Apache Kafka 0.8 basic training

Michael G. Noll, Verisign

mnoll@verisign.com / @miguno

July 2014

Update 2015-08-01:

Shameless plug! Since publishing this Kafka training deck about a year ago I joined Confluent Inc. as their Developer Evangelist.

Confluent is the US startup founded in 2014 by the creators of Apache Kafka who developed Kafka while at LinkedIn (see Forbes about Confluent). Next to building the world's best stream data platform we are also providing professional Kafka trainings, which go even deeper as well as beyond my extensive training deck below.

http://www.confluent.io/training

I can say with confidence that these are the best and most effective Apache Kafka trainings available on the market. But you don't have to take my word for it – feel free to take a look yourself and reach out to us if you're interested.

—Michael

Kafka?

- **Part 1: Introducing Kafka**
  - "Why should I stay awake for the full duration of this workshop?"
- **Part 2: Kafka core concepts**
  - Topics, partitions, replicas, producers, consumers, brokers
- **Part 3: Operating Kafka**
  - Architecture, hardware specs, deploying, monitoring, P&S tuning
- **Part 4: Developing Kafka apps**
  - Writing to Kafka, reading from Kafka, testing, serialization, compression, example apps
- **Part 5: Playing with Kafka using Wirbelsturm**
- **Wrapping up**

Part 1: Introducing Kafka

Overview of Part 1: Introducing Kafka

- Kafka?

- Kafka adoption and use cases in the wild

  - At LinkedIn

  - At other companies

- How fast is Kafka, and why?

- Kafka + X for processing

  - Storm, Samza, Spark Streaming, custom apps

Kafka?

- http://kafka.apache.org/

- Originated at LinkedIn, open sourced in early 2011

- Implemented in Scala, some Java

- 9 core committers, plus ~ 20 contributors

https://kafka.apache.org/committers.html
https://github.com/apache/kafka/graphs/contributors

Kafka?

- LinkedIn's motivation for Kafka was:
  - "A unified platform for handling all the real-time data feeds a large company might have."

- Must haves
  - High throughput to support **high volume event feeds**.
  - Support real-time processing of these feeds to create **new, derived feeds**.
  - Support large data backlogs to handle periodic ingestion from **offline systems**.
  - Support low-latency delivery to handle more traditional **messaging use cases**.
  - Guarantee **fault-tolerance** in the presence of machine failures.

http://kafka.apache.org/documentation.html#majordesignelements

Kafka @ LinkedIn, 2014

Kafka At LinkedIn

- 300+ Kafka brokers
- Over 18,000 topics
- 140,000+ Partitions

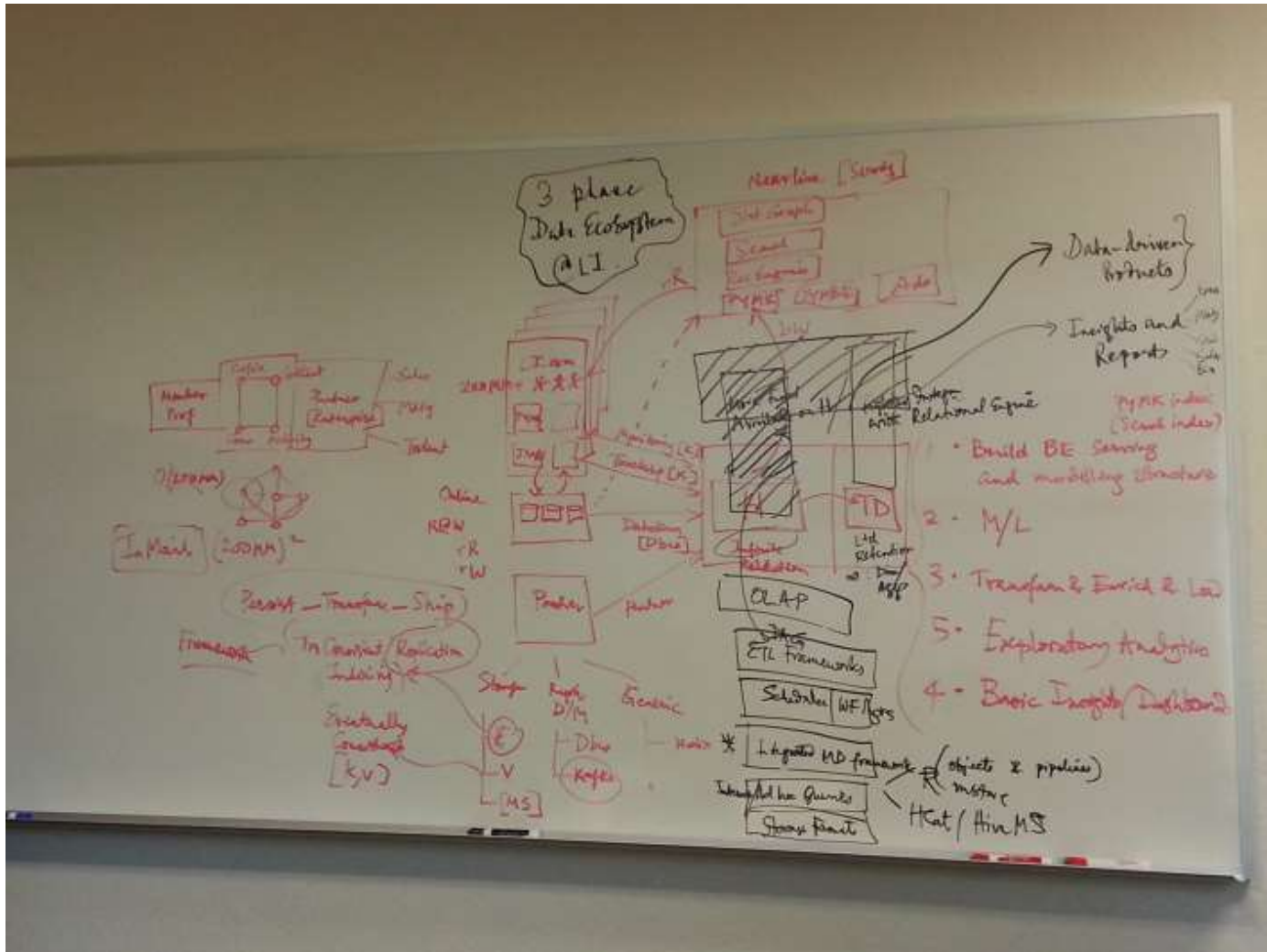- 220 Billion messages per day
- 40 Terabytes In
- 160 Terabytes Out

- Peak Load
  - 3.25 Million messages per second
  - 5.5 Gigabits/sec Inbound
  - 18 Gigabits/sec Outbound

(Numbers have increased since.)

https://twitter.com/SalesforceEng/status/466033231800713216/photo/1
http://www.hakkalabs.co/articles/site-reliability-engineering-linkedin-kafka-service

Data architecture @ LinkedIn, Feb 2013

http://gigaom.com/2013/12/09/netflix-open-sources-its-data-traffic-cop-suro/

Kafka @ LinkedIn, 2014

- Multiple data centers, multiple clusters
  - Mirroring between clusters / data centers

- What type of data is being transported through Kafka?
  - **Metrics**: operational telemetry data
  - **Tracking**: everything a LinkedIn.com user does
  - **Queuing**: between LinkedIn apps, e.g. for sending emails

- To transport data from LinkedIn's apps to Hadoop, and back
- In total **~ 200 billion events/day** via Kafka
  - Tens of thousands of data producers, thousands of consumers
  - 7 million events/sec (write), 35 million events/sec (read)   <<< may include replicated events
  - But: LinkedIn is not even the largest Kafka user anymore as of 2014

http://www.hakkalabs.co/articles/site-reliability-engineering-linkedin-kafka-service
http://www.slideshare.net/JayKreps1/i-32858698
http://search-hadoop.com/m/4TaT4qAFQW1

Kafka @ LinkedIn, 2014

"For reference, here are the stats on one of
 LinkedIn's busiest clusters (at peak):

        15 brokers
     15,500 partitions (replication factor 2)
   400,000 msg/s inbound
     70 MB/s inbound
    400 MB/s outbound"

https://kafka.apache.org/documentation.html#java

Staffing: Kafka team @ LinkedIn

- Team of **8+** engineers
    - Site reliability engineers (Ops): at least **3**
    - Developers: at least **5**
- SRE's as well as DEV's are on call 24x7

https://kafka.apache.org/committers.html
http://www.hakkalabs.co/articles/site-reliability-engineering-linkedin-kafka-service

Kafka adoption and use cases

- **LinkedIn:** activity streams, operational metrics, data bus
  - 400 nodes, 18k topics, 220B msg/day (peak 3.2M msg/s), May 2014
- **Netflix**: real-time monitoring and event processing
- **Twitter**: as part of their Storm real-time data pipelines
- **Spotify**: log delivery (from 4h down to 10s), Hadoop
- **Loggly**: log collection and processing
- **Mozilla**: telemetry data
- Airbnb, Cisco, Gnip, InfoChimps, Ooyala, Square, Uber, …

https://cwiki.apache.org/confluence/display/KAFKA/Powered+By

powered by **VERISIGN**

Kafka @ Spotify

How fast is Kafka?

- **"Up to 2 million writes/sec on 3 cheap machines"**
  - Using 3 producers on 3 different machines, 3x async replication
    - Only 1 producer/machine because NIC already saturated
- **Sustained throughput as stored data grows**
  - Slightly different test config than 2M writes/sec above.

Throughput vs Size
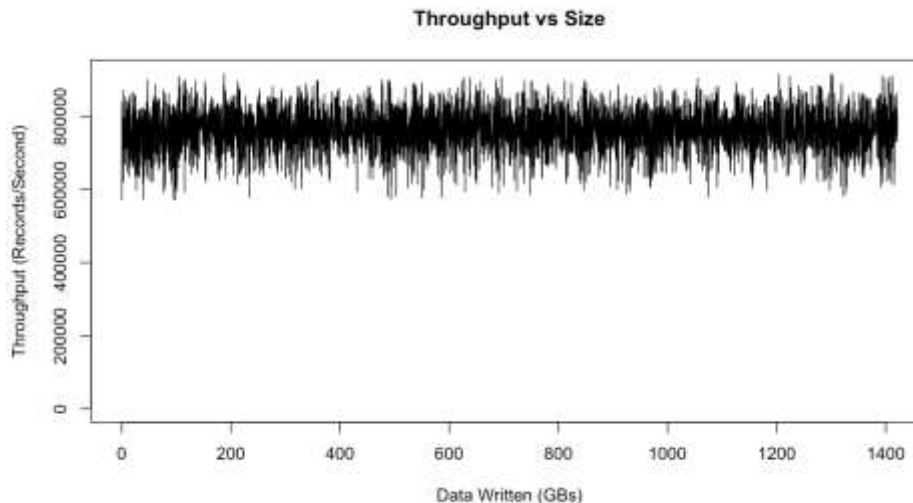
- Test setup
  - Kafka trunk as of April 2013, but 0.8.1+ should be similar.
  - 3 machines: 6-core Intel Xeon 2.5 GHz, 32GB RAM, 6x 7200rpm SATA, 1GigE

  http://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines

Why is Kafka so fast?

- Fast **writes**:

  - While Kafka persists all data to disk, essentially all writes go to the **page cache** of OS, i.e. RAM.

  - Cf. hardware specs and OS tuning (we cover this later)

- Fast **reads**:

  - Very efficient to transfer data from page cache to a network **socket**

  - Linux: `sendfile()` system call

- Combination of the two = fast Kafka!

  - Example (Operations): On a Kafka cluster where the consumers are mostly caught up you will see no read activity on the disks as they will be serving data entirely from cache.

http://kafka.apache.org/documentation.html#persistence

Why is Kafka so fast?

- Example: Loggly.com, who run Kafka & Co. on Amazon AWS

  - "99.99999% of the time our data is coming from disk cache and RAM; only very rarely do we hit the disk."

  - "One of our consumer groups (8 threads) which maps a log to a customer can process about 200,000 events per second draining from 192 partitions spread across 3 brokers."

    - Brokers run on m2.xlarge Amazon EC2 instances backed by provisioned IOPS

http://www.developer-tech.com/news/2014/jun/10/why-loggly-loves-apache-kafka-how-unbreakable-infinitely-scalable-messaging-makes-log-management-better/

powered by **VERISIGN**

Kafka + X for processing the data?

- Kafka + **Storm** often used in combination, e.g. Twitter

- Kafka + **custom**
  - "Normal" Java multi-threaded setups
  - Akka actors with Scala or Java, e.g. Ooyala

- Recent additions:
  - **Samza** (since Aug '13) – also by LinkedIn
  - **Spark Streaming**, part of Spark (since Feb '13)

- Kafka + **Camus** for Kafka->Hadoop ingestion

https://cwiki.apache.org/confluence/display/KAFKA/Powered+By

Part 2: Kafka core concepts

Overview of Part 2: Kafka core concepts

- A first look

- Topics, partitions, replicas, offsets

- Producers, brokers, consumers

- Putting it all together

A first look

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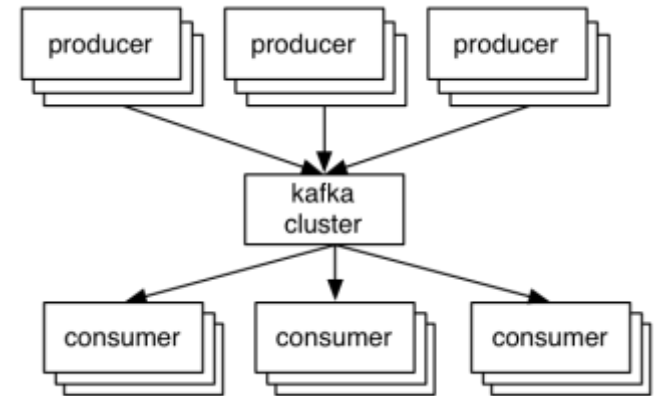
- The data
  - Data is stored in **topics**.
  - **Topics** are split into **partitions**, which are **replicated**.

A first look

http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/
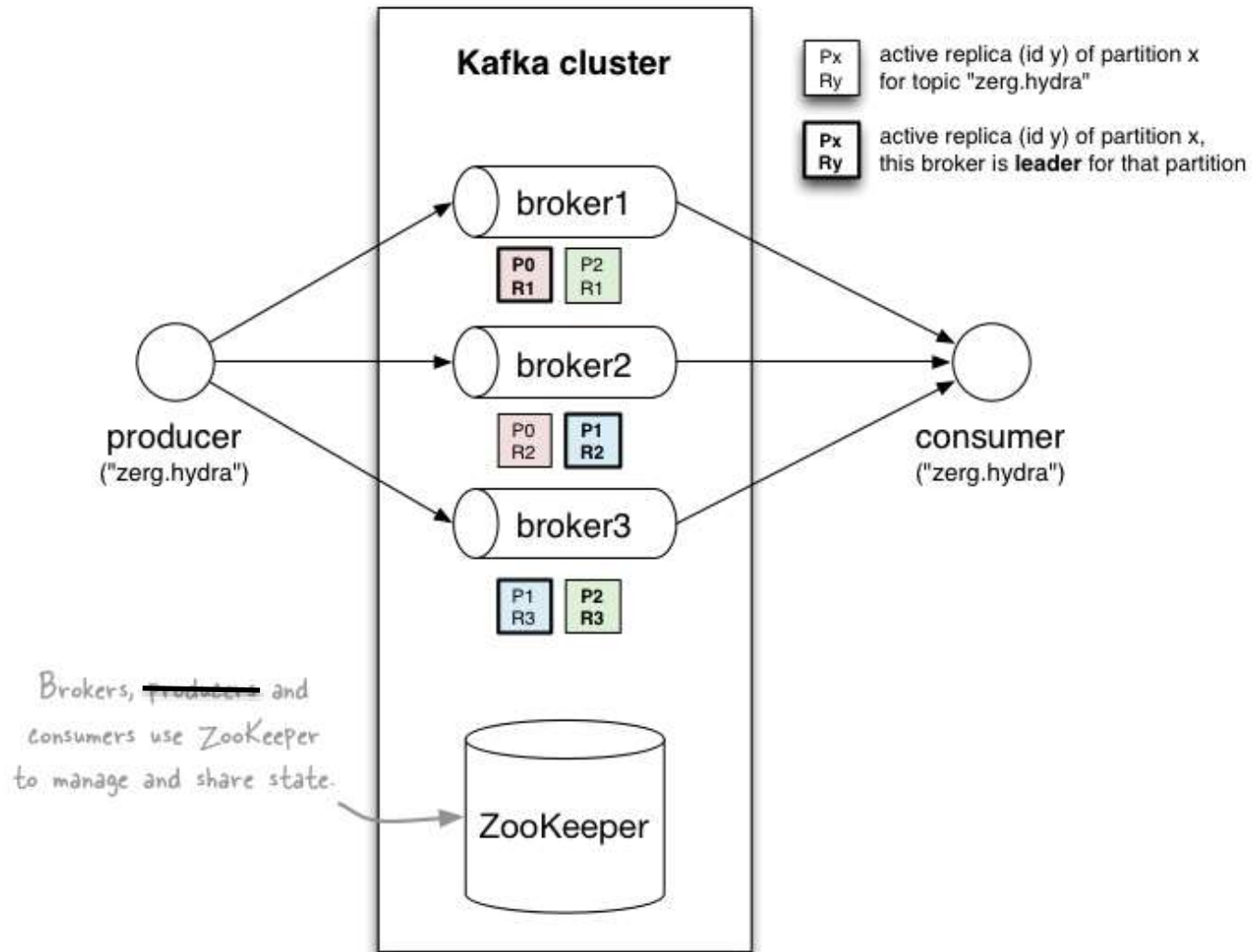
Topics

- **Topic:** feed name to which messages are published
  - Example: "zerg.hydra"

Kafka prunes "head" based on age or max size or "key"

Kafka topic

... Older msgs   Newer msgs

ne w

Producer A1

Producer A2
…
Producer An

Producers always append to "tail"
(think: append to a file)

Broker(s)

Topics

Consumer group C1

Consumer group C2

Consumers use an "offset pointer" to track/control their read progress (and decide the pace of consumption)

Producer A1

Producer A2

…

Producer A*n*

… | | | | | | | | | ne w |

Older msgs   Newer msgs

Producers always append to "tail" (think: append to a file)

Broker(s)

Topics

- Creating a topic
  - CLI

```
$ kafka-topics.sh --zookeeper zookeeper1:2181 --create --topic zerg.hydra \
    --partitions 3 --replication-factor 2 \
    --config x=y
```

  - API
    https://github.com/miguno/kafka-storm-starter/blob/develop/src/main/scala/com/miguno/kafkastorm/storm/KafkaStormDemo.scala
  - Auto-create via `auto.create.topics.enable = true`

- Modifying a topic
  - https://kafka.apache.org/documentation.html#basic_ops_modify_topic
- Deleting a topic: DON'T in 0.8.1.x!

Partitions

- A topic consists of **partitions.**

- Partition: **ordered + immutable** sequence of messages that is continually appended to

Anatomy of a Topic

Partition 0: 0 1 2 3 4 5 6 7 8 9 10 11 12

Partition 1: 0 1 2 3 4 5 6 7 8 9

Partition 2: 0 1 2 3 4 5 6 7 8 9 10 11 12

Writes

Old ———————→ New
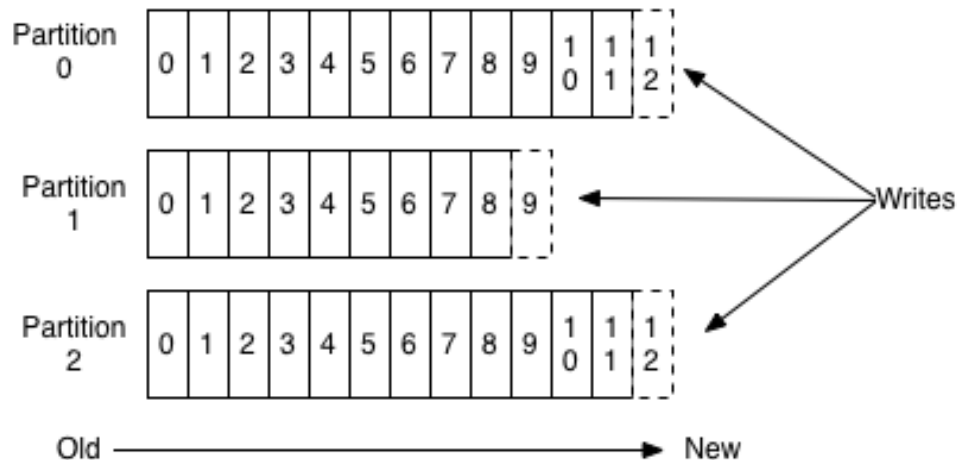
Partitions

- #partitions of a topic is configurable
- #partitions determines **max** consumer (group) parallelism
  - Cf. parallelism of Storm's KafkaSpout via `builder.setSpout(,,N)`
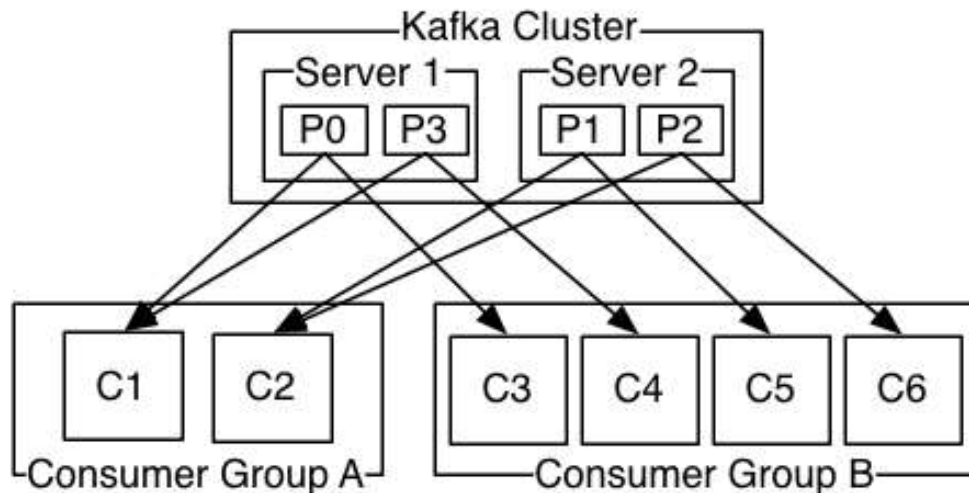
- Consumer group A, with 2 consumers, reads from a 4-partition topic
- Consumer group B, with 4 consumers, reads from the same topic

Partition offsets

- **Offset**: 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
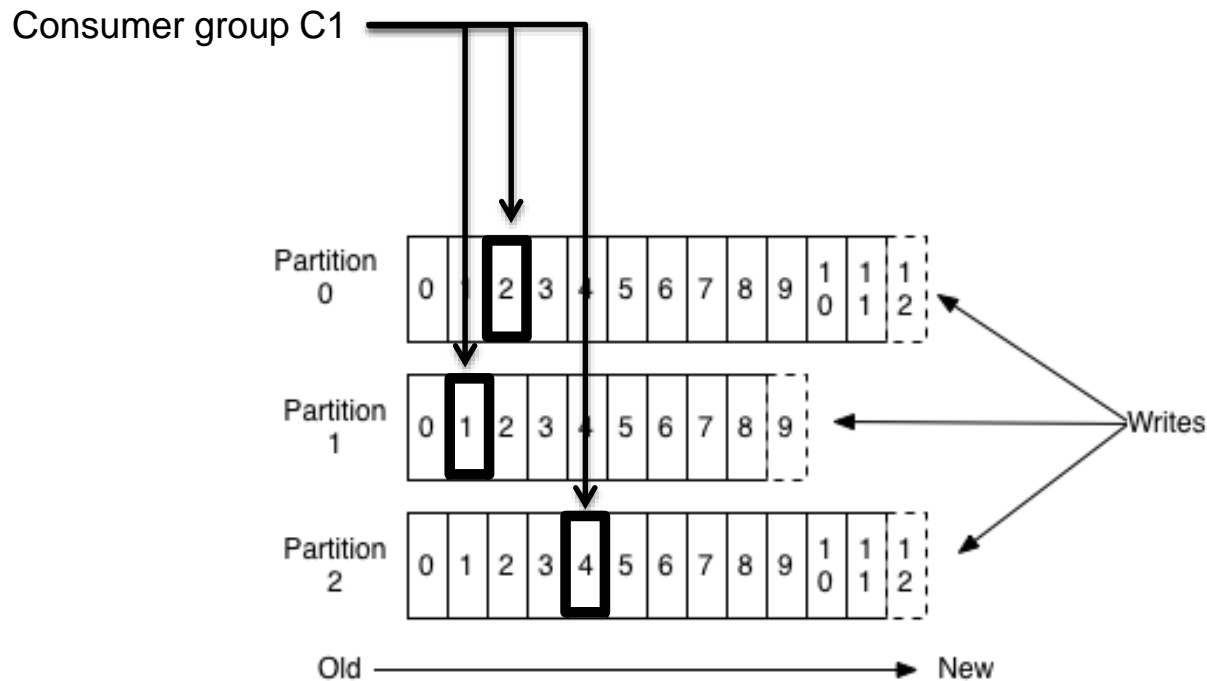
Replicas of a partition

- **Replicas:** "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
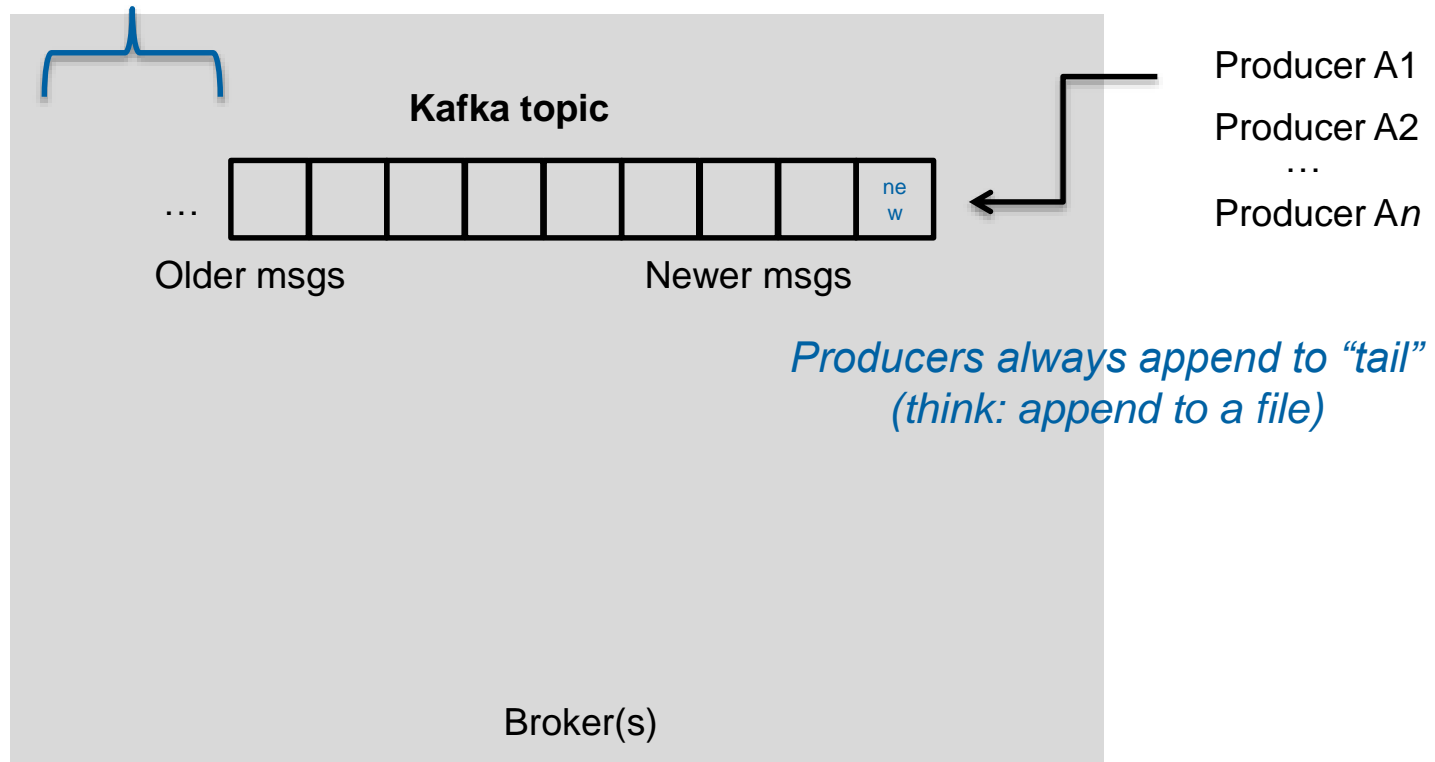   - LinkedIn: numReplicas == 2 → 1 broker can die

Topics vs. Partitions vs. Replicas

http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/
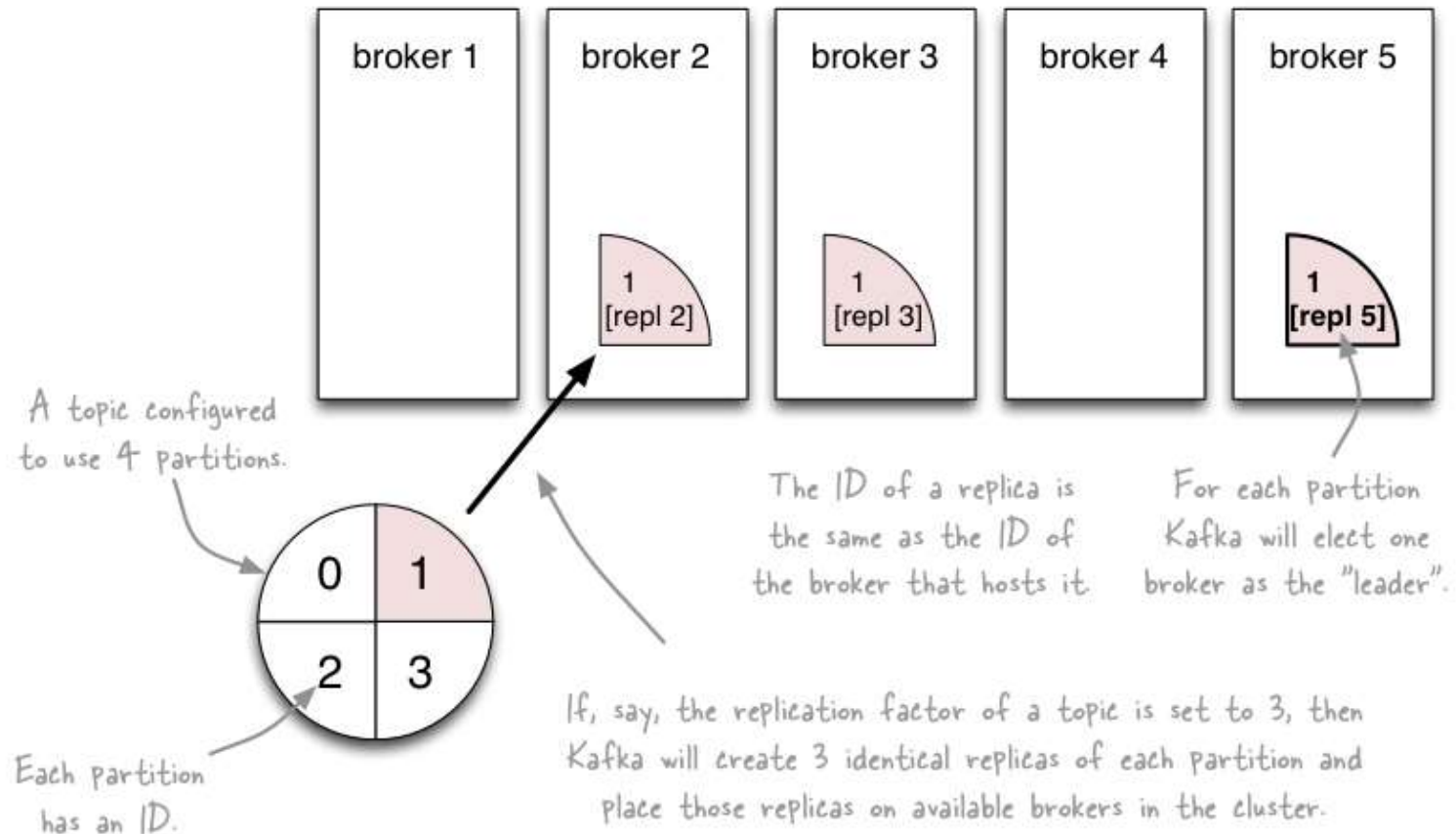
Inspecting the current state of a topic
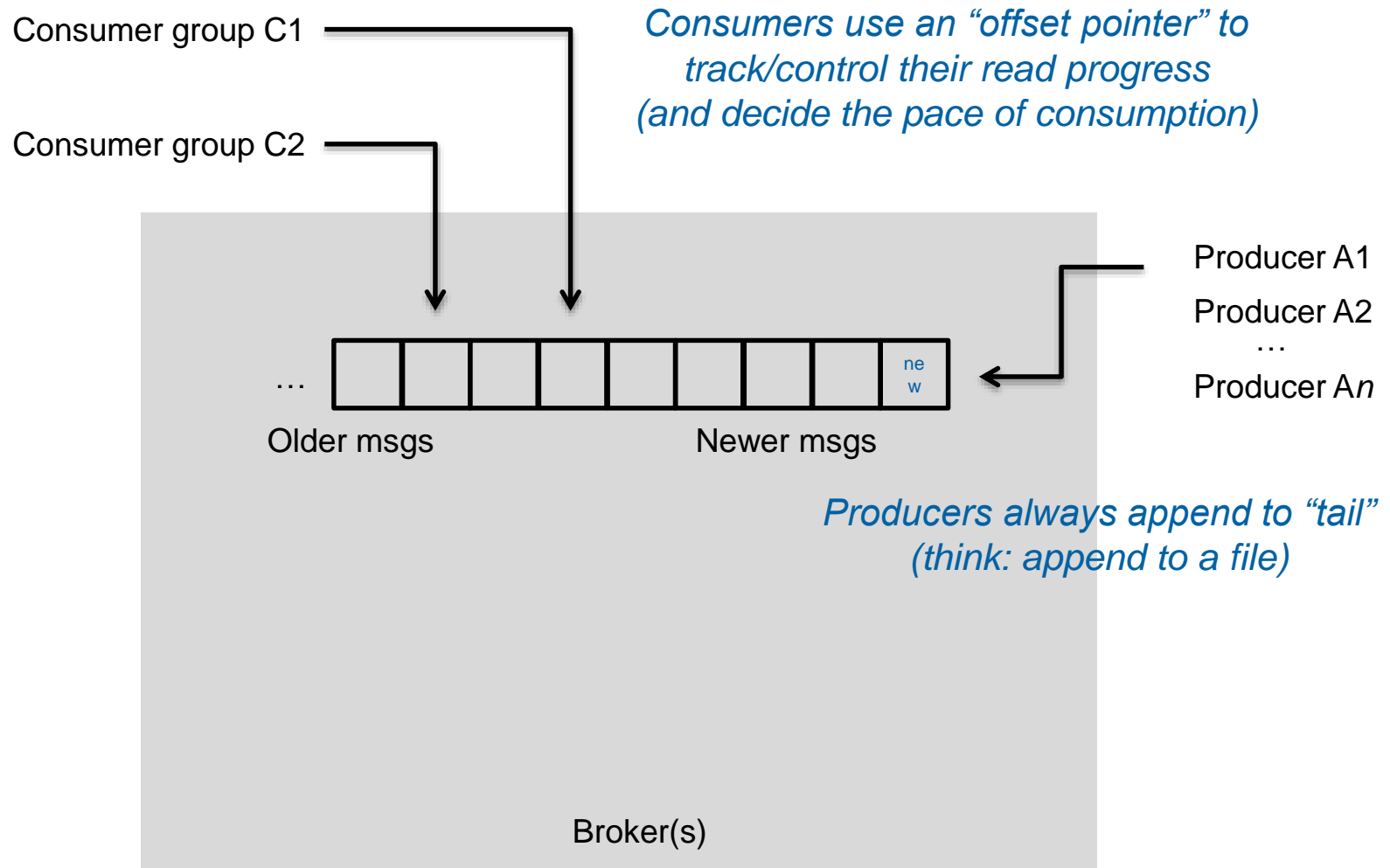
- `--describe` the topic

```
$ kafka-topics.sh --zookeeper zookeeper1:2181 --describe --topic zerg.hydra
Topic:zerg2.hydra PartitionCount:3 ReplicationFactor:2 Configs:
  Topic: zerg2.hydra Partition: 0 Leader: 1 Replicas: 1,0 Isr: 1,0
  Topic: zerg2.hydra Partition: 1 Leader: 0 Replicas: 0,1 Isr: 0,1
  Topic: zerg2.hydra Partition: 2 Leader: 1 Replicas: 1,0 Isr: 1,0
```

- Leader: brokerID of the currently elected leader broker
    - Replica ID's = broker ID's
  - ISR = "in-sync replica", replicas that are in sync with the leader

- In this example:
  - Broker 0 is leader for partition 1.
  - Broker 1 is leader for partitions 0 and 2.
  - All replicas are in-sync with their respective leader partitions.

Let's recap

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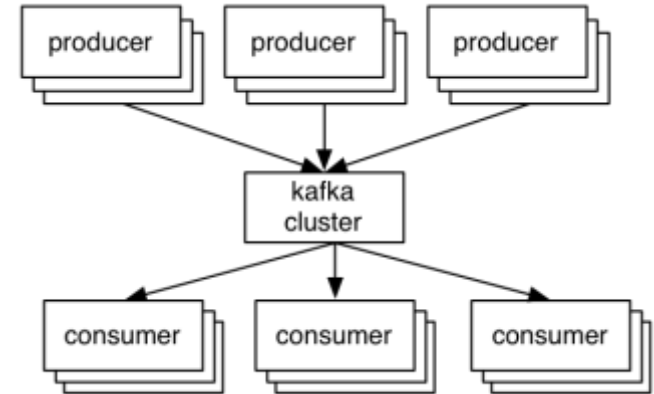
- The data
  - Data is stored in **topics**.
  - **Topics** are split into **partitions** which are **replicated**.

Putting it all together
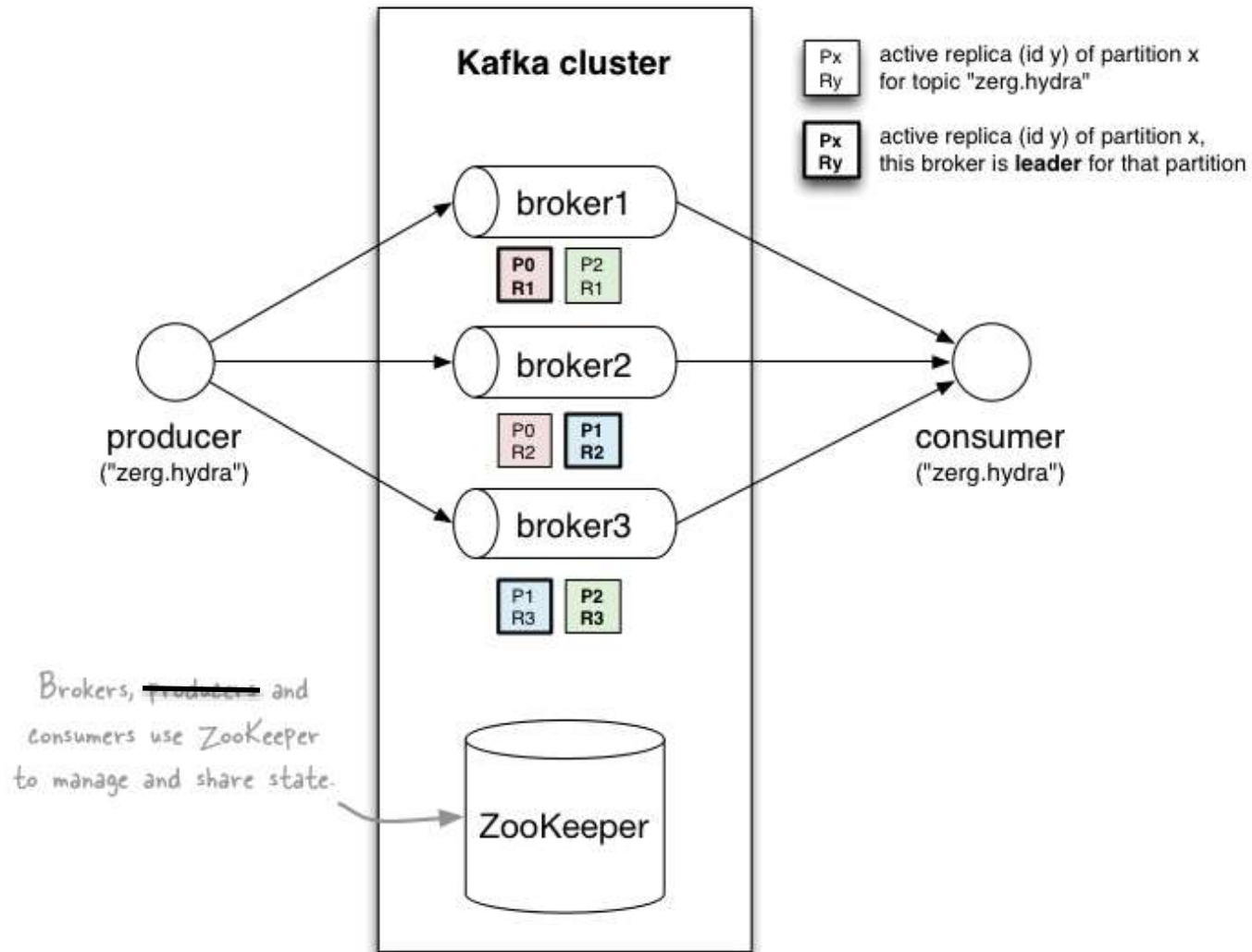
Side note (opinion)

- Drawing a conceptual line from Kafka to Clojure's `core.async`

> **Michael G. Noll**
> @miguno
>
> Clojure core.async and its 'channels' are cool
> for the same reasons that Kafka is for
> distributed systems. infoq.com/presentations/
> … (go 31:54)
>
> ↩ Reply   🗑 Delete   ★ Favorite   ••• More

- Cf. talk "Clojure core.async Channels", by Rich Hickey, at ~ 31m54
  http://www.infoq.com/presentations/clojure-core-async

Part 3: Operating Kafka

Overview of Part 3: Operating Kafka

- Kafka architecture

- Kafka hardware specs

- Deploying Kafka

- Monitoring Kafka

  - Kafka apps

  - Kafka itself

  - ZooKeeper

  - "Auditing" Kafka (not: security audit)

- P&S tuning

- Ops-related Kafka references

Kafka architecture

- ## Kafka brokers

  - You can run clusters with 1+ brokers.

  - Each broker in a cluster must have
    a unique `broker.id`.
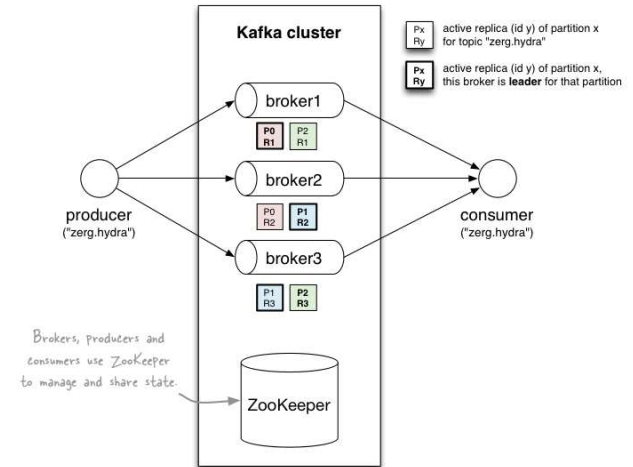
Kafka architecture

- Kafka requires **ZooKeeper**
  - LinkedIn runs (old) ZK 3.3.4, but latest 3.4.5 works, too.

- ZooKeeper
  - v0.8: used by **brokers** and **consumers**, but not by producers.
    - Brokers: general state information, leader election, etc.
    - Consumers: primarily for tracking message offsets (cf. later)
  - v0.9: used by **brokers only**
    - Consumers will use special Kafka topics instead of ZooKeeper
      - Will substantially reduce the load on ZooKeeper for large deployments
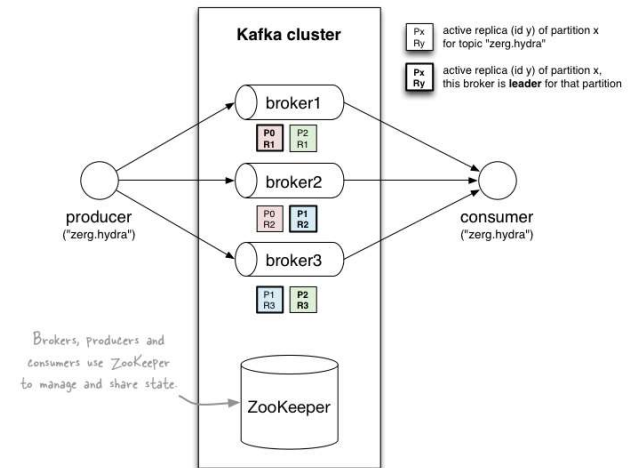
Kafka broker hardware specs @ LinkedIn

- Solely dedicated to running Kafka, run nothing else.

  - 1 Kafka broker instance per machine

- 2x 4-core Intel Xeon (info outdated?)

- **64 GB RAM (up from 24 GB)**

  - Only 4 GB used for Kafka broker, remaining 60 GB for page cache

  - Page cache is what makes Kafka **fast**

- **RAID10 with 14 spindles**

  - More spindles = higher disk throughput

  - Cache on RAID, with battery backup

  - Before H/W upgrade: 8x SATA drives (7200rpm), not sure about RAID

- 1 GigE (?) NICs

- EC2 example: m2.2xlarge @ $0.34/hour, with provisioned IOPS

ZooKeeper hardware specs @ LinkedIn

- ZooKeeper servers
  - Solely dedicated to running ZooKeeper, run nothing else.
    - 1 ZooKeeper instance per machine
  - SSD's dramatically improve performance
    - In v0.8.x, brokers and consumers must talk to ZK. In large-scale environments (many consumers, many topics and partitions) this means ZK can become a bottleneck because it processes requests *serially*. And this processing depends primarily on I/O performance.
  - 1 GigE (?) NICs

- ZooKeeper in LinkedIn's architecture
  - 5-node ZK ensembles = tolerates 2 dead nodes
  - 1 ZK ensemble for all Kafka clusters within a data center
    - LinkedIn runs multiple data centers, with multiple Kafka clusters

Deploying Kafka

- Puppet module

  - https://github.com/miguno/puppet-kafka

  - Hiera-compatible, rspec tests, Travis CI setup (e.g. to test against multiple versions of Puppet and Ruby, Puppet style checker/lint, etc.)

- RPM packaging script for RHEL 6

  - https://github.com/miguno/wirbelsturm-rpm-kafka

  - Digitally signed by yum@michael-noll.com

  - RPM is built on a Wirbelsturm-managed build server

- Public (Wirbelsturm) S3-backed yum repo

  - https://s3.amazonaws.com/yum.miguno.com/bigdata/
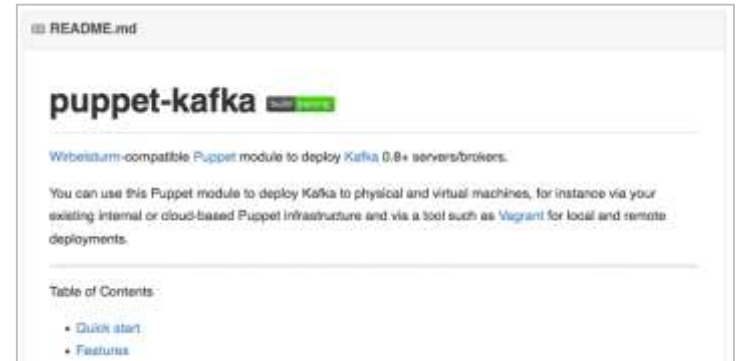
Deploying Kafka

- Hiera example

```
---
classes:
  - kafka::service
  - supervisor

## Kafka
kafka::broker_id: 0
kafka::config_map:
  log.roll.hours: 48
  log.retention.hours: 48
kafka::kafka_heap_opts: '-Xms2G -Xmx2G -XX:NewSize=256m -XX:MaxNewSize=256m'
kafka::kafka_opts: '-XX:CMSInitiatingOccupancyFraction=70 -XX:+PrintTenuringDistribution'
kafka::zookeeper_connect:
  - 'zookeeper1:2181'

# Optional: Manage /etc/security/limits.conf to tune the maximum number
# of open files, which is a typical setting you must change for Kafka
# production environments.  Default: false (do not manage)
kafka::limits_manage: true
kafka::limits_nofile: 65536
```

Operating Kafka

- Typical operations tasks include:
  - Adding or removing brokers
    - Example: ensure a newly added broker actually receives data, which requires moving partitions from existing brokers to the new broker
    - Kafka provides helper scripts (cf. below) but still manual work involved
  - Balancing data/partitions to ensure best performance
  - Add new topics, re-configure topics
    - Example: Increasing #partitions of a topic to increase max parallelism
  - Apps management: new producers, new consumers

- See Ops-related references at the end of this part

Lessons learned from operating Kafka at LinkedIn

- Biggest challenge has been to **manage hyper growth**
  - Growth of Kafka adoption: more producers, more consumers, …
  - Growth of data: more LinkedIn.com users, more user activity, …

- Typical tasks at LinkedIn
  - Educating and coaching Kafka users.
  - Expanding Kafka clusters, shrinking clusters.
  - Monitoring consumer apps – "Hey, my stuff stopped. Kafka's fault!"

http://www.hakkalabs.co/articles/site-reliability-engineering-linkedin-kafka-service
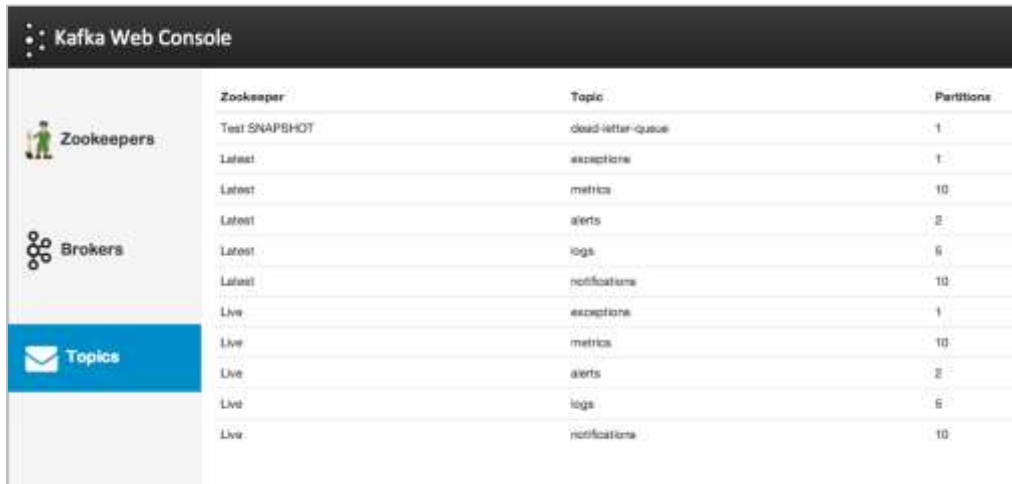
Kafka security

- Original design was not created with security in mind.

- Discussion started in June 2014 to add security features.

    - Covers transport layer security, data encryption at rest, non-repudiation, A&A, …

    - See [DISCUSS] Kafka Security Specific Features

- At the moment there's basically no security built-in.

Monitoring Kafka

Monitoring Kafka

- Nothing fancy built into Kafka (e.g. no UI) but see:
    - https://cwiki.apache.org/confluence/display/KAFKA/System+Tools
    - https://cwiki.apache.org/confluence/display/KAFKA/Ecosystem

Kafka Web Console

Kafka Offset Monitor

Monitoring Kafka

- Use of standard monitoring tools recommended

  - Graphite

    - Puppet module: https://github.com/miguno/puppet-graphite

    - Java API, also used by Kafka: http://metrics.codahale.com/

  - JMX

    - https://kafka.apache.org/documentation.html#monitoring

- Collect logging files into a central place

  - Logstash/Kibana and friends

  - Helps with troubleshooting, debugging, etc. – notably if you can correlate logging data with numeric metrics

Monitoring Kafka apps

- **Almost all problems are due to:**

  1. Consumer lag

  2. Rebalancing   <<< we cover this later in part 4

powered by **VERISIGN**

Monitoring Kafka apps: consumer lag

Lag = how far your consumer is behind the producers

Consumer group C1

Producer A1

Producer A2

...

Producer A*n*

new

... Older msgs   Newer msgs

Broker(s)
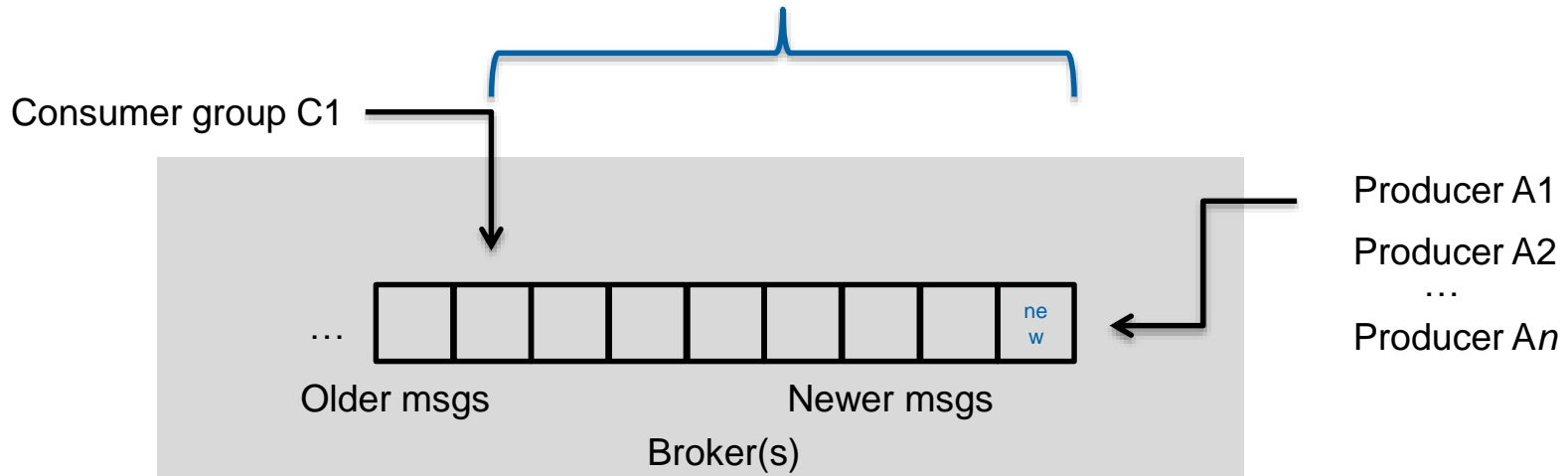
- **Lag** is a consumer problem

  - Too slow, too much GC, losing connection to ZK or Kafka, ...

  - Bug or design flaw in consumer

  - Operational mistakes: e.g. you brought up 6 servers in parallel, each one in turn triggering rebalancing, then hit Kafka's rebalance limit; cf. `rebalance.max.retries` (default: 4) & friends

Monitoring Kafka itself (1 of 3)

- **Under-replicated partitions**
  - For example, because a broker is down.
  - Means cluster runs in degraded state.
    - FYI: LinkedIn runs with replication factor of 2 => 1 broker can die.

- Offline partitions
  - Even worse than under-replicated partitions!
  - *Serious* problem (data loss) if anything but 0 offline partitions.

Monitoring Kafka itself (1 of 3)

- Data size on disk

  - Should be balanced across disks/brokers

  - Data balance even more important than partition balance

  - FYI: New script in v0.8.1 to balance data/partitions across brokers

- Broker partition balance

  - Count of partitions should be balanced evenly across brokers

  - See new script above.

Monitoring Kafka itself (1 of 3)

- Leader partition count
  - Should be balanced across brokers so that each broker gets the same amount of load
  - Only 1 broker is ever the leader of a given partition, and only this broker is going to talk to producers + consumers for that partition
    - Non-leader replicas are used solely as safeguards against data loss
  - Feature in v0.8.1 to auto-rebalance the leaders and partitions in case a broker dies, but it does not work that well yet (SRE's still have to do this manually at this point).

- Network utilization
  - Maxed network one reason for under-replicated partitions
  - LinkedIn don't run anything but Kafka on the brokers, so network max is due to Kafka. Hence, when they max the network, they need to add more capacity across the board.

Monitoring ZooKeeper

- Ensemble (= cluster) availability

  - LinkedIn run 5-node ensembles = tolerates 2 dead

  - Twitter run 13-node ensembles = tolerates 6 dead

- Latency of requests

  - Metric target is 0 ms when using SSD's in ZooKeeper machines.

    - Why? Because SSD's are so fast they typically bring down latency below ZK's metric granularity (which is per-ms).

- Outstanding requests

  - Metric target is 0.

  - Why? Because ZK processes all incoming requests serially. Non-zero values mean that requests are backing up.

"Auditing" Kafka

LinkedIn's way to detect data loss etc.

"Auditing" Kafka

- LinkedIn's way to detect data loss etc. in Kafka
  - Not part of open source stack yet. May come in the future.
  - In short: custom producer+consumer app that is hooked into monitoring.

- Value proposition
  - Monitor whether you're losing messages/data.
  - Monitor whether your pipelines can handle the incoming data load.

http://www.hakkalabs.co/articles/site-reliability-engineering-linkedin-kafka-service

powered by **VERISIGN**

LinkedIn's Audit UI: a first look

- Example 1: Count discrepancy
  - Caused by messages failing to reach a downstream Kafka cluster

- Example 2: Load lag

"Auditing" Kafka

- Every **producer** is also writing messages into a special topic about how many messages it produced, every 10mins.

  - Example: "Over the last 10mins, I sent N messages to topic X."

  - This metadata gets mirrored like any other Kafka data.

- **Audit consumer**

  - 1 audit consumer per Kafka cluster

  - Reads *every* single message out of "its" Kafka cluster. It then calculates counts for each topic, and writes those counts back into the same special topic, every 10mins.

    - Example: "I saw M messages in the last 10mins for topic X in THIS cluster"

  - And the next audit consumer in the next, downstream cluster does the same thing.

"Auditing" Kafka

- Monitoring audit consumers
  - Completeness check
    - "#msgs according to producer == #msgs seen by audit consumer?"
  - Lag
    - "Can the audit consumers keep up with the incoming data rate?"
    - If audit consumers fall behind, then all your tracking data falls behind as well, and you don't know how many messages got produced.

"Auditing" Kafka

- Audit UI
    - Only reads data from that special "metrics/monitoring" topic, but this data is reads from every Kafka cluster at LinkedIn.
        - What they producers said they wrote in.
        - What the audit consumers said they saw.
    - Shows correlation graphs (producers vs. audit consumers)
        - For each tier, it shows how many messages there were in each topic over any given period of time.
        - Percentage of how much data got through (from cluster to cluster).
        - If the percentage drops below 100%, then emails are sent to Kafka SRE+DEV as well as their Hadoop ETL team because that stops the Hadoop pipelines from functioning properly.

LinkedIn's Audit UI: a closing look

- Example 1: Count discrepancy
  - Caused by messages failing to reach a downstream Kafka cluster

- Example 2: Load lag

Kafka performance tuning

OS tuning

- ## Kernel tuning

  - ### Don't swap! `vm.swappiness = 0` (RHEL 6.5 onwards: 1)

  - ### Allow more dirty pages but less dirty cache.

    - LinkedIn have lots of RAM in servers, most of it is for page cache (60 of 64 GB). They let dirty pages built up, but cache should be available as Kafka does lots of disk and network I/O.

    - See `vm.dirty_*_ratio` & friends

- ## Disk throughput

  - ### Longer commit interval on mount points. (ext3 or ext4?)

    - Normal interval for ext3 mount point is 30s (?) between flushes; LinkedIn: 120s. They can tolerate losing 2mins worth of data (because of partition replicas) so they rather prefer higher throughput here.

  - ### More spindles (RAID10 w/ 14 disks)

Java/JVM tuning

- Biggest issue: garbage collection
  - And, most of the time, the only issue

- Goal is to **minimize GC pause times**
  - Aka "stop-the-world" events – apps are halted until GC finishes

Java garbage collection in Kafka @ Spotify

% of time spent doing Full GC before tuning

% of time spent doing Full GC after tuning
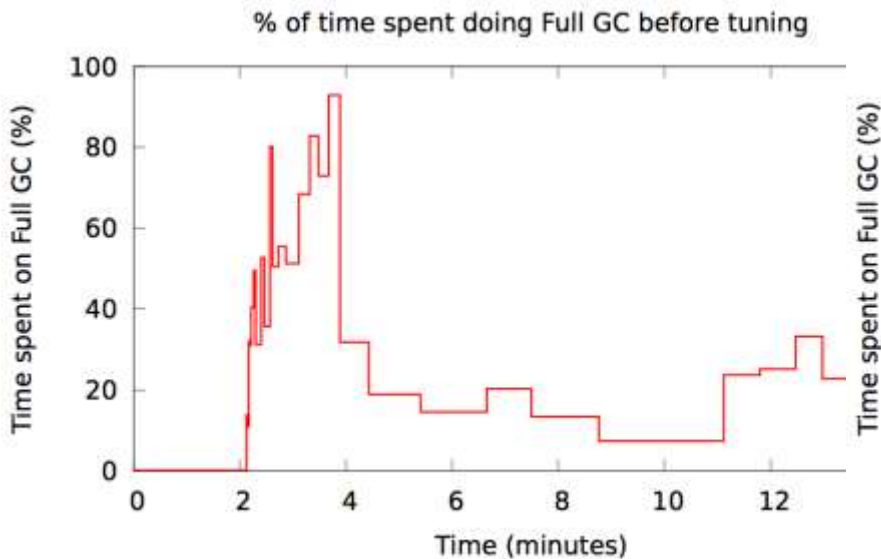
Before tuning   **After tuning**

https://www.jfokus.se/jfokus14/preso/Reliable-real-time-processing-with-Kafka-and-Storm.pdf
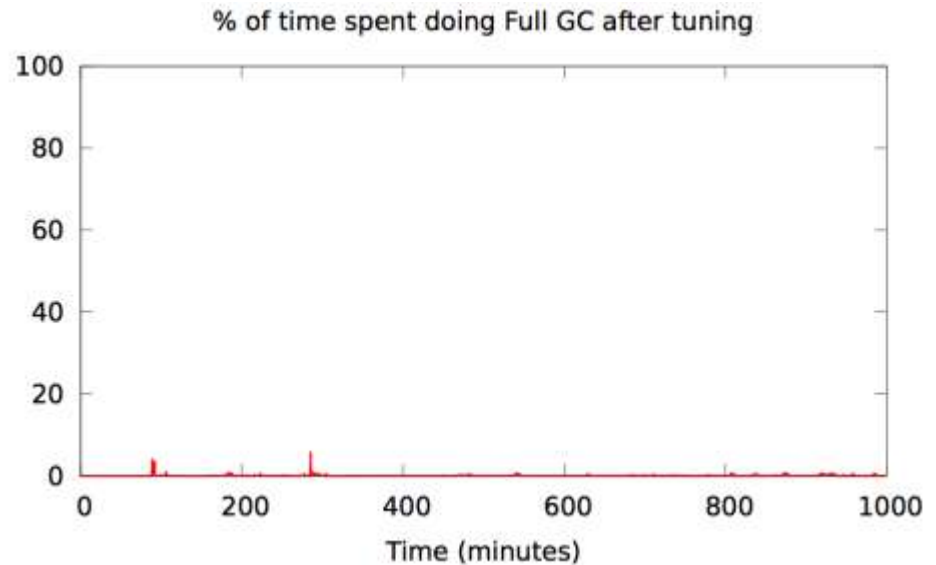
Java/JVM tuning

- Good news: use JDK7u51 or later and have a quiet life!
  - LinkedIn: Oracle JDK, not OpenJDK

- Silver bullet is new G1 "garbage-first" garbage collector
  - Available since JDK7u4.
  - Substantial improvement over all previous GC's, at least for Kafka.

```
$ java -Xms4g -Xmx4g -XX:PermSize=48m -XX:MaxPermSize=48m
        -XX:+UseG1GC
        -XX:MaxGCPauseMillis=20
        -XX:InitiatingHeapOccupancyPercent=35
```

Kafka configuration tuning

- Often not much to do beyond using the defaults, yay. ☺

- Key candidates for tuning:

`num.io.threads`	should be >= #disks (start testing with == #disks)
`num.network.threads`	adjust it based on (concurrent) #producers, #consumers, and replication factor

Kafka usage tuning – lessons learned from others

- Don't break things up into separate topics unless the data in them is truly independent.

  - Consumer behavior can (and will) be extremely variable, don't assume you will always be consuming as fast as you are producing.

- Keep time related messages in the same partition.

  - Consumer behavior can extremely variable, don't assume the lag on all your partitions will be similar.

  - Design a partitioning scheme, so that the owner of one partition can stop consuming for a long period of time and your application will be minimally impacted (for example, partition by transaction id)

http://grokbase.com/t/kafka/users/145qtx4z1c/topic-partitioning-strategy-for-large-data

Ops-related references

- Kafka FAQ

  - https://cwiki.apache.org/confluence/display/KAFKA/FAQ

- Kafka operations

  - https://kafka.apache.org/documentation.html#operations

- Kafka system tools

  - https://cwiki.apache.org/confluence/display/KAFKA/System+Tools

  - Consumer offset checker, get offsets for a topic, print metrics via JMX to console, read from topic A and write to topic B, verify consumer rebalance

- Kafka replication tools

  - https://cwiki.apache.org/confluence/display/KAFKA/Replication+tools

  - Caveat: Some sections of this document are slightly outdated.

  - Controlled shutdown, preferred leader election tool, reassign partitions tool

- Kafka tutorial

  - http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/

Part 4: Developing Kafka apps

Overview of Part 4: Developing Kafka apps

- Writing data to Kafka with producers
    - Example producer
    - Producer types (async, sync)
    - Message acking and batching of messages
    - Write operations behind the scenes – caveats ahead!
- Reading data from Kafka with consumers
    - High-level consumer API and simple consumer API
    - Consumer groups
    - Rebalancing
- Testing Kafka
- Serialization in Kafka
- Data compression in Kafka
- Example Kafka applications
- Dev-related Kafka references

powered by **VERISIGN**

Writing data to Kafka

Writing data to Kafka

- You use Kafka "producers" 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.

- A simple example producer:

```
1   Properties props = new Properties();
2   props.put("metadata.broker.list", "...");
3   ProducerConfig config = new ProducerConfig(props);
4
5   Producer p = new Producer(ProducerConfig config);
6   KeyedMessage<K, V> msg = ...; // cf. later slides
7   p.send(KeyedMessage<K,V> message);
```

- Full details at:
  - https://cwiki.apache.org/confluence/display/KAFKA/0.8.0+Producer+Example

Producers

- The Java producer API is very simple.

  - We'll talk about the slightly confusing details next. ☺

```
1   class kafka.javaapi.producer.Producer<K,V>
2   {
3     public Producer(ProducerConfig config);
4
5     /**
6      * Sends the data to a single topic, partitioned by key, using either the
7      * synchronous or the asynchronous producer.
8      */
9     public void send(KeyedMessage<K,V> message);
10
11    /**
12     * Use this API to send data to multiple topics.
13     */
14    public void send(List<KeyedMessage<K,V>> messages);
15
16    /**
17     * Close API to close the producer pool connections to all Kafka brokers.
18     */
19    public void close();
20  }
```

Producers

- Two types of producers: "async" and "sync"

```
1  Properties props = new Properties();
2  props.put("producer.type", "async");
3  ProducerConfig config = new ProducerConfig(props);
```

  - Same API and configuration, but slightly different semantics.
  - What applies to a sync producer almost always applies to async, too.
  - Async producer is preferred when you want higher throughput.

- Important configuration settings for either producer type:

client.id	identifies producer app, e.g. in system logs
producer.type	async or sync
request.required.acks	acking semantics, cf. next slides
serializer.class	configure encoder, cf. slides on Avro usage
metadata.broker.list	cf. slides on bootstrapping list of brokers

Sync producers

- Straight-forward so I won't cover sync producers here

  - Please go to https://kafka.apache.org/documentation.html

- Most important thing to remember: `producer.send()` will block!

Async producer

- Sends messages in background = no blocking in client.

- Provides more powerful batching of messages (see later).

- Wraps a sync producer, or rather a pool of them.

  - Communication from async->sync producer happens via a queue.

    - Which explains why you may see `kafka.producer.async.QueueFullException`

  - Each sync producer gets a copy of the original async producer config, including the `request.required.acks` setting (see later).

  - Implementation details: Producer, async.AsyncProducer, async.ProducerSendThread, ProducerPool, async.DefaultEventHandler#send()

Async producer

- Caveats

  - Async producer may drop messages if its queue is full.

    - Solution 1: Don't push data to producer faster than it is able to send to brokers.

    - Solution 2: Queue full == need more brokers, add them now! Use this solution in favor of solution 3 particularly if your producer cannot block (async producers).

    - Solution 3: Set `queue.enqueue.timeout.ms` to -1 (default). Now the producer will block indefinitely and will never willingly drop a message.

    - Solution 4: Increase `queue.buffering.max.messages` (default: 10,000).

  - In 0.8 an async producer does not have a callback for `send()` to register error handlers. Callbacks will be available in 0.9.

Producers

- Two aspects worth mentioning because they significantly influence Kafka performance:

    1. Message acking

    2. Batching of messages

1) Message acking

- Background:

  - In Kafka, a message is considered *committed* when "any required" ISR (in-sync replicas) for that partition have applied it to their data log.

  - Message acking is about conveying this "Yes, committed!" information back from the brokers to the producer client.

  - Exact meaning of "any required" is defined by `request.required.acks`.

- Only **producers** must configure acking

  - Exact behavior is configured via `request.required.acks`, which determines when a produce request is considered completed.

  - Allows you to trade **latency (speed)** <-> **durability (data safety)**.

  - Consumers: Acking and how you configured it on the side of producers do not matter to consumers because only committed messages are ever given out to consumers. They don't need to worry about potentially seeing a message that could be lost if the leader fails.

powered by **VERISIGN**

1) Message acking

- Typical values of `request.required.acks`

  - **0**: producer never waits for an ack from the broker.

    - Gives **the lowest latency** but the weakest durability guarantees.

  - **1**: producer gets an ack after the leader replica has received the data.

    - Gives better durability as the we wait until the lead broker acks the request. Only msgs that were written to the now-dead leader but not yet replicated will be lost.

  - **-1**: producer gets an ack after *all* ISR have received the data.

    - Gives **the best durability** as Kafka guarantees that no data will be lost as long as at least one ISR remains.

better latency ↕ better durability

- Beware of interplay with `request.timeout.ms`!

  - "The amount of time the broker will wait trying to meet the `request.required.acks` requirement before sending back an error to the client."

  - Caveat: Message may be committed even when broker sends timeout error to client (e.g. because not all ISR ack'ed in time). One reason for this is that the producer acknowledgement is independent of the leader-follower replication, and ISR's send their acks to the leader, the latter of which will reply to the client.

2) Batching of messages

- Batching improves throughput
  - Tradeoff is data loss if client dies before pending messages have been sent.

- You have two options to "batch" messages in 0.8:
  1. Use send(**listOfMessages**).

     ```
     1   producer.send(List<KeyedMessage<K,V>> messages);
     ```

     - Sync producer: will send this list ("batch") of messages *right now*. Blocks!

     - Async producer: will send this list of messages in background "as usual", i.e. according to batch-related configuration settings. Does not block!

  2. Use send(**singleMessage**) with async producer.

     ```
     1   producer.send(KeyedMessage<K,V> message);
     ```

     - For async the behavior is the same as send(listOfMessages).

2) Batching of messages

- Option 1: How `send(listOfMessages)` works behind the scenes

```
1   producer.send(List<KeyedMessage<K,V>> messages);
```

- The original list of messages is partitioned (randomly if the default partitioner is used) based on their destination partitions/topics, i.e. split into smaller batches.
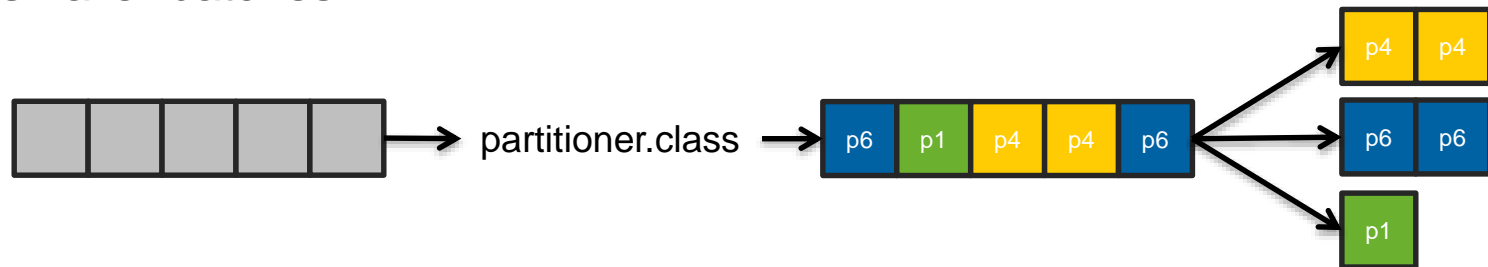
- Each post-split batch is sent to the respective leader broker/ISR (the individual `send()`'s happen sequentially), and each is acked by its respective leader broker according to `request.required.acks`.

powered by **VERISIGN**
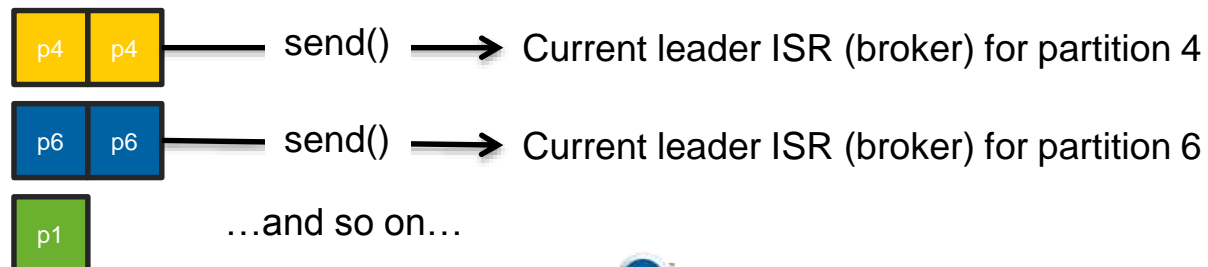
2) Batching of messages

- Option 2: Async producer

  - Standard behavior is to batch messages

  - Semantics are controlled via producer configuration settings

    - `batch.num.messages`

    - `queue.buffering.max.ms` + `queue.buffering.max.messages`

    - `queue.enqueue.timeout.ms`

    - And more, see producer configuration docs.

- Remember: Async producer simply wraps sync producer!

  - But the batch-related config settings above have no effect on "true" sync producers, i.e. when used without a wrapping async producer.

FYI: upcoming producer configuration changes

Kafka 0.8	Kafka 0.9 (unreleased)
`metadata.broker.list`	`bootstrap.servers`
`request.required.acks`	`acks`
`batch.`**`num.messages`**	`batch.`**`size`**
`message.send.max.retries`	`retries`

([This list is not complete, see Kafka docs for details.](#))

Write operations behind the scenes

- When writing to a topic in Kafka, producers write directly to the partition leaders (brokers) of that topic

  - Remember: Writes always go to the leader ISR of a partition!
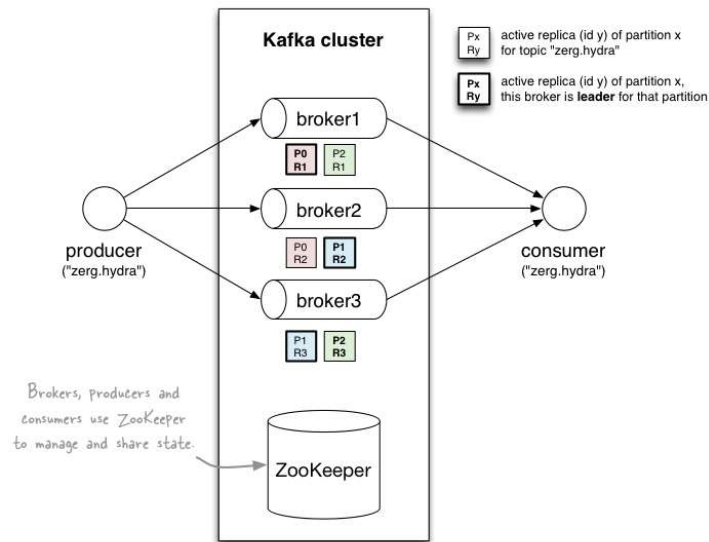
- This raises two questions:

  - How to know the "right" partition for a given topic?

  - How to know the current leader broker/replica of a partition?

1) How to know the "right" partition when sending?

- In Kafka, a producer – i.e. the **client** – decides to which target partition a message will be sent.

  - Can be random ~ load balancing across receiving brokers.

  - Can be semantic based on message "key", e.g. by user ID or domain name.

    - Here, Kafka guarantees that all data for the same key will go to the same partition, so consumers can make locality assumptions.

```
1    // Java example.  Topic is "zerg.hydra".
2    KeyedMessage<String, String> msg = new KeyedMessage<>("zerg.hydra", "myValue");
3    KeyedMessage<String, String> msg = new KeyedMessage<>("zerg.hydra", "myKey", "myValue");
```

- But there's one catch with line 2 (i.e. no key) in Kafka 0.8.

Keyed vs. non-keyed messages in Kafka 0.8

- If a key **is not** specified:

```
2   KeyedMessage<String, String> msg = new KeyedMessage<>("zerg.hydra", "myValue");
```

- - Producer will *ignore* any configured partitioner.

  - It will pick a random partition from the list of *available* partitions **and stick to it** for some time before switching to another one = NOT round robin or similar!

    - Why? To reduce number of open sockets in large Kafka deployments (KAFKA-1017).

    - Default: 10mins, cf. `topic.metadata.refresh.interval.ms`

    - See implementation in `DefaultEventHandler#getPartition()`

  - If there are fewer producers than partitions at a given point of time, some partitions may not receive any data. How to fix if needed?

    - Try to reduce the metadata refresh interval `topic.metadata.refresh.interval.ms`

    - Specify a message key and a customized random partitioner.

  - In practice it is not trivial to implement a correct "random" partitioner in Kafka 0.8.

    - Partitioner interface in Kafka 0.8 lacks sufficient information to let a partitioner select a random and *available* partition. Same issue with DefaultPartitioner.

Keyed vs. non-keyed messages in Kafka 0.8

- If a key **is** specified:

```
3   KeyedMessage<String, String> msg = new KeyedMessage<>("zerg.hydra", "myKey", "myValue");
```

  - Key is retained as part of the msg, will be stored in the broker.

  - One can design a partition function to route the msg based on key.

  - The *default partitioner* assigns messages to a partition based on their key hashes, via `key.hashCode % numPartitions`.

  - Caveat:

    - If you specify a key for a message but do not explicitly wire in a custom partitioner via `partitioner.class`, your producer will use the default partitioner.

    - So without a custom partitioner, messages with the same key will still end up in the same partition! (cf. default partitioner's behavior above)

2) How to know the current leader of a partition?

- Producers: broker discovery aka bootstrapping

  - Producers don't talk to ZooKeeper, so it's not through ZK.

  - Broker discovery is achieved by providing producers with a "bootstrapping" broker list, cf. `metadata.broker.list`

    - These brokers inform the producer about all alive brokers and where to find current partition leaders. The bootstrap brokers do use ZK for that.

- Impacts on failure handling

  - In Kafka 0.8 the bootstrap list is static/immutable during producer run-time. This has limitations and problems as shown in next slide.

  - The current bootstrap approach will improve in Kafka 0.9. This change will make the life of Ops easier.

Bootstrapping in Kafka 0.8

- Scenario: N=5 brokers total, 2 of which are for bootstrap

```
1  Properties props = new Properties();
2  props.put("metadata.broker.list", "broker1:9092,broker2:9092");
3  ProducerConfig config = new ProducerConfig(props);
```

broker1	broker2	broker3	broker4	broker5

- Do's:
  - Take down one bootstrap broker (e.g. broker2), repair it, and bring it back.
  - In terms of impacts on *broker discovery*, you can do whatever you want to brokers 3-5.
- Don'ts:
  - Stop all bootstrap brokers 1+2. If you do, the producer stops working!
- To improve operational flexibility, use VIP's or similar for values in `metadata.broker.list`.

Reading data from Kafka

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**
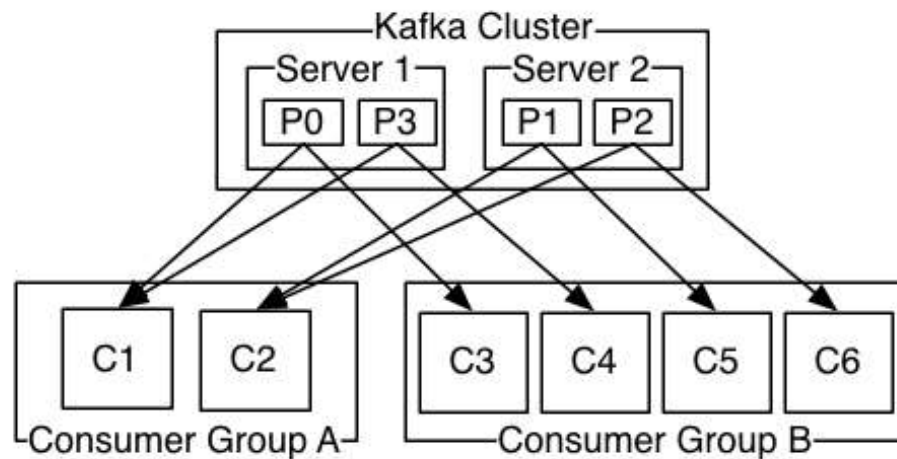
# Guarantees when reading data from Kafka

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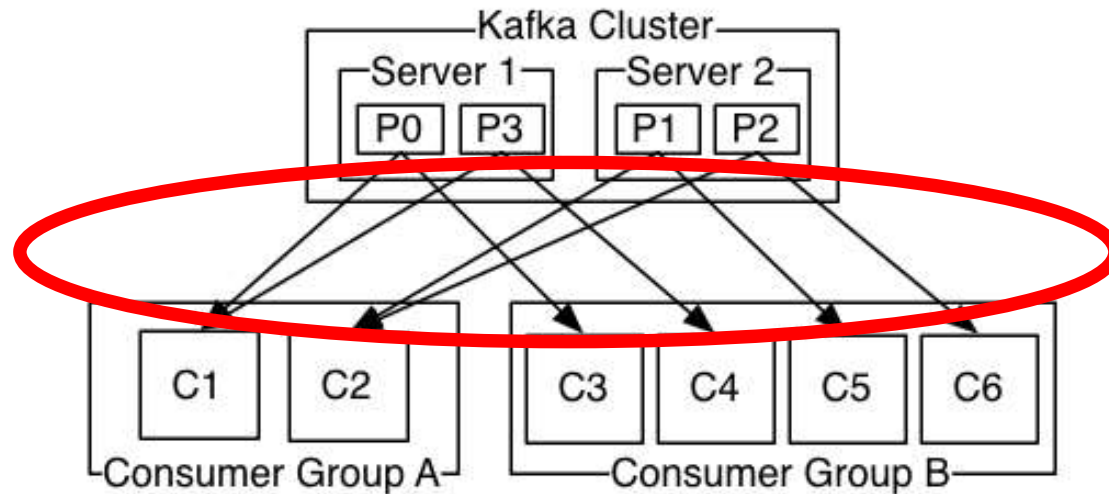
- The assignment of brokers – via the partitions of a topic – to consumers is quite **important**, and it is **dynamic** at run-time.

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.
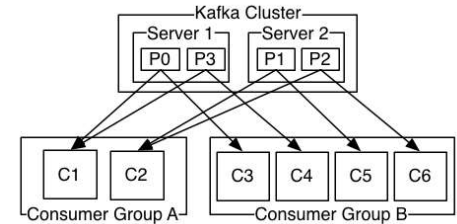
Rebalancing: how consumers meet brokers

- **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.

Testing Kafka apps

Testing Kafka apps

- Won't have the time to cover testing in this workshop.

- Some hints:

  - Unit-test your individual classes like usual

  - When integration testing, use in-memory instances of Kafka and ZK

  - Test-drive your producers/consumers in virtual Kafka clusters via Wirbelsturm

- Starting points:

  - Kafka's own test suite

    - 0.8.1: https://github.com/apache/kafka/tree/0.8.1/core/src/test

    - trunk: https://github.com/apache/kafka/tree/trunk/core/src/test/

  - Kafka tests in kafka-storm-starter

    - https://github.com/miguno/kafka-storm-starter/

Serialization in Kafka

Serialization in Kafka

- Kafka does not care about data format of msg payload
- Up to developer (= you) to handle serialization/deserialization
  - Common choices in practice: Avro, JSON

```
1 ▼  /**
2      *  Create a list of message streams of type T for each topic.
3      *
4      *  @param topicCountMap   a map of (topic, #streams) pair
5      *  @param decoder a decoder that converts from Message to T
6      *  @return a map of (topic, list of  KafkaStream) pairs.
7      *          The number of items in the list is #streams. Each stream supports
8      *          an iterator over message/metadata pairs.
9      */
10    public <K,V> Map<String, List<KafkaStream<K,V>>>
11      createMessageStreams(Map<String, Integer> topicCountMap, Decoder<K> keyDecoder, Decoder<V> valueDecoder);
12
13 ▼  /**
14     *  Create a list of message streams of type T for each topic, using the default decoder.
15     */
16    public Map<String, List<KafkaStream<byte[], byte[]>>> createMessageStreams(Map<String, Integer> topicCountMap);
```

(Code above is from the High Level Consumer API)

powered by **VERISIGN**

Serialization in Kafka: using Avro

- One way to use Avro in Kafka is via Twitter Bijection.

  - https://github.com/twitter/bijection

- Approach: Convert pojo to byte[], then send byte[] to Kafka.

  - Bijection in Scala:

```scala
5    val tweet = new Tweet("miguno", "Terran is the cheese race.", 1366190000)
6
7    // From POJO to byte array
8    val bytes = Injection[Tweet, Array[Byte]](tweet)
9
10   // From byte array back to POJO
11   val recovered_tweet = Injection.invert[Tweet, Array[Byte]](bytes)
```

  - Bijection in Java: https://github.com/twitter/bijection/wiki/Using-bijection-from-java

- Full Kafka/Bijection example:

  - KafkaSpec in kafka-storm-starter

- Alternatives to Bijection:

  - e.g. https://github.com/miguno/kafka-avro-codec

Data compression in Kafka

Data compression in Kafka

- Again, no time to cover compression in this training.

  - But worth looking into!

  - Interplay with batching of messages, e.g. larger batches typically achieve better compression ratios.

- Details about compression in Kafka:

  - https://cwiki.apache.org/confluence/display/KAFKA/Compression

  - Blog post by Neha Narkhede, Kafka committer @ LinkedIn: http://geekmantra.wordpress.com/2013/03/28/compression-in-kafka-gzip-or-snappy/

Example Kafka applications

kafka-storm-starter

- Written by yours truly

- https://github.com/miguno/kafka-storm-starter

```
$ git clone https://github.com/miguno/kafka-storm-starter
$ cd kafka-storm-starter

# Now ready for mayhem!
```

(Must have JDK 6 installed.)

kafka-storm-starter: run the test suite

```
$ ./sbt test
```

- Will run unit tests plus end-to-end tests of Kafka, Storm, and Kafka-Storm integration.

```
[info]      And I Avro-encode the tweets and use the Kafka producer app to sent them to Kafka
[info]      Then the Kafka consumer app should receive the original tweets from the Storm topology
[info] Feature: AvroScheme[T] for Kafka spout
[info]    Scenario: User creates a Storm topology that uses AvroScheme in Kafka spout
[info]      Given a ZooKeeper instance
[info]      And a Kafka broker instance
[info]      And a Storm topology that uses AvroScheme and that reads tweets from topic testing-input and writes them as-is to topic tes
[info]      And some tweets
[info]      And a synchronous Kafka producer app that writes to the topic testing-input
[info]      And a single-threaded Kafka consumer app that reads from topic testing-output and Avro-decodes the incoming data
[info]      And a Storm topology configuration that registers an Avro Kryo decorator for Tweet
[info]      When I run the Storm topology
[info]      And I Avro-encode the tweets and use the Kafka producer app to sent them to Kafka
[info]      Then the Kafka consumer app should receive the original tweets from the Storm topology
[info] Run completed in 28 seconds, 597 milliseconds.
[info] Total number of tests run: 25
[info] Suites: completed 8, aborted 0
[info] Tests: succeeded 25, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
27901 [Thread-147-kafka-spout-EventThread] INFO  com.netflix.curator.framework.state.ConnectionStateManager - State change: SUSPENDED
27901 [Thread-106-kafka-spout-EventThread] INFO  com.netflix.curator.framework.state.ConnectionStateManager - State change: SUSPENDED
27901 [Thread-147-kafka-spout-EventThread] INFO  com.netflix.curator.framework.state.ConnectionStateManager - State change: SUSPENDED
27901 [Thread-106-kafka-spout-EventThread] INFO  com.netflix.curator.framework.state.ConnectionStateManager - State change: SUSPENDED
27901 [ConnectionStateManager-0] WARN  com.netflix.curator.framework.state.ConnectionStateManager - There are no ConnectionStateLister
27903 [ConnectionStateManager-0] WARN  com.netflix.curator.framework.state.ConnectionStateManager - There are no ConnectionStateLister
27905 [ConnectionStateManager-0] WARN  com.netflix.curator.framework.state.ConnectionStateManager - There are no ConnectionStateLister
27905 [ConnectionStateManager-0] WARN  com.netflix.curator.framework.state.ConnectionStateManager - There are no ConnectionStateLister
[success] Total time: 40 s, completed Jun 10, 2014 2:44:03 PM
:kafka-storm-starter (develop) $
```

kafka-storm-starter: run the KafkaStormDemo app

```
$ ./sbt run
```

- Starts in-memory instances of ZooKeeper, Kafka, and Storm. Then runs a Storm topology that reads from Kafka.

```
7031 [Thread-19] INFO  backtype.storm.daemon.worker - Worker 3f7f1a51-5c9e-43a5-b431-e39a7272215e for storm
7033 [Thread-29-kafka-spout] INFO  storm.kafka.DynamicBrokersReader - Read partition info from zookeeper: Gl
7050 [Thread-29-kafka-spout] INFO  backtype.storm.daemon.executor - Opened spout kafka-spout:(1)
7051 [Thread-29-kafka-spout] INFO  backtype.storm.daemon.executor - Activating spout kafka-spout:(1)
7051 [Thread-29-kafka-spout] INFO  storm.kafka.ZkCoordinator - Refreshing partition manager connections
7065 [Thread-29-kafka-spout] INFO  storm.kafka.DynamicBrokersReader - Read partition info from zookeeper: Gl
7066 [Thread-29-kafka-spout] INFO  storm.kafka.ZkCoordinator - Deleted partition managers: []
7066 [Thread-29-kafka-spout] INFO  storm.kafka.ZkCoordinator - New partition managers: [Partition{host=127.0
7083 [Thread-29-kafka-spout] INFO  storm.kafka.PartitionManager - Read partition information from: /kafka-sp
7100 [Thread-29-kafka-spout] INFO  storm.kafka.PartitionManager - No partition information found, using conf
7105 [Thread-29-kafka-spout] INFO  storm.kafka.PartitionManager - Starting Kafka 127.0.0.1:0 from offset 18
7106 [Thread-29-kafka-spout] INFO  storm.kafka.ZkCoordinator - Finished refreshing
7126 [Thread-29-kafka-spout] INFO  storm.kafka.PartitionManager - Committing offset for Partition{host=127.0
7126 [Thread-29-kafka-spout] INFO  storm.kafka.PartitionManager - Committed offset 18 for Partition{host=127
9128 [Thread-29-kafka-spout] INFO  storm.kafka.PartitionManager - Committing offset for Partition{host=127.0
9129 [Thread-29-kafka-spout] INFO  storm.kafka.PartitionManager - Committed offset 18 for Partition{host=127
```

Kafka related code in kafka-storm-starter

- KafkaProducerApp

  - https://github.com/miguno/kafka-storm-starter/blob/develop/src/main/scala/com/miguno/kafkastorm/kafka/KafkaProducerApp.scala

- KafkaConsumerApp

  - https://github.com/miguno/kafka-storm-starter/blob/develop/src/main/scala/com/miguno/kafkastorm/kafka/KafkaConsumerApp.scala

- KafkaSpec: test-drives producer and consumer above

  - https://github.com/miguno/kafka-storm-starter/blob/develop/src/test/scala/com/miguno/kafkastorm/integration/KafkaSpec.scala

Dev-related references

- [Kafka documentation](#)

- [Kafka FAQ](#)

- Kafka tutorials

  - [http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/](#)

- Code examples

  - [https://github.com/miguno/kafka-storm-starter/](#)

powered by **VERISIGN**

Part 5: Playing with Kafka using Wirbelsturm

1-click Kafka deployments

Deploying Kafka via Wirbelsturm

- Written by yours truly
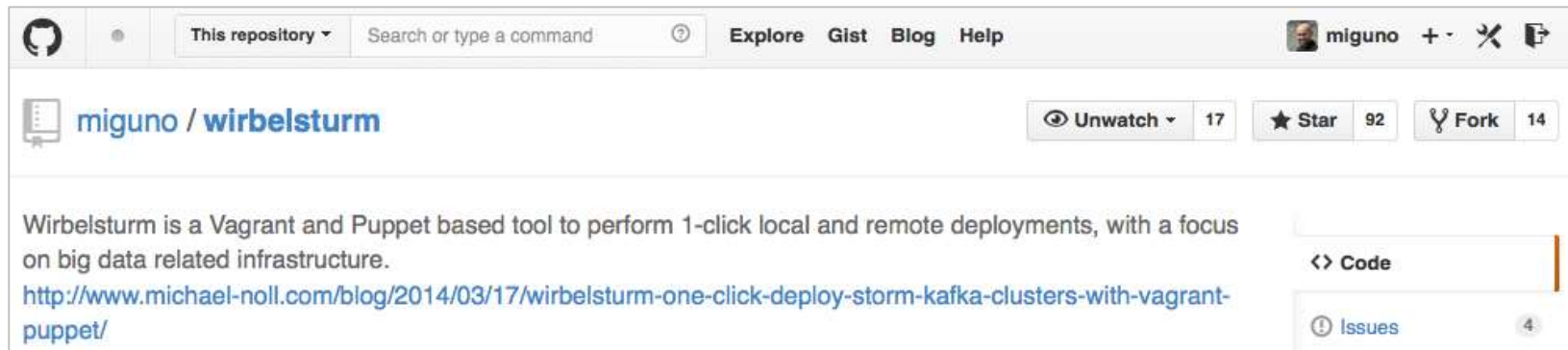
- https://github.com/miguno/wirbelsturm

```
$ git clone https://github.com/miguno/wirbelsturm.git
$ cd wirbelsturm
$ ./bootstrap
$ vi wirbelsturm.yaml  # uncomment Kafka section
$ vagrant up zookeeper1 kafka1
```

(Must have Vagrant 1.6.1+ and VirtualBox 4.3+ installed.)

What can I do with Wirbelsturm?

- Get a first impression of Kafka

- Test-drive your producer apps and consumer apps

- Test failure handling

  - Stop/kill brokers, check what happens to producers or consumers.

  - Stop/kill ZooKeeper instances, check what happens to brokers.

- Use as sandbox environment to test/validate deployments

  - "What will actually happen when I run this reassign partition tool?"

  - "What will actually happen when I delete a topic?"

  - "Will my Hiera changes actually work?"

- Reproduce production issues, share results with Dev

  - Also helpful when reporting back to Kafka project and mailing lists.

- Any further cool ideas? ☺

Wrapping up

Where to find help

- No (good) Kafka book available yet.
- Kafka documentation
  - http://kafka.apache.org/documentation.html
  - https://cwiki.apache.org/confluence/display/KAFKA/Index
- Kafka ecosystem, e.g. Storm integration, Puppet
  - https://cwiki.apache.org/confluence/display/KAFKA/Ecosystem
- Mailing lists
  - http://kafka.apache.org/contact.html
- Code examples
  - `examples/` directory in Kafka, https://github.com/apache/kafka/
  - https://github.com/miguno/kafka-storm-starter/