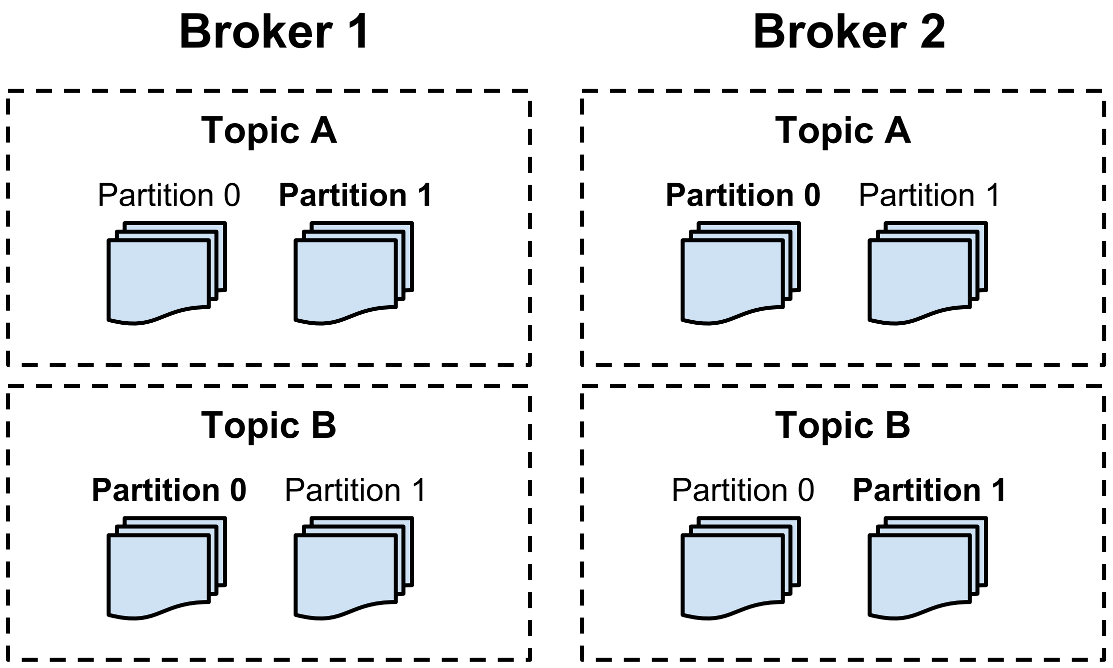
Messages are persisted to append-only log files by the broker. Producers are appending to these log files (sequential write), and consumers are reading a range of these files (sequential reads).

Log-based queue



Topics

Topics are queues. They are logical collections of partitions (the physical files). A broker contains some of the partitions for a topic



Replication

Partitions of a topic are replicated. One broker is the "leader" of a partition. All writes and reads *must* go to the leader. Replicas exist for fault-tolerance, not scalability.

When writing, messages can be synchronously written to N replicas (depending on the producer's ACKiness)

Producers

Producers are responsible for load balancing messages to the various brokers. They can discover all brokers from any one broker.

In 0.7, producers are fire-and-forget. In 0.8, there are 3 ack levels:

- No ack (0)
- Ack from N replicas (1..N)
- Ack from all replicas (-1)

Consumers

Consumers request a range of messages from a Broker. They are responsible for their own state

Default implementation uses ZooKeeper to manage state. In 0.8.1, Brokers will expose an API for offset management to remove direct communication between consumers and ZooKeeper (a good thing).

Result?

- Brokers keep very little state, mostly just open file pointers and connections
- Can scale to thousands of producers and consumers*
- Stable performance, good scalability
- In 0.7, 50MB/s producer throughput, 100MB/s consumer throughput
- No numbers yet from 0.8, but the same comparable (with acks=0)

TOC

- Project overview
- Architecture
- Implementation
- Miscellany

High-level producer

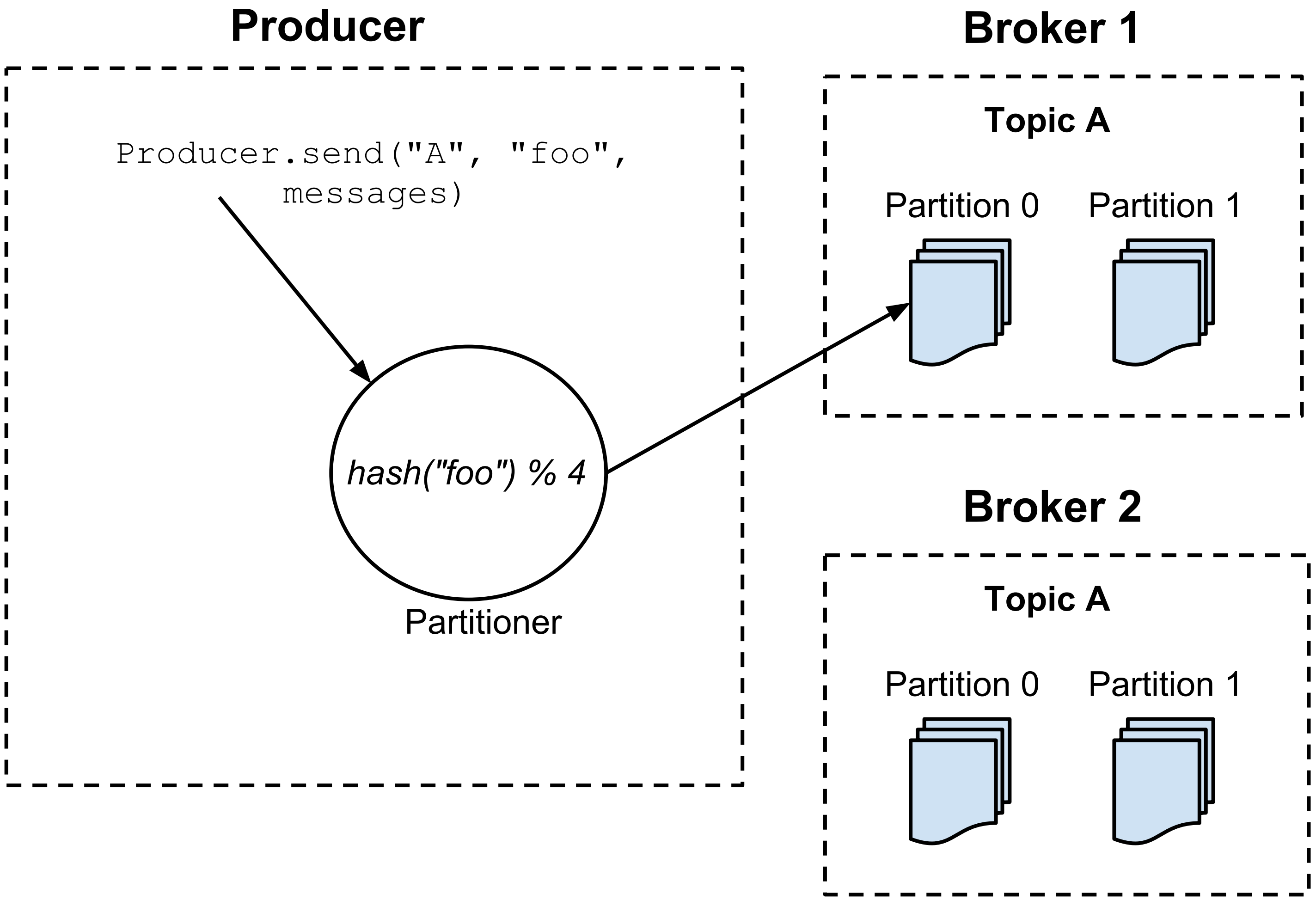
API

```
Producer#send(KeyedMessage<K,V> datum)

KeyedMessage<K,V>(String topic, K key,
                  V value)
```

The Producer class determines where a message is sent based on the routing key. If a null key is given, the message is sent to a random partition

Message routing



Message routing

- Producers can be configured with a custom routing function (implementing the Partitioner interface)
- Default is hash-mod
- One side effect of routing is that there is no total ordering for a topic, but there is within a partition (this is actually really useful)

(Partially) Ordered Messages

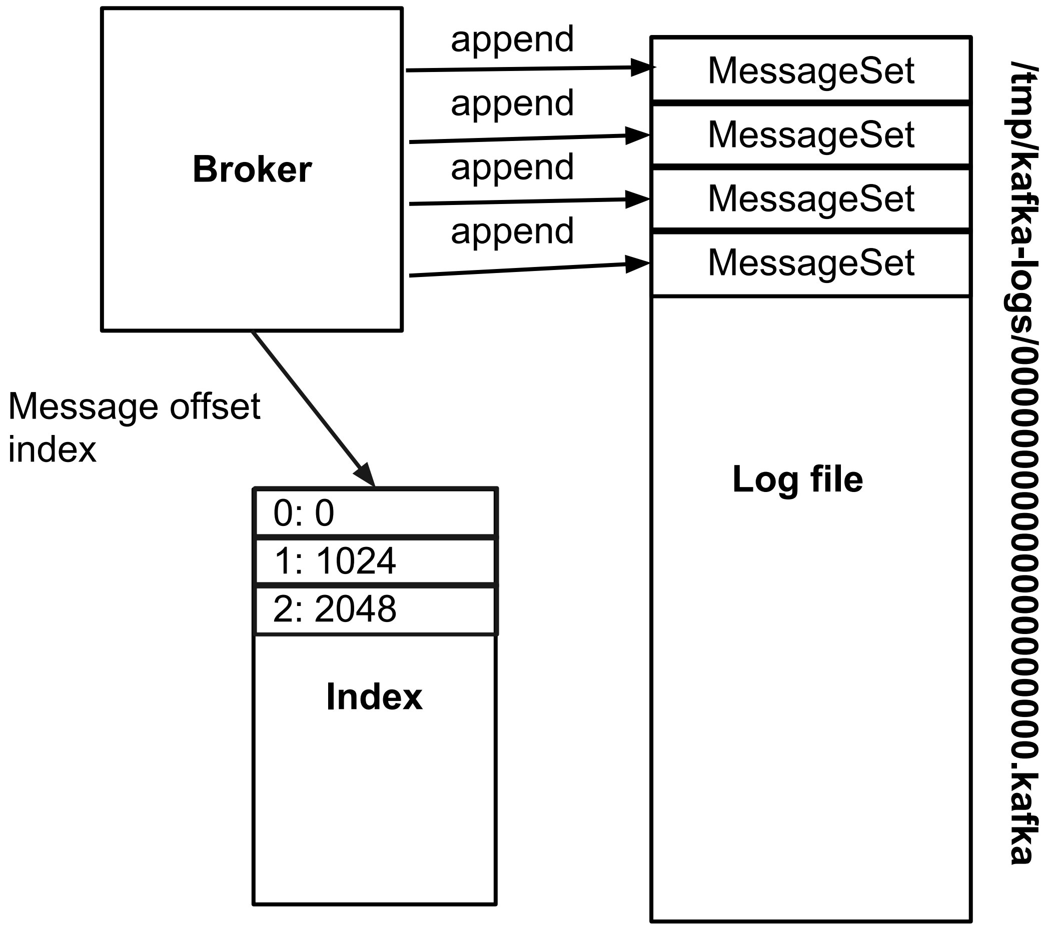
Consider a system that is processing updates. If you partition the messages based on the primary key you guarantee all messages for a given key end up in the same partition. Since Kafka guarantees ordering of the messages at the partition level, your updates will be processed in the correct sequence.

Persistent Messages

- MessageSets received by the broker and flushed to append-only log files
- Simple log format (next slide)
- Zero-copy (i.e., sendfile) for file to socket transfer
- Log files are *not* kept forever

Log Format

`ByteBufferMessageSet#writeTo(FileChannel)`



Message sets

- To maximize throughput, the same binary format for messages is used throughout the system (producer API, consumer API, and log file format)
- Broker only decodes far enough to read the checksum and validate it, then writes it to the disk

Compressed message sets or, message batching

- The value of a Message can be a compressed MessageSet

Message(value=gzip(MessageSet(...)))

- Useful for increasing throughput (especially if you are buffering messages in the producer)
- Can also be used as a way to atomically write multiple messages

Zero-copy

Reading data out of Kafka is super fast thanks to `java.nio.channels.FileChannel#transferTo`. This method uses the "sendfile" system call which allows for very efficient transfer of data from a file to another file (including sockets).

This optimization is what allows us to pull ~100MB/s out of a single broker

High-level Consumer

API

```
Map topicMap = Collections.singletonMap("topic", 2);
Map<String,List<KafkaStream>> streams =
 Consumer.createJavaConsumerConnector(topicMap);
```

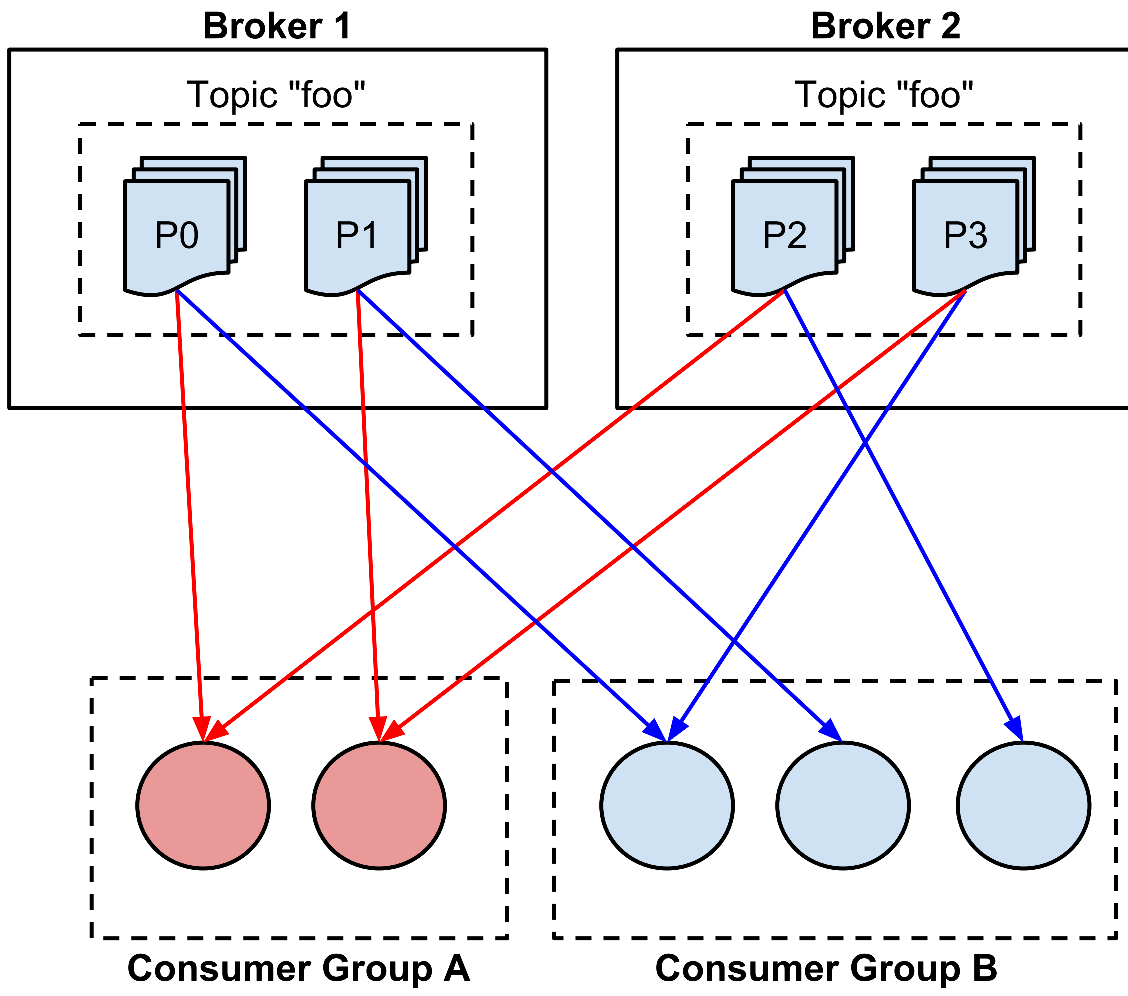
- KafkaStream is an iterable
- Blocking or non-blocking behavior
- Auto-commit offset, or manual commit
- Participate in a "consumer group"

Consumer Groups

Multiple high-level consumers can participate in a single "consumer group". A consumer group is coordinated using ZooKeeper, so it can span multiple machines. In a group, each partition will be consumed by exactly one consumer (i.e., KafkaStream).

This allows for broadcast or pub/sub type of messaging pattern

Consumer Groups



TOC

- Project overview
- Architecture
- Implementation
- Miscellany

But, I already have a MQ

Many people with distributed systems already have something like JMS, RabbitMQ, Kestrel, or ØMQ in place.

These are all different systems with different implementation details and semantics. Decide what features you need, and then chose a MQ - not the other way around :)

RabbitMQ discussion: http://bit.ly/140ZMOx

Caveats

- Not designed for large payloads. Decoding is done for a whole message (no streaming decoding).
- Rebalancing can screw things up if you're doing any aggregation in the consumer by the partition key
- Number of partitions cannot be easily changed (chose wisely)
- Lots of topics can hurt I/O performance

Clients

- Many exist, some more complete than others
- No official clients (other than Java/Scala)
- Community maintained
- Most lack support for high-level consumer due to ZooKeeper dependency (it's tricky)
- Excellent Python client: https://github.com/mumrah/kafka-python (yes, I wrote it)
- Only Python, C, and Clojure support the 0.8 protocol

Hadoop InputFormat

- InputFormat/OutputFormat implementations
- Uses low-level consumer, no offsets in ZK (they are stored on HDFS instead)
- Useful for long term storage of messages
- We use this!
- LinkedIn released Camus, a Kafka to HDFS pipeline

Integration

A few good integration points. I expect more to show up as Kafka gains adoption

- log4j appender (in Kafka itself)
- storm-kafka (in storm-contrib)
- camel-kafka
- LogStash (loges)

More listed on Kafka wiki http://bit.ly/1btZz8p

Applications
Log Aggregation

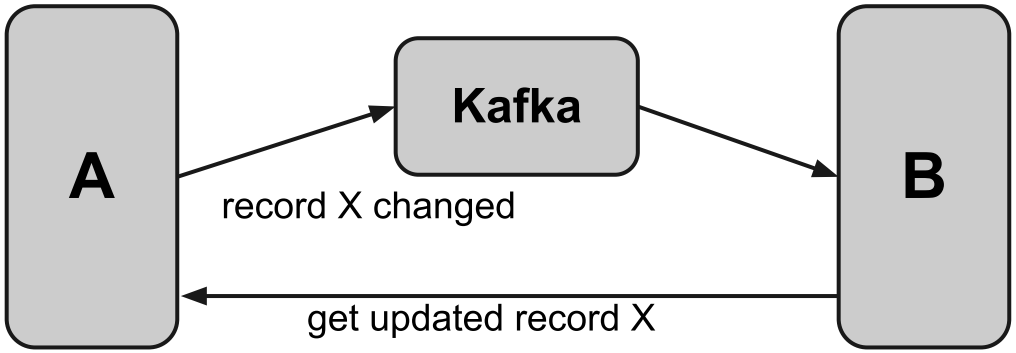
Many applications running on many machines. You want centralized logging, but don't want to install an agent (Flume, etc).

Kafka includes a log4j appender

```
<appender class="kafka.producer.KafkaLog4jAppender" name="kafka-solr">
  <param name="zkConnect" value="localhost:2181"/>
  <param name="topic" value="solr-logs"/>
  <layout class="org.apache.log4j.PatternLayout">
    <param value="%d{ISO8601} %p %c %m" name="ConversionPattern"/>
  </layout>
</appender>
```

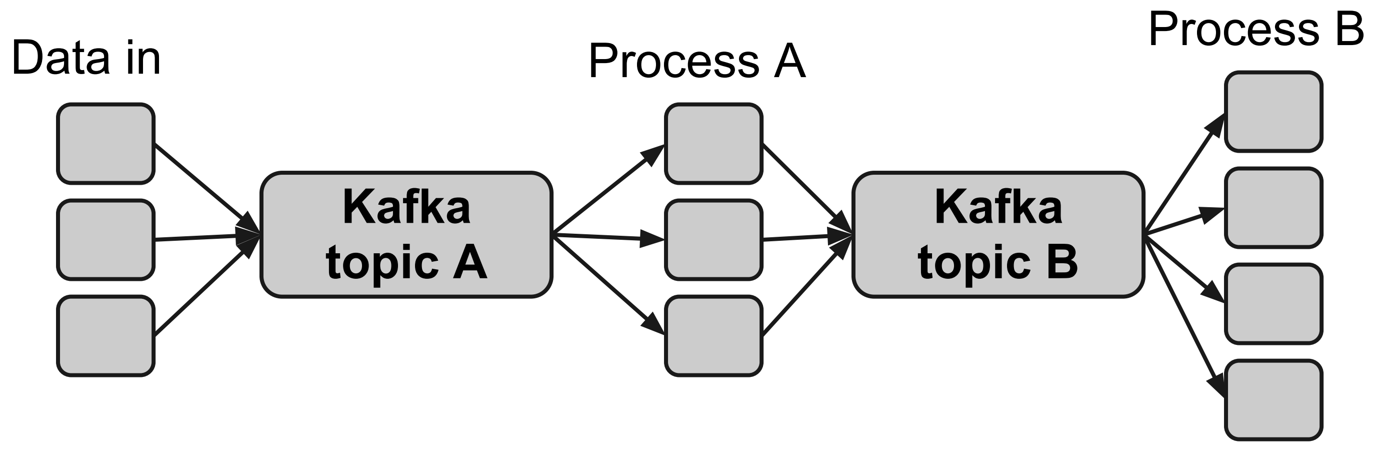
Applications
Notifications

Store data into data store A, need to sync it to data store B. Suppose you can send a message to Kafka when an update happens in A



Applications
Stream Processing

Using a stream processing tool like Storm or Apache Camel, Kafka provides an excellent backbone.



Bonus Slides
Stats

LinkedIn stats:

- Peak writes per second: 460k
- Average writes per day: 28 billion
- Average reads per second: 2.3 million
- ~700 topics
- Thousands of producers
- ~1000 consumers

Bonus Slides

Logos!

Being heatedly debated in JIRA as we speak!

KAFKA

kafka

KAFKA

KAFKA

KAFKA

Links!

- Slides: http://bit.ly/kafka-trihug
- Kafka Project: http://kafka.apache.org/
- Kafka Code: https://github.com/apache/kafka
- LinkedIn Camus: https://github.com/linkedin/camus
- Zero-Copy IBM article: http://ibm.co/gsETNm
- LinkedIn Kafka paper: http://bit.ly/mC8TLS
- LinkedIn blog post on replication: http://linkd.in/YAWslH