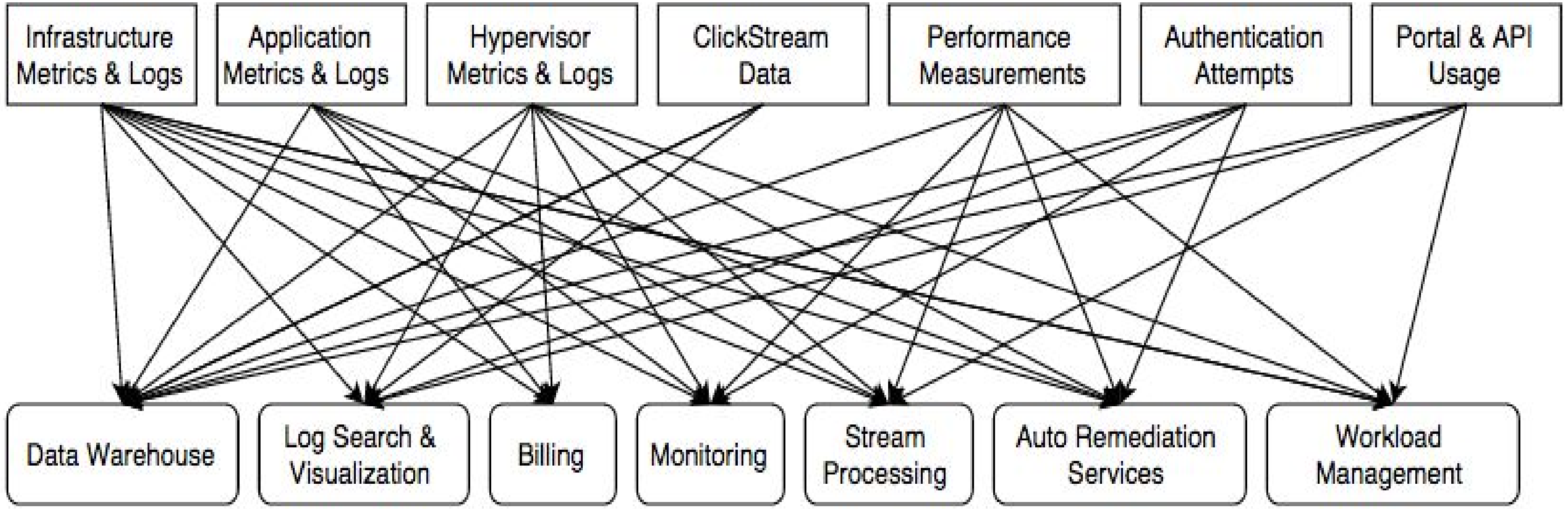
Introduction to Messaging Systems

Kafka

Synchronous vs Asynchronous Processing

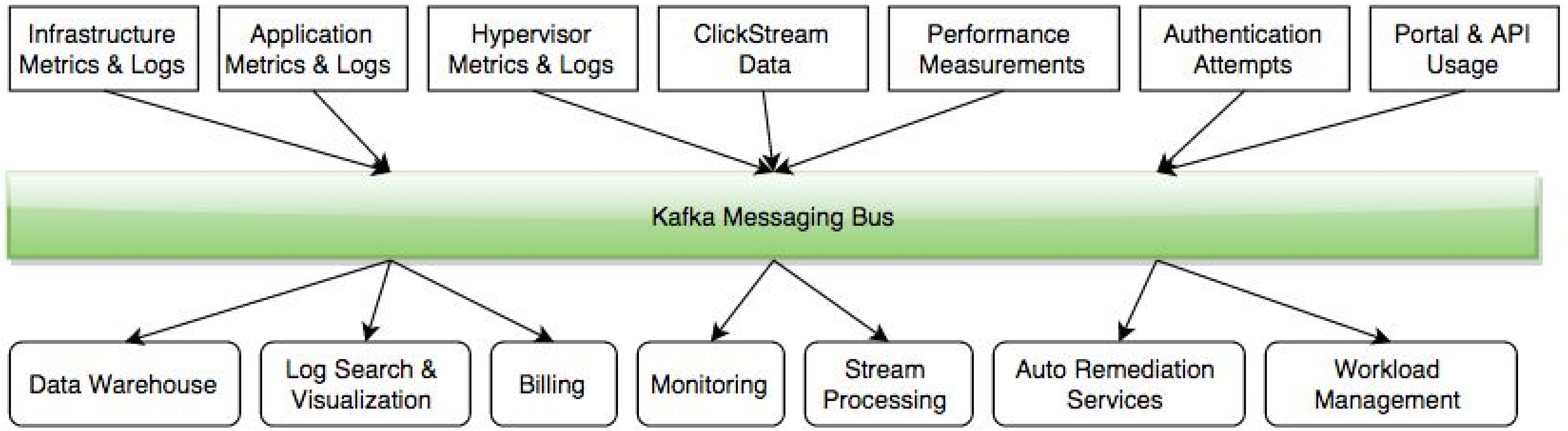
- Problems with synchronous processes
- Advantages of Asynchronous
  - thread don't have to wait for call to finish.
  - services can afford to be slow.
  - More stable
- Where to and not to use Asynchronous
- Application of Asynchronous behaviour.
  - chat application
  - real time bidding systems
  - payment processing systems
  - http://google.com

Point-to-point pipeline



Problems with point to point pipeline

- What if a service is slow?
- What if a service is down? (what if it is on purpose - maintenance, code upload, bug fix, etc)
- What if another service now wants to consume data?
- Many services (even faster) will make the system slow
- Complicated…
- Network Load


Infrastructure Metrics & Logs
Application Metrics & Logs
Hypervisor Metrics & Logs
ClickStream Data
Performance Measurements
Authentication Attempts
Portal & API Usage
Kafka Messaging Bus
Data Warehouse
Log Search & Visualization
Billing
Monitoring
Stream Processing
Auto Remediation Services
Workload Management

Benefits of messaging systems

- Decoupling of services
- Asynchronous producer and consumers services
- Communication can be driven by events
- Application can assign priority to a message(Events can be delayed)
- Recovery support
- Scalability

Must have features

- Fast - high throughput
- Low latency delivery
- Scalable
- Reliable
- Fault tolerant
- durable

Design

- Delivery semantics
- Styles: Queuing vs publish-subscribe
- Consuming policies: Push vs pull

Message Delivery Semantics:

- At most once—Messages may be lost but are never redelivered. (Fire and Forget strategy)
- *At least once*—Messages are never lost but may be redelivered.
- *Exactly once*—this is what people actually want, each message is delivered once and only once.

Queuing vs pub/sub

Queuing

One message is placed on the queue and one application receives that message.

Publish/Subscribe

A copy of each message published by a publishing application is delivered to every interested application.

Push vs Pull:

- Most messaging systems follow a **Pull based** Design, where data is pushed to the broker from the producer and pulled from the broker by the consumer.
- Advantages of pull based system for consumers:
    - The consumer wont be overwhelmed with messages when its rate of consumption falls below the production rate (a denial of service attack, in essence) as in case of push based system
    - In a pull based system consumer can do aggressive batching of data and consume

Design a messaging queue

- Need Things done in order
- Responsible Message delivery
- Persistent
- High Throughput
- High Available
- Multiple Events

Two such systems (Kafka and RabbitMQ)

- Different design decisions (push vs pull)
- Different implementations (in memory vs disk)
- Different use cases

Apache Kafka

A high-throughput distributed messaging system.

Topics

- A topic is a category or feed name to which messages are published

Partitions

- A topic consists of **partitions.**
- Partition: **ordered + immutable** sequence of messages that is continually appended to

Partitions

- No of partitions of a topic is configurable
- No of partitions determines **max** consumers.They act as the unit of parallelism

Replicas of a partition

- “backups” of a partition
- They exist solely to prevent data loss.
- Replicas are never read from, never written to.
- They do NOT help to increase producer or consumer parallelism!
- Kafka tolerates *(numReplicas - 1)* dead brokers before losing data

Producer

- The producer sends data directly to the broker that is the leader for the partition without any intervening routing tier.
- **The producer is responsible for choosing which message to assign to which partition within the topic**
- This can be done in a round-robin fashion simply to balance load or it can be done according to some semantic partition function (say based on some key in the message)
- Main configs : https://github.com/apache/kafka/blob/0.9.0/config/producer.properties

Consumers

- Messaging traditionally has two models: **queuing** and **publish-subscribe**.
- If all the consumer instances have the **same consumer group**, then this works just like a traditional queue balancing load over the consumers.
- If all the consumer instances have **different consumer groups**, then this works like publish-subscribe and all messages are broadcast to all consumers.

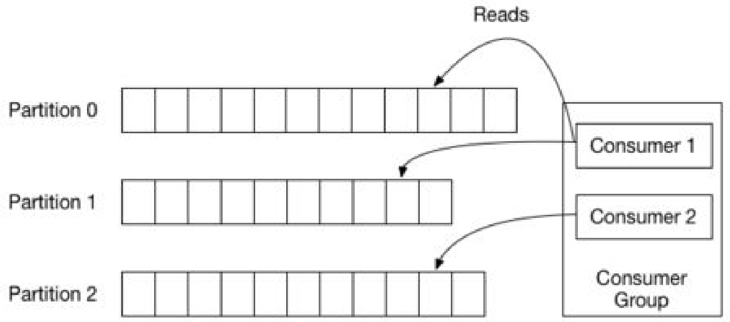
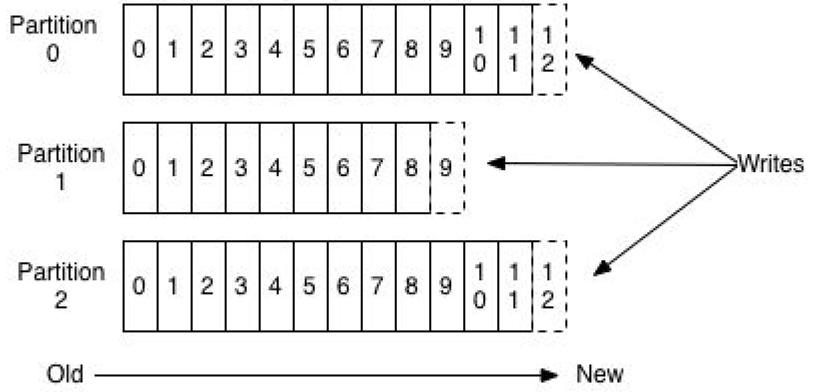


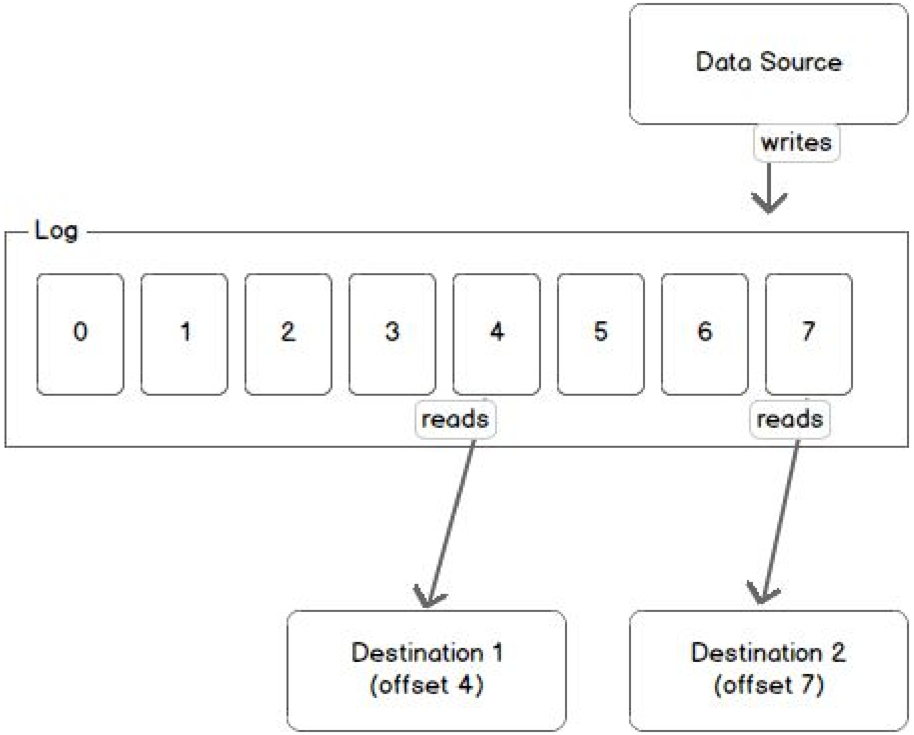
Figure 1: Consumer Group

Kafka: A first look

- **Producers** write data to **brokers**.
- **Consumers** read data from **brokers**.
- All this is distributed.
- Data is stored in **topics**.
- **Topics** are split into **partitions**, which are **replicated**.

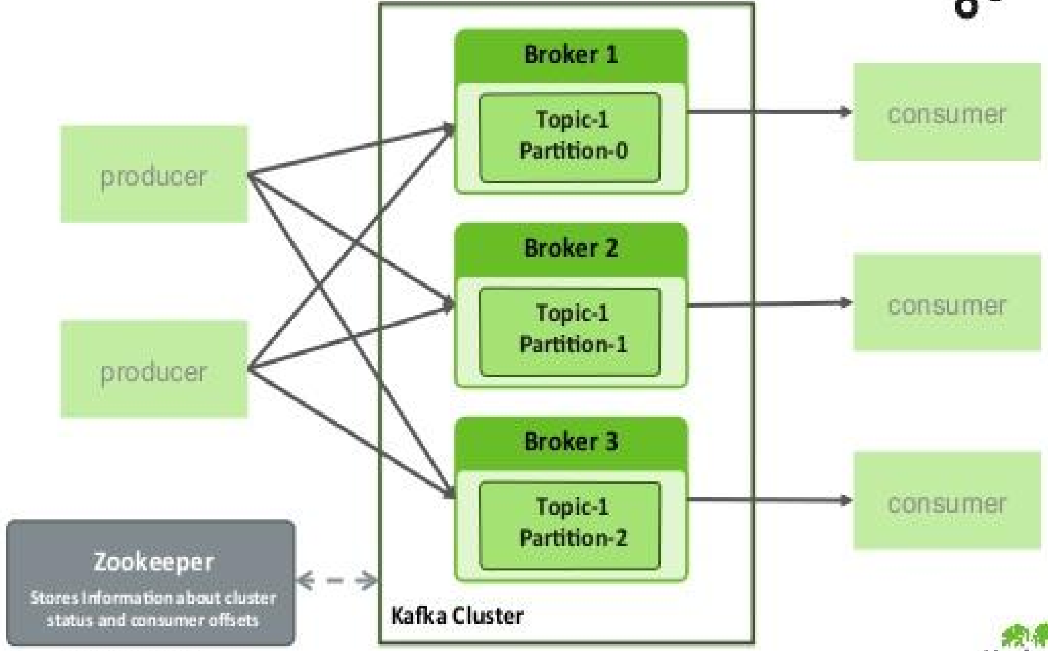
Anatomy of a Topic





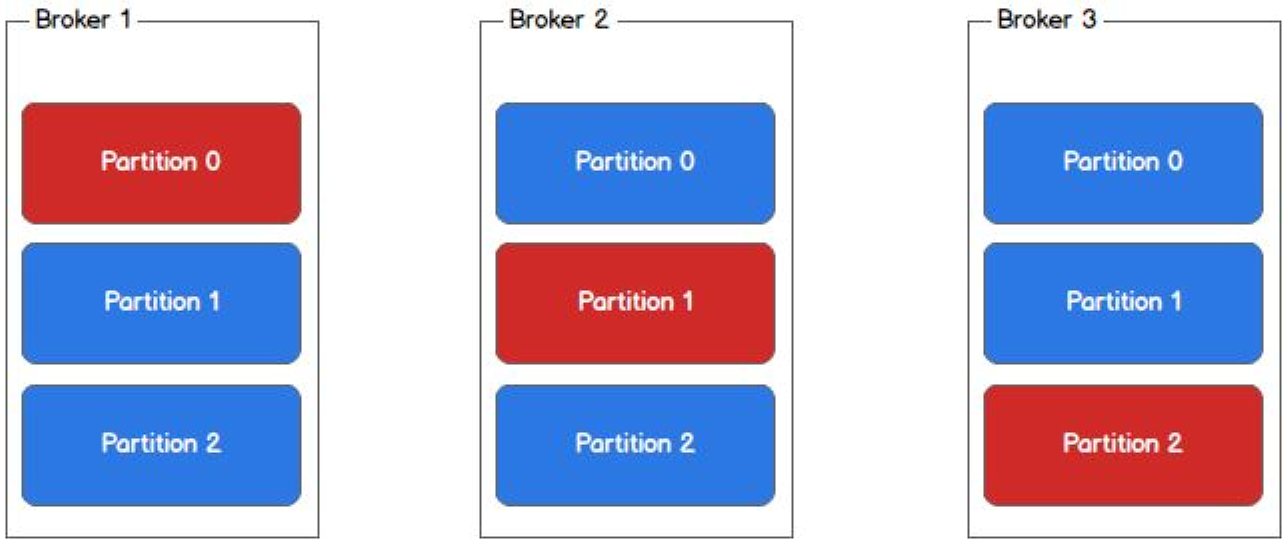
Kafka: Under the Hood

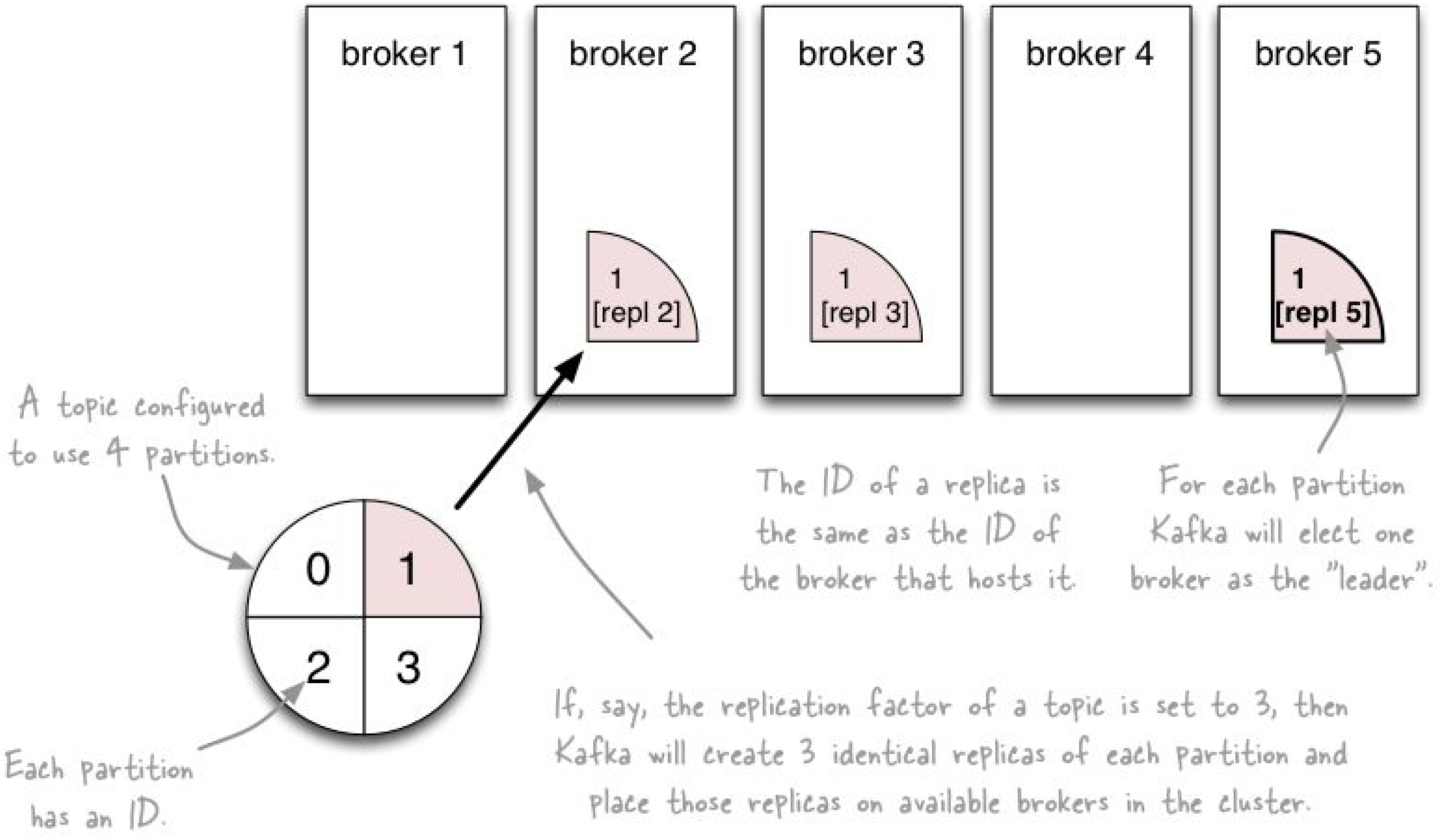
APACHE KAFKA



Hortonworks

Leader (red) and replicas (blue)




broker 1
broker 2
broker 3
broker 4
broker 5
1 [repl 2]
1 [repl 3]
1 [repl 5]
0
1
2
3
A topic configured to use 4 partitions.
Each partition has an ID.
The ID of a replica is the same as the ID of the broker that hosts it.
For each partition Kafka will elect one broker as the "leader".
If, say, the replication factor of a topic is set to 3, then Kafka will create 3 identical replicas of each partition and place those replicas on available brokers in the cluster.

Ordering Guarantees

- This is achieved by assigning the partitions in the topic to the consumers in the consumer group so that each partition is consumed by exactly one consumer in the group
- By doing this we ensure that the consumer is the only reader of that partition and consumes the data in order.
- **Note however that there cannot be more consumer instances in a consumer group than partitions.**
- Kafka only provides a total order over messages *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.

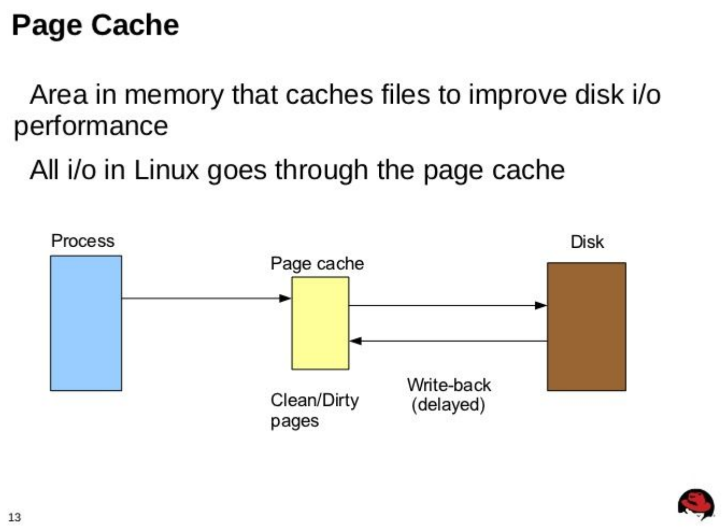
Practical Session

Design

Persistence:

- Kafka relies heavily on the OS pagecache for data storage.
- Although kafka writes to disk immediately, that is not completely true. Actually Kafka just writes to the filesystem immediately, which is really just writing to the kernel's memory pool which is asynchronously flushed to disk.
- Kafka does only sequential file I/O. Sequential disk access can in some cases be faster than random memory access .

- Kafka doesn’t use an in memory cache , it relies on the OS page cache and the file system . This is a good idea: Kafka runs on the JVM and keeping data in the heap of a garbage collected language isn't wise because of GC overhead of continually scanning your in-memory cache
- How is Kafka faster



Page Cache

Area in memory that caches files to improve disk i/o performance

All i/o in Linux goes through the page cache

Design

Efficiency:

- Kafka has to deal with high volumes of messages which lead to two common causes of inefficiency - too many small I/O operations,
- To avoid small I/O problem , messages are grouped together and sent .This reduces the overhead of the network round trip rather than sending a single message at a time . The server in turn appends chunks of messages to its log in one go, and the consumer fetches large linear chunks at a time.

Design

Push vs Pull:

- Kafka follows a traditional design, shared by most messaging systems, where data is pushed to the broker from the producer and pulled from the broker by the consumer.
- Advantages of pull based system for consumers:
  - The consumer wont be overwhelmed with messages when its rate of consumption falls below the production rate (a denial of service attack, in essence) as in case of push based system
  - In a pull based system kafka consumer can do aggressive batching of data and consume

Design

Consumer Position:

- Keeping track of *what* has been consumed is, surprisingly, one of the key performance points of a messaging system.
- Problem: If the broker records a message as **consumed** immediately every time it is handed out over the network, then if the consumer fails to process the message (say because it crashes or the request times out or whatever) that message will be lost.

- To solve this problem, many messaging systems add an acknowledgement feature

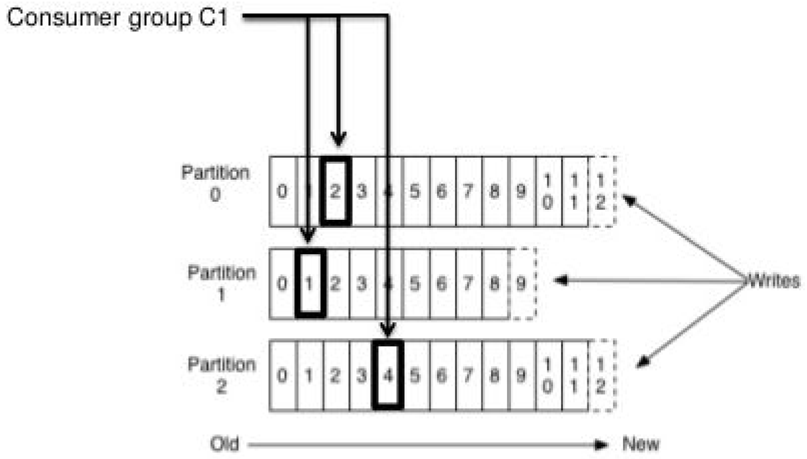
- This acknowledgement system created new problems:
  - First of all, if the consumer processes the message but fails before it can send an acknowledgement then the message will be consumed twice.
  - The second problem is around performance, now the broker must keep multiple states about every single message (first to lock it so it is not given out a second time, and then to mark it as permanently consumed so that it can be removed).

Solution:

- A topic is divided into a set of partitions, each of which is consumed by one consumer at any given time. This means that the position of a consumer in each partition is just a single integer, the offset of the next message to consume.
- This makes the state about what has been consumed very small, just one number for each partition. This state can be periodically checkpointed. This makes the equivalent of message acknowledgements very cheap.
- With this approach consumer can deliberately *rewind* back to an old offset and re-consume data. For example, if the consumer code has a bug and is discovered after some messages are consumed, the consumer can re-consume those messages once the bug is fixed.

Partition offsets

- messages in the partitions are each assigned a unique (per partition) and sequential id called the *offset.* Consumers track their pointers via *(offset, partition, topic)* tuples



Design

Message Delivery Semantics:

- *At most once*—Messages may be lost but are never redelivered.
- *At least once*—Messages are never lost but may be redelivered.
- *Exactly once*—this is what people actually want, each message is delivered once and only once.

Producer side:

- We follow **at-least once** model
- Implemented with the help of **acks**
    - =0
    - =1
    - =all

Consumer side Message Semantics

- **At-most-once:**
  - Consumer can read the messages, then save its position in the log, and finally process the messages.
  - In this case there is a possibility that the consumer process crashes after saving its position but before saving the output of its message processing.
  - In this case the process that took over processing would start at the saved position even though a few messages prior to that position had not been processed.

- **At-least-once**
  - can read the messages, process the messages, and finally save its position.
  - In this case there is a possibility that the consumer process crashes after processing messages but before saving its position.
  - In this case when the new process takes over the first few messages it receives will already have been processed.

- For **exact-once** semantics we can have the consumer store its offset in the same place as its output so that it is guaranteed that either data and offsets are both updated or neither is

- **So effectively Kafka guarantees at-least-once delivery by default and allows the user to implement at most once delivery by disabling retries on the producer and committing its offset prior to processing a batch of messages.**

Replication

- Kafka replicates the log for each topic's partitions across a configurable number of servers .
- The unit of replication is the topic partition. Under non-failure conditions, each partition in Kafka has a single leader and zero or more followers.
- The total number of replicas including the leader constitute the replication factor. **All reads and writes go to the leader of the partition**.

For Kafka node to be considered as “**in-sync**” node

- A node must be able to maintain its session with ZooKeeper
- If it is a slave it must replicate the writes happening on the leader and not fall too far behind . The determination of stuck and lagging replicas is controlled by the **replica.lag.time.max.ms** configuration.

Kafka dynamically maintains a set of in-sync replicas (ISR) that are caught-up to the leader. Only members of this set are eligible for election as leader. A write to a Kafka partition is not considered committed until *all* in-sync replicas have received the write. This ISR set is persisted to ZooKeeper whenever it changes.

What if all the brokers die

There are two behaviors that could be implemented:

- Wait for a replica in the ISR to come back to life and choose this replica as the leader (hopefully it still has all its data).
- Choose the first replica (not necessarily in the ISR) that comes back to life as the leader.

In kafka’s current release the second strategy is favoured , choosing a potentially inconsistent replica when all replicas in the ISR are dead. This is called **Unclean Leader Election**

Balancing Leadership of partitions

- Whenever a broker stops or crashes leadership for that broker's partitions transfers to other replicas. This means that by default when the broker is restarted it will only be a follower for all its partitions, meaning it will not be used for client reads and writes.
- To avoid this imbalance, Kafka has a notion of preferred replicas. If the list of replicas for a partition is 1,5,9 then node 1 is preferred as the leader to either node 5 or 9 because it is earlier in the replica list.
- bin/kafka-preferred-replica-election.sh --zookeeper zk_host:port/chroot
- Since running this command can be tedious you can also configure Kafka to do this automatically by setting the following configuration: `auto.leader.rebalance.enable=true`