



APACHE KAFKA

DISTRIBUTED STREAMING PLATFORM

Bin Jiang

04/22/2017

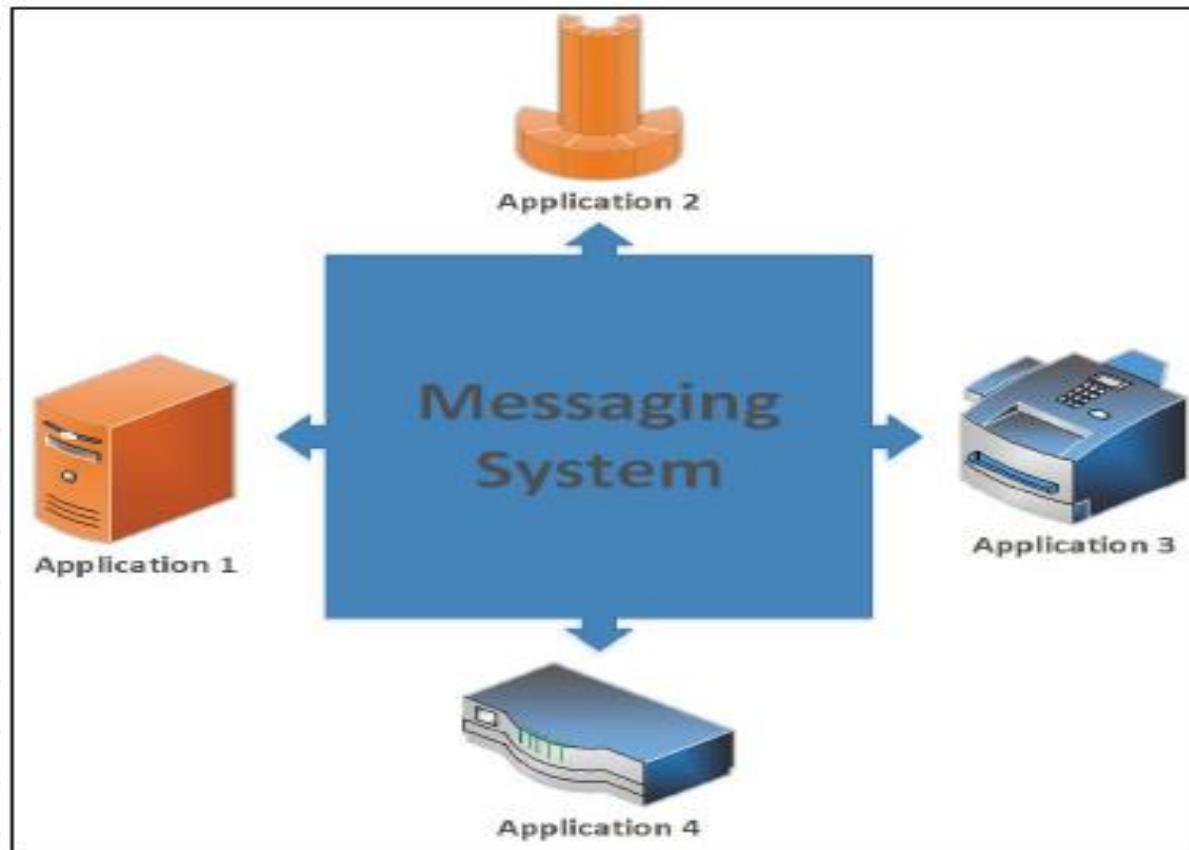
Objective

- Understand Kafka architecture and Kafka configuration on cluster
- Master various Kafka components – consumer, producer and brokers
- Perform different operations on topic
- Integrate Kafka with various consumers
- Play with Kafka partitions and distribute data between them
- Understand insights of high & low level Kafka APIs
- Learn the concepts of latest version of Kafka
- Learn Zookeeper
- Learn Kafka Best Practices
- Master balancing of Kafka Cluster
- Understand Replication and its importance in Kafka
- Develop Live Kafka Project

Principles of Messaging Systems

- Loose coupling
- Common interface definitions
- Latency
- Reliability

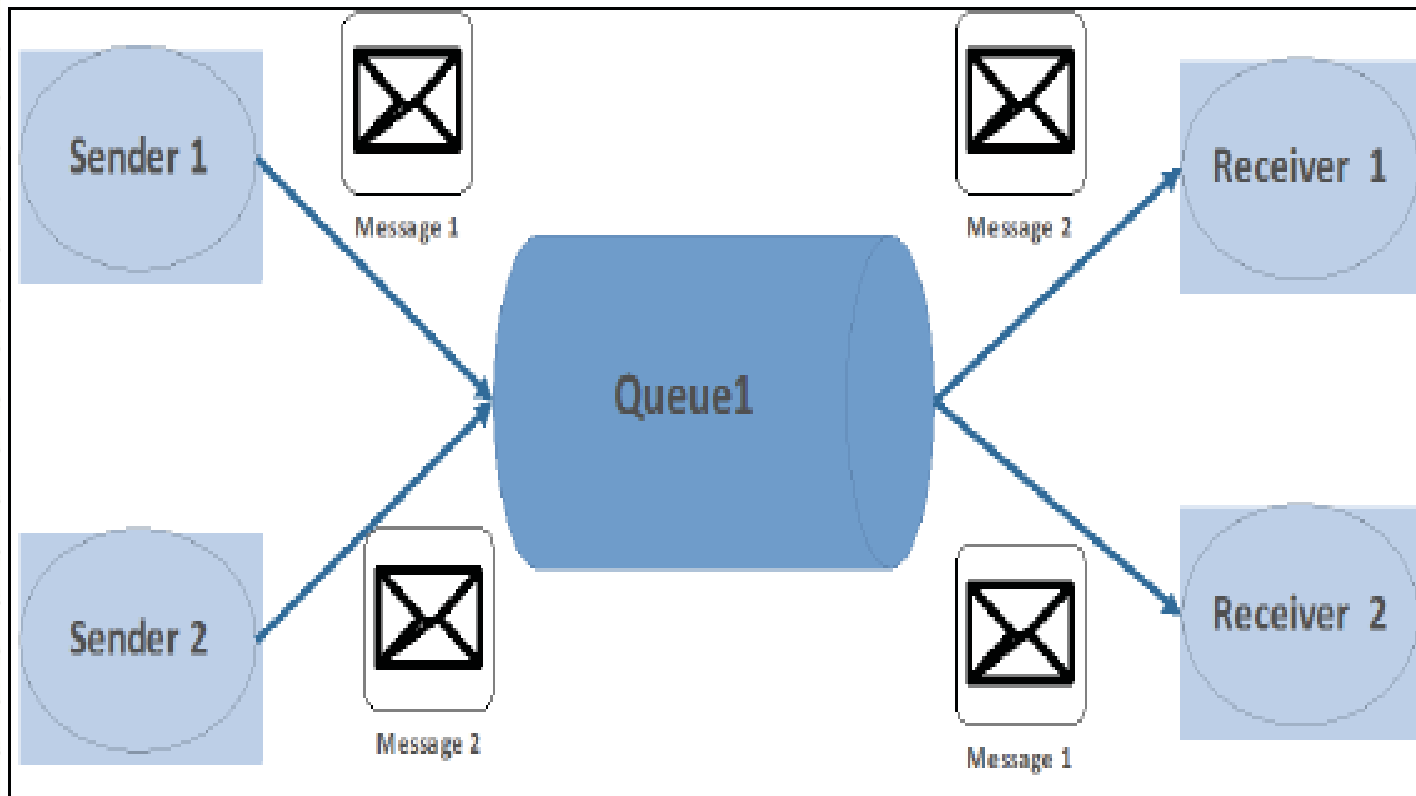
Understanding Messaging Systems



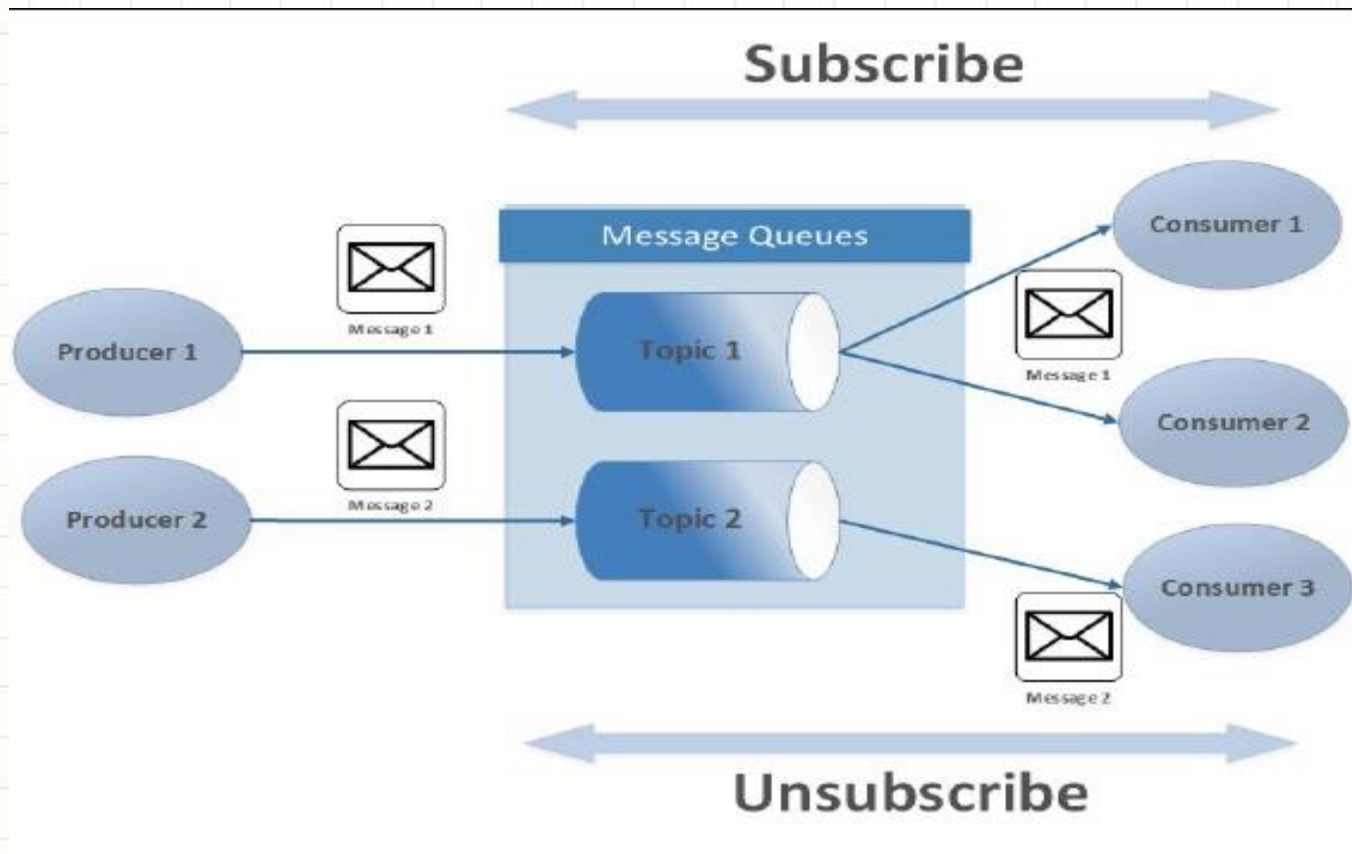
Understanding Messaging Systems

- Message queues
- Messages
- Sender (producer)
- Receiver (consumer)
- Data transmission protocols

Point to Point



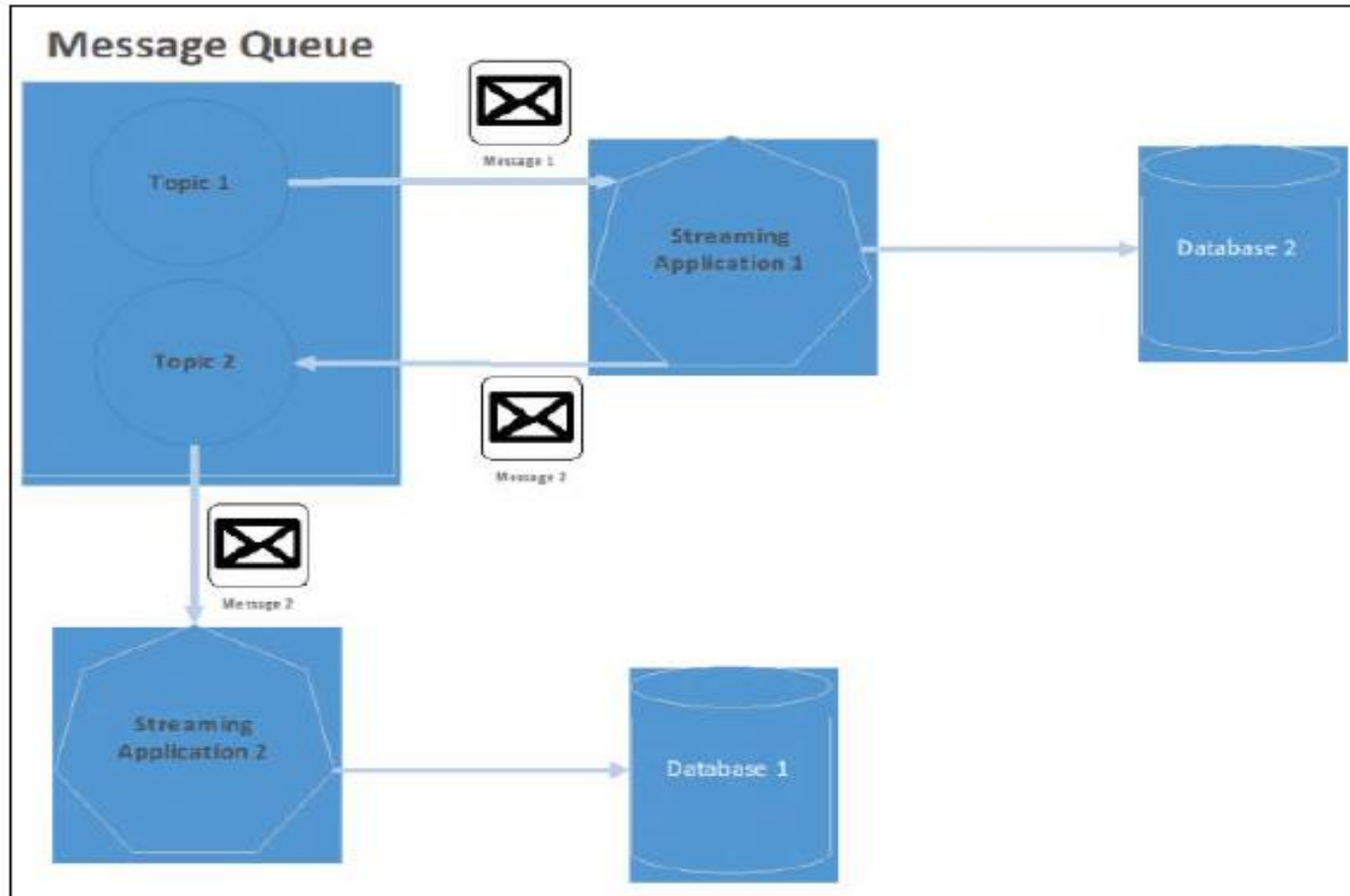
Publish Subscribe



Messaging Systems and Big Data

- Ingestion layer
- Processing layer
- Consumption layer

Messaging Systems and Big Data



Streaming Application

- High consuming rate
- Guaranteed delivery
- Persisting capability
- Security
- Fault tolerance

Overview of Kafka

Apache Kafka is an open source, distributed, partitioned, and replicated commit-log-based publish-subscribe messaging system.

Kafka provides a real-time publish-subscribe solution that overcomes the challenges of consuming the real-time and batch data volumes that may grow in order of magnitude to be larger than the real data. Kafka also supports parallel data loading in the Hadoop systems

Importance for messaging queue

- Scalability
- Load balancing
- Reduced coupling
- Reliability

Conventional problems

- Have strong or loose ordering
- Have idempotent put
- Have more queues than what can fit on a single machine?
- How many machines can crash before potentially lose data?
- Tolerate disconnection?
- Reconcile data automatically when the disconnection is fixed?
- Guarantee delivery when clients can crash
- Guarantee that the same message is not delivered more than once
- Node crash at any given point, come back up, and not send out junk
- Add nodes to, or remove nodes from, a running cluster without down time
- Upgrade nodes in a running cluster without down time?
- Run without problems on heterogeneous servers?
- Stick queues to a group of servers?
- Make sure to put data replicas in at least two datacenters

Why Kafka

- Overheads associated with JMS and its various implementations
- Limitations with the scaling architecture
- Challenges of consuming the real-time and batch multiple sets data volumes from production system
- Kafka supports unify offline and online processing
- Kafka is able to partition real-time consumption over a cluster of machines
- Kafka provides a mechanism to parallel load in Hadoop systems
- Kafka offers better throughput, built-in partitioning, replication, and fault-tolerance

Why Kafka

- As with publish-subscribe, Kafka allows you to broadcast messages to multiple consumer groups
- The advantage of Kafka's model is that every topic has both these properties—it can scale processing and is also multi-subscriber
- Kafka has stronger ordering guarantees than a traditional messaging system
- Kafka as a kind of special purpose distributed filesystem dedicated to high-performance, low-latency commit log storage, replication, and propagation
- Kafka is the combination of messaging, storage, and stream processing

Features of Kafka

- Persistent messaging
- High throughput
- Distributed
- Partitions
- Replications
- Multiple client support
- Low Latency
- Scalable
- Fault-tolerant

Features of Kafka – Persistent messaging

- Disk based storage
- Messages are persisted on disk as well as replicated within the cluster to prevent data loss
- The Kafka cluster retains all published records—whether or not they have been consumed—using a configurable retention period

Features of Kafka – High throughput

- Work on commodity hardware
- Handle hundreds of MBs of reads and writes per second from large number of clients
- $O(1)$ disk structures that provide constant-time performance even with very large volumes of stored messages that are in the order of TBs

Features of Kafka – Distributed

- Message partitioning over Kafka servers
- Distributing consumption over a cluster of consumer machines while maintaining per-partition ordering semantics
- Each partition is replicated across a configurable number of servers for fault tolerance
- Each partition has one server which acts as the "leader" and zero or more servers which act as "followers"

Features of Kafka – Partitions

- Each partition is an ordered, immutable sequence of records that is continually appended to—a structured commit log. The records in the partitions are each assigned a sequential id number called the offset that uniquely identifies each record within the partition.
- Topic partitioning
- Support large volume of messages and parallel consumption
- Order between messages is kept between messages in the same partition
- High availability is achieved by replicating partition to other brokers so in case of a failure in one node

Features of Kafka – Partitions

- The only metadata retained on a per-consumer basis is the offset or position of that consumer in the log
- The partitions in the log allow the log to scale beyond a size that will fit on a single server
- Each individual partition must fit on the servers that host it, but a topic may have many partitions so it can handle an arbitrary amount of data

Features of Kafka – Multiple clients

- Java
- Scala
- .NET
- PHP
- Ruby
- Python

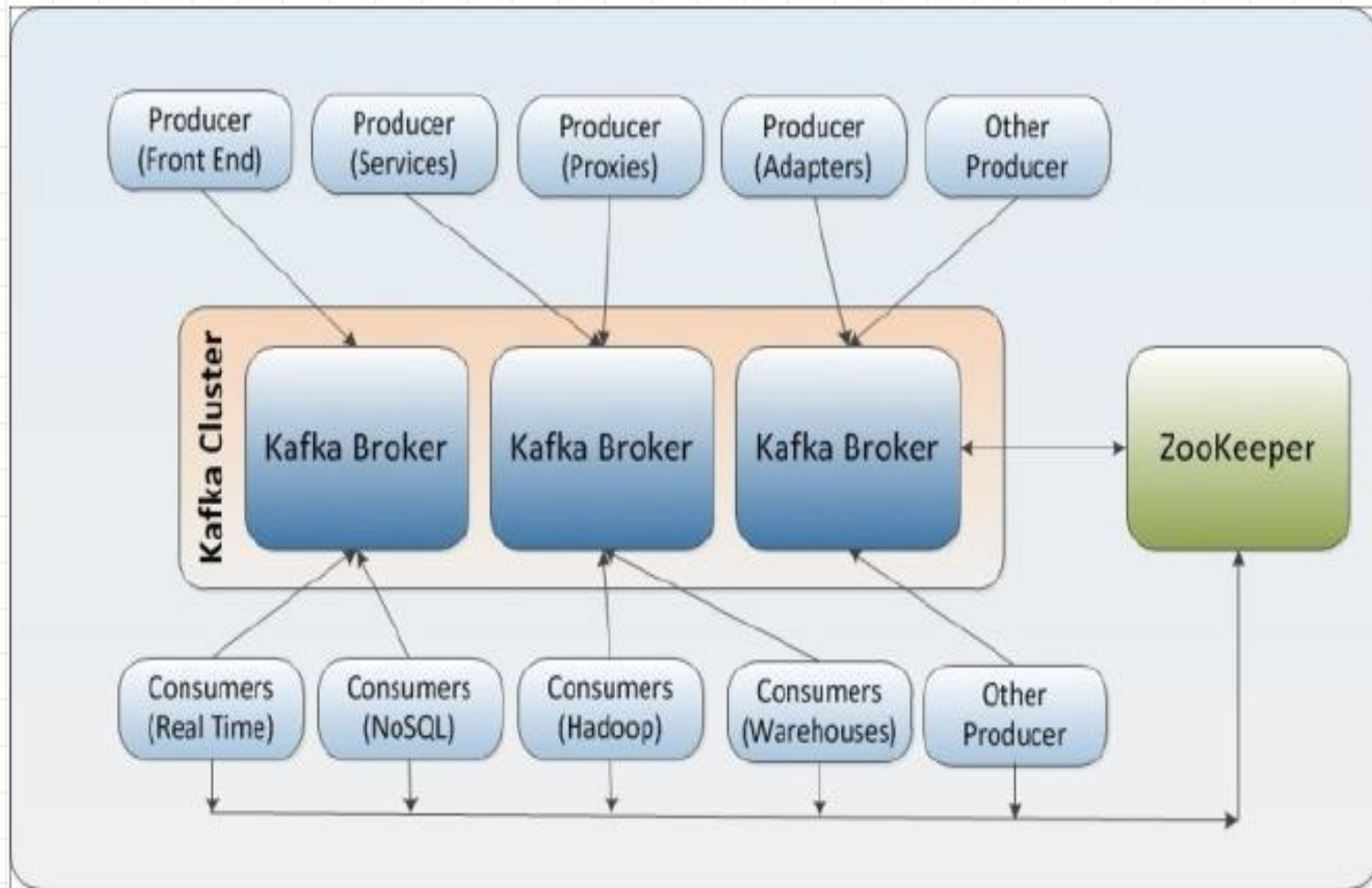
Features of Kafka – Low Latency

- Messages produced by the producer threads should be immediately visible to consumer threads

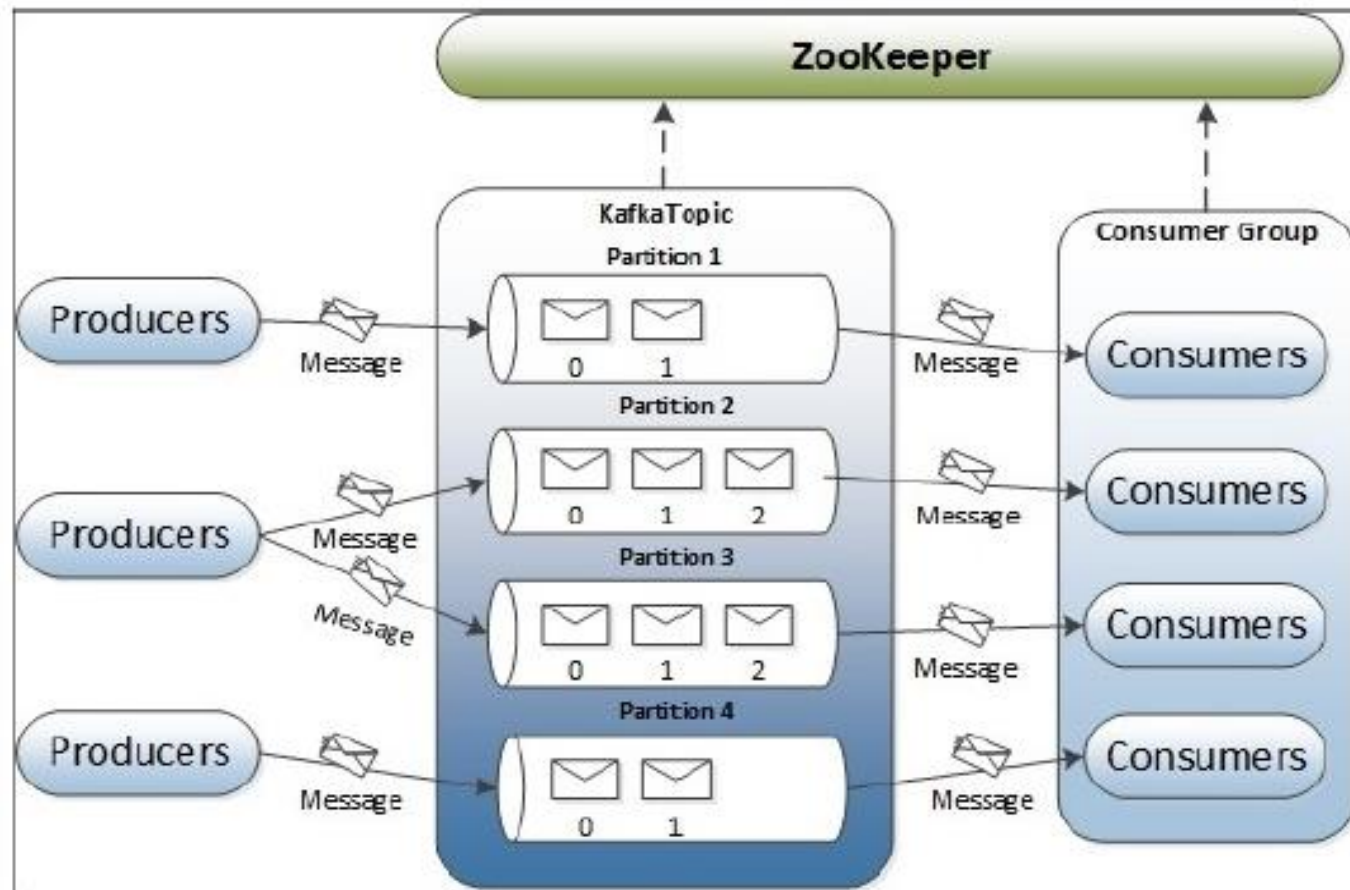
Features of Kafka – Scalable

- Kafka cluster can grow elastically and transparently without any downtime
- Should Consume the real-time and batch data volumes that may grow in order of magnitude to be larger than the real data
- The partitions in the log allow the log to scale beyond a size that will fit on a single server.
- The disk structures Kafka uses scale well—Kafka will perform the same whether you have 50 KB or 50 TB of persistent data on the server

High Level Kafka Architecture



High Level Kafka Architecture



Kafka Concepts

- **Broker**
 - A server process
- **Cluster**
 - A set of brokers
- **Topic**
 - A queue (that has log partitions)
- **Offset**
 - A message identifier
- **Partition**
 - An ordered and immutable sequence of records continually appended to a structured commit log

Kafka Concepts

- **Producer**
 - Those who publish data to topics
- **Consumer**
 - Those who process the feed
- **ZooKeeper**
 - The coordinator
- **Retention period**
 - The time to keep messages available for consumption

Kafka Concepts

- **Three types of clusters:**
 - Single node: Single broker
 - Single node: Multiple Broker
 - Multiple node: Multiple Broker
- **Three ways to deliver messages:**
 - Never redelivered: The messages may be lost
 - May be redelivered: The messages are never lost
 - Delivered once: The message is delivered exactly once
- **Two types of log compaction:**
 - Coarse grained: By time
 - Finer grained: By message

Kafka Core Components

- **Broker** - A Kafka cluster consists of one or more servers where each one may have one or more server processes running and is called the broker
- **Topic** - a category or feed name to which messages are published by the message producers
- **Producer** - Publish data to the topics by choosing the appropriate partition within the topic
- **Consumer** - Applications or processes that subscribe to topics and process the feed of published messages
- **Zookeeper** - Serves as the coordination interface between the Kafka broker and consumers
- **Message** - Each message consists of a key, a value, and a timestamp

Real life Kafka Case Studies

- **Log aggregation** - Provides clean abstraction of event data as a stream of messages and remove any dependency over file details
- **Stream processing** - Collect data undergoes processing at multiple stages
- **Commit logs** - Replicated logs over Kafka cluster to recover the failed node
- **Click stream tracking** - Capture user click stream or social media stream in real time
- **Messaging** - Replacement for many traditional message brokers
- **Metrics** - Aggregate statistics from distributed applications to produce centralized feeds of operational data
- **Event Sourcing** - Provide backend for an application by logged as a time-ordered sequence of records
- **Persistent Message Storage**

Working of Broker

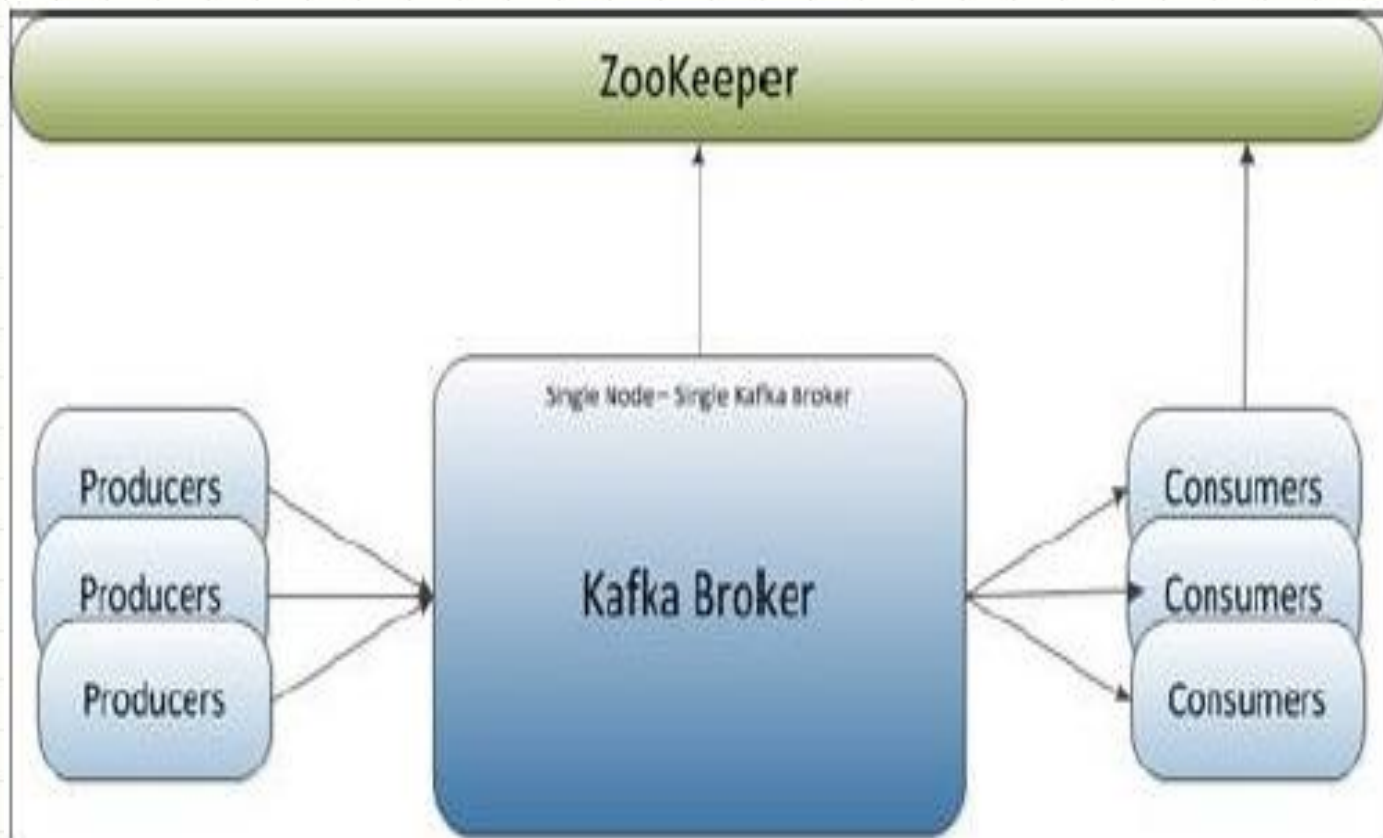
- A Kafka cluster is composed from nodes which are called brokers. A Kafka cluster is composed from nodes which are called brokers
- A typical broker will have several managed partitions and several replication for other partitions which are managed in different brokers
- Create a sequence number per message, append the message to a log file located in the node's local hard disk
- Manage an offset table with a sequence number as the key and value of the seek location of the message in the log file and its size
- brokers are stateless, which means the message state of any consumed message is maintained within the message consumer

Broker Configuration

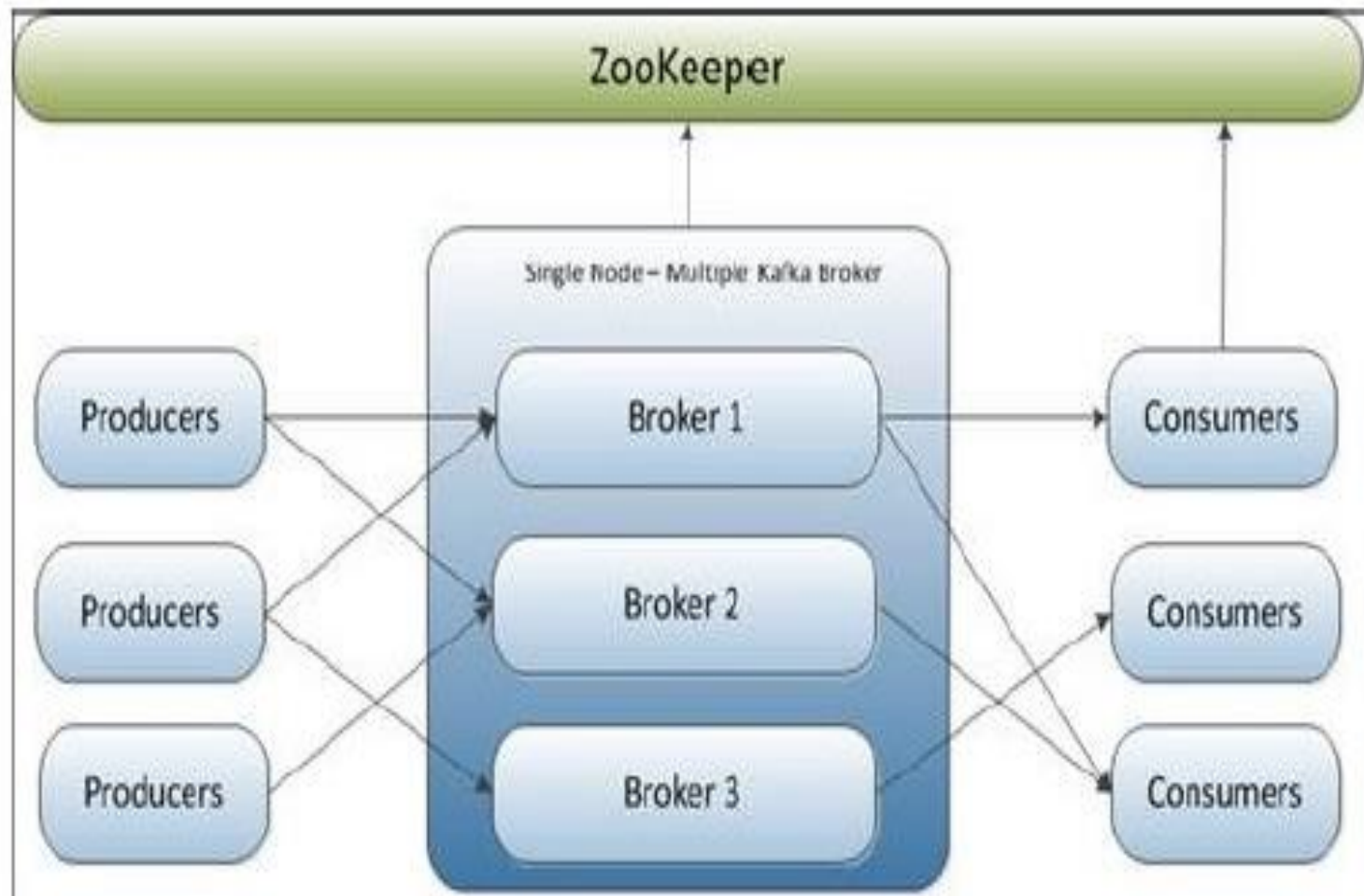
The essential configurations are the following:

- broker.id
- log.dirs
- zookeeper.connect
- auto.create.topics.enable
- auto.leader.rebalance.enable
- delete.topic.enable
- compression.type
- log.flush.interval.messages
- log.flush.interval.ms
- log.retention.bytes
- log.retention.hours
- log.roll.hours
- log.segment.bytes
- message.max.bytes
- num.partitions

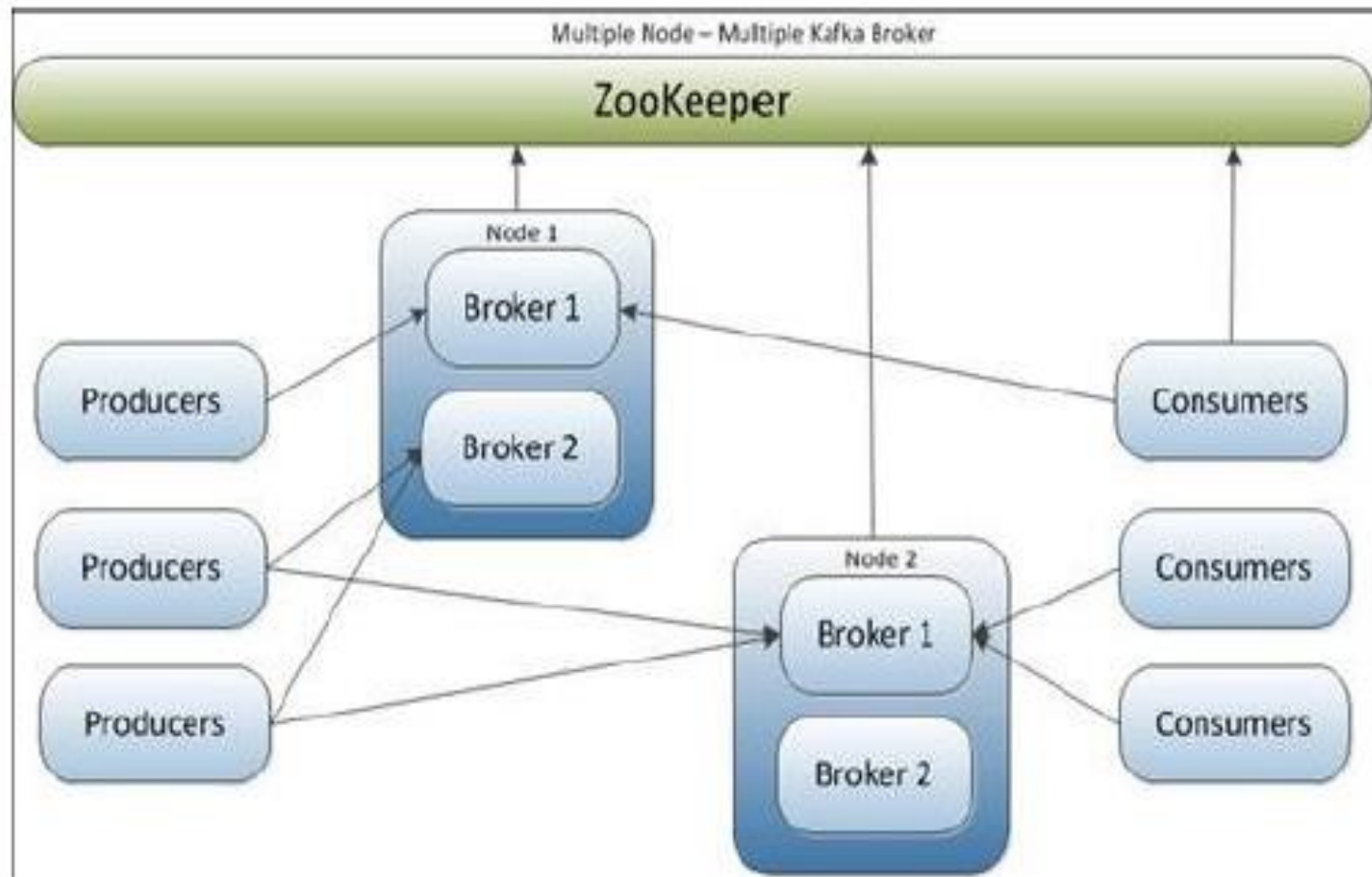
Single broker on single machine



Multiple brokers on single machine



Multiple brokers on multiple machines



Advantages of the cluster mode

- **Parallelism**
 - Run tasks simultaneously among the cluster members
- **Redundancy**
 - Warrants that when a Kafka node goes down, the cluster is safe and accessible from the other nodes

Basics of topic

- Several producers and consumers are interested in different type of message
- Kafka is using the abstraction of 'topic'. Producer is sending messages to a topic and consumers are reading messages from a topics
- The topic is actually implemented by continuous log file on node's local disk
- Topics are partitioned and each partition is represented by the ordered immutable sequence of messages
- A Kafka cluster maintains the partitioned log for each topic
- Each message in the partition is assigned a unique sequential ID called the offset
- Topics are created within the context of broker processes

Basics of topic

- A topic is a category or feed name to which records are published.
- Topics in Kafka are always multi-subscriber; that is, a topic can have zero, one, or many consumers that subscribe to the data written to it.

Basics of topic

- Retention Period
- Space Retention Policy
- Offset
- Partition
- Compaction
- Leader
- Buffering

Topic Configuration

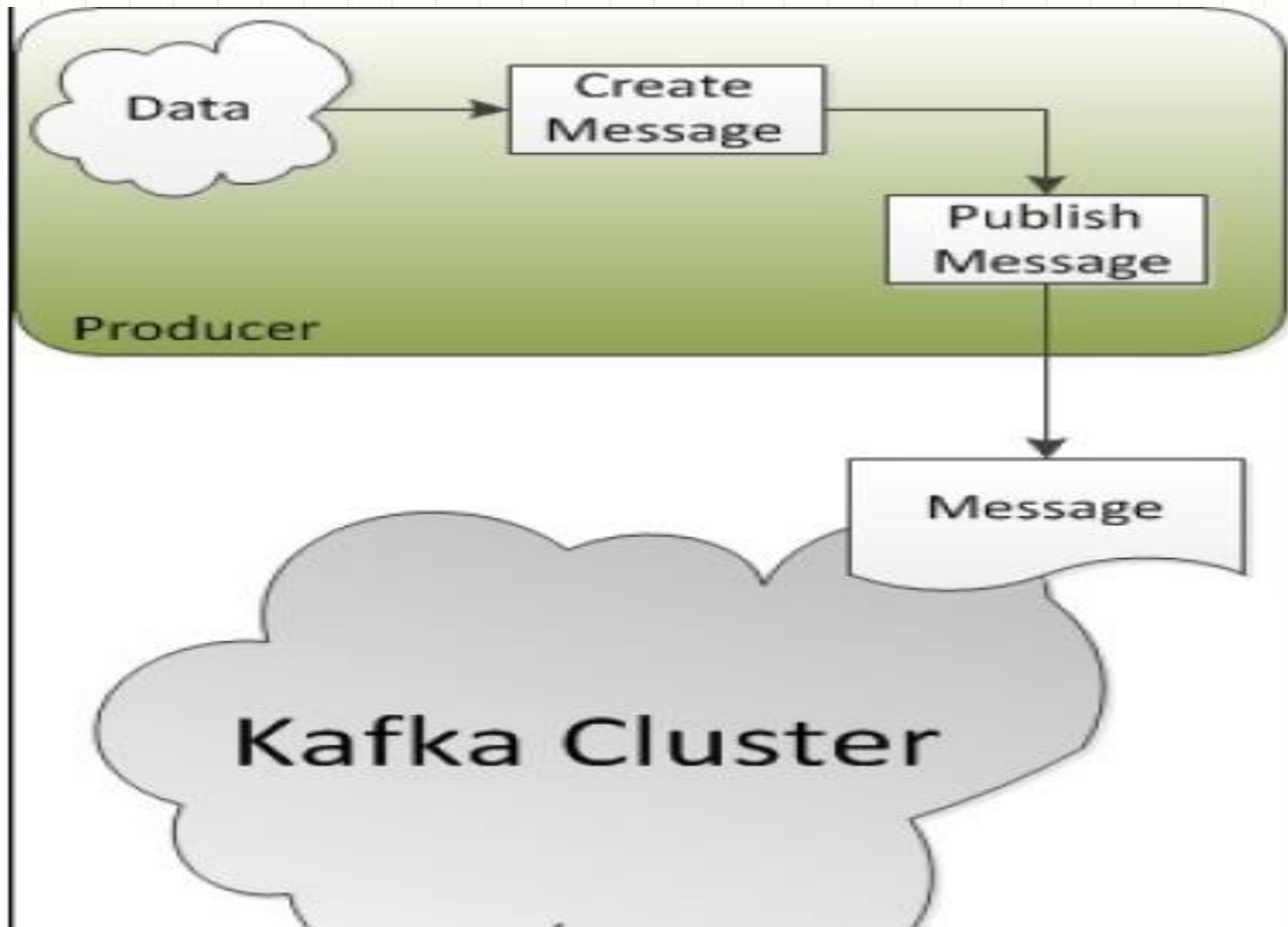
The essential configurations are the following:

- `cleanup.policy`
- `compression.type`
- `delete.retention.ms`
- `flush.messages`
- `flush.ms`
- `retention.bytes`
- `retention.ms`
- `segment.bytes`

Basics of producer

- Producers are applications that create messages and publish them to the Kafka broker for further consumption
- These producers can be different in nature; for example, frontend applications, backend services, proxy applications, adapters to legacy systems, and producers for Hadoop
- These producers can also be implemented in different languages such as Java, C, and Python
- Producers publish data to the topics of their choice
- The producer is responsible for choosing which record to assign to which partition within the topic, which can be done in a round-robin fashion

Producer Architecture



What does Producer do

- Bootstrapping Kafka broker URLs
- Data serialization
- Determining topic partition
- Determining the leader of the partition
- Failure handling/retry ability
- Batching

Producer Partition

- The producer connects to any of the alive nodes and requests metadata about the leaders for the partitions of a topic. This allows the producer to put the message directly to the lead broker for the partition
- The Kafka producer API exposes the interface for semantic partitioning by allowing the producer to specify a key to partition by and using this to hash to a partition
- The producer can completely control which partition it publishes messages to
- This also allows data consumers to make locality assumptions about customer data

Custom Producer Partition

- Kafka generally selects a partition based on the hash value of the key specified in messages. If the key is not specified/null, it will distribute the message in a round-robin fashion
- some scenarios where a percentage of data for one key is very large, we may be required to allocate a separate
- partition for that key

Common Publishing Patterns

- Fire-and-forget
- One message transfers
- Batching

Type of Producer – sync and async

- For high efficiency in Kafka, producers can also publish the messages in batches that work in asynchronous mode only
- In asynchronous mode, the producer works either with a fixed number of messages or fixed latency defined by producer configuration, `queue.time` or `batch.size`, respectively
- Data is accumulated in memory at the producer's end and published in batches in a single request
- Asynchronous mode also brings the risk of losing the data in the case of a producer crash with accumulated non-published, in-memory data

Producer Configuration

The essential configurations are the following:

- bootstrap.servers
- key.serializer
- value.serializer
- acks (0,1,all)
- buffer.memory
- compression.type
- retries
- batch.size
- client.id
- connections.max.idle.ms
- linger.ms
- max.block.ms
- partitioner.class
- timeout.ms
- block.on.buffer.full
- interceptor.classes
- metadata.fetch.timeout.ms

Sync Producer

- `acks=all`
 - The leader will wait for the full set of in-sync replicas to acknowledge the record
 - This guarantees that the record will not be lost as long as at least one in-sync replica remains alive
 - This is the strongest available guarantee
 - This is equivalent to the `acks=-1` setting

Async Producer

- `acks=0`
 - Set to zero then the producer will not wait for any acknowledgment from the server at all
 - The record will be immediately added to the socket buffer and considered sent
 - No guarantee can be made that the server has received the record in this case
 - The retries configuration will not take effect
 - The offset given back for each record will always be set to -1

Async Producer

- `acks=1`
 - The leader will write the record to its local log but will respond without awaiting full acknowledgement from all followers
 - In this case should the leader fail immediately after acknowledging the record but before the followers have replicated it then the record will be lost

Producer Best Practices

- Data validation
- Exception handling
- Number of retries
- Number of bootstrap URLs
- Avoid poor partitioning mechanism
- Temporary persistence of messages
- Avoid adding new partitions to existing topics

Basics of Consumer

- Consumers are the applications that consume the messages published by Kafka producers and process the data extracted from them
- Like producers, consumers can also be different in nature, such as applications doing real-time or near real-time analysis, applications with NoSQL or data warehousing solutions, backend services, consumers for Hadoop, or other subscriber-based solutions
- These consumers can also be implemented in different languages such as Java, C, and Python
- Consumers always consume messages from a particular partition sequentially and also acknowledge the message offset

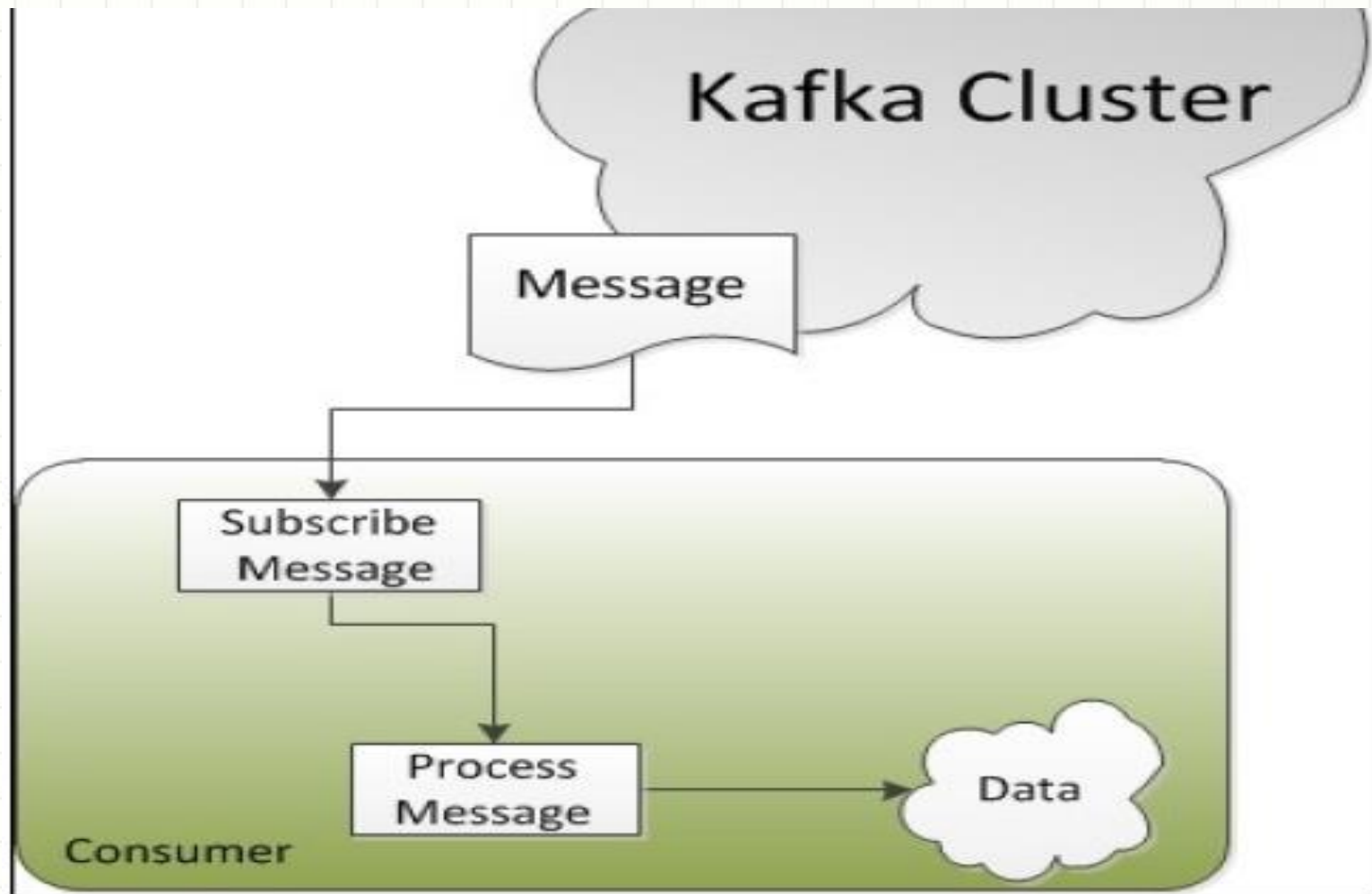
Basics of Consumer

- The consumer subscribes to the message consumption from a specific topic on the Kafka broker
- The consumer then issues a fetch request to the lead broker to consume the message partition by specifying the message offset
- The Kafka consumer works in the pull model and always pulls all available messages after its current position in the Kafka log (the Kafka internal data representation)
- While subscribing, the consumer connects to any of the live nodes and requests metadata about the leaders for the partitions of a topic
- The only metadata retained on a per-consumer basis is the offset or position of that consumer in the log

Basics of Consumer

- This allows the consumer to communicate directly with the lead broker receiving the messages
- Kafka topics are divided into a set of ordered partitions and each partition is consumed by one consumer only
- Once a partition is consumed, the consumer changes the message offset to the next partition to be consumed
- This represents the states about what has been consumed and also provides the flexibility of deliberately rewinding back to an old offset and re-consuming the partition
- The consumer can come and go without much impact on the cluster or on other consumers

Consumer Architecture



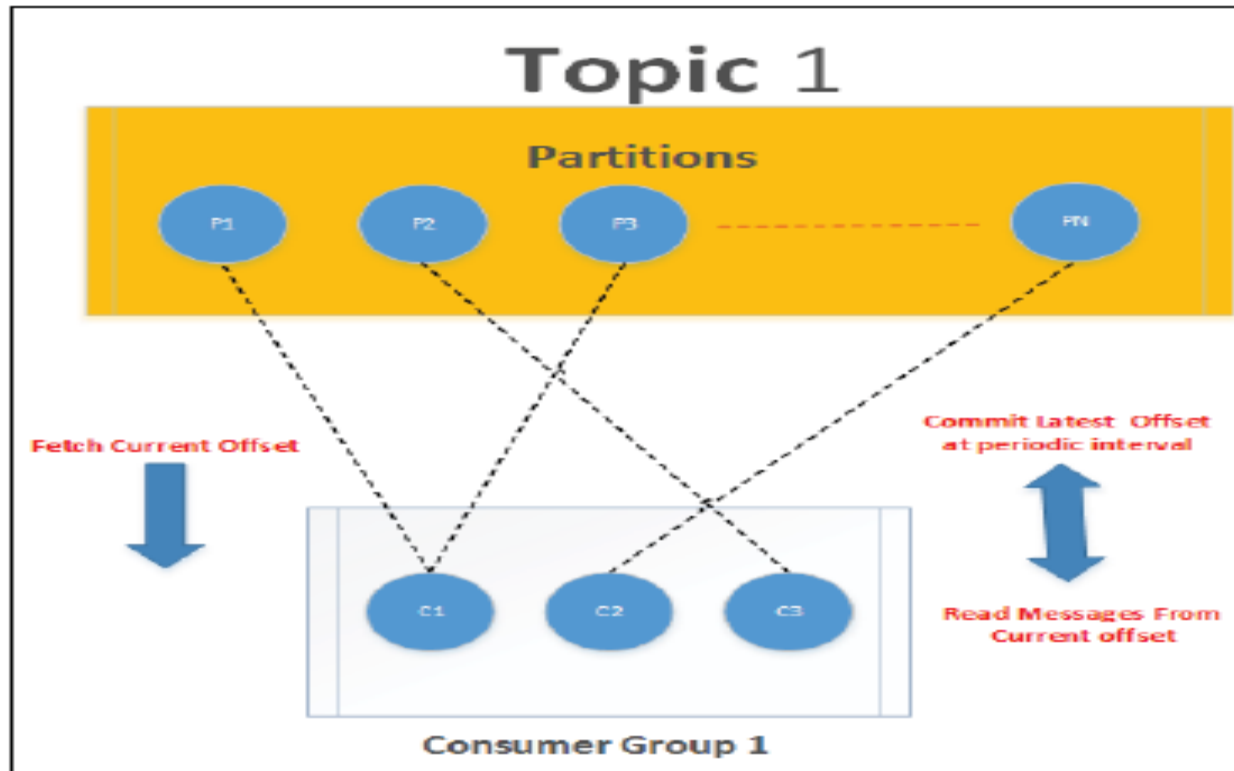
Responsibility of Consumer

- Subscribing to a topic
- Consumer offset position
- Replay/rewind/skip messages
- Heartbeats
- Offset commits
- Deserialization

Consumer Group

- Each consumer is represented as a process and these processes are organized within groups called consumer groups
- Allow consumers to share the consumption of a topic so each message is return to only one consumer
- A topic can be consumed by multiple consumer groups
- A set of consumers would like to consume the same messages of a topic then they all need to belong to different consumer-group
- If all the consumer instances have the same consumer group, then the records will effectively be load balanced over the consumer instances
- If all the consumer instances have different consumer groups, then each record will be broadcast to all the consumer processes

Consumer Group



Consumer Rebalancing

- The consumer rebalancing algorithms allows all the consumers in a group to come into consensus on which consumer is consuming which partitions
- Consumer rebalancing is triggered on each addition or removal of both broker nodes and other consumers within the same group
- For a given topic and a given consumer group, broker partitions are divided evenly among consumers within the group

Consumer Configuration

The essential configurations are the following:

- bootstrap.servers
- key.deserializer
- value.deserializer
- group.id
- heartbeat.interval.ms
- auto.offset.reset
- enable.auto.commit
- exclude.internal.topics
- partition.assignment.strategy
- security.protocol
- auto.commit.interval.ms

Common Consuming Patterns

- Consumer group - continuous data processing
- Consumer group - discrete data processing

Consumer Best Practices

- Exception handling
- Handling rebalances
- Commit offsets at the right time
- Automatic offset commits

Using Partitions

- Every partition is mapped to a logical log file that is represented as a set of segment files of equal sizes
- Every partition is an ordered, immutable sequence of messages
- A message is published to a partition, the broker appends the message to the last segment file
- The segment files are flushed to disk after configurable numbers of messages have been published or after a certain amount of time has elapsed
- Once the segment file is flushed, messages are made available to the consumers for consumption
- All the message partitions are assigned a unique sequential number called the offset

Distribution of Partitions

- Each partition is optionally replicated across a configurable number of servers for fault tolerance
- Each partition available on either of the servers acts as the leader and has zero or more servers acting as followers
- Leader is responsible for handling all read and write requests for the partition
- Followers asynchronously replicate data from the leader

Partitions Reassignment

- When you add machines to your cluster you will want to migrate some existing data to these machines
- Kafka will add the new server as a follower of the partition it is migrating and allow it to fully replicate the existing data in that partition
- New server has fully replicated the contents of this partition and joined the in-sync replica one of the existing replicas will delete their partition's data
- The partition reassignment tool can be used to move partitions across brokers
- The partition reassignment tool does not have the capability to automatically study the data distribution in a Kafka cluster and move partitions around to attain an even load distribution

Partitions Reassignment

The partition reassignment tool can run in 3 mutually exclusive modes:

-generate: the tool generates a candidate reassignment to move all partitions of the specified topics to the new brokers

--execute: the tool kicks off the reassignment of partitions based on the user provided reassignment plan

--verify: the tool verifies the status of the reassignment for all partitions listed during the last --execute

Message Ordering

- Messages sent by a producer to a particular topic partition will be appended in the order they are sent
- A consumer instance sees records in the order they are stored in the log
- Kafka only provides a total order over records within a partition, not between different partitions in a topic
- Per-partition ordering combined with the ability to partition data by key is sufficient for most applications
- if you require a total order over records this can be achieved with a topic that has only one partition

Using Replication - HA

- For better durability of messages and high availability of Kafka clusters, replication guarantees that the message will be published and consumed even in the case of broker failure, which may be caused by any reason

ISR – In Sync Replication

- Kafka dynamically maintains a set of in-sync replicas (ISR) that are caught-up to the leader and always persist the latest ISR set to ZooKeeper
- If the leader fails, one of the followers (in-sync replicas) will automatically become the new leader
- Each server plays a dual role; it acts as a leader for some of its partitions and also a follower for other partitions

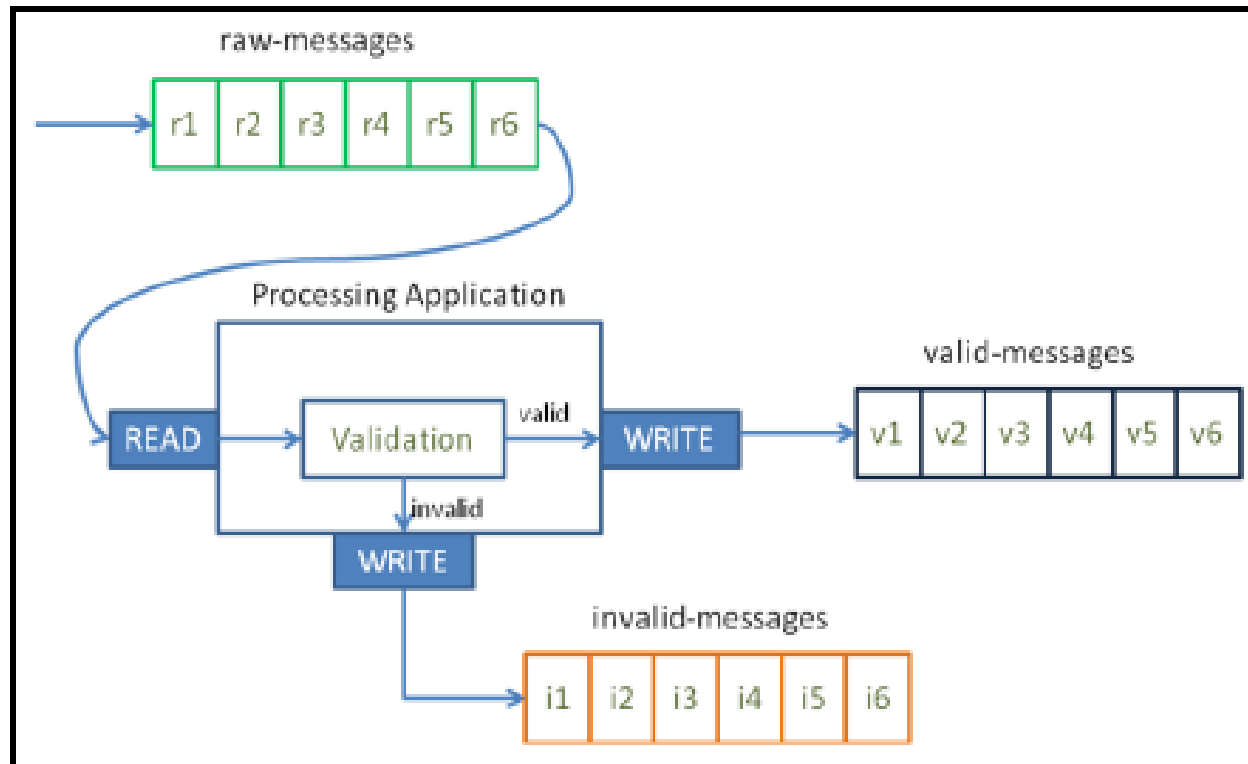
Kafka Design Fundamentals

- Provide an API for producers and consumers to support custom implementation
- Low overheads for network and storage with message persistence on disk
- A high throughput supporting millions of messages for both publishing and subscribing—for example, real-time log aggregation or data feeds
- Distributed and highly scalable architecture to handle low-latency delivery
- Auto-balancing multiple consumers in the case of failure
- Guaranteed fault-tolerance in the case of server failures

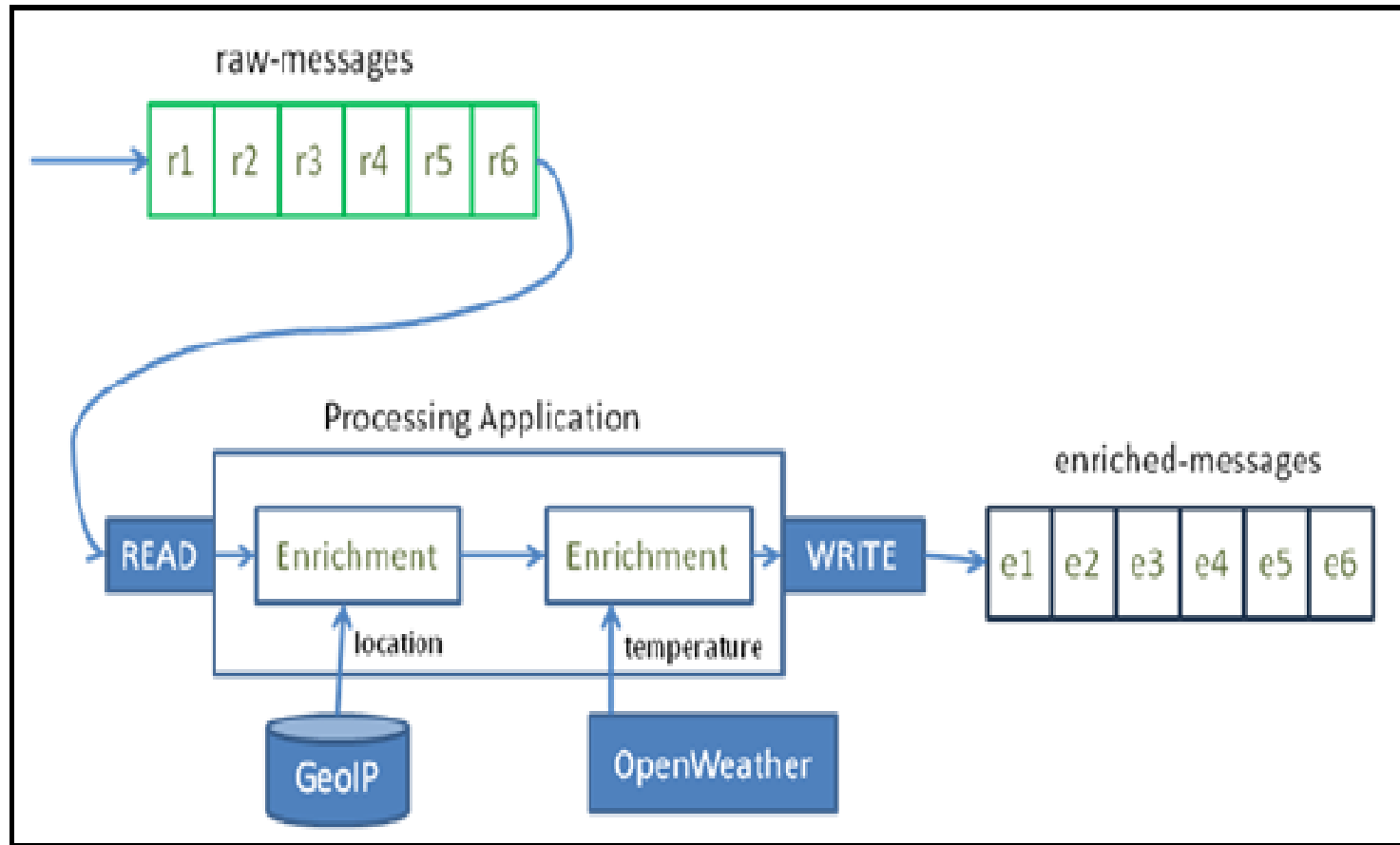
Message Processing

- The operation of processing messages involves the following:
 - An message stream to filter some messages from the stream
 - Message validation against an message schema
 - Message enrichment with additional data
 - Message composition (aggregation) to produce a new message from two or more messages

Message Validation



Message Enrichment



Message Compression

- Kafka provides a message group compression feature for efficient message delivery
- Efficient compression requires compressing multiple messages together rather than compressing each message individually
- A batch of messages is compressed together and sent to the broker
- In Kafka, data is compressed by the message producer using either the GZIP or Snappy compression protocols
- Message compression techniques are very useful for mirroring data across datacenters using Kafka

Message Acknowledge

- Consumers always consume messages from a particular partition sequentially and also acknowledge the message offset
- The acknowledgement implies that the consumer has consumed all prior messages
- Consumers issue an asynchronous pull request containing the offset of the message to be consumed to the broker and get the buffer of bytes

Message Retention

- Kafka defines the time-based SLA (service level agreement) as a message retention policy
- In line with this policy, a message will be automatically deleted if it has been retained in the broker longer than the defined SLA period
- This message retention policy empowers consumers to deliberately rewind to an old offset and re-consume data although
- Time based, size-based, or log compaction-based

Persistence

- A properly designed disk structure can often be as fast as the network
- The performance of linear writes but the performance of random writes is slow
- The linear reads and writes are the most predictable of all usage patterns, and are heavily optimized by the operating system
- A modern OS will happily divert all free memory to disk caching with little performance penalty when the memory is reclaimed. All disk reads and writes will go through this unified cache
- Using the filesystem and relying on pagecache is superior to maintaining an in-memory cache or other structure

Persistence

- Pagecache-centric design and constant time suffices
- All data is immediately written to a persistent log on the filesystem without necessarily flushing to disk. In effect this just means that it is transferred into the kernel's pagecache.
- The persistent data structure used in messaging systems are often a per-consumer queue with an associated BTree, Btree operations are $O(\log N)$
- Each disk can do only one seek at a time so parallelism is limited. Hence even a handful of disk seeks leads to very high overhead

Efficiency

- Two common causes of inefficiency in this type of system: too many small I/O operations, and excessive byte copying
- The small I/O problem happens both between the client and the server and in the server's own persistent operations
- Efficient compression requires compressing multiple messages together rather than compressing each message individually
- Kafka supports this by allowing recursive message sets. A batch of messages can be clumped together compressed and sent to the server in this form
- Kafka manages efficient reads and writes to local disk
- Kafka interaction with the local disk - appends new writes to an open file

Efficiency

- Kafka interaction with the local disk - reads continues area from disk
- Kafka rely on OS disk cache for optimizing its reads from and writes to disk
- Kafka uses the optimization strategy (Linux's sendfile API) to reduce unnecessary copy of data (e.g. copy of data from network buffers from the OS into the application buffers)
- Standardized binary message format that is shared by the producer, the broker, and the consumer
- Data is copied into pagecache exactly once and reused on each consumption instead of being stored in memory and copied out to kernel space every time it is read

OS level cache

- The OS cache is highly efficient, always enabled and it's optimized for appends calls and for multi reads from a continues area on disk.
- The OS level cache let Kafka's processes uses little JVM's heap space so avoiding all GC related issues
- No need to run any cache-warm-up if the broker restart as the OS cache is not cleared when process terminates

Linux's sendfile API

Linux's sendfile API read data directly from disk (hopefully from OS disk cache) and routing it directly to a network socket without passing through the application and so avoid the unnecessary copy of buffers

The Producer – Load Balancing

- All Kafka nodes can answer a request for metadata about which servers are alive and where the leaders for the partitions of a topic
- The client controls which partition it publishes messages to
- Random load balancing, or some semantic partitioning
- Semantic partitioning by allowing the user to specify a key to partition by and using this to hash to a partition

The Producer – Asynchronous send

- Enable batching the Kafka producer will attempt to accumulate data in memory and to send out larger batches in a single request
- The batching can be configured to accumulate no more than a fixed number of messages and to wait no longer than some fixed latency bound
- This buffering is configurable and gives a mechanism to trade off a small amount of additional latency for better throughput

The Consumer

- The state of the consumed messages is maintained at the consumer level. This also addresses issues such as:
 - Losing messages due to failure
 - Multiple deliveries of the same message
- By default, consumers store the state in Zookeeper but Kafka also allows storing it in offset topic or within other storage systems used for Online Transaction Processing (OLTP) applications as well

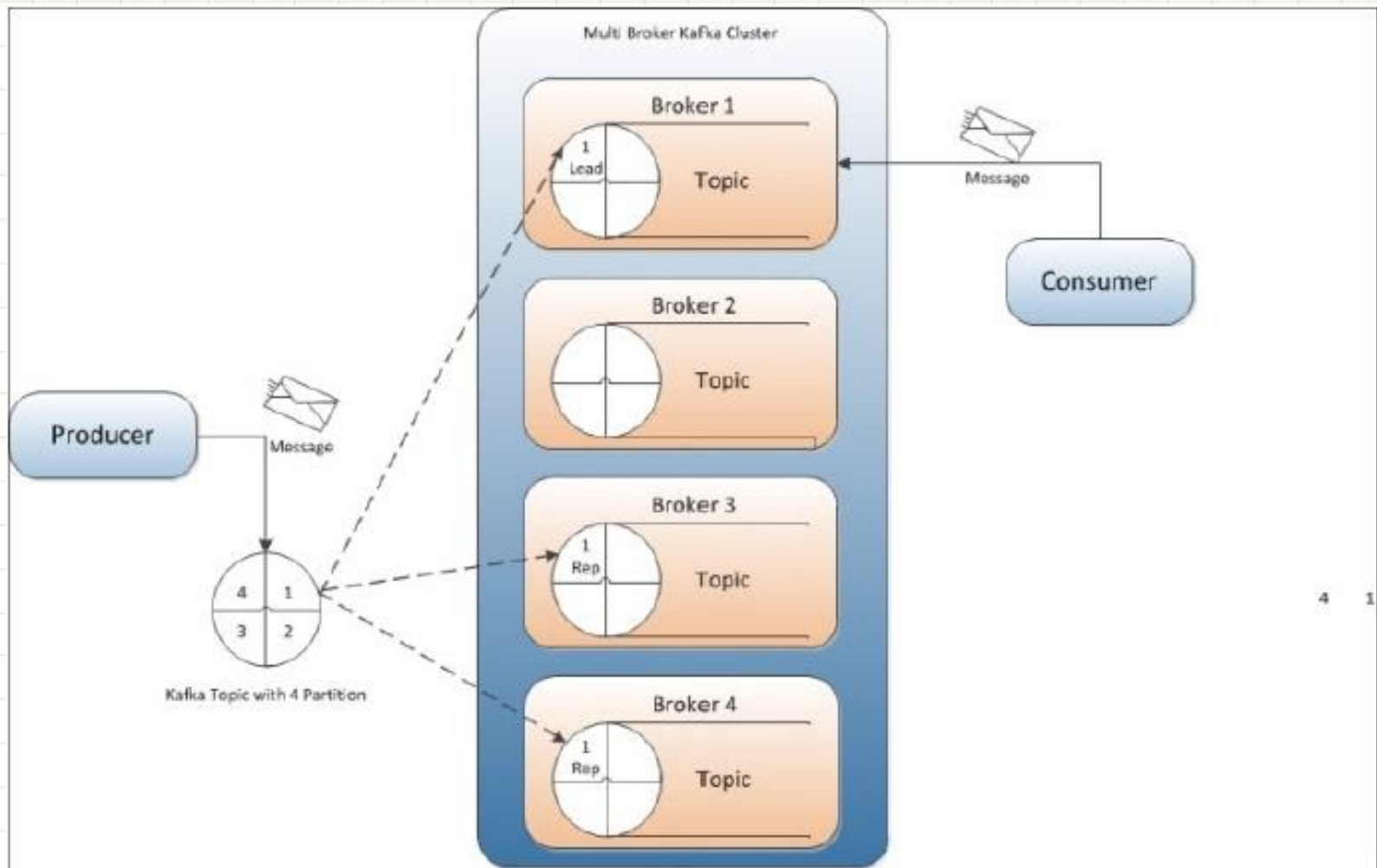
Message Delivery Semantics

- There are multiple possible ways to deliver messages:
 - Messages are never redelivered but may be lost
 - Messages may be redelivered but never lost
 - Messages are delivered once and only once
- For guaranteed message publishing, configurations such as getting acknowledgements and the waiting time for messages being committed are provided at the producer's end
- For consumers, Kafka guarantees that the message will be delivered at least once by reading the messages, processing the messages, and finally saving their position

Replication

- In Kafka, a message partitioning strategy is used at the Kafka broker end.
- The decision about how the message is partitioned is taken by the producer
- The broker stores the messages in the same order as they arrive
- The number of partitions can be configured for each topic within the Kafka broker
- Replication guarantees that the message will be published and consumed even in the case of broker failure for better durability of messages and high availability of Kafka clusters
- Both producers and consumers are replication-aware in Kafka

Replication



Replication

- In replication, each partition of a message has n replicas and can afford $n-1$ failures to guarantee message delivery
- Out of the n replicas, one replica acts as the lead replica for the rest of the replicas
- Zookeeper keeps the information about the lead replica and the current follower in-sync replicas (ISR)
- The lead replica maintains the list of all in-sync follower replicas
- Each replica stores its part of the message in local logs and offsets, and is periodically synced to the disk
- The process also ensures that either a message is written to all the replicas or to none of them

Synchronous Replication

- A producer first identifies the lead replica from ZooKeeper and publishes the message
- As soon as the message is published, it is written to the log of the lead replica and all the followers of the lead start pulling the message
- By using a single channel, the order of messages is ensured.
- Each follower replica sends an acknowledgement to the lead replica once the message is written to its respective logs
- Once replications are complete and all expected acknowledgements are received, the lead replica sends an acknowledgement to the producer
- On the consumer's side, all the pulling of messages is done from the lead replica

Asynchronous Replication

- As soon as a lead replica writes the message to its local log, it sends the acknowledgement to the message client and does not wait for acknowledgements from follower replicas
- This mode does not ensure message delivery in case of a broker failure

Replication Failure - Follower

- The leader drops the failed follower from its ISR list after the configured timeout period and writes will continue on the remaining replicas in ISRs
- Whenever the failed follower comes back, it first truncates its log to the last checkpoint (the offset of the last committed message) and then starts to catch up with all messages from the leader, starting from the checkpoint
- As soon as the follower becomes fully synced with the leader, the leader adds it back to the current ISR list

Replication Failure - Leader

- Either while writing the message partition to its local log or before sending the acknowledgement to the message producer, a message partition is resent by the producer to the new lead broker
- The process of choosing the new lead replica involves all the followers' ISRs registering themselves with Zookeeper. The very first registered replica becomes the new lead replica and its log end offset (LEO) becomes the offset of the last committed message (also known as **high watermark (HW)**)

Log Compaction

- Log compaction is a mechanism to achieve finer-grained, per-message retention, rather than coarser-grained, time-based retention
- The last known value for each message key within the log for a topic partition must be retained by removing the records where a more recent update with the same primary key is done
- Log compaction also addresses system failure cases or system restarts
- The retention policy can be set on a per-topic basis such as time based, size-based, or log compaction-based

Log Compaction

Log compaction ensures the following:

- Ordering of messages is always maintained
- The messages will have sequential offsets and the offset never changes
- Reads progressing from offset 0, or the consumer progressing from the start of the
- log, will see at least the final state of all records in the order they were written

Quotas

- The Kafka cluster has the ability to enforce quotas on produce and fetch requests
- Quotas are basically byte-rate thresholds defined per group of clients sharing a quota. Quotas are basically byte-rate thresholds defined per group of clients sharing a quota.
- Quotas can be applied to (user, client-id), user or client-id groups
- Quota configuration may be defined for (user, client-id), user and client-id groups.
- User and (user, client-id) quota overrides are written to ZooKeeper under /config/users and client-id quota overrides are written under /config/clients
- Broker properties (quota.producer.default, quota.consumer.default) can also be used to set defaults for client-id groups

Important Kafka Design Facts

- The fundamental backbone of Kafka is message caching and storing on the filesystem. In Kafka, data is immediately written to the OS kernel page. Caching and flushing of data to the disk are configurable
- Kafka provides longer retention of messages even after consumption, allowing consumers to re-consume, if required
- Kafka uses a message set to group messages to allow lesser network overhead
- Unlike most messaging systems, where metadata of the consumed messages are kept at the server level, in Kafka the state of the consumed messages is maintained at the consumer level

Important Kafka Design Facts

- In Kafka, producers and consumers work on the traditional push-and-pull model, where producers push the message to a Kafka broker and consumers pull the message from the broker
- Kafka does not have any concept of a master and treats all the brokers as peers. This approach facilitates addition and removal of a Kafka broker at any point, as the metadata of brokers are maintained in Zookeeper and shared with consumers
- Producers also have an option to choose between asynchronous or synchronous mode to send messages to a broker

Migrating offsets from ZooKeeper to Kafka

- Set `offsets.storage=kafka` and `dual.commit.enabled=true` in the consumer config.
- Do a rolling bounce of the consumers and then verify that the consumers are healthy.
- Set `dual.commit.enabled=false` in the consumer config.
- Do a rolling bounce of the consumers and then verify that the consumers are healthy.

Operating Kafka - Creating Topic

- `kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic kafkatopic`
- `kafka-topics.sh --list --zookeeper localhost:2181 kafkatopic`
- `kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 3 --partitions 1 --topic replicated-kafkatopic`
- `kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-replicated-topic`

Operating Kafka - Creating Topic

- **--zookeeper:** This specifies the ZooKeeper connect string
- **--create:** This keyword specifies that a topic needs to be created.
- **--delete:** This keyword specifies that the topic needs to be deleted. The server configuration has to be `delete.topic.enable=true`
- **--topic:** This is used to specify the topic name.
- **--partitions:** This is used to specify the number of partitions to be created for the topic
- **--replication-factor:** This specifies the number of replicas to be created for the topic. This number must be less than the number of nodes in the cluster
- **--config x=y,** where x is config name and y is the value for the configuration. These are used to override the default property set on the server

Operating Kafka - Creating Topic

Details of the various configurations are as follows:

- **cleanup.policy:** This keyword can take either of two values: delete or compaction. The default value is delete, which will delete the logs once the logs reach the time or size limits.
- **delete.retention.ms:** This is used to change the length of time you want the logs to be retained in Kafka.

Operating Kafka - Modifying Topic

- `kafka-topics.sh --zookeeper zk_host:port/chroot --alter --topic my_topic_name --partitions 40`
- `kafka-topics.sh --zookeeper zk_host:port/chroot --alter --topic my_topic_name --config x=y`
- `kafka-topics.sh --zookeeper zk_host:port/chroot --alter --topic my_topic_name --delete-config x`
- `kafka-topics.sh --zookeeper localhost:2181 --delete --topic testtopic`

Listing all topic names

```
kafka-topics.sh --list --zookeeper localhost:2181
```

Producing the message

```
kafka-console-producer.sh --broker-list localhost:6667 --topic kafkatopic
```

```
kafka-console-producer.sh --broker-list localhost:9092, localhost:9093 --  
topic replicated-kafkatopic
```

Producing the message

- **--broker-list:** This specifies the ZooKeeper servers
- **--topic:** This specifies the name of the topic
- **--sync:** This specifies that the messages should be sent synchronously—one at a time as they survive
- **--compression-codec:** This specifies the compression codec that will be used to produce the messages
- **--batch-size:** This specifies the number of messages to be sent in a single batch
- **--message-send-max-retries:** This property specifies the number of retries before a producer gives up and drops the message
- **--retry-backoff-ms:** This parameter specifies some time before producer retries

Consuming the message

```
kafka-console-consumer.sh --zookeeper localhost:2181 --topic kafkatopic --from-beginning
```

```
kafka-console-consumer.sh --zookeeper localhost:2181 --from-beginning --topic replicated-kafkatopic
```

Consuming the message

- **--fetch-size:** This specifies the amount of data to be fetched in a single request
- **--socket-buffer-size:** This specifies the size of the TCP RECV size
- **--autocommit.interval.ms:** This specifies the time interval in which the current offset is saved in ms
- **--max-messages:** This specifies the maximum number of messages to consume before exiting
- **--skip-message-on-error:** This specifies that, if there is an error while processing a message, the system should not stop

Graceful shutdown

- It will sync all its logs to disk to avoid needing to do any log recovery when it restarts
- It will migrate any partitions the server is the leader for to other replicas prior to shutting down.
- Controlled leadership migration requires `controlled.shutdown.enable=true`

Balancing leadership

- Kafka cluster restores leadership to the restored replicas by running the command:
 - `kafka-preferred-replica-election.sh --zookeeper zk_host:port/chroot`
- Configure Kafka to do this automatically by setting the following configuration:
 - `auto.leader.rebalance.enable=true`

Checking consumer position

- Consumer offset checker is one of the most important tools used to debug consumers, using this tool you can figure out the offsets till which your consumer have completed consuming the Kafka logs
- `kafka-run-class.sh kafka.tools.ConsumerOffsetChecker --brokerinfo --zookeeper localhost:2181 --group test-consumer-group`

Understanding dump log segments

- Sometimes, you may want to debug the Kafka logs data for various debugging purposes such as understanding how much of the logs have been written and what's the status of the various segments
- `kafka-run-class.sh kafka.tools.DumpLogSegments --deep-iteration --files /tmp/kafka-logs/my-replicated-topic-2/00000000000000000000.log`
- **--deep-iteration:** uses deep instead of shallow iteration to examine the log files
- **--files:** This is the only mandatory field. It is a comma-separated list of data and index log files that need to be dumped
- **--max-message-size:** This is used to set the size of the largest message. The default value of this is 5242880
- **--print-data-log:** This parameter must be set if you want the messages' content while dumping the data logs
- **--verify-index-only:** This parameter must be set if you want to verify the index log without printing its content

Exporting the ZooKeeper offsets

- Take a backup of the offsets saved in ZooKeeper
- `kafka-run-class.sh kafka.tools.ExportZkOffsets --zkconnect localhost:2181 --group test-consumer-group --output-file /tmp/out.txt`

Importing the ZooKeeper offsets

- Restore the offsets in ZooKeeper
- `kafka-run-class.sh kafka.tools.ImportZkOffsets --inputfile /tmp/zkoffset.txt --zkconnect localhost:2181`

Using GetOffsetShell

- To get the offset values of the various topics is needed while debugging Kafka based Big Data
- `kafka-run-class.sh kafka.tools.GetOffsetShell --broker-list localhost:9092 -
-topic mytesttopic --time -1`
- **--broker-list**: This specifies the list of server ports to connect to
- **--max-wait-ms**: This specifies the maximum amount of time each of the fetch requests will wait
- **--offsets**: This specifies the number of offsets that are returned
- **--partitions**: This is a comma-separated list of partition IDs
- **--time**: This specifies the timestamp to fetch for the offsets
- **--topic**: This specifies the topic for which the offset needs to be fetched

Using the JMX tool

- The tool gets the JMX report for Kafka in an easy way
- `kafka-run-class.sh kafka.tools.JmxTool --jmx-url service:jmx:rmi:///jndi/rmi://127.0.0.1:9999/jmxrmi`
- **--attributes:** This is a comma-separated list of objects that acts as a whitelist of attributes to be queried
- **--date-format:** This specifies the data format to be used for the time field
- **--jmx-url:** This specifies the URL to connect to the poll JMX data
- **--object-name:** This specifies the JMX object name to be used as a query
- **--reporting-interval:** This specifies the interval in milliseconds with the poll JMX stats

Mirroring data between clusters

- The MirrorMaker tool comes in handy to replicate the same data in a different Kafka cluster
- `kafka-run-class.sh kafka.tools.MirrorMaker --consumer.config config/consumer.config --producer.config config/producer.config --whitelist mytesttopic`
- **--blacklist**: This specifies the blacklist of topics to be mirrored
- **--consumer.config**: This specifies the path to the consumer configuration file to consume from a source cluster
- **--num.producers**: This specifies the number of producer instances
- **--num.streams**: This specifies the number of consumption streams' threads
- **--producer.config**: This specifies the path to the embedded producer configuration file
- **--queue.size**: This specifies the queue size in the number of messages that are buffered terms between the consumer and producer
- **--whitelist**: This specifies the whitelist of topics to be mirrored

Replay Log Producer

- If you want to move data from one topic to another, Replay Log Producer is the ideal tool for you
- `kafka-run-class.sh kafka.tools.ReplayLogProducer --sync --brokerlist localhost:9092 --inputtopic mytesttopic --outputtopic mytesttopic2 --zookeeper localhost:2181`
- **--sync**: If this is specified, the messages are sent synchronously; else they are sent asynchronously
- **--broker-list**: This specifies the broker list. This is a mandatory field
- **--inputtopic**: This specifies the topic to consume from
- **--messages**: This specifies the number of messages to be sent
- **--outputtopic**: This specifies the topic to produce to
- **--reporting-interval**: This specifies the interval in milliseconds to print the progress information
- **--threads**: This specifies the number of sending threads
- **--zookeeper**: This specifies the connection string for the zookeeper connection in the host:port form

State Change Log Merger

- This is a utility that merges the state change logs from different brokers for easy analysis later on
- `kafka-run-class.sh kafka.tools.StateChangeLogMerger --log-regex /tmp/state-change.log* --partitions 0,1,2 --topic statelog`
- **--end-time**: This specifies the latest timestamp of state change entries to be merged in the `java.text.SimpleDateFormat` format
- **--logs**: This is used to specify a comma-separated list of state change logs or regex for the log file names
- **--logs-regex**: This is used to specify a regex to match the state change log files to be merged
- **--partitions**: This specifies a comma-separated list of partition IDs whose state change logs should be merged
- **--start-time**: This specifies the earliest timestamp of state change entries to be merged in the `java.text.SimpleDateFormat` format
- **--topic**: This specifies the topic whose state change logs should be merged

Updating offsets in ZooKeeper

- This tool is perfect when you want to reset the offset of a consumer in ZooKeeper
- `kafka-run-class.sh kafka.tools.UpdateOffsetsInZK earliest config/consumer.properties mytopic`

Verifying consumer rebalance

- After the rebalancing operation, each partition must have selected an owner. One way to verify that is by reading the data from ZooKeeper at `/consumers/[consumer_group]/owners/[topic]/[broker_id-partition_id]` and finding an entry for each of `/brokers/topics/[topic]/[broker-id]`. This tool will make your life easier to make sure there is an owner for every partition
- `kafka-run-class.sh kafka.tools.VerifyConsumerRebalance --zookeeper.connect localhost:2181 --group mytestcons`

Managing Consumer Groups

- `kafka-consumer-groups.sh --bootstrap-server broker1:9092 --list`
- `kafka-consumer-groups.sh --bootstrap-server broker1:9092 --describe --group test-consumer-group`

Expanding your cluster

- `kafka-reassign-partitions.sh --zookeeper localhost:2181 --topics-to-move-json-file topics-to-move.json --broker-list "5,6" --generate`
- `kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file expand-cluster-reassignment.json --execute`
- `kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file expand-cluster-reassignment.json --verify`

Decommissioning brokers

- The partition reassignment tool does not have the ability to automatically generate a reassignment plan for decommissioning brokers
- the admin has to come up with a reassignment plan to move the replica for all partitions hosted on the broker to be decommissioned, to the rest of the brokers

Increasing replication factor

- Increasing the replication factor of an existing partition is easy. Just specify the extra replicas in the custom reassignment json file and use it with the --execute option to increase the replication factor of the specified partitions.
- `kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file increase-replication-factor.json --execute`

Kafka Producer API

- The Producer API allows applications to send streams of data to topics in the Kafka cluster
- The producer is thread safe and sharing a single producer instance across threads will generally be faster than having multiple instances
- The producer consists of a pool of buffer space that holds records that haven't yet been transmitted to the server as well as a background I/O thread that is responsible for turning these records into requests and transmitting them to the cluster
- Failure to close the producer after use will leak these resources
- The producer maintains buffers of unsent records for each partition. These buffers are of a size specified by the `batch.size` config

Kafka Producer API - Example

```
Properties props = new Properties();  
props.put("bootstrap.servers", "localhost:9092");  
props.put("acks", "all");  
props.put("retries", 0);  
props.put("batch.size", 16384);  
props.put("linger.ms", 1);  
props.put("buffer.memory", 33554432);  
props.put("key.serializer",  
"org.apache.kafka.common.serialization.StringSerializer");  
props.put("value.serializer",  
"org.apache.kafka.common.serialization.StringSerializer");
```

```
Producer<String, String> producer = new KafkaProducer<>(props);  
for(int i = 0; i < 100; i++)  
    producer.send(new ProducerRecord<String, String>("my-topic",  
Integer.toString(i), Integer.toString(i)));
```

```
producer.close();
```

Kafka Consumer API

- The Consumer API allows applications to read streams of data from topics in the Kafka cluster
- The low-level "simple" API maintains a connection to a single broker and has a close correspondence to the network requests sent to the server
- The low-level API is completely stateless, with the offset being passed in on every request, allowing the user to maintain this metadata however they choose
- The high-level API hides the details of brokers from the consumer and allows consuming off the cluster of machines without concern for the underlying topology
- The high-level API maintains the state of what has been consumed
- The high-level API provides the ability to subscribe to topics that match a filter expression

Kafka Consumer API

- This client can communicate with brokers
- This client transparently handles the failure of Kafka brokers, and transparently adapts as topic partitions it fetches migrate within the cluster
- This client also interacts with the broker to allow groups of consumers to load balance consumption using consumer groups
- The consumer is not thread-safe

Kafka Consumer Low Level API

```
class SimpleConsumer {  
  
    public ByteBufferMessageSet fetch(FetchRequest request);  
  
    public MultiFetchResponse multifetch(List<FetchRequest> fetches);  
  
    public long[] getOffsetsBefore(String topic, int partition, long time, int  
maxNumOffsets);  
}  
  
}
```


Kafka Consumer High Level API

```
interface ConsumerConnector {  
  
    public Map<String,List<KafkaStream>>  
    createMessageStreams(Map<String,Int> topicCountMap);  
  
    public List<KafkaStream> createMessageStreamsByFilter(  
        TopicFilter topicFilter, int numStreams);  
  
    public commitOffsets()  
  
    public shutdown()  
}
```

Kafka Streaming API

The Streams API allows transforming streams of data from input topics to output topics

Kafka Connect API

The Connect API allows implementing connectors that continually pull from some source data system into Kafka or push from Kafka into some sink data system

Kafka Security - Overview

- Authentication of connections to brokers from clients, other brokers and tools, using either SSL or SASL (Kerberos) or SASL/PLAIN
- Authentication of connections from brokers to ZooKeeper
- Encryption of data transferred between brokers and clients, between brokers, or between brokers and tools using SSL
- Authorization of read / write operations by clients
- Authorization is pluggable and integration with external authorization services is supported

Kafka Security - Authentication

- Encryption and Authentication using SSL
- Authentication using SASL/Kerberos
- Authentication using SASL/PLAIN

Kafka Security - Authorization

- Kafka ships with a pluggable Authorizer and an out-of-box authorizer implementation that uses zookeeper to store all the acls
- Apache Ranger

Kafka OS

Two potentially important OS-level configurations:

- **File descriptor limits:** Kafka uses file descriptors for log segments and open connections. At least 100000 allowed file descriptors for the broker processes as a starting point.
- **Max socket buffer size:** can be increased to enable high-performance data transfer between data centers as described here.

Disks and Filesystems

- Using multiple drives to get good throughput and not sharing the same drives used for Kafka data with application logs
- Either RAID these drives together into a single volume or format and mount each drive as its own directory

Application vs OS Flush Management

- Using the default flush settings which disable application fsync entirely. This means relying on the background flush done by the OS and Kafka's own background flush
- The drawback of using application level flush settings is that it is less efficient and it can introduce latency as fsync in most Linux filesystems blocks writes to the file whereas the background flushing does much more granular page-level locking

Linux Flush Behavior

- In Linux, data written to the filesystem is maintained in pagecache until it must be written out to disk
- An application-level fsync or the OS's own flush policy
- The flushing of data is done by a set of background threads called pdflush
- Pdflush has a configurable policy that controls how much dirty data can be maintained in cache and for how long before it must be written back to disk

Monitoring Kafka

- Monitoring server statistics
- Monitoring producer statistics
- Monitoring consumer statistics
- Monitoring with the help of Graphite
- Monitoring with the help of Ganglia

Kafka Best Practices - Broker

- **Java Version**

- Recommend latest java 1.8 with G1 collector

- **OS Settings**

- Once the JVM size is determined leave rest of the RAM to OS for page caching, need sufficient memory for page cache to buffer for the active writers and readers
- disk throughput is a performance bottleneck and more disks are better
- File Descriptors limits: Kafka needs open file descriptors for files and network connections . recommend at least 128000 allowed for file descriptors.
- Max socket buffer size , can be increased to enable high-performance data transfer

Kafka Best Practices - Broker

- **Disks And File System**

- Recommend using multiple drives to get good throughput, o not share the same drives with any other application or for kafka application logs
- Recommend users to create alerts on disk usage for kafka drives to avoid any interruptions to running Kafka service
- RAID doesn't provide much real availability improvement

-

Kafka Best Practices - Broker

- **Log Flush Management**

- Kafka always write data to files immediately and allows users to configure `log.flush.interval.messages` to enforce flush for every configure number of messages
- Kafka flushes the log file to disk whenever a log file reaches `log.segment.bytes` or `log.roll.hours`
- Recommend using the default flush settings which disables the explicit `fsync` entirely, This means relying on background flush done by OS and Kafka's own background flush

Kafka Best Practices - Broker

- **FileSystem Selection**

- Recommend EXT4 or XFS
- Do not use mounted shared drives and any network file systems

- **Zookeeper**

- Do not co-locate zookeeper on the same boxes as Kafka
- Recommend zookeeper to isolate and only use for Kafka not any other systems should be depend on this zookeeper cluster
- Make sure you allocate sufficient JVM , good starting point is 4Gb
- Monitor: Use JMX metrics to monitor the zookeeper instance

Kafka Best Practices - Broker

- **Choosing Topic/Partitions**

- Higher the number of partitions more parallel consumers can be added , thus resulting in a higher throughput Do not use mounted shared drives and any network file systems
- Based on throughput requirements one can pick a rough number of partitions
 - ✓ Lets call the throughput from producer to a single partition is P
 - ✓ Throughput from a single partition to a consumer is C
 - ✓ Target throughput is T
 - ✓ Required partitions = $\text{Max} (T/P, T/C)$
- More partitions can increase the latency

Kafka Best Practices - Broker

- **Factors Affecting Performance**

- Main memory. More specifically File system buffer cache
- Multiple dedicated disks
- Partitions per topic. More partitions allows increased parallelism
- Ethernet bandwidth

- **Kafka Broker configs**

- Set kafka broker JVM by exporting KAFKA_HEAP_OPTS
- Log.retention.hours
- Message.max.bytes
- Delete.topic.enable
- unclean.leader.election

Kafka Best Practices - Producer

- **Critical Configs**

- Batch.size (size based batching)
- Linger.ms (time based batching)
- Compression.type
- Max.in.flight.requests.per.connection (affects ordering)
- Acks (affects durability)

Kafka Best Practices - Producer

- **Performance Notes**

- A producer thread going to the same partition is faster than a producer thread that sprays to multiple partitions
- If producer throughput maxes out and there is spare CPU and network capacity on box, add more producer processes
- Performance is sensitive to event size
- No simple rule of thumb for linger.ms. Needs to be tried out on specific use cases

Kafka Best Practices - Producer

- **Lifecycle of a request from Producer to Broker**
 - Polls batch from the batch queue , 1 batch per partition
 - Groups batches based on the leader broker
 - Sends the grouped batches to the brokers
 - Pipelining if `max.in.flight.requests.per.connection > 1`

Kafka Best Practices - Producer

- **A batch is ready when one of the following is true**
 - Batch.size is reached
 - Linger.ms is reached
 - Another batch to the same broker is ready
 - flush() or close() is called

Kafka Best Practices - Producer

- **Big batching**
 - Better compression ratio , higher throughput
 - Higher Latency
- **Compression.type**
 - Compression is major part of producer's work
 - Speed of different compression types differs a lot
 - Compression is in user thread, so adding more threads helps with the throughput if compression is slow

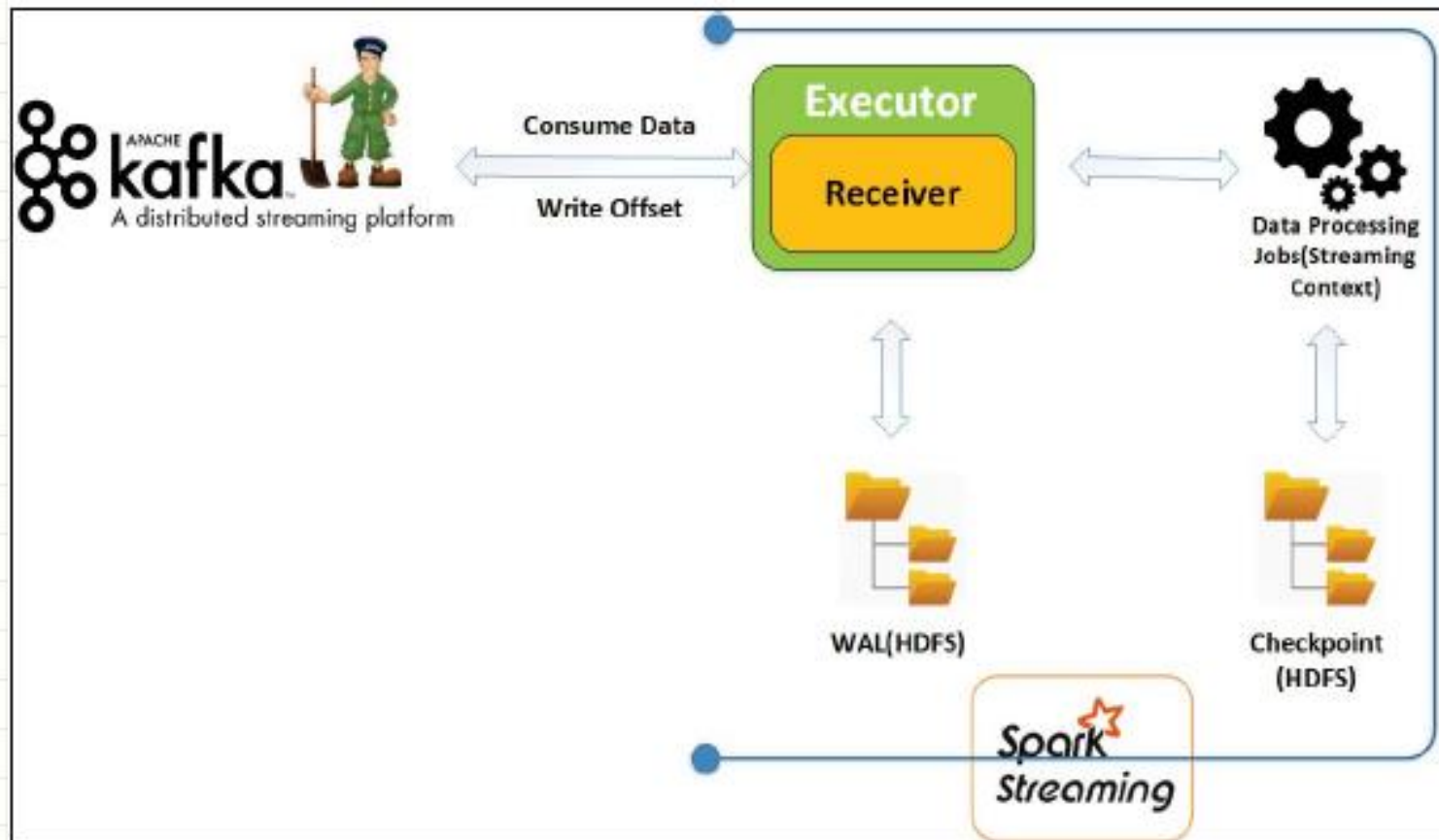
Kafka Best Practices - Producer

- **ACKs**
- **Max.in.flight.requests.per.connection**
 - Max.in.flight.requests.per.connection > 1 means pipelining
 - Gives better throughput
 - May cause out of order delivery when retry occurs
 - Excessive pipelining , drops throughput

Kafka Best Practices - Consumer

- On the consumer side it is generally easy to get good performance without need for tweaking
- Simple rule of thumb for good consumer performance is to keep Number of consumer threads = Partition count

Building Spark Streaming Applications with Kafka



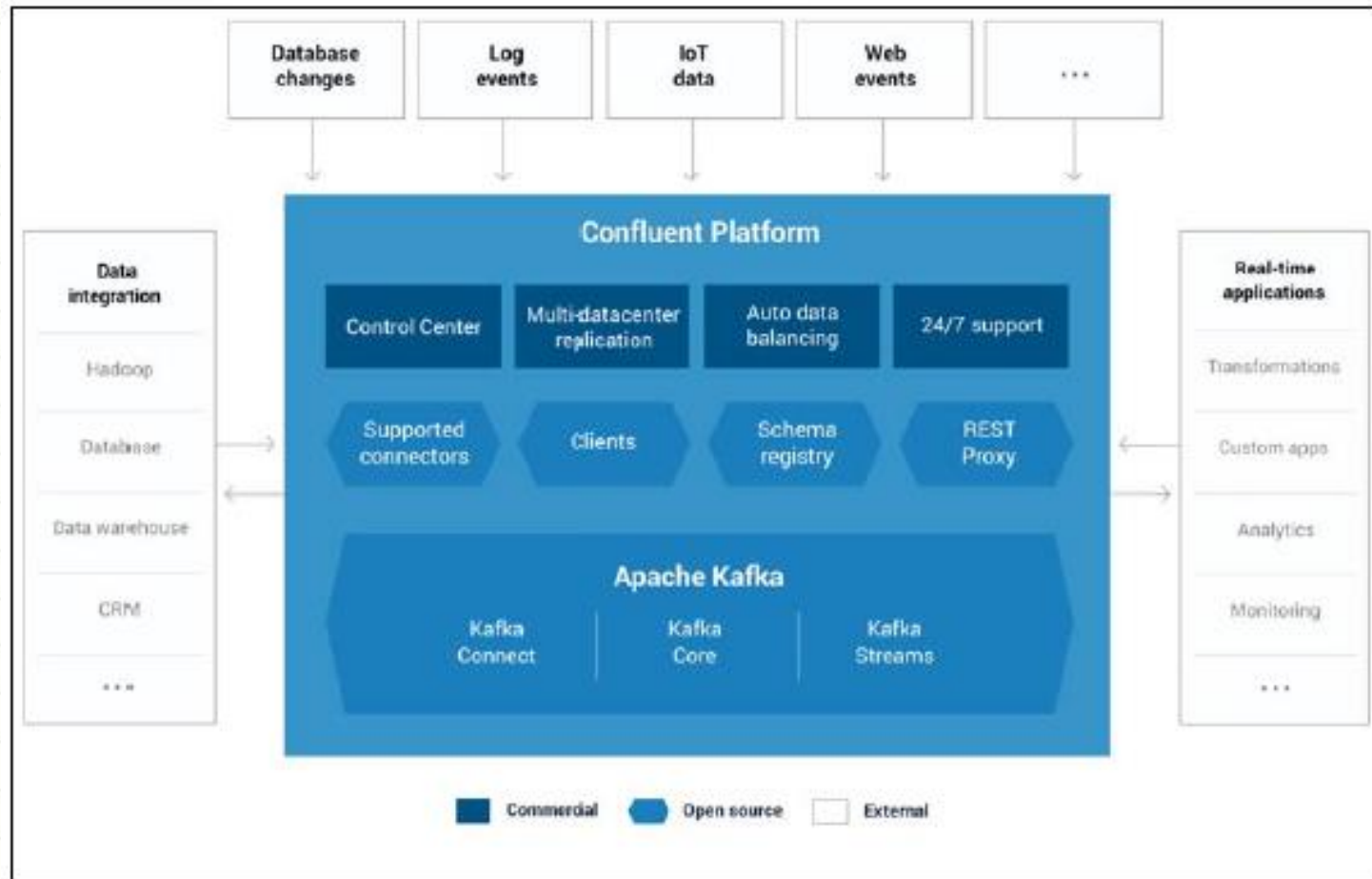
The Confluent Platform

- The Confluent Platform is a full stream data system. It enables you to organize and manage data from several sources in one high-performance and reliable system
- The Confluent Platform has these parts:
 - Confluent Platform open source
 - Confluent Platform enterprise
 - Confluent Cloud

The Confluent Platform

- The Confluent Platform open source has the following components:
 - Apache Kafka core
 - Kafka Streams
 - Kafka Connect
 - Kafka clients
 - Kafka REST Proxy
 - Kafka Schema Registry

The Confluent Platform



Confluent Platform – Components

- **Kafka core**
 - The Kafka brokers discussed at the moment in this book
- **Kafka Streams**
 - The Kafka library used to build stream processing systems
- **Kafka Connect**
 - The framework used to connect Kafka with databases, stores, and filesystems
- **Kafka clients**
 - The libraries for writing/reading messages to/from Kafka. Note that there clients for these languages: Java, Scala, C/C++, Python, and Go

Confluent Platform – Components

- **Kafka REST Proxy**

- If the application doesn't run in the Kafka clients, programming languages, this proxy allows connecting to Kafka through HTTP

- **Kafka Schema Registry**

- Recall that an enterprise service bus should have a message template repository. The Schema Registry is the repository of all the schemas and their historical versions, made to ensure that if an endpoint changes, then all the involved parts are acknowledged.

Confluent Platform – Components

- **Confluent Control Center**

- A powerful web graphic user interface for managing and monitoring Kafka systems

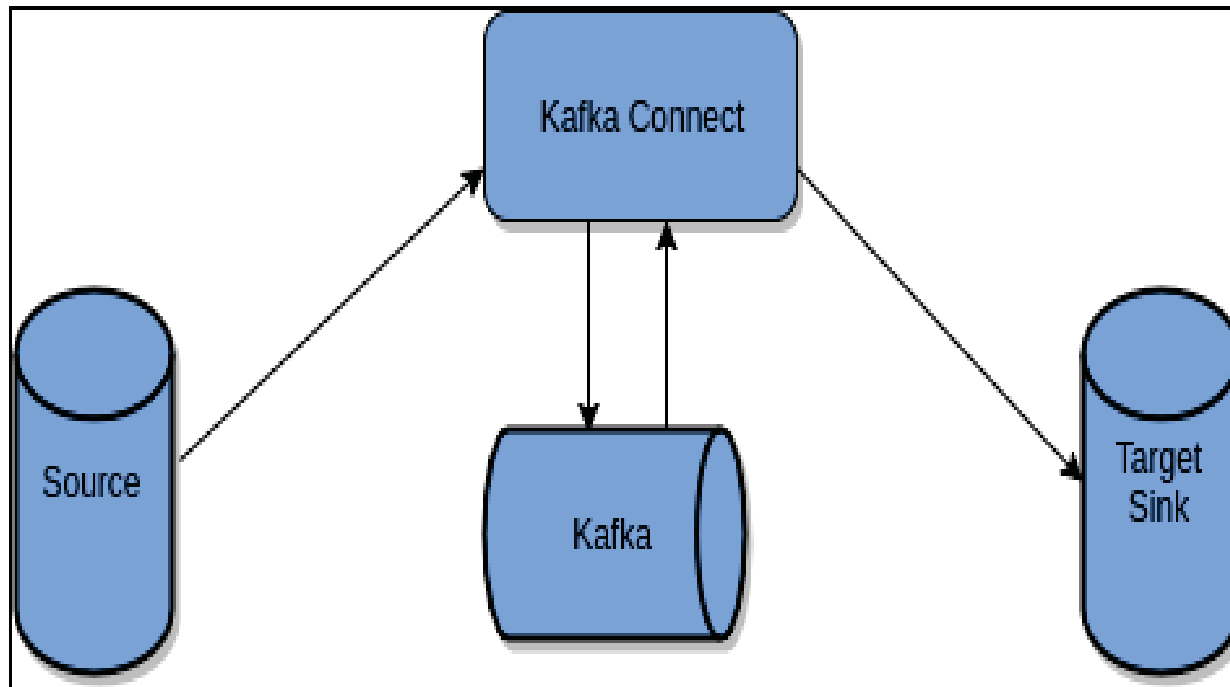
- **Confluent Cloud**

- Kafka as a service—a cloud service to reduce the burden of operations

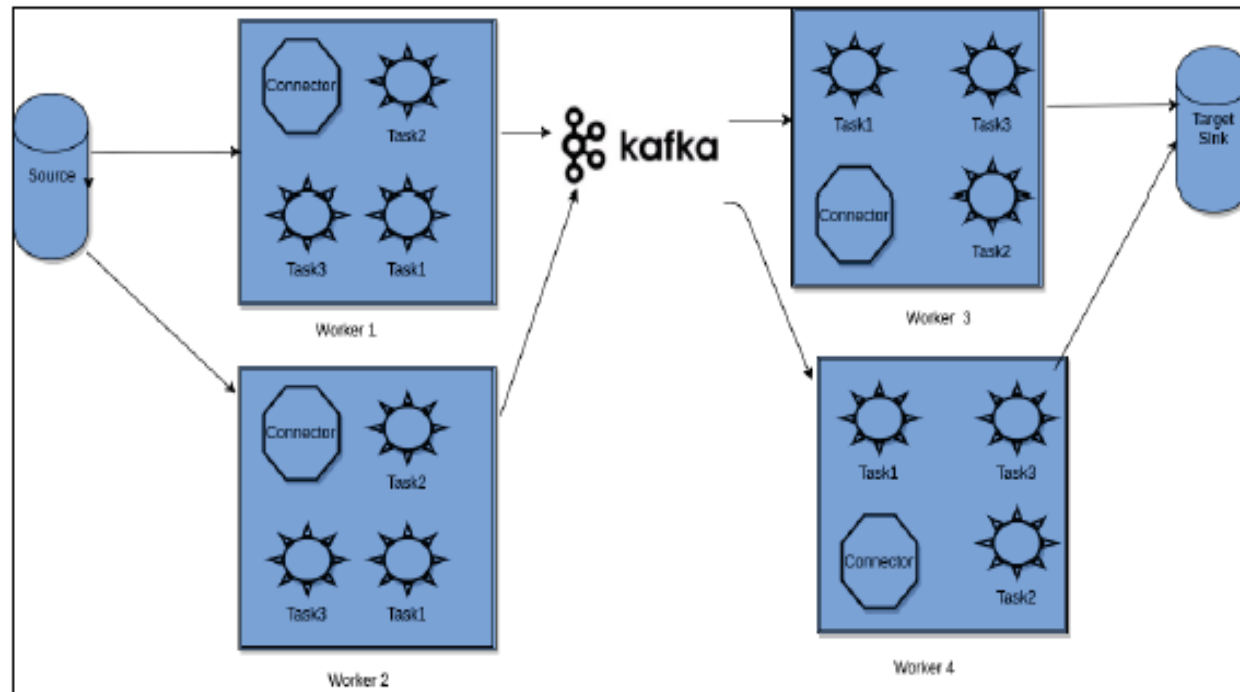
Kafka Streams

- **Stream processing application**
 - Any program that utilizes the Kafka streams library is known as a stream processing application
- **Processor topology**
 - This is a topology that defines the computational logic of the data processing that a stream processing application requires to be performed. A topology is a graph of stream processors (nodes) connected by streams (edges)

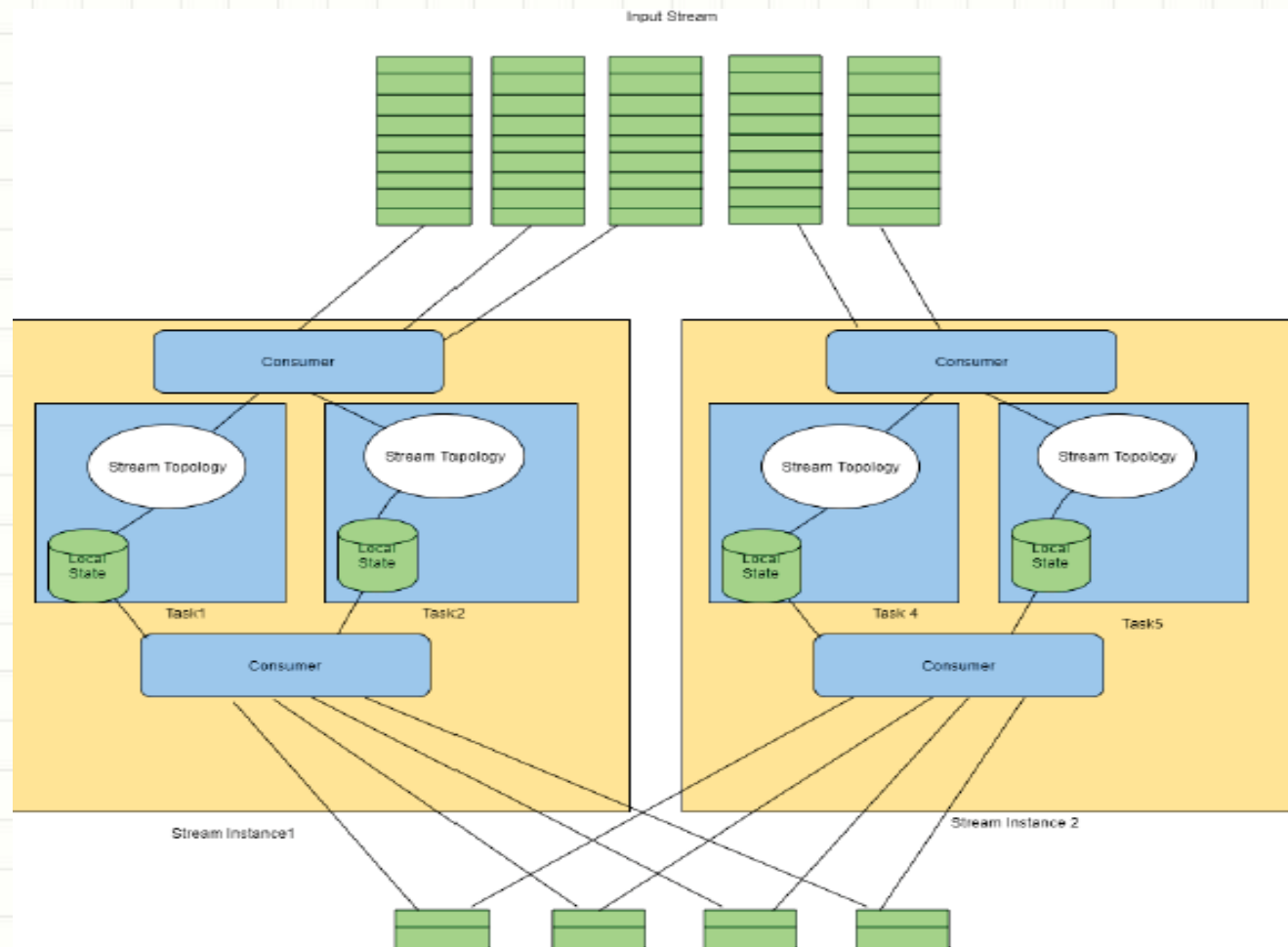
Building ETL Pipelines Using Kafka



Building ETL Pipelines Using Kafka



Building Streaming Applications Using Kafka Streams



Using Kafka in Big Data Applications

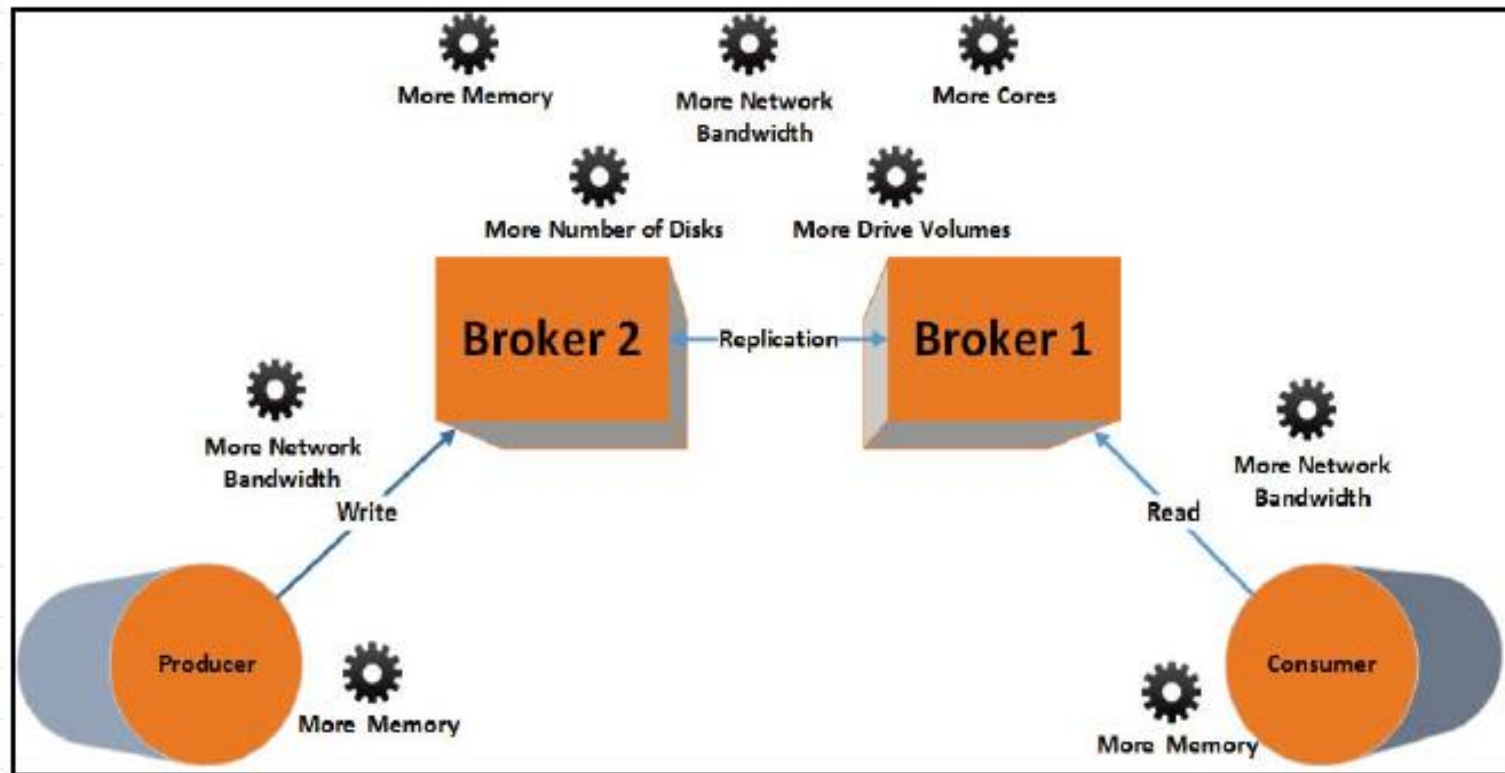
- Managing high volumes in Kafka
- Kafka message delivery semantics
- Failure handling and retry-ability
- Big data and Kafka common usage patterns
- Kafka and data governance
- Alerting and monitoring
- Useful Kafka matrices

Managing high volumes in Kafka

- High volume of writes or high message writing throughput
- High volumes of reads or high message reading throughput
- High volume of replication rate
- High disk flush or I/O

Managing high volumes in Kafka

- Appropriate hardware choices

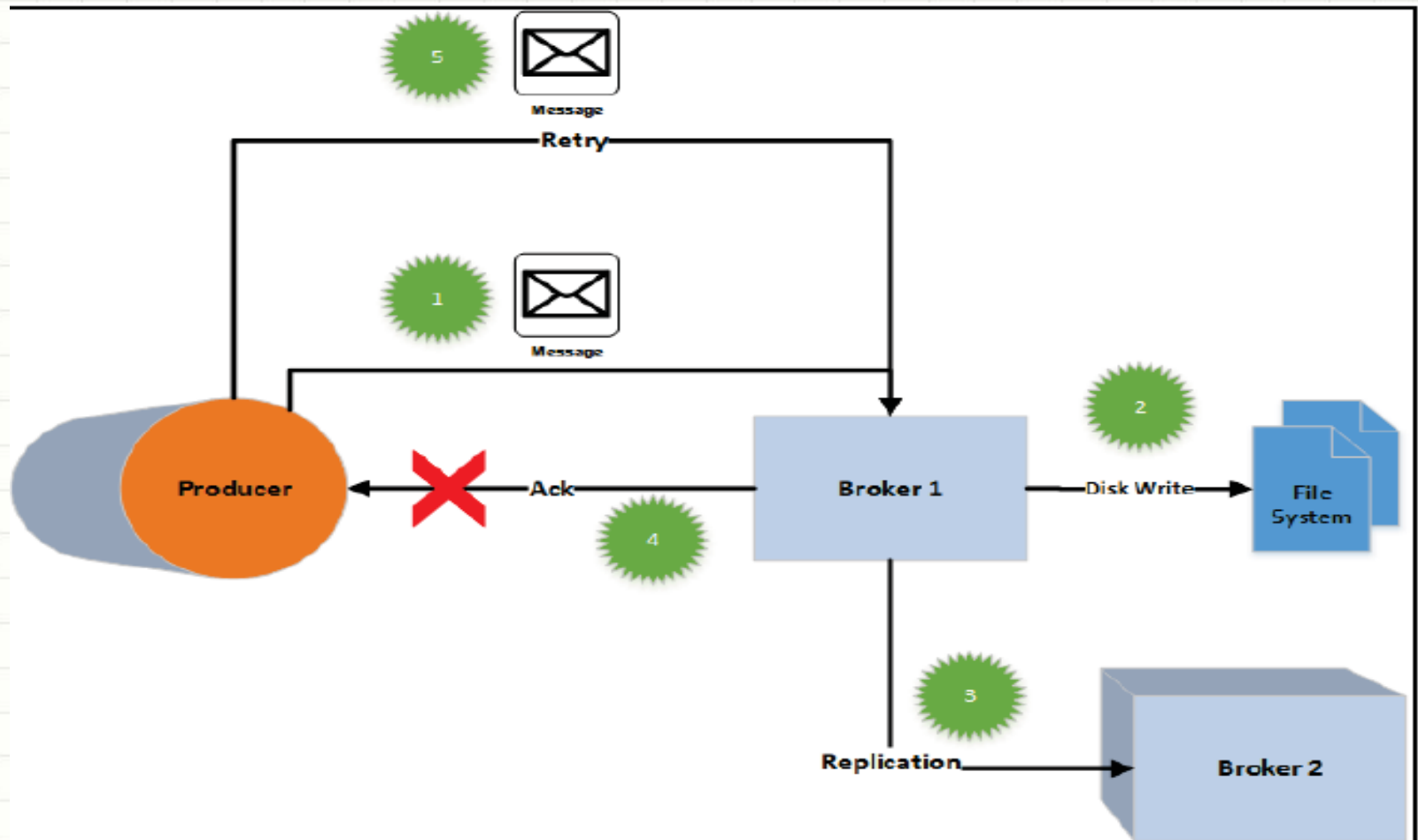


Managing high volumes in Kafka

- Producer read and consumer write choices
 - Message compression
 - Message batches
 - Asynchronous send
 - Linger time
 - Fetch size

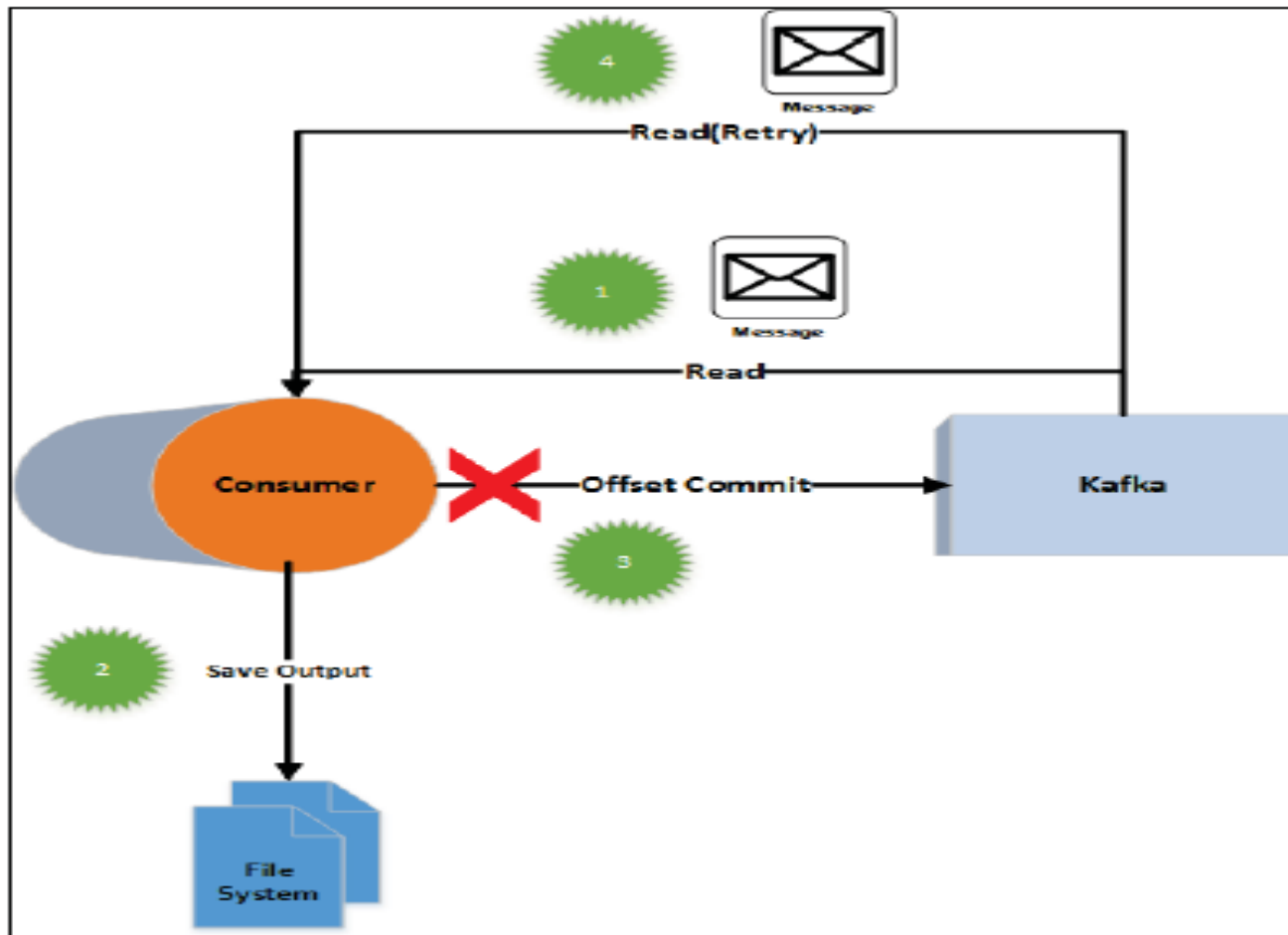
Kafka Message Delivery Semantics

- At least once delivery



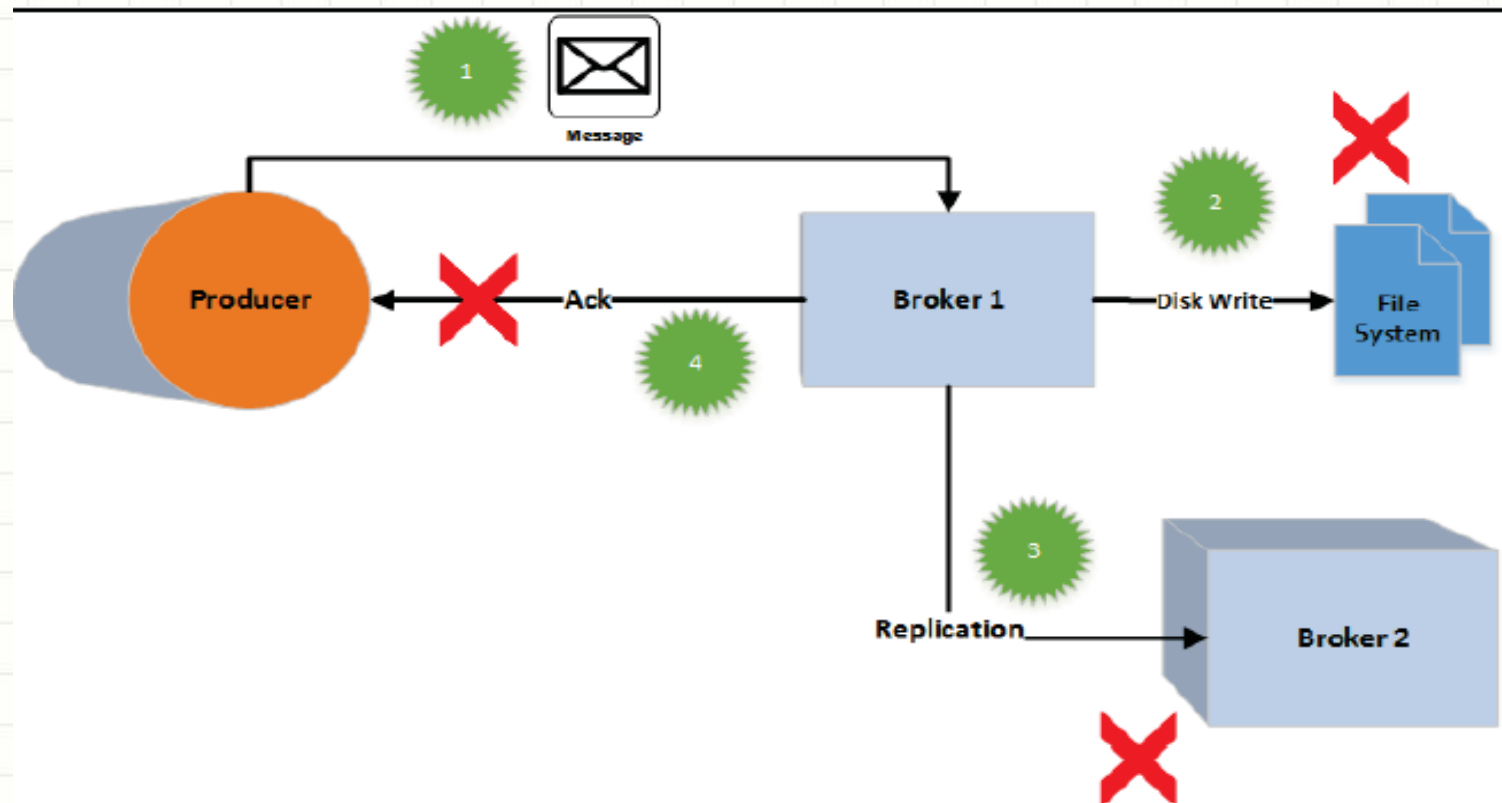
Kafka Message Delivery Semantics

- At least once delivery



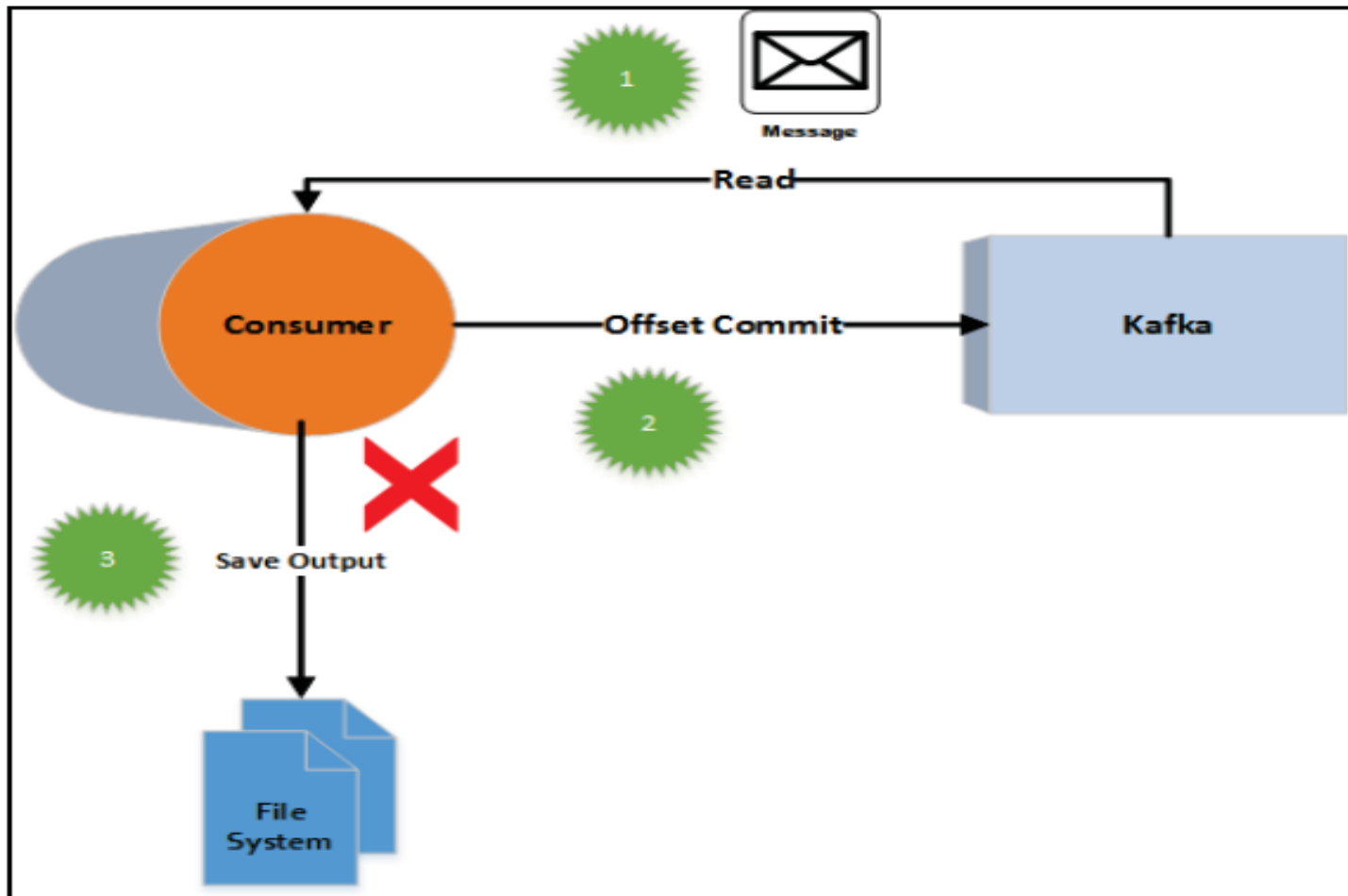
Kafka Message Delivery Semantics

- At most once delivery



Kafka Message Delivery Semantics

- At most once delivery



Kafka Message Delivery Semantics

- **Exactly once delivery**

- **Producer side**

- ✓ Idempotent Producers

- ❖ Idempotent producers generate a unique key identifier for each batch of messages, When the message batches are stored by the broker in Kafka logs, they also have a unique number

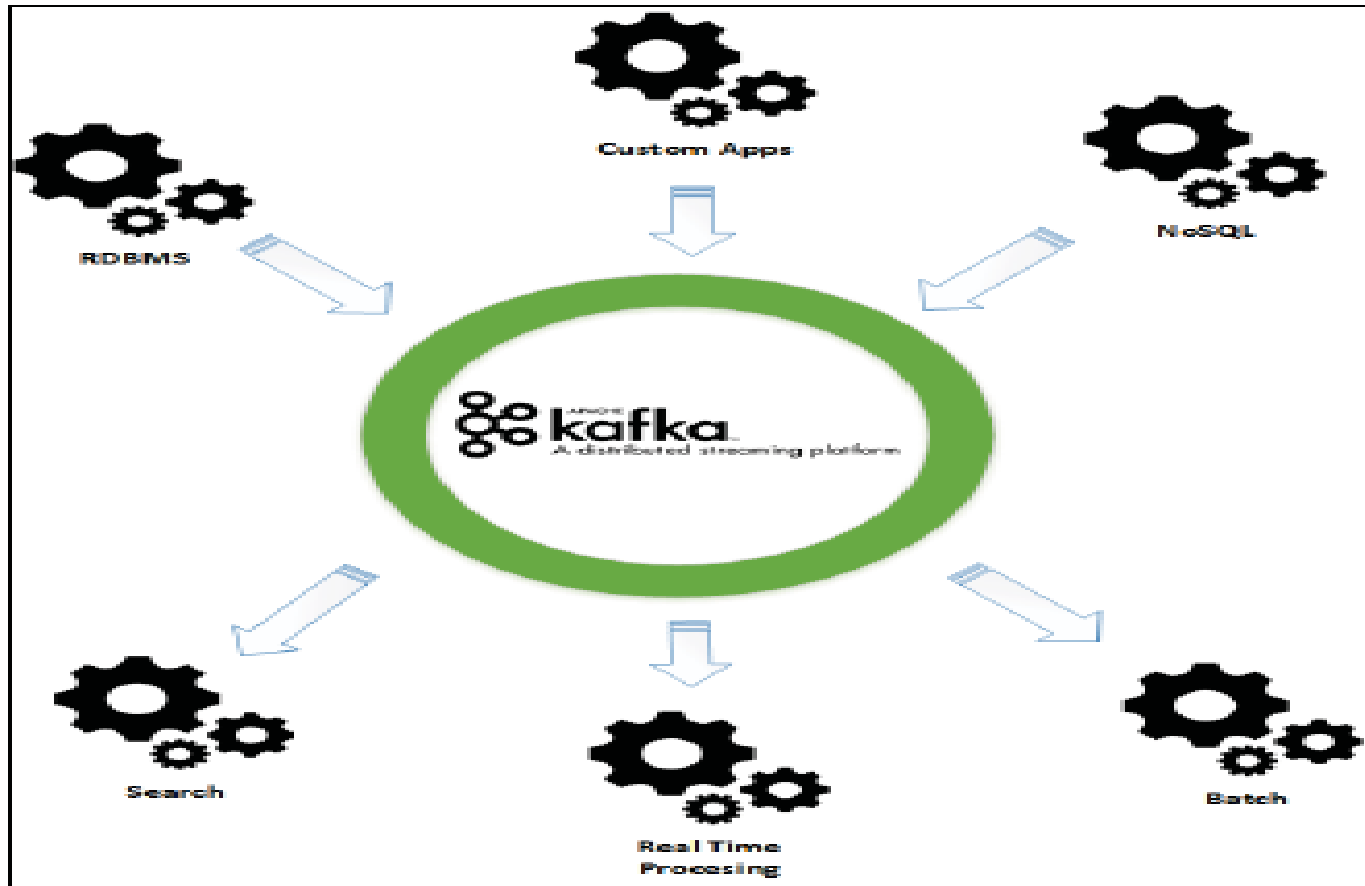
- ❖ support for transactions APIs

- **Consumer side**

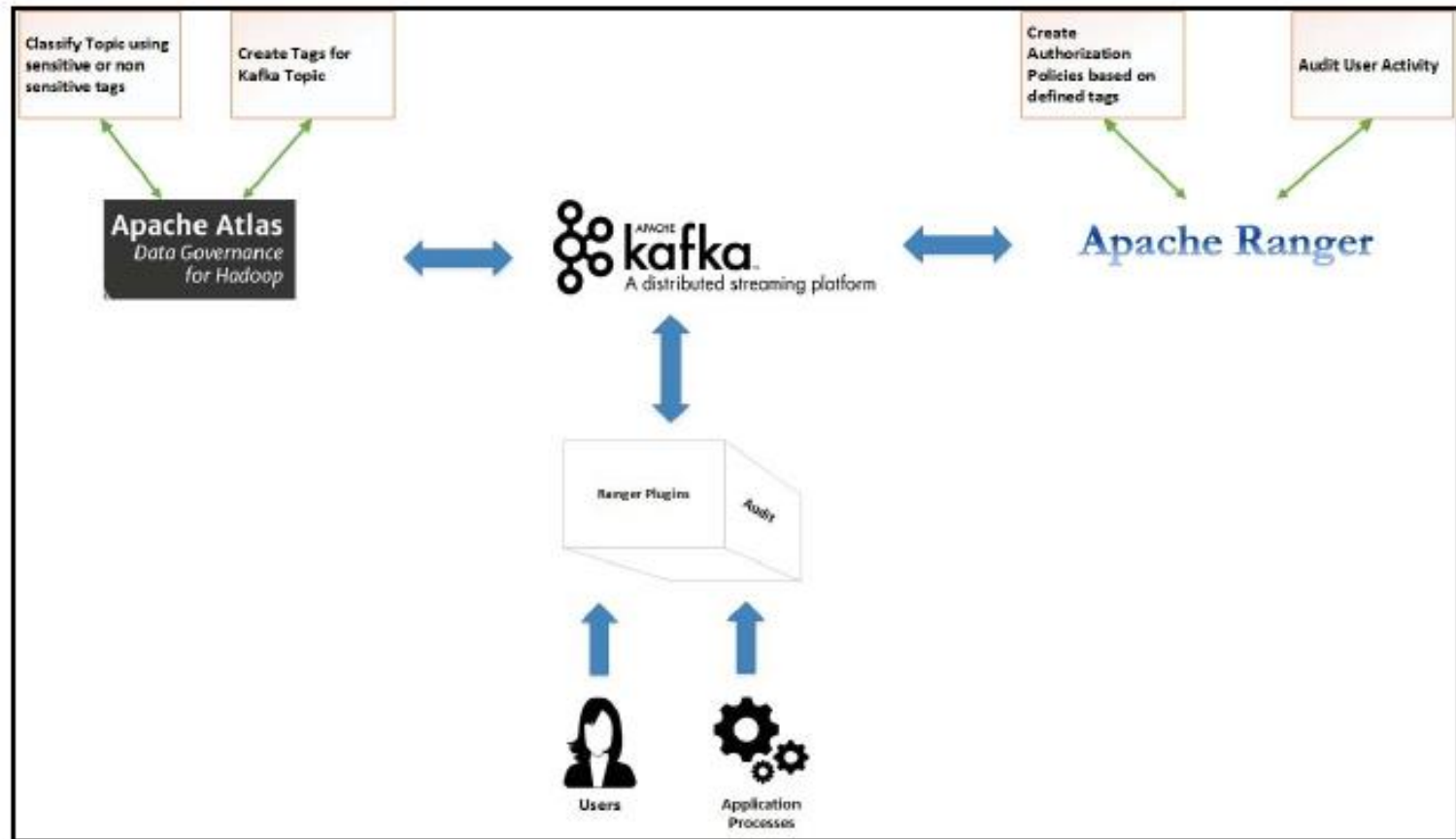
- ✓ read_committed

- ✓ read_uncommitted

Big data and Kafka Common usage patterns



Kafka and Data Governance



Alerting and Monitoring

- Alerting and monitoring
 - Avoid data loss
 - Producer performance
 - Consumer performance
 - Data availability

Useful Kafka Matrices

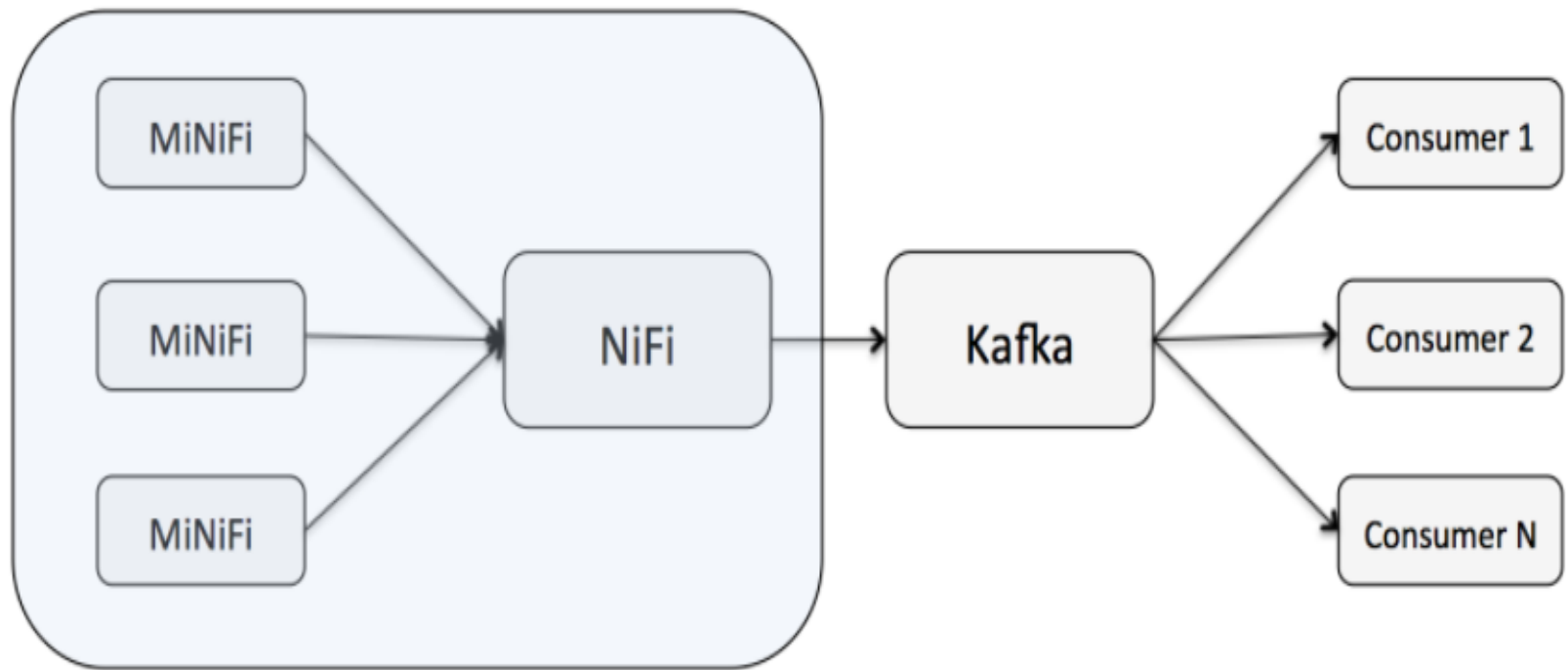
- Useful Kafka matrices
 - Kafka producer matrices
 - Kafka broker matrices
 - Kafka consumer matrices

Streaming Application Design Considerations

- Latency and throughput
- Data persistence
- Data sources
- Data lookups
- Data formats
- Data serialization
- Level of parallelism
- Data skews
- Out-of-order events
- Memory tuning
- <https>

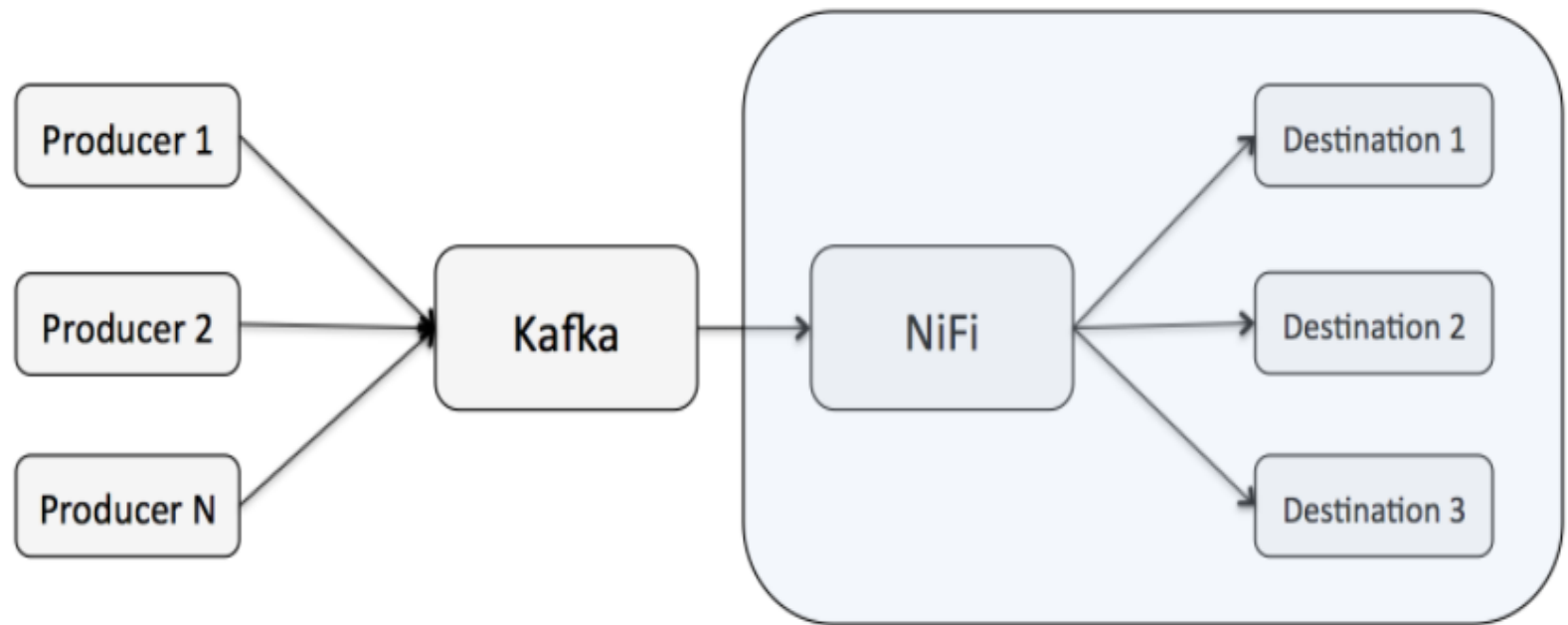
Apache NiFi and Kafka

- **NiFi as a Producer**



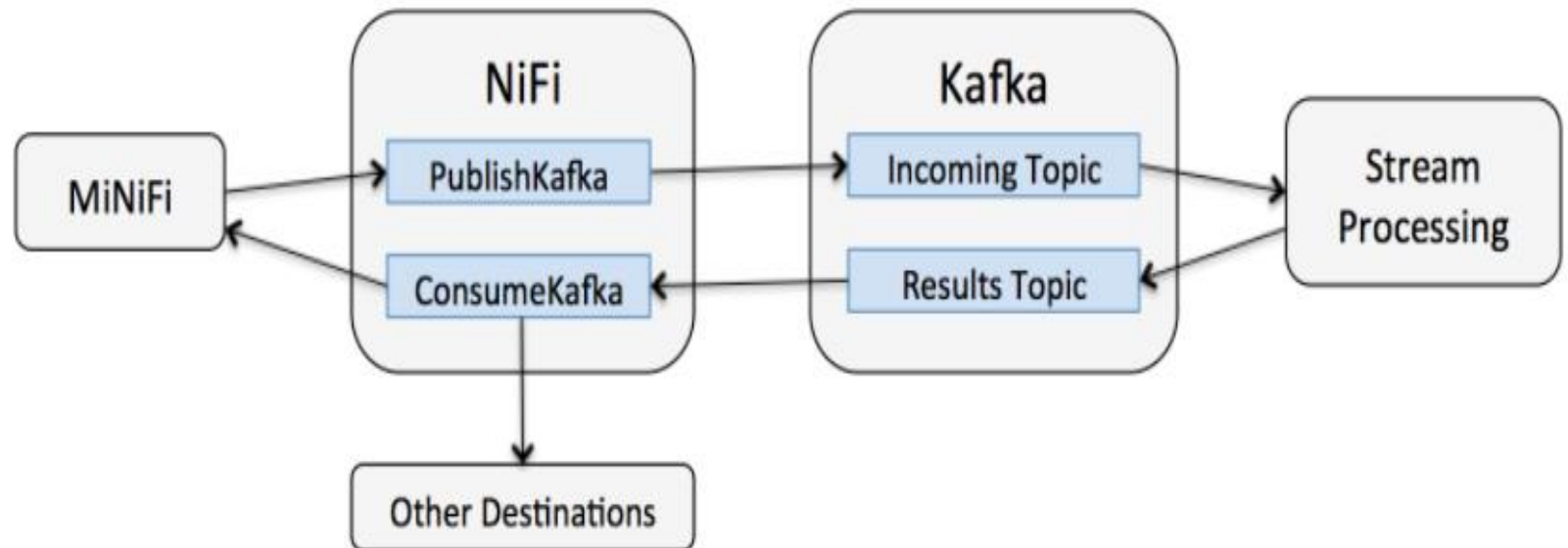
Apache NiFi and Kafka

- **NiFi as a Consumer**



Apache NiFi and Kafka

- **Bi-Directional Data Flows**



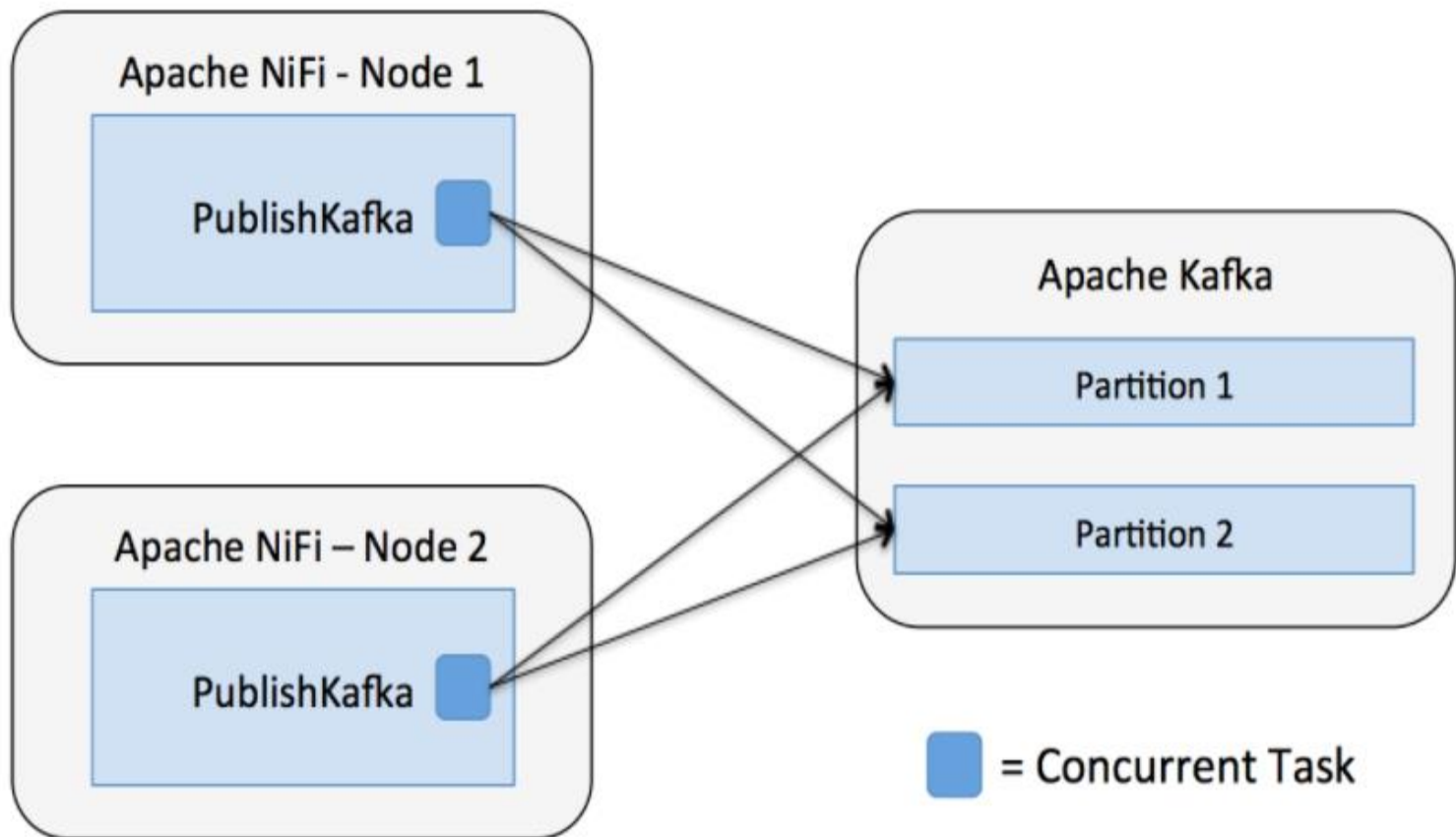
Apache NiFi and Kafka

The Apache NiFi contains the following Kafka processors:

- GetKafka & PutKafka using the 0.8 client
- ConsumeKafka & PublishKafka using the 0.9 client
- ConsumeKafka_0_10 & PublishKafka_0_10 using the 0.10 client

Apache NiFi and Kafka

- PublishKafka



Apache NiFi and Kafka

- PublishKafka

SETTINGS

SCHEDULING

PROPERTIES

COMMENTS

Required field

+

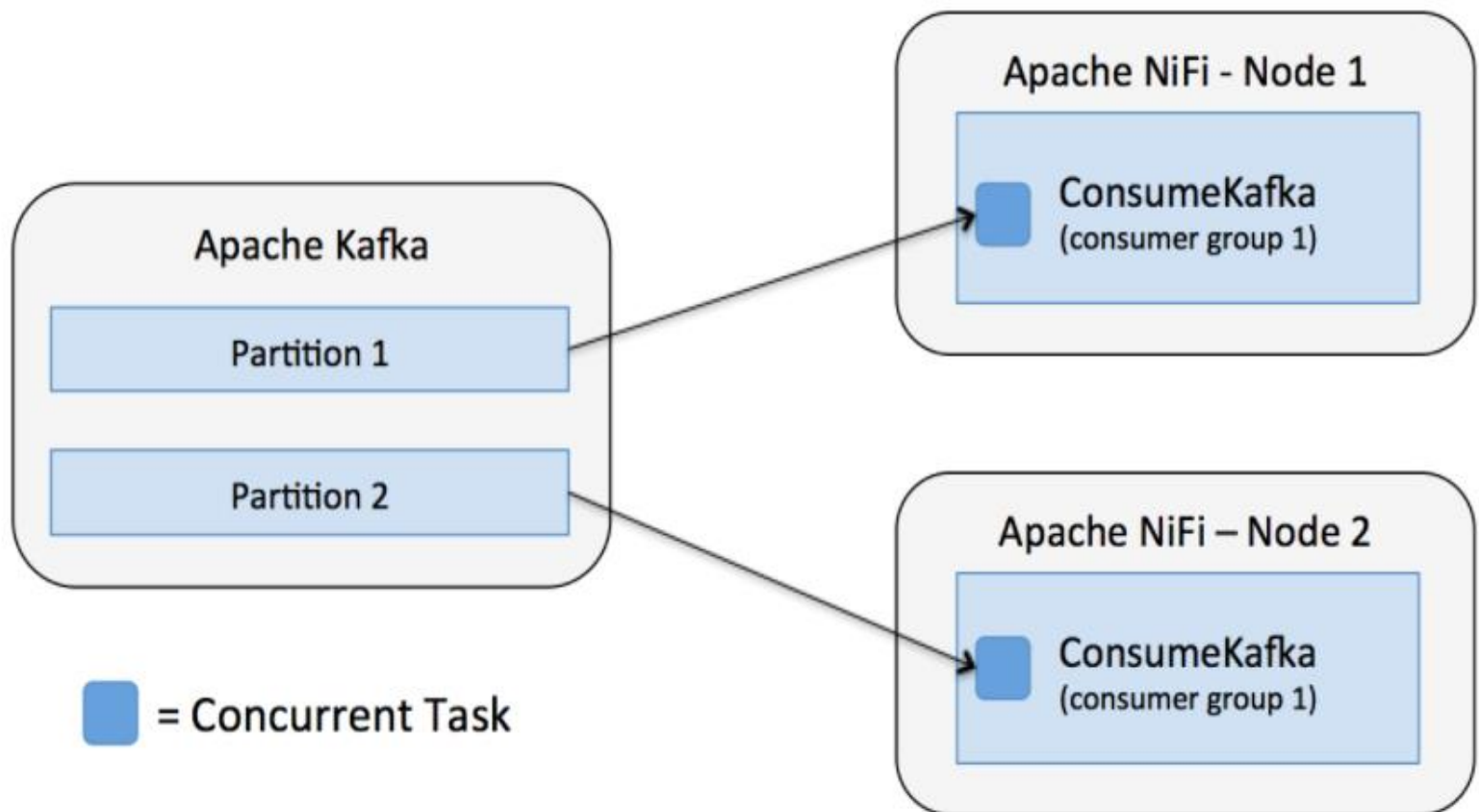
Property		Value	
Kafka Brokers	?	localhost:9092	
Security Protocol	?	PLAINTEXT	
Kerberos Service Name	?	No value set	
SSL Context Service	?	No value set	
Topic Name	?	No value set	
Delivery Guarantee	?	Best Effort	
Kafka Key	?	No value set	
Key Attribute Encoding	?	UTF-8 Encoded	
Message Demarcator	?	No value set	
Max Request Size	?	1 MB	
Meta Data Wait Time	?	30 sec	
Partitioner class	?	DefaultPartitioner	
Compression Type	?	none	

CANCEL

APPLY

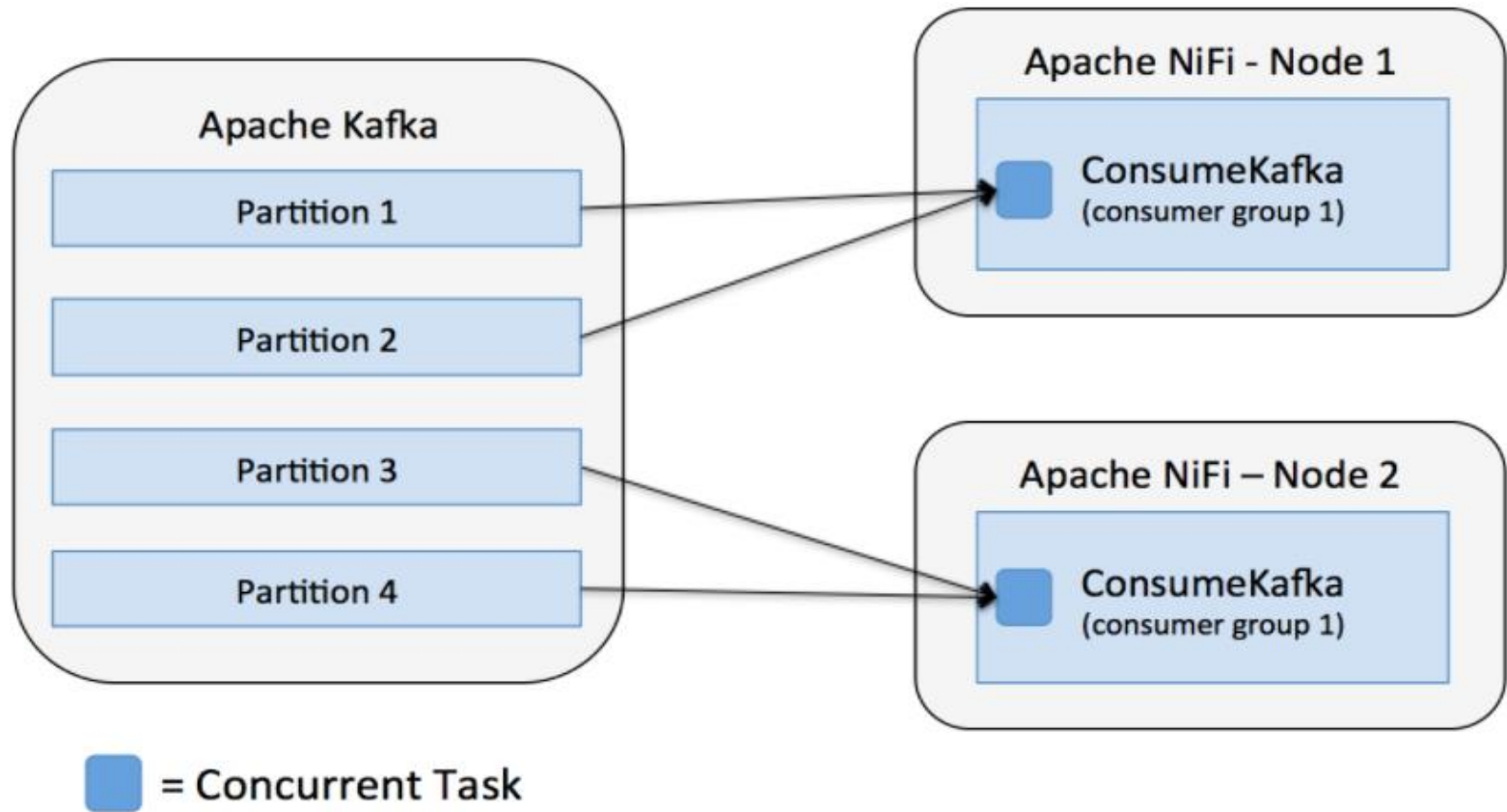
Apache NiFi and Kafka

- **ConsumeKafka**



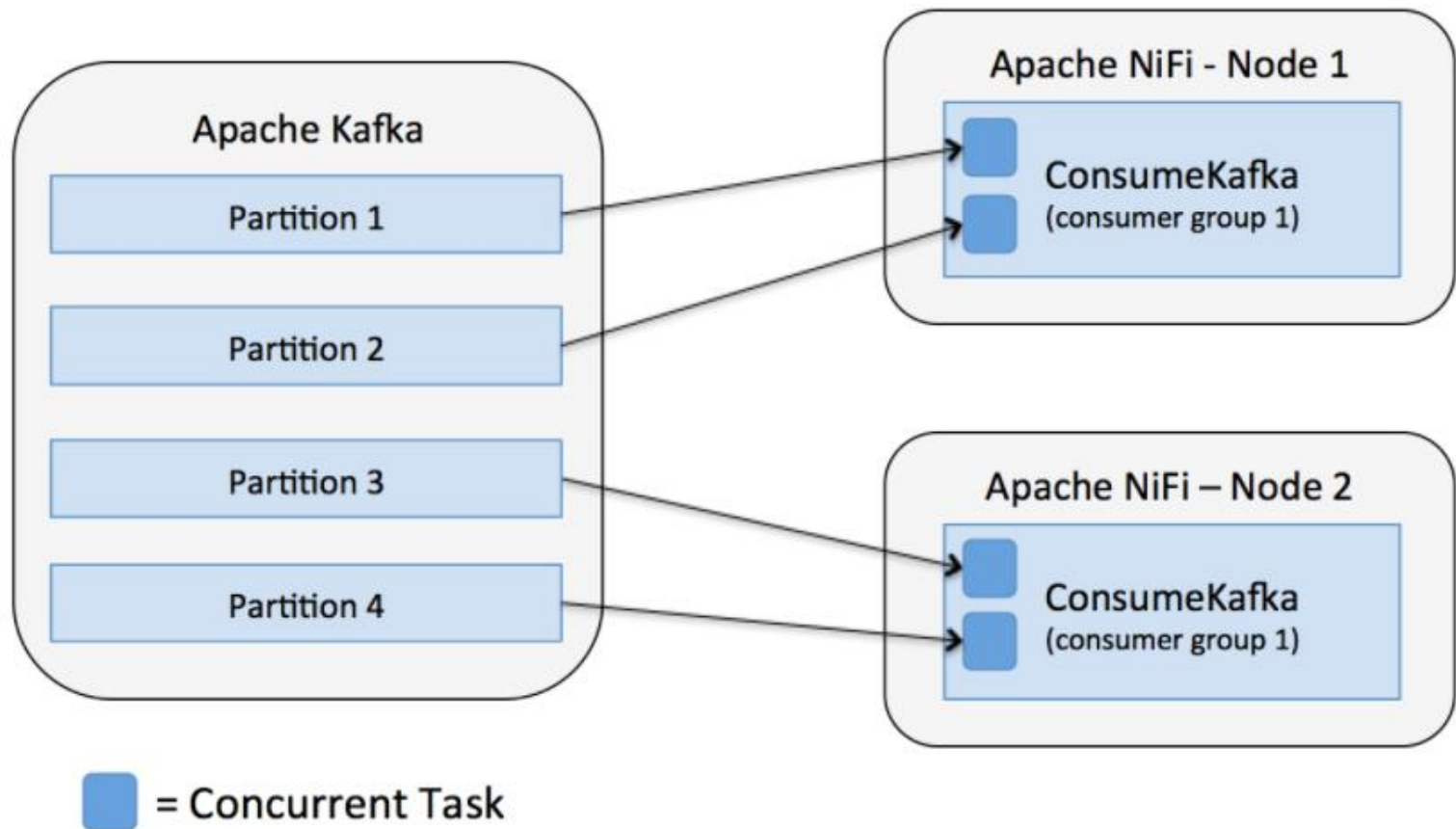
Apache NiFi and Kafka

- **ConsumeKafka**



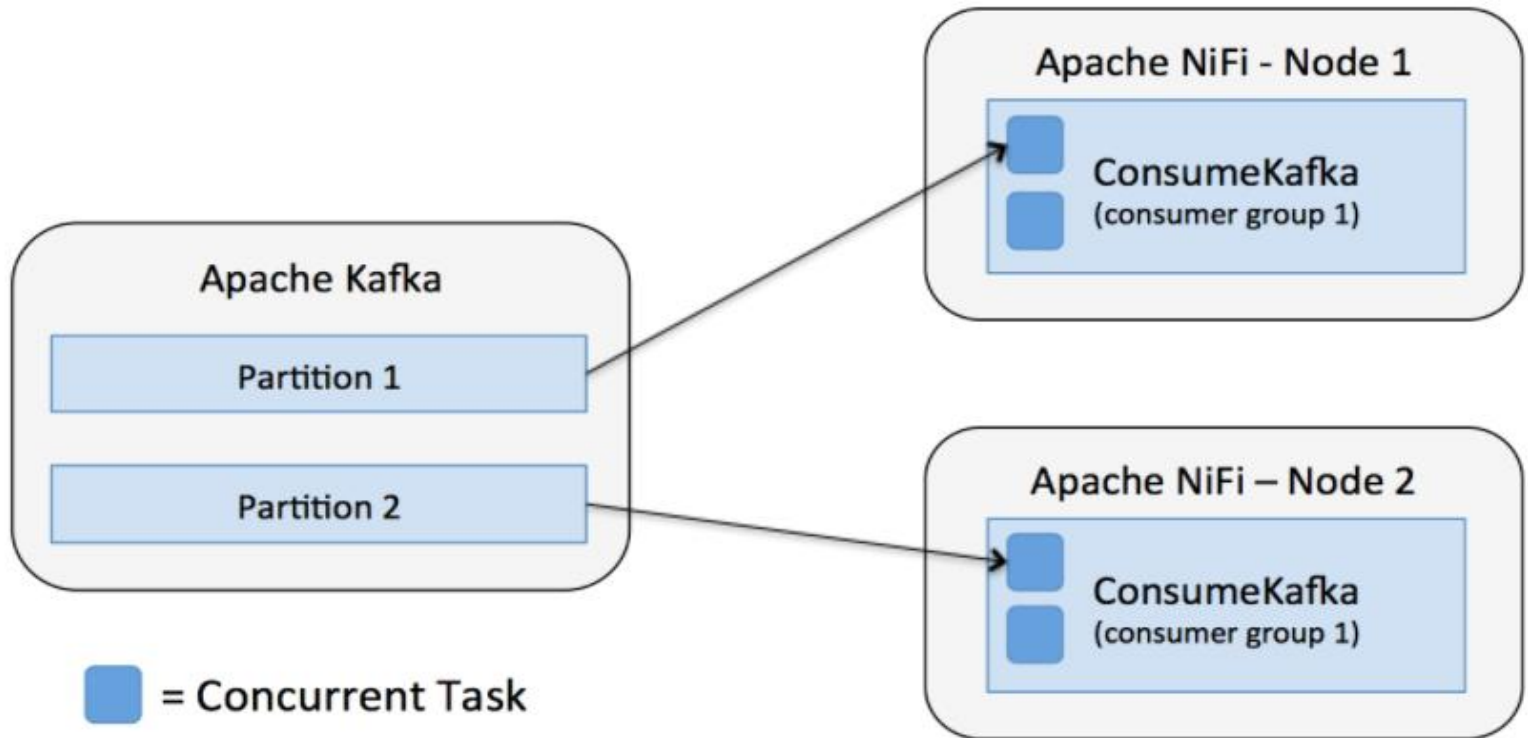
Apache NiFi and Kafka

- **ConsumeKafka**



Apache NiFi and Kafka

- **ConsumeKafka**



Apache NiFi and Kafka

- **ConsumeKafka**

Configure Processor

SETTINGS

SCHEDULING

PROPERTIES

COMMENTS

Required field



Property		Value	
Kafka Brokers	?	localhost:9092	
Security Protocol	?	PLAINTEXT	
Kerberos Service Name	?	No value set	
SSL Context Service	?	No value set	
Topic Name(s)	?	No value set	
Group ID	?	No value set	
Offset Reset	?	latest	
Key Attribute Encoding	?	UTF-8 Encoded	
Message Demarcator	?	No value set	
Max Poll Records	?	10000	
Max Uncommitted Time	?	1 secs	

Apache NiFi and Kafka

- **Security scenarios**

This is controlled through the Security Protocol property which has the following options:

- PLAINTEXT
- SSL
- SASL_PLAINTEXT
- SASL_SSL

Apache NiFi and Kafka

- **Performance Considerations**

Message Demarcator:

- On the publishing side, the demarcator indicates that incoming flow files will have multiple messages in the content
- On the consuming side, the demarcator indicates that ConsumeKafka should produce a single flow file with the content containing all of the messages received from Kafka in a single poll

Building Spark Streaming Applications with Kafka

