Kafka Consumers

Reading data from Kafka

- You use Kafka "consumers" to write data to Kafka brokers.
  - Available for JVM (Java, Scala), C/C++, Python, Ruby, etc.
  - The Kafka project only provides the JVM implementation.
    - Has risk that a new Kafka release will break non-JVM clients.
- Examples will be shown later in the "Example Kafka apps" section.
- Three API options for JVM users:
  1. High-level consumer API <<< in most cases you want to use this one!
  2. Simple consumer API
  3. Hadoop consumer API
- Most noteworthy: The "simple" API is anything but simple. ☺
  - Prefer to use the high-level consumer API if it meets your needs (it should).
  - Counter-example: Kafka spout in Storm 0.9.2 uses simple consumer API to integrate well with Storm's model of guaranteed message processing.

Reading data from Kafka

- Consumers *pull* from Kafka (there's no push)
  - Allows consumers to control their pace of consumption.
  - Allows to design downstream apps for **average** load, not peak load (cf. Loggly talk)

- Consumers are responsible to track their read positions aka "offsets"
  - High-level consumer API: takes care of this for you, stores offsets in ZooKeeper
  - Simple consumer API: nothing provided, it's totally up to you
  - What does this offset management allow you to do?
    - Consumers can deliberately rewind "in time" (up to the point where Kafka prunes), e.g. to replay older messages.
      - Cf. Kafka spout in Storm 0.9.2.
    - Consumers can decide to only read a specific subset of partitions for a given topic.
      - Cf. Loggly's setup of (down)sampling a production Kafka topic to a manageable volume for testing
    - Run offline, batch ingestion tools that write (say) from Kafka to Hadoop HDFS every hour.
      - Cf. LinkedIn Camus, Pinterest Secor

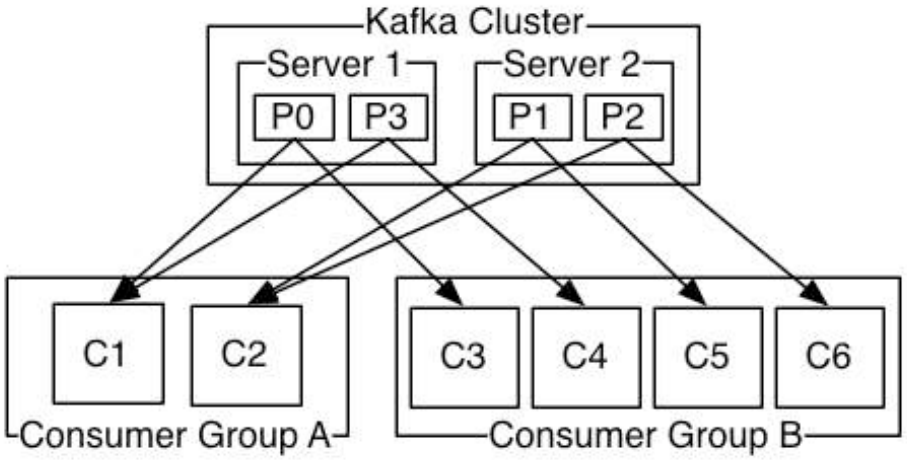
Reading data from Kafka

- Important consumer configuration settings

`group.id`	assigns an individual consumer to a “group”
`zookeeper.connect`	to discover brokers/topics/etc., and to store consumer state (e.g. when using the high-level consumer API)
`fetch.message.max.bytes`	number of message bytes to (attempt to) fetch for each partition; must be >= broker’s `message.max.bytes`

Reading data from Kafka

- Consumer “groups”
  - Allows multi-threaded and/or multi-machine consumption from Kafka topics.
  - Consumers “join” a group by using the same `group.id`
  - Kafka guarantees a message is only ever read by a single consumer in a group.
    - Kafka assigns the partitions of a topic to the consumers in a group so that each partition is consumed by exactly one consumer in the group.
    - Maximum parallelism of a consumer group: **#consumers** (in the group) <= **#partitions**

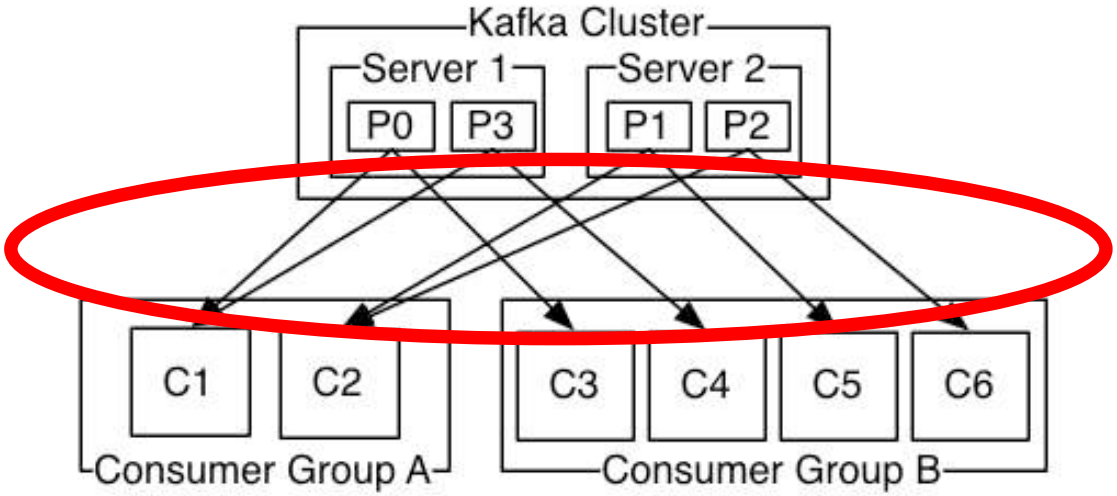


Guarantees when reading data from Kafka

- A message is only ever read by a single consumer in a group.
- A consumer sees messages in the order they were stored in the log.
- The order of messages is only guaranteed within a partition.
  - No order guarantee across partitions, which includes no order guarantee per-topic.
  - If total order (per topic) is required you can consider, for instance:
    - Use #partition = 1. Good: total order. Bad: Only 1 consumer process at a time.
    - “Add” total ordering in your consumer application, e.g. a Storm topology.
- Some gotchas:
  - If you have multiple partitions per thread there is NO guarantee about the order you receive messages, other than that within the partition the offsets will be sequential.
    - Example: You may receive 5 messages from partition 10 and 6 from partition 11, then 5 more from partition 10 followed by 5 more from partition 10, even if partition 11 has data available.
  - Adding more processes/threads will cause Kafka to rebalance, possibly changing the assignment of a partition to a thread (whoops).

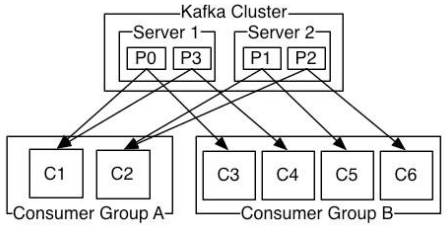
Rebalancing: how consumers meet brokers

- Remember?



- The assignment of brokers – via the partitions of a topic – to consumers is quite **important**, and it is **dynamic** at run-time.

Rebalancing: how consumers meet brokers



- Why “dynamic at run-time”?
  - Machines can die, be added, …
  - Consumer apps may die, be re-configured, added, …
- Whenever this happens a **rebalancing** occurs.
  - Rebalancing is a normal and expected lifecycle event in Kafka.
  - But it’s also a nice way to shoot yourself or Ops in the foot.
- Why is this important?
  - Most Ops issues are due to 1) **rebalancing** and 2) **consumer lag**.
  - So Dev + Ops must understand what goes on.

Rebalancing: how consumers meet brokers

- **Rebalancing?**
  - Consumers in a group come into consensus on which consumer is consuming which partitions → required for distributed consumption
  - Divides broker partitions evenly across consumers, tries to reduce the number of broker nodes each consumer has to connect to
- When does it happen? Each time:
  - a **consumer** joins or leaves a consumer group, OR
  - a **broker** joins or leaves, OR
  - a topic "joins/leaves" via a filter, cf. `createMessageStreamsByFilter()`
- Examples:
  - If a consumer or broker fails to heartbeat to ZK → rebalance!
  - `createMessageStreams()` registers consumers for a topic, which results in a rebalance of the consumer-broker assignment.