

Apache Kafka® Administration By Confluent

Version 6.0.0-v2.0.1



CONFLUENT

Table of Contents

Introduction	1
Class Logistics and Overview	2
Fundamentals Review	9
1: Bridging From Fundamentals	12
1a: How Can You Leverage Replication?	16
Lab: Introduction	24
Lab: Using Kafka's Command Line Tools	25
Lab: Producing Records with a Null Key	26
2: Producing Messages Reliably	27
2a: How Can Producers Provide Durability?	29
2b: How Can You Guarantee Messages are Written without Duplication?	35
2c: How Does Kafka Handle the Notion of Producers Sending Messages in Transactions?	38
3: Replicating Data: A Deeper Dive	44
3a: How Does Kafka Determine Which Messages Can be Consumed?	49
3b: How Does Kafka Place Replicas and How Can You Control Replication Further?	59
3c: How Does Kafka React When a Leader Dies?	67
3d: How Does Kafka Track Leadership Changes?	72
3e: What Are Some Other Replication Considerations?	

4: Providing Durability in Other Ways	85
4a: How Does Kafka Organize Files to Store Partition Data?	87
4b: What are the Basics of Scaling Consumption?	94
4c: How Does Kafka Maintain Consumer Offsets?	99
Lab: Investigating the Distributed Log	105
5: Configuring a Kafka Cluster	106
5a: How Do You Configure Brokers?	108
5b: What if You Want to Adjust Settings Dynamically or Apply at the Topic Level?	115
Lab: Exploring Configuration	127
Lab: Increasing Replication Factor	128
6: Managing a Kafka Cluster	129
6a: What Should You Consider When Installing and Upgrading Kafka?	131
6b: What are the Basics of Monitoring Kafka?	136
6c: How Can You Decide How Kafka Keeps Messages?	145
6d: How Can You Move Partitions To New Brokers Easily?	156
6e: What Should You Consider When Shrinking a Cluster?	163
Lab: Kafka Administrative Tools	167
7: Balancing Load with Consumer Groups and Partitions	168
7a: How Do Partitions and Consumers Scale?	170
7b: How Do Groups Distribute Work Across Partitions?	177

7c: How Does Kafka Manage Groups?	183
Lab: Modifying Partitions and Viewing Offsets	189
8: Optimizing Kafka's Performance	190
8a: How Does Kafka Handle the Idea of Sending Many Messages at Once?	193
Lab: Exploring Producer Performance	201
8b: How Do Produce and Fetch Requests Get Processed on a Broker?	202
8c: How Can You Measure and Control How Requests Make It Through a Broker?	209
8d: What Else Can Affect Broker Performance?	219
8e: How Do You Control It So One Client Does Not Dominate the Broker Resources?	225
8f: What Should You Consider in Assessing Client Performance?	234
8g: How Can You Test How Clients Perform?	239
Lab: Performance Tuning	243
9: Securing a Kafka Cluster	244
9a: What are the Basic Ideas You Should Know about Kafka Security?	247
9b: What Options Do You Have For Securing a Kafka/Confluent Deployment?	254
9c: How Can You Easily Control Who Can Access What?	258
9d: What Should You Know Securing a Deployment Beyond Kafka Itself?	272
Lab: Securing the Kafka Cluster	278
10: Understanding Kafka Connect	279
10a: What Can You Do with Kafka Connect?	281

10b: How Do You Configure Workers and Connectors?	292
10c: Deep Dive into a Connector & Finding Connectors	301
10d: What Else Can One Do With Connect?	309
Lab: Running Kafka Connect	312
11: Deploying Kafka in Production	313
11a: What Does Confluent Advise for Deploying Brokers in Production?	316
11b: What Does Confluent Advise for Deploying Zookeeper in Production?	330
11c: What Does Confluent Advise for Deploying Kafka Connect in Production?	335
11d: What Does Confluent Advise for Deploying Schema Registry in Production?	339
11e: What Does Confluent Advise for Deploying the REST Proxy in Production?	343
11f: What Does Confluent Advise for Deploying Kafka Streams and ksqlDB in Production?	347
11g: What Does Confluent Advise for Deploying Control Center in Production?	351
Conclusion	354
Appendix: Additional Problems to Solve	361
Problem A: Partitioning with Keys	363
Problem B: Partitions and More	365
Problem C: Log Segment Files	366
Problem D: Groups, Consumers, and Partitions	367
Problem E: Compaction: A Deeper Dive	369
Problem F: Partitioning without Keys	370

Appendix: Confluent Technical Fundamentals of Apache Kafka® Content

373

1: Getting Started	375
2: How are Messages Organized?	384
3: How Do I Scale and Do More Things With My Data?	389
4: What's Going On Inside Kafka?	398
5: Recapping and Going Further	407
Appendix: Additional Content	416
Appendix A: Detailed Transactions Demo	418
Appendix B: How Can You Monitor Replication?	433
Appendix C: Multi-Region Clusters	438
Appendix D: SSL and SASL Details	452
Appendix E: How Can You Connect to a Cluster?	478
Appendix: Fundamentals Review	484
Appendix: Some Alternative Slides	496
Alt Slides: How Does a Produce Request Get Processed on a Broker?	498
Alt Slides: How Does a Fetch Request Get Processed on a Broker?	507

Introduction



CONFLUENT Global Education

Class Logistics and Overview

Copyright & Trademarks

Copyright © Confluent, Inc. 2014-2022. [Privacy Policy](#) | [Terms & Conditions](#).

Apache, Apache Kafka, Kafka, and the Kafka logo are trademarks of the
[Apache Software Foundation](#)

All other trademarks, product names, and company names or logos cited herein are the property of their respective owners.

Prerequisite

This course requires a working knowledge of the Apache Kafka architecture.

New to Kafka? Need a refresher?

Sign up for free ***Confluent Fundamentals for Apache Kafka*** course at

<https://confluent.io/training>

Agenda



This course consists of these modules:

- Bridging From Fundamentals
- Producing Messages Reliably
- Replicating Data: A Deeper Dive
- Providing Durability in Other Ways
- Configuring a Kafka Cluster
- Managing a Kafka Cluster
- Balancing Load with Consumer Groups and Partitions
- Optimizing Kafka's Performance
- Securing a Kafka Cluster
- Understanding Kafka Connect
- Deploying Kafka in Production

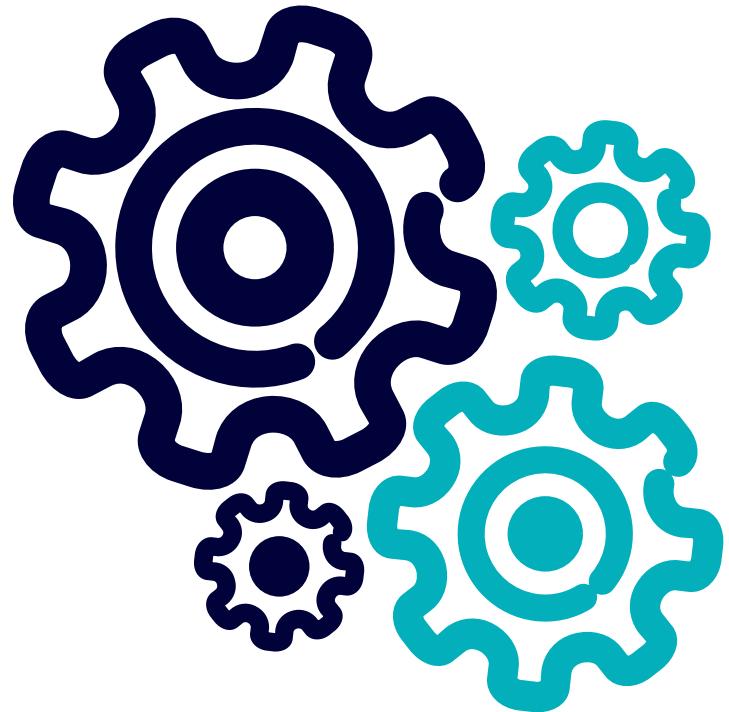
Course Objectives

Upon completion of this course, you should be able to:

- Describe how Kafka brokers, producers, and consumers work
- Describe how replication works within the cluster
- Understand hardware and runtime configuration options
- Monitor and administer your Kafka cluster
- Integrate Kafka with external systems using Kafka Connect
- Design a Kafka cluster for high availability & fault tolerance

Throughout the course, Hands-On Exercises and Activities will reinforce the topics being discussed.

Class Logistics



- Timing
 - Start and end times
 - Can I come in early/stay late?
 - Breaks
 - Lunch
- Physical Class Concerns
 - Restrooms
 - Wi-Fi and other information
 - Emergency procedures
 - Don't leave belongings unattended



No recording, please!

How to get the courseware?



1. Register at **training.confluent.io**
2. Verify your email
3. Log in to **training.confluent.io** and enter your **license activation key**
4. Go to the **Classes** dashboard and select your class

Introductions



- About you:
 - What is your name, your company, and your role?
 - Where are you located (city, timezone)?
 - What is your experience with Kafka?
 - Which other Confluent courses have you attended, if any?
 - Optional talking points:
 - What are some other distributed systems you like to work with?
 - What technology most excited you early in your life?
 - Anything else you want to share?
- About your instructor

Fundamentals Review

Discussion

Question Set 1 [6 mins]

Determine if each statement is true or false and why:

1. All messages in a topic are on the same broker.
2. All messages in a partition are on the same broker.
3. All messages that have the same key will be on the same broker.
4. The more partitions a topic has, the better.

Question Set 2 [3 mins]

Determine the best answer to each question.

1. What are the roles of a producer and a consumer?
2. How is it decided which messages consumers read?
3. Who initiates the reading of messages: consumers or the Kafka cluster?

Discussion, Cont'd.

Question 3 [1 min]

Suppose there is a message in our Kafka cluster about my breakfast purchase of \$12.73. Consumer c_0 has consumed it to process the charge. Could consumer c_7 consume this same message this afternoon?

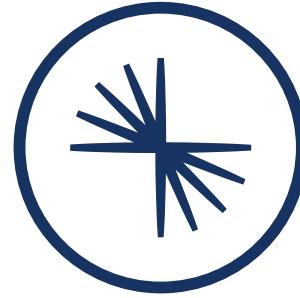
Question 4 [2 mins]

Kafka has a transactions API. When we know that all messages that are part of a transaction successfully made it to the cluster, we want to tag those messages as "good." When we know that not all messages in a transaction made it, we want to tag those messages that did make it as "bad." Kafka uses markers in the logs, that are effectively new messages written after existing messages to do this. Why not just put something in the metadata? Why not delete "bad" messages?

Instructor-Led Review

Some time is allocated here for an instructor-led review/Q&A on prerequisite concepts from Fundamentals.

1: Bridging From Fundamentals



CONFLUENT
Global Education

Module Overview



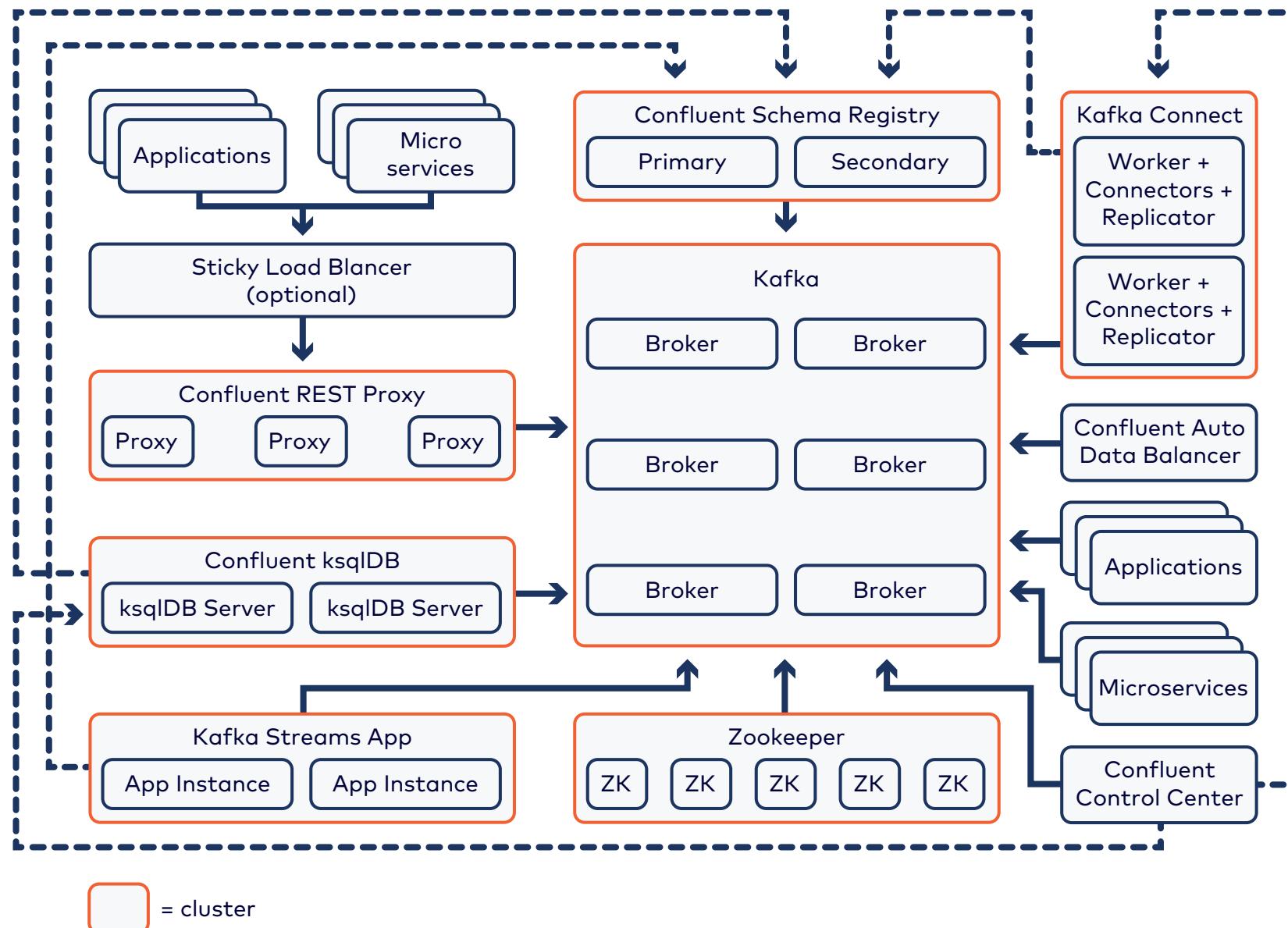
This module contains 1 lesson:

- How Can You Leverage Replication?

Where this fits in:

- Recommended Prerequisite: Fundamentals course

Kafka Deployment Architecture



What Does Confluent Platform Add to Kafka?

CONFLUENT PLATFORM

SECURITY & RESILIENCY

RBAC | Audit Logs | Schema Validation | Multi-Region Clusters | Replicator | Cluster Linking

PERFORMANCE & SCALABILITY

Tiered Storage | Self-Balancing Clusters | K8s Operator

MANAGEMENT & MONITORING

Control Center | Proactive Support

DEVELOPMENT & CONNECTIVITY

Connectors | Non-Java Clients | REST Proxy | Schema Registry | ksqlDB

APACHE KAFKA®

Core | Connect API | Streams API

1a: How Can You Leverage Replication?

Description

Review of leaders vs. followers. Replication factor. How messages get from leaders to followers and config. ISRs. Leader failover / leader election.

Review: Basics of Replication

- Ensure high availability of data with backup copies

- Replicas:

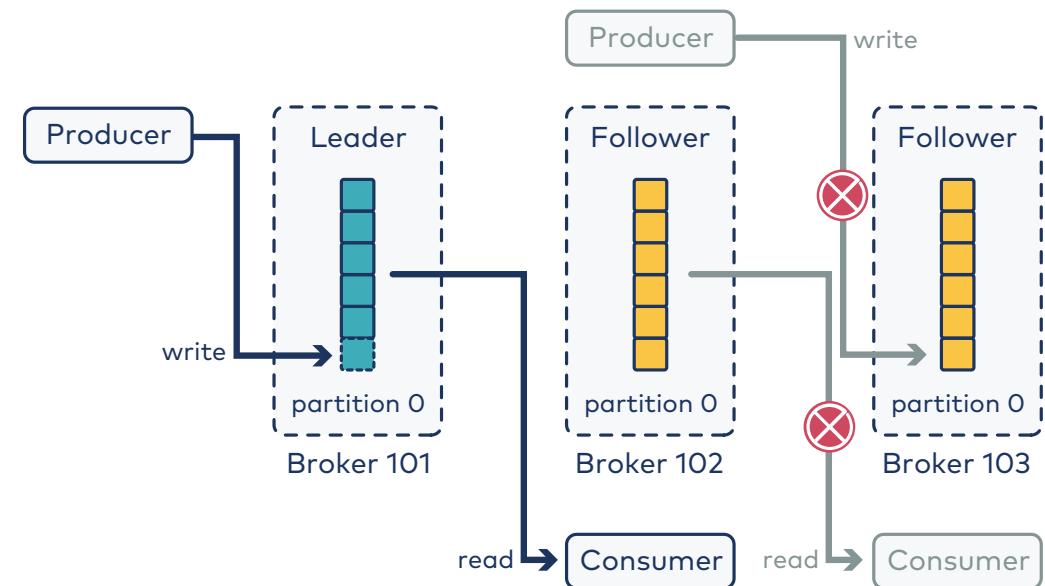
Leader

Clients write to and read from, always one leader

Follower

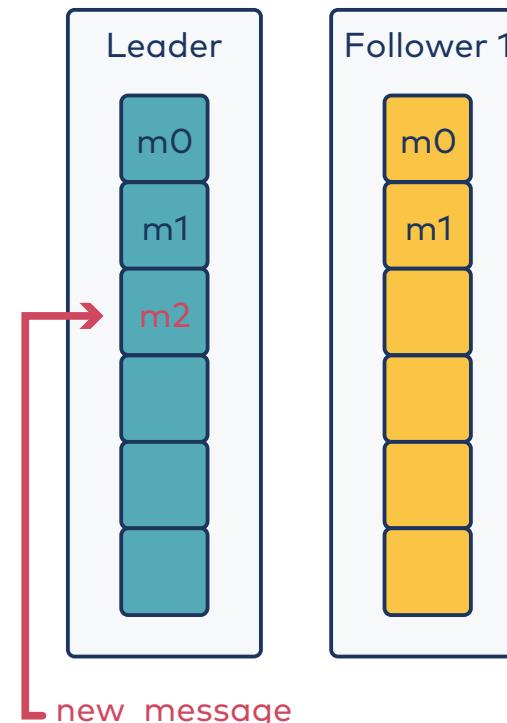
Backup copies, keep up with leader, generally multiple followers

- Topic setting `replication.factor`

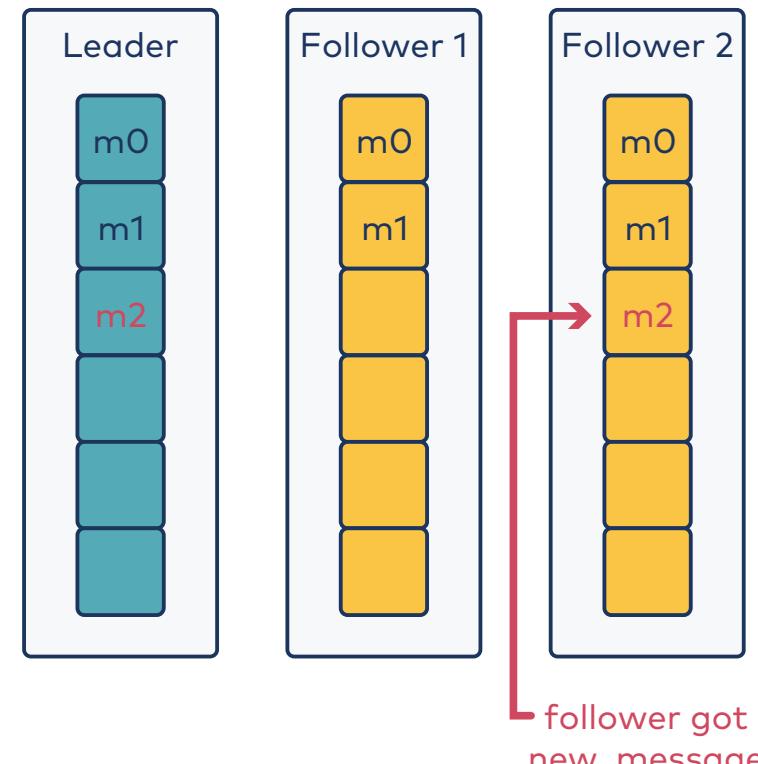


"Follow the Leader"

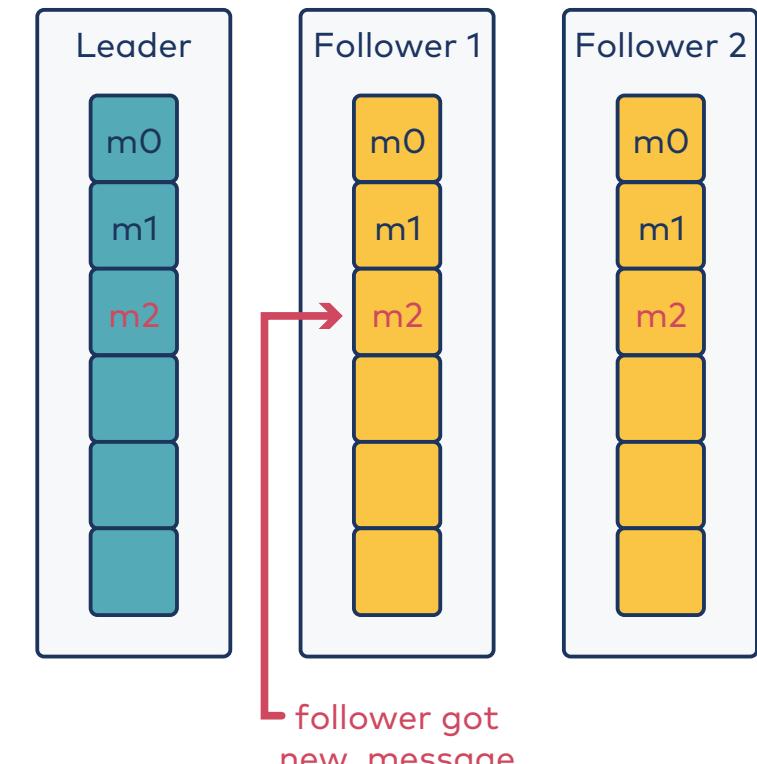
Step 1



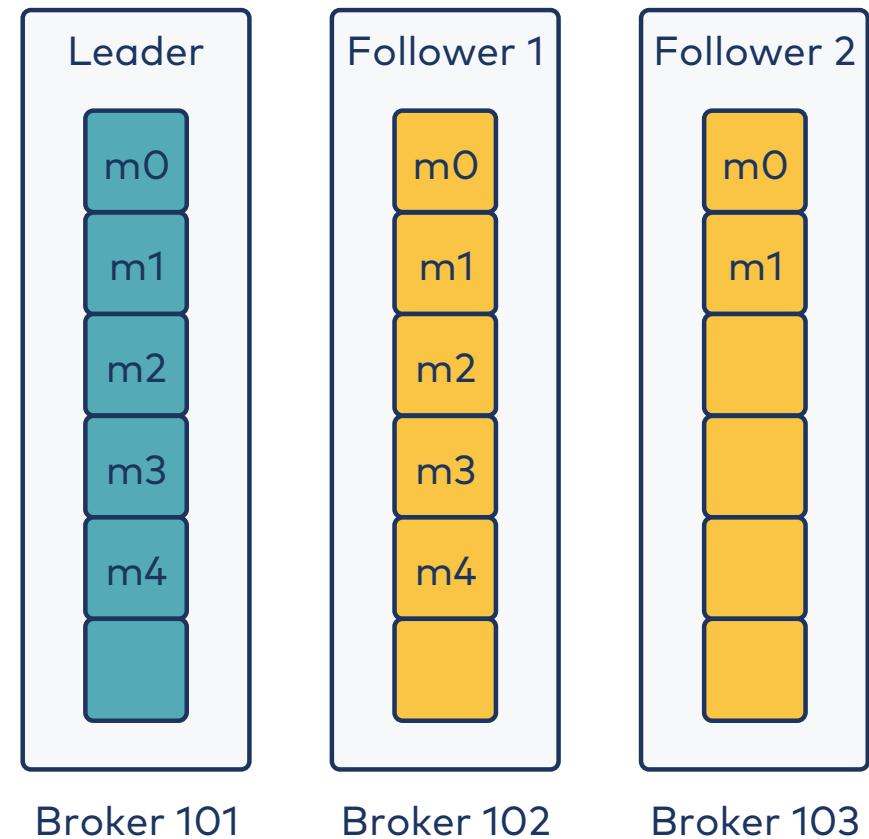
Step 2



Step 3



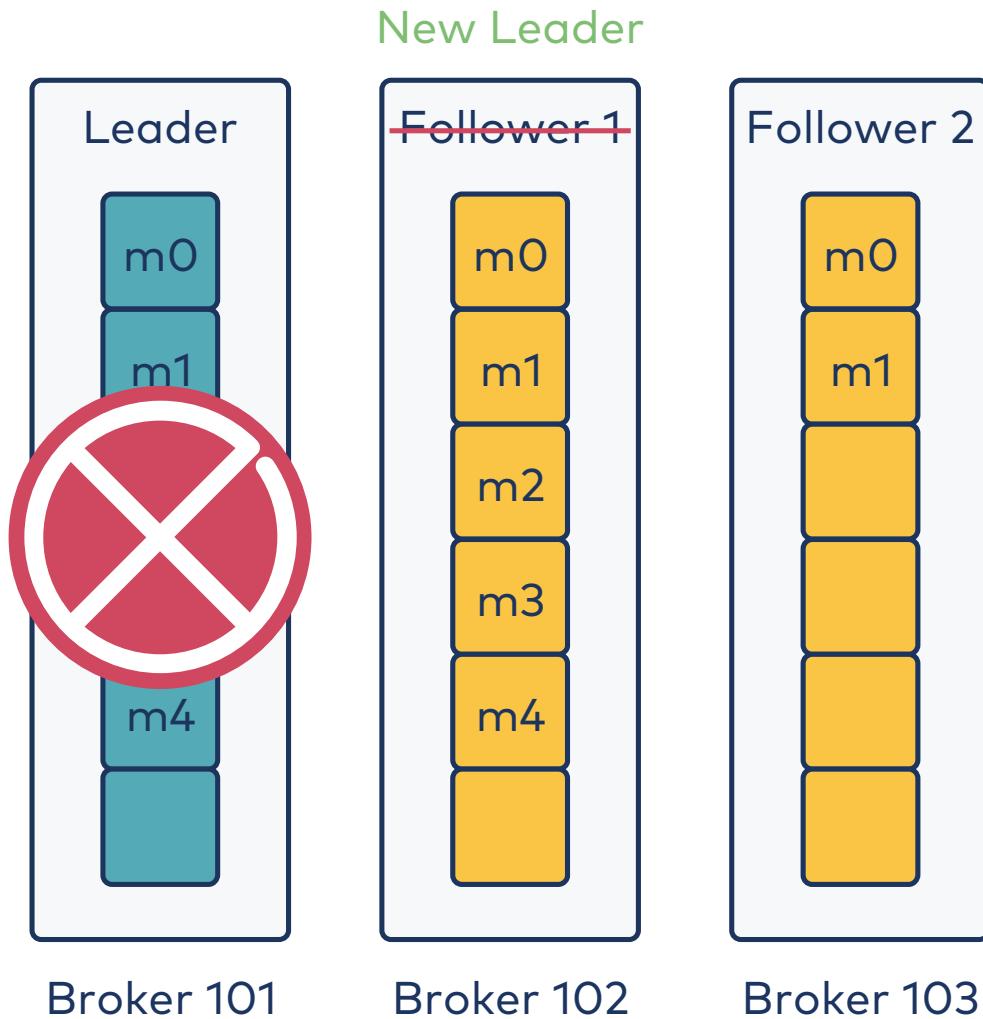
But...



Observe:

- Follower 1 (on Broker 102) is an **in-sync replica (ISR)**
- Follower 2 (on Broker 103) is **not**

Leader Failover



Question: Would either choice of follower have been equally good to replace the leader that had died?

How Does Kafka Choose Leaders?

- Leader election happens automatically
- Kafka will generally choose an in-sync follower to become leader
- Leader election does not happen in parallel
- Background processes manage balance of leadership

Activity: Exploring Replica Placement & Replication Behavior



Say we have 5 brokers - b_0, b_1, \dots, b_4 .

Say we have a replication factor of 4.

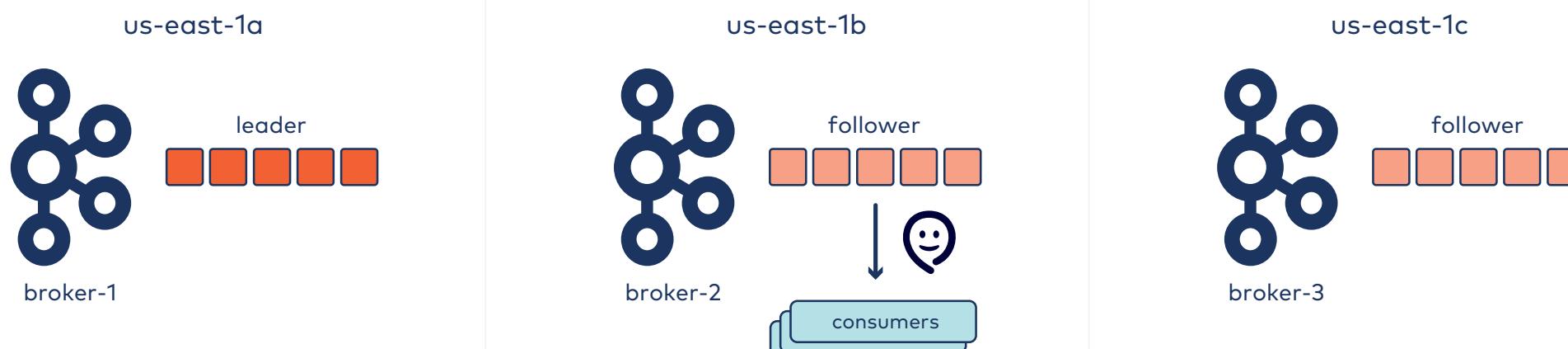
1. How many followers would we have?
2. Say leader is on broker b_4 .
 - a. Where could the followers be?
 - b. Where could a follower **not** be?
3. Say we have 3 successfully written messages that have been properly replicated. It's time to write the fourth message.
 - a. Where does it go?
 - b. What happens next?
4. Say broker b_4 fails. What happens? Why?



A Step Beyond

Follower Fetching

In this lesson, we told you all clients must interact with the leader. But, it is possible to configure it so consumers fetch from followers in the same AZ to reduce costs...



`broker.rack=us-east-1a`

`broker.rack=us-east-1b`
`client.rack=us-east-1b`

`broker.rack=us-east-1c`

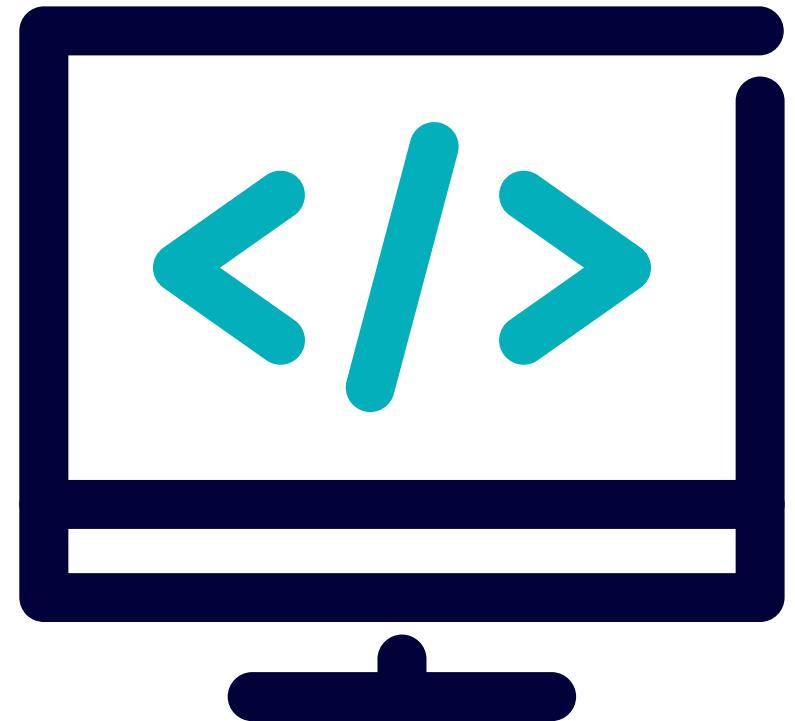
All brokers: `replica.selector.class=org.apache.kafka.common.replica.RackAwareReplicaSelector`

QUESTION: What is the tradeoff?

Lab: Introduction

Please work on **Lab 1a: Introduction**

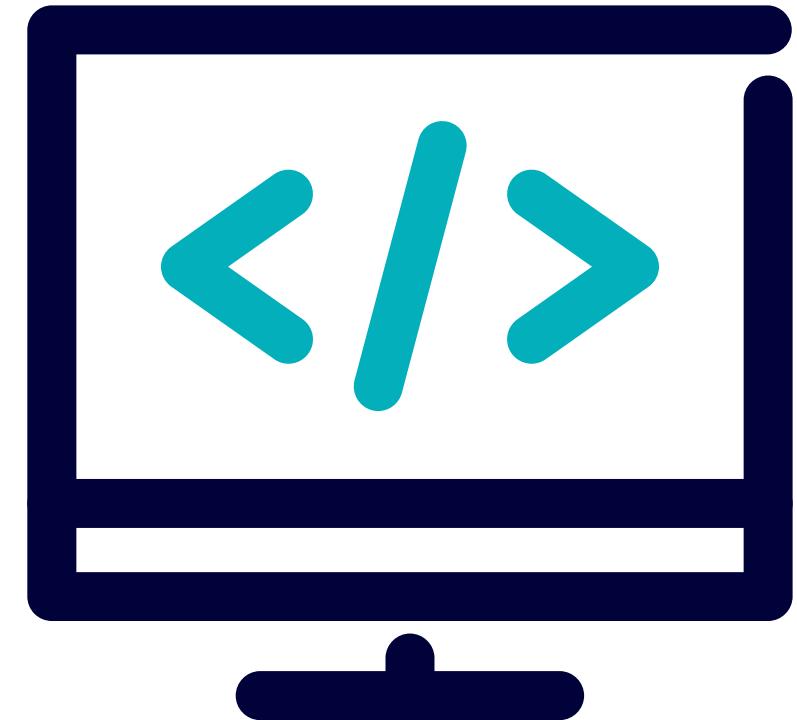
Refer to the Exercise Guide



Lab: Using Kafka's Command Line Tools

Please work on **Lab 1b: Using Kafka's Command Line Tools**

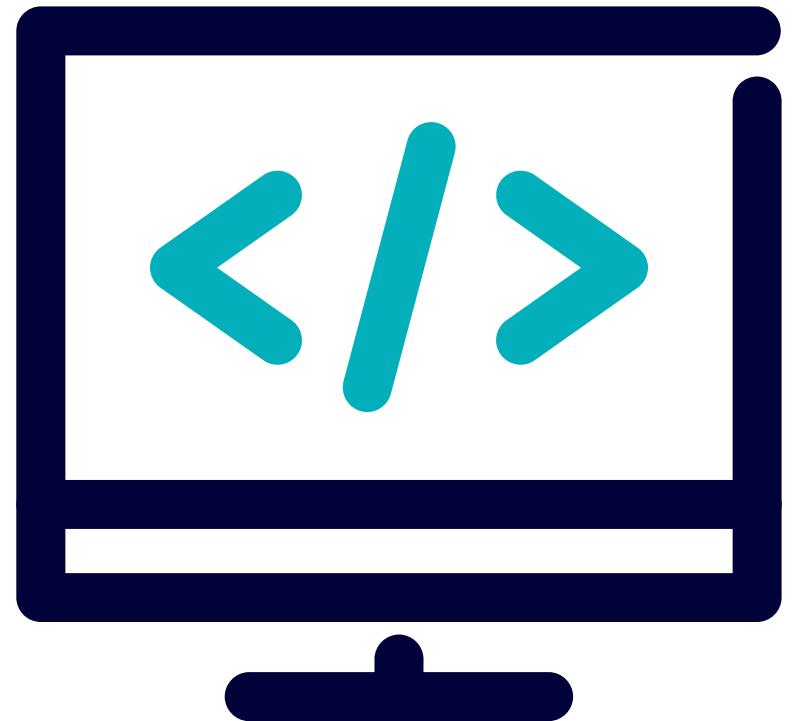
Refer to the Exercise Guide



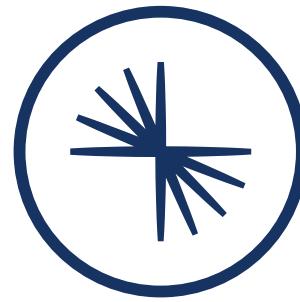
Lab: Producing Records with a Null Key

Please work on **Lab 1c: Producing Records with a Null Key**

Refer to the Exercise Guide



2: Producing Messages Reliably



CONFLUENT
Global Education

Module Overview



This module contains 3 lessons:

- How Can Producers Provide Durability?
- How Can You Guarantee Messages are Written without Duplication?
- How Does Kafka Handle the Notion of Producers Sending Messages in Transactions?

Where this fits in:

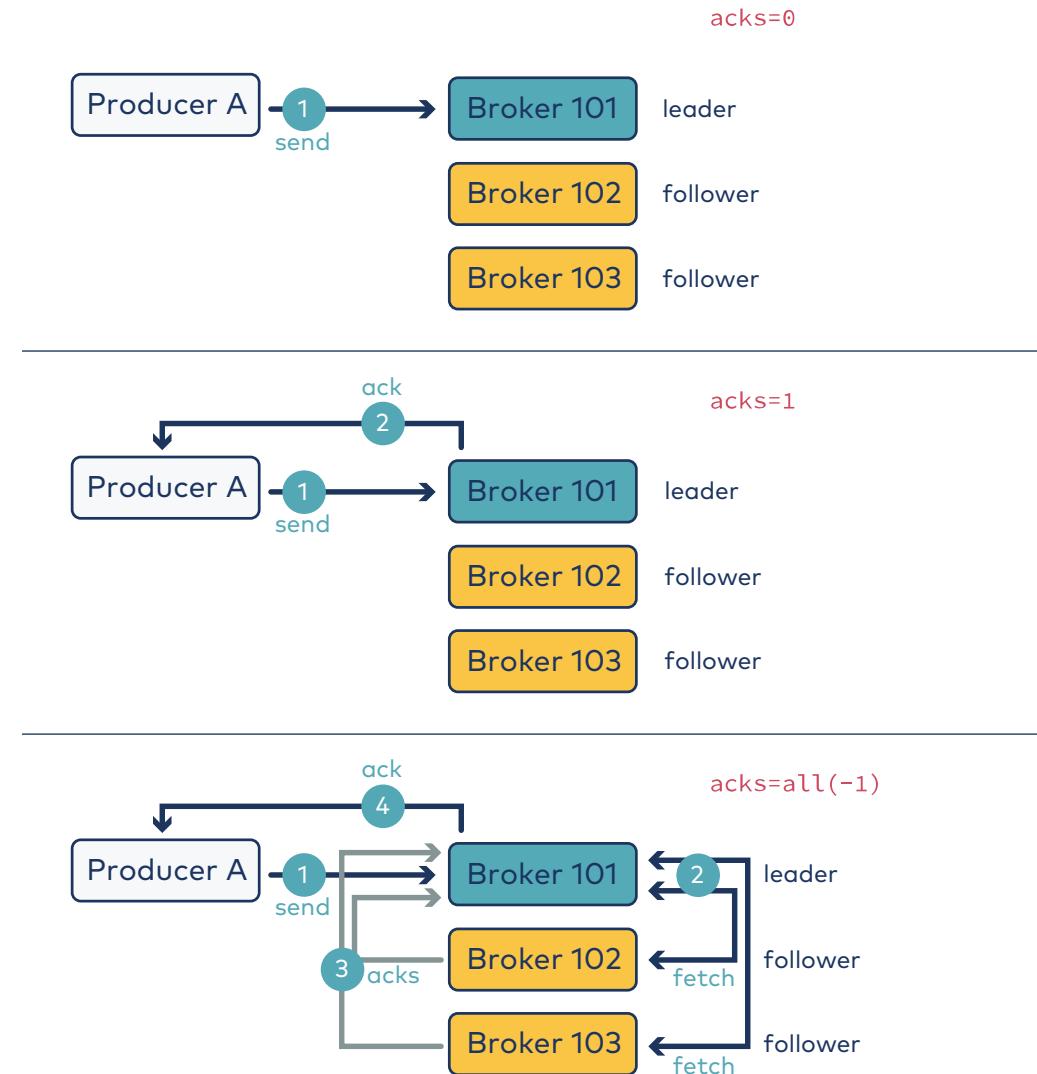
- Hard Prerequisite: Fundamentals course

2a: How Can Producers Provide Durability?

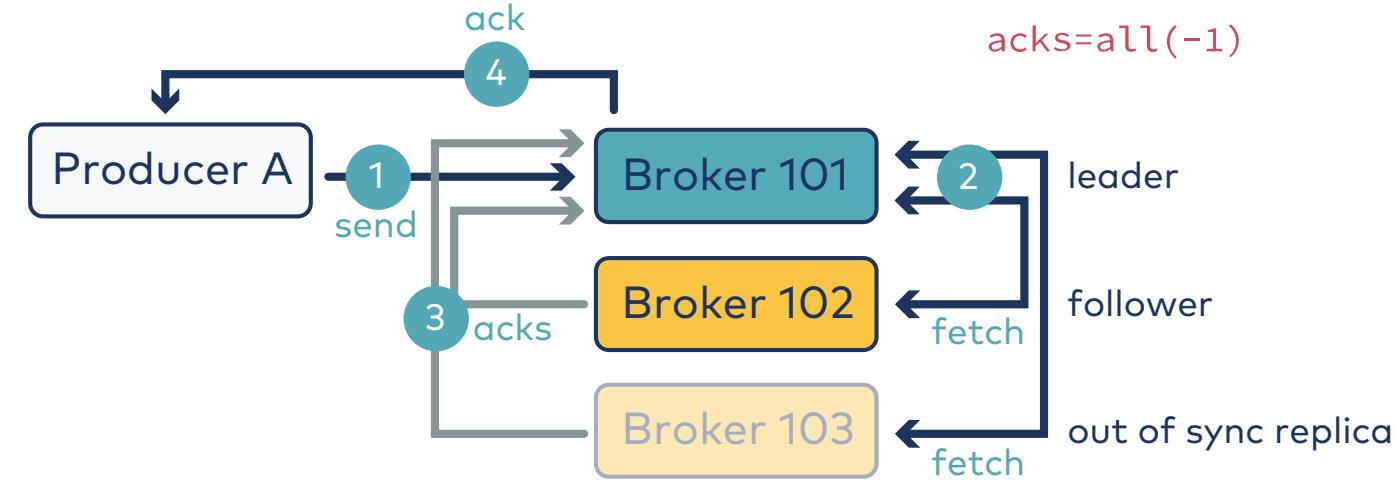
Description

Producer acknowledgements. Retries. Timeouts. Tuning best practices. Multiple in-flight requests and implications.

acks - Three Cases, Ideal Performance



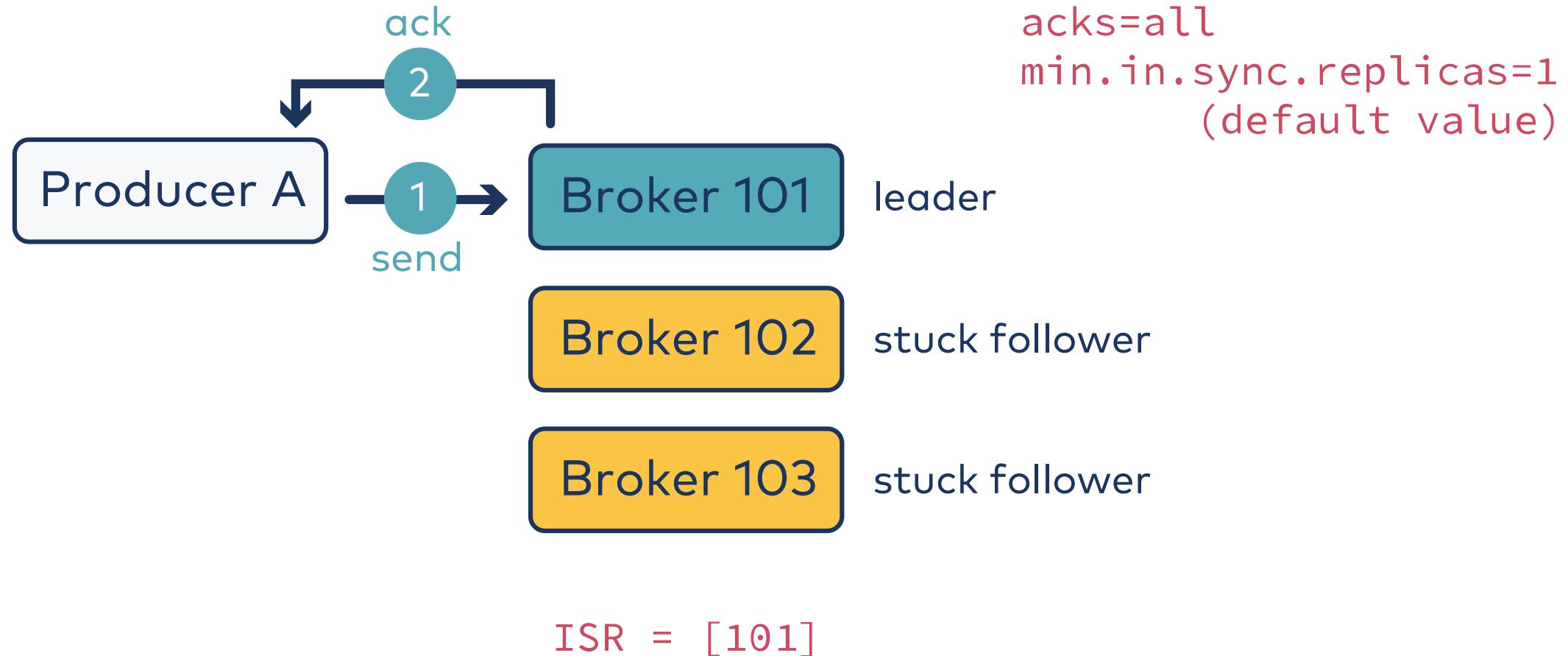
...But not all followers are in-sync...



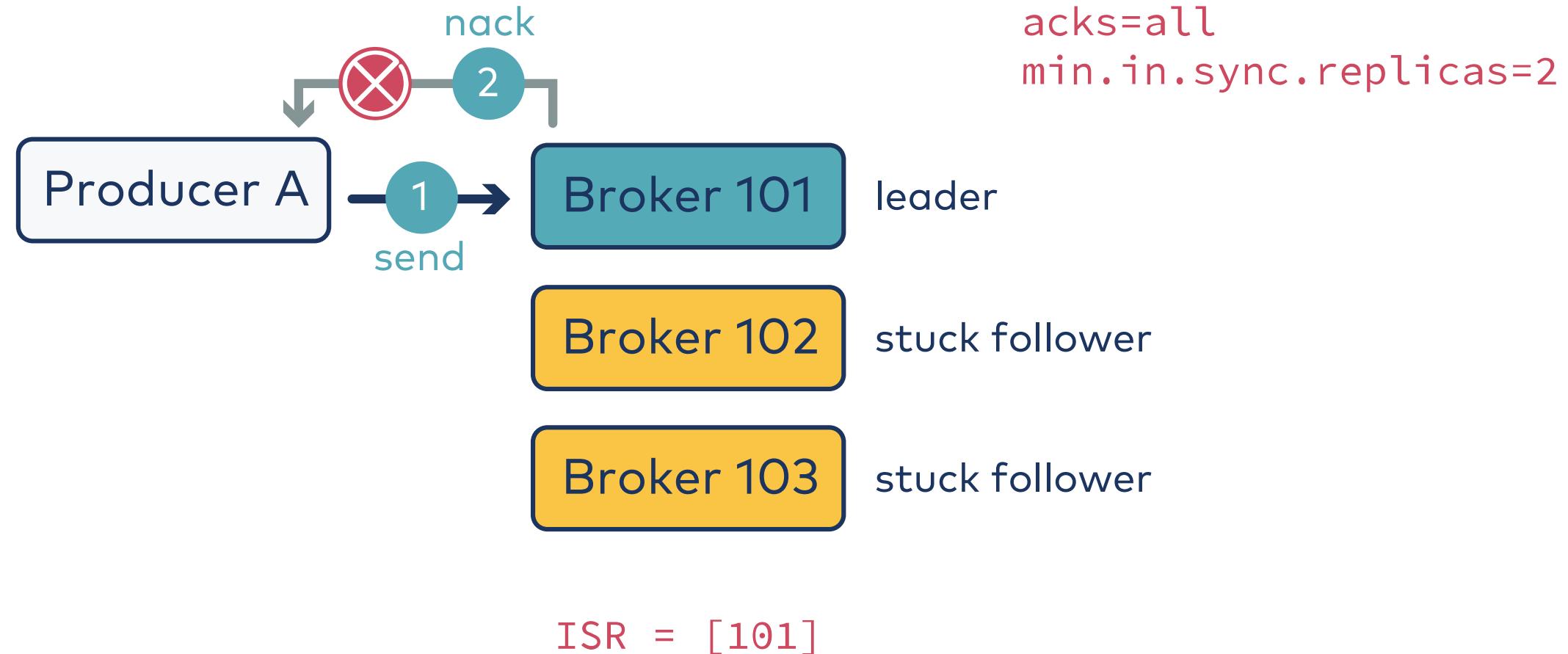
So... when the leader gets a new message and `acks=all`...

1. The leader notes which followers are **in sync** with the leader at the time it receives the message
2. Followers fetch from the leader and send acks to the leader
3. When the leader receives acks from all of the followers from (1), it sends an ack to the producer

What if We Don't Have Any In-Sync Followers?



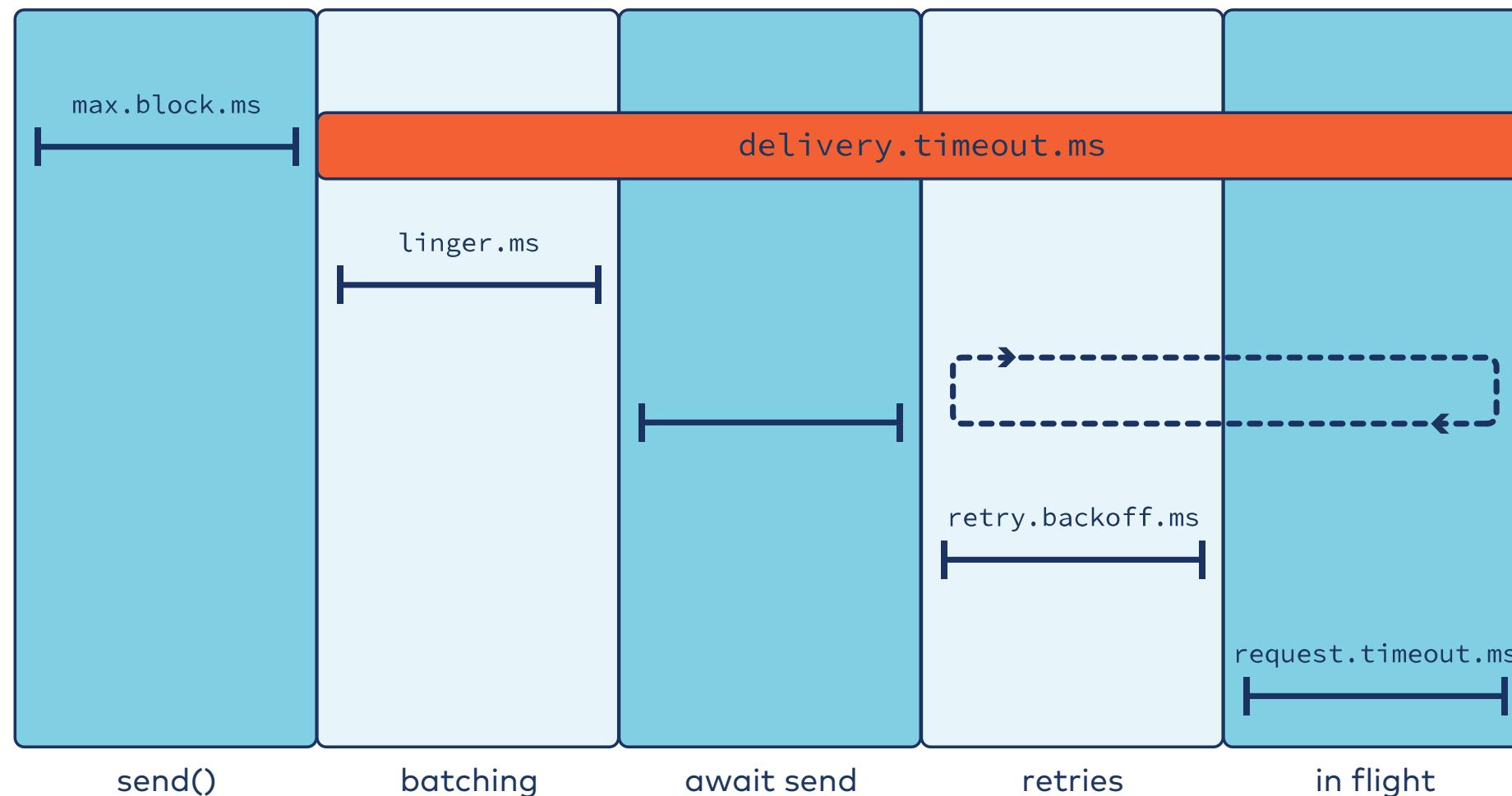
Guaranteeing Meaningful `acks=all`





A Step Beyond

Producer Time Limits



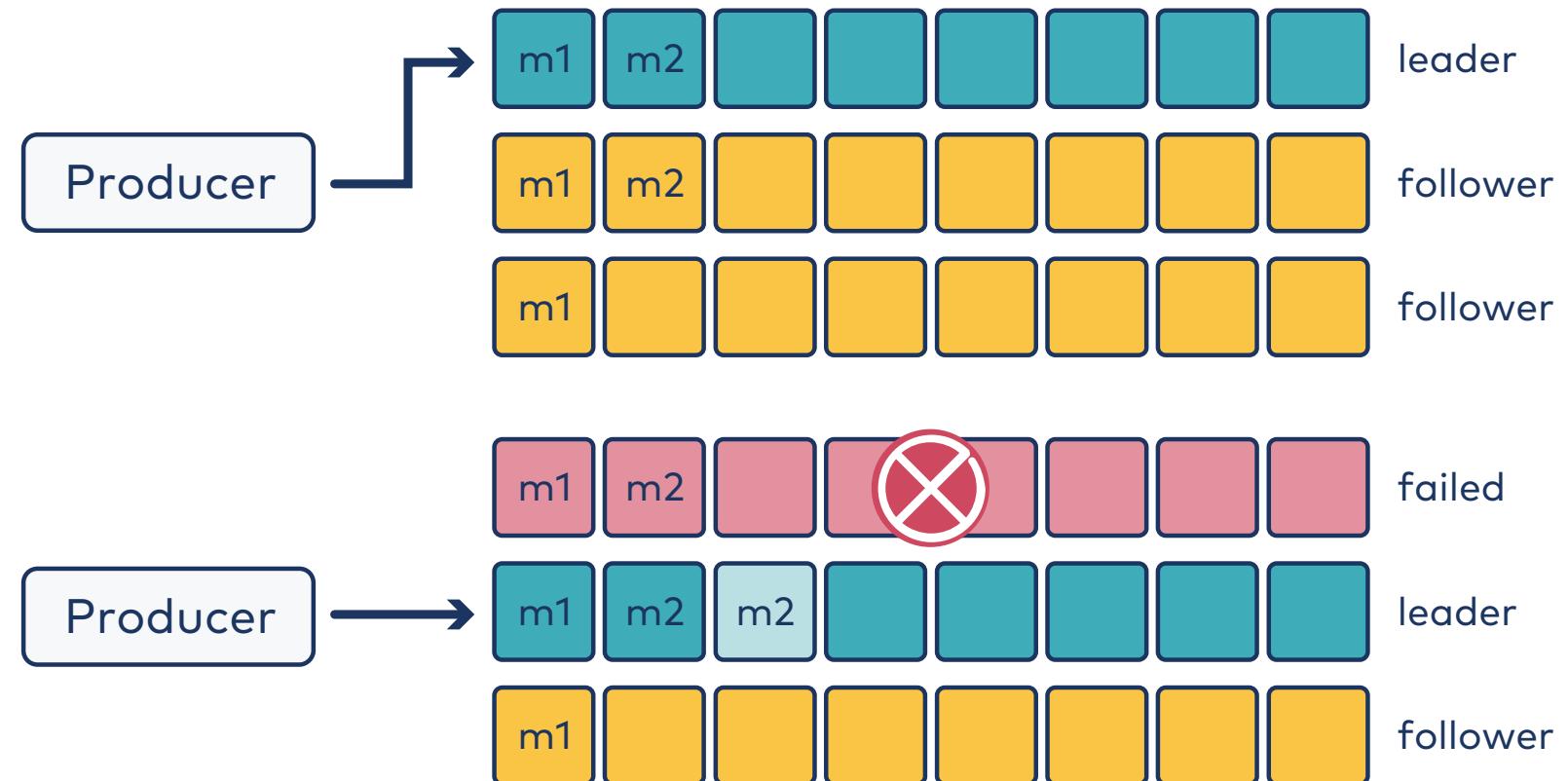
2b: How Can You Guarantee Messages are Written without Duplication?

Description

Idempotent producers: why, metadata, benefits.

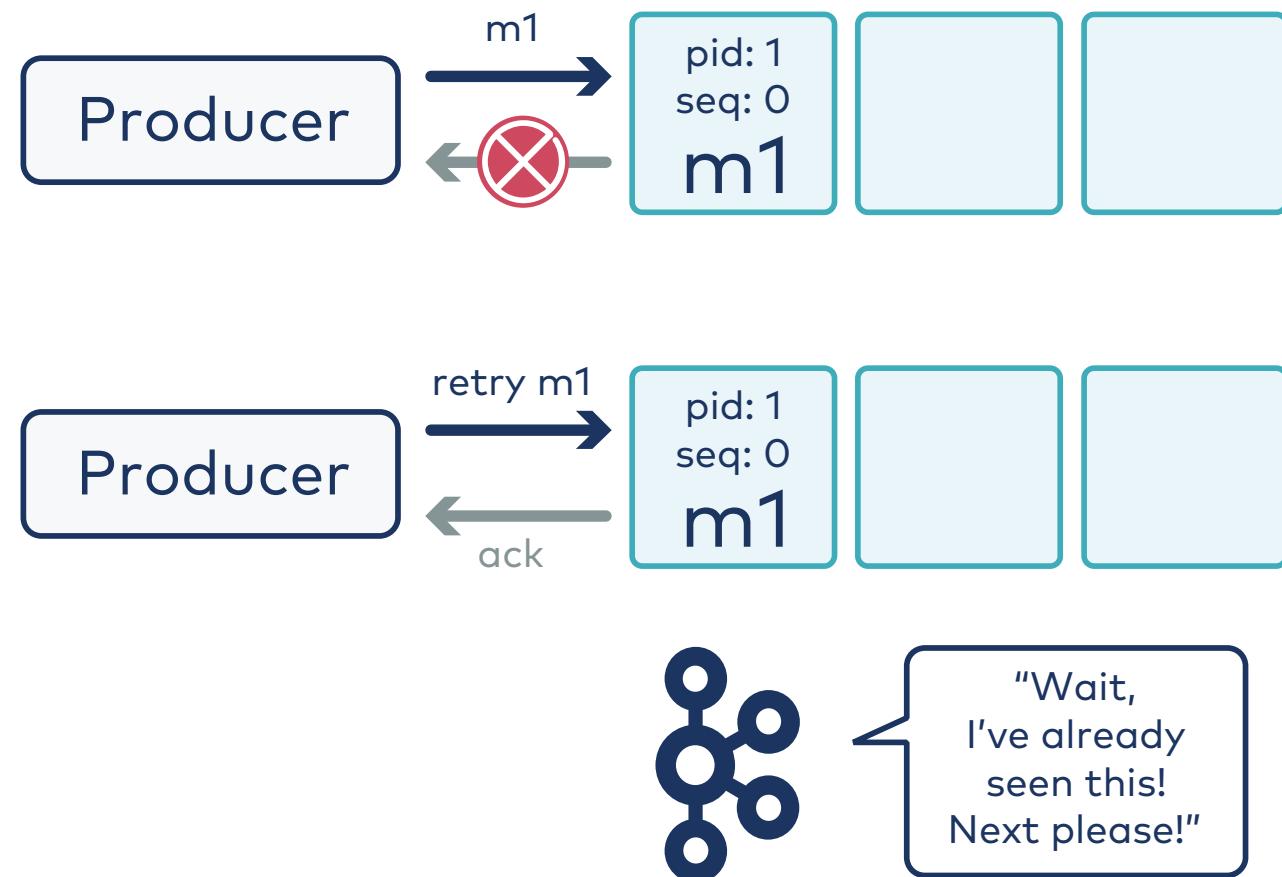
Problem: Producing Duplicates to the Log

- `acks = all`
- `retries > 0`



Solution: Idempotent Producers

- `enable.idempotence = true`
- `acks = all`



2c: How Does Kafka Handle the Notion of Producers Sending Messages in Transactions?

Description

Transactions. How to enable on producers. Metadata. Effects. How brokers handle messages in transactions. How consumers handle transactional messages.

Motivation for Exactly Once Semantics (EOS)

- Write real-time, mission-critical streaming applications that require guarantees that data is processed “exactly once”
- Exactly Once Semantics (EOS) bring strong **transactional** guarantees to Kafka
 - Prevents duplicate messages from being produced by client applications (idempotent producers)
 - Ensures messages in a transaction are all consumed or none are consumed (atomic transactions)
- Sample use cases:
 - tracking ad views for billing
 - processing financial transactions
 - tracking inventory in the supply chain

Enabling Exactly Once Semantics

Two components:

1. Idempotent Producers

- Set `enable.idempotence = true` on the producer

2. Transactions

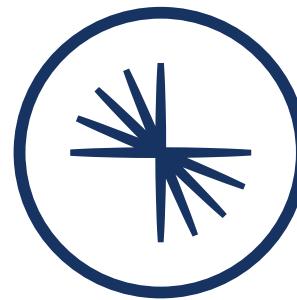
- Producer:

- Set a unique `transactional.id` for each producer
 - Use the transactions API in the producer code

- Consumer:

- Set `isolation.level = read_committed` on the consumer

3: Replicating Data: A Deeper Dive



CONFLUENT
Global Education

Module Overview



This module contains 6 lessons:

- How Does Kafka Determine Which Messages Can be Consumed?
- How Does Kafka Place Replicas and How Can You Control Replication Further?
- How Does Kafka React When a Leader Dies?
- How Does Kafka Track Leadership Changes?
- How Can You Monitor Replication?
- What Are Some Other Replication Considerations?

Where this fits in:

- Hard Prerequisite: Fundamentals course
- Recommended Prerequisite: How Producers Write Messages Reliably

Review—Replica Leaders and Followers

For a given partition:

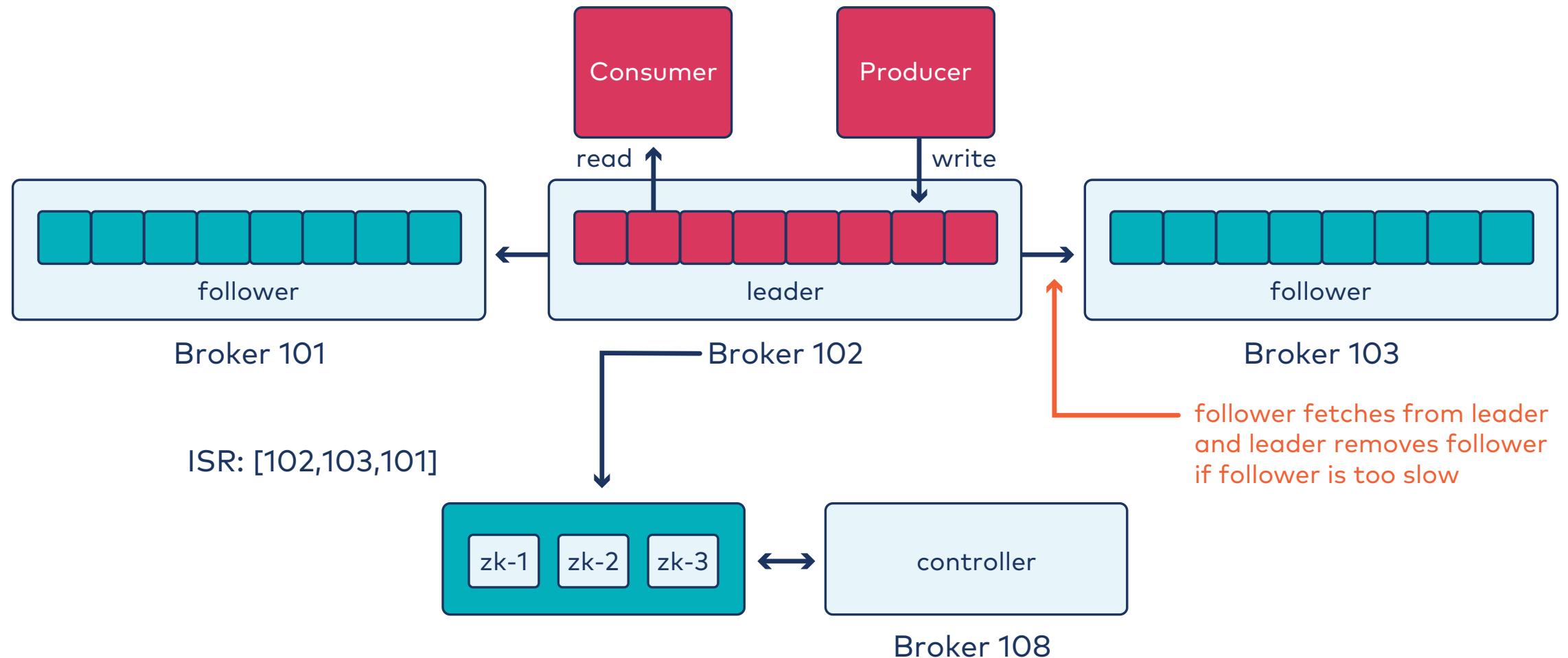
- Leader
 - Accepts all writes and reads for a specific partition
 - Monitors health of follower replicas
- Followers
 - Attempt to keep up with the leader
 - Provide fault tolerance

Configuring Replication Factor

- Increase the replication factor for better durability guarantees
- For auto-created topics
 - `default.replication.factor` (Default: 1)
 - Configure in `server.properties` on each broker
- For manually-created topics:

```
$ kafka-topics \
  --bootstrap-server kafka-1:9092 \
  --create --topic my_topic \
  --replication-factor 3 \
  --partitions 2
```

In-Sync Replicas



3a: How Does Kafka Determine Which Messages Can be Consumed?

Description

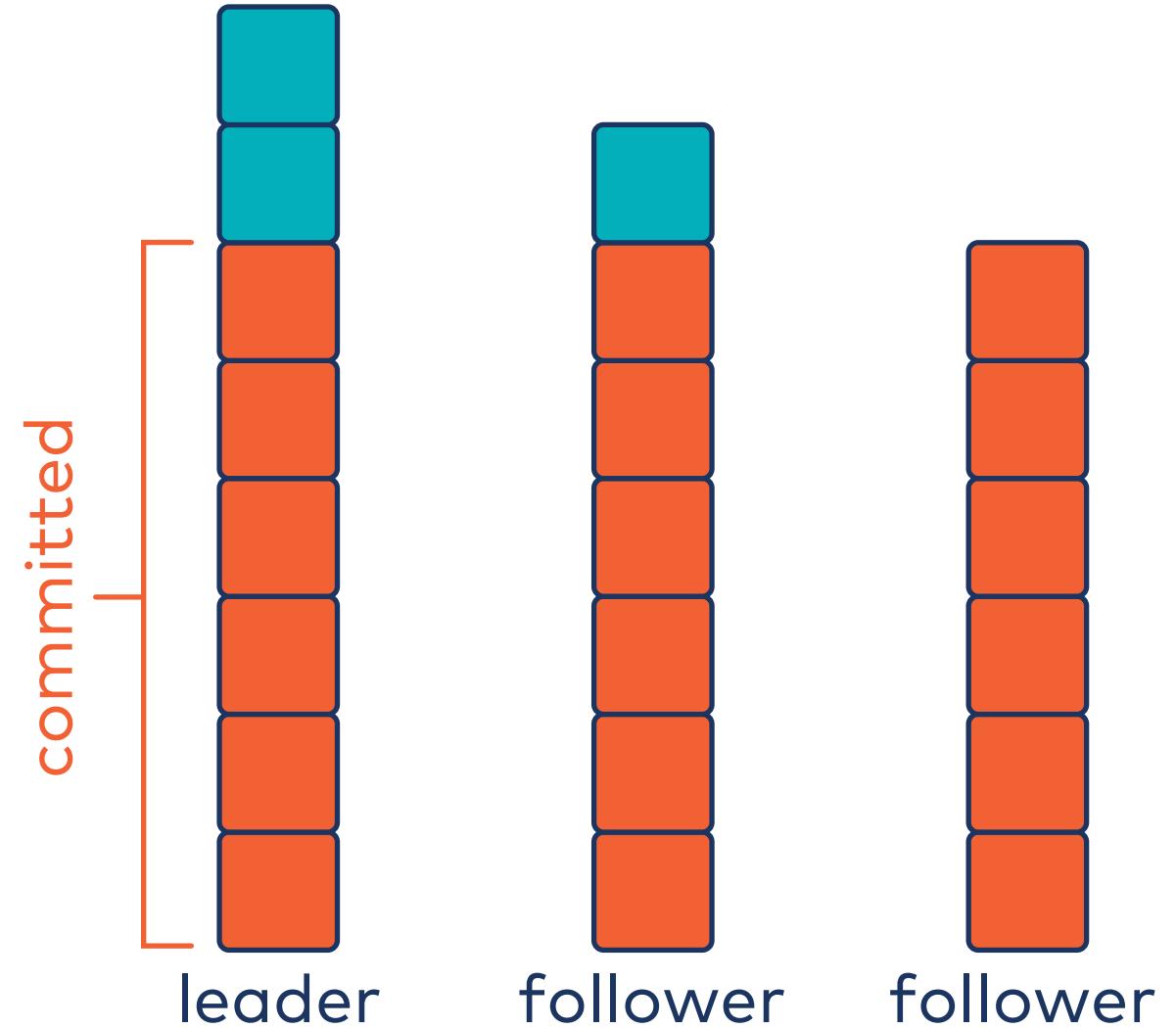
High water mark. Committing messages.

High Water Mark / Committed Messages

- Consumers can only read messages that are **committed**.
- The **high water mark** (HWM) separates messages in a log that are committed from those that are not yet committed.
 - HWM == offset of most-recently committed message
- HWM is per-replica.

What Does "Committed" Mean?

- A message is called "committed" when it is received by all the replicas in the ISR list
- Consumers can only read committed messages
- The leader decides when to commit a message
 - Committed state is checkpointed to disk

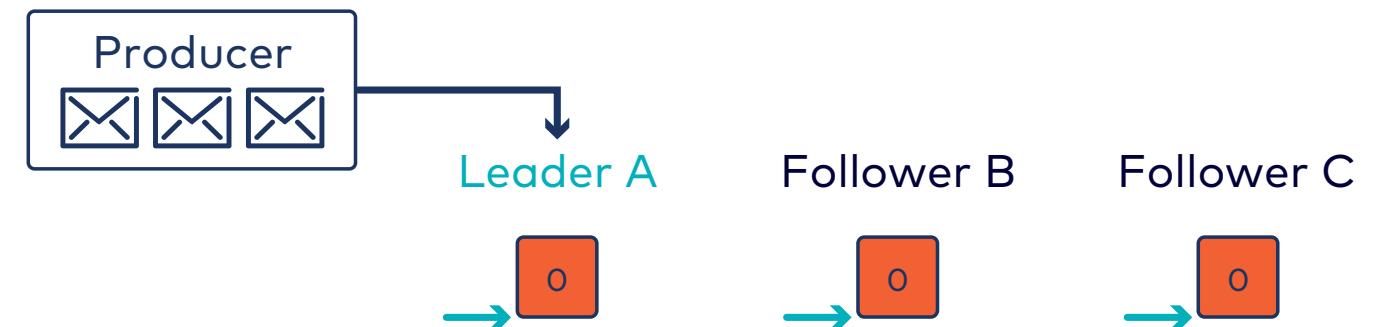


Refining ISR Definition

- Before we said a replica that has all of the messages the leader has is considered an **in-sync replica**
- This isn't quite true...
- More accurately, a replica is an **in-sync replica** iff it has all of the messages the leader has *up to and including the high water mark*. So...
 - A leader may have messages that are not committed
 - A follower that does not have some or all of the not-committed messages is still an **in-sync replica**

Committing Messages with Replicas (1)

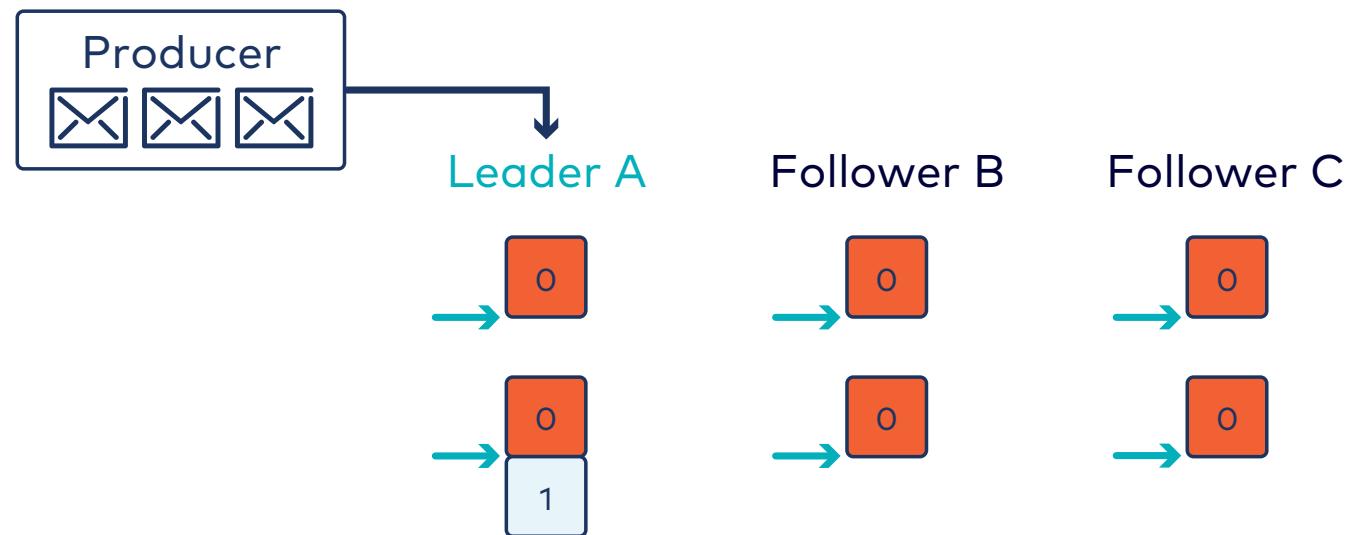
1 Initial state



Committing Messages with Replicas (2)

1 Initial state

2 A appends new message at offset 1

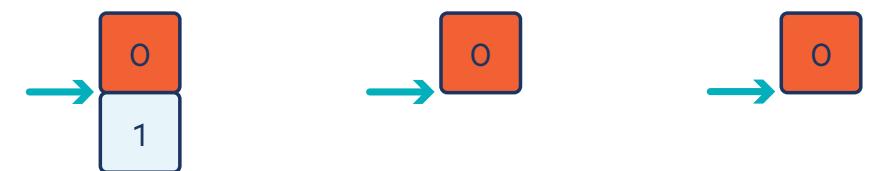


Committing Messages with Replicas (3)

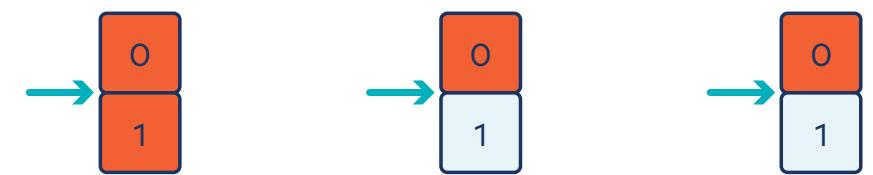
1 Initial state



2 A appends new message at offset 1



3 B and C fetch and append message at offset 1



Committing Messages with Replicas (4)

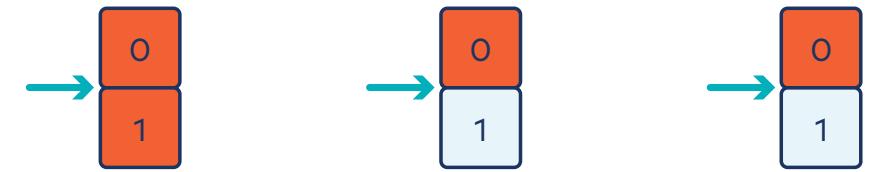
1 Initial state



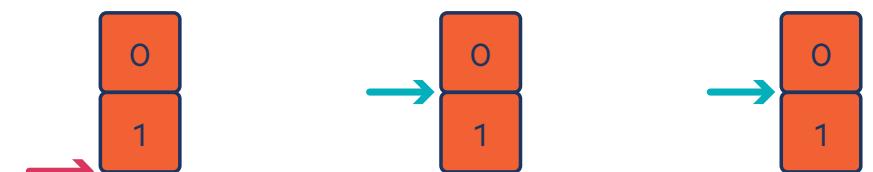
2 A appends new message at offset 1



3 B and C fetch and append message at offset 1

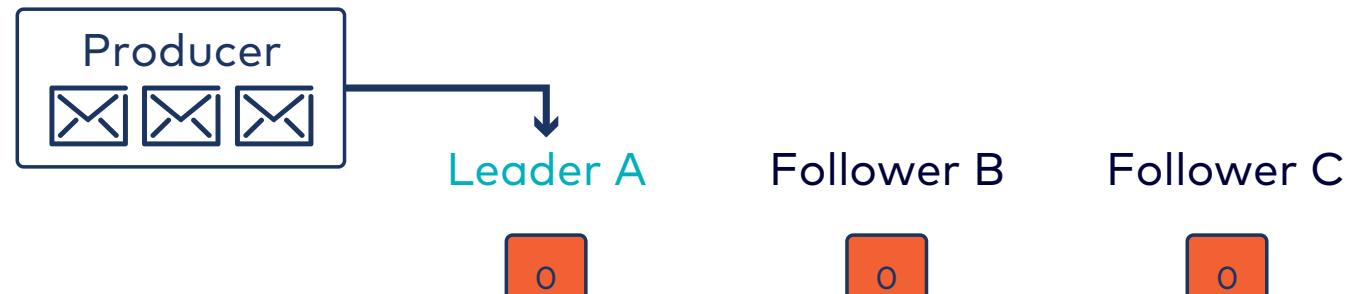


4 B and C fetch null at offset 2; A advances high water mark

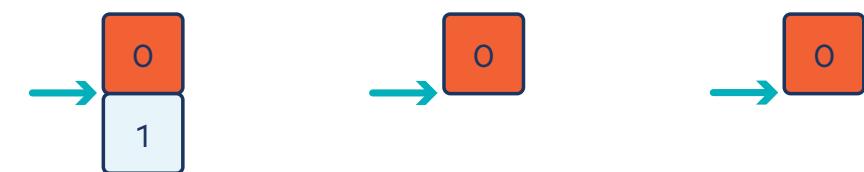


Committing Messages with Replicas (5)

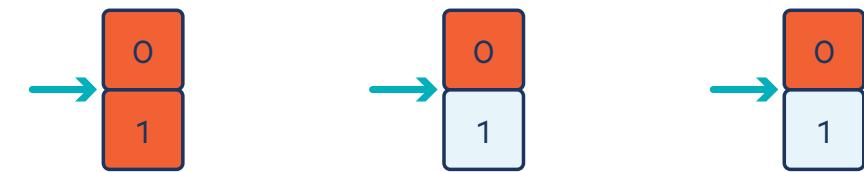
1 Initial state



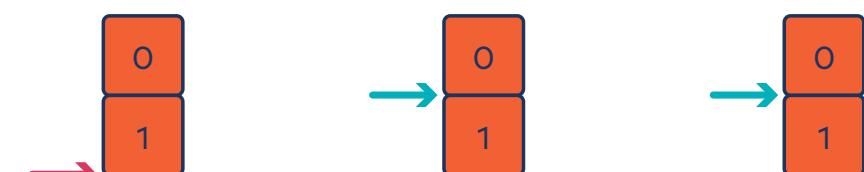
2 A appends new message at offset 1



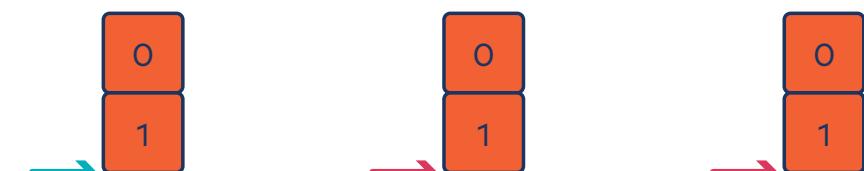
3 B and C fetch and append message at offset 1



4 B and C fetch null at offset 2; A advances high water mark



5 B and C fetch null at offset 2 again and receive new high water mark



3b: How Does Kafka Place Replicas and How Can You Control Replication Further?

Description

Replica placement. Preferred replicas. Under-replicated and offline partitions.

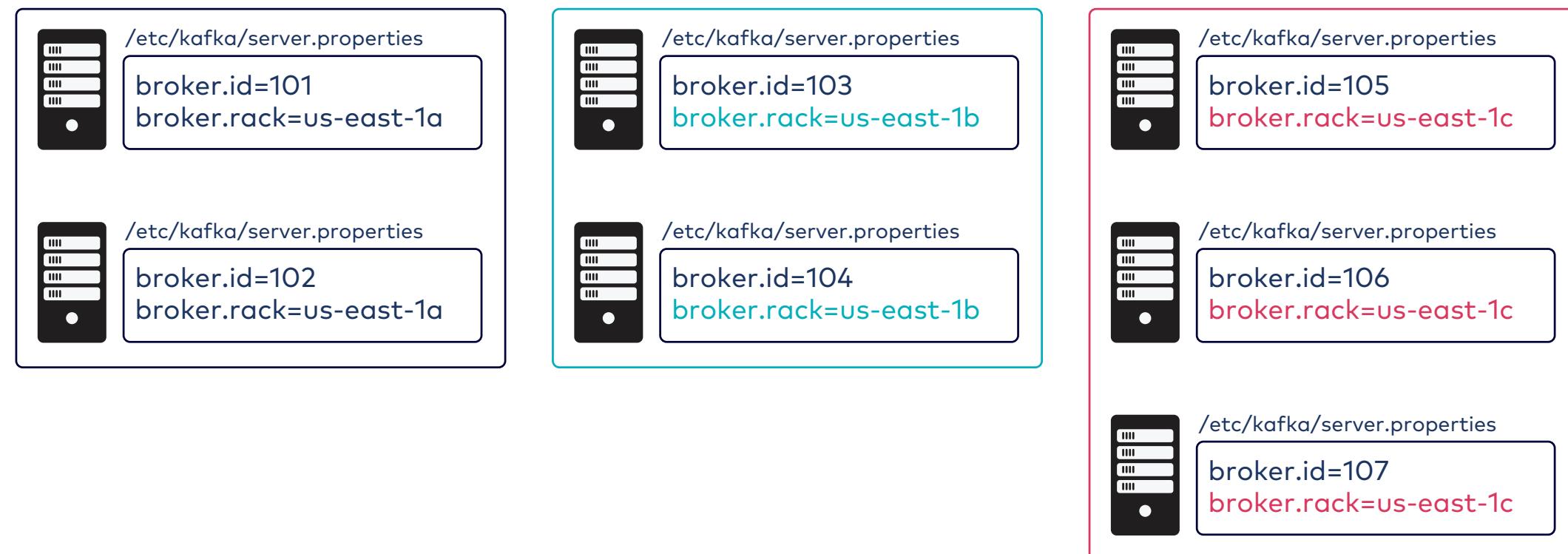
Replica Placement

Kafka...

- places replicas on brokers
- tries to balance the placement of replicas across brokers

Rack Awareness

- Configure `broker.rack` in `server.properties`
- Specify the same "rack name" for brokers in the same Availability Zone
- Replicas will be balanced across racks with best effort
- Only enforced on topic creation or with Confluent Auto Data Balancer
- Feature is **all or nothing**



Balancing Leadership and Preferred Replicas

Kafka attempts to balance brokers in terms of how many leaders are on one broker

For each partition...

- a broker is designated as the ideal place for it - the *preferred replica*
- failover can move where the leader is
- Kafka monitors how many leaders are on the ideal broker
 - When a threshold is exceeded, leaders are moved

Viewing Partition Placement Across Cluster (1)

The same data tracked from the CLI with:

```
$ kafka-topics \
  --bootstrap-server kafka-1:9092 \
  --describe \
  --topic i-love-kafka
```

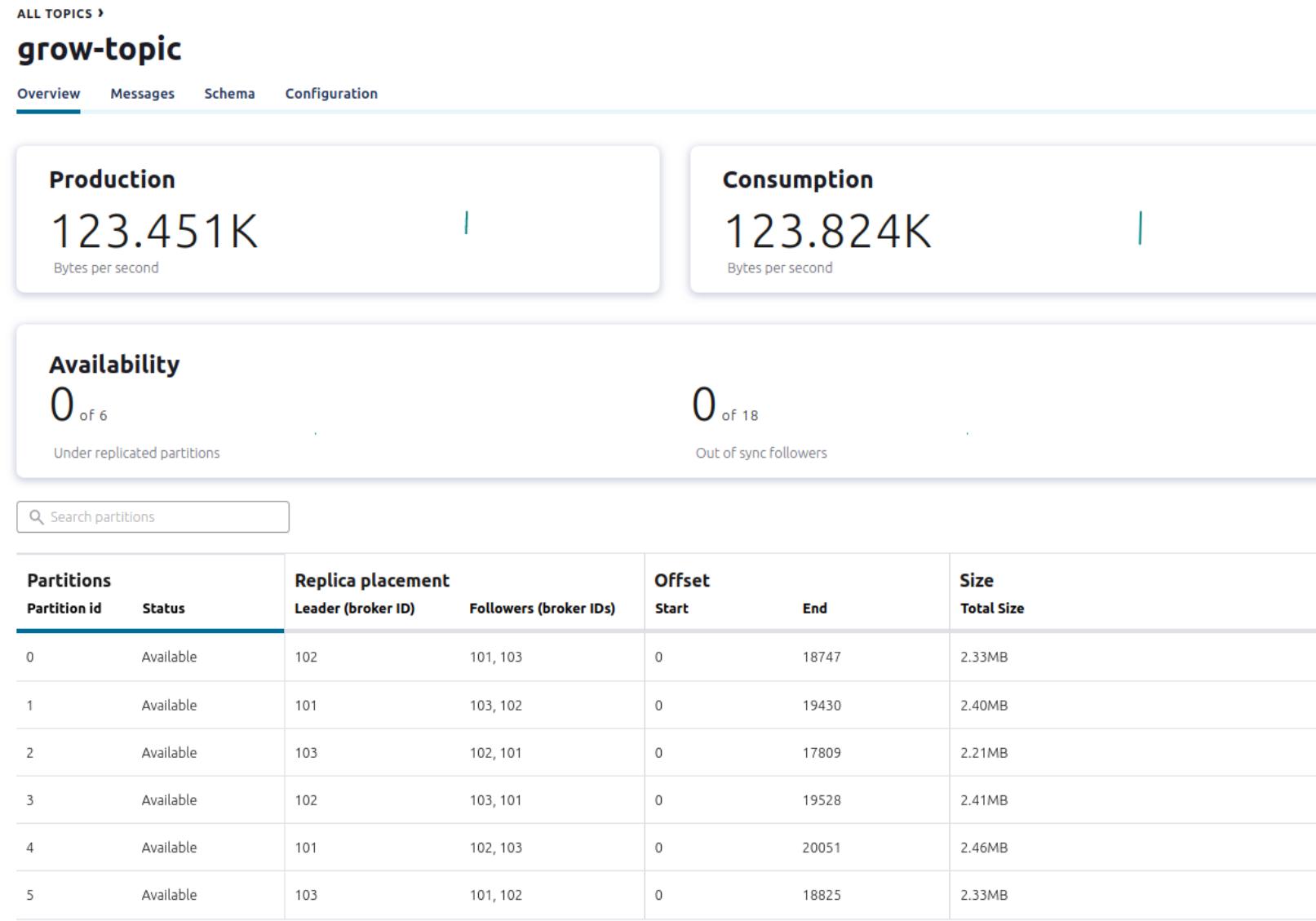
Topic:i-love-kafka PartitionCount:3	ReplicationFactor:3 Configs:
Topic: i-love-kafka Partition: 0	Leader: 101 Replicas: 101 ,102,103 Isr: 101,102,103
Topic: i-love-kafka Partition: 1	Leader: 103 Replicas: 103 ,101,102 Isr: 103,101,102
Topic: i-love-kafka Partition: 2	Leader: 102 Replicas: 102 ,103,101 Isr: 102,103,101



Preferred replicas highlighted in bold

Viewing Partition Placement Across Cluster (2)

Confluent Control Center provides per-topic replica view:



Two More Important Definitions

Under-Replicated Partition

partition where the number of replicas != replication factor

Offline partition

partition for which no leader exists



You can monitor both of these. More on that in the appendix.

Activity: Analyzing Replication Factor & Partition Status



Suppose we have the following replica placement - and all replicas are in their respective ISR lists:

broker 101	broker 102	broker 103	broker 104
$p_{0,L}$	$p_{1,L}$	$p_{2,L}$	$p_{1,F0}$
$p_{2,F0}$	$p_{0,F0}$	$p_{1,F1}$	$p_{2,F1}$

Then:

1. What can you deduce about replication factors from the figure?
2. How many topics are in play?
3. Suppose broker 102 goes down. Normal behavior happens. What new designations apply to any (which) partition(s)?
4. Suppose 102 is still down and 101 goes down too. What new designations apply to any (which) partition(s) now?

3c: How Does Kafka React When a Leader Dies?

Description

Controller. Leader election. Unclean leader election.

The Controller

- A process that runs on one broker in a cluster
- Facilitates leader election
- Monitors broker liveness
- Communicates leader and replica information to other brokers
- Pushes partition metadata to ZooKeeper and brokers

What if...

We have replication to handle the case where the leader is on a broker that dies.

The controller runs on a broker.

Do you see any problems with this?

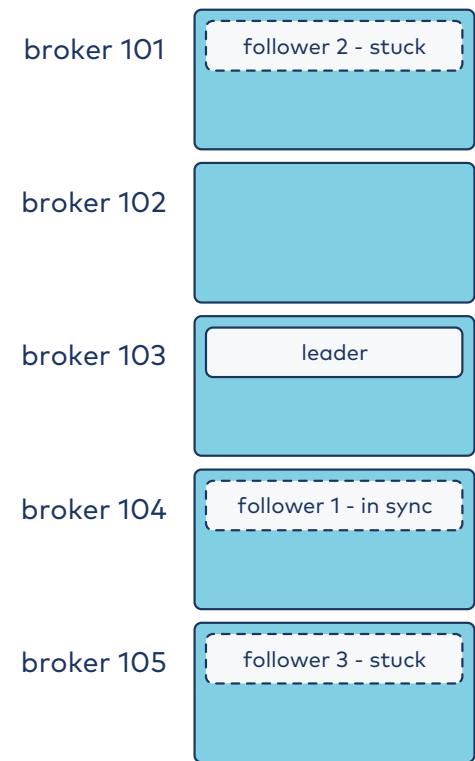
Scenario

Consider the scenario at right.

Suppose broker 103 goes down.

We know Kafka will automatically do leader election and pick a follower from the ISR to become the leader. But...

Suppose broker 103 remains down, nothing has changed, and broker 104 also goes down. Now what?



Unclean Leaders

- Kafka will always choose a follower from the ISR
- What if you want to allow followers that aren't in the ISR to become the leader?
 - Topic configuration property: `unclean.leader.election.enable`
 - Determines whether a new leader can be elected even if it is not in-sync, if there is no other choice
 - Can result in data loss if enabled (Default: `false`)

3d: How Does Kafka Track Leadership Changes?

Description

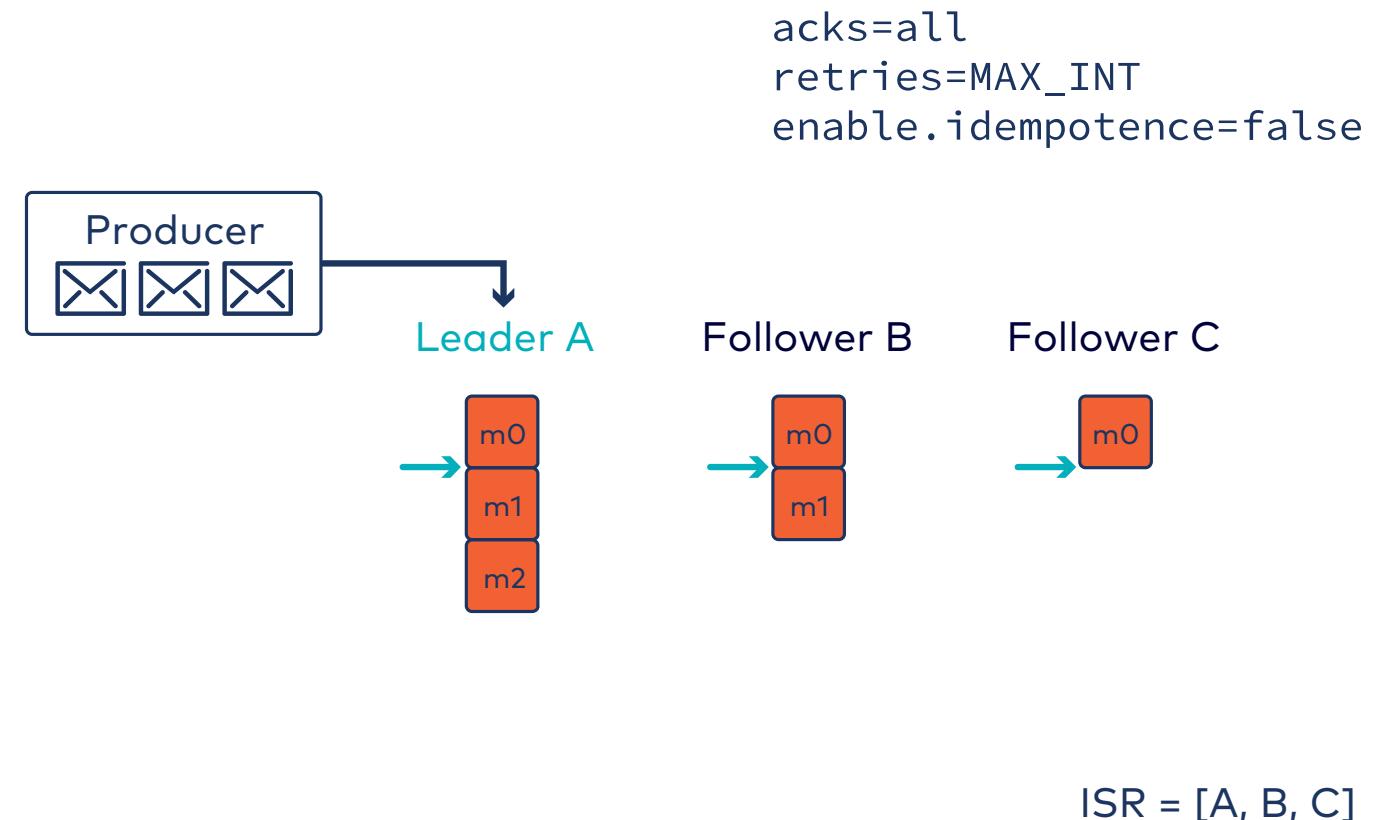
Leader epoch. Broker recovery.

Leader Epoch

- Marks offsets where new leaders are elected
- Used during broker recovery to truncate messages to a checkpoint and then follow the current leader
- Checkpointed to disk in `leader-epoch-checkpoint`
- Starts at `0`, increments
- What is stored:
 - leader epoch, offset at epoch pairs

Example of Replica Recovery (1)

Initial state. Only m0 is committed.



Example of Replica Recovery (2)

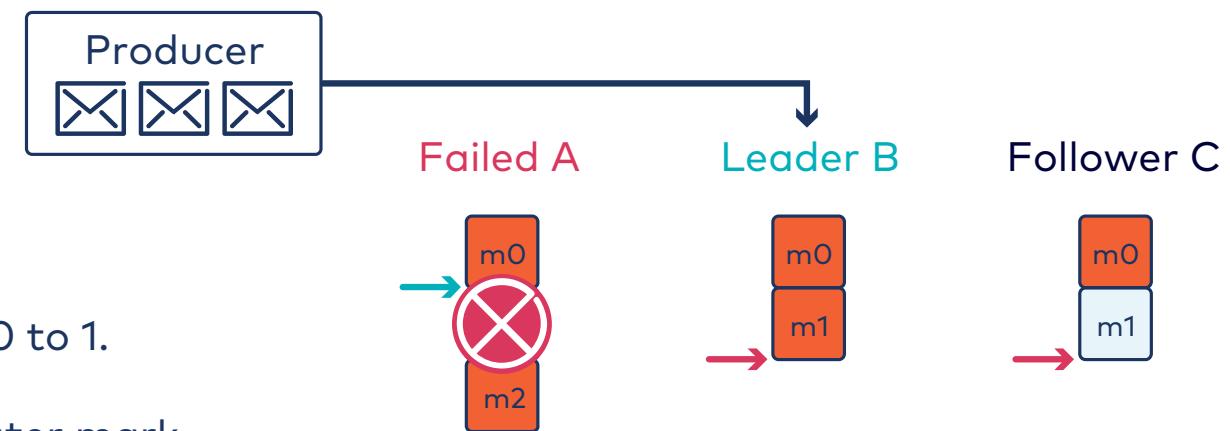
A fails before committing m1 and m2.

B becomes leader. Leader epoch increments from 0 to 1.

C follows B and fetches m1. B advances its high water mark.

C fetches again to advance its high water mark.

```
acks=all  
retries=MAX_INT  
enable.idempotence=false
```



ISR = [B, C]

Example of Replica Recovery (3)

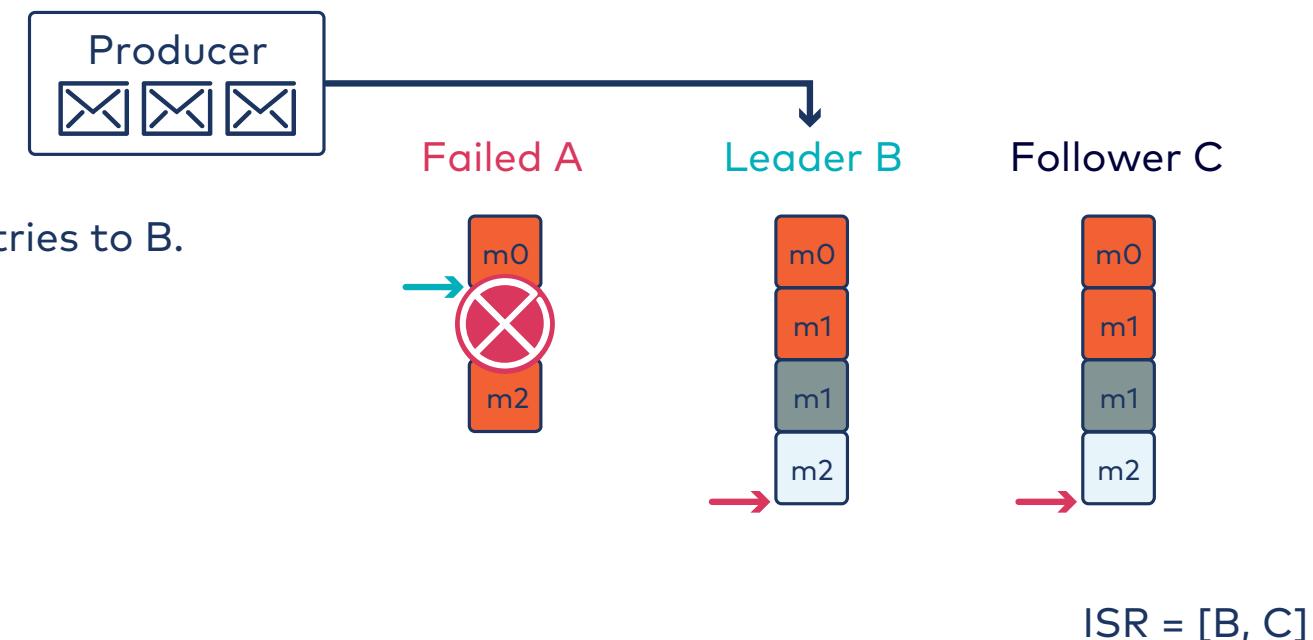
Producer never received acks for m1 or m2, so it retries to B.

Idempotence is disabled, so m1 is duplicated.

C fetches once to get m1 (dup) and m2.

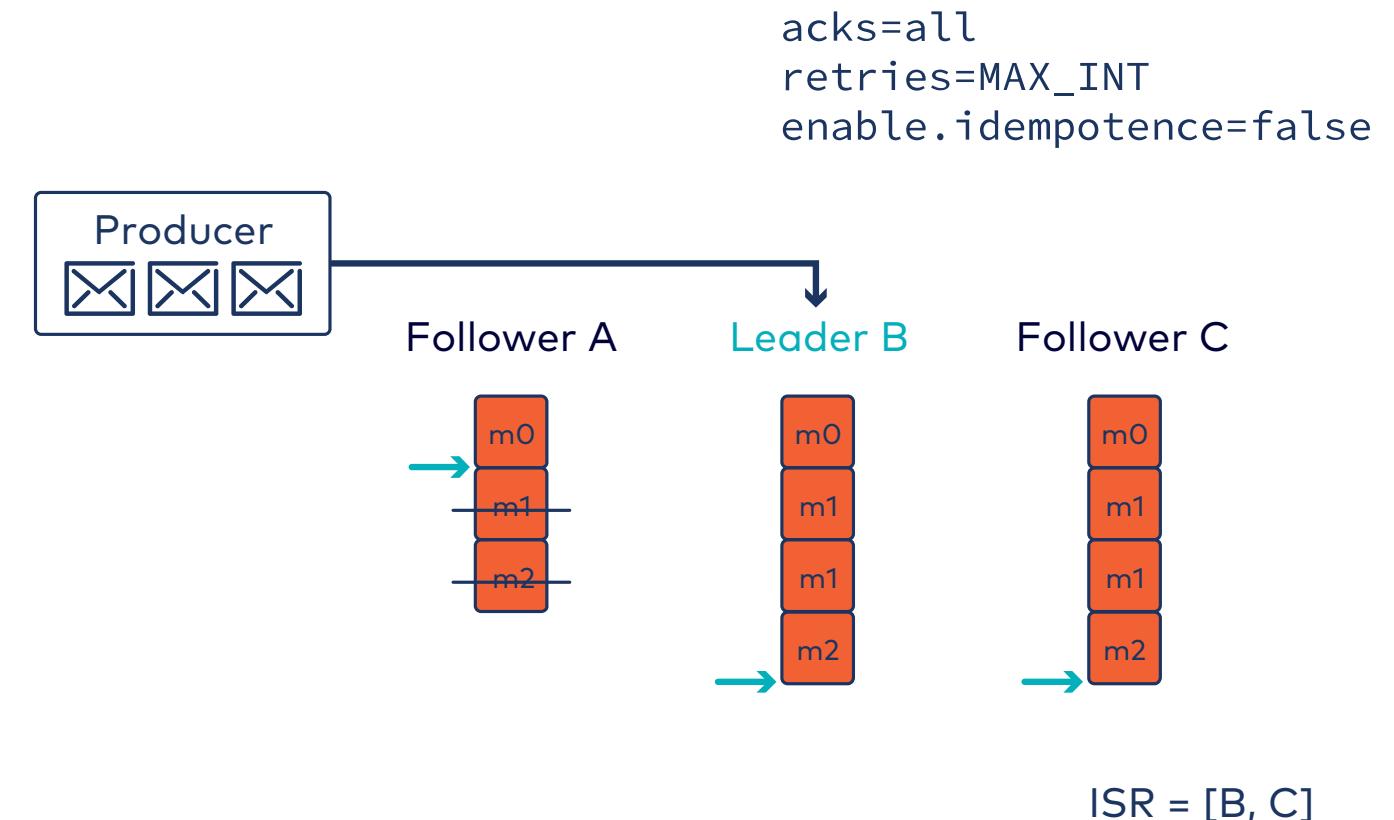
C fetches again to advance high water mark.

```
acks=all  
retries=MAX_INT  
enable.idempotence=false
```

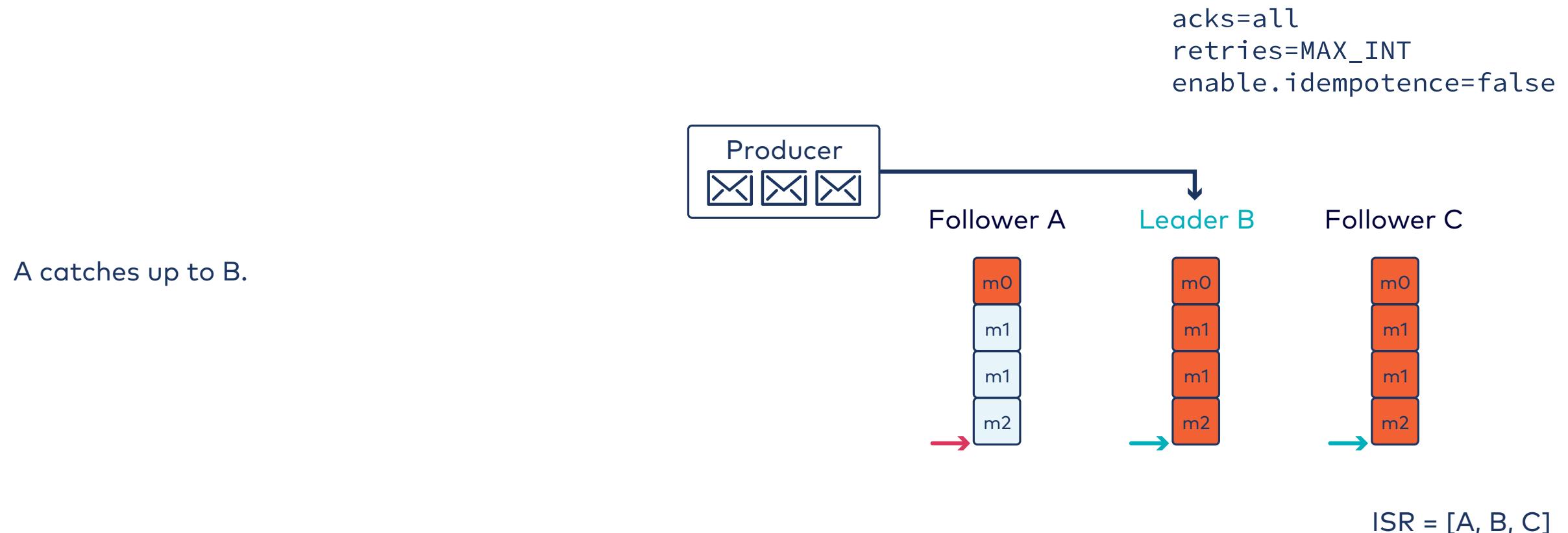


Example of Replica Recovery (4)

- A restarts and follows B.
- A receives metadata from Controller.
- A fetches leader epoch from Leader B.
- A truncates to place where leadership changed.



Example of Replica Recovery (5)

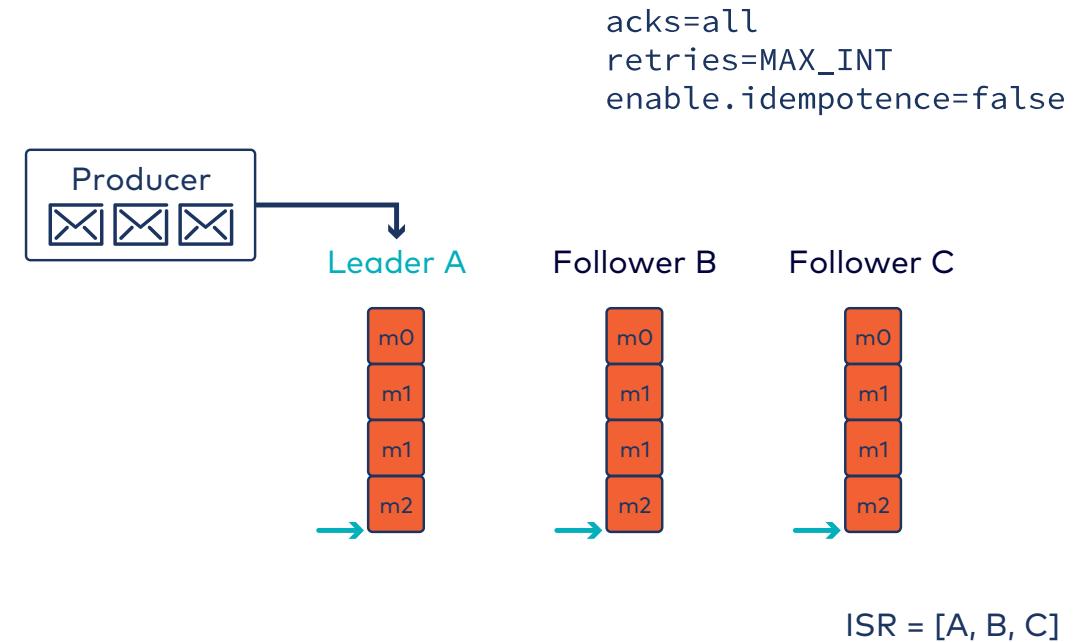


Example of Replica Recovery (6)

After a while, Controller notices imbalanced leadership and re-elects the preferred replica A.

Leader epoch increments from 1 to 2.

B and C now follow A.



3e: What Are Some Other Replication Considerations?

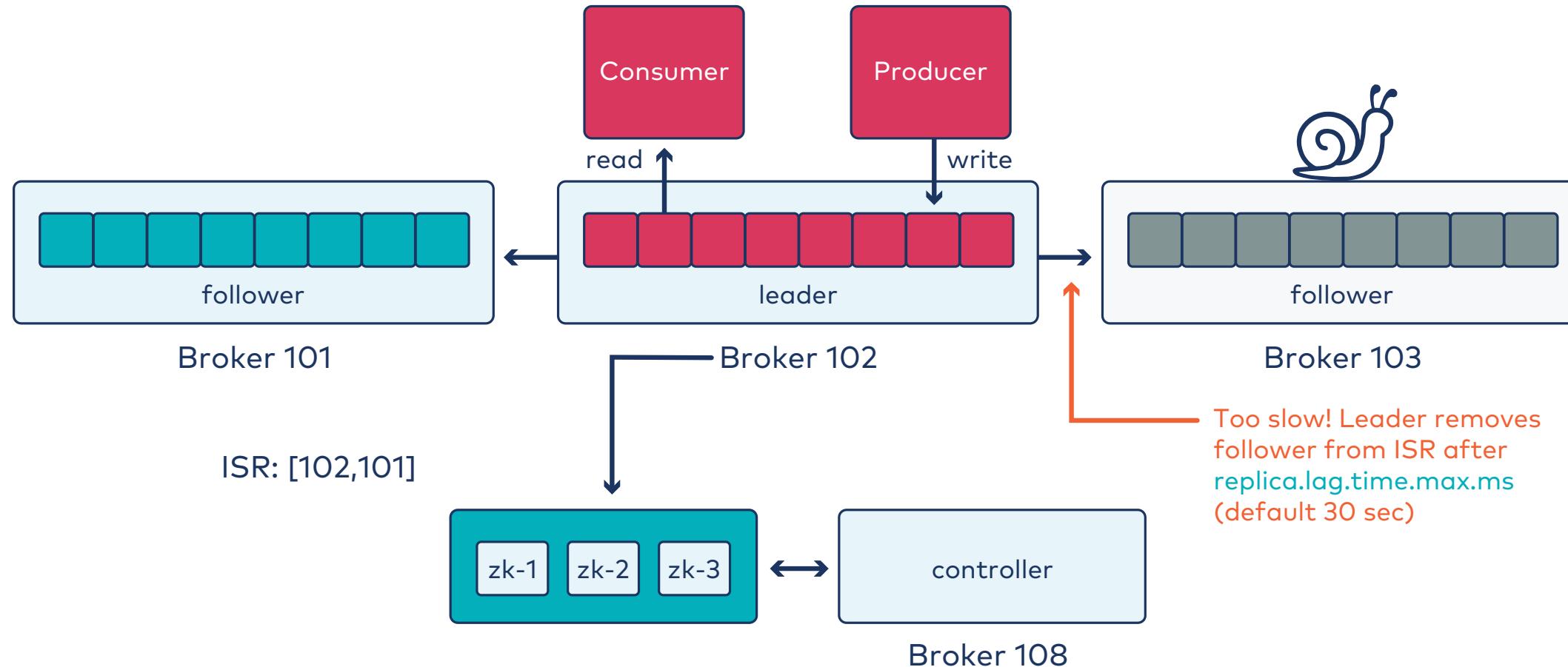
Description

Review of replication details. Slow replicas.

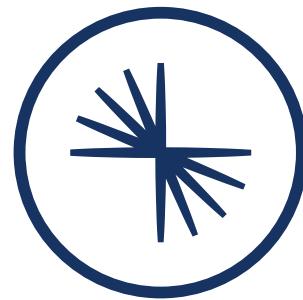
Maintaining the List of In-Sync Replicas

- A message is considered committed if it is received by every replica in the ISR list
- Leader persists changes to ISR list in ZooKeeper, which is monitored by the cluster's controller
- If a follower fails:
 - It is dropped from the ISR list by the leader
 - Leader commits using the new ISRs
- If a leader fails:
 - Controller elects new leader from followers
 - Controller pushes new leader and ISR to ZooKeeper first and then to brokers for local caching

Detecting Slow Replicas



4: Providing Durability in Other Ways



CONFLUENT
Global Education

Module Overview



This module contains 2 lessons:

- How Does Kafka Organize Files to Store Partition Data?
- How Does Kafka Maintain Consumer Offsets?

Where this fits in:

- Hard Prerequisite: Fundamentals course
- Recommended Prerequisite: Replication & Replica Placement

4a: How Does Kafka Organize Files to Store Partition Data?

Description

Logs vs. segments. Details of files created per partition and per segment. Rolling of segments.

Log File Subdirectories

- Kafka **log segment** files are sometimes called data files.
- Each broker has one or more data directories specified in the `server.properties` file, e.g.,

```
log.dirs = /var/lib/kafka/data-a, /var/lib/kafka/data-b, /var/lib/kafka/data-c
```

- Each topic-partition has a separate subdirectory
 - e.g., `/var/lib/kafka/data-a/my_topic-0` for partition 0 of topic `my_topic`
- Brokers detect log directory failures and notify controller

File Types Per Topic Partition

- Per log segment
 - `.log`Log segment file holds the messages and metadata
 - `.index`Index file that maps message offsets to their byte position in the log file
 - `.timeindex`Time-based index file that maps message timestamps to their offset number
- Additional per log segment for *certain producers*:
 - `.snapshot`If using idempotent producers, checkpoints PID and seq #
 - `.txnindex`If using transactional producers, indexes aborted transactions
- Per partition
 - `leader-epoch-checkpoint`Maps the leader epoch to its corresponding start offset

Example of Log Files

Example of one broker's subdirectory for topic `my_topic` with partition `0`

```
$ ls /var/lib/kafka/data-b/my_topic-0
000000000000283423.index
000000000000283423.timeindex
000000000000283423.log
...
0000000000008296402.index
0000000000008296402.timeindex
0000000000008296402.log
leader-epoch-checkpoint
```

Log Segment Properties

- Each `.log` filename is equal to the offset of the first message it contains, e.g.,
 - `00000000000049288237.log`
- Messages are written to the **active** segment
- The active segment **rolls** to a new segment file if any are exceeded:
 - `log.segment.bytes` (Default: 1GB)
 - `log.roll.ms` (Default: 168 hours = 1 week)
 - `log.index.size.max.bytes` (Default: 10MB)
- A former active segment is called an **inactive** segment after it has rolled over

Checkpoint Files

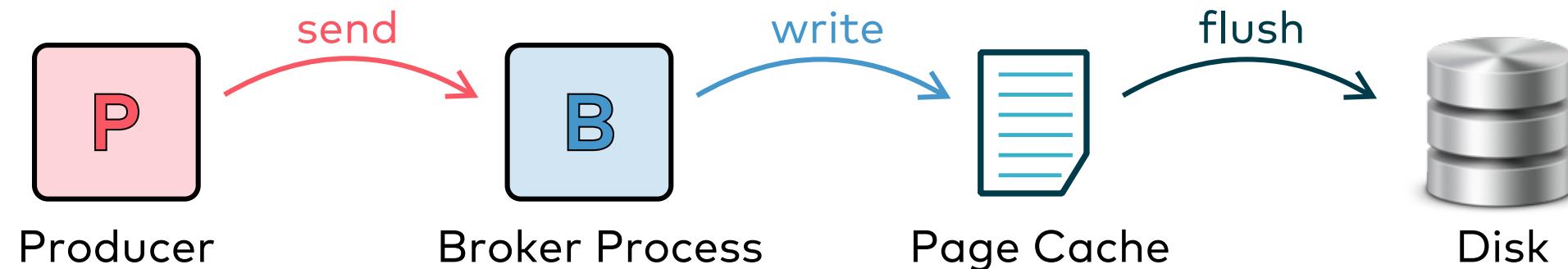
In addition, each broker has two checkpoint files:

- **replication-offset-checkpoint**
 - Contains the **high water mark** (the offset of the last committed message)
 - On startup, followers use this to truncate any uncommitted messages
- **recovery-point-offset-checkpoint**
 - Contains the offset up to which data has been flushed to disk
 - During recovery, broker checks whether messages past this point have been lost



A Step Beyond

Page Cache and Flushing to Disk



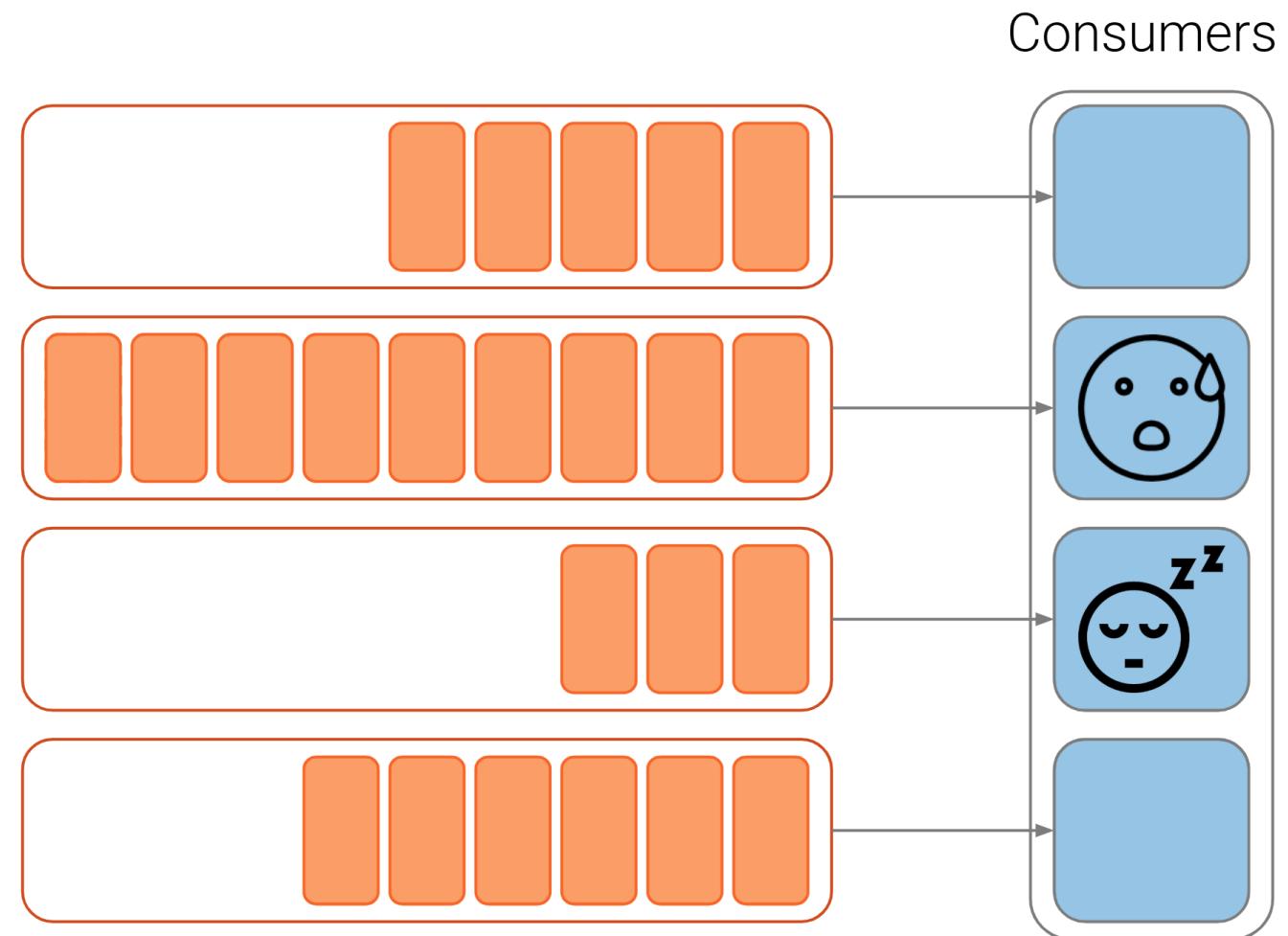
- Messages are written to partitions
- Partitions are made up of **log segment files** (new segment every 1 GB by default)
- Log segments are written to the in-memory **page cache** for performance
 - Kafka client fetch requests benefit from **zero-copy transfer**
- Page cache is **flushed to disk** when:
 - Brokers have a clean shutdown
 - OS background “flusher threads” run

4b: What are the Basics of Scaling Consumption?

Description

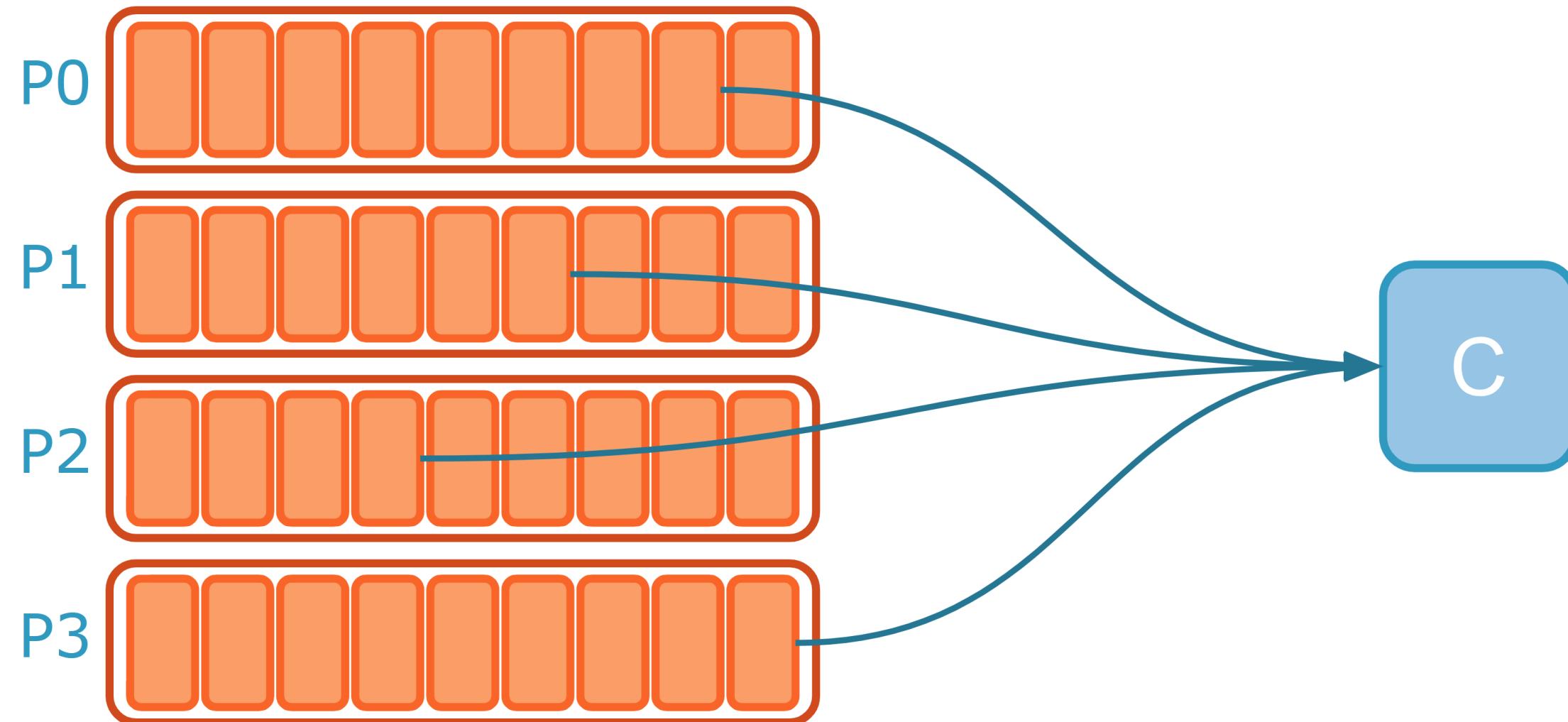
Consuming with one consumer vs. multiple consumers in a group vs. multiple groups.

Cardinality



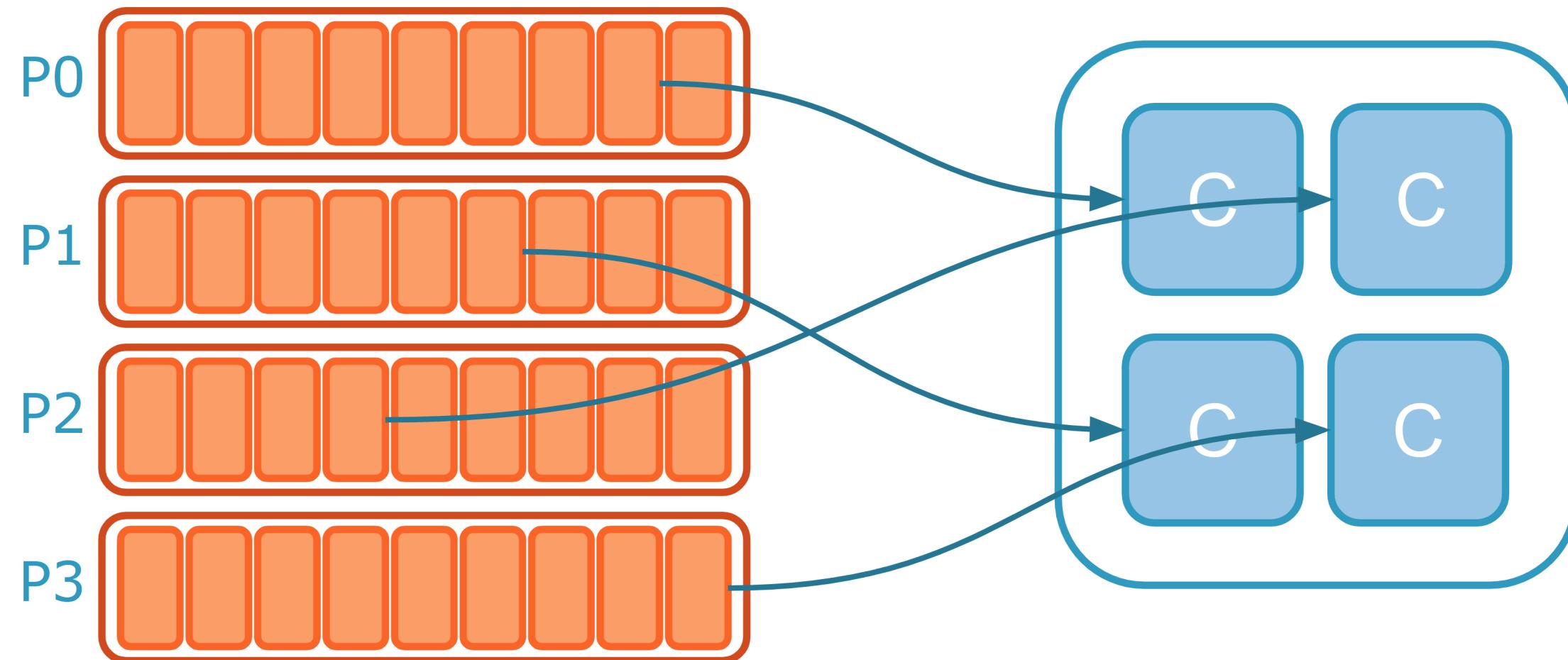
Consuming from Kafka - Single Consumer

Topic driver-positions

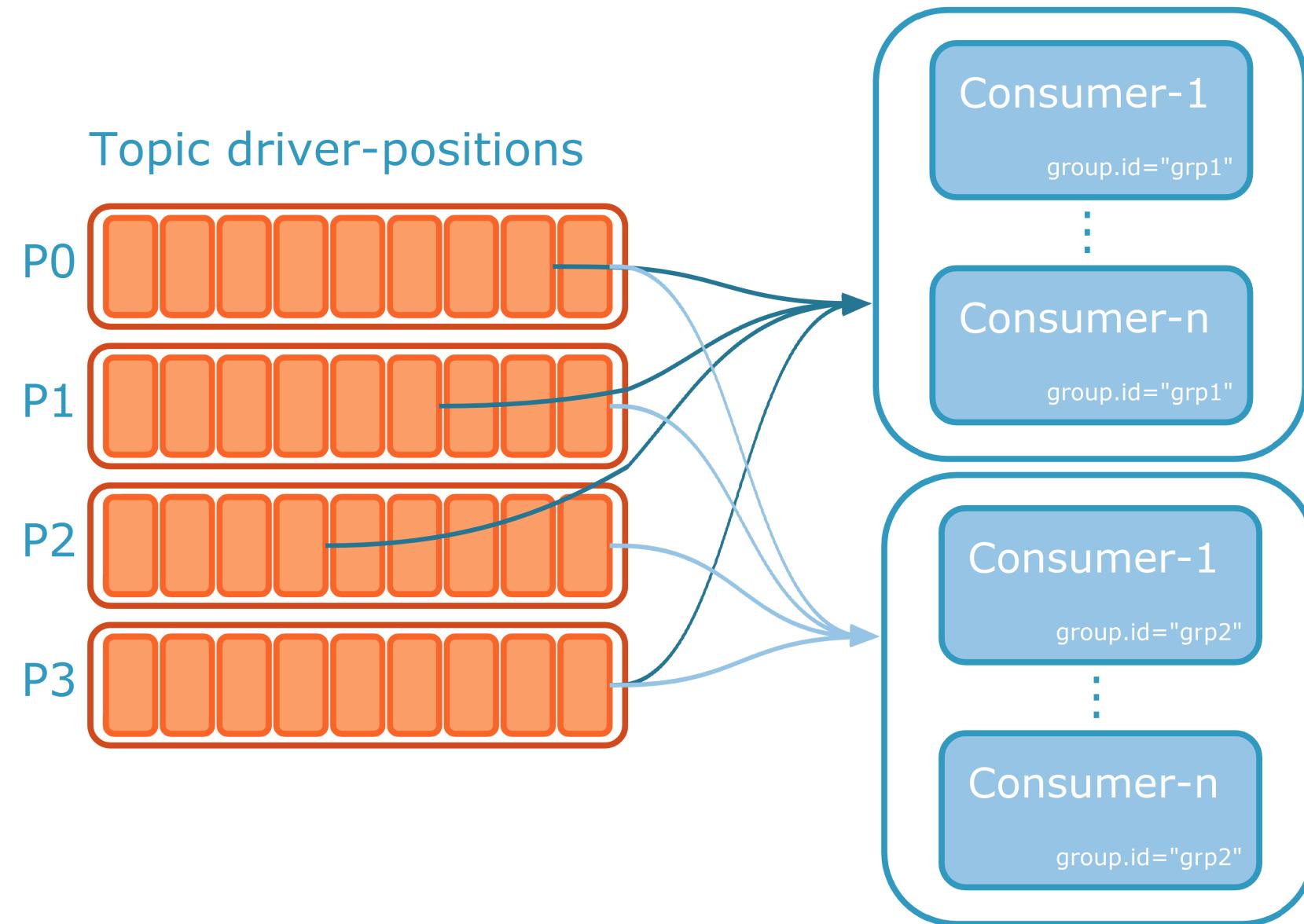


Consuming as a Group

Topic driver-positions



Multiple Consumer Groups



4c: How Does Kafka Maintain Consumer Offsets?

Description

Consumer offsets topic: uses, special properties. Viewing offsets.

Consumer Offset Management

- Consumption is tracked per topic-partition
- Each consumer maintains an offset in its memory for each partition it is reading
- As a consumer processes messages, it periodically commits the offset of the next message to be consumed
 - Offsets are committed to an internal Kafka topic `_consumer_offsets`
 - Offsets can be committed automatically or manually by the consumer

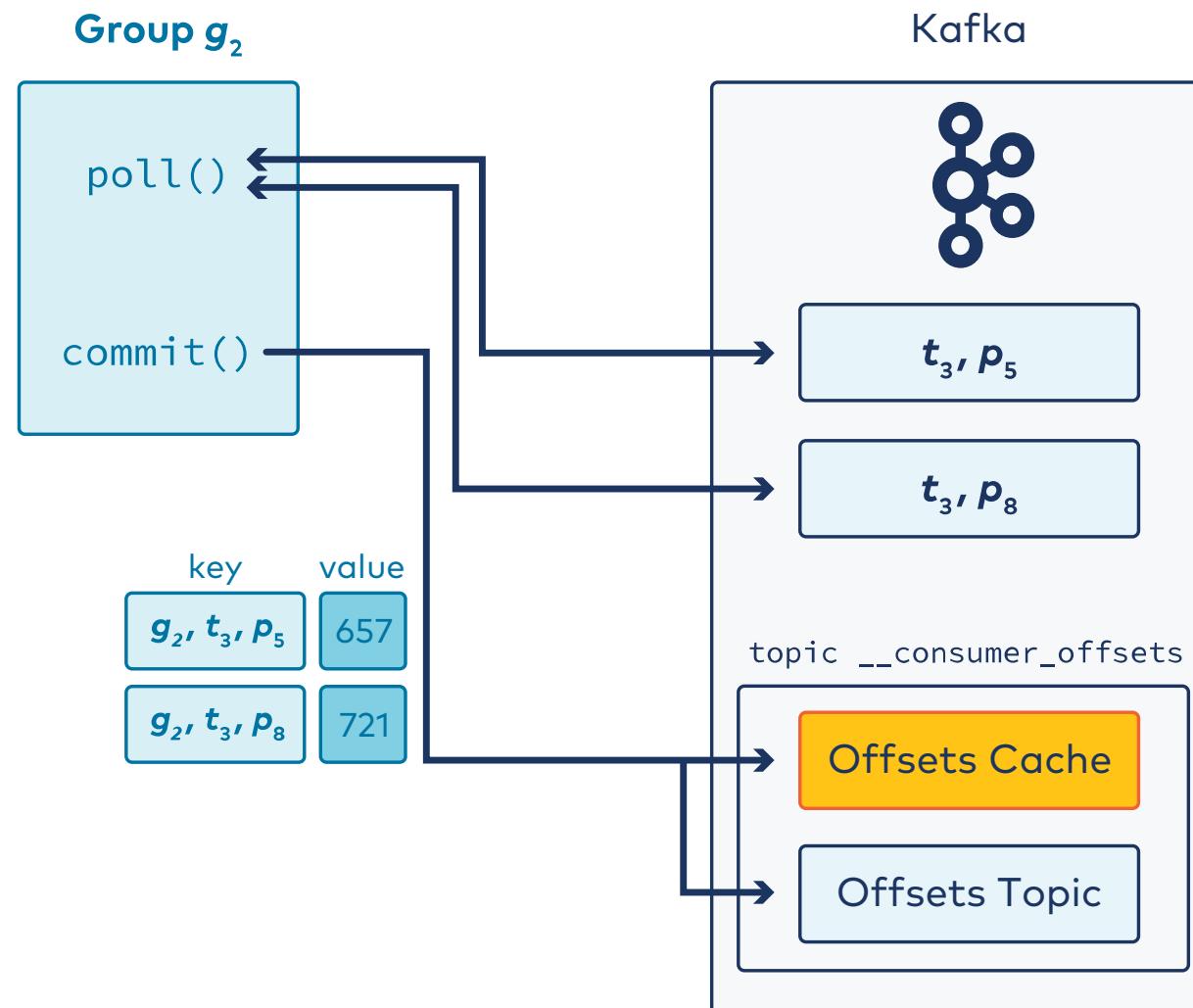
Important Configuration Settings for Offsets

- `--consumer_offsets` auto-created upon first consumption
- Scalability: `offsets.topic.num.partitions` (Default: 50)
- Resiliency: `offsets.topic.replication.factor` (Default: 3)

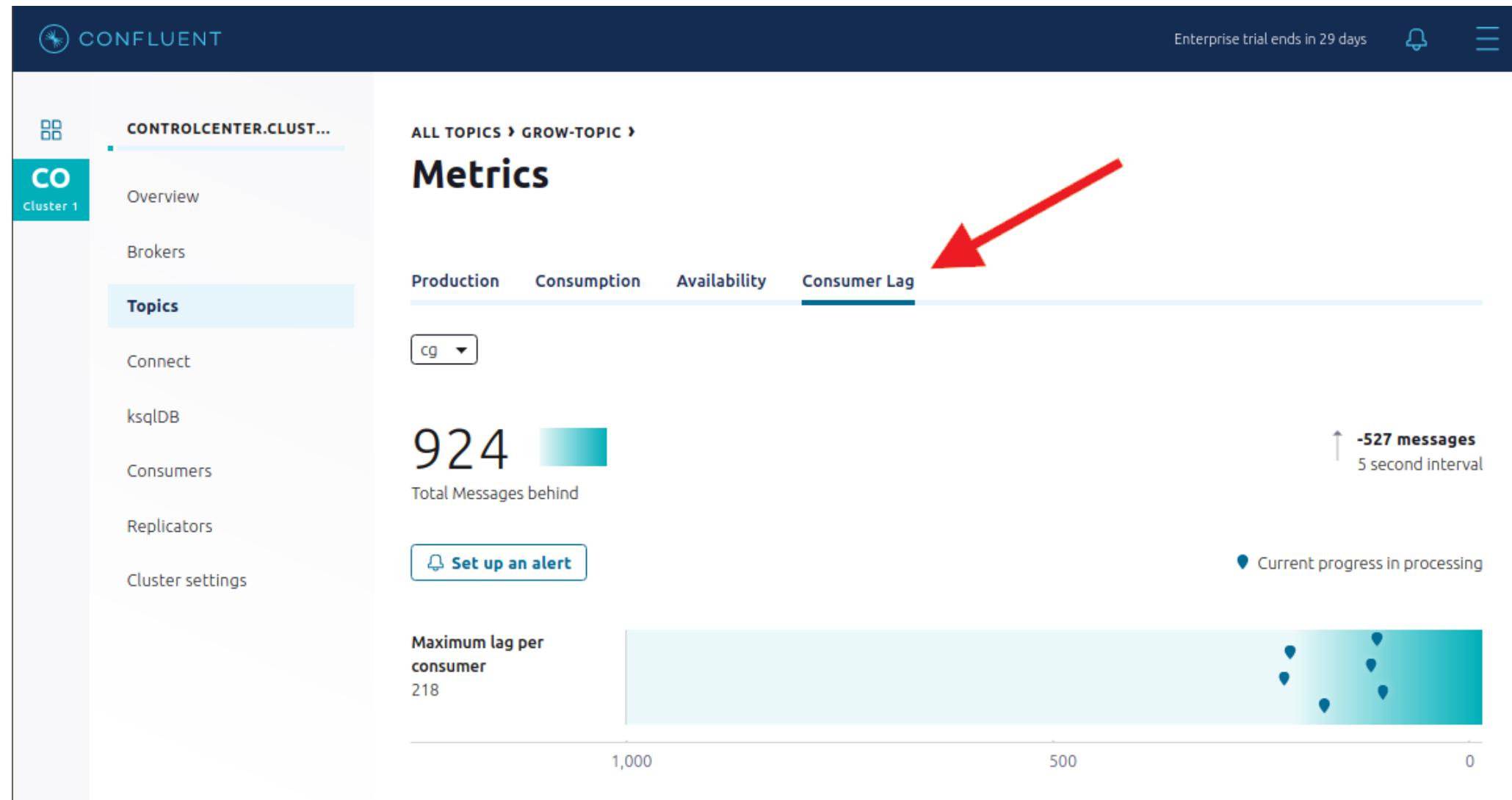


If there aren't enough brokers, auto-creation of `--consumer_offsets` fails. Consumers should only begin consuming after all brokers are running.

Consumers and Offsets (Kafka Topic Storage)



Checking Consumer Offsets (1)



Checking Consumer Offsets (2)

Look for the current offset and lag:

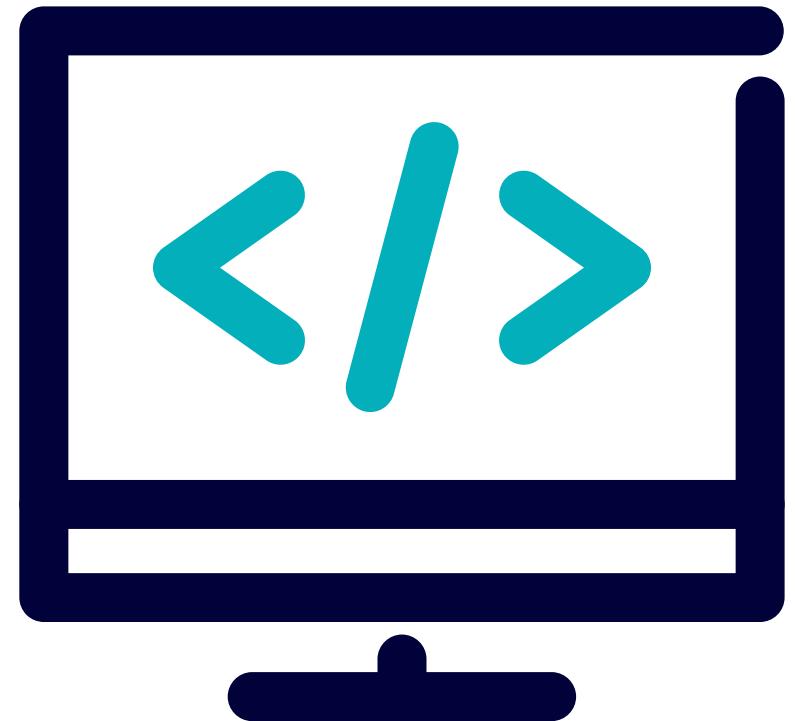
```
$ kafka-consumer-groups --group my-group \
    --describe \
    --bootstrap-server=broker101:9092,broker102:9092,broker103:9092
```

TOPIC,	PARTITION,	CURRENT OFFSET,	LOG END OFFSET,	LAG,	CONSUMER-ID
my_topic,	0,	400,	500,	100,	consumer-1_/127.0.0.1
my_topic,	1,	500,	500,	0,	consumer-1_/127.0.0.1

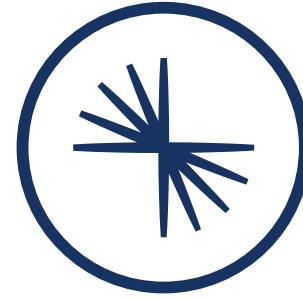
Lab: Investigating the Distributed Log

Please work on **Lab 4a: Investigating the Distributed Log**

Refer to the Exercise Guide



5: Configuring a Kafka Cluster



CONFLUENT
Global Education

Module Overview



This module contains 2 lessons:

- How Do You Configure Brokers?
- What if You Want to Adjust Settings Dynamically or Topic-wise?

5a: How Do You Configure Brokers?

Description

Static broker settings. Where to set. Examples of classes of broker settings.

Where to Set Broker Settings?

- Properties file: `server.properties`
- On each broker
 - In `/etc/kafka`
- Must be set before starting broker; change → restart broker
- Applies to
 - Broker itself
 - All partitions on the broker (by default...)



It is possible to override these broker settings. More on that in the next lesson.

Some Broker Properties

Some properties are clearly "this broker" things, e.g.,

- `log.dirs` - where data directories for logs are
- `num.network.threads` - how many network threads
- `num.io.threads` - how many worker/IO threads
- `broker.id`

Another "This Broker" Config: Listeners

First off, recall, we use `host:port` pairs to describe connections, e.g., `broker1:9092`

Two broker settings guide what ports are used and their security mechanism:

- `listeners` - for traffic within the local network
- `advertised.listeners` - to also allow traffic from an outside network

You'd specify a listener something like `listeners = [protocol]://[host]:[port]`

So you might have settings like this:

- `listeners = PLAINTEXT://broker1:9092`
- `listeners = SSL://broker1:9093`
- `advertised.listeners = SSL://broker1:9093`

Some Broker Properties Related to Logs

Some broker properties apply to all logs (partitions) on the broker:

- log rolling threshold: `log.segment.bytes` and `log.roll.ms`
- `cleanup.policy` - `delete`, `compact`



Some of these can be overridden at the topic level.

Cluster Defaults

Some properties you set in `server.properties` are meant as cluster defaults for all brokers.

Examples:

- `default.replication.factor`
- `num.partitions`
- `auto.create.topics.enable`



But `server.properties` is for **this one** broker. If the cluster defaults are not consistent across the brokers in the cluster, you may experience unpredictable behavior.

What if I Don't Configure a Property?

Kafka defaults apply!

See documentation.

But, there's more. See next lesson...

5b: What if You Want to Adjust Settings Dynamically or Apply at the Topic Level?

Description

Dynamic vs. static broker settings. Cluster-wide vs. per-broker settings. Topic overrides. Order of precedence.

There Aren't Only Static Broker Settings...

Other options:

- Dynamic broker settings
- Dynamic cluster-wide settings
- Topic overrides

Order of Precedence

1. Topic settings
2. Dynamic per-broker config
3. Dynamic cluster-wide default config
4. Static broker config in `server.properties`
5. Kafka default

Viewing Dynamic Broker Configurations

Display dynamic broker configurations for broker with ID 103:

```
$ kafka-configs \
  --bootstrap-server kafka-1:9092 \
  --broker 103 \
  --describe
```



To display all config settings, not just dynamic changes, specify --all.

Changing Broker Configurations Dynamically

- Change a cluster-wide default configuration

```
$ kafka-configs \
  --bootstrap-server kafka-1:9092 \
  --broker-defaults \
  --alter \
  --add-config log.cleaner.threads=2
```

- Change a broker configuration

```
$ kafka-configs \
  --bootstrap-server kafka-1:9092 \
  --broker 101 \
  --alter \
  --add-config log.cleaner.threads=2
```



To alter multiple config settings, use `--add-config-file new.properties`

Deleting a Broker Config

Delete a broker configuration

```
$ kafka-configs \
  --bootstrap-server kafka-1:9092 \
  --broker 107 \
  --alter \
  --delete-config min.insync.replicas
```

Question: What will the value of `min.insync.replicas` be now?



Topic Overrides

Topic level configurations to override broker defaults



The names are often different, but similar.

Examples:

Meaning	Topic Override	Broker Config
Threshold log segment size for rolling active segment	<code>segment.bytes</code>	<code>log.segment.bytes</code>
Maximum size of a message	<code>max.message.bytes</code>	<code>message.max.bytes</code>

Setting Topic Configurations from the CLI (1)

Set a topic configuration at time of topic creation

```
$ kafka-topics \  
  --bootstrap-server kafka-1:9092 \  
  --create \  
  --topic my_topic \  
  --partitions 1 \  
  --replication-factor 3 \  
  --config segment.bytes=1000000
```

Setting Topic Configurations from the CLI (2)

- Change a topic configuration for an existing topic

```
$ kafka-configs \
  --bootstrap-server broker_host:9092 \
  --alter \
  --topic my_topic \
  --add-config segment.bytes=1000000
```

- Delete a topic configuration

```
$ kafka-configs \
  --bootstrap-server broker_host:9092 \
  --alter \
  --topic my_topic \
  --delete-config segment.bytes
```

Viewing Topic Settings from the CLI

- Show the topic configuration settings

```
$ kafka-configs \  
  --bootstrap-server broker_host:9092 \  
  --describe \  
  --topic my_topic
```

Configs for topic 'my_topic' are segment.bytes=1000000

- Show the partition, leader, replica, ISR information

```
$ kafka-topics \  
  --bootstrap-server broker_host:9092 \  
  --describe \  
  --topic my_topic
```

Topic:my_topic PartitionCount:1 ReplicationFactor:3 Configs:segment.bytes=1000000
Topic: my_topic Partition: 0 Leader: 101 Replicas: 101,102,103 Isr: 101,102,103



Deleting Topics

- Topic deletion is enabled by default on brokers in the `server.properties` file
 - `delete.topic.enable` (Default: true)
- Caveats
 - Stop all producers/consumers before deleting
 - All brokers must be running for the `delete` to be successful
- Command:

```
$ kafka-topics \
  --bootstrap-server kafka-1:9092 \
  --delete \
  --topic my_topic
```

Activity: Troubleshooting Configuration Confusion



Discussion:

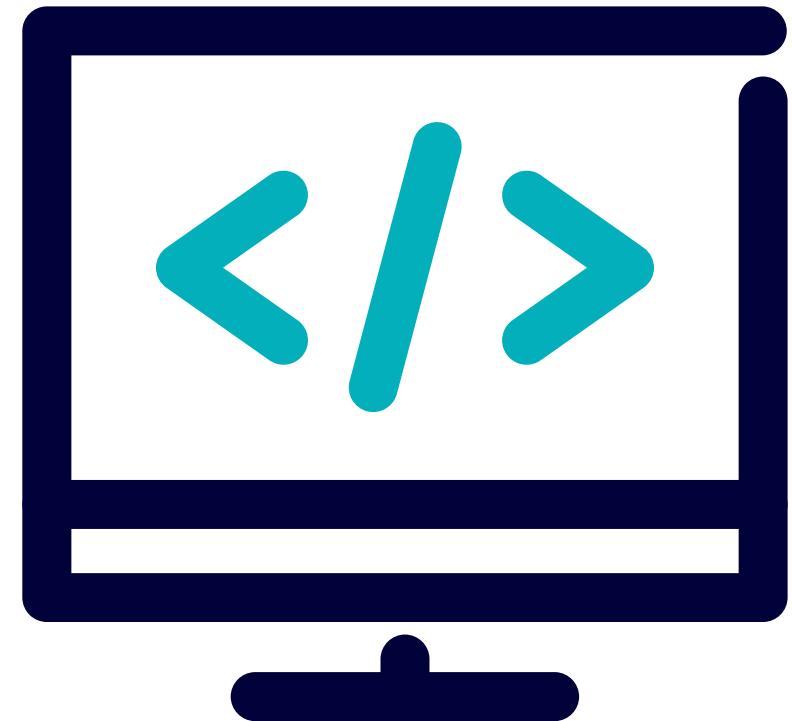
- A colleague insists he changed the rollover time for the active segment to 2 hours.
- But another colleague is reporting that she has seen some log segments for topic t_7 , partition p_{12} are on broker 103 have been the active log segment with timestamps 4 and 5 hours in past.

Another colleague wants answers and explanations. What do you tell them?

Lab: Exploring Configuration

Please work on **Lab 5a: Exploring Configuration**

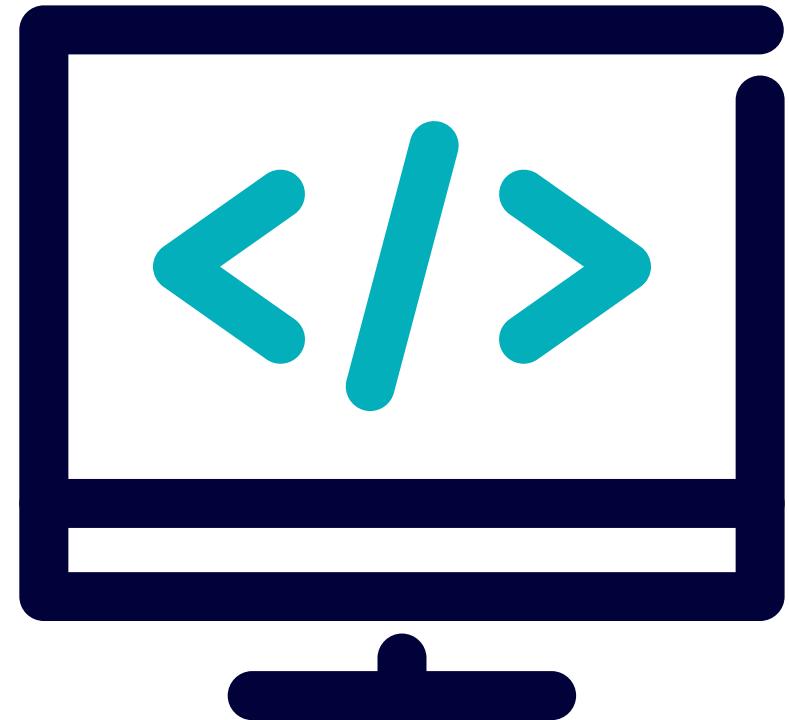
Refer to the Exercise Guide



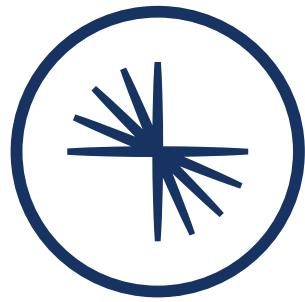
Lab: Increasing Replication Factor

Please work on **Lab 5b: Increasing Replication Factor**

Refer to the Exercise Guide



6: Managing a Kafka Cluster



CONFLUENT
Global Education

Module Overview



This module contains 5 lessons:

- What Should You Consider When Installing and Upgrading Kafka?
- What are the Basics of Monitoring Kafka?
- How Can You Decide How Kafka Keeps Messages?
- How Can You Move Partitions To New Brokers Easily?
- What Should You Consider When Shrinking a Cluster?

6a: What Should You Consider When Installing and Upgrading Kafka?

Description

Considerations for installation and upgrading.

Methods of Installing

- You can deploy Confluent Platform...
 - Using a cloud provider
 - A Tar archive
 - DEB or RPM package
 - Docker container
- Deploy in a distributed environment using one of...
 - Confluent for Kubernetes (formerly called Confluent Operator)
 - Ansible playbooks



- Kafka brokers need to have ZooKeeper installed first
- Schema Registry needs to have Kafka brokers installed first

Local vs. Distributed Installation

Local	Distributed
<ul style="list-style-type: none">• When installed on a single machine.• Installed via zip, tar, or Docker images	<ul style="list-style-type: none">• When services are installed across several machines• Installed via Kubernetes or Ansible

Upgrading a Cluster (1)

There are many things to consider when upgrading the Confluent Platform:

- The order of upgrade for ZooKeeper nodes, brokers, and the rest of the Confluent Platform is very important.
- Always read the upgrade documentation on our website to get the full scope of what the upgrades entails.

Upgrading a Cluster (2)

- Be aware of the broker protocol version and the log message format version between the various releases.
- Apply a license key for the Confluent Platform.
 - Know how to apply the key.
- Do rolling restart—the entire cluster stays up as you take down each broker one by one to upgrade.



See link in your guide!

6b: What are the Basics of Monitoring Kafka?

Description

Basics of metrics and monitoring tools. Kafka logs vs. system logs.

Logs vs. Logs

- Kafka topic data
 - `log.dirs` property in `server.properties`
- Application logging with Apache `log4j`
 - `LOG_DIR`: configure the `log4j` files directory by exporting the environment variable
(Default: `/var/log/kafka`)

Important log4j Files

- By default, the broker log4j log files are written to `/var/log/kafka`
 - `server.log`: broker configuration properties and transactions
 - `controller.log`: all broker failures and actions taken because of them
 - `state-change.log`: every decision broker has received from the controller
 - `log-cleaner.log`: compaction activities
 - `kafka-authorizer.log`: requests being authorized
 - `kafka-request.log`: fetch requests from clients
- Manage logging via `/etc/kafka/log4j.properties`

Tools for Collecting Metrics

- **Confluent Control Center**
- Other metrics tools:
 - JConsole
 - Graphite
 - Grafana
 - CloudWatch
 - DataDog

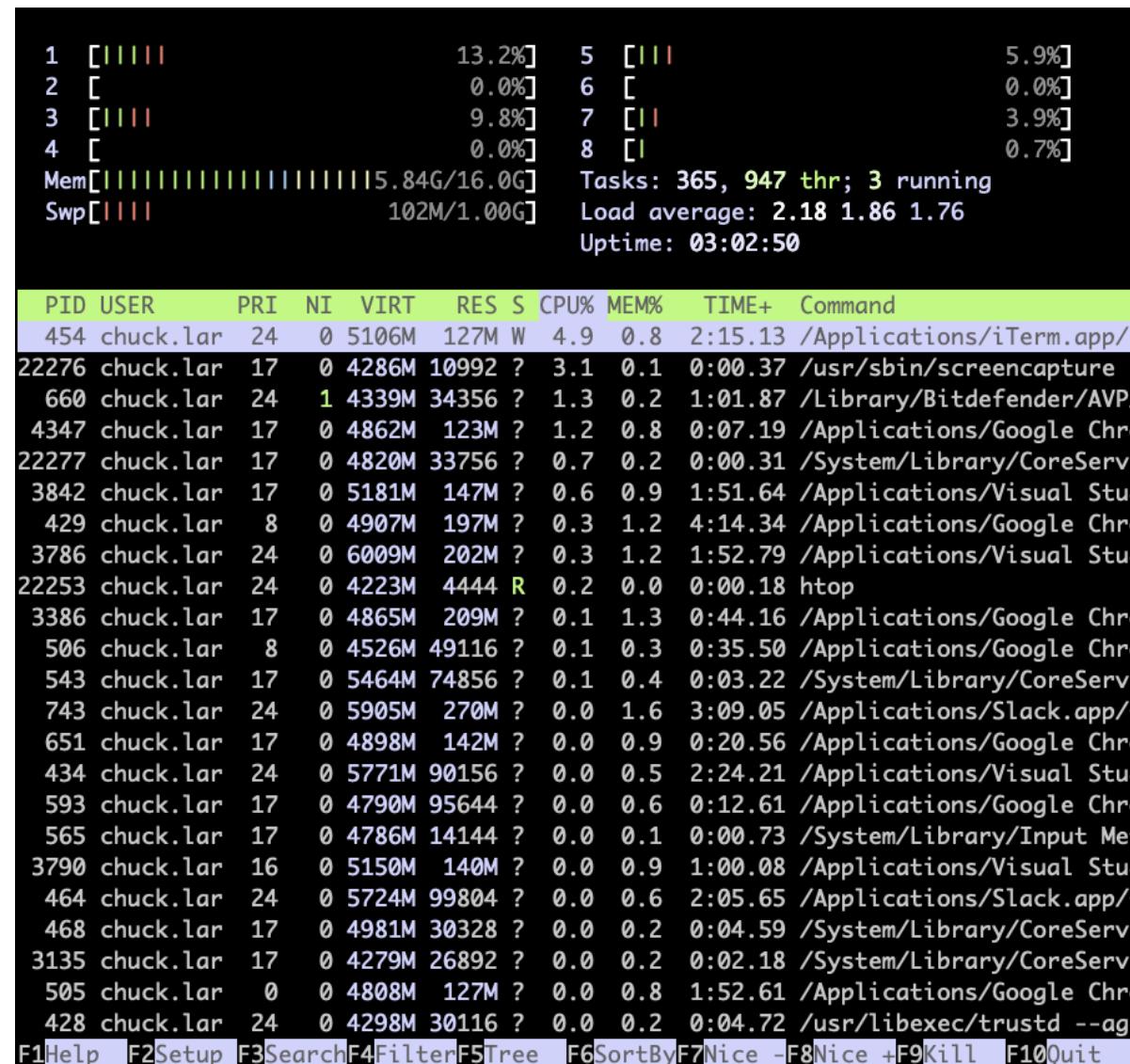
Configuring the Cluster for Monitoring

- Enable JMX metrics by setting `JMX_PORT` environment variable

```
$ export JMX_PORT=9990
```

- Configure `client.id` on producers and consumers
 - Monitor by application
 - Used in logs and JMX metrics

Monitoring Kafka at the OS Level



- Open file handles
 - Set `ulimit -n 100000`
 - Alert at 60% of the limit
- Disk
 - Alert at 60% capacity
- Network bytesIn/bytesOut
 - Alert at 60% capacity

Troubleshooting Issues

- Parse the `log4j` logs
- Check metrics
- Avoid unnecessary restarts

6c: How Can You Decide How Kafka Keeps Messages?

Description

Deletion. Compaction. Examples. Details of implementation of compaction. Monitoring and logging of compaction.

Managing Log File Growth

- `log.cleanup.policy`
 - `delete`
 - `compact`
 - `both`: `delete,compact`



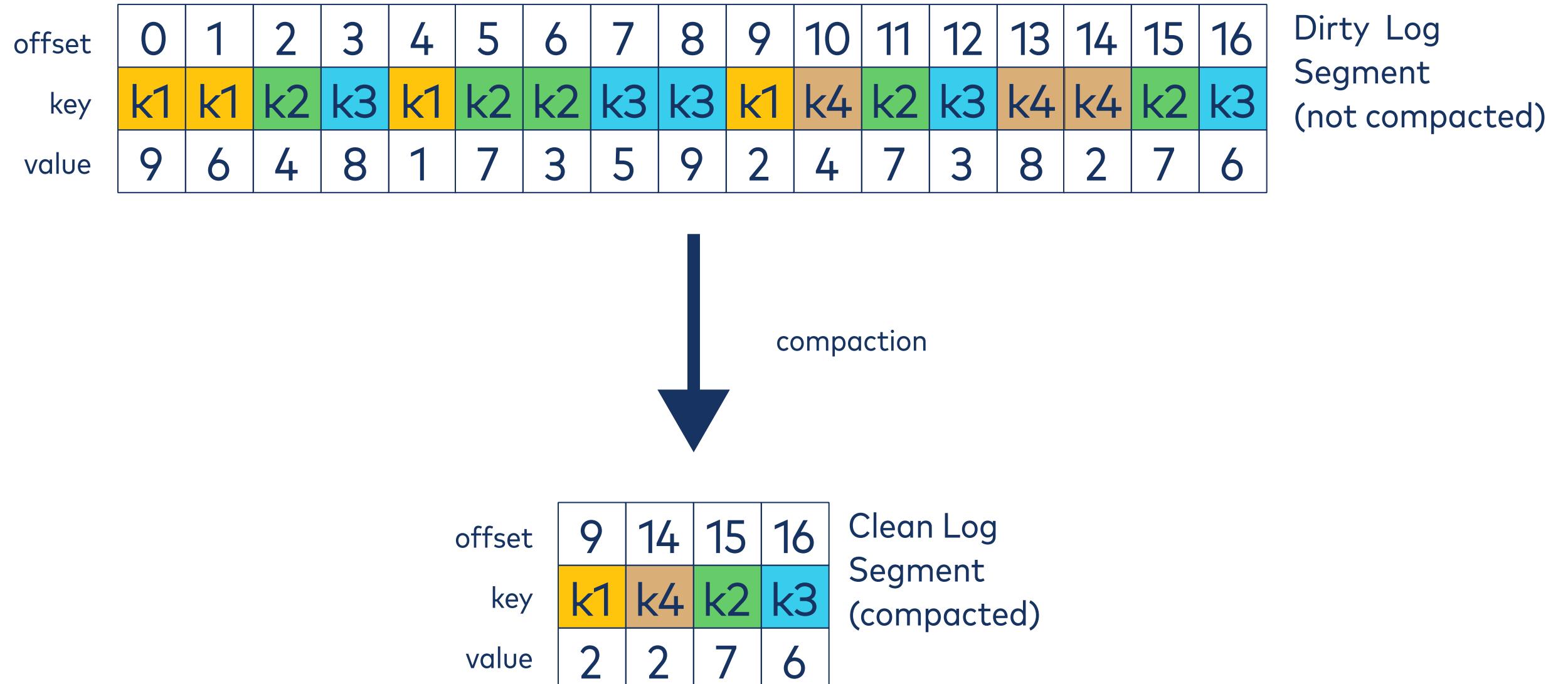
Cleanup policy can also be set on a per-topic basis

Compact Policy Use Case

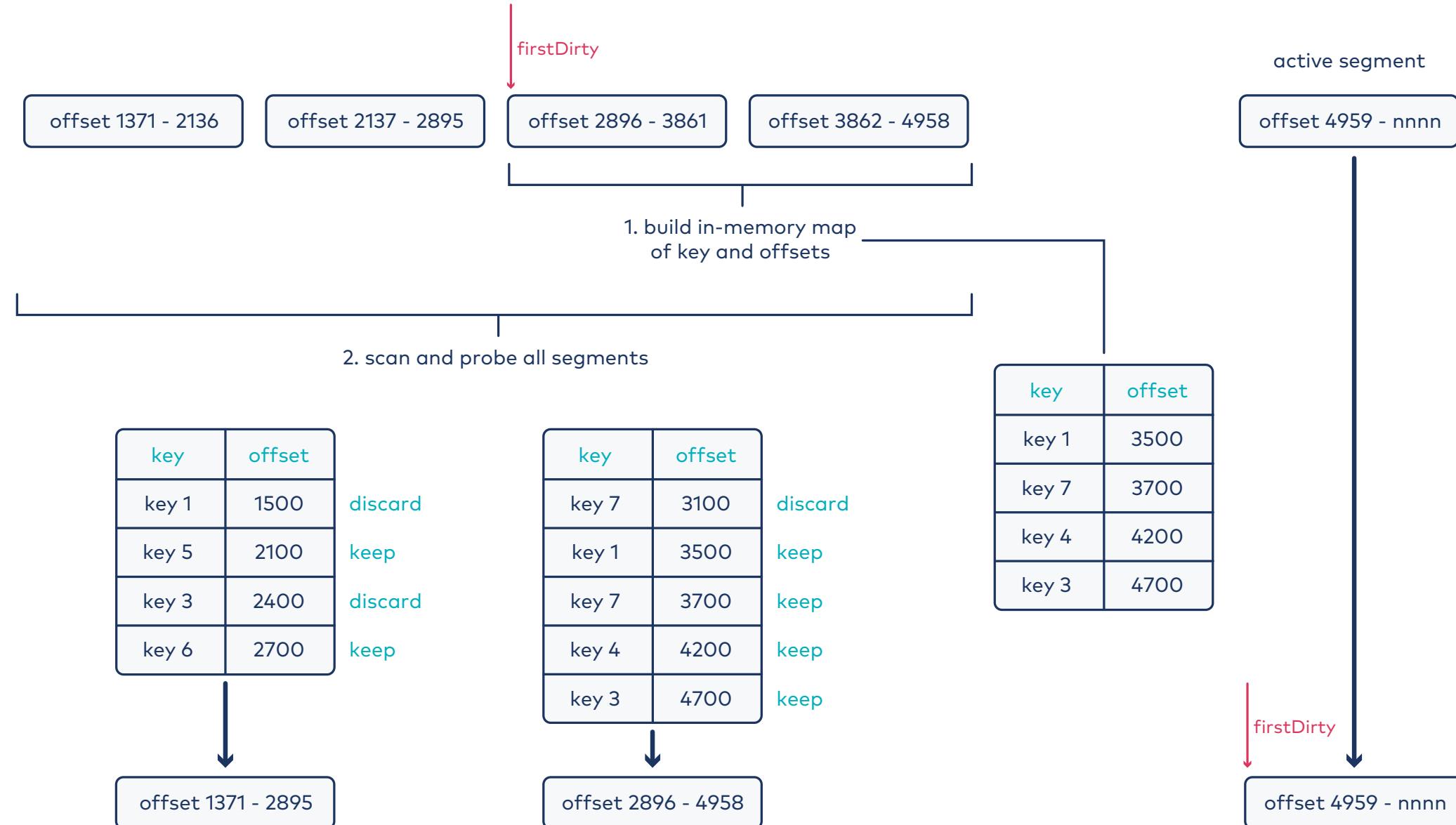
Keep only the **most recent** value for a given key

- Examples:
 - Database change capture
 - Real-time table lookups during stream processing
 - Maintaining a topic of temperatures per postal code
 - Tracking the progress of e-commerce orders

Log Compaction: What is it?



Log Compaction: Implementation



Log Messages During Cleaning

When the cleaning process is taking place, you will see log messages like this:

```
Beginning cleaning of log my_topic,0
```

```
Building offset map for my_topic,0 ...
```

```
Log cleaner thread 0 cleaned log my_topic,0 (dirty section=[100111,200011])
```

6d: How Can You Move Partitions To New Brokers Easily?

Description

Basics of Auto Data Balancer.

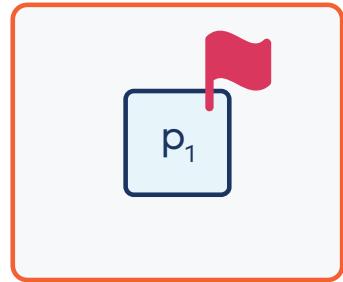
Auto Data Balancer Overview

- Applies when:
 - You've added new (empty) brokers and want to move partitions to them
 - You've noticed disk usage among brokers is not balanced
- What it does:
 - Calculates an optimal placement of partitions among brokers
 - Automatically moves partitions
- Paid Confluent feature

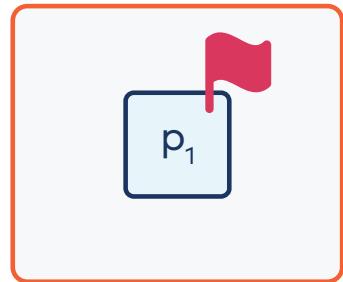


See also: Self-Balancing Clusters

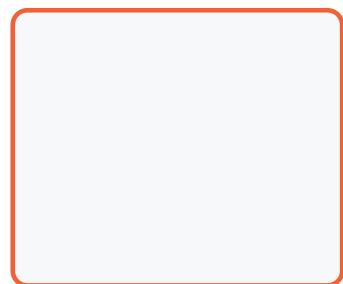
Partition Movement with Auto Data Balancer



ISR list: [0, 1]



Add new replica of p_1 on new broker
ISR list: [0, 1, 2]



Switch leader of p_1 over to 2 & delete original replica
ISR list: [2, 1]



Disk utilization increases during rebalance

6e: What Should You Consider When Shrinking a Cluster?

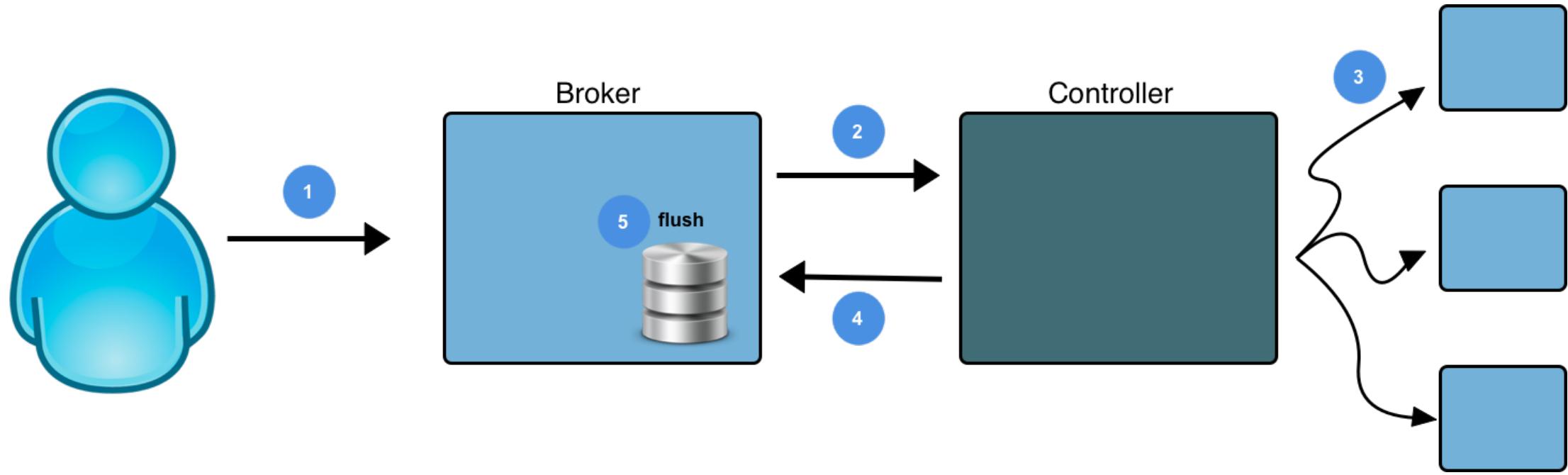
Description

Tips for shrinking a cluster. Controlled shutdown of a broker.

Shrinking the Cluster

- Why reduce the number of brokers in the cluster?
 - Maintenance on a broker
 - Reduce cost during periods of low cluster utilization
- Decommissioning a broker
 1. Use Auto Data Balancer or `kafka-reassign-partitions` to reassign its partitions to other brokers
 2. Perform a controlled shutdown

Controlled Shutdown



1. Administrator sends a `SIGTERM` to the broker Java process, e.g., `kafka-server-stop`
2. The broker sends request to controller.
3. The controller facilitates leader elections
4. Controller `acks` broker.
5. Broker flushes file system caches to disk.
6. Broker shuts down.

Replacing a Failed Broker

- On a new server, start a new broker with the same value for `broker.id`
- The new server will automatically bootstrap data
- If possible, do the broker replacement at an off-peak time



Broker will copy the data as fast as it can during recovery. This can have a significant impact on the network.

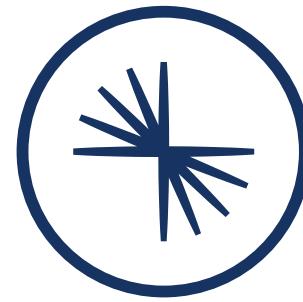
Lab: Kafka Administrative Tools

Please work on **Lab 6a: Kafka Administrative Tools**

Refer to the Exercise Guide



7: Balancing Load with Consumer Groups and Partitions



CONFLUENT
Global Education

Module Overview



This module contains 3 lessons:

- How Do Partitions and Consumers Scale?
- How Do Groups Distribute Work Across Partitions?
- How Does Kafka Manage Groups?

Where this fits in:

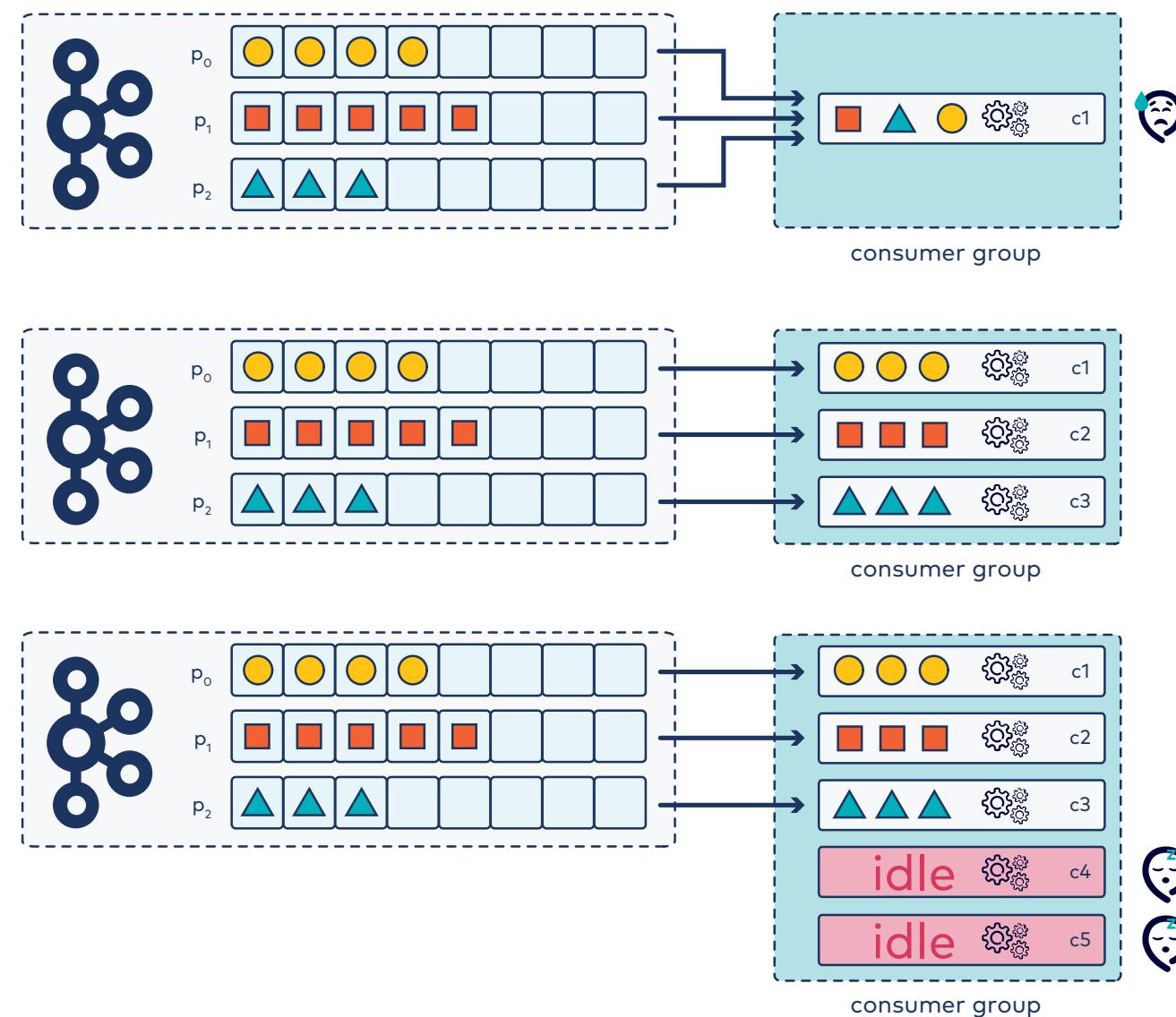
- Hard Prerequisite: Fundamentals Course
- Recommended Prereq: Other Ways Kafka Provides Durability

7a: How Do Partitions and Consumers Scale?

Description

Scalability of consumer groups. Adding partitions. Benefits and challenges of adding partitions.

Consumer Group Scalability



Adding Partitions

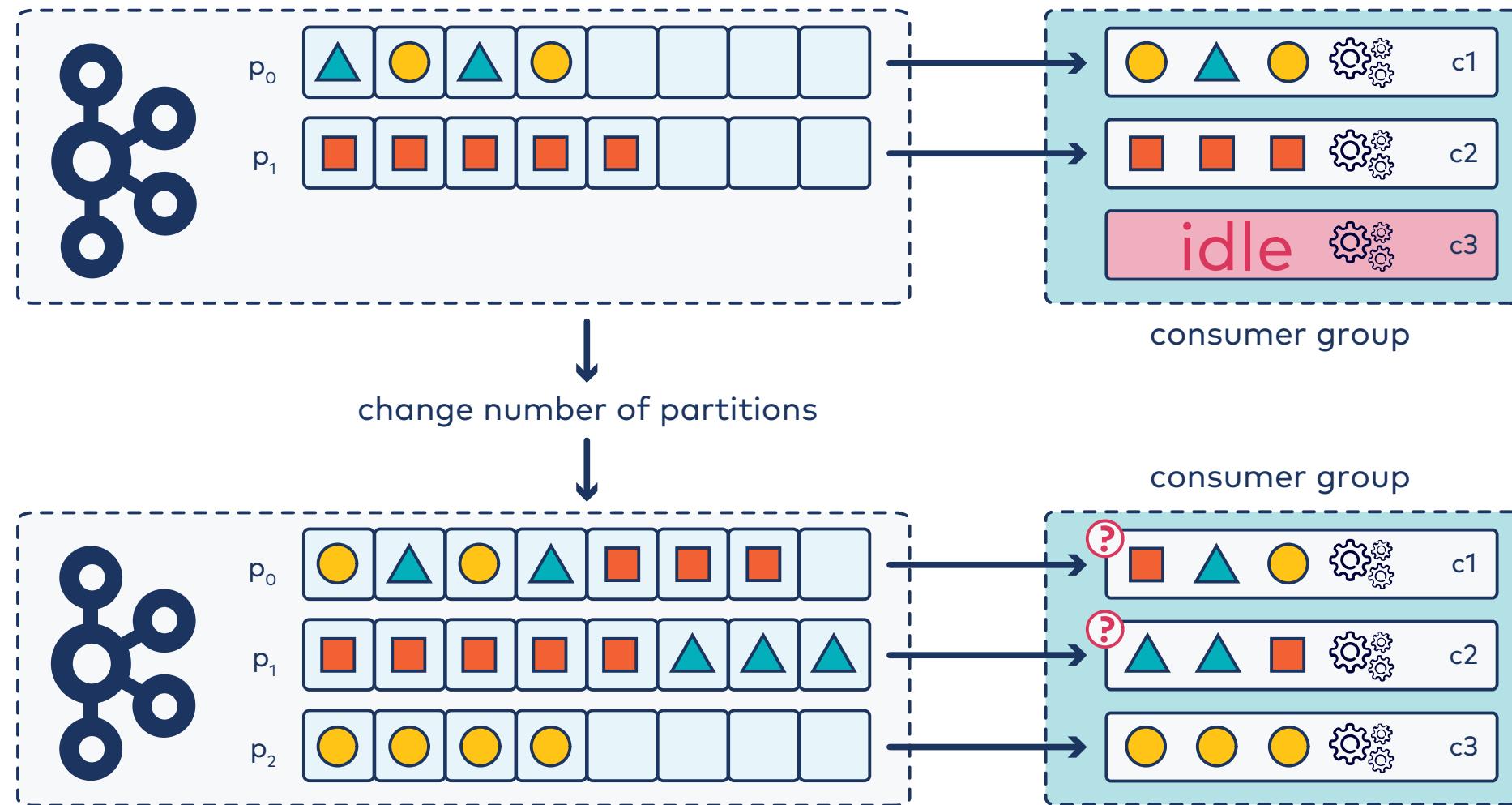
Use the `kafka-topics` command, e.g.:

```
$ kafka-topics \
  --bootstrap-server broker_host:9092 \
  --alter \
  --topic my_topic \
  --partitions 30
```

Notes:

- Doesn't move data from existing partitions
- Messages with the same key will no longer be on the same partition
 - Workaround: Consume from old topic and produce to a new topic with the correct number of partitions

Consumer Groups: Caution When Changing Partitions



Number of Partitions

- Ideal number of partitions: $\max(t/p, t/c)$
 - t : target throughput
 - p : Producer throughput per Partition
 - c : Consumer throughput per Partition

Improving Throughput With More Partitions

- More partitions → higher throughput
- Rule of thumb for maximums:
 - Up to 4,000 partitions per broker
 - Up to 200,000 partitions per cluster

Downside to More Partitions

- More open file handles
- Longer leader elections → more downtime after broker failure
- Higher latency due to replication
- More client memory (buffering per partition)



When producing keyed messages: avoid unbalanced key utilization. This leads to "hot partitions."

7b: How Do Groups Distribute Work Across Partitions?

Description

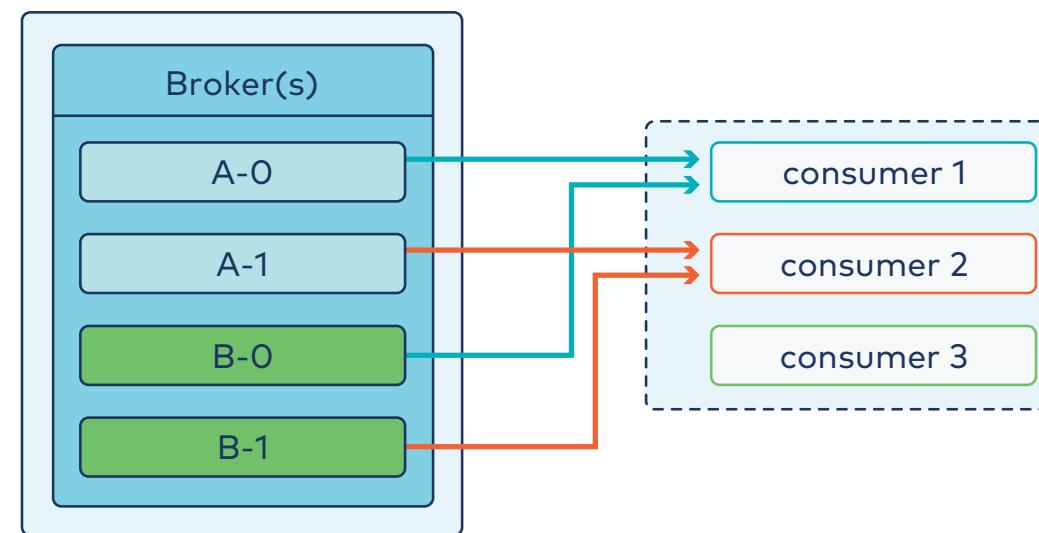
Assignment of partitions to consumers. Strategies: range, round robin, sticky, cooperative sticky.

Partition Assignment within a Consumer Group

- Partitions are 'assigned' to consumers
- A single partition is consumed by only one consumer in any given consumer group
 - Messages with same key will go to same consumer (unless you change number of partitions)
 - `partition.assignment.strategy` in the consumer configuration

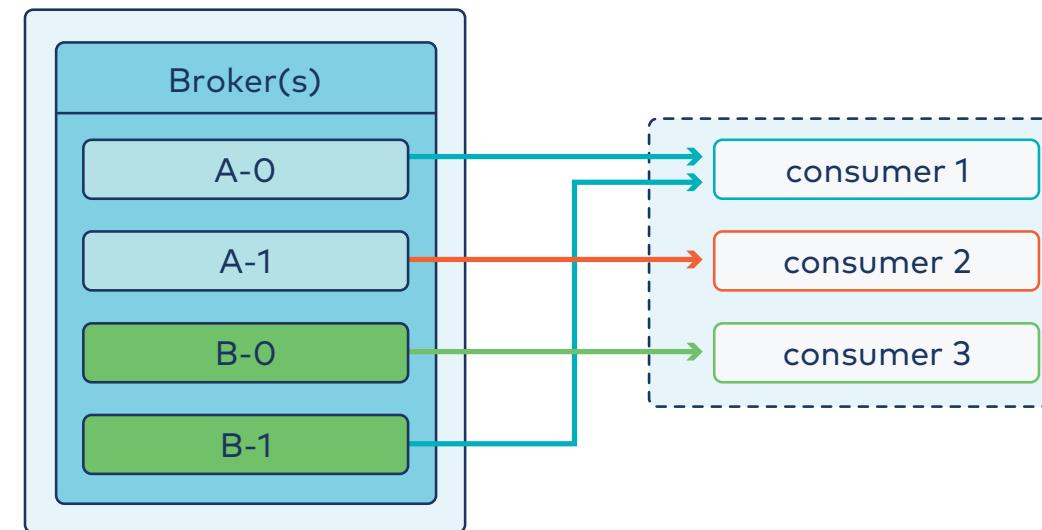
Partition Assignment Strategy: Range

- Range is the default `partition.assignment.strategy`
- Useful for co-partitioning across topics, e.g.:
 - Package ID across `delivery_status` and `package_location`
 - User ID across `search_results` and `search_clicks`



Partition Assignment Strategy: RoundRobin

- Partitions assigned one at a time in rotating fashion



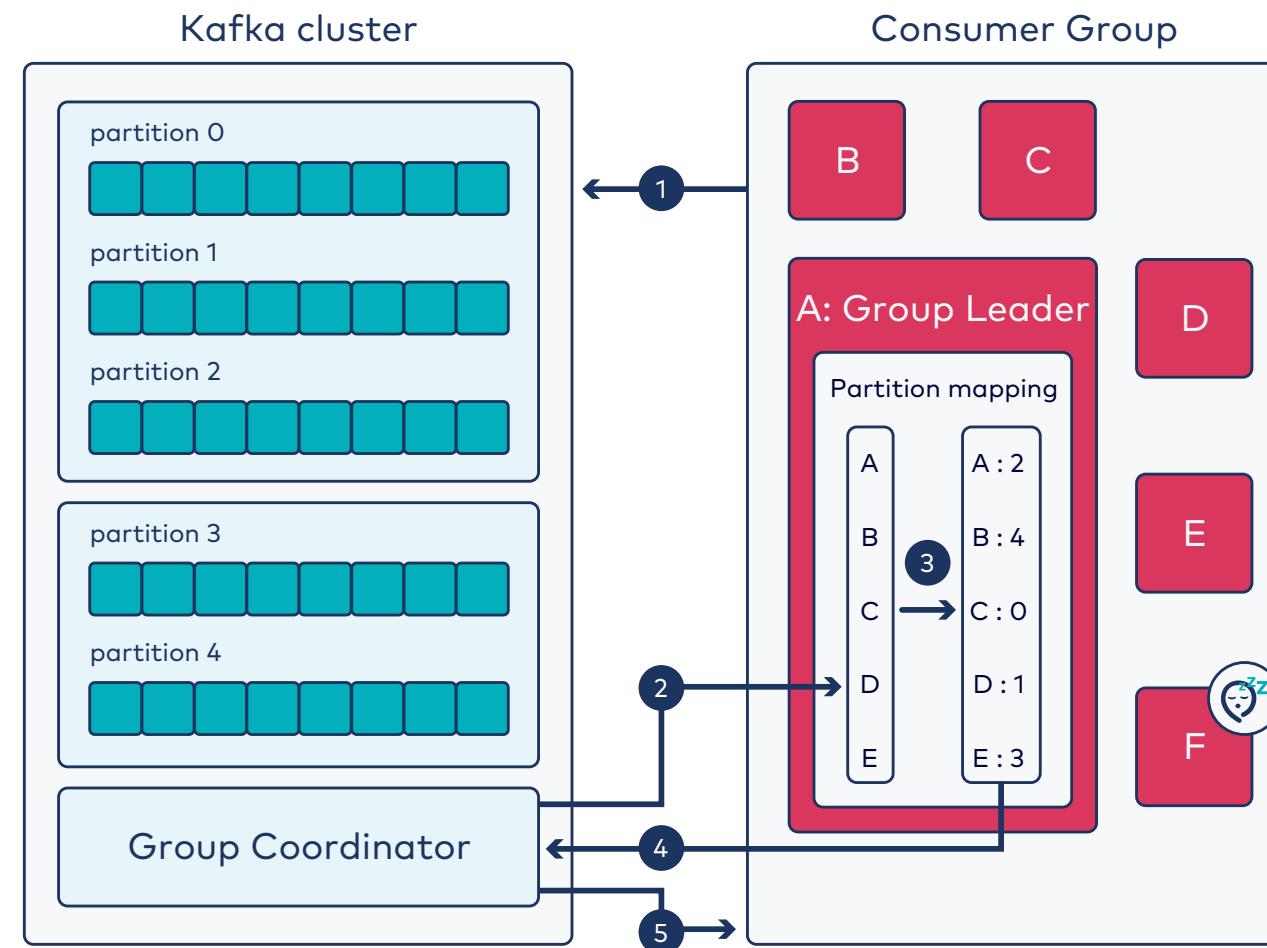
Partition Assignment Strategy: Sticky and CooperativeSticky

- Sticky
 - Is RoundRobin with assignment preservation across rebalances
- CooperativeSticky
 - Is Sticky without its "stop-the-world" rebalancing of all partitions



A Step Beyond

Registering a Consumer Group



7c: How Does Kafka Manage Groups?

Description

Group management. Rebalances. Heartbeats and failure detection.

Consumer Rebalancing

Rebalance triggers:

- Consumer leaves consumer group
- New consumer joins consumer group
- Consumer changes its topic subscription
- Consumer group notices change to topic metadata (e.g., increase # partitions)

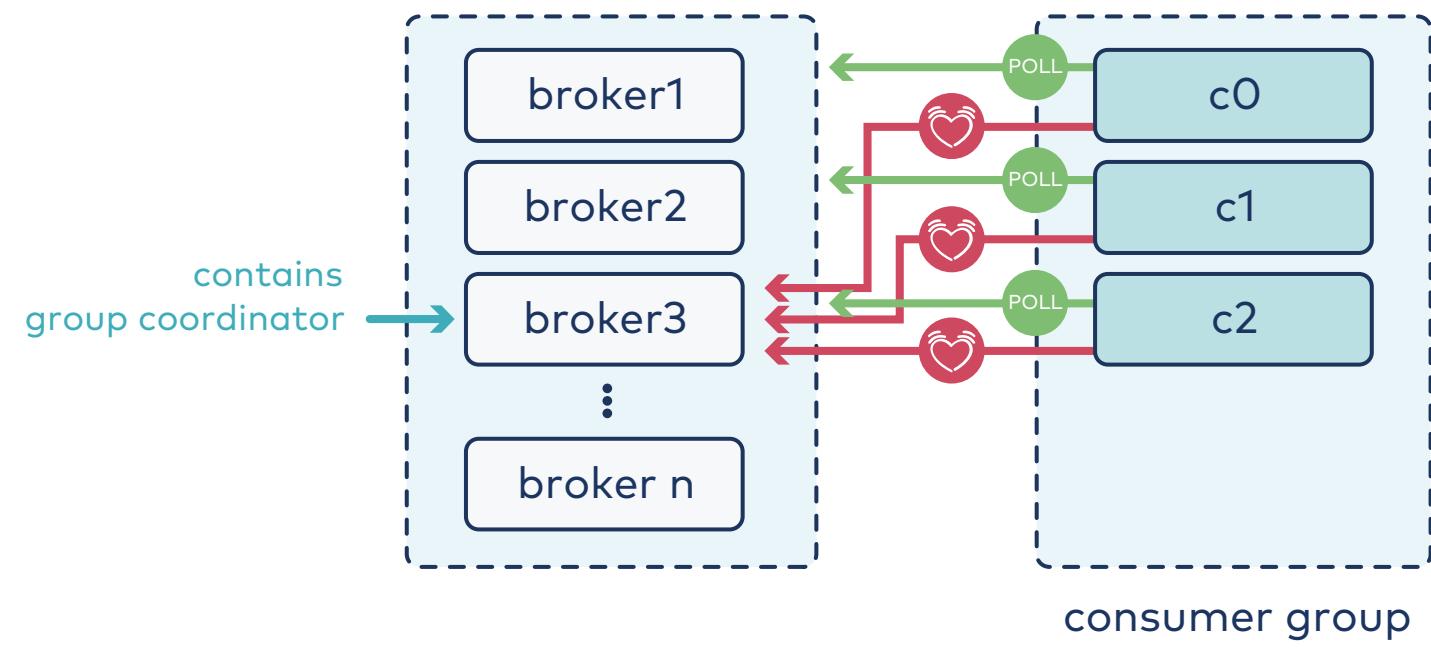
Rebalance Process:

1. Group coordinator uses flag in heartbeat to signal rebalance to consumers
2. Consumers pause, commit offsets
3. Consumers rejoin into new "generation" of consumer group
4. Partitions are reassigned
5. Consumers resume from new partitions



Consumption pauses during rebalance. Avoid unnecessary rebalances.

Consumer Failure Detection



- Consumers send heartbeats in background thread, separate from `poll()`
 - `heartbeat.interval.ms` (Default: 3 s)
- `session.timeout.ms` (Default: 45 s)
 - If no heartbeat is received in this time, consumer is dropped from group
- `poll()` must still be called periodically
 - `max.poll.interval.ms` (Default: 5 minutes)

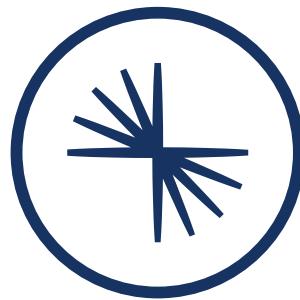
Lab: Modifying Partitions and Viewing Offsets

Please work on **Lab 7a: Modifying Partitions and Viewing Offsets**

Refer to the Exercise Guide



8: Optimizing Kafka's Performance



CONFLUENT
Global Education

Module Overview



This module contains 7 lessons:

- How Does Kafka Handle the Idea of Sending Many Messages at Once?
- How Do Produce and Fetch Requests Get Processed on a Broker?
- How Can You Measure and Control How Requests Make It Through a Broker?
- What Else Can Affect Broker Performance?
- How Do You Control It So One Client Does Not Dominate the Broker Resources?
- What Should You Consider in Assessing Client Performance?
- How Can You Test How Clients Perform?

The Meaning of Performance

- **Throughput**
 - amount of data moving through Kafka per second
- **Latency**
 - The delay from the time data is written to the time it is read
- **Recovery Time**
 - The time to return to a “good” state after some failure

8a: How Does Kafka Handle the Idea of Sending Many Messages at Once?

Description

Batching. Pipelining. Tuning batching. Compression.

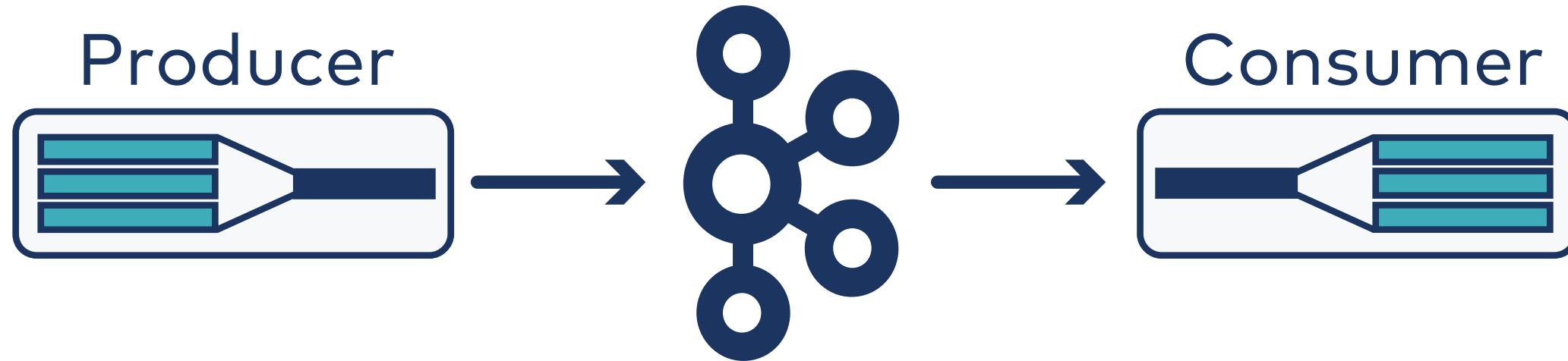
Batching for Higher Throughput



Batching Messages (2)

- `batch.size` (Default: 16 KB):
 - The maximum size of a batch before sending
- `linger.ms` (Default: 0, i.e., send immediately):
 - Time to wait for messages to batch together

End-To-End Batch Compression



1. Producer batches and compresses into single message
2. Compressed message stored in Kafka
3. Consumer decompresses

Tuning Producer Throughput and Latency

- `batch.size`, `linger.ms`
 - High throughput: large `batch.size` and `linger.ms`, or flush manually
 - Low latency: small `batch.size` and `linger.ms`
- `buffer.memory`
 - Default: 32 MB
 - The producer's buffer for messages to be sent to the cluster
 - Increase if producers are sending faster than brokers are acknowledging, to prevent blocking
- `compression.type`
 - `gzip`, `snappy`, `lz4`, `zstd`
 - Configurable per producer, topic, or broker

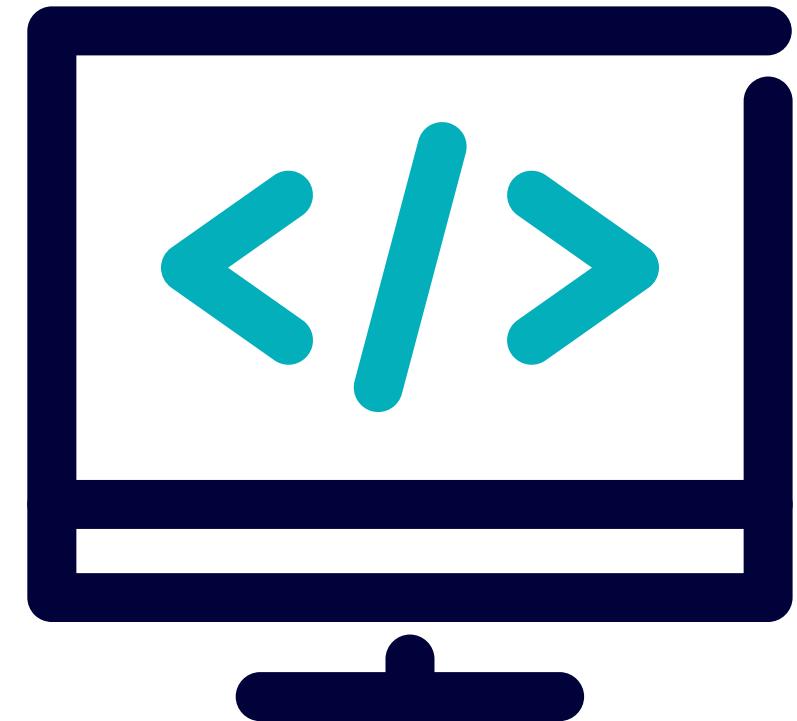
Tuning Consumer Throughput and Latency

- High throughput:
 - Large `fetch.min.bytes` (Default: 1)
 - Reasonable `fetch.max.wait.ms` (Default: 500)
- Low latency:
 - `fetch.min.bytes=1`

Lab: Exploring Producer Performance

Please work on **Lab 8a: Exploring Producer Performance**

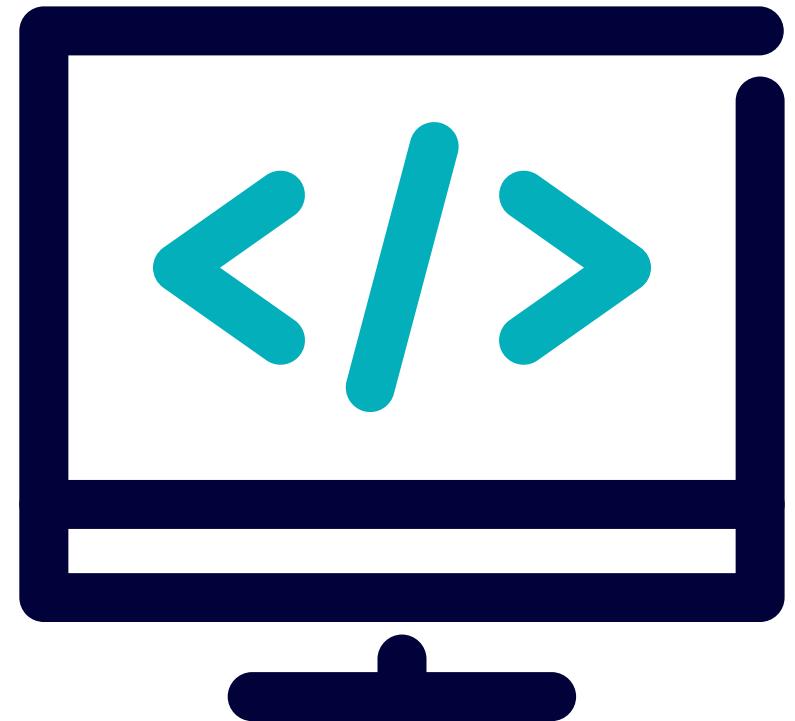
Refer to the Exercise Guide



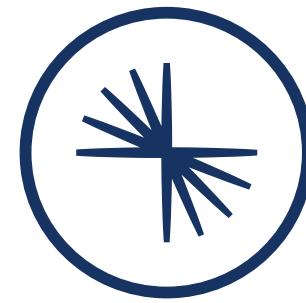
Lab: Performance Tuning

Please work on **Lab 8b: Performance Tuning**

Refer to the Exercise Guide



9: Securing a Kafka Cluster



CONFLUENT
Global Education

Module Overview



This module contains 4 lessons:

- What are the Basic Ideas You Should Know about Kafka Security?
- What Options Do You Have For Securing a Kafka/Confluent Deployment?
- How Can You Easily Control Who Can Access What?
- What Should You Know Securing a Deployment Beyond Kafka Itself?

Where this fits in:

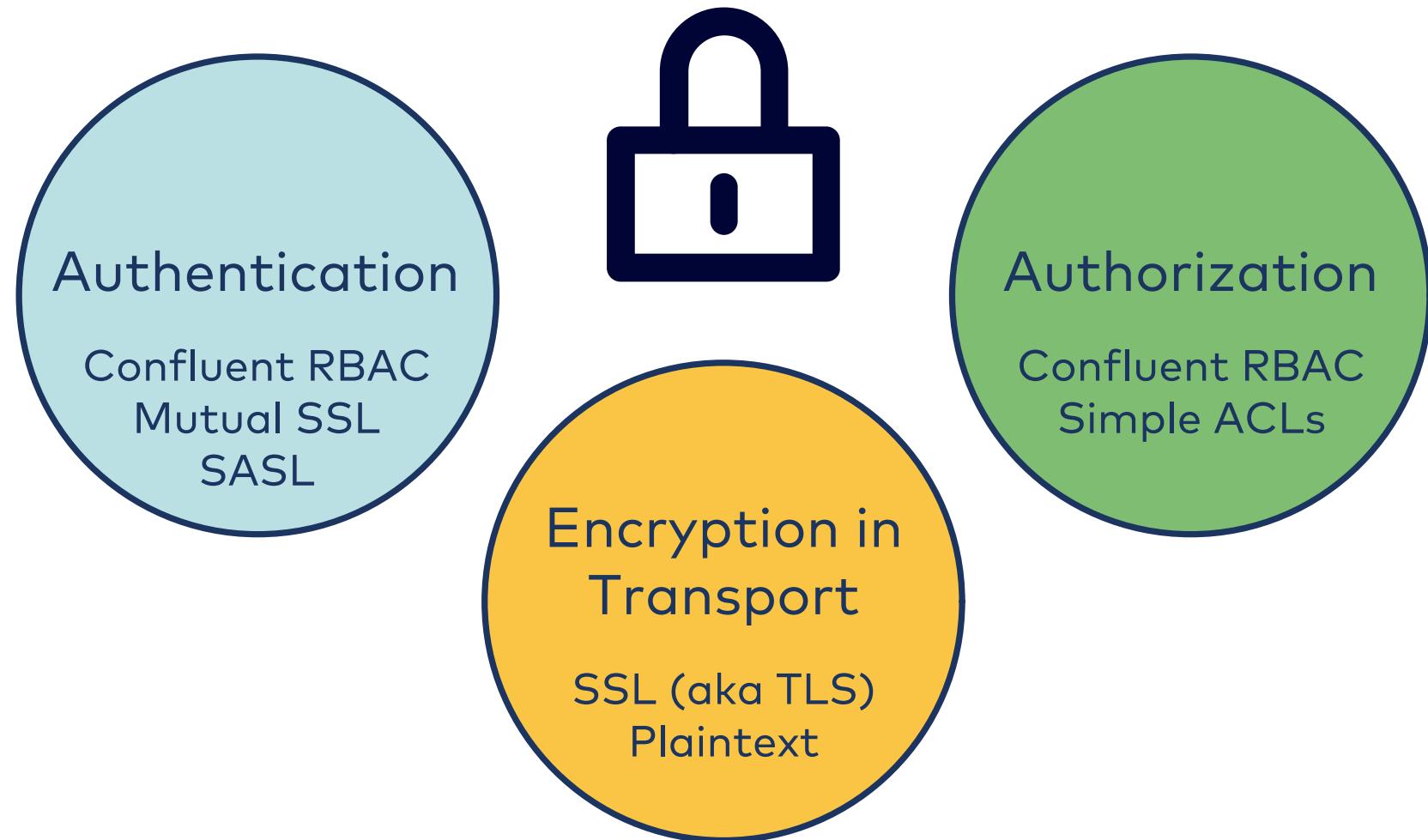
- Hard Prerequisite: Fundamentals Course
- Recommended Prerequisite: Managing a Kafka Cluster

9a: What are the Basic Ideas You Should Know about Kafka Security?

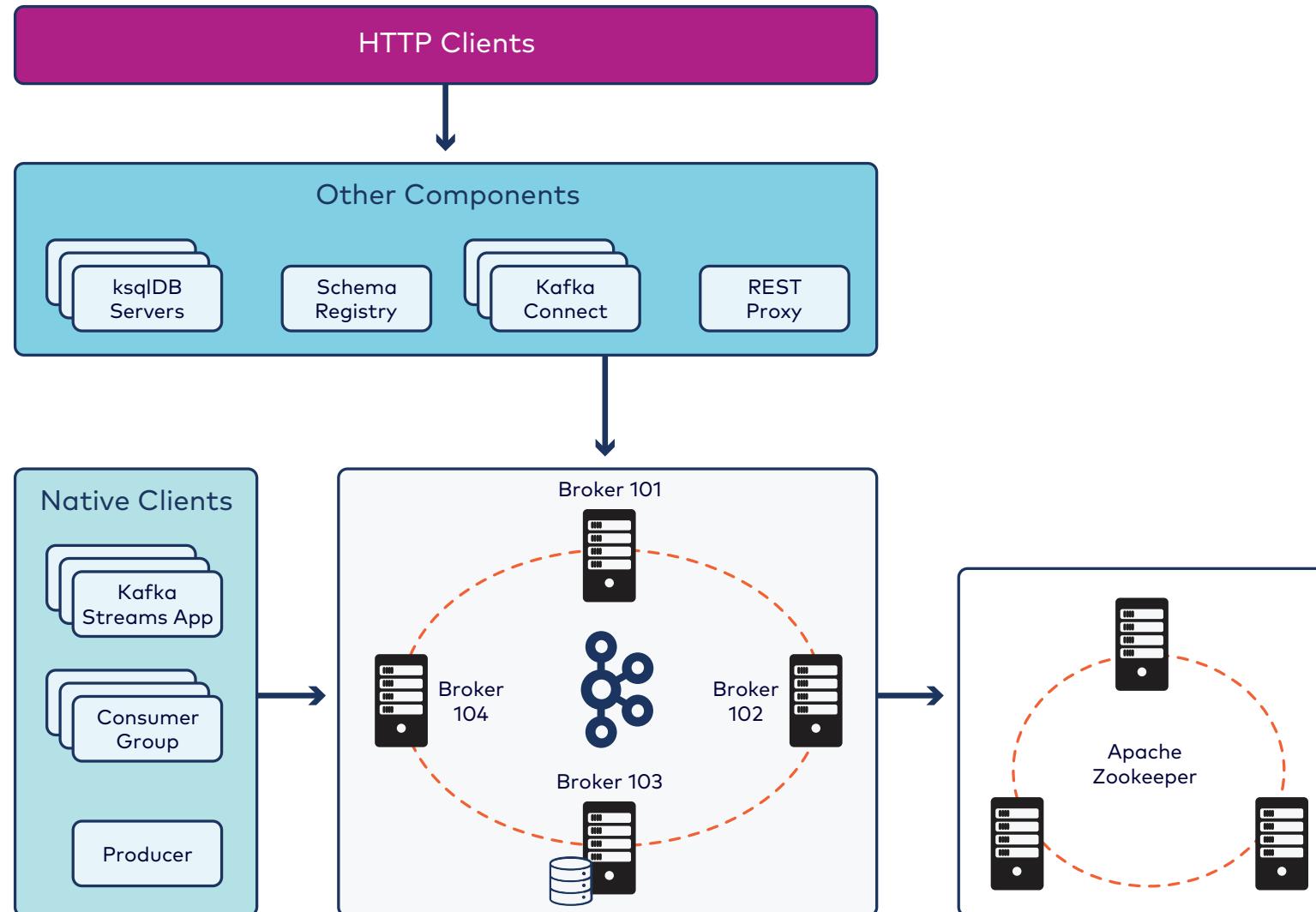
Description

Overview of security in Kafka. Authentication vs. authorization. Encryption. Points of vulnerability.

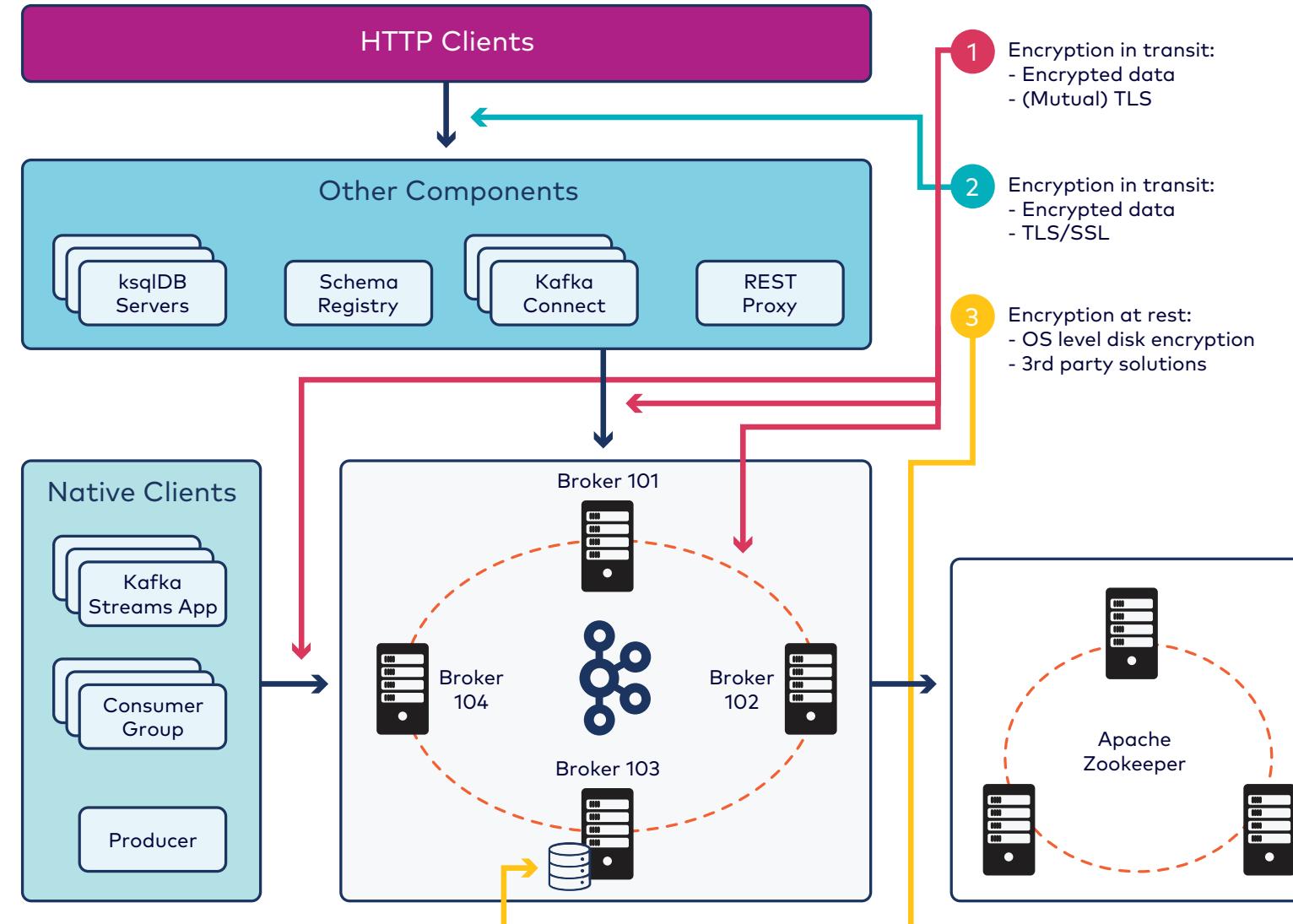
Security Overview



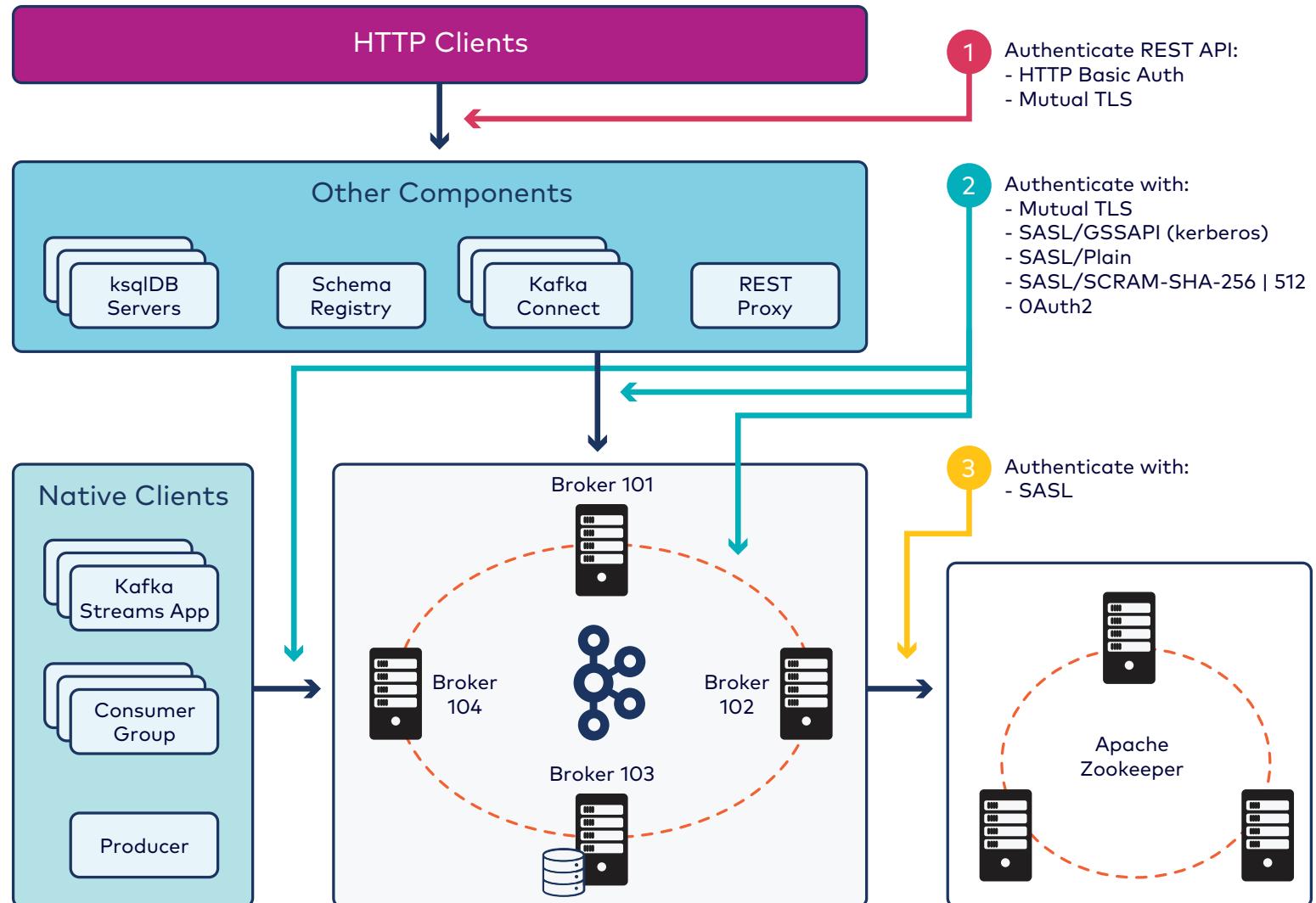
Security - Architecture



Security - Encryption at Rest & in Transit



Authentication



Broker Ports for Security

- Plain text (no wire encryption, no authentication)

```
listeners=PLAINTEXT://kafka-1:9092
```

- SSL (wire encryption, authentication)

```
listeners=SSL://kafka-1:9093
```

- SASL (authentication)

```
listeners=SASL_PLAINTEXT://kafka-1:9094
```

- SSL + SASL (SSL for wire encryption, SASL for authentication)

```
listeners=SASL_SSL://kafka-1:9095
```

- Clients choose **only one** port to use

If clients connect from an outside network, make sure to set

`advertised.listeners` in addition to `listeners` so that clients can discover the brokers



An Advanced Listeners Config Example

We can configure different listeners for different sources of traffic

- Useful to designate one interface for clients and one interface for replication traffic:

```
listeners = CLIENTS://kafka-1a:9092,REPLICATION://kafka-1b:9093
```

- Listeners can have any name as long as `listener.security.protocol.map` is defined to map each name to a security protocol:

```
listener.security.protocol.map = CLIENTS:SASL_SSL, REPLICATION:SASL_PLAINTEXT
```

9b: What Options Do You Have For Securing a Kafka/Confluent Deployment?

Description

Survey of security options.

SSL/TLS or SASL Manual Configuration

- Free
- But you have to do all of the work
- You will experience this in lab.



See the appendix for examples and details regarding SSL/TLS and SASL.

Confluent RBAC

- **Role Based Access Control**
- Paid Confluent feature
- Includes both authentication and authorization...
- ...by defined roles



Self-paced training available

Security - Confluent Cloud

- Security enabled **out-of-the-box**
- User → Cloud communication secured by TLS
- Data encrypted in motion & at rest
- CCloud is hosted in multiple AWS, GCP, and Azure regions

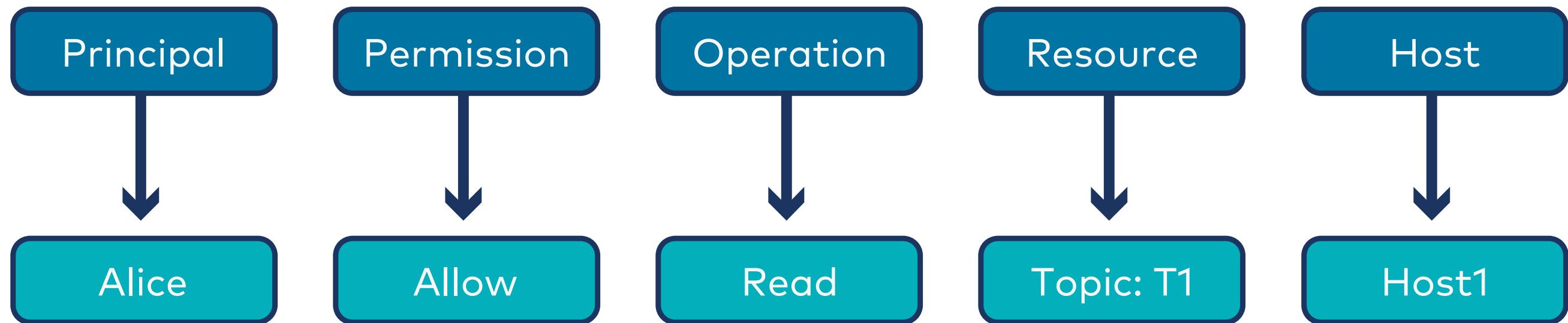
9c: How Can You Easily Control Who Can Access What?

Description

Components of Kafka ACL entries. How to add and remove ACLs. Wildcards.

Access Control Lists (ACLs)

ACL example: Alice is allowed to read data from topic `T1` from host `Host1`



Principal

- Type + name
- Supported types: User
 - `User:Alice`
- Extensible, so users can add their own types (e.g., group)

Permissions

- Allow and Deny
 - Deny takes precedence
 - Deny makes it easy to specify "everything but"
- By default, anyone without an explicit Allow ACL is denied

Operations and Resources

- Operations:
 - `Read, Write, Create, Describe, ClusterAction, All`
- Resources:
 - `Topic, Cluster, and ConsumerGroup`

Operations	Resources
<code>Read, Write, Describe</code>	Topic
<code>Read and Write imply Describe</code>	
<code>Read</code>	ConsumerGroup
<code>Create, ClusterAction</code> communication between controller and brokers	Cluster

Hosts

- Allows firewall-type security, even in a non-secure environment
 - Without needing system/network administrators to get involved

Configuring a Broker ACL

`SimpleAclAuthorizer` is the default authorizer implementation

```
authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer
```

- Provides a CLI for adding and removing ACLs
- ACLs are stored in ZooKeeper and propagated to brokers asynchronously
- ACLs are cached in the broker for better performance
- Make Kafka principal superusers
 - Or grant `ClusterAction` and `Read` on all Topics to the Kafka principal

Configuring ACLs - Producers

- `kafka-acls` can be used to add authorization
- Producer:
 - Grant `Write` on the topic, `Create` on the Cluster (for topic auto-creation)
 - Or use `--producer` option in the CLI

```
$ kafka-acls \
  --bootstrap-server kafka-1:9092 \
  --add \
  --allow-principal User:Bob \
  --producer \
  --topic my_topic
```

Configuring ACLs - Consumers

- Consumer:
 - Grant `Read` on the topic, `Read` on the ConsumerGroup
 - Or use the `--consumer` option in the CLI

```
$ kafka-acls \
  --bootstrap-server kafka-1:9092 \
  --add \
  --allow-principal User:Bob \
  --consumer \
  --topic my_topic \
  --group group1
```

Wildcard Support (1)

- Allow user **Jane** to produce to any topic whose name starts with "Test-"

```
$ kafka-acls \
--bootstrap-server kafka-1:9092 \
--add \
--allow-principal User:Jane \
--producer --topic Test- \
--resource-pattern-type prefixed
```

- Allow all users **except** **BadBob** and all hosts **except** **198.51.100.3** to read from **Test-topic**:

```
$ kafka-acls \
--bootstrap-server kafka-1:9092 --add \
--allow-principal User:'*' \
--allow-host '*' \
--deny-principal User:BadBob \
--deny-host 198.51.100.3 \
--operation Read --topic Test-topic
```

Wildcard Support (2)

- List all ACLs for the topic `Test-topic`:

```
$ kafka-acls \
--bootstrap-server kafka-1:9092,kafka-2:9092 \
--list --topic Test-topic \
--resource-pattern-type match
```

9d: What Should You Know Securing a Deployment Beyond Kafka Itself?

Description

Securing the whole environment.

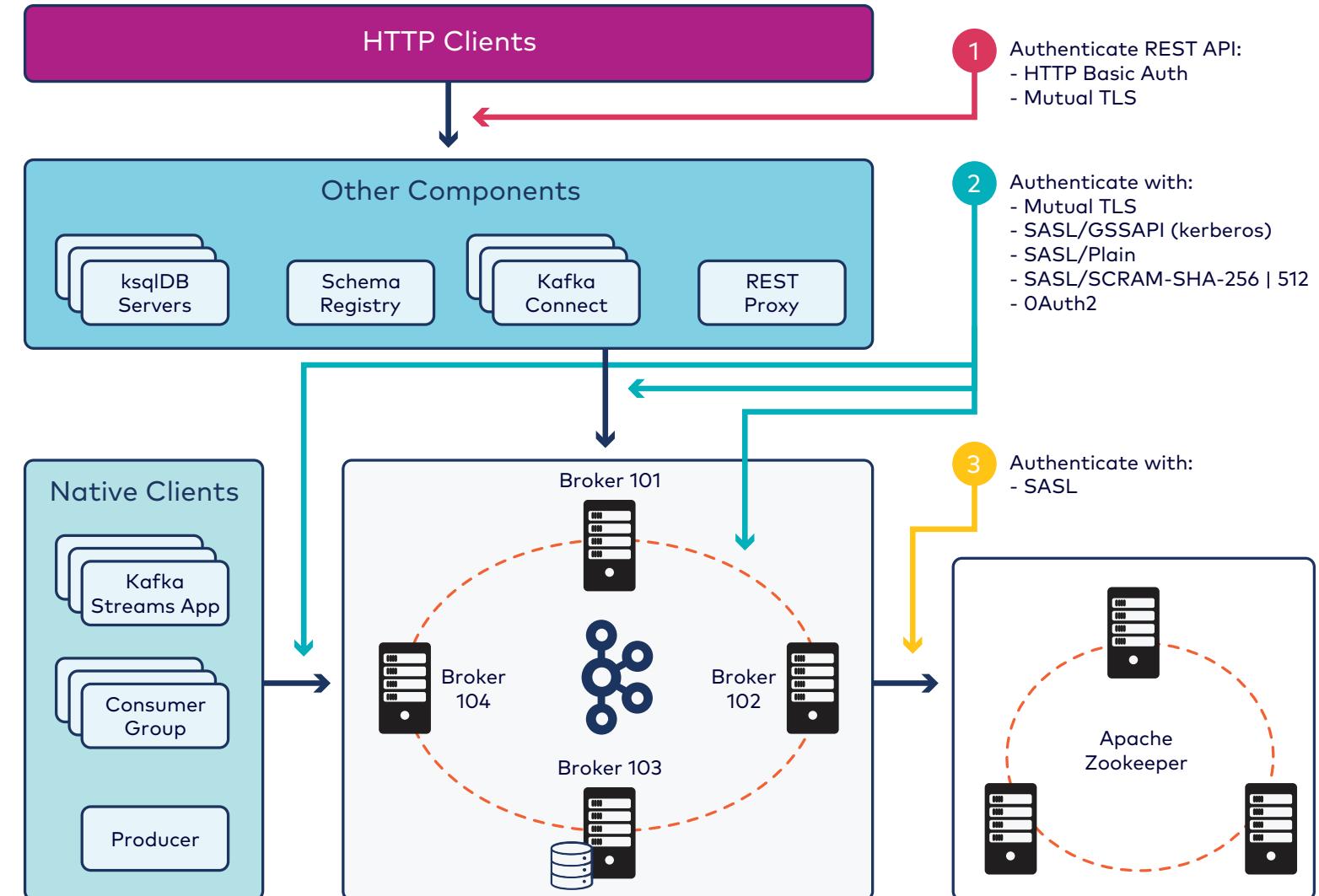
ZooKeeper Security is Important

- ZooKeeper stores:
 - Critical Kafka metadata
 - ACLs
- We need to prevent untrusted users from modifying this data



See your student handbook for some technical details.

Securing Schema Registry and REST Proxy



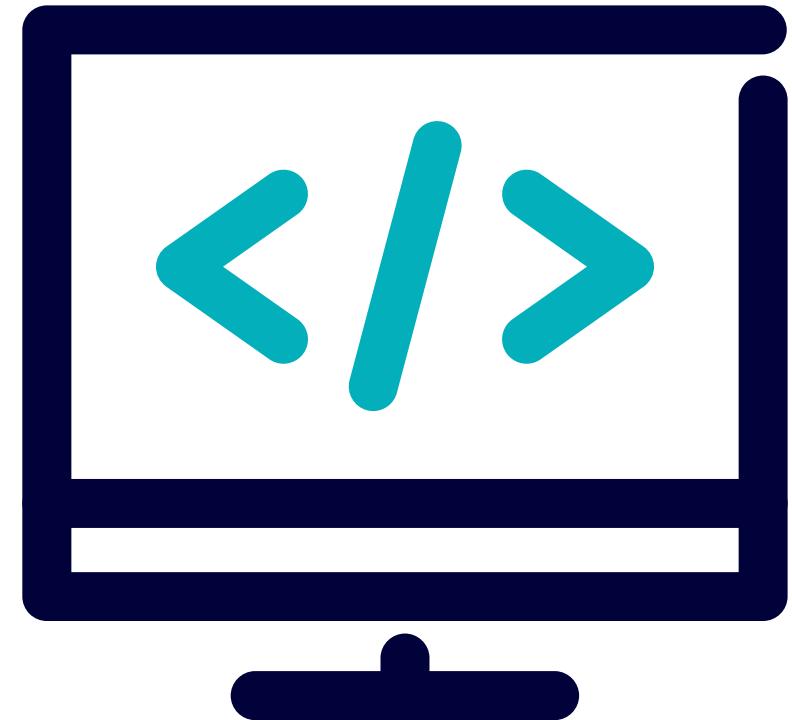
Securing the Schema Registry

- Secure communication between REST client and Schema Registry (HTTPS):
 - HTTP Basic Authentication
 - SSL (transport)
- Secure transport and authentication between the Schema Registry and the Kafka cluster:
 - SSL (transport)
 - SASL (authentication)
 - Mutual SSL (transport + authentication)
- Confluent Enterprise **security plugin**:
 - Restricts schema evolution to administrative users
 - Client application users get read-only access

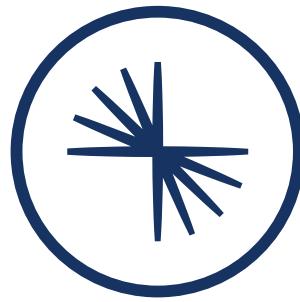
Lab: Securing the Kafka Cluster

Please work on **Lab 9a: Securing the Kafka Cluster**

Refer to the Exercise Guide



10: Understanding Kafka Connect



CONFLUENT
Global Education

Module Overview



This module contains 4 lessons:

- What Can You Do with Kafka Connect?
- How Do You Configure Workers and Connectors?
- Deep Dive into a Connector & Finding Connectors
- What Else Can One Do With Connect?

Where this fits in:

- Hard Prerequisite: Fundamentals Course
- Recommended Prerequisite: Consumer Groups and Load Balancing

10a: What Can You Do with Kafka Connect?

Description

Motivating what Connect can do and why to use it over self-made solutions. Motivating how it can “factor out” common behavior yet leverage Connectors. Connectors vs. tasks vs. workers. Relating Connect to other components of Kafka and how it works at a high level, e.g., scalability, converters, offsets.

Wanted: Data From Another System in Kafka; Kafka Data To Another System

Suppose you have

- Some data in some other system and you want to get it into Kafka
- Some data in Kafka and want to export it to another system

Your development team could program custom producers or consumers with hooks into the other system to make this happen...

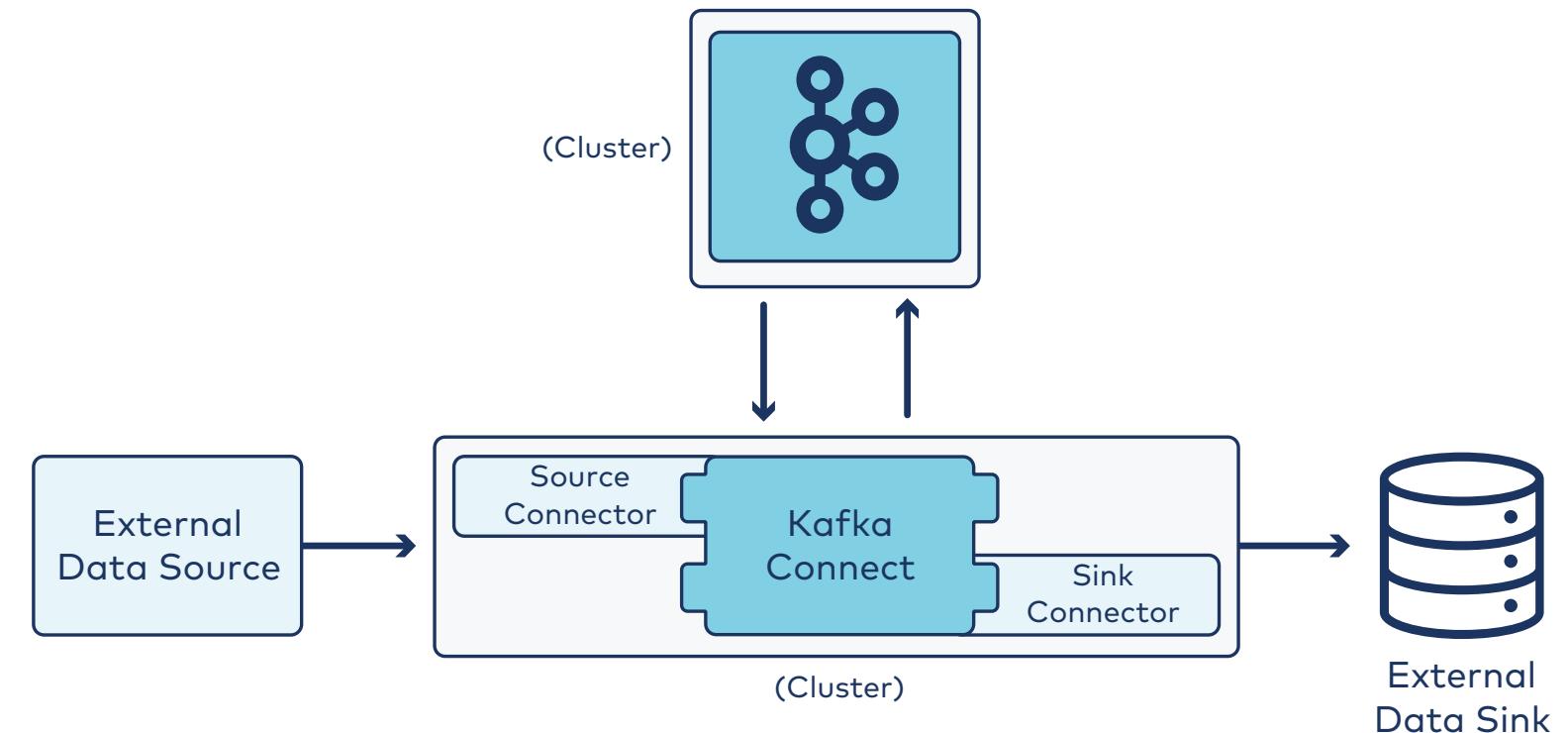
But... there's a better way...

Kafka Connect to the Rescue!

Kafka Connect does the work for us!

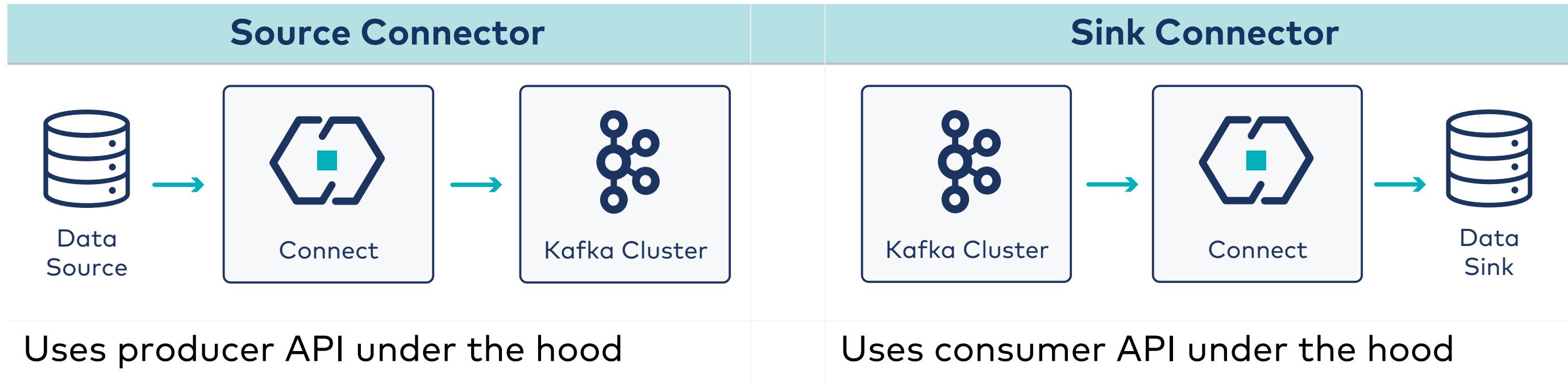
All copying behavior is in Kafka Connect.

Plugins called **Connectors** contain the logic specific to particular external systems.

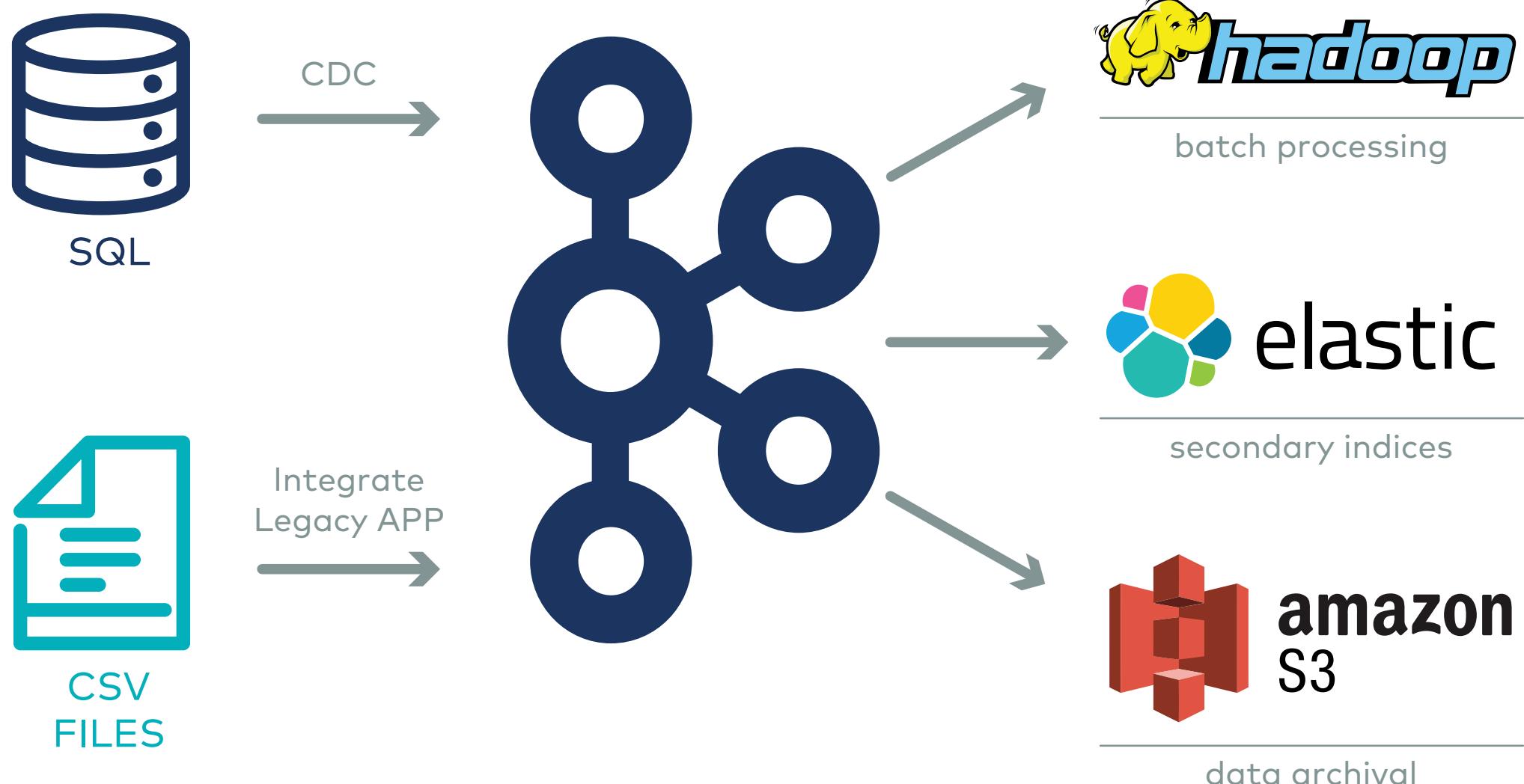


Sources and Sinks

Two kinds of connectors...



Use Cases



10b: How Do You Configure Workers and Connectors?

Description

Configuration of workers in distributed mode and configuration of connectors in general. Quick overview of standalone mode differences.

Configuring Connectors

Name	Description	Default
name	Connector's unique name	
connector.class	Name of the Java bytecodes file for the connector	
tasks.max	Maximum number of tasks to create - if possible	1
key.converter	Converter to (de)serialize keys	(worker setting)
value.converter	Converter to (de)serialize values	(worker setting)
topics	For sink connectors only , comma-separated list of topics to consume from	

10c: Deep Dive into a Connector & Finding Connectors

Description

Details of the JDBC Source Connector, configuration details, working through why one would do certain configs with examples. Finding Connectors on Confluent Hub.

JDBC Source Connector

- Java Database Connectivity (JDBC) API is common amongst databases.
- JDBC Source Connector is a great way to get database tables into Kafka topics.
- JDBC Source periodically polls a relational database for new or recently modified rows.
 - Creates a record for each row, and Produces that record as a Kafka message.
- Each table gets its own Kafka topic.
- New and deleted tables are handled automatically.

JDBC Source Connector Config Example

```
1 {
2   "name": "Driver-Connector",
3   "config": {
4     "connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",
5     "connection.url": "jdbc:postgresql://postgres:5432/postgres",
6     "connection.user": "postgres",
7     "table.whitelist": "driver",
8     "topic.prefix": "",
9     "mode": "timestamp+incrementing",
10    "incrementing.column.name": "id",
11    "timestamp.column.name": "timestamp",
12    "table.types": "TABLE",
13    "numeric.mapping": "best_fit",
14  }
15 }
```

Other Connectors

Search Confluent Hub at confluent.io/hub for connectors!

The screenshot shows the Confluent Hub homepage with a search bar and a list of results. On the left, there are filters for Plugin type (Sink, Source, Transform, Converter), Enterprise support (Confluent supported, Partner supported, None), Verification (Confluent built, Confluent Tested, Verified gold, Verified standard, None), and License (Commercial, Free). The results section shows two connectors: "Kafka Connect GCP Pub-Sub" (Source Connector) and "Kafka Connect S3" (Sink Connector). Both connectors are marked as available on Confluent Cloud.

Confluent Hub

Discover Kafka® connectors
and more

What plugin are you looking for?

Filters

Plugin type

Sink

Source

Transform

Converter

Enterprise support

Confluent supported

Partner supported

None

Verification

Confluent built

Confluent Tested

Verified gold

Verified standard

None

License

Commercial

Free

Results (158)

+ Submit a plugin

Kafka Connect GCP Pub-Sub

A Kafka Connect plugin for GCP Pub-Sub

Available fully-managed on Confluent Cloud

Enterprise support: Confluent supported

Installation: Confluent Hub CLI, Download

Verification: Confluent built

Author: Confluent, Inc.

License: Commercial

Version: 1.0.2

Kafka Connect S3

The S3 connector, currently available as a sink, allows you to export data from Kafka topics to S3 objects in either Avro or JSON formats

Available fully-managed on Confluent Cloud

Enterprise support: Confluent supported

Installation: Confluent Hub CLI, Download

Verification: Confluent built

Author: Confluent, Inc.

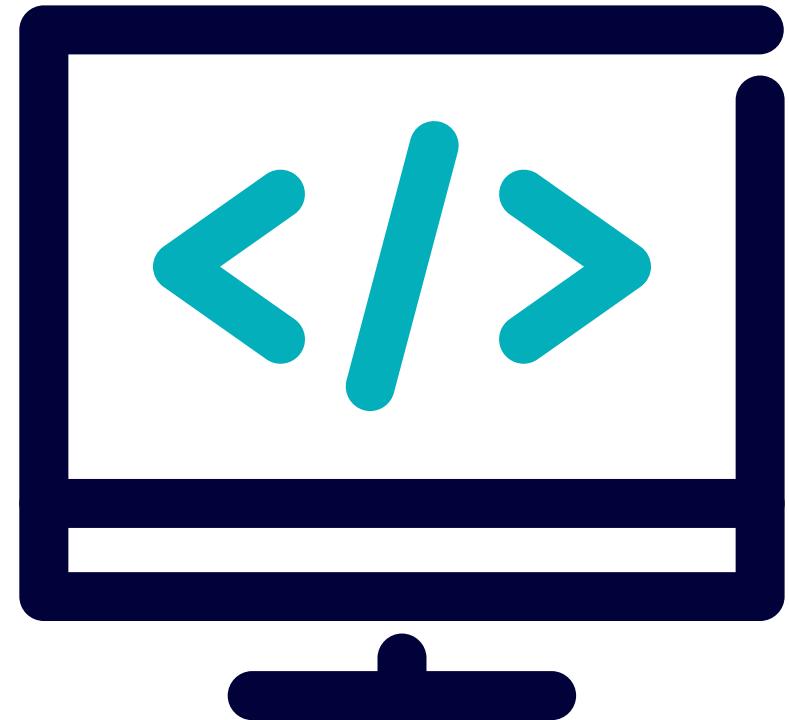
License: Free

Version: 5.5.1

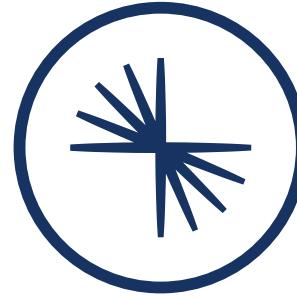
Lab: Running Kafka Connect

Please work on **Lab 10a: Running Kafka Connect**

Refer to the Exercise Guide



11: Deploying Kafka in Production



CONFLUENT
Global Education

Module Overview

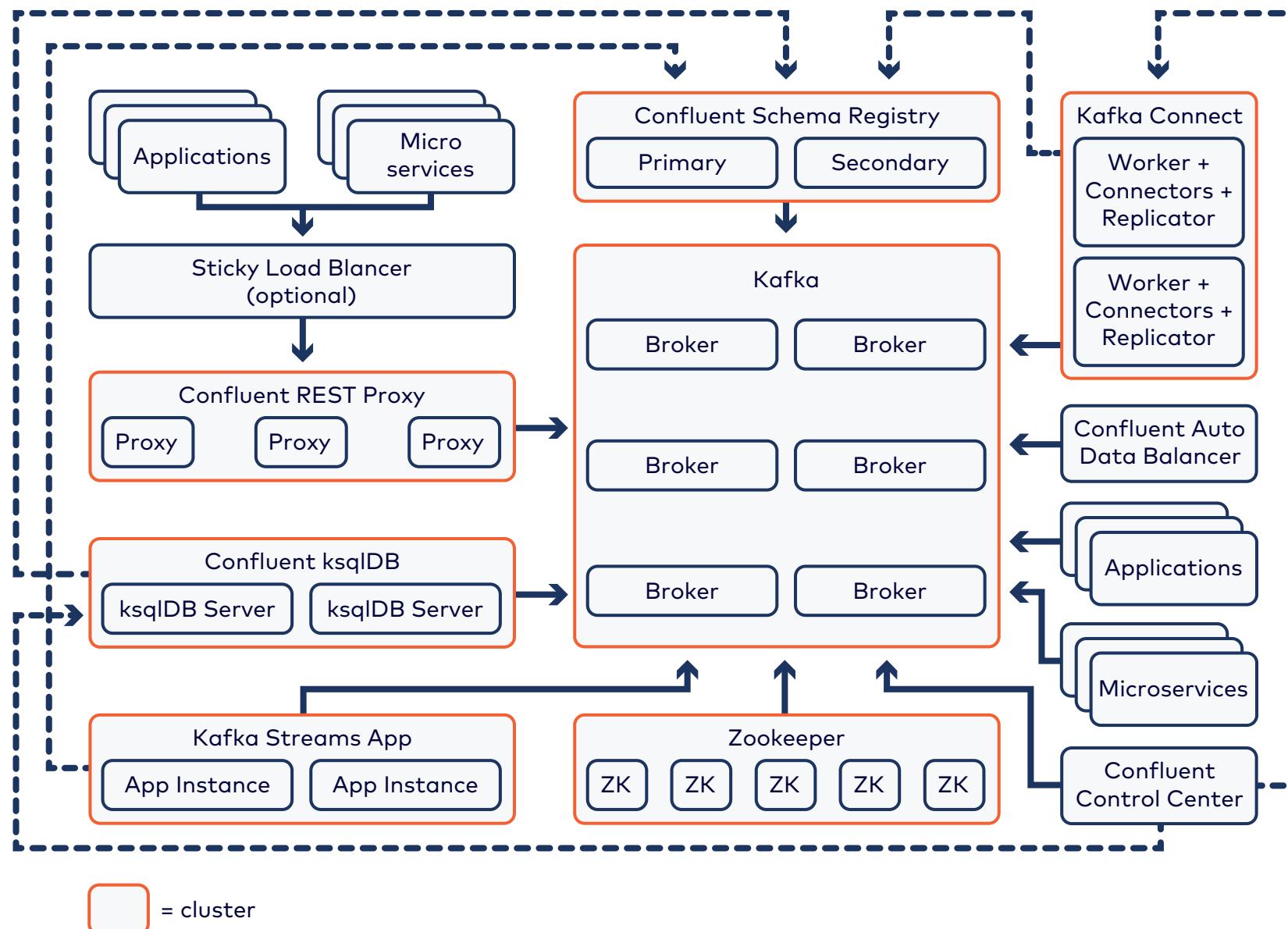


This module contains 7 lessons:

Each lesson enables you to answer what Confluent advises for deploying each of the following in production:

- Kafka
- ZooKeeper
- Kafka Connect
- Confluent Schema Registry
- Confluent REST Proxy
- Kafka Streams and ksqlDB
- Confluent Control Center

Kafka Reference Architecture

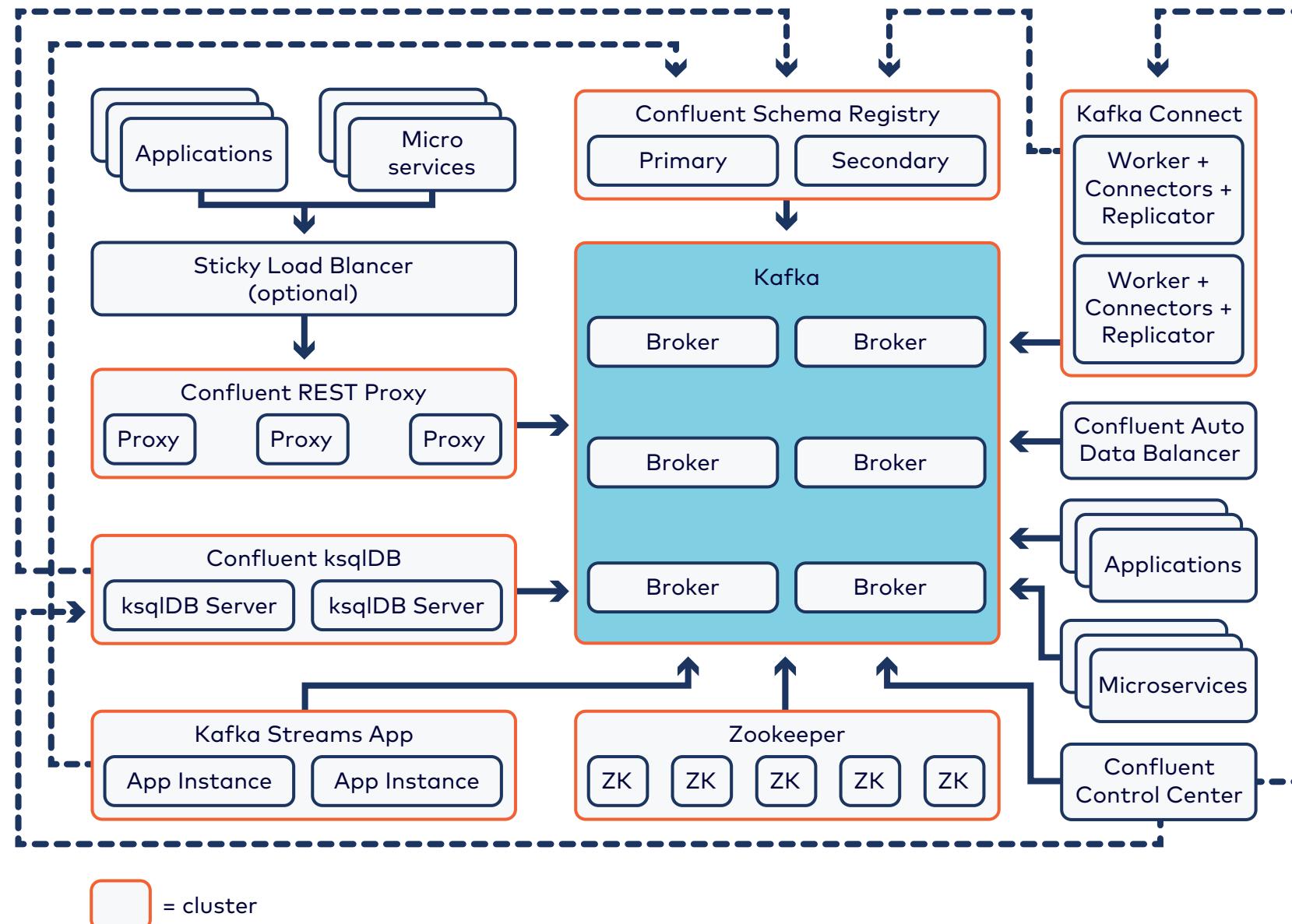


11a: What Does Confluent Advise for Deploying Brokers in Production?

Description

Best practices and capacity planning for brokers in production.

Kafka Reference Architecture: Brokers



Broker Design

- Run on dedicated servers
- n brokers \rightarrow replication factor up to n
- Can use virtual IP + load balancer as `bootstrap.servers`
 - Pro: don't have to change client config code when cluster changes
 - Con: More infrastructure to worry about
- Discussion Questions:
 1. What replication factor is acceptable for mission critical data?
 2. How many brokers do you need to accommodate this?
 3. With that many brokers, how many can fail without permanent data loss? What would happen to write access and read access?

Capacity Planning: Brokers

- Disk space and I/O
- Network bandwidth
- RAM (for page cache)
- CPU

Broker Disk

- 12 x 1TB filesystems mounted to data directories for topic data + separate disks for OS
 - A single partition can only live on **one** volume
 - Partitions are balanced across `log.dirs` in round-robin
 - RAID-10 optional
- Use XFS or EXT4 filesystems
 - Mount with `noatime`

Network Bandwidth

- Gigabit Ethernet sufficient for smaller applications
- 10Gb Ethernet needed for large installations
- Enable compression on producers
- Optional: Isolate Broker-ZooKeeper traffic to separate network

Broker Memory

- JVM heap ~ 6 GB
- OS ~ 1 GB
- Page cache:
 - Lots and lots!
 - What might your consumer lag be?

Open File Descriptors and Client Connections

- Brokers can have a **lot** of open files
 - `$ ulimit -n 100000`
- Brokers can have a **lot** of client connections:
 - `max.connections.per.ip` (Default: 2 billion)

Broker CPU

- Dual 12-core sockets
- Relevant broker properties:
 - `num.io.threads` (Default: 8)
 - `num.network.threads` (Default: 3)
 - `num.recovery.threads.per.data.dir`
(Default: 1)
 - `background.threads` (Default: 10)
 - `num.replica.fetchers` (Default: 1)
 - `log.cleaner.threads` (Default: 1)
- Discussion:
 - Given 12 data disks and dual 12-core CPU sockets, how would you modify the default broker threading properties?

Capacity Planning: Number of Brokers

- Storage

Number of brokers =

(messages per day * message size * Retention days * Replication) / (disk space per Broker)

- Network bandwidth

Number of brokers =

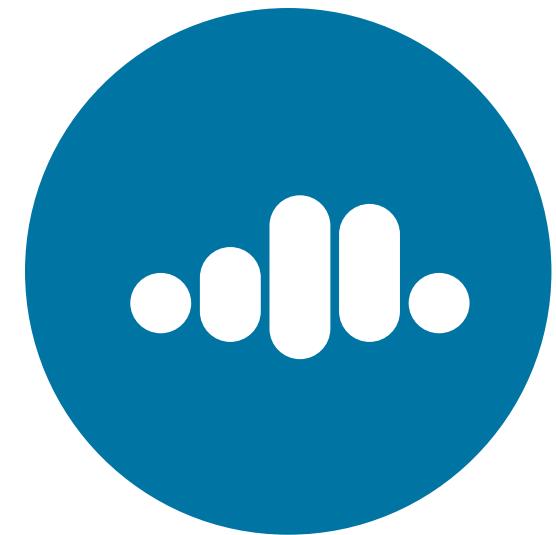
(messages per sec * message size * Number of Consumers) / (Network bandwidth per Broker)



Maximums: 4,000 partitions per broker and 200,000 partitions per cluster.

Deploying Kafka in the Cloud

- Self-managed cloud deployment:
 - Memory optimized compute instances
 - Multiple availability zones (`broker.rack`)
 - Private subnet for inter-broker traffic
 - Private subnet for ZooKeeper
 - Lockdown firewall rules, Kafka security
 - For AWS: "EBS optimized" instances
- Or consider:



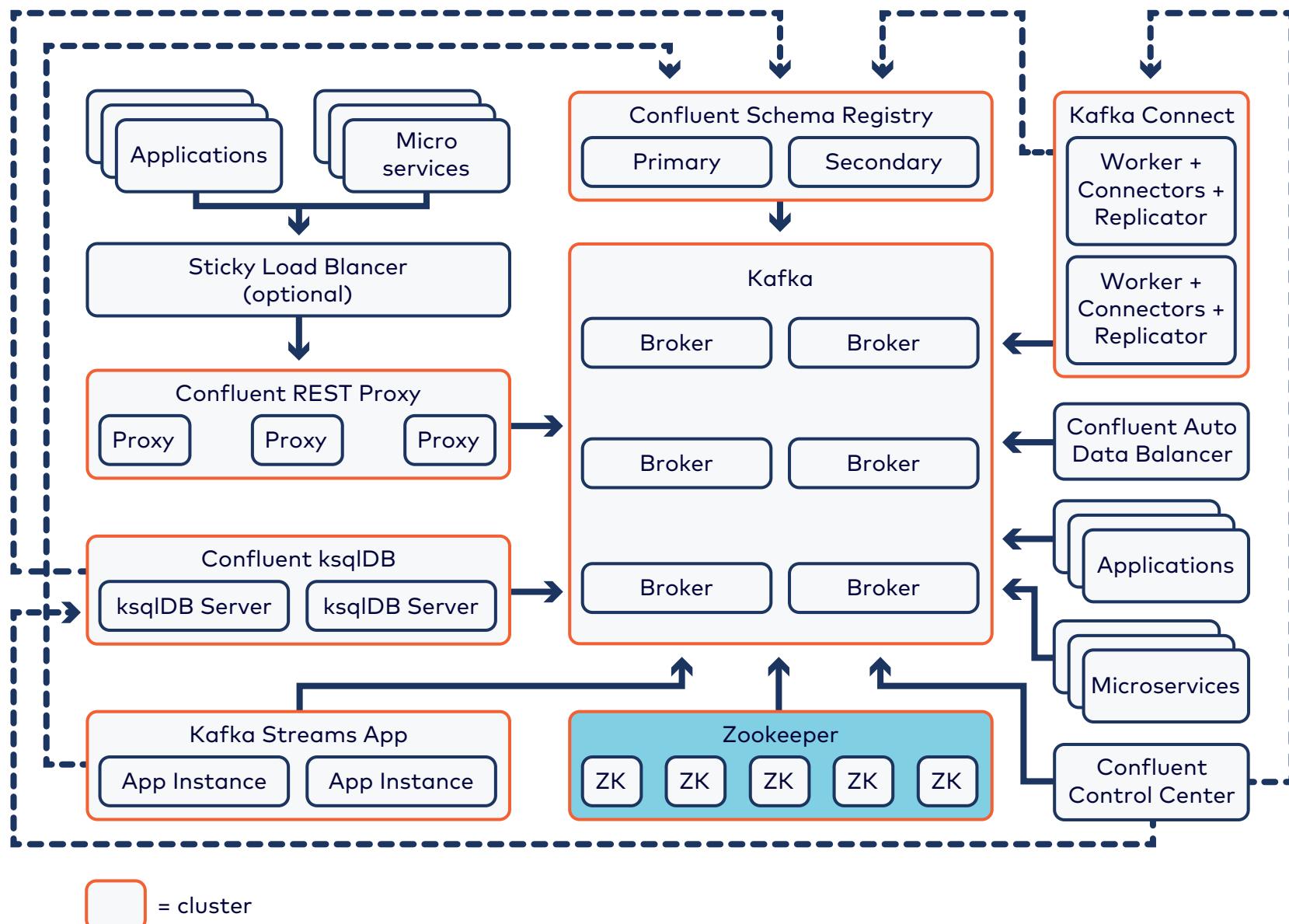
Virtual cores are weaker than physical cores

11b: What Does Confluent Advise for Deploying Zookeeper in Production?

Description

What Does Confluent Advise for Deploying Zookeepr in Production?

Kafka Reference Architecture: ZooKeeper



Capacity Planning: ZooKeeper

- Must have **odd** number of ZooKeeper instances:
 - ZooKeeper works by quorum, i.e., majority votes
 - 3 nodes allow for one node failure
 - 5 nodes allow for two node failures (recommended)
 - Deploy across **availability zones**
- Hardware:
 - RAM: 8 GB
 - Transaction log (`dataLogDir`): 512 GB SSD
 - Database snapshots (`dataDir`): 2 TB RAID 10
 - CPU: minimal 2-4 cores

Disk Space on ZooKeeper

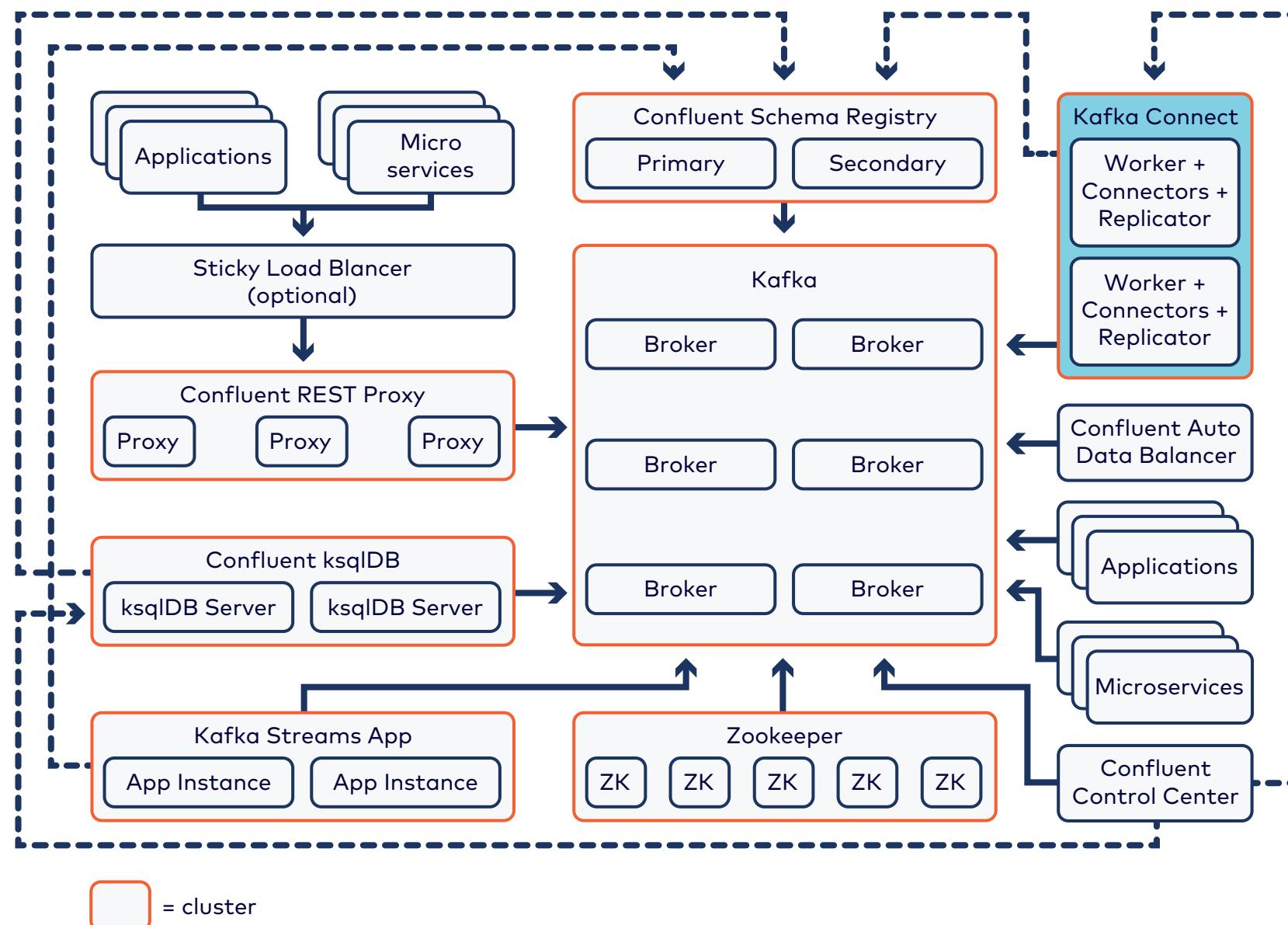
- ZooKeeper saves **snapshots** and **transactional log** files
- Transaction log should have a dedicated SSD
- Configure ZooKeeper to purge snapshots in its `server.properties` file:
 - `autopurge.snapRetainCount`: number of snapshots to retain
 - `autopurge.purgeInterval`: hours between purges

11c: What Does Confluent Advise for Deploying Kafka Connect in Production?

Description

Best practices and capacity planning for Kafka Connect in production.

Reference Architecture: Kafka Connect



Connect Workers for High Availability

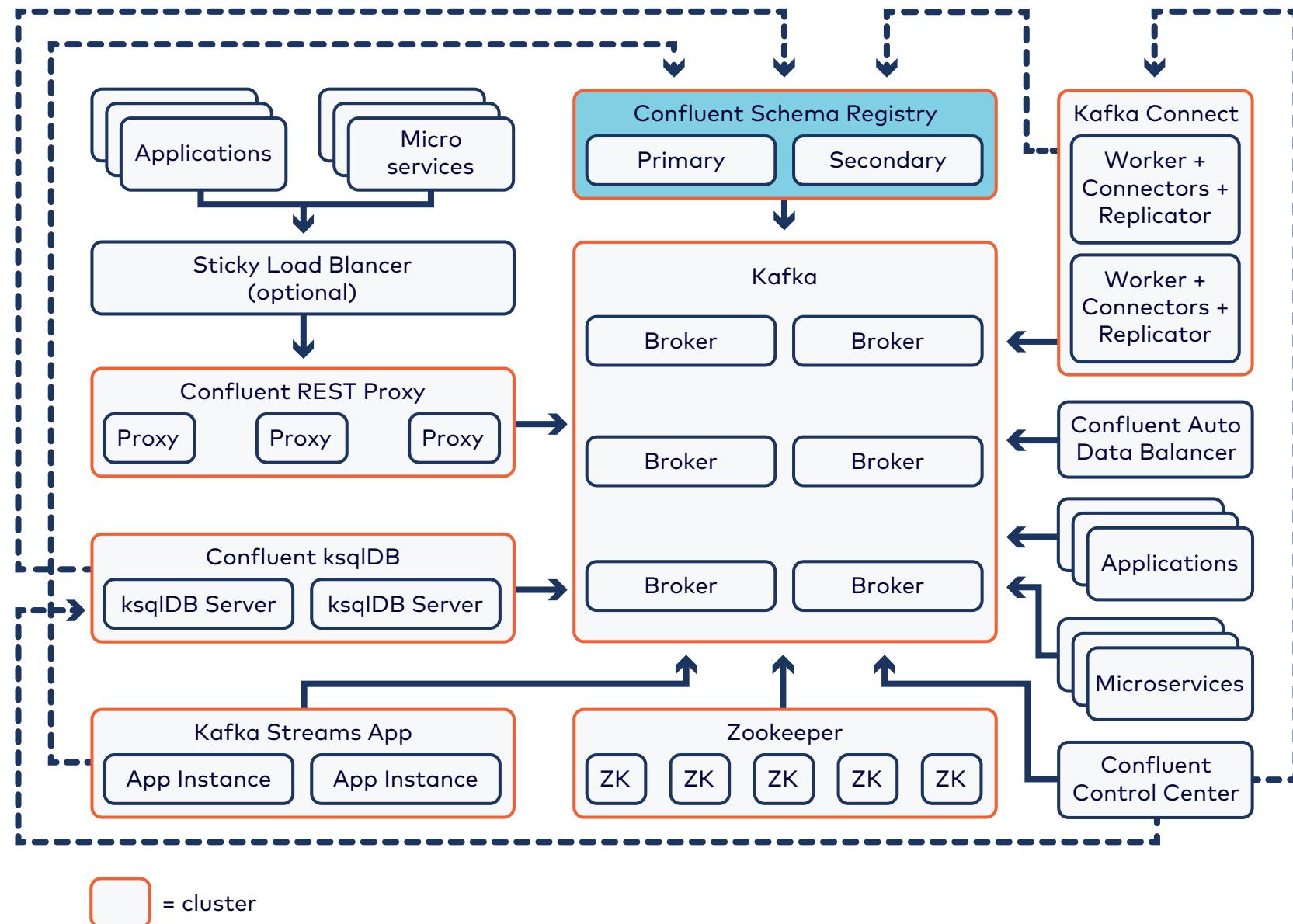
- Deploy machines with same `group.id` to form cluster
- Deploy at least 2 machines behind load balancer
- Add machines with same `group.id` to add capacity

11d: What Does Confluent Advise for Deploying Schema Registry in Production?

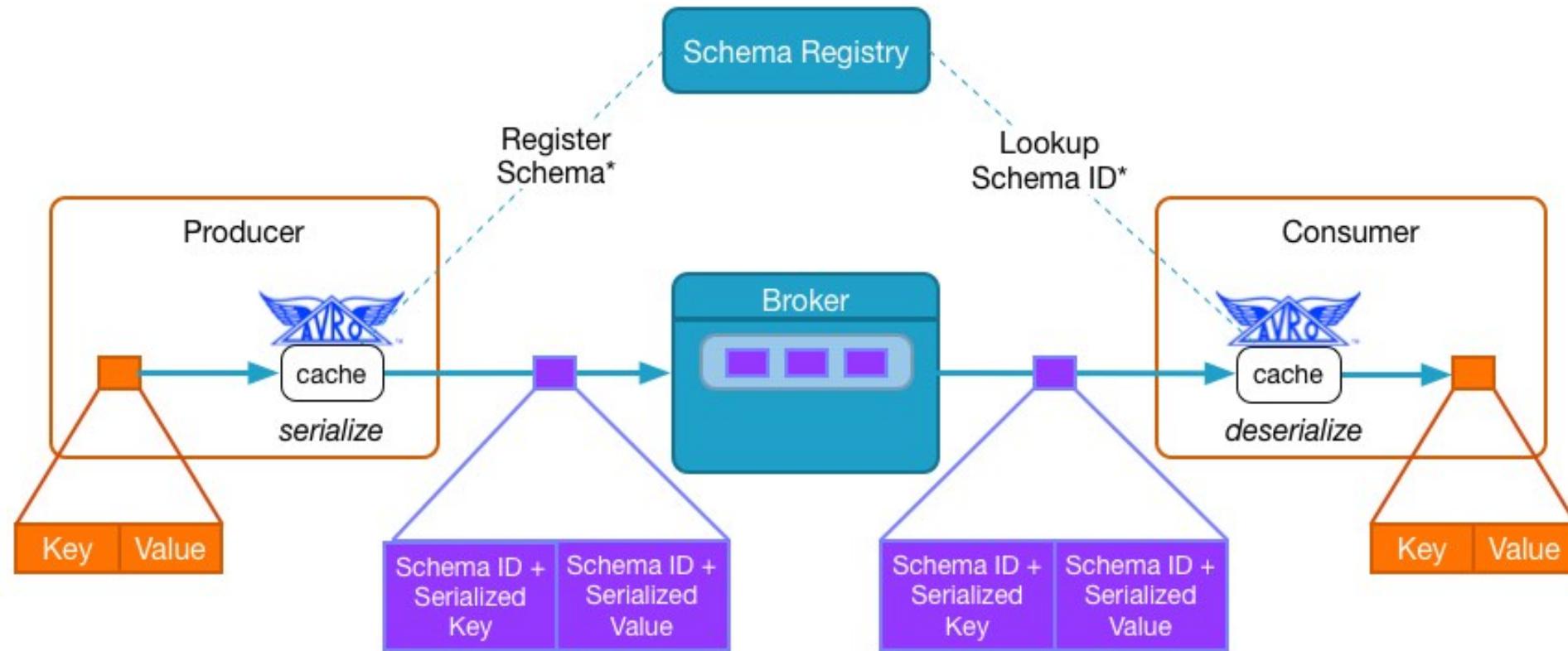
Description

Best practices and capacity planning for Schema Registry in production.

Reference Architecture: Confluent Schema Registry



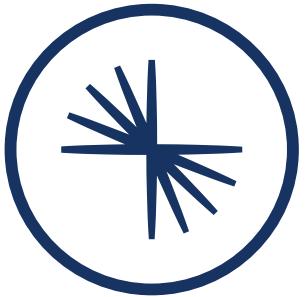
Schema Registry Overview



Capacity Planning: Schema Registry

- Minimal system requirements
- Deploy 2+ servers behind load balancer
- Single-primary architecture
 - One primary node at a time
 - Primary node responds to write requests
 - Secondary nodes forward write requests to primary node
 - All nodes respond to read requests

Conclusion



CONFLUENT
Global Education

Course Contents



Now that you have completed this course, you should have the skills to:

- Describe how Kafka brokers, producers, and consumers work
- Describe how replication works within the cluster
- Understand hardware and runtime configuration options
- Monitor and administer your Kafka cluster
- Integrate Kafka with external systems using Kafka Connect
- Design a Kafka cluster for high availability & fault tolerance

Other Confluent Training Courses

- Confluent Developer Skills for Building Apache Kafka®
- Confluent Stream Processing Using Apache Kafka® Streams & ksqlDB
- Confluent Advanced Skills for Optimizing Apache Kafka®
- Managing Data in Motion with Confluent Cloud



For more details, see <https://confluent.io/training>

Confluent Certified Developer for Apache Kafka

Duration: 90 minutes

Qualifications: Solid understanding of Apache Kafka and Confluent products, and 6-to-9 months hands-on experience

Availability: Live, online, 24-hours a day!

Cost: \$150

Register online: www.confluent.io/certification



Confluent Certified Administrator for Apache Kafka

Duration: 90 minutes

Qualifications: Solid work foundation in Confluent products and 6-to-9 months hands-on experience

Availability: Live, online, 24-hours per day!

Cost: \$150

Register online: www.confluent.io/certification



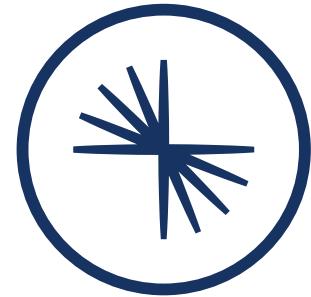
We Appreciate Your Feedback!



Please complete the course survey now.

Thank You!

Appendix: Additional Problems to Solve



CONFLUENT
Global Education

Overview

This section contains a few additional problems to be solved that will reinforce the concepts in this course.

These problems were originally written as warm-up problems for instructor-led training for this course. Your instructor may or may not choose to incorporate some or all of these problems in class; you may find them to provide additional enrichment in any case. Some other problems originally created as warm-up problems have been adapted into activities in the content of this version of this course.

Problem A: Partitioning with Keys

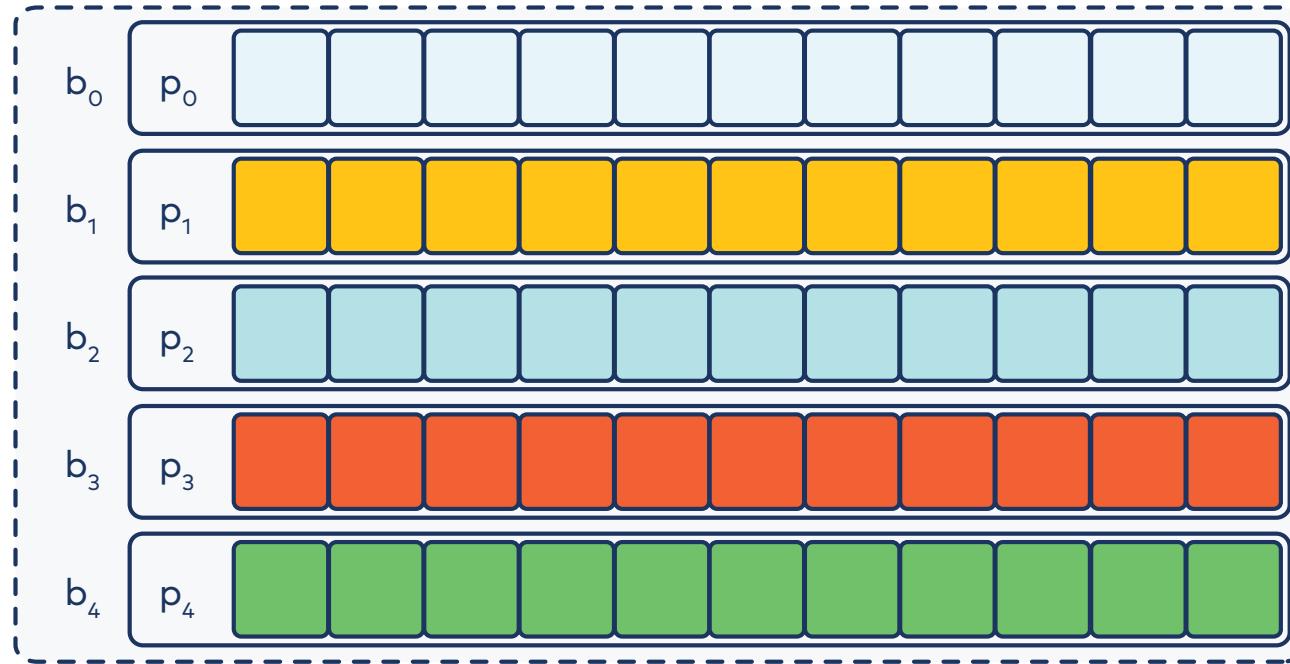
Suppose we have 5 brokers and 5 partitions. The five partitions are p_0, p_1, p_2, p_3 , and p_4 ; and they are stored on brokers b_0, b_1, b_2, b_3 , and b_4 , respectively. Suppose we are using default Kafka settings and $\text{hash}(n) = n$. Then...

Suppose we send these messages:

- m_0 with key 1
- m_1 with key 7
- m_2 with key 12
- m_3 with key 18
- m_4 with key 27
- m_5 with key 10

Which partitions contain which messages after the Kafka cluster has received them all?

Here's an illustration of the situation:



Problem B: Partitions and More

Building on the last....

- a. Suppose all given partitions are from the same topic. Suppose we add 2 partitions. Give a plausible scenario for right after we make this change.
- b. Suppose we set replication factor to 3. For **one** partition, what would be different?
- c. Repeat the last question but with replication factor of 9.
- d. Now suppose we set **acks** to all. For the partition from (b) as you've illustrated it, suppose a producer sent a message that went to that partition. What happens?

Problem C: Log Segment Files

Consider this log:

offset	3	6	8	9	17	26	27	28	29	30	31	32	33	34	35
key	2	1	12	11	10	7	9	1	7	15	7	15	7	12	6
inactive clean segment 1 size: 10				inactive clean segment 2 size: 10				inactive dirty segment 3 size: 15				active segment size: 8			

Then:

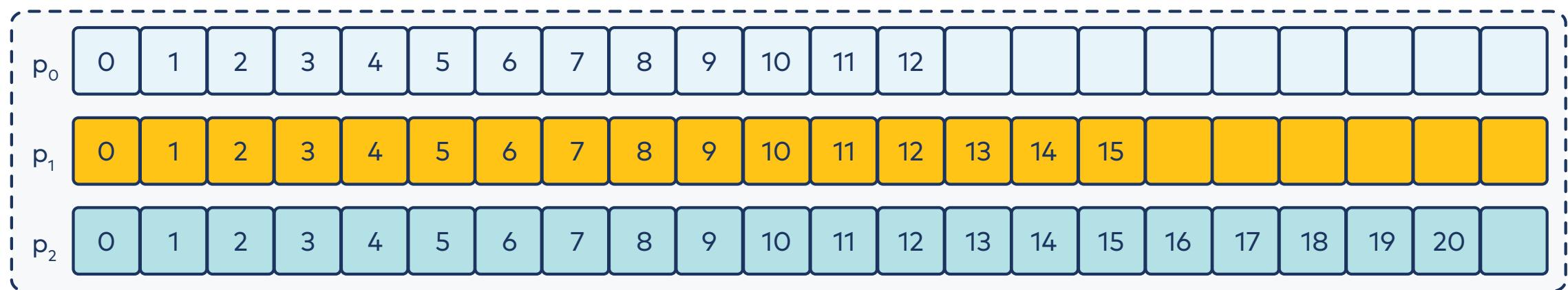
1. How many files will there be to store this log alone?
2. What might a possible list of filenames be?

Problem D: Groups, Consumers, and Partitions

Suppose we have partitions p_0, p_1, p_2 each with first offset 0 and with most recent offsets of 12, 15, and 20, respectively (where every offset between contains data).

Suppose we have consumer group g_0 with consumers c_0, c_1 , and c_2 and consumer group g_1 with consumers c_3 , and c_4 . (Some details may be missing - you interpret!)

The setup might look like this:



Your quest:

- a. Give a valid assignment of consumers to partitions that could result.
- b. How many different consumer offsets are stored in this scenario? Explain/list them.
- c. For each consumer offset in your list, give a valid value it could have
- d. Suppose c_1 , goes down. What happens? Concretely illustrate the scenario now.
- e. For any one consumer, tell the offsets of the messages read in the case that it gets 2 messages in the next batch. Repeat for some other consumer in the case that this other consumer gets 3 messages in the next batch.
- f. Give two examples of things that could happen that would trigger a consumer/partition assignment rebalance. One must not involve anything with consumers changing (before the rebalance).

Problem E: Compaction: A Deeper Dive

Suppose we have a log like this for a partition:

offset	3	6	8	9	17	26	27	28	29	30	31	32	33	34	35
key	2	1	12	11	10	7	9	1	7	15	7	15	7	12	6

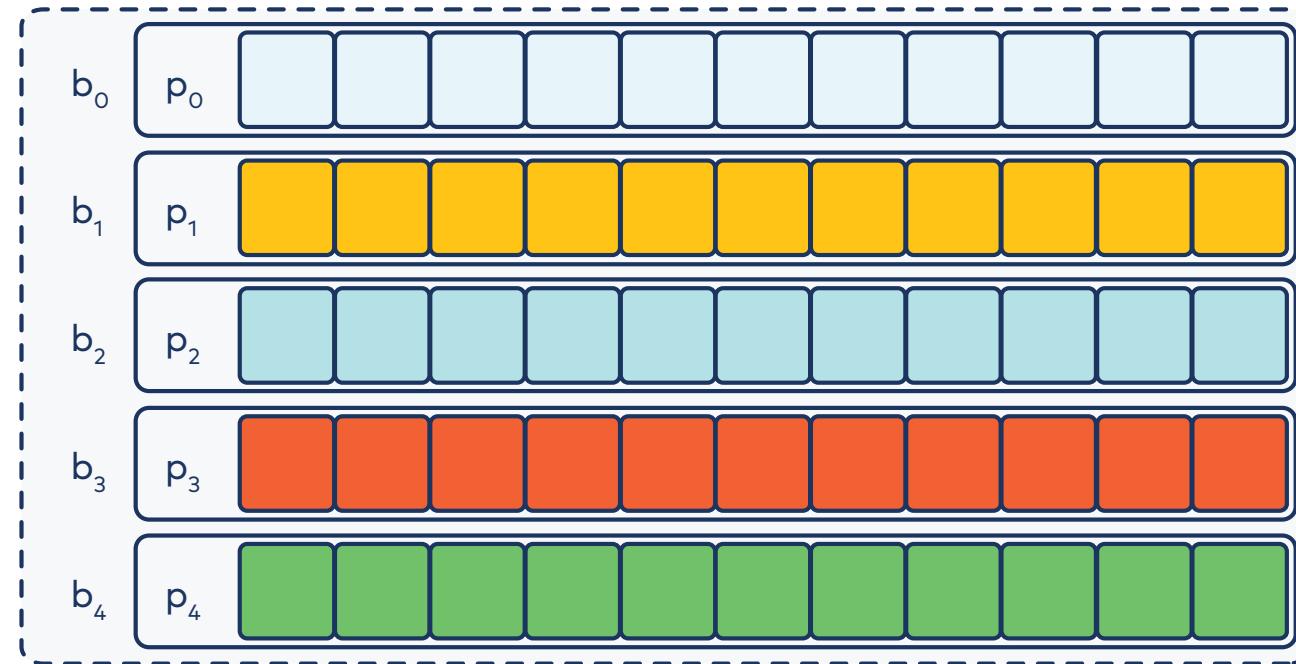
inactive clean inactive clean inactive dirty active
segment 1 segment 2 segment 3 segment
size: 10 **size: 10** **size: 15** **size: 8**

From this picture, we can see that compaction must have happened. Then...

- Suppose it's time for compaction to happen. What is the resulting log?
- Suppose default compaction settings are on. Does log compaction actually happen? If yes, why? If no, what would have to be different to trigger compaction?

Problem F: Partitioning without Keys

We will again work with this scenario of five brokers and five partitions:



Suppose instead of the keyed messages in another problem, we have messages with null keys.

Skim [KIP 480](#) for more on how Kafka handles partitioning by default when messages do not have keys.

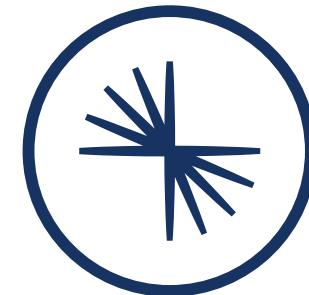
Suppose we have messages that get assigned to partitions in the buffer and these were the batches before being flushed:

batch 0:	<i>a, b, c</i>
batch 1:	<i>d, e</i>
batch 2:	<i>f, g, h, i</i>
batch 3:	<i>j</i>
batch 4:	<i>k, l, m, n</i>
batch 5:	<i>o, p</i>
batch 6:	<i>q, r</i>
batch 7:	<i>s, t, u</i>
batch 8:	<i>v, w, x, y, z</i>

Give an illustration of which messages would land on which partitions...

- ...if you are in a breakout room with an even number: assume we had relatively decent luck, or
- ...if you are in a breakout room with an odd number: assume we had not such great luck

Appendix: Confluent Technical Fundamentals of Apache Kafka® Content



CONFLUENT
Global Education

Module Overview

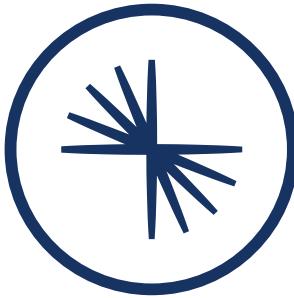


This section contains 5 lessons - the content lessons from the Fundamentals prerequisite:

Lessons of Presentation:

1. Getting Started
2. How are Messages Organized?
3. How Do I Scale and Do More Things With My Data?
4. What's Going On Inside Kafka?
5. Recapping and Going Further

1: Getting Started



CONFLUENT Global Education

Why Kafka?

In a nutshell...

Kafka is good for

- data in motion
- real-time processing

Kafka is not meant for

- batch processing
- archiving data

One Example Use Case: Ordering Food

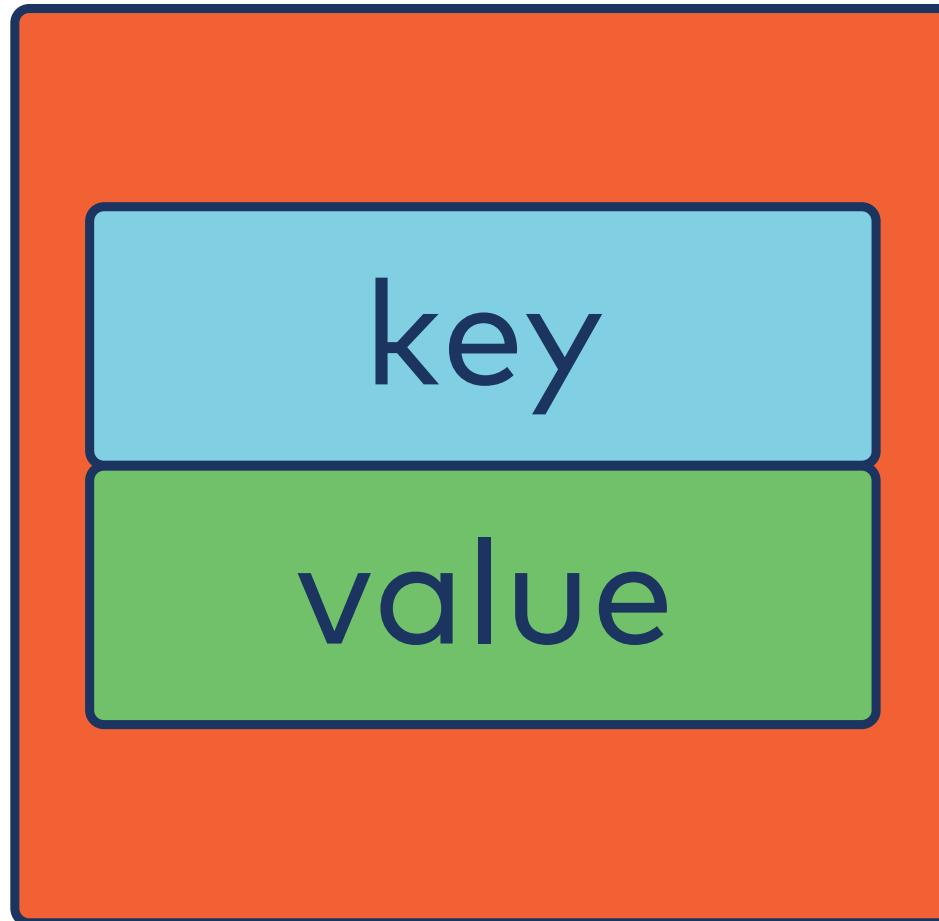
Suppose we are building a system for a restaurant chain:

- customers order food via an app - mobile or kiosk
- staff receive orders to fulfill in real-time
- management tracks inventory based on orders

We will build up some of the fundamental details of Kafka and use this example.

Messages

The atomic unit of Kafka is a **message** or **record** or **event**



Topics

Messages are organized in logical groups called **topics**.

Example topics:

- **orders**
- menu items
- customers
- restaurants

Three Basic Components

Let's start simple - with an **orders** topic in place in Kafka, a producer, and a consumer:



Life Cycle of a Message: Producing

A **producer** prepares messages and publishes them to Kafka.



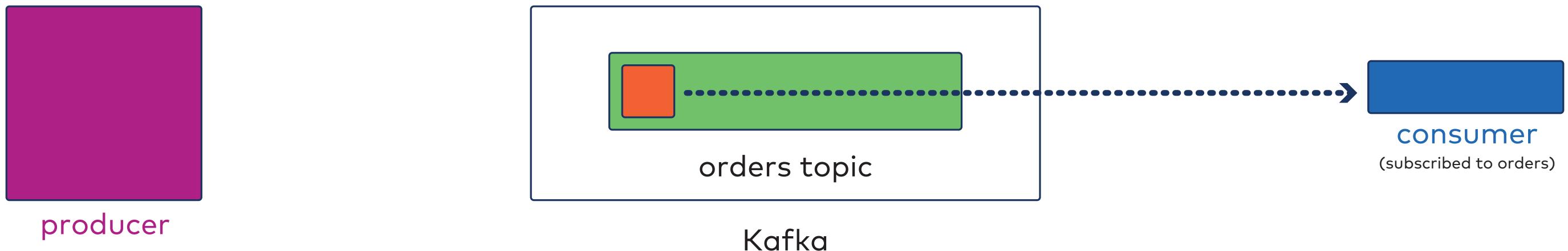
Life Cycle of a Message: Kafka

Produced messages live in Kafka, organized by topic.

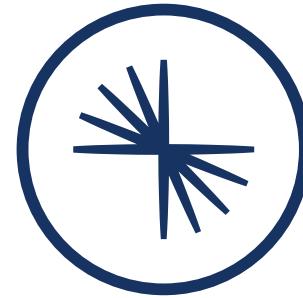


Life Cycle of a Message: Consumption

Consumers subscribe to topics in Kafka and poll for new messages.



2: How are Messages Organized?

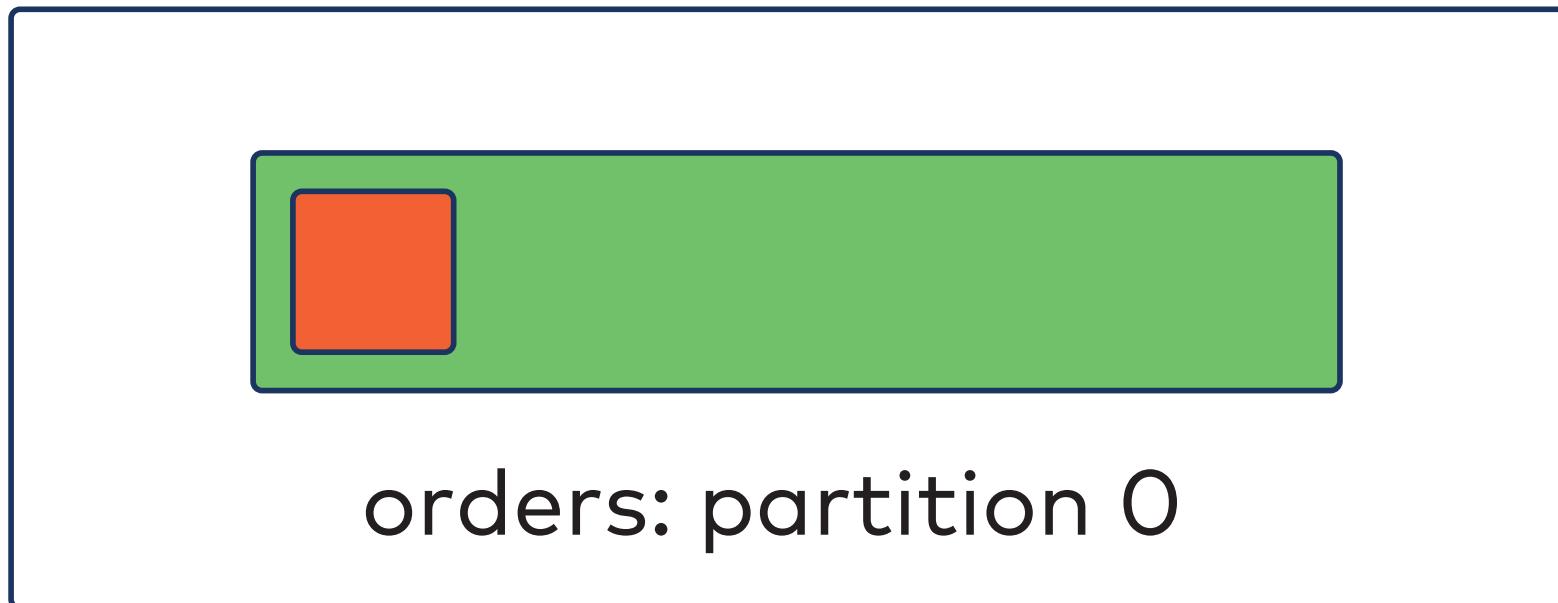


CONFLUENT
Global Education

Topics and Partitions

Topics are broken down into **partitions**

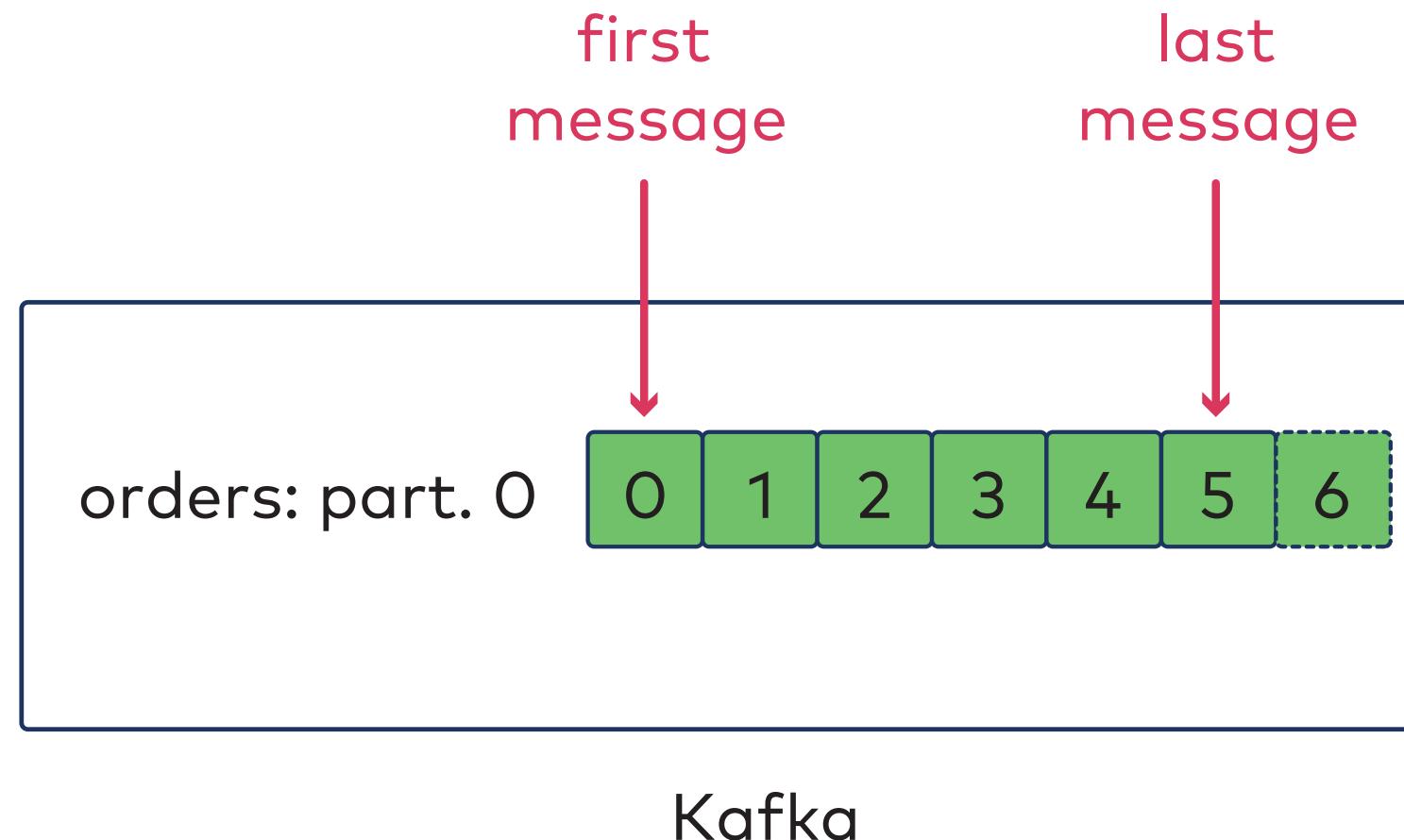
Simplest case: Topic with one partition



Kafka

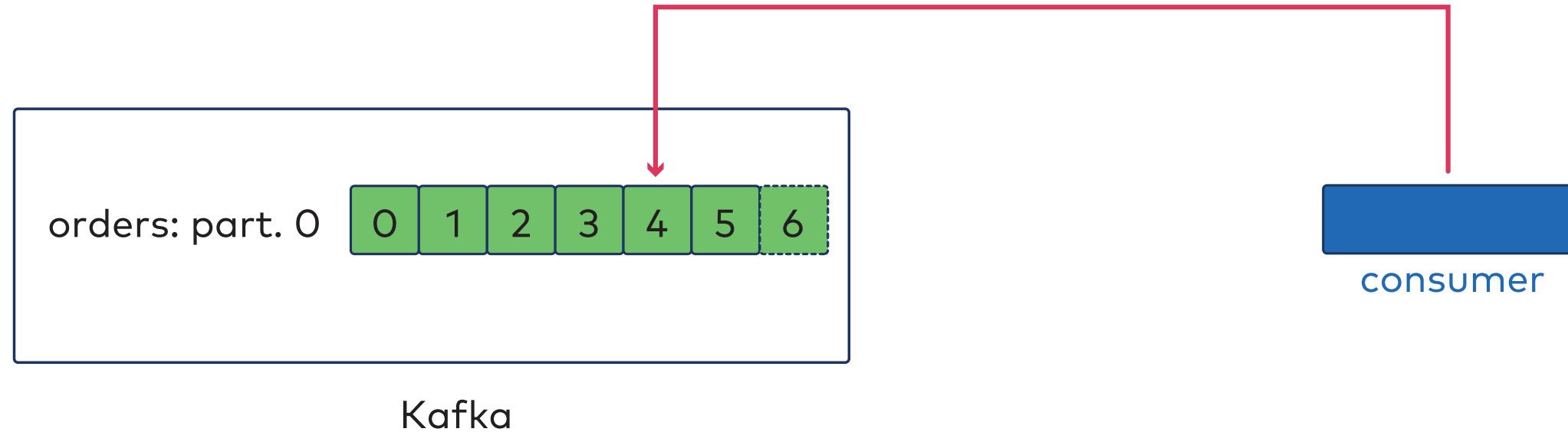
Offsets—in Kafka

- Each message in a partition has an **offset**
- Starting from 0



Offsets—Consumer Offsets

- Consumers track where they will read next via a **consumer offset**



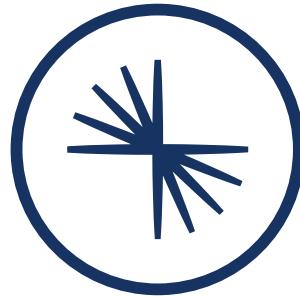
In this picture, the consumer has last read the message at offset 3.

Check Your Knowledge!

Try a [quick quiz on Lessons 1 and 2.](#)



3: How Do I Scale and Do More Things With My Data?



CONFLUENT
Global Education

Scaling Up...

So far, we have seen...

- one partition
- one consumer

In practice...

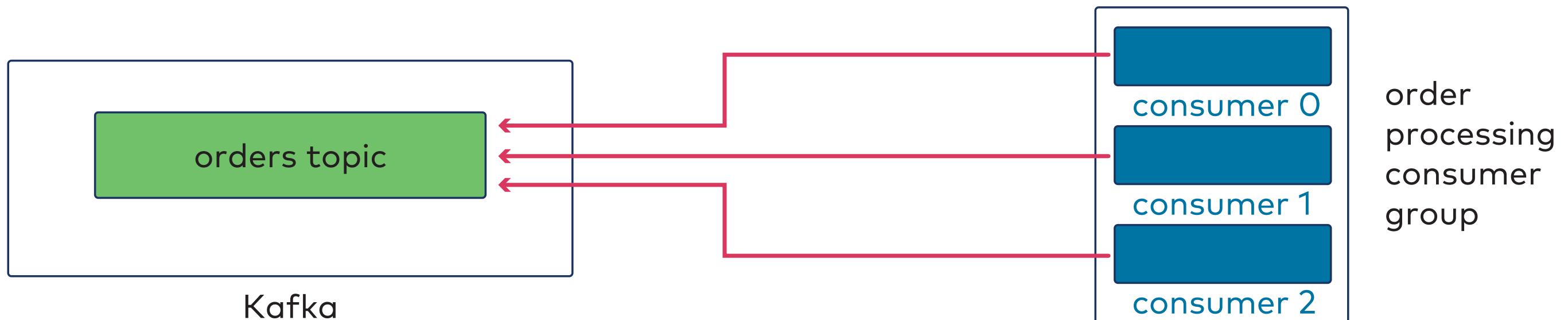
- multiple consumers in a consumer group
- multiple consumer groups
- multiple partitions in a topic

Consumer Groups

Consumers exist in **consumer groups**

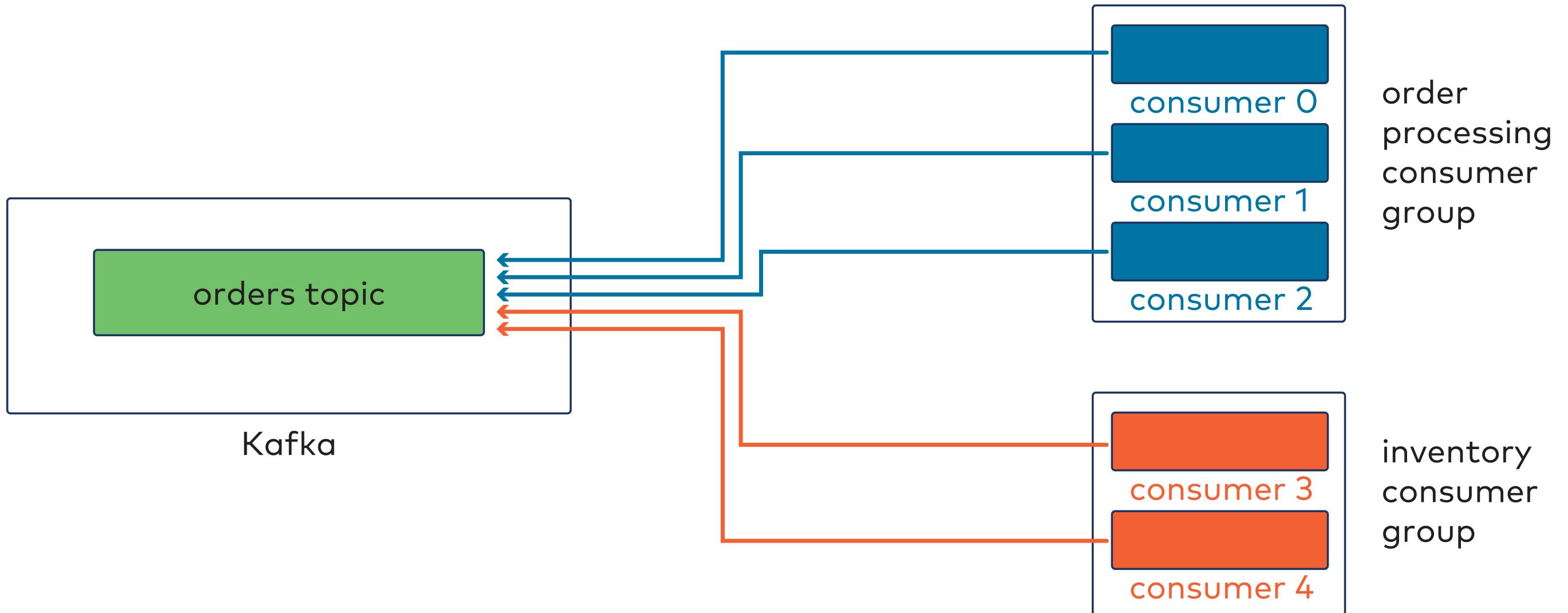
Consumers in a group:

- **same application**
- **different data**



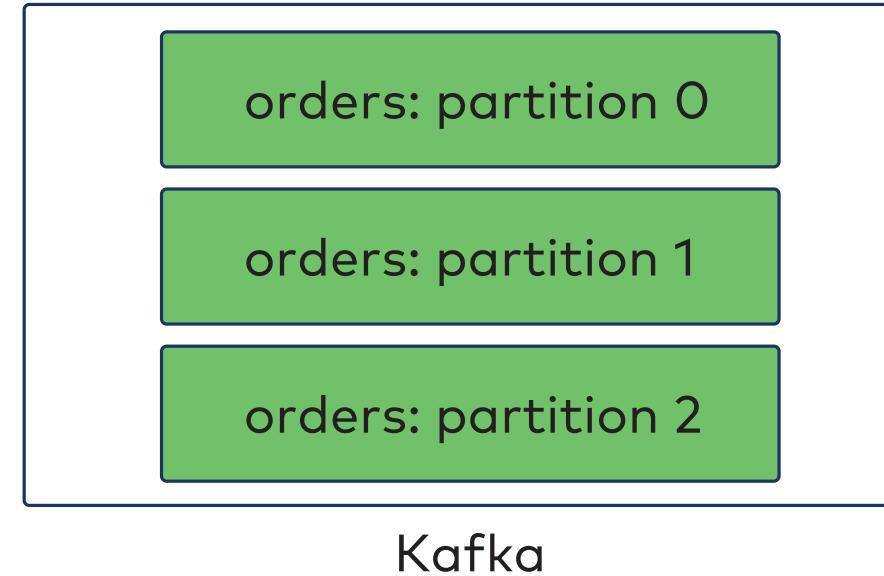
Multiple Consumption

Could have multiple groups using the same data...



Multiple Partitions

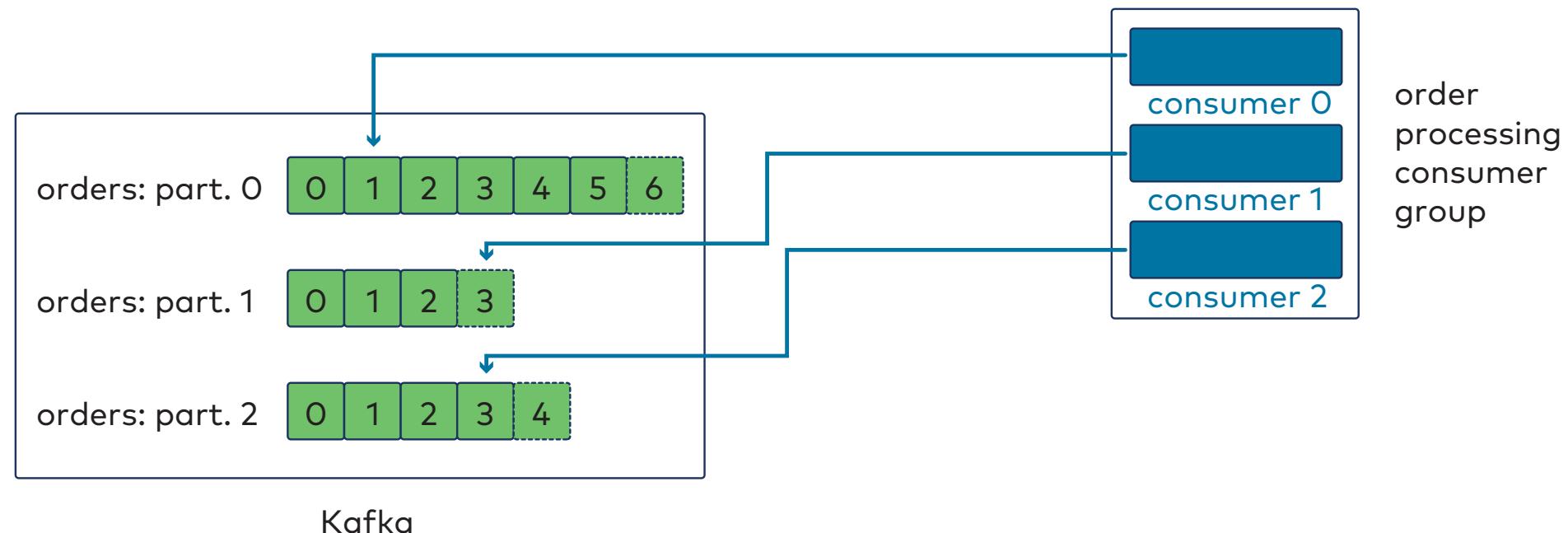
In practice, we want topics to have multiple partitions.



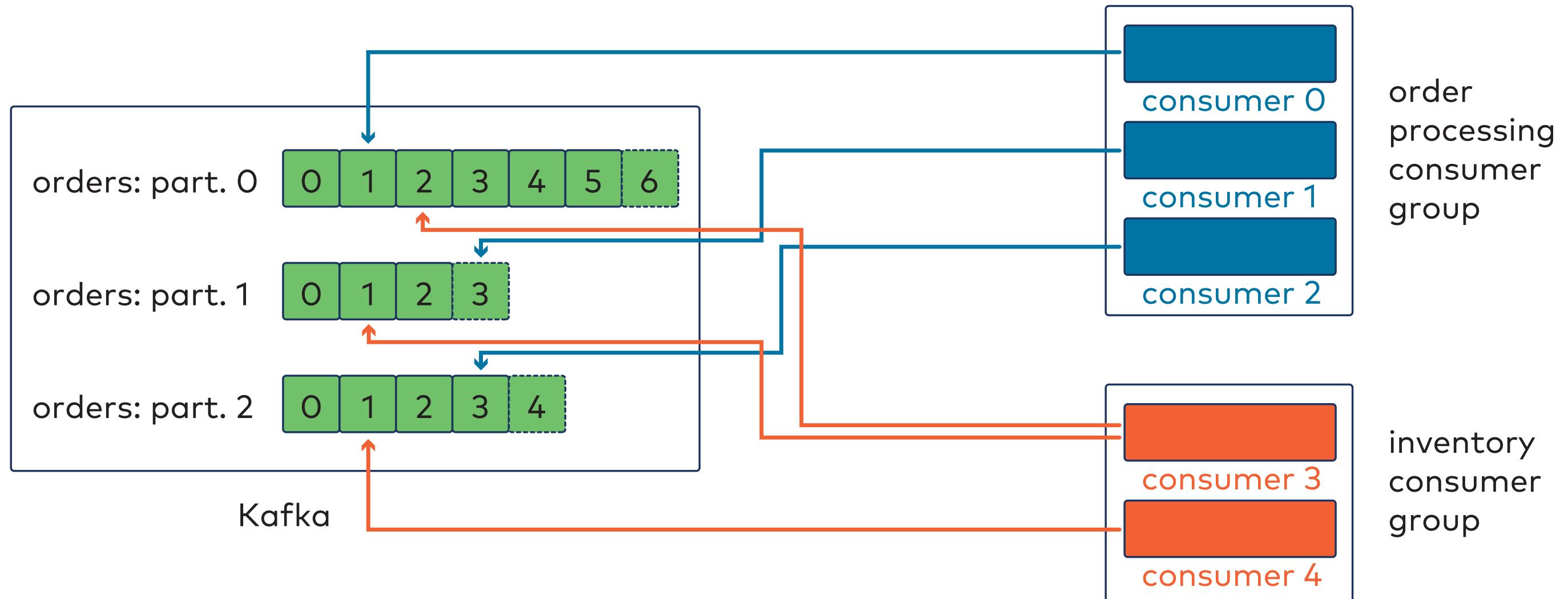
Consuming from Multiple Partitions

Now our consumers can consume in parallel:

- Consumers subscribed to a topic are assigned partitions
- Group covers all partitions
- Each consumer has an offset for each partition



Expanding the Last Picture



How Do Messages Get Partitioned?

- **Producers** decide which messages go to which partition
- Partitions are indexed from 0 to `numberOfPartitions - 1`
- Default partitioner: `partitionIndex = hash(key) % numberOfPartitions`

Scaling is Easy!

Say you want to

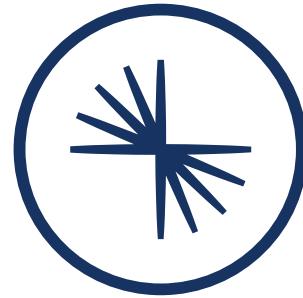
- Increase the number of consumers in a group during a busy season
- Decrease the number of consumers in a group when things are slow
- Increase the number of partitions for a topic

When you do, Kafka *automatically* redistributes the assignments of consumers to partitions!



More on how all of this works in both our Developer and Administrator training!

4: What's Going On Inside Kafka?



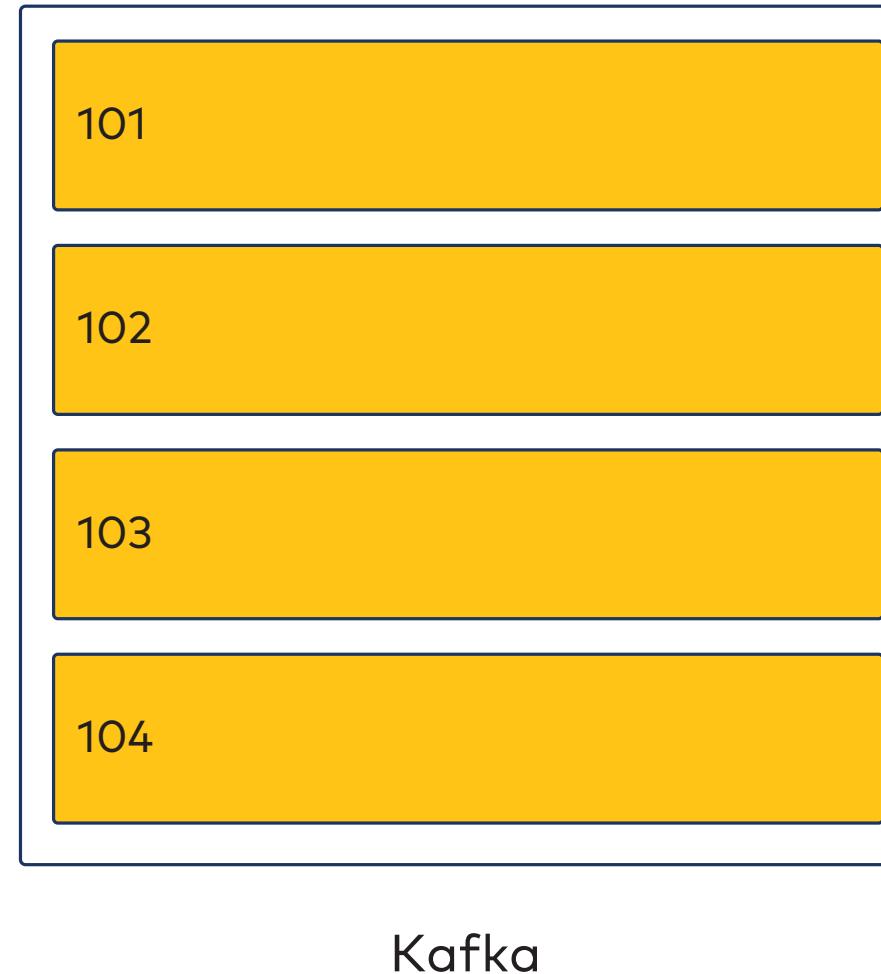
CONFLUENT
Global Education

Going Deeper...

Now let's learn about some more details about a Kafka cluster, especially *physical* things...

Brokers

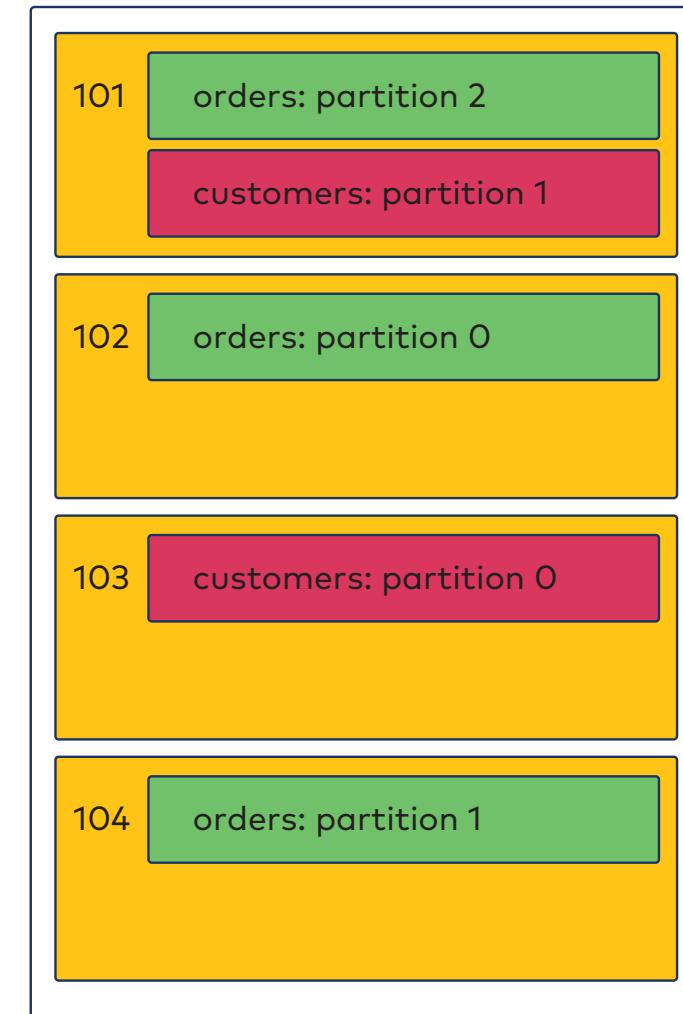
A Kafka cluster consists of multiple **brokers**



Partitions & Brokers (2)

The number of partitions is a topic setting.

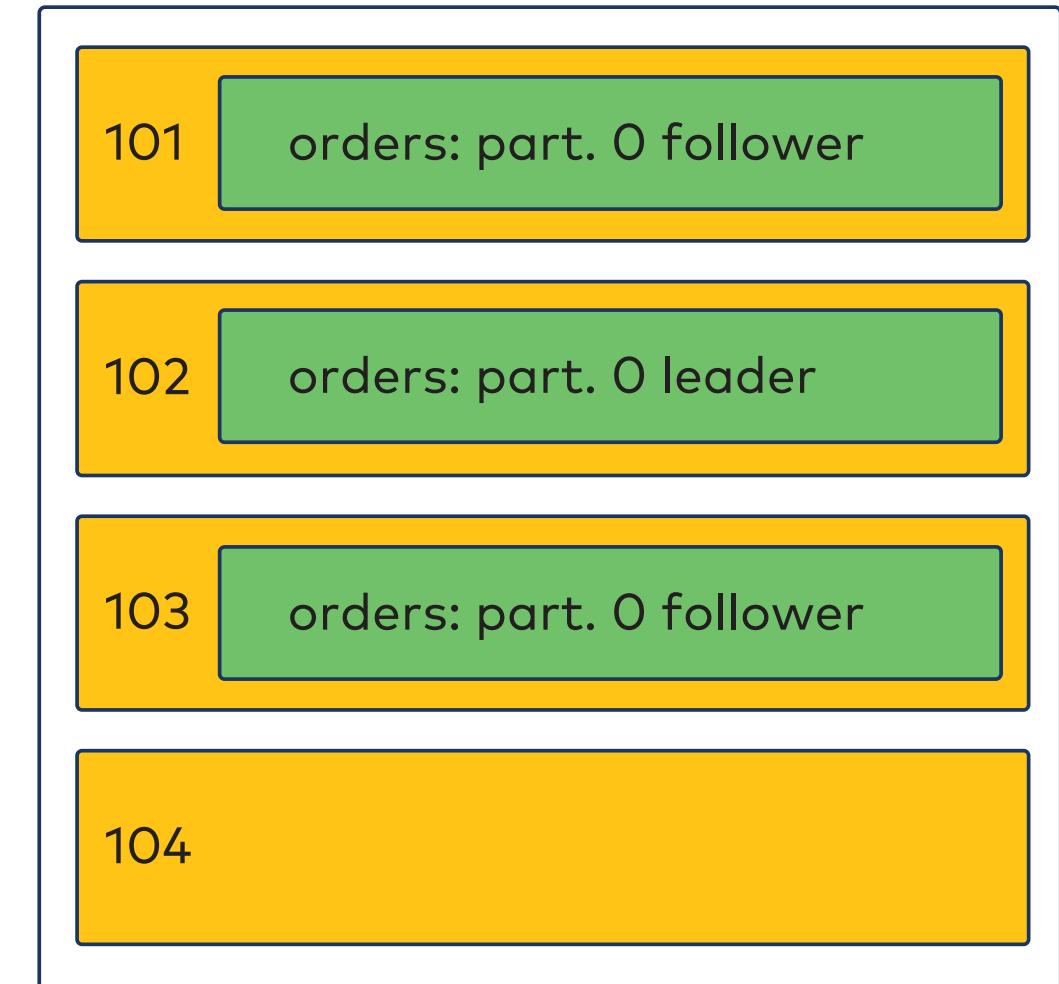
Kafka decides how partitions get distributed across brokers.



Kafka

What if a Broker Goes Down?

- Want *high availability* of data in partitions
- Achieved via **replication**
- Writes are reads go to **leader** replica
- **Follower** replicas keep backup copies of the leader
- If leader dies, a follower becomes the leader



Kafka

Serialization and Deserialization

- Kafka stores messages as byte arrays
- Producers must **serialize** messages
- Consumers must **deserialize** messages

Immutable Messages

- Messages are **immutable**
- Once written, we cannot change anything about them

...But We Don't Keep Messages Forever...

Control which messages stay in Kafka via a **retention policy**:

Policy	Deletion	Compaction																																										
Idea	Remove messages older than a certain age (default 7 days)	Keep only the latest value for each key																																										
Before	<table><thead><tr><th></th><th>offset</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th></tr><tr><th></th><th>key</th><td>a</td><td>b</td><td>b</td><td>a</td><td>a</td></tr><tr><th></th><th>age in days</th><td>12</td><td>10</td><td>6</td><td>5</td><td>2</td></tr></thead></table>		offset	0	1	2	3	4		key	a	b	b	a	a		age in days	12	10	6	5	2	<table><thead><tr><th></th><th>offset</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th></tr><tr><th></th><th>key</th><td>a</td><td>b</td><td>b</td><td>a</td><td>a</td></tr><tr><th></th><th>age in days</th><td>12</td><td>10</td><td>6</td><td>5</td><td>2</td></tr></thead></table>		offset	0	1	2	3	4		key	a	b	b	a	a		age in days	12	10	6	5	2
	offset	0	1	2	3	4																																						
	key	a	b	b	a	a																																						
	age in days	12	10	6	5	2																																						
	offset	0	1	2	3	4																																						
	key	a	b	b	a	a																																						
	age in days	12	10	6	5	2																																						
After	<table><thead><tr><th></th><th>offset</th><th>2</th><th>3</th><th>4</th></tr><tr><th></th><th>age in days</th><td>6</td><td>5</td><td>2</td></tr></thead></table>		offset	2	3	4		age in days	6	5	2	<table><thead><tr><th></th><th>offset</th><th>2</th><th>4</th></tr><tr><th></th><th>key</th><td>b</td><td>a</td></tr></thead></table>		offset	2	4		key	b	a																								
	offset	2	3	4																																								
	age in days	6	5	2																																								
	offset	2	4																																									
	key	b	a																																									



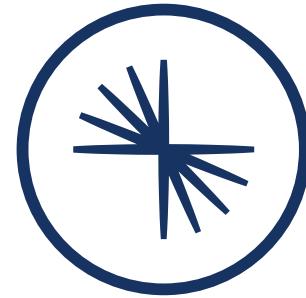
Partitions are divided into segments, which affect both retention policies.

Check Your Knowledge!

Try a [quick quiz on Lessons 3 and 4.](#)

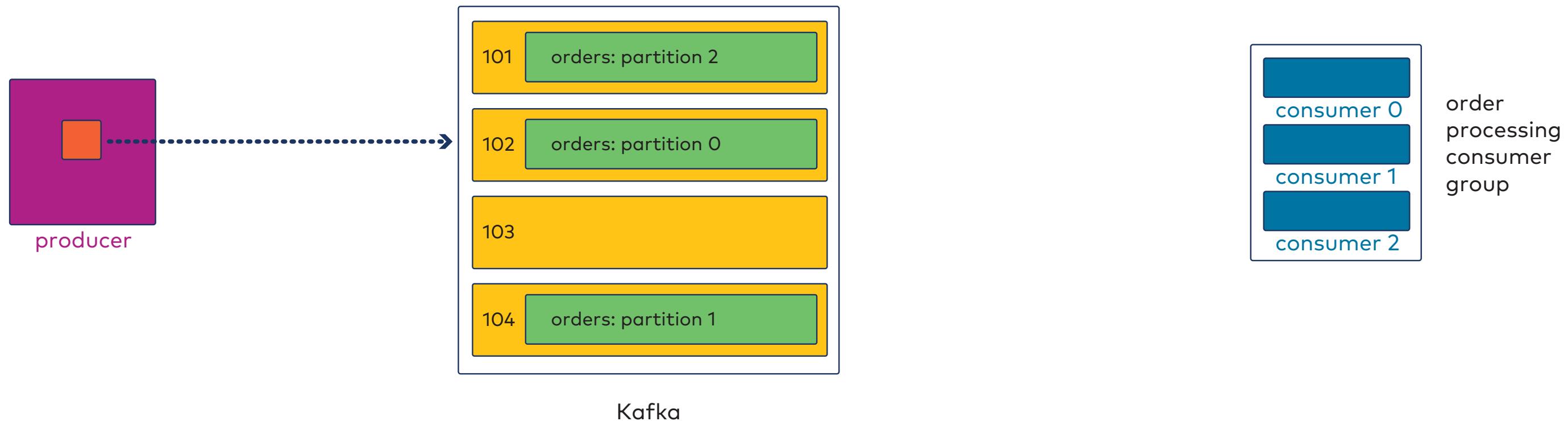


5: Recapping and Going Further



CONFLUENT
Global Education

Life Cycle of a Message: Producing



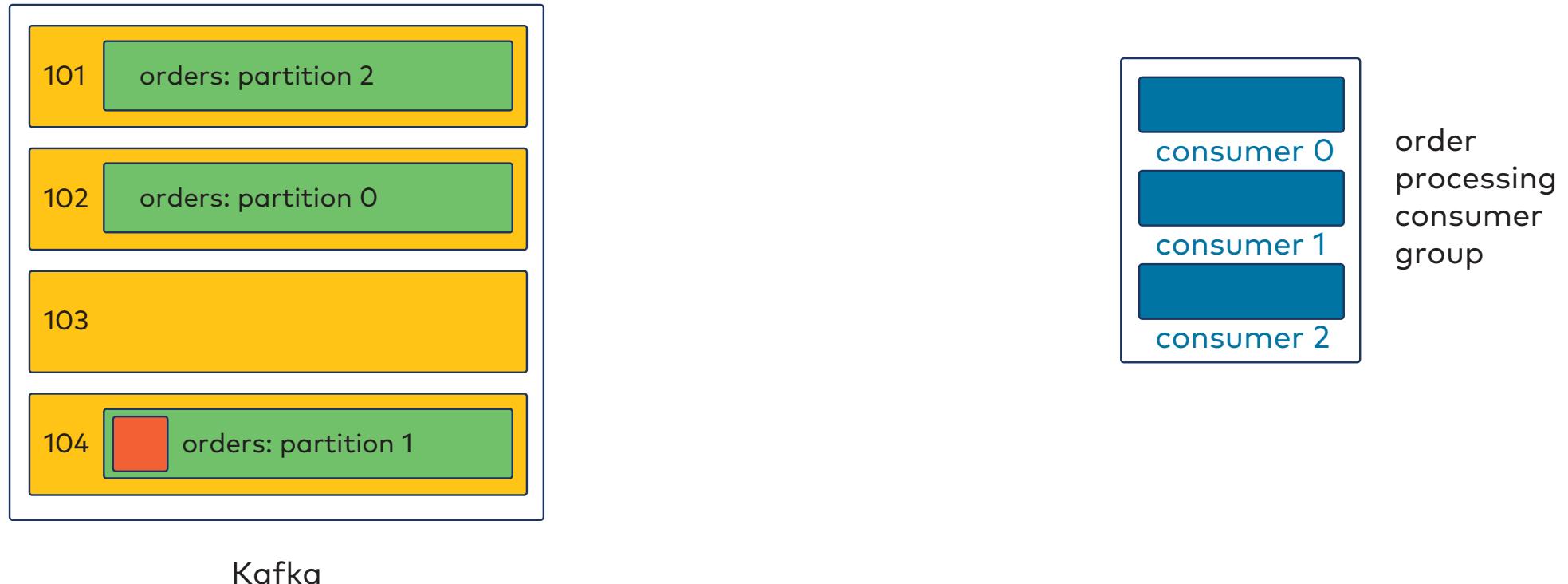
- Producers serialize and partition messages
- Producers send messages
 - ...in batches - can be configured for throughput and latency desires

Life Cycle of a Message: Kafka

Produced messages live in Kafka, organized by topic.



producer



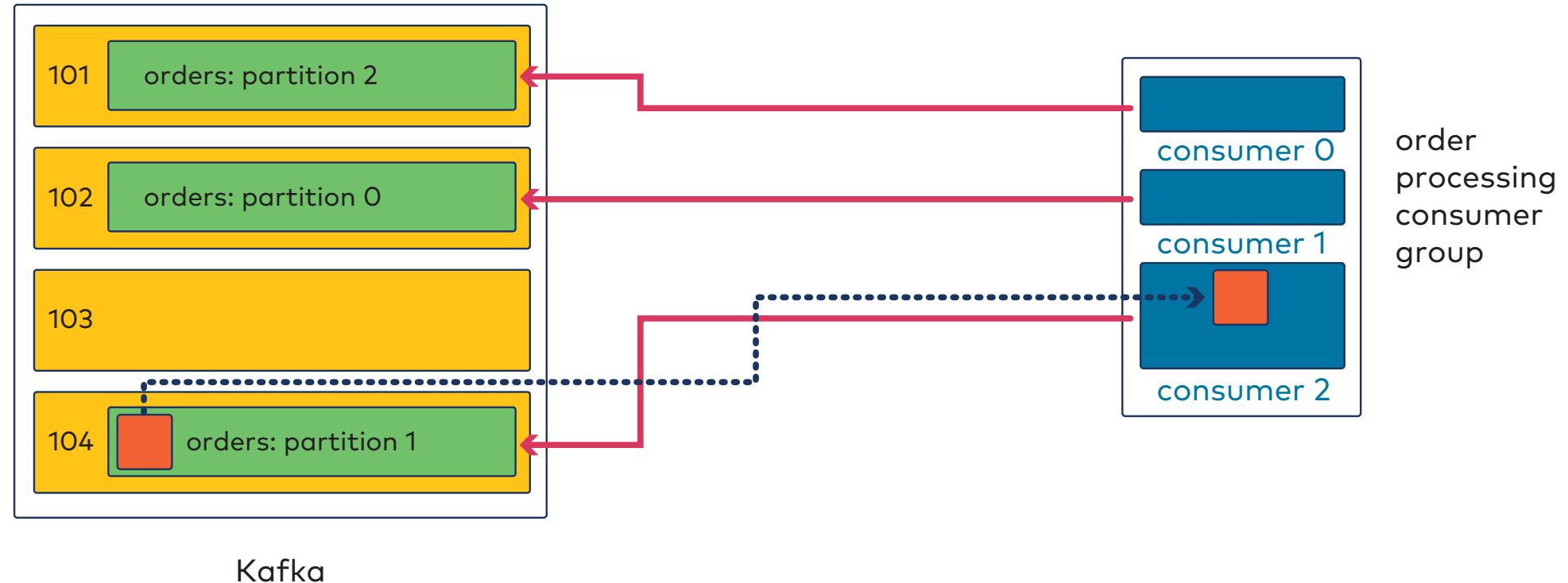
- Kafka consists of brokers
- Brokers contain partitions, which contain messages
- Brokers handle retention and replication

Life Cycle of a Message: Consumption

Consumers subscribe to topics in Kafka and poll for new messages.



producer



- Consumers operate in groups
- Consumers subscribe to topics, are assigned partitions of those topics
- Consumers poll for messages in partitions at consumer offsets
 - ...and fetch in batches - can be configured for throughput and latency desires

A Step Beyond Fundamentals: Other Components

We've addressed some aspects of Core Kafka in this course. Some other topics you may want to learn about include:

- **Kafka Connect** - a tool that helps you copy data to Kafka from other systems and vice-versa
- **Kafka Streams** - a layer on top of the Producer and Consumer APIs that allows for stream processing
- **Confluent ksqlDB** - a tool for stream processing using a more-accessible SQL-like syntax, among other things
- **Confluent Schema Registry** - a tool for managing schemas, guiding schema evolution, and enforcing data integrity

You can learn more about these topics in our Confluent Developer Skills for Building Apache Kafka® and Apache Kafka® Administration by Confluent courses.

What Does Confluent Platform Add to Kafka?

CONFLUENT PLATFORM

SECURITY & RESILIENCY

RBAC | Audit Logs | Schema Validation | Multi-Region Clusters | Replicator | Cluster Linking

PERFORMANCE & SCALABILITY

Tiered Storage | Self-Balancing Clusters | K8s Operator

MANAGEMENT & MONITORING

Control Center | Proactive Support

DEVELOPMENT & CONNECTIVITY

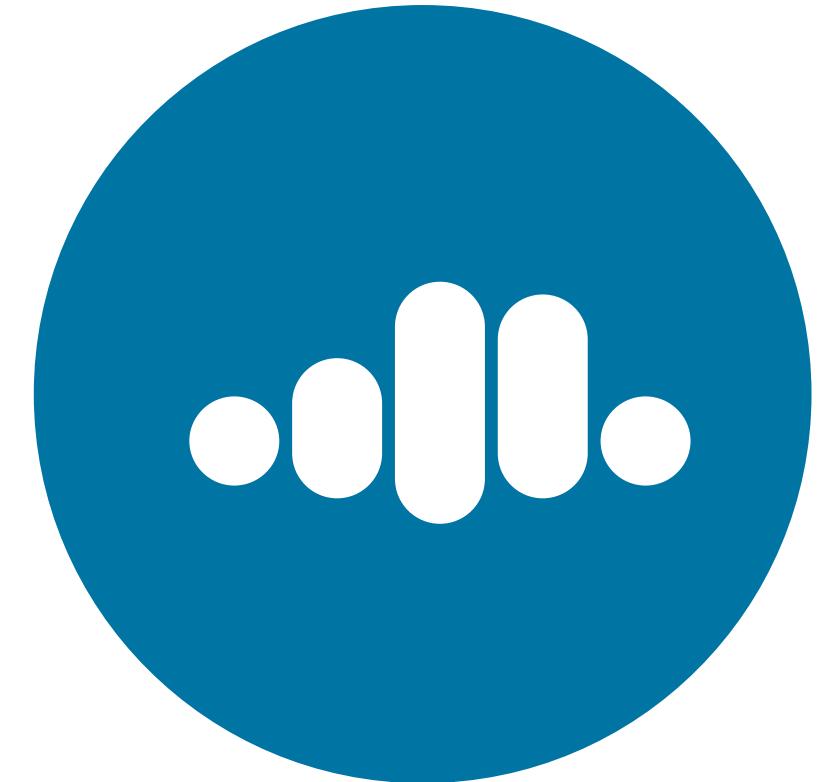
Connectors | Non-Java Clients | REST Proxy | Schema Registry | ksqlDB

APACHE KAFKA®

Core | Connect API | Streams API

Confluent Cloud

- Can deploy CP as self-managed software but...
- Confluent Cloud = **fully-managed** deployment of CP
 - Many administrative tasks done for you
- Confluent Cloud available on
 - AWS
 - Google Cloud Platform
 - Microsoft Azure



Your Next Steps

1. Complete interactive lab on seeing console producers and consumers in action.
 - [Short Confluent Cloud version](#)
 - Gitpod version: [More involved version using Gitpod](#)
2. Work though other Critical Thinking Challenge Exercises.
 - [On the web](#)
 - Solutions on the web too!
3. Enroll in and complete one of these courses, as suits your role:
 - Apache Kafka® Administration by Confluent
 - Confluent Developer Skills for Building Apache Kafka®

Labs:



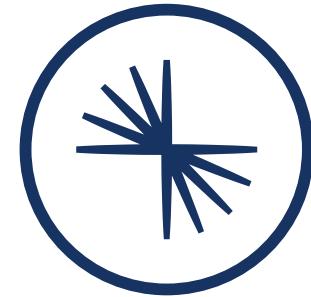
Critical
Thinking:



Thank You

Thank you for attending the course!

Appendix: Additional Content



CONFLUENT
Global Education

Overview

This appendix contains a few additional lessons. These lessons are for additional information for you, but are not designed the same as the rest; namely, they do not have activities or labs to reinforce the content like the rest.

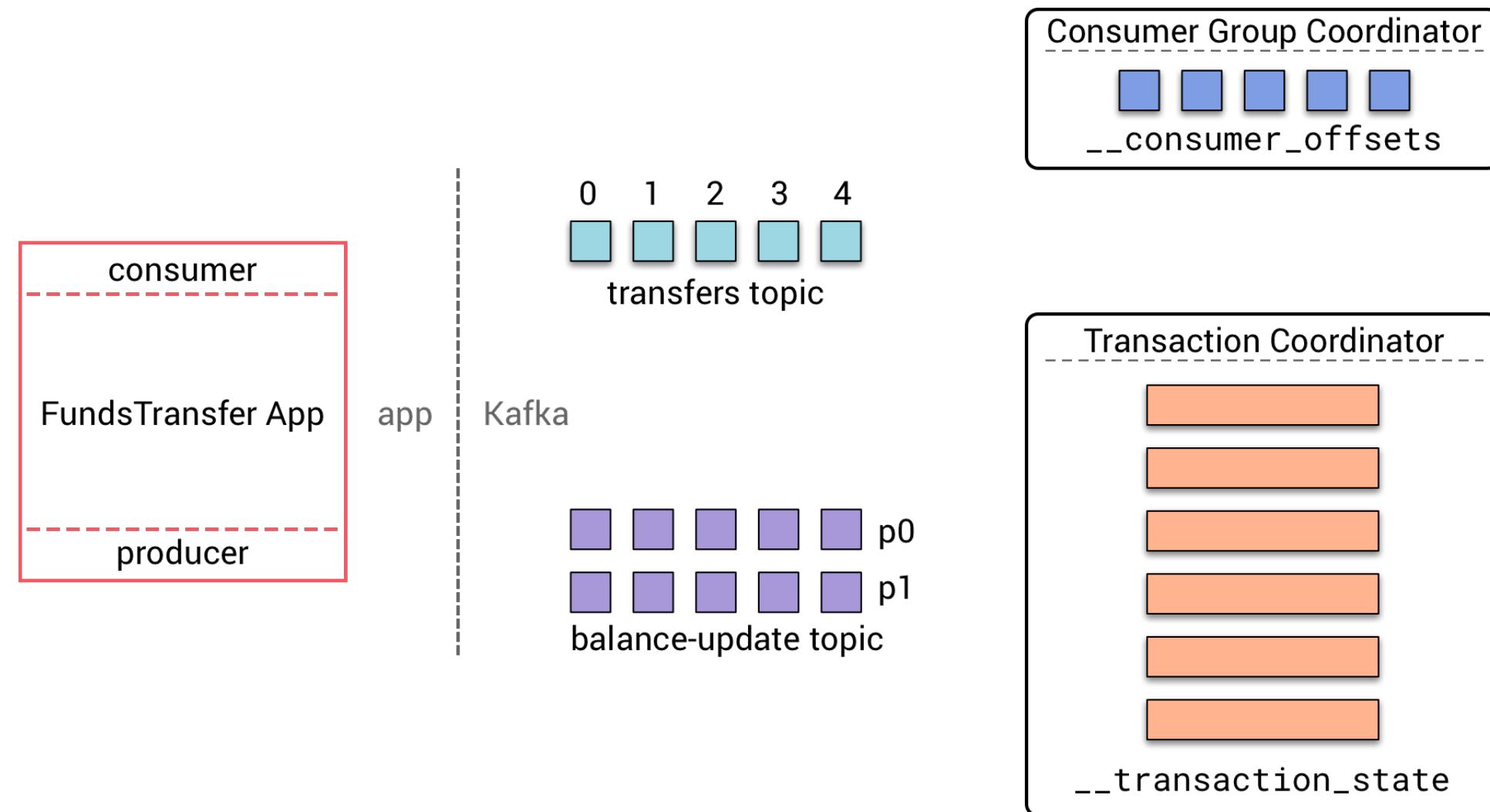
Some lessons that were part of modules of a previous version of this course. Some are lessons taken from another course.

Appendix A: Detailed Transactions Demo

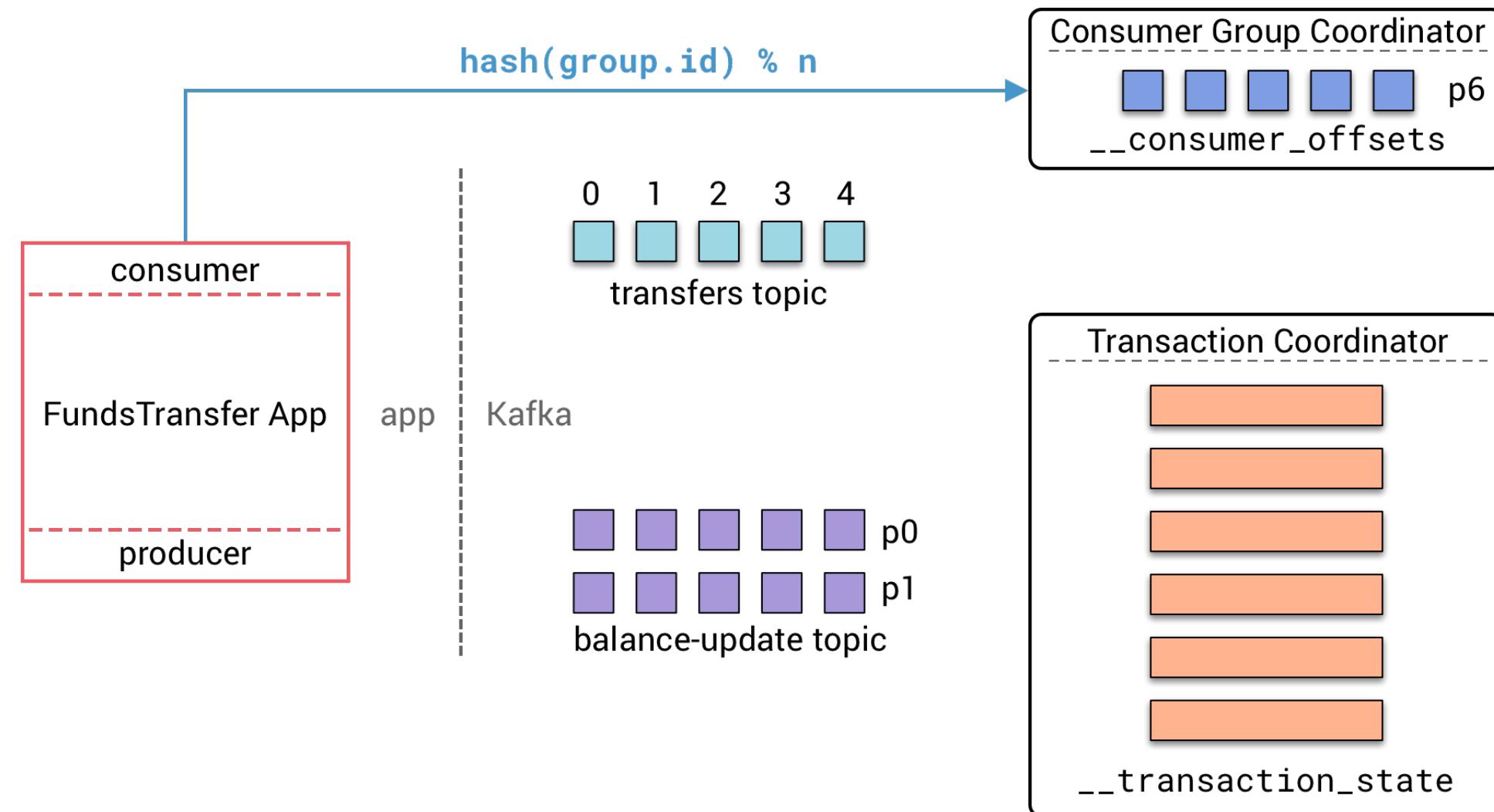
Description

This section presents a more detailed demo of a consume-process-produce application that uses transactions.

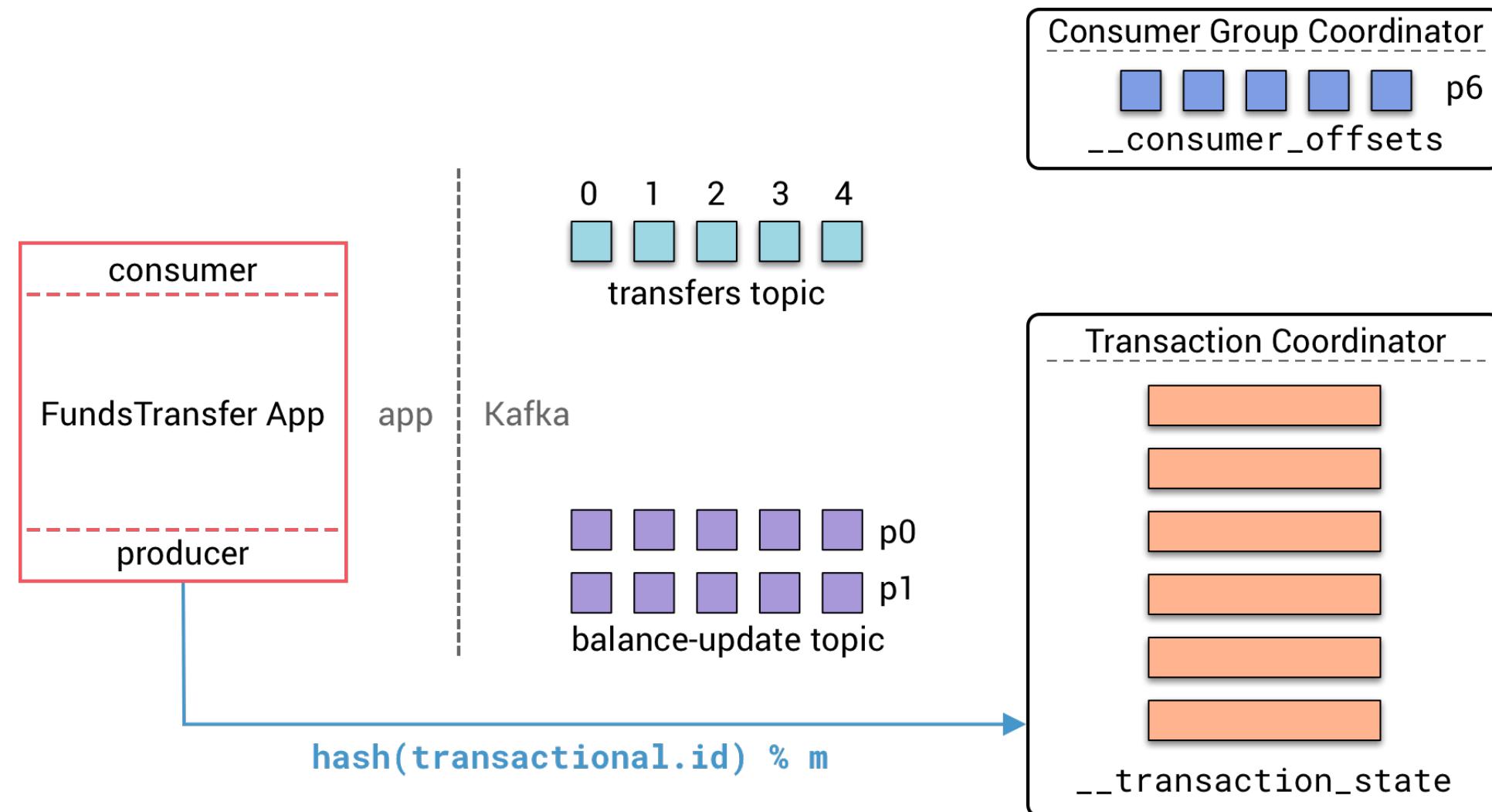
Transactions (1/14)



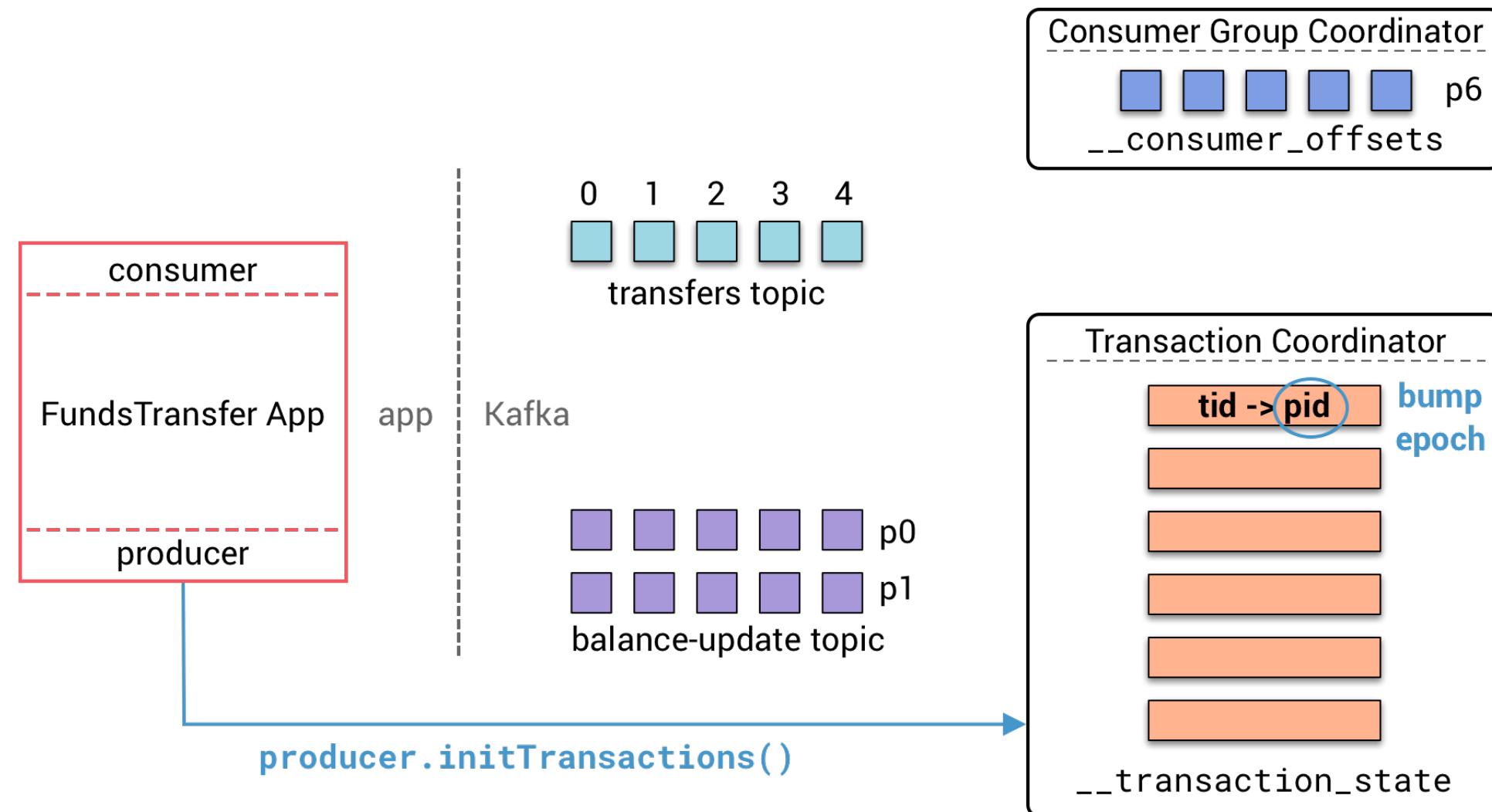
Transactions - Initialize Consumer Group (2/14)



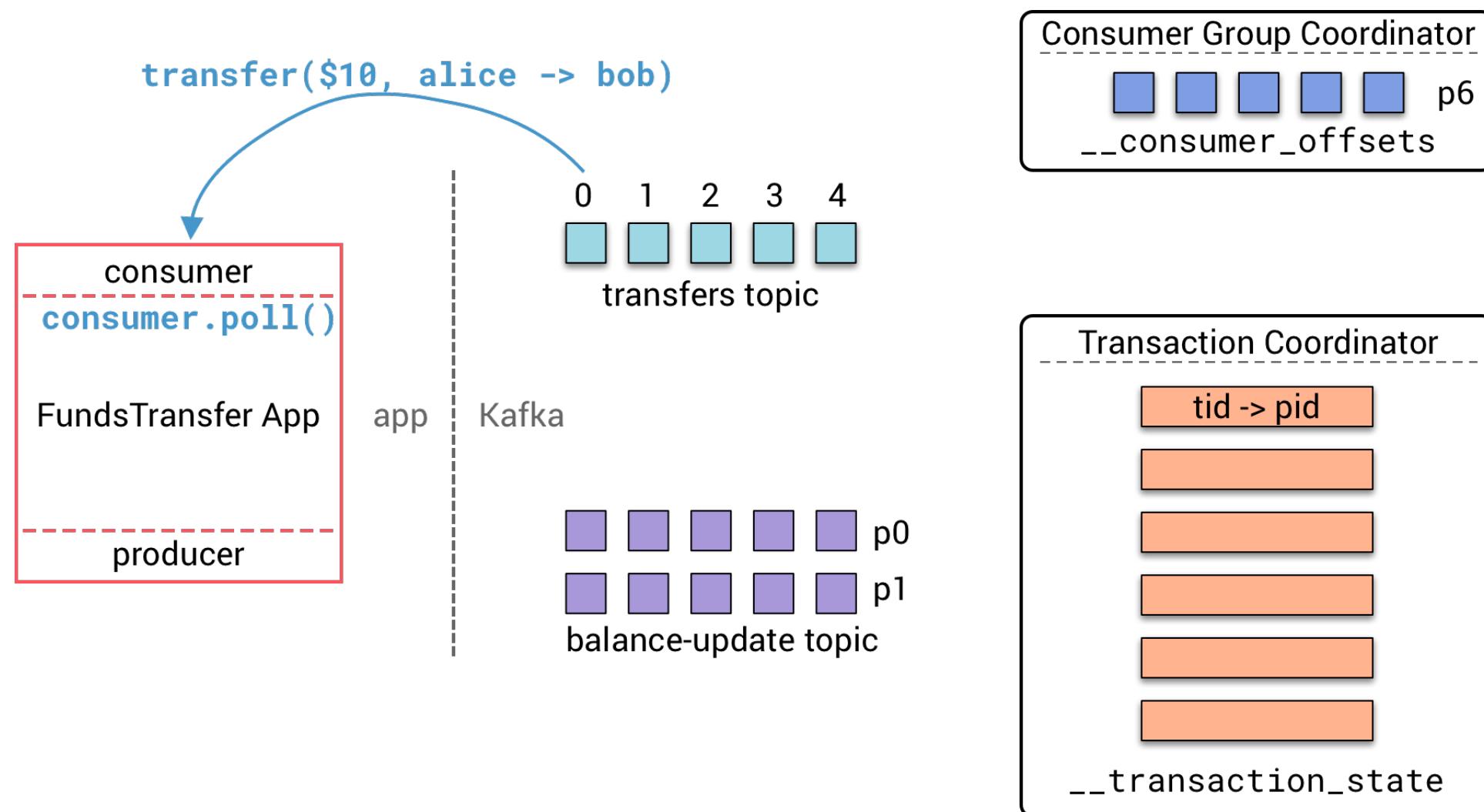
Transactions - Transaction Coordinator (3/14)



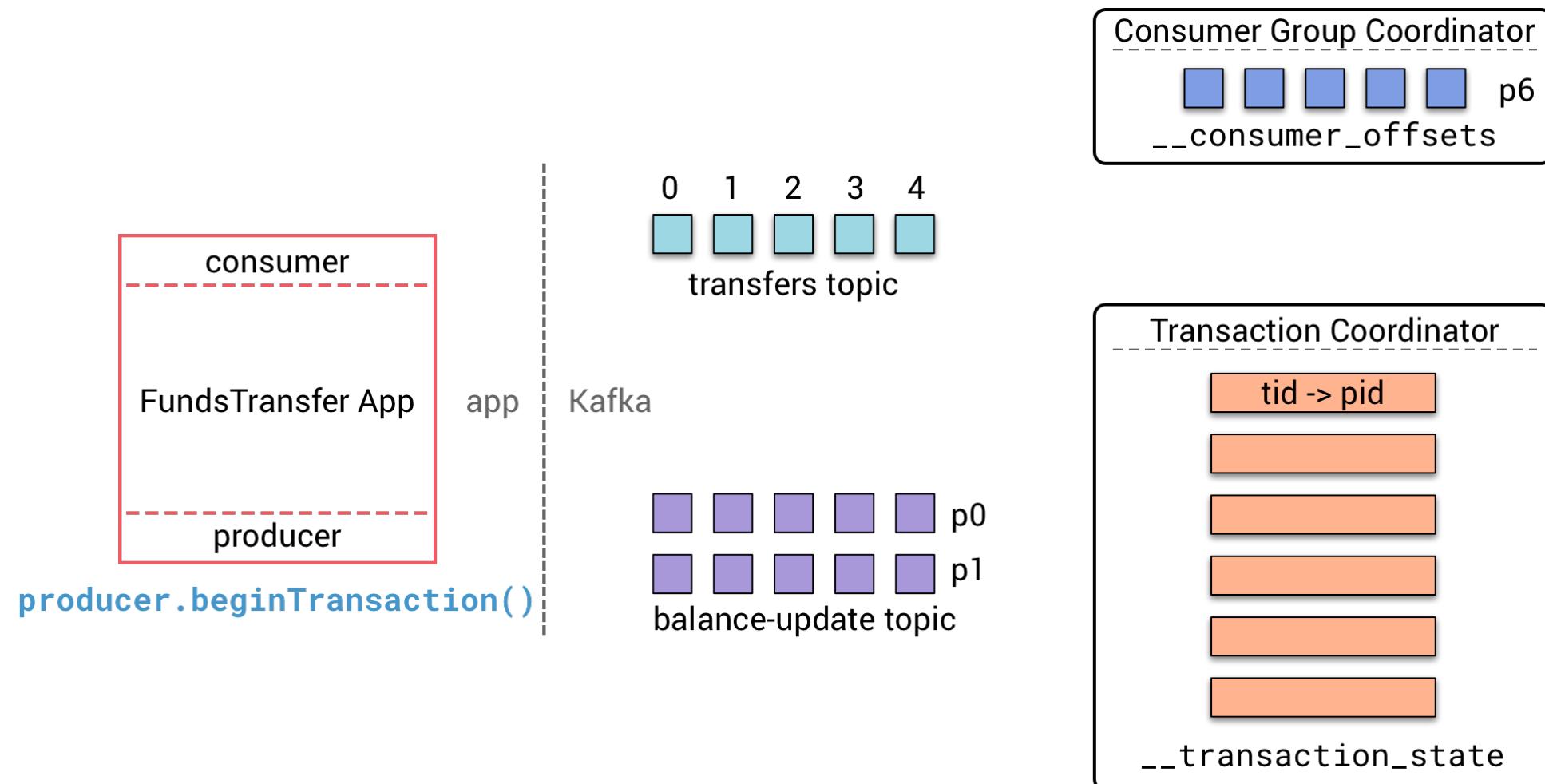
Transactions - Initialize (4/14)



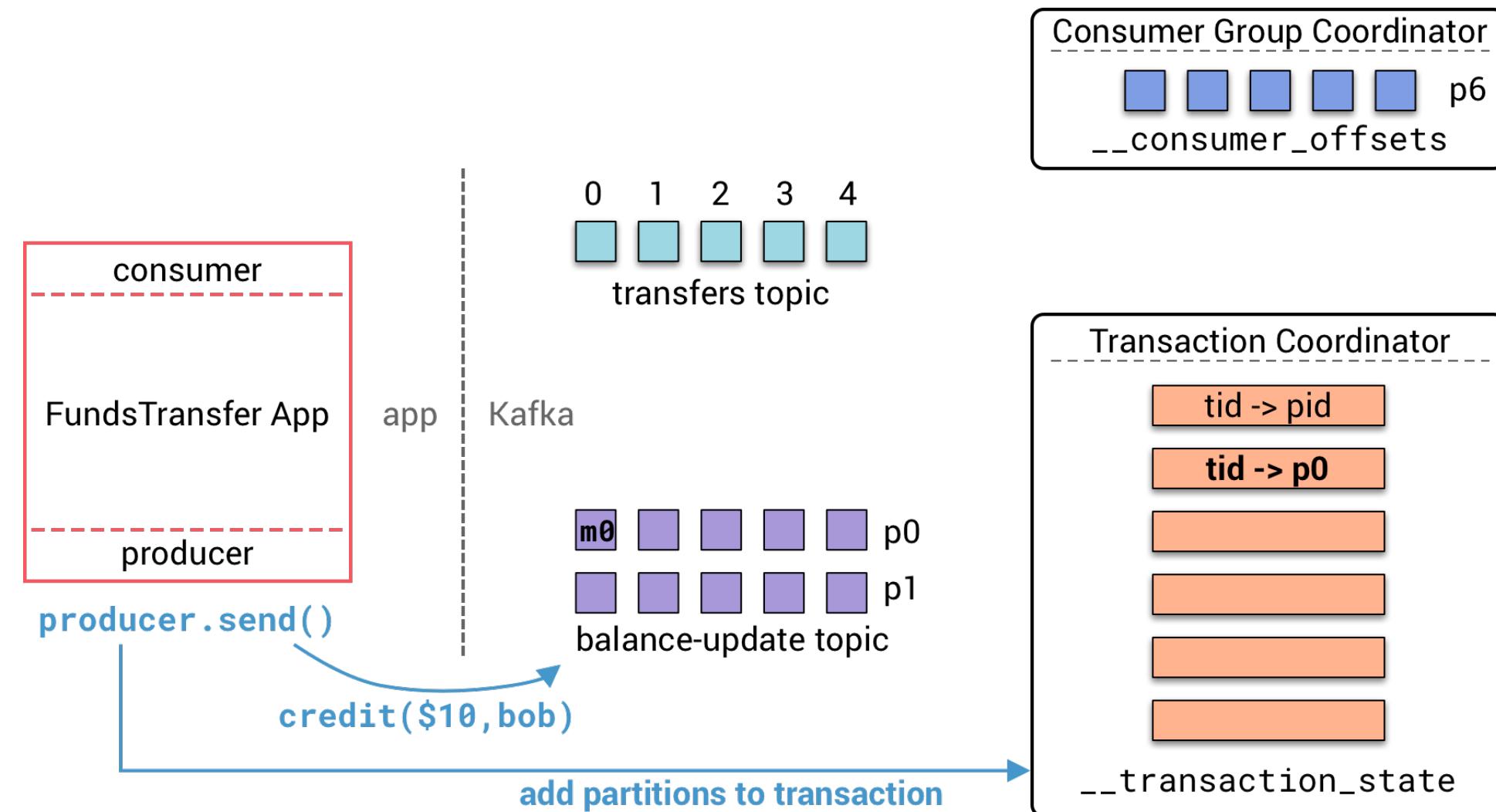
Transactions - Consume and Process (5/14)



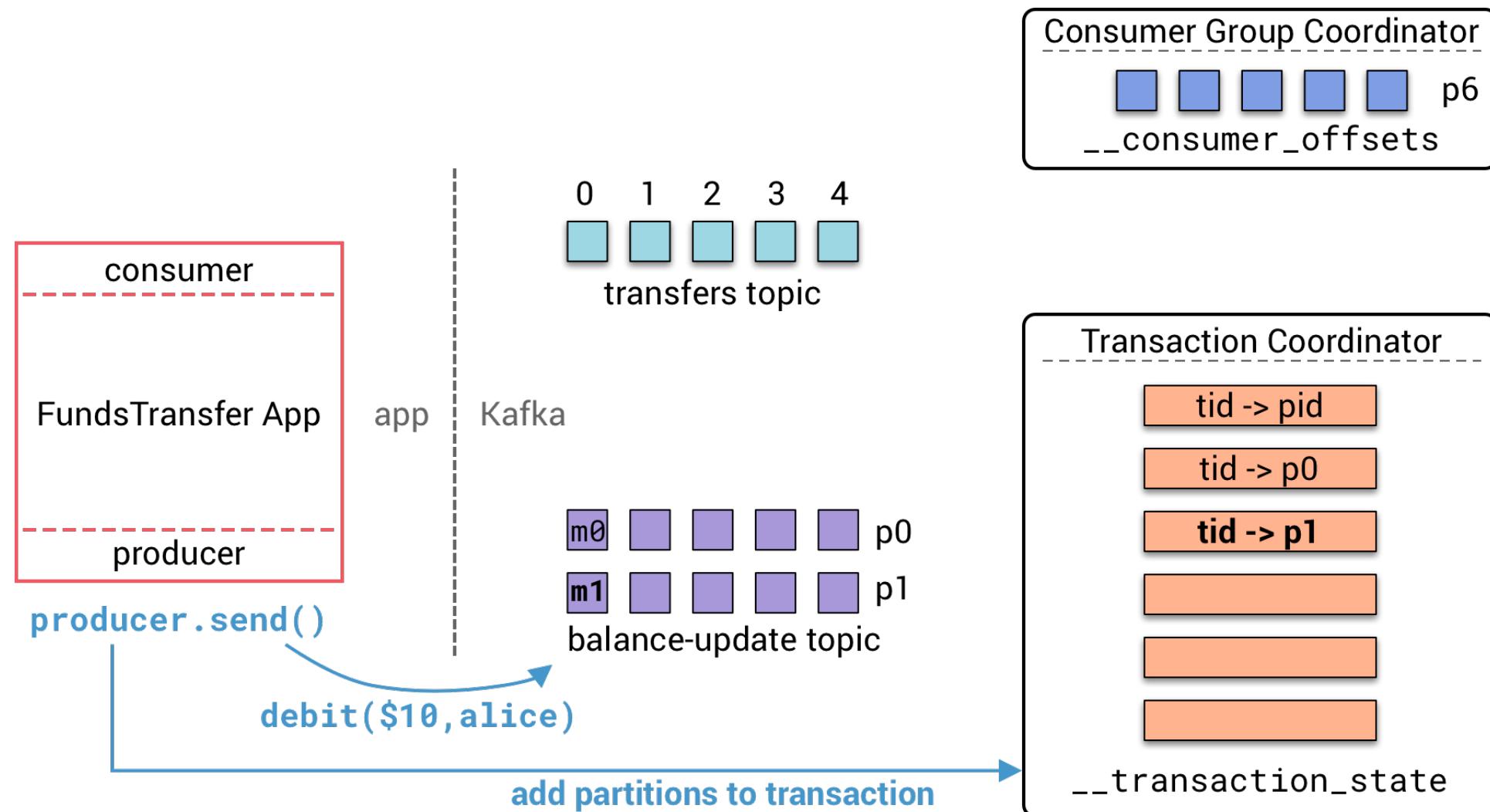
Transactions - Begin Transaction (6/14)



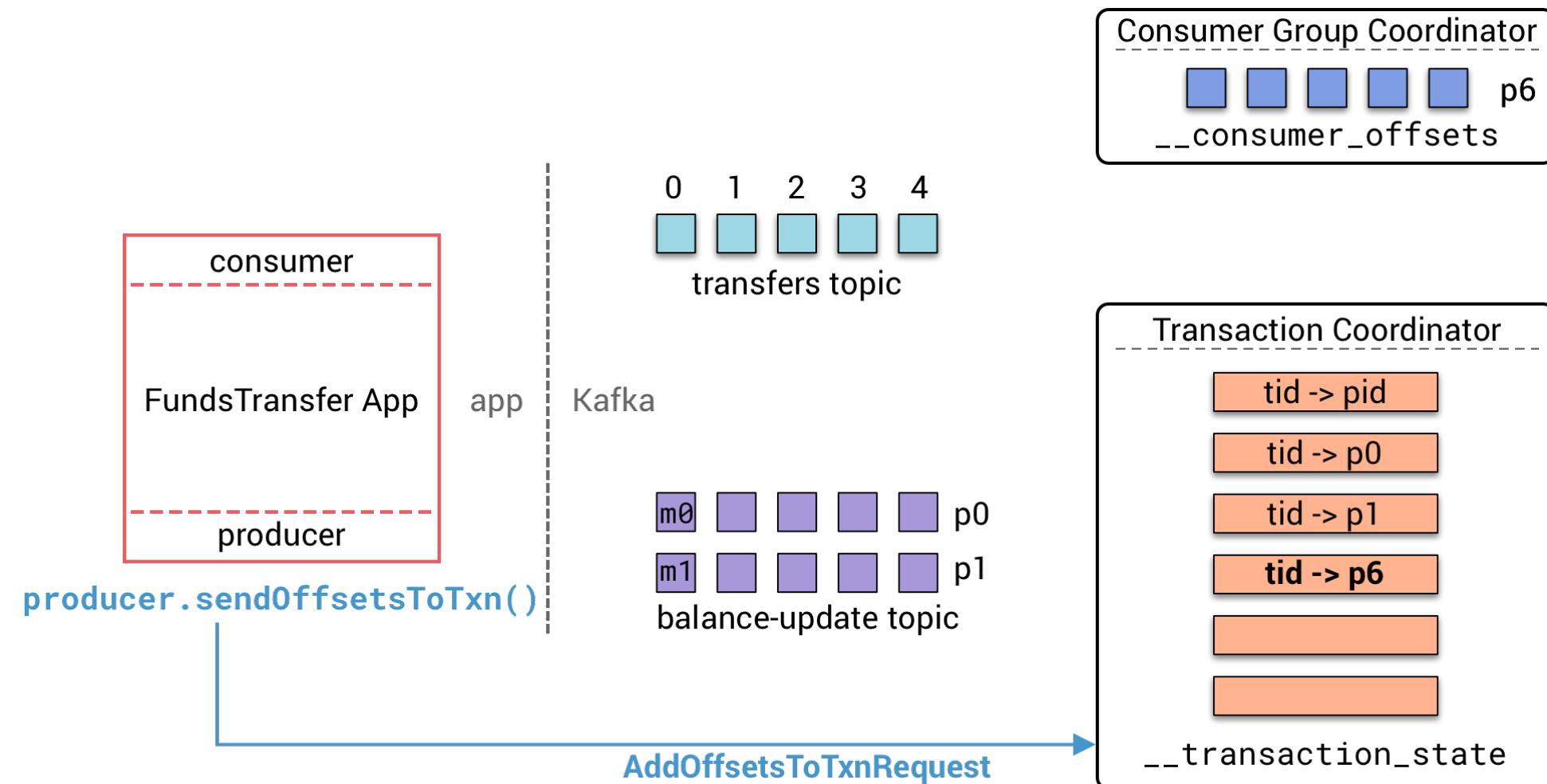
Transactions - Send (7/14)



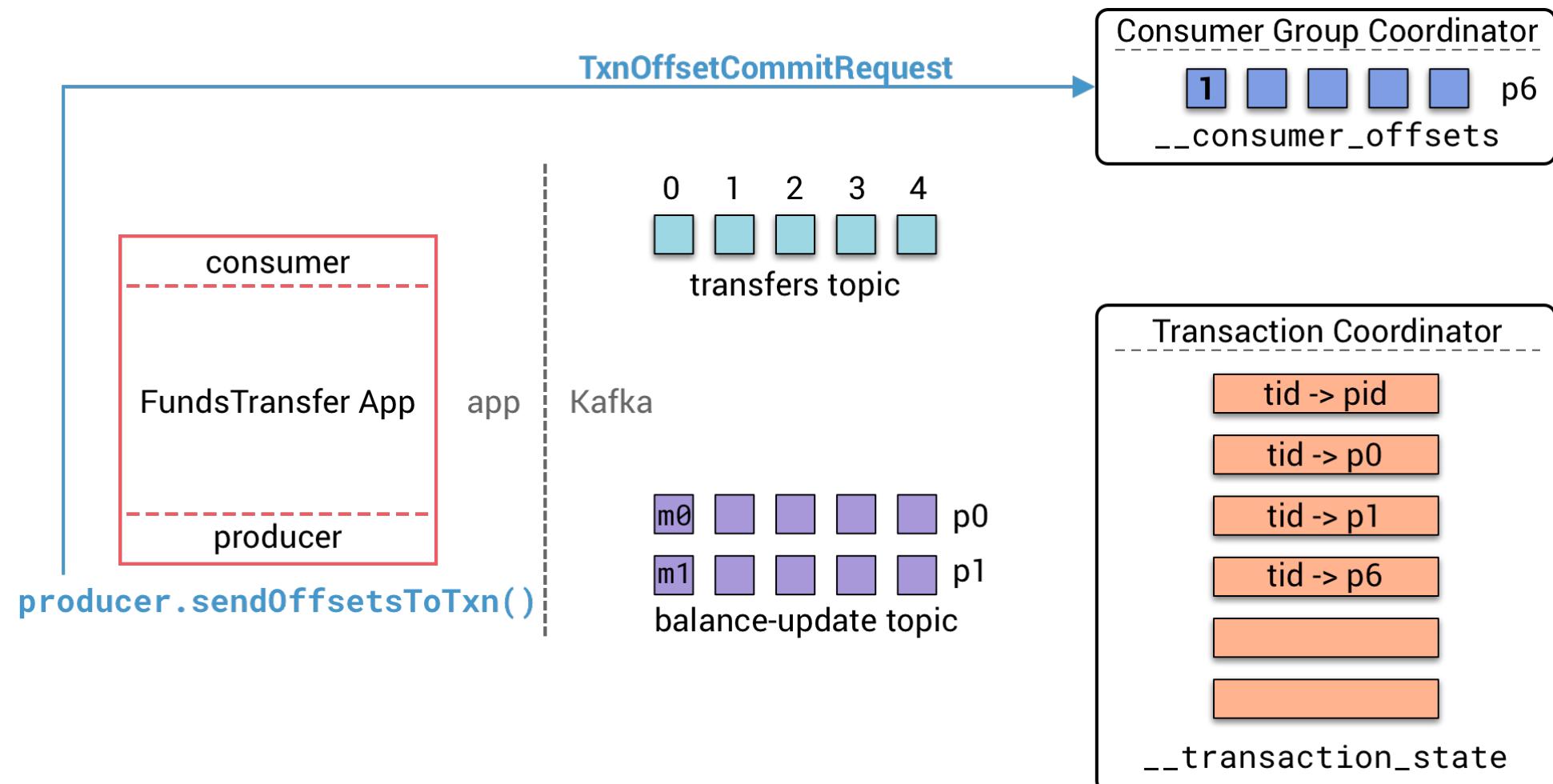
Transactions - Send (8/14)



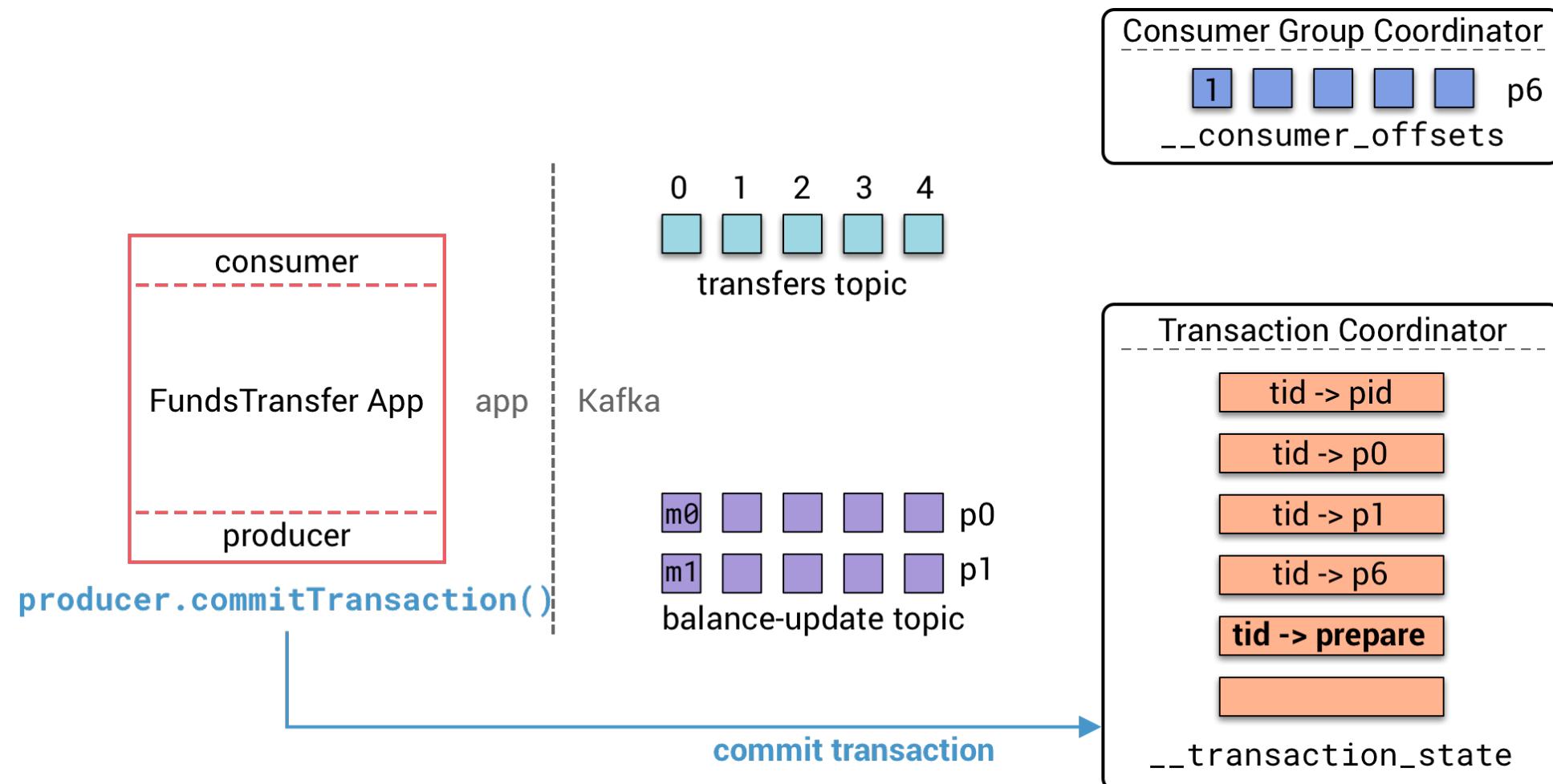
Transactions - Track Consumer Offset (9/14)



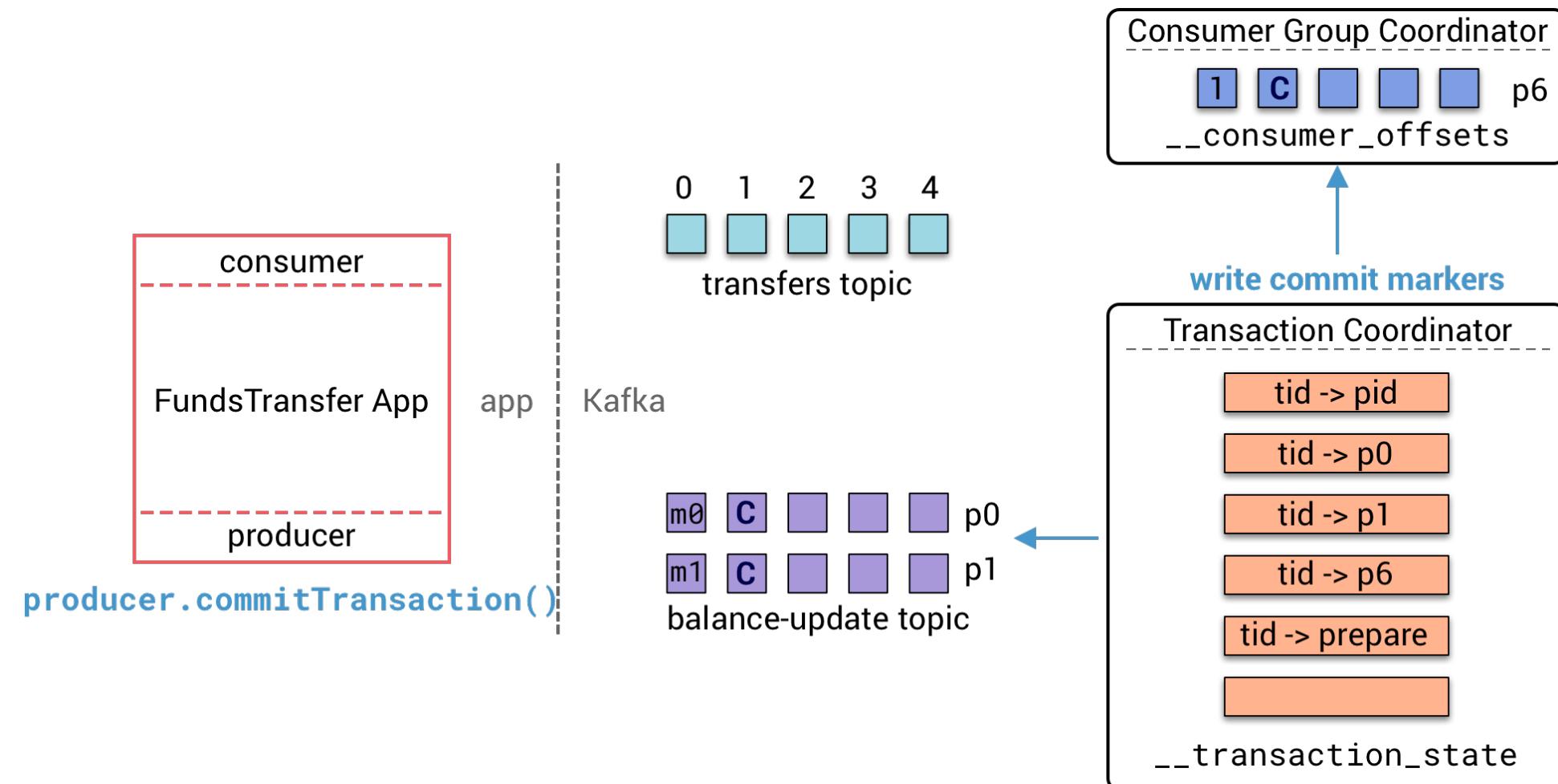
Transactions - Commit Consumer Offset (10/14)



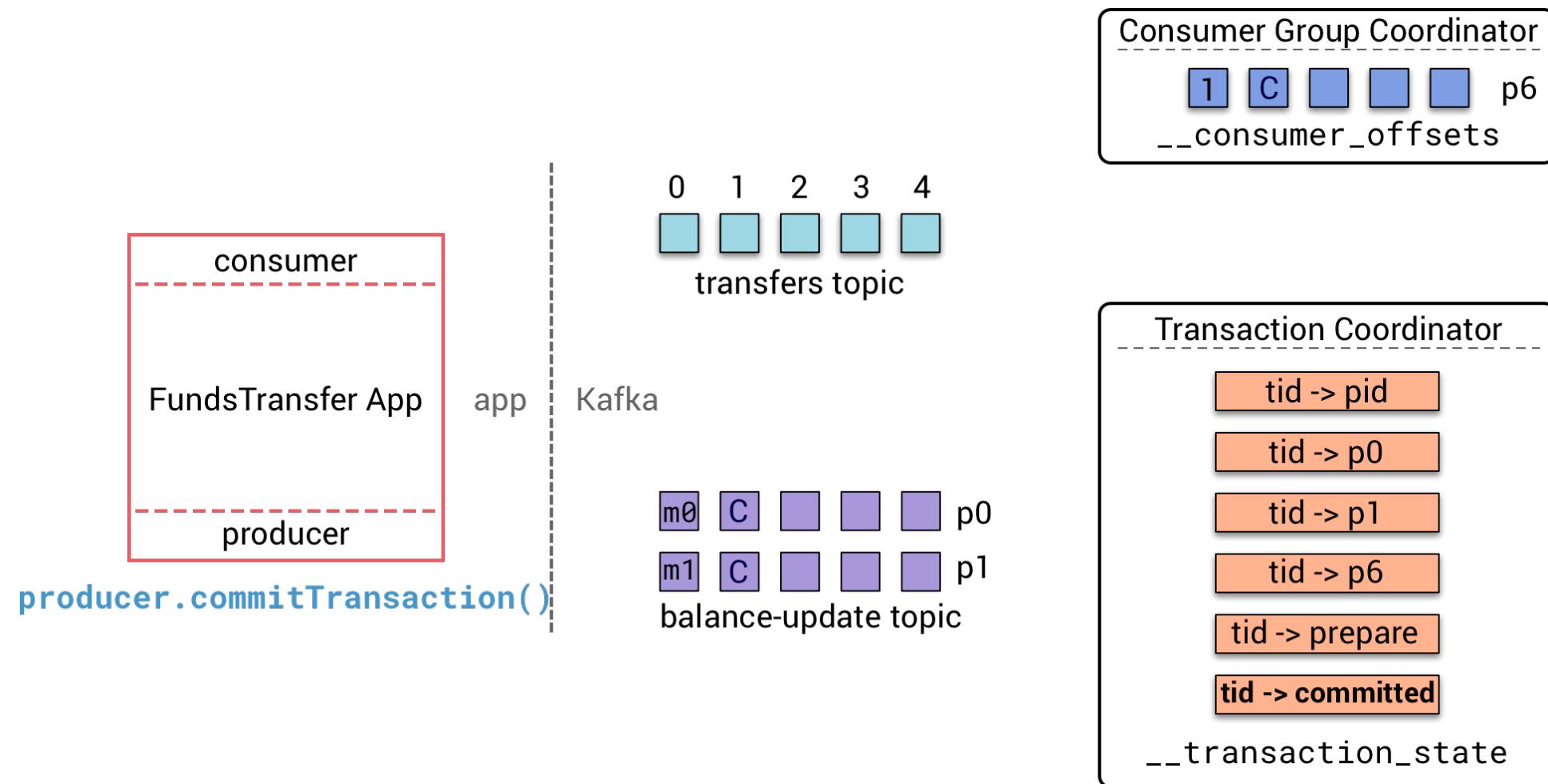
Transactions - Prepare Commit (11/14)



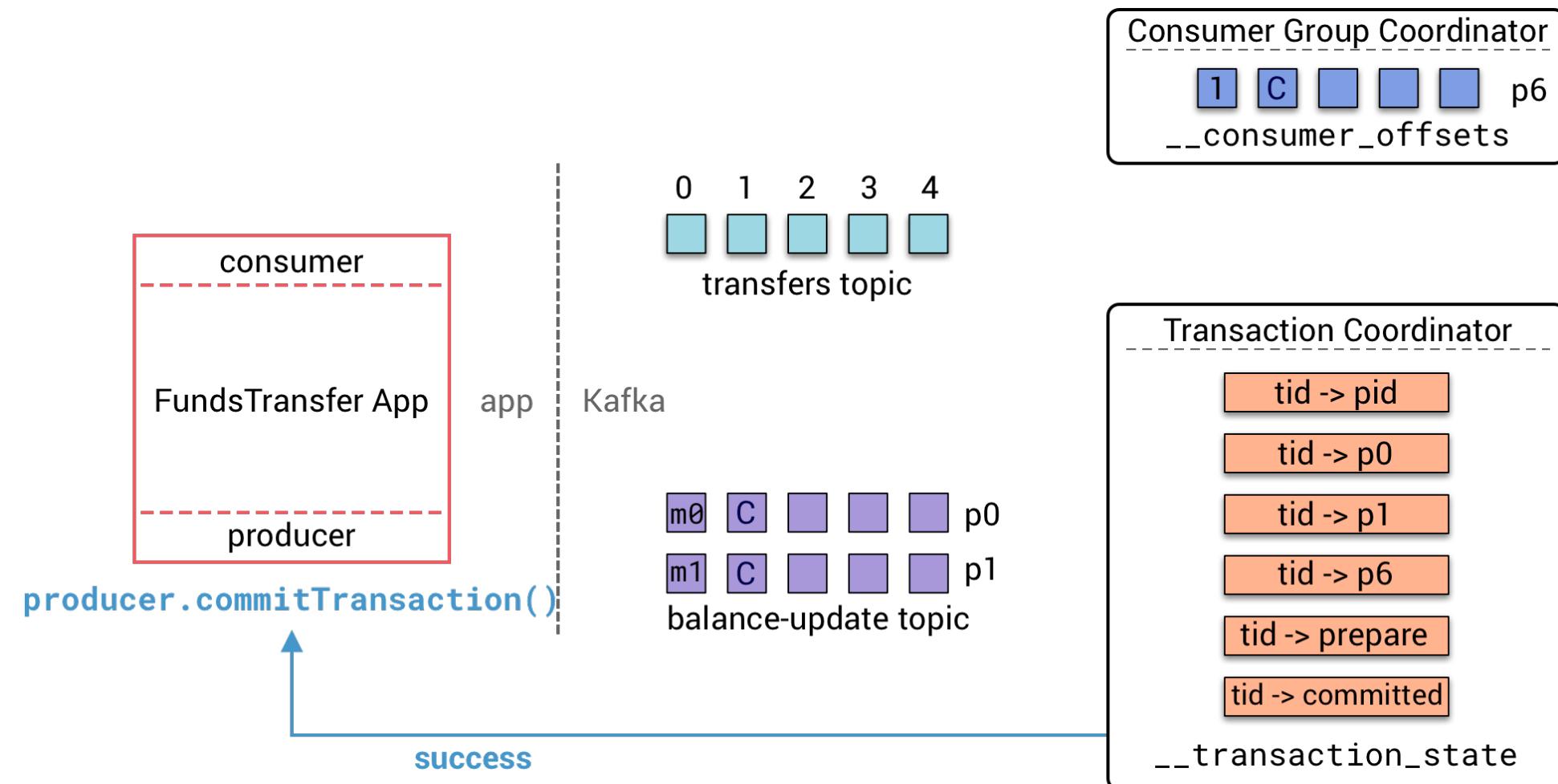
Transactions - Write Commit Markers (12/14)



Transactions - Commit (13/14)



Transactions - Success (14/14)



Appendix B: How Can You Monitor Replication?

Description

Monitoring considerations for replication.

Monitoring Leader Election Rate

- Leader election rate (JMX metric)

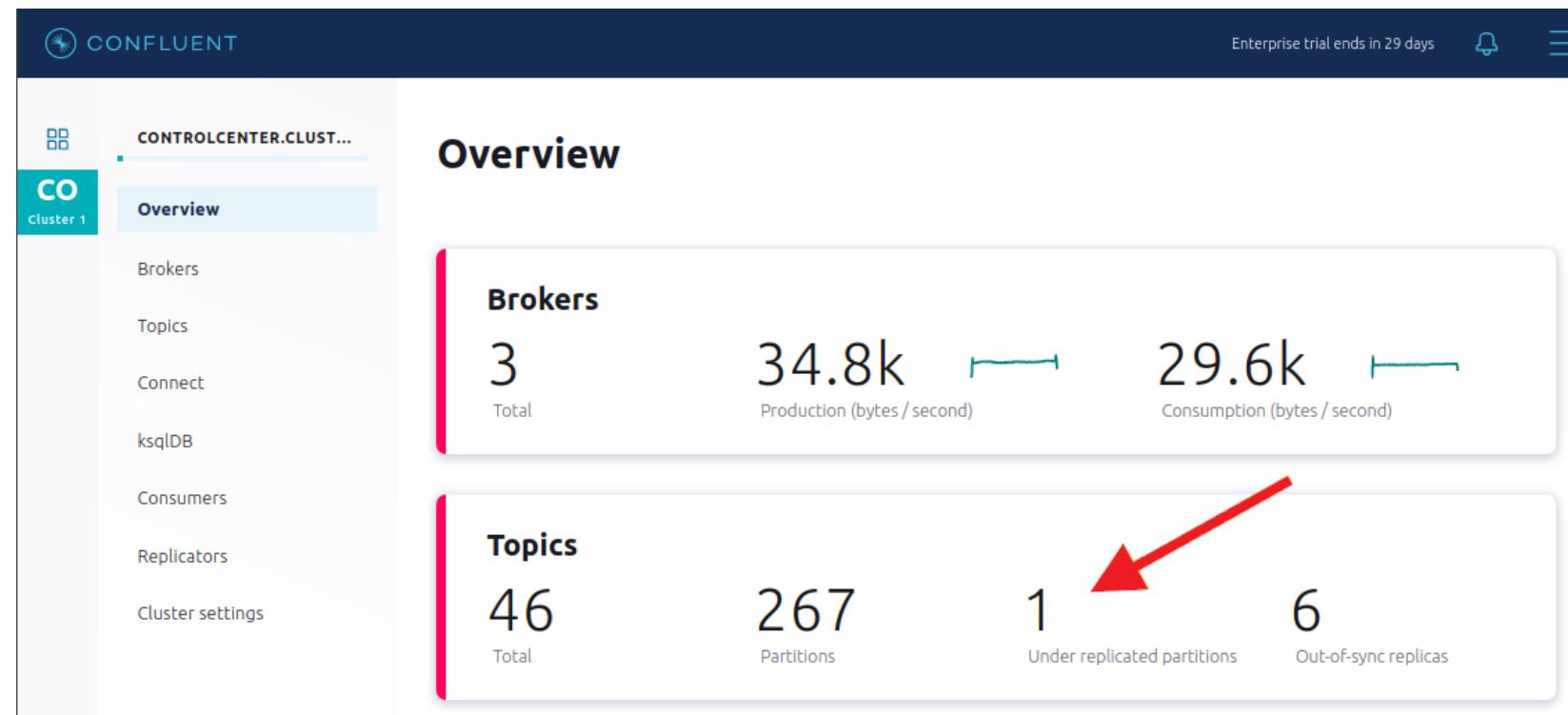
```
kafka.controller:type=ControllerStats, name=LeaderElectionRateAndTimeMs
```

Monitoring ISR

- Monitor under-replicated partitions with the JMX metric:
 - `kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions`
 - Alert if the value is greater than 0 for a long time
- Track changes of ISR lists (shrinks and adds) with these JMX metrics:
 - `kafka.server:type=ReplicaManager,name=IsrExpandsPerSec`
 - `kafka.server:type=ReplicaManager,name=IsrShrinksPerSec`

Monitoring for Under Replicated Partitions

- If a broker goes down, the ISR for some partitions will shrink
- Confluent Control Center shows partition health at a glance:



Monitoring Offline Partitions

- Track offline partitions with JMX metric:
 - `kafka.controller:type=KafkaController,Name=OfflinePartitionsCount`
- Leader failure makes partition unavailable until re-election
 - Producer `send()` will retry according to `retries` configuration
 - Callback raises `NetworkException` if `retries == 0`

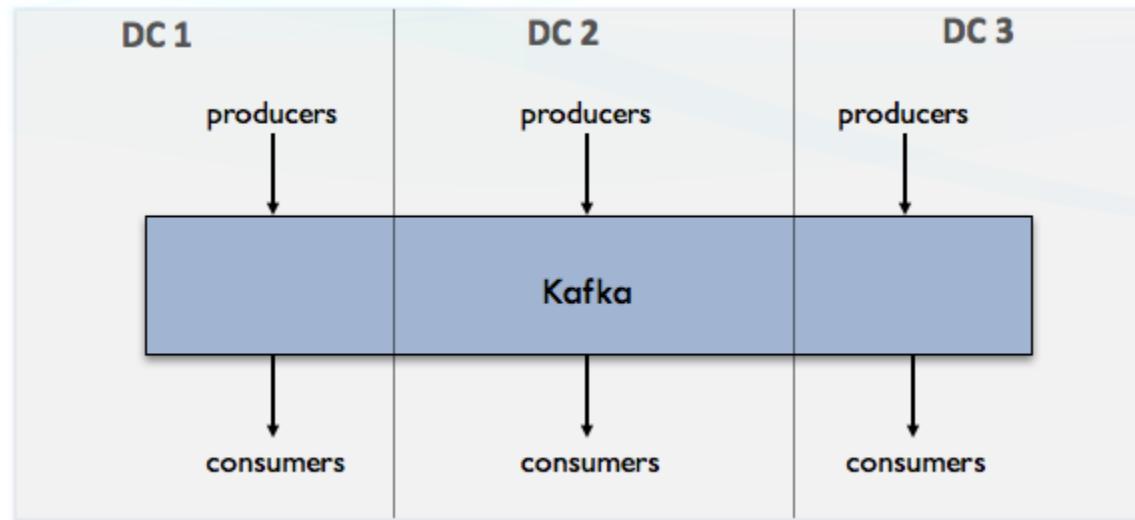
Appendix C: Multi-Region Clusters

Multiple Data Centers

- Kafka only:
 - Stretched (a.k.a. multi-AZ)
- Confluent Replicator:
 - Cluster aggregation
 - Active/Passive
 - Active/Active



Stretched Deployment



- Discussion Questions
 - How should you deploy ZooKeeper and Kafka across 3 availability zones?
 - What are possible tradeoffs between a stretched cluster vs. a single DC cluster?
 - What are some possible failure scenarios and how does Kafka respond?

Confluent Replicator or Apache Kafka MirrorMaker

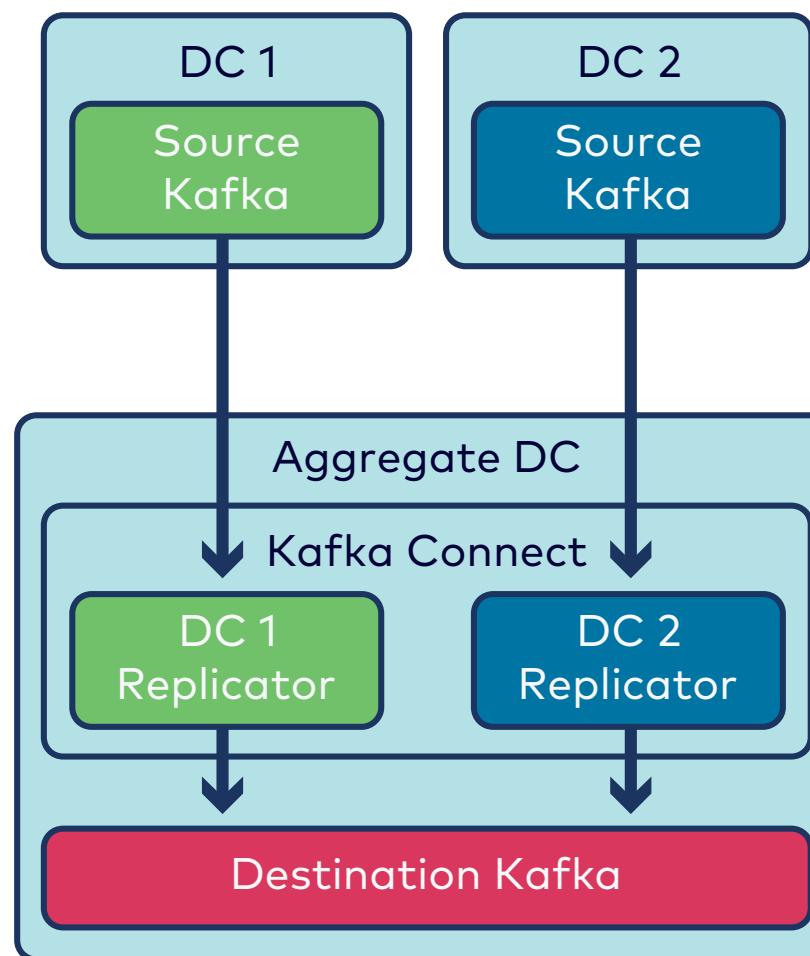
Deploying Replicator

1. Provision machines in **destination data center**
2. Install with `confluent-hub` if not already using CP
3. Configure `worker.properties` file on each Connect machine
4. Ways to start Replicator:
 - Submit HTTP request with Replicator-specific properties, **or**
 - Use the `replicator` command on each Kafka Connect machine

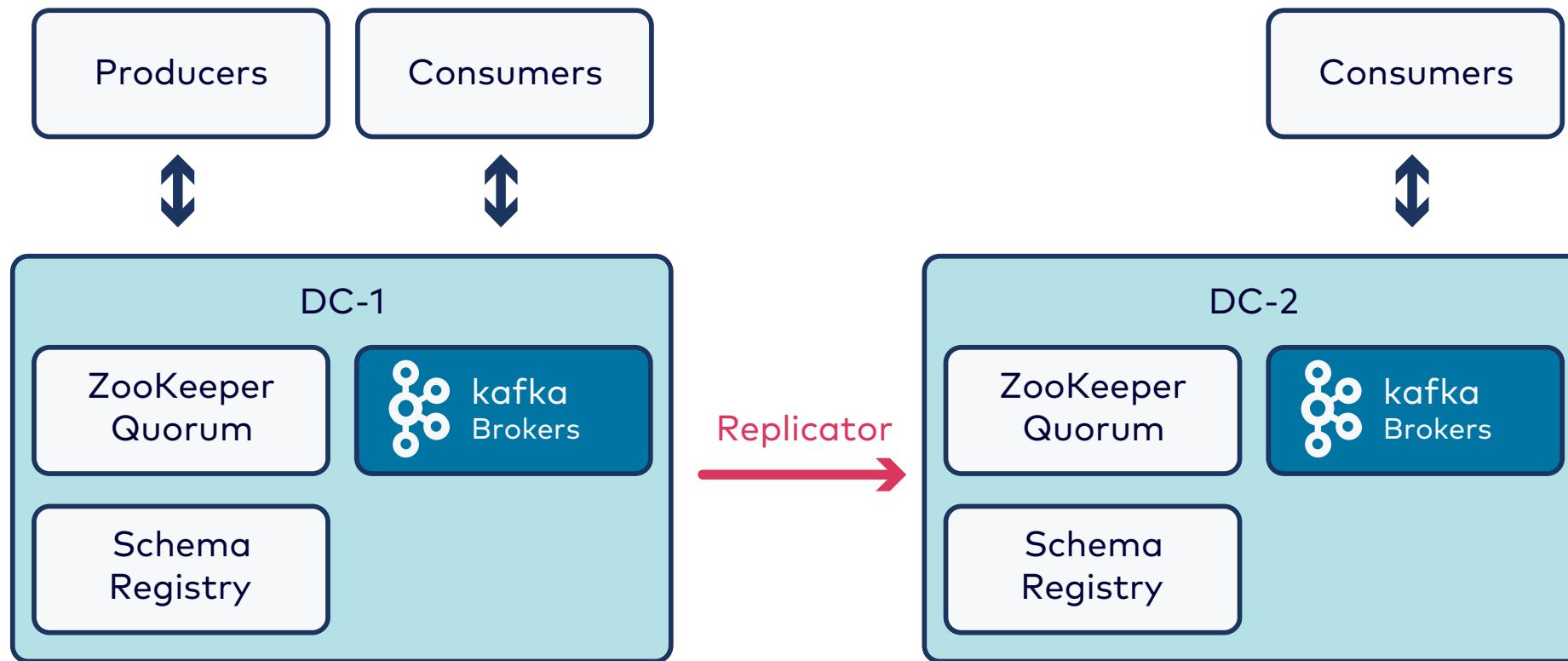


Confluent Replicator requires an enterprise license

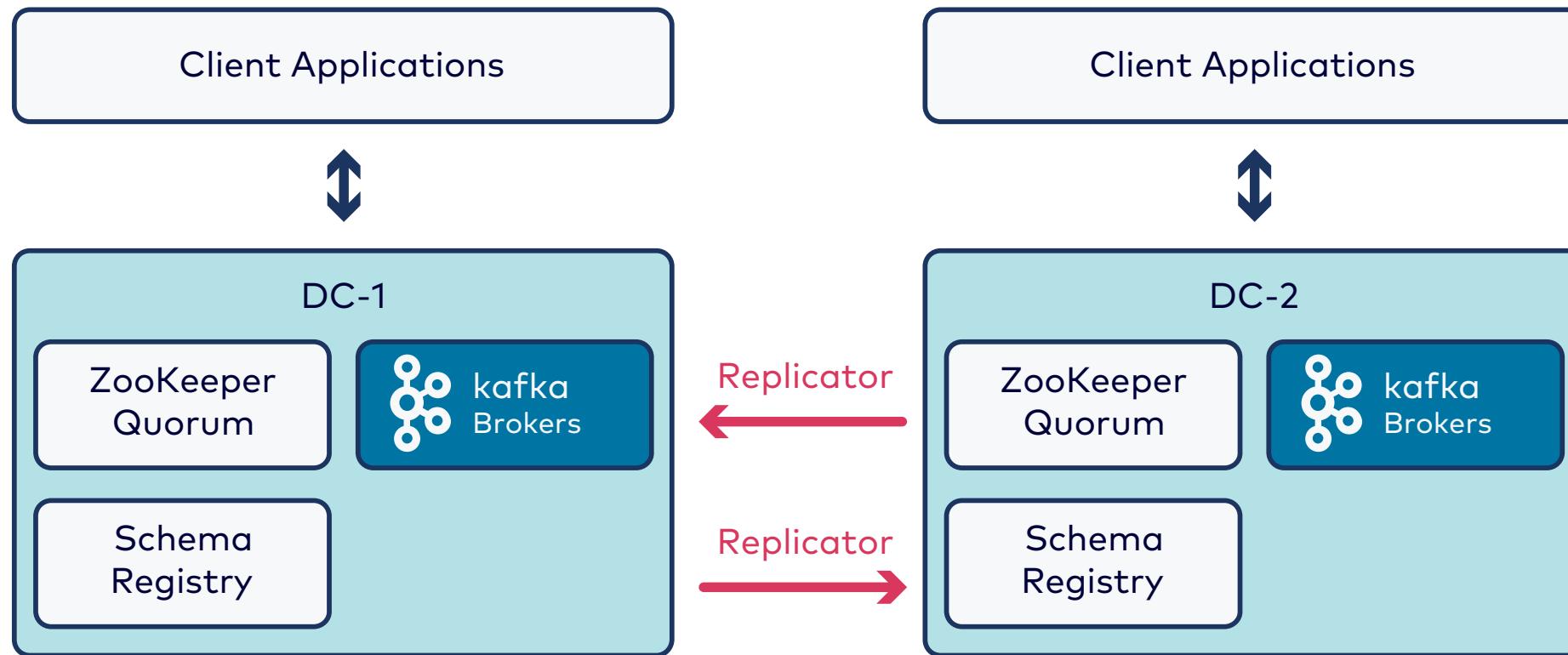
Cluster Aggregation



Active-Passive

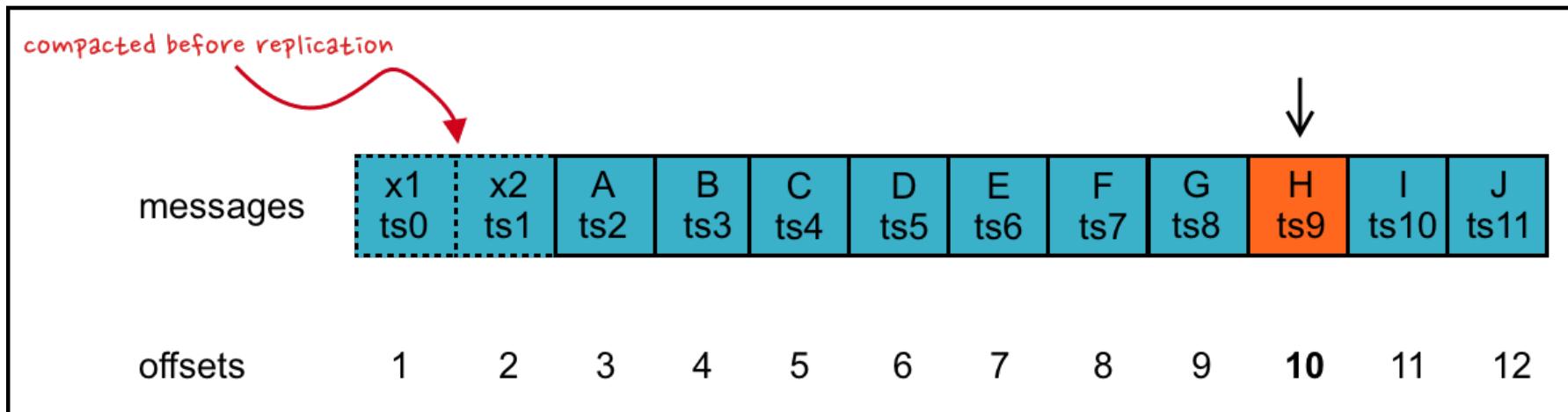


Active-Active



Client Offset Translation (1)

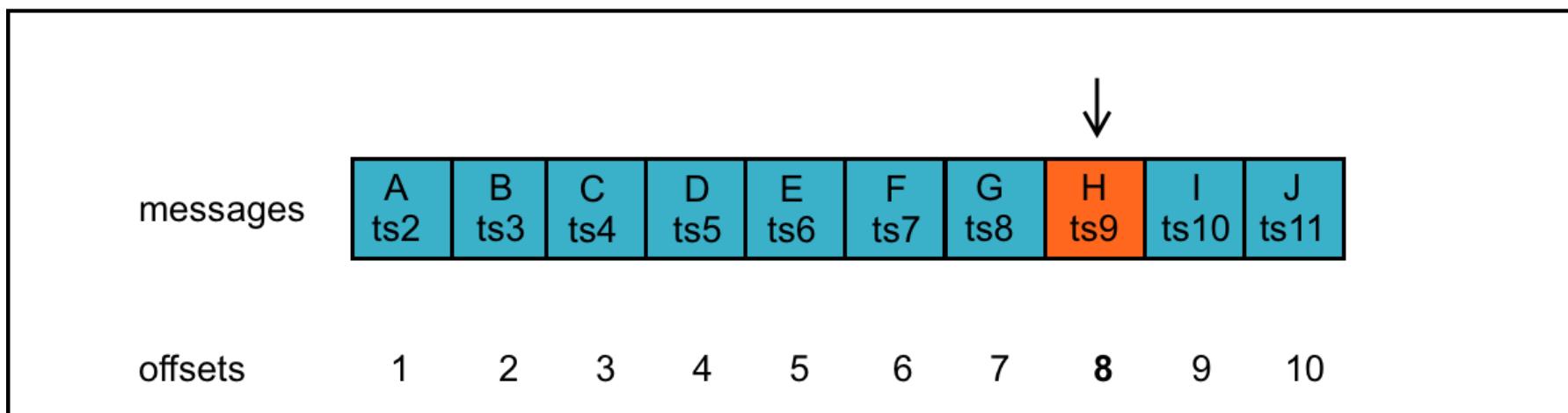
Data Center 1



Consumer Group	Offset	Timestamp
my-group-1	10	ts9



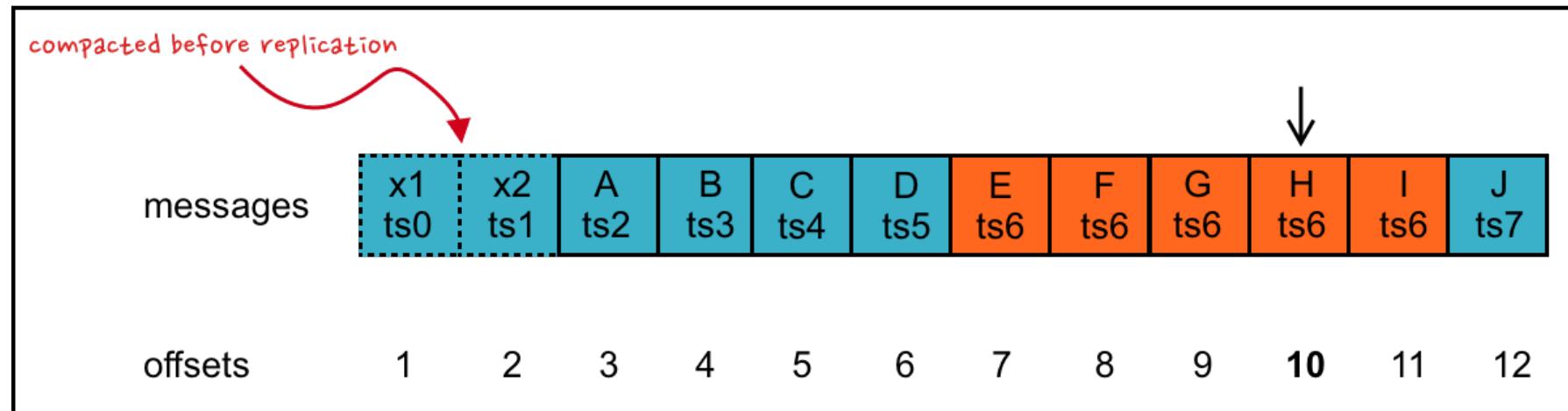
Data Center 2



Consumer Group	Offset
my-group-1	8

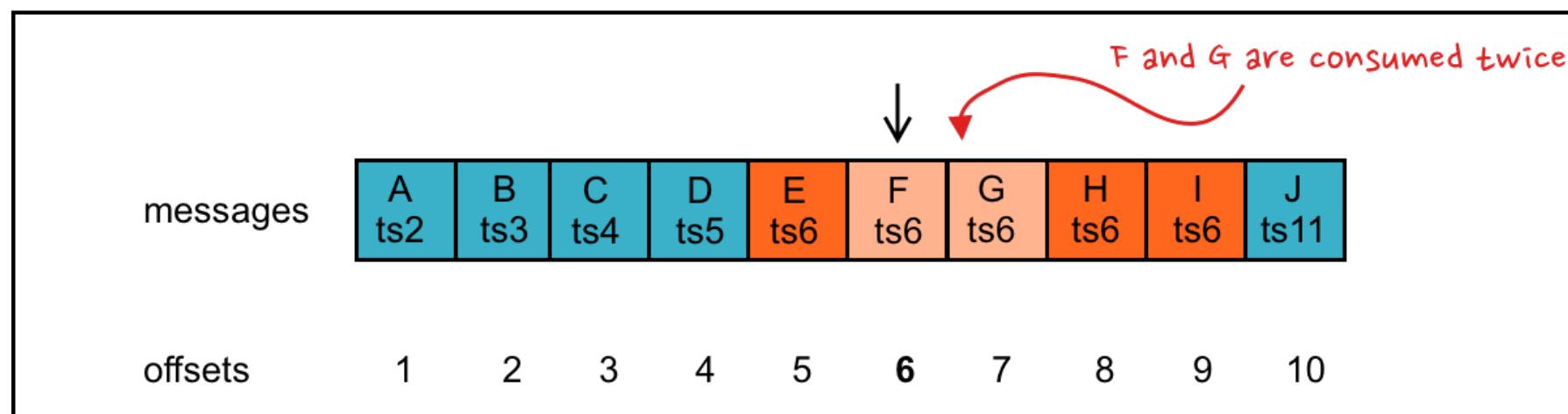
Client Offset Translation (2)

Data Center 1



Consumer Group	Offset	Timestamp
my-group-1	10	ts6

Data Center 2



Consumer Group	Offset
my-group-1	6

Manual Disaster Recovery: Consumer Group Tool

The command `kafka-consumer-groups` can set a Consumer Group to seek to an offset derived from a timestamp

```
$ kafka-consumer-groups \
  --bootstrap-server dc2-broker-101:9092 \
  --reset-offsets \
  --topic dc1-topic \
  --group my-group \
  --execute \
  --to-datetime 2017-08-01T17:14:23.933
```

Manual Disaster Recovery: Java Client API

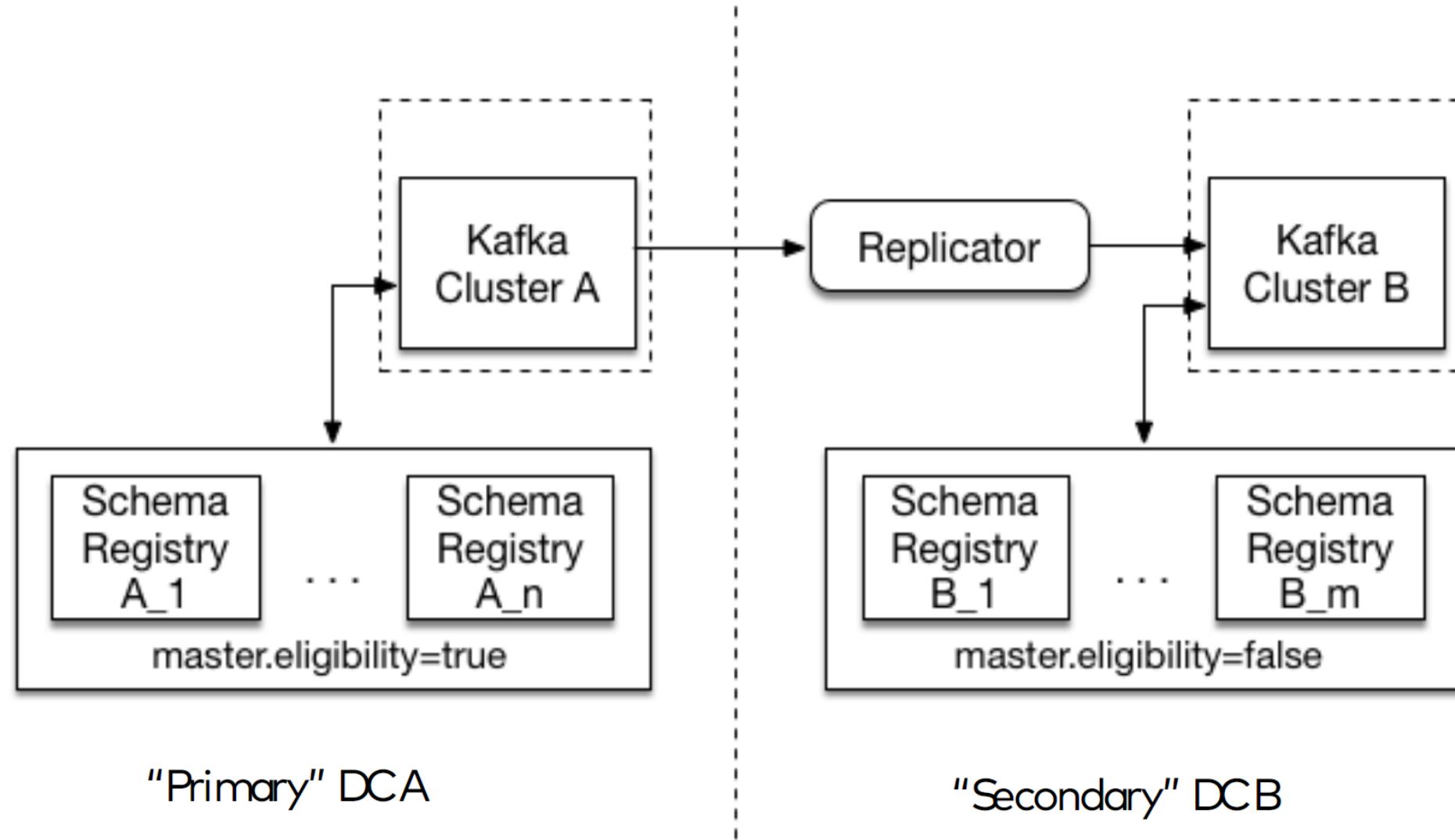
Producer API:

- Replication process must use same partitioner class to preserve message ordering for keyed messages
- Topic properties must be same in source and destination clusters (automatic with Replicator)

Consumer API:

- Use `offsetsForTimes()` method to find an offset for a given timestamp
- Use `seek()` to move to the desired offset

Schema Registry Across Data Centers



Improving Network Utilization

- If network latency is high, increase the TCP socket buffer size in Kafka
 - `socket.send.buffer.bytes` on the origin cluster's broker (default: 102400 bytes)
 - `receive.buffer.bytes` on Replicator's consumer (default: 65536 bytes)
 - Increase corresponding OS socket buffer size

Appendix D: SSL and SASL Details

Overview

This appendix contains two sections (as one appendix):

- SSL
- SASL



This section is a literal copy of old content - no formatting done, just making sure everything renders.

Why is SSL Useful?

- Organization or legal requirements
- One-way authentication
 - Secure wire transfer using encryption
 - Client knows identity of Broker
 - Use case: Wire encryption during cross-data center mirroring
- Two-way authentication with **Mutual SSL**
 - Broker knows the identity of the client as well
 - Use case: authorization based on SSL principals
- Easy to get started
 - Just requires configuring the client and server
 - No extra servers needed, but can integrate with enterprise

SSL Data Transfer

- After an initial handshake, data is encrypted with the agreed-upon encryption algorithm
- There is overhead involved with data encryption:
 - Overhead to encrypt/decrypt the data
 - Can no longer use zero-copy data transfer to the Consumer
 - SSL overhead will increase

```
kafka.network:type=RequestMetrics,name=ResponseSendTimeMs
```

SSL Performance Impact

- Performance was measured on Amazon EC2 r3.xlarge instances

	Throughput(MB/s)	CPU on client	CPU on Broker
Producer (plaintext)	83	12%	30%
Producer (SSL)	69	28%	48%
Consumer (plaintext)	83	8%	2%
Consumer (SSL)	69	27%	24%

Data at Rest Encryption

- "Data at rest encryption" refers to encrypting data stored on a hard drive that is currently not moving through the network
- Options for encrypting data at rest:
 1. Linux OS encryption utilities that encrypt full disk or disk partitions (e.g. LUKS)
 2. Producers encrypt messages before sending to Kafka, and Consumers decrypt after consuming
- Consider running Brokers on a machine with an encrypted filesystem or encrypted RAID controller

SSL: Keystores and Truststores

Kafka client truststore.jks



CA certificate

Kafka Broker keystore.jks



Broker private key



Broker certificate
with CA signature



- Client truststore:
 - CA certificate used to verify signature on Broker cert
 - If signature is valid, Broker is **trusted**
- Broker keystore:
 - Uses private key to create **certificate**
 - Cert must be signed by Certificate Authority (CA)
 - Cert includes public key, which client will use to establish secure connection

Preparing for SSL

1. Generate certificate (X.509) in Broker **keystore**
2. Create your own Certificate Authority (CA) or use a well known CA
3. Sign Broker certificate with CA
4. Import signed certificate and CA certificate to Broker **keystore**
5. Import CA certificate to client **truststore**



Mutual SSL: generate client **keystore** and corresponding Broker **truststore** in the same way.

SSL Everywhere! (1)

- Client and Broker `*.properties` Files:

```
ssl.keystore.location = /var/private/ssl/kafka.server.keystore.jks  
ssl.keystore.password = password-to-keystore-file  
ssl.key.password = password-to-private-key  
ssl.truststore.location = /var/private/ssl/kafka.server.truststore.jks  
ssl.truststore.password = password-to-truststore-file
```

- Brokers:

```
listeners = SSL://<host>:<port>  
ssl.client.auth = required  
inter.broker.listener.name = SSL
```

- Client:

```
security.protocol = SSL
```

SSL Everywhere! (2)

Discussion Questions:

- How could you secure the clear credentials that are stored in the `*.properties` files?
- What configurations would you change to do one-way SSL (SSL from Broker to client only) and plaintext between Brokers?

SSL Principal Name

- By default, Principal Name is the distinguished name of the certificate

```
CN=host1.example.com,OU=organizational unit,O=organization,L=location,ST=state,C=country
```

- Can be customized through `principal.builder.class`
 - Has access to X509 Certificate
 - Makes setting the Broker principal and application principal convenient

Troubleshooting SSL

- To troubleshoot SSL issues
 - Verify all Broker security configurations
 - Verify client security configuration
 - On the Broker, check that the following command returns a certificate:

```
$ openssl s_client -connect <broker>:<port> -tls1
```

- Enable SSL debugging using the `KAFKA_OPTS` environment variable
 - The output can be verbose but it will show the SSL handshake sequence, etc.
 - If you are debugging on the Broker, you must restart the Broker

```
$ export KAFKA_OPTS="-Djavax.net.debug=ssl $KAFKA_OPTS"
```

SASL for Authentication (1)

- SASL: Simple Authentication and Security Layer
 - Challenge/response protocols
 - Server issues challenge; client sends response
 - Continues until server is satisfied
- All of the SASL authentication methods send data over the wire in **PLAINTEXT** by default, but SASL authentication can (should) be combined with **SSL** transport encryption.

SASL for Authentication (2)

- SASL supports different mechanisms
 - GSSAPI: Kerberos
 - SCRAM-SHA-256, SCRAM-SHA-512: “salted” and hashed passwords
 - SCRAM (Salted Challenge Response Authentication Mechanism)
 - Can use username/password stored in ZK or delegation token (OAuth 2)
 - PLAIN: cleartext username/password
 - OAUTHBEARER: authentication tokens

Using SASL SCRAM

- SCRAM should be used with SSL for secure authentication
- ZooKeeper is used for the credential store
 - Create credentials for Brokers and clients
 - Credentials must be created before Brokers are started
 - ZooKeeper should be secure and on a private network

Configuring SASL SCRAM Credentials

- Create credentials for inter-Broker communication (user "admin")

```
$ kafka-configs --bootstrap-server broker_host:port \
  --alter \
  --add-config \
  'SCRAM-SHA-256=[password=admin-secret],SCRAM-SHA-512=[password=admin-secret]' \
  --user admin
```

- Create credentials for Broker-client communication (e.g. user "alice")

```
$ kafka-configs --bootstrap-server broker_host:port \
  --alter \
  --add-config \
  'SCRAM-SHA-256=[password=alice-secret],SCRAM-SHA-512=[password=alice-secret]' \
  --user alice
```

Configuring SASL Using a JAAS File (Broker)

- JAAS (Java Authentication and Authorization Service) can be included in `*.properties` files on clients and Brokers using the `sasl.jaas.config` property

Broker (`server.properties`):

```
...
listeners=SASL_SSL://<host>:<port>
inter.broker.listener.name=SASL_SSL
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-256
sasl.enabled.mechanisms=SCRAM-SHA-256

listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config= \
    org.apache.kafka.common.security.scram.ScramLoginModule required \
    username="admin" \
    password="admin-secret";
...
```

Configuring SASL Using a JAAS File (Client)

Client configuration file:

```
...
security.protocol=SASL_SSL
sasl.mechanism=SCRAM-SHA-256

sasl.jaas.config= \
    org.apache.kafka.common.security.scram.ScramLoginModule required \
    username="alice" \
    password="alice-secret";
...
```

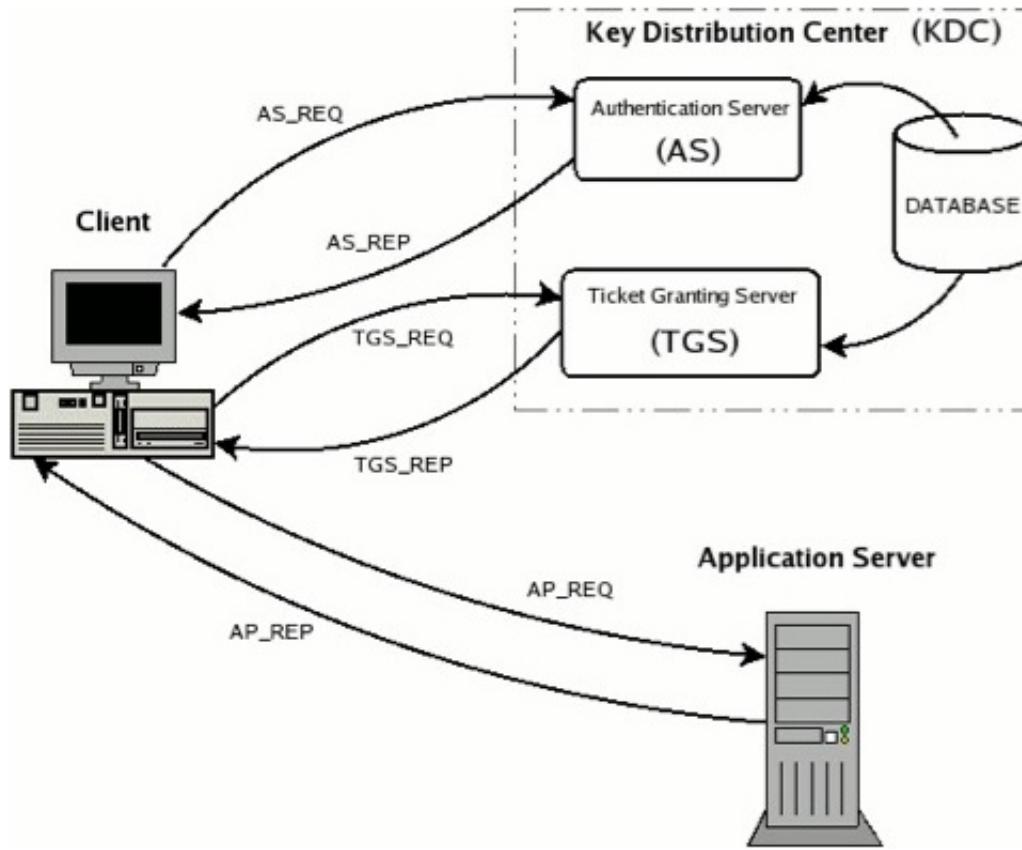
Discussion questions:

- Why is `SASL_SSL` wire encryption recommended when using SASL SCRAM?
- What are the tradeoffs of an environment where clients authenticate with SASL SCRAM over SSL, but inter-Broker communication is done over plaintext? How would you change these `*.properties` files for this situation?
- Notice again that clear passwords are stored in these files. What would you do to ensure the security of these credentials?

Why Kerberos?

- Kerberos provides secure single sign-on
 - An organization may provide multiple services, but a user just needs a single Kerberos password to use all services
- More convenient where there are many users
- Requires a Key Distribution Center (KDC)
 - Each service and each user must register a Kerberos principal in the KDC

How Kerberos Works



1. Services authenticate with the Key Distribution Center on startup
 - a. Client authenticates with the Authentication Server on startup
 - b. Client obtains a service ticket from Ticket Granting Server
2. Client authenticates with the service using the service ticket

Preparing Kerberos

- Create principals in the KDC for:
 - Each Kafka Broker
 - Application clients
- Create Keytabs for each principal
 - Keytab includes the principal and encrypted Kerberos password
 - Allows authentication without typing a password

The Kerberos Principal Name

- Kerberos principal
 - Primary[/Instance]@REALM
 - Examples:
 - kafka/kafka1.hostname.com@EXAMPLE.COM
 - kafka-client-1@EXAMPLE.COM
- Primary is extracted as the default principal name
- Can customize the username through sasl.kerberos.principal.to.local.rules

Configuring Kerberos (Broker)

Broker configuration file:

```
listeners = SASL_PLAINTEXT://<host>:<port>    # or SASL_SSL://<host>:<port>
inter.broker.listener.name=SASL_PLAINTEXT      # or SASL_SSL
sasl.kerberos.service.name=kafka

listener.name.sasl_plaintext.gssapi.sasl.jaas.config= \
  com.sun.security.auth.module.Krb5LoginModule required \
  useKeyTab=true \
  storeKey=true \
  keyTab="/etc/security/keytabs/kafka_server.keytab" \
  principal="kafka/kafka1.hostname.com@EXAMPLE.COM";
```

Configuring Kerberos (Client)

Client configuration file:

```
...
security.protocol=SASL_PLAINTEXT      # or SASL_SSL
sasl.kerberos.service.name=kafka

sasl.jaas.config= \
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/etc/security/keytabs/kafka_client.keytab"
    principal="kafka-client-1@EXAMPLE.COM";
...
...
```

Advanced Security Configurations

- Configure multiple SASL mechanisms on the Broker
 - Useful for mixing internal (e.g., GSSAPI) and external (e.g., SCRAM) clients

```
sasl.enabled.mechanisms = GSSAPI,SCRAM-SHA-256
```

- Configure different listeners for different sources of traffic
 - Useful to designate one interface for clients and one interface for replication traffic:

```
listeners = CLIENTS://kafka-1a:9092,REPLICATION://kafka-1b:9093
```

- Listeners can have any name as long as `listener.security.protocol.map` is defined to map each name to a security protocol:

```
listener.security.protocol.map=CLIENTS:SASL_SSL,REPLICATION:SASL_PLAINTEXT
```

- Specify listener for communication between brokers and controller with `control.plane.listener.name`

Troubleshooting SASL

- To troubleshoot SASL issues
 - Verify all Broker security configurations
 - Verify client security configuration
 - Verify JAAS files and passwords
 - Enable SASL debugging for Kerberos using the `KAFKA_OPTS` environment variable
 - If you are debugging on the Broker, you will have to restart the Broker

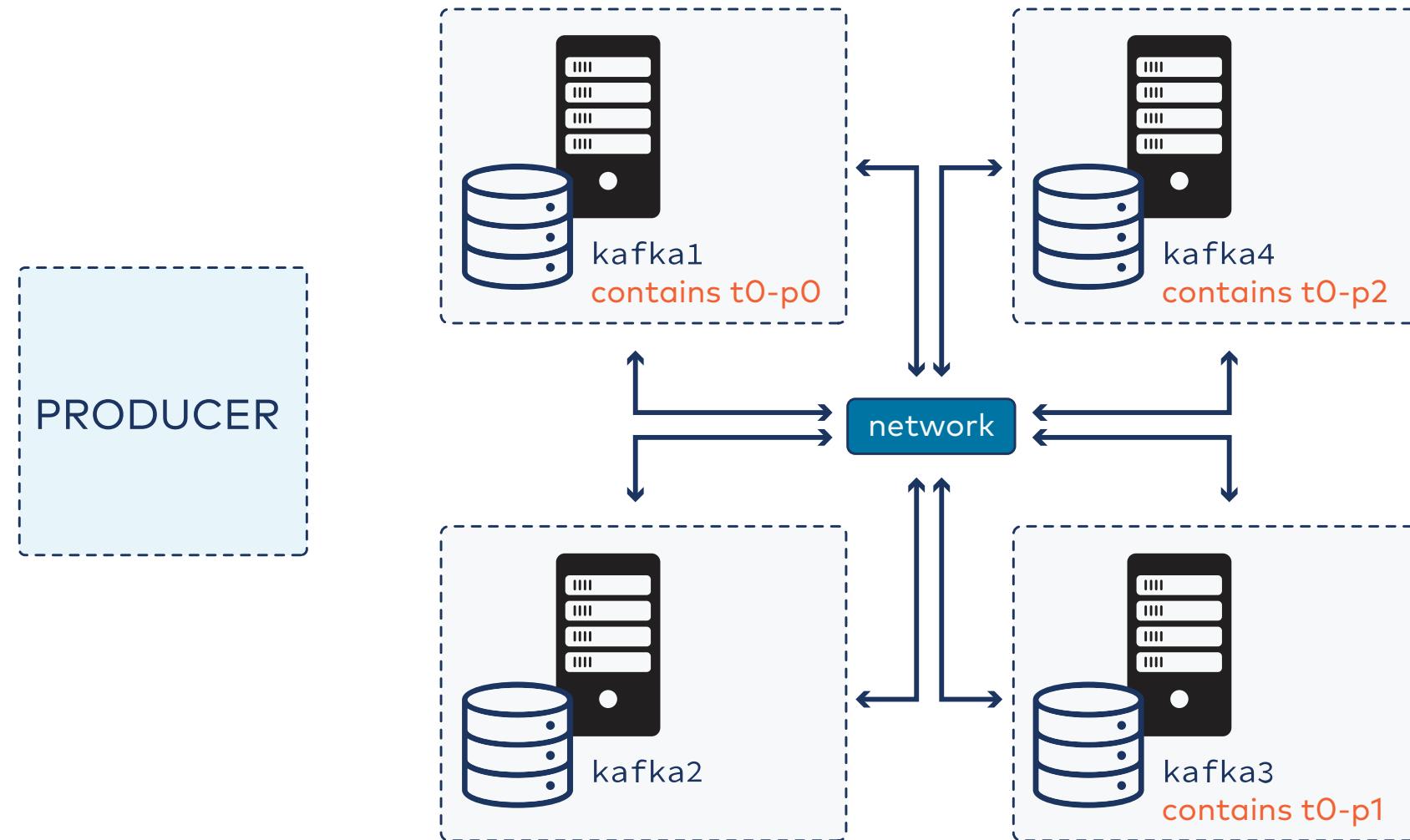
```
$ export KAFKA_OPTS="-Dsun.security.krb5.debug=true"
```

Appendix E: How Can You Connect to a Cluster?

Description

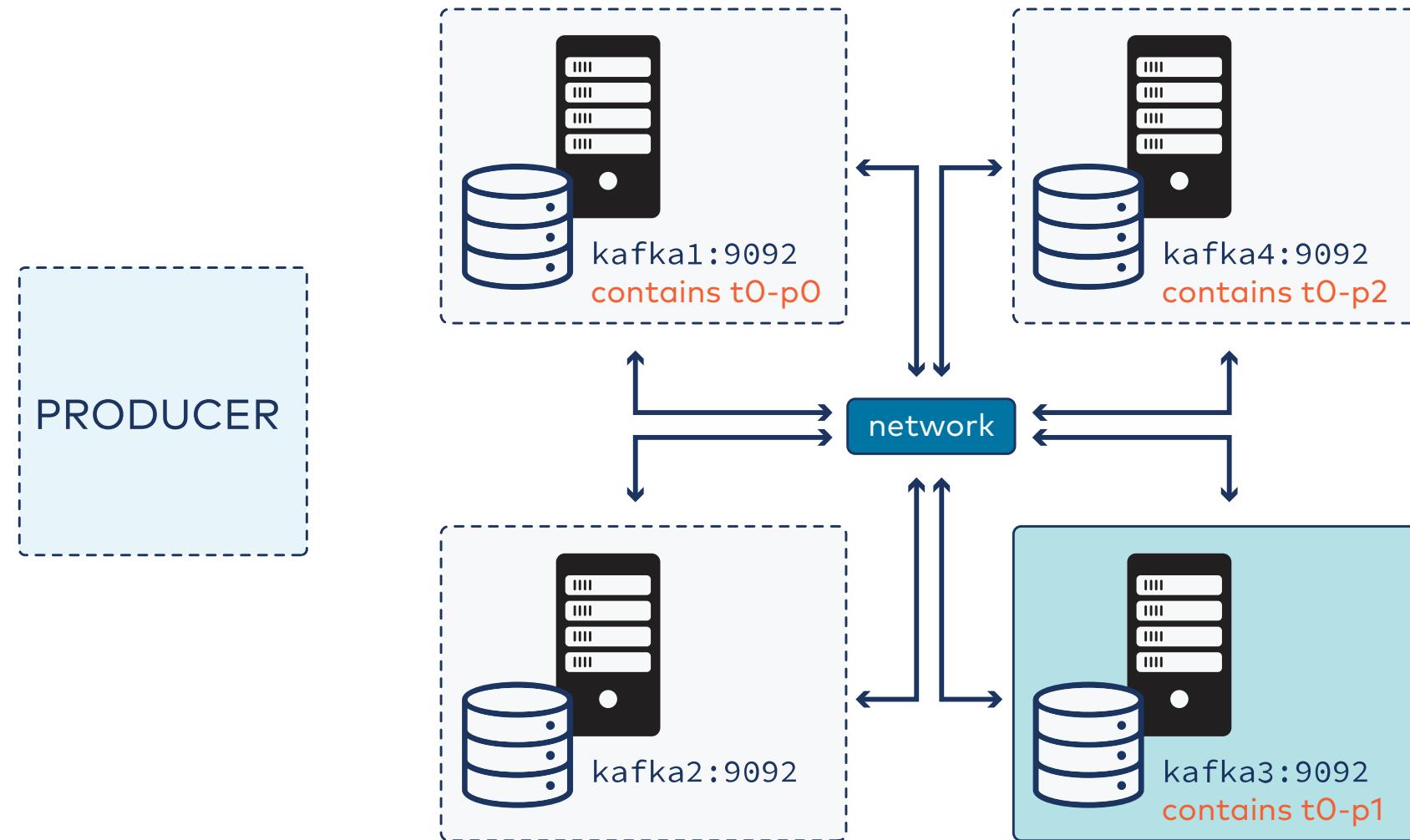
How brokers are interconnected and how this allows us to leverage bootstrap servers. Best practice for bootstrap servers and examples of how to configure for various clients and in a CLI example as well.

View of a Cluster and Producer



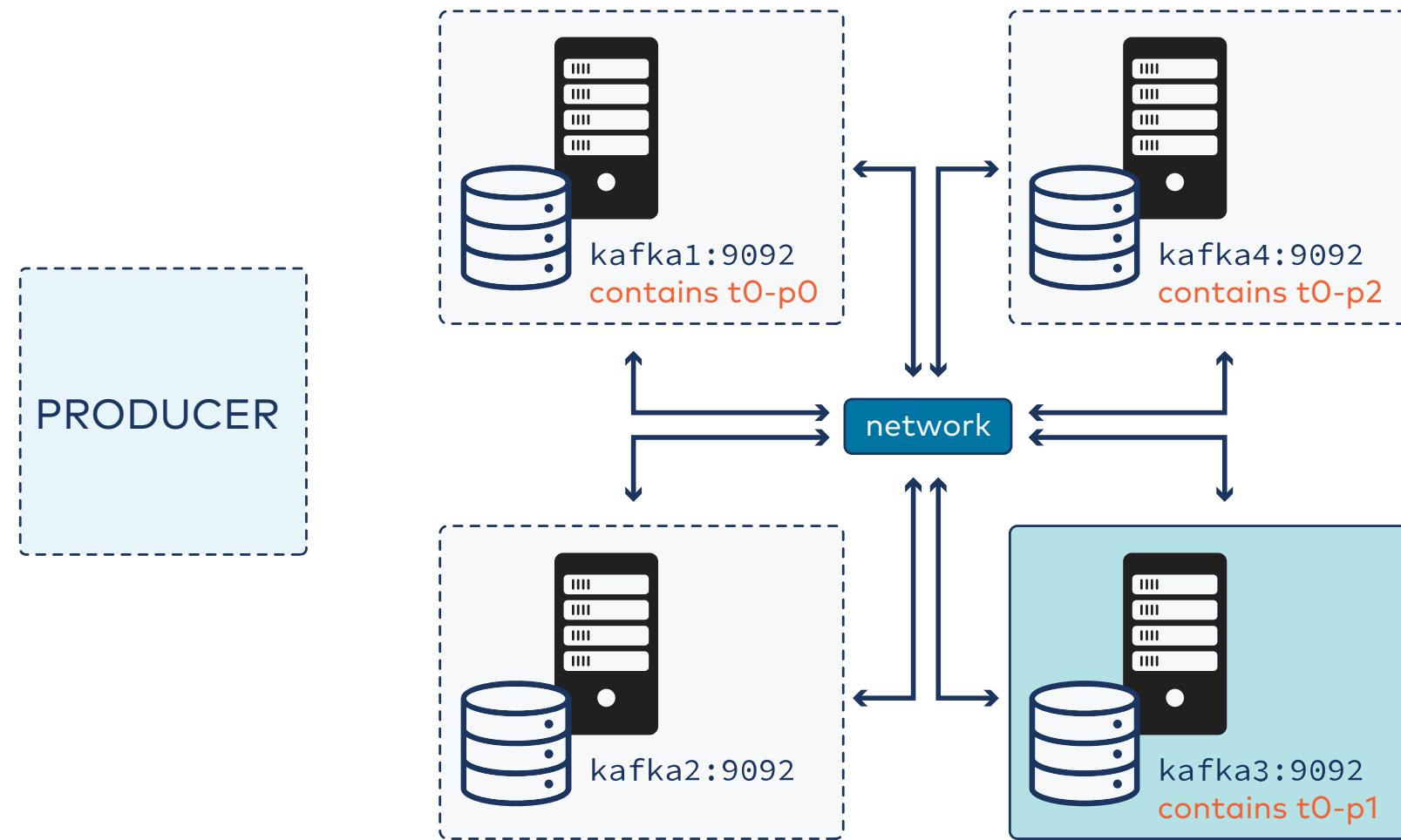
- Producer wants to send new message with key 10
- Producer chooses partition (how?)

Identifying Brokers



- Brokers identified by **host:port** pairs
- Which broker do we write our new message to?

Bootstrap Server



- Clients, like producers, need to specify a **bootstrap server**
- This is the **initial** connection to the cluster and all you need to specify in your code, configuration

Bootstrap Servers in Practice (1)

Specify bootstrap server in CLI commands, e.g.,

```
kafka-topics  
  --bootstrap-server kafka:9092  
  --create  
  --partitions 1  
  --replication-factor 1  
  --topic testing
```

Specify bootstrap server in client code, e.g.,

```
props.put("bootstrap.servers", "kafka:9092");
```

Specify bootstrap server in a properties file, e.g. for a Connect worker, e.g.,

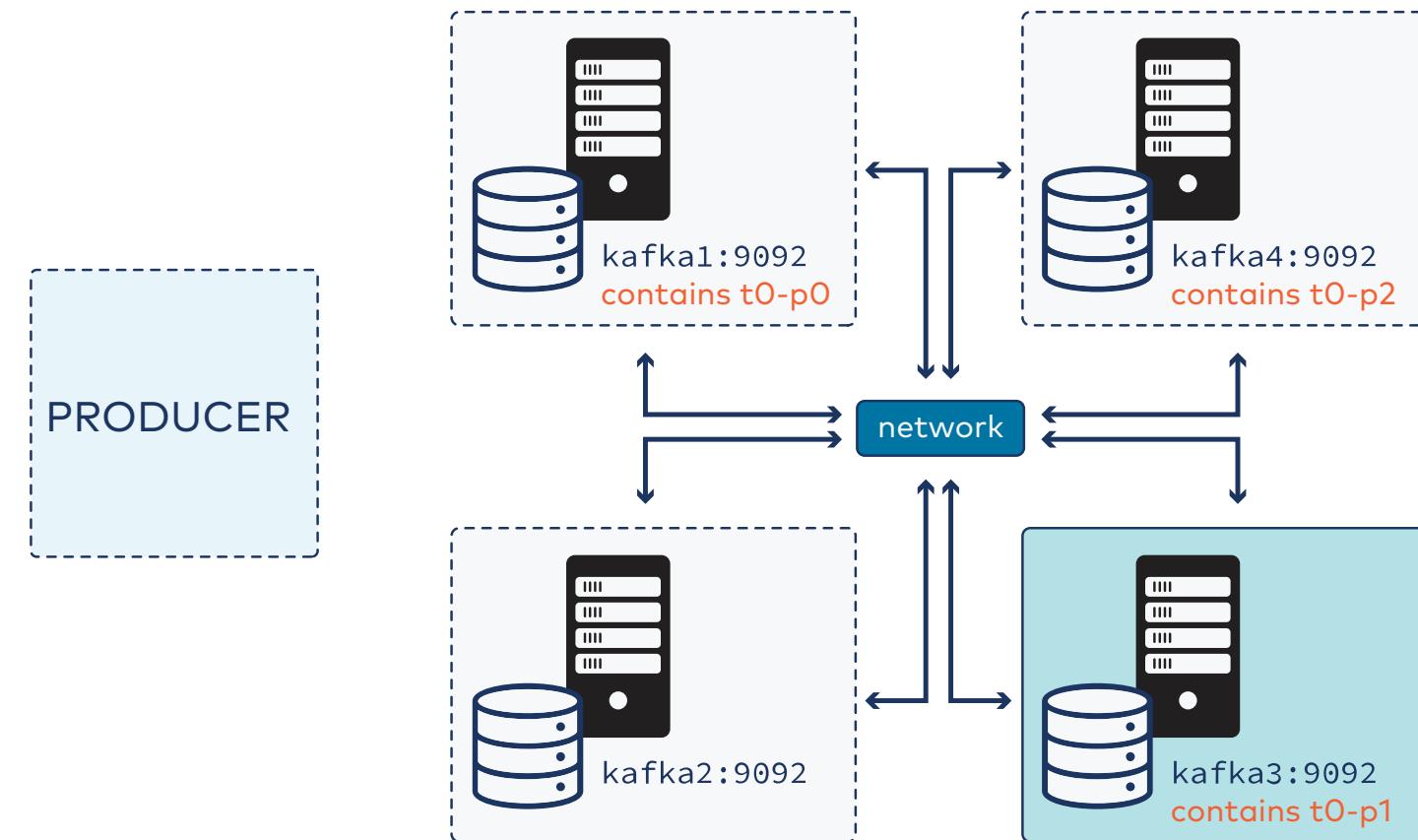
```
bootstrap.servers = kafka:9092
```

Bootstrap Servers in Practice (2)

In practice, specify a comma-separated list of bootstrap servers:

```
props.put("bootstrap.servers", "kafka1:9092, kafka2:9092, kafka3:9092");
```

Why?

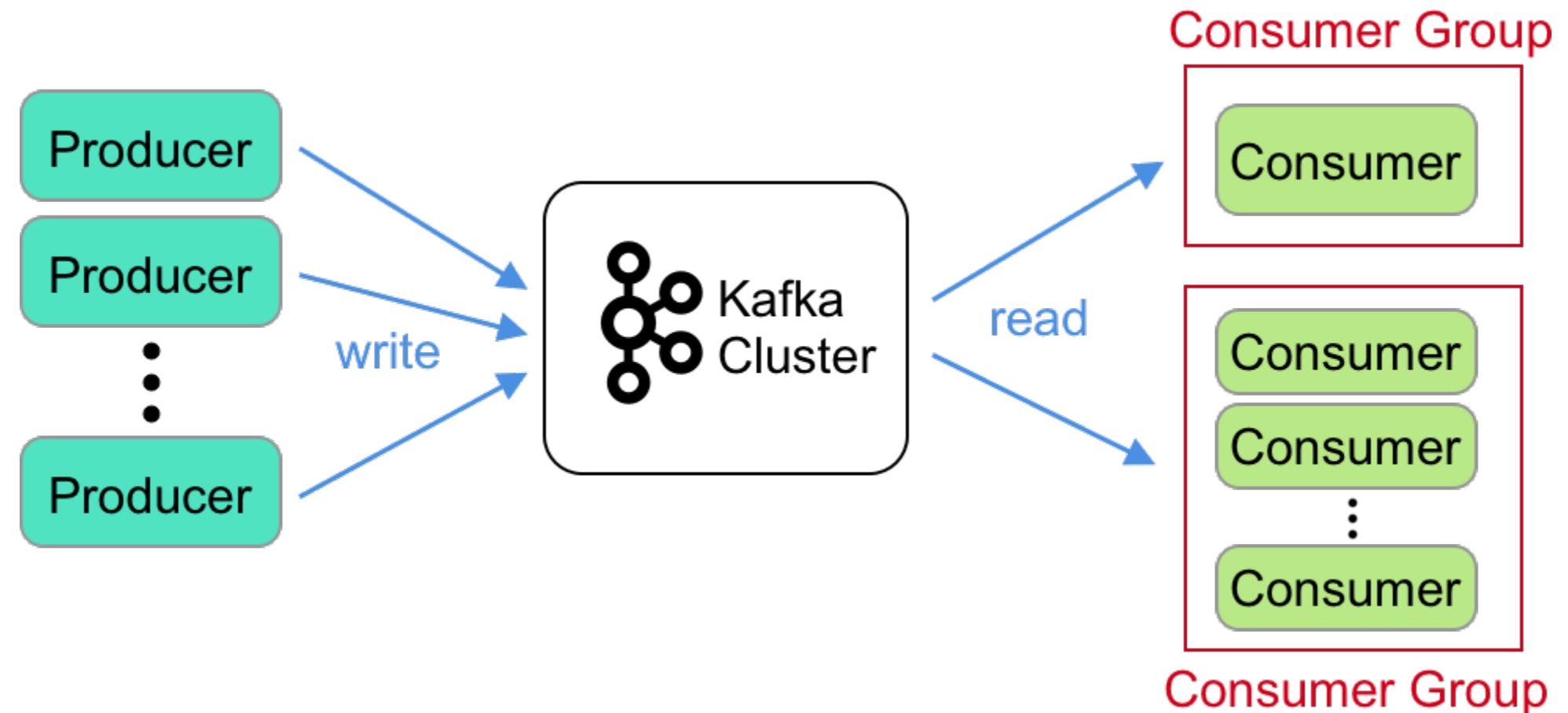


Appendix: Fundamentals Review

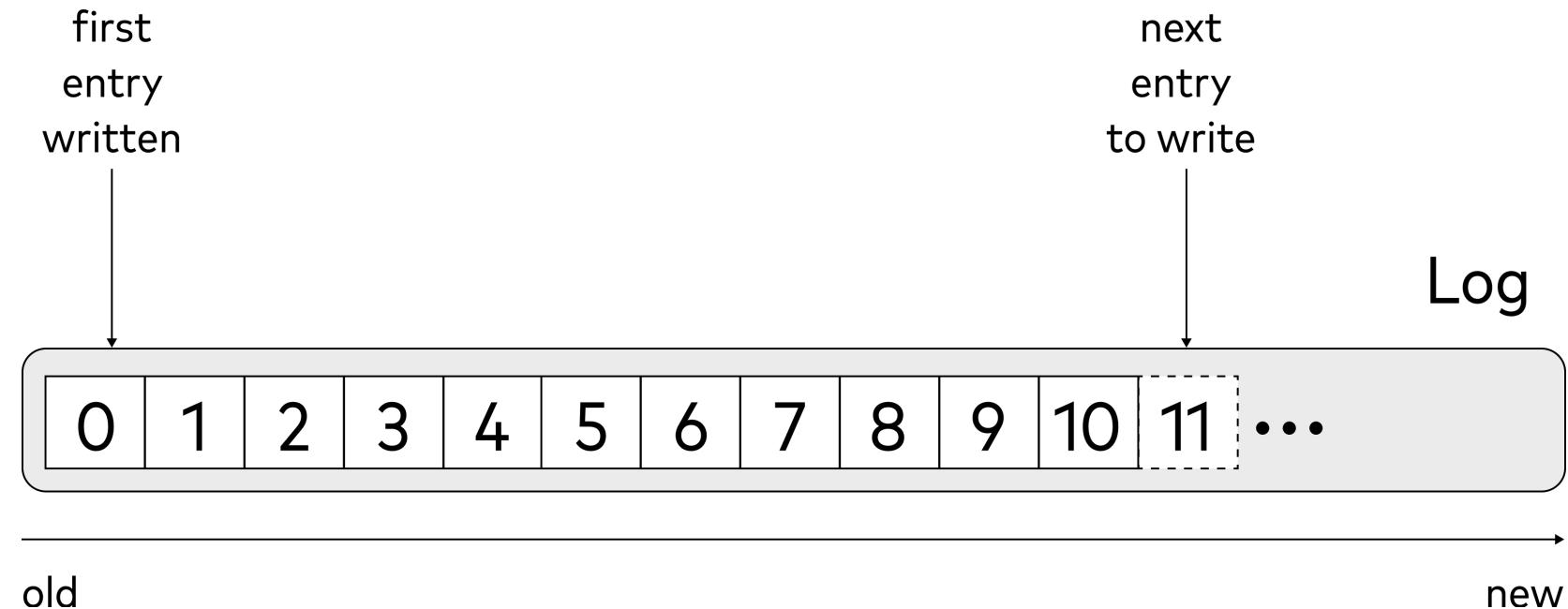
Description

Review of Fundamentals.

Kafka Clients - Producers & Consumers

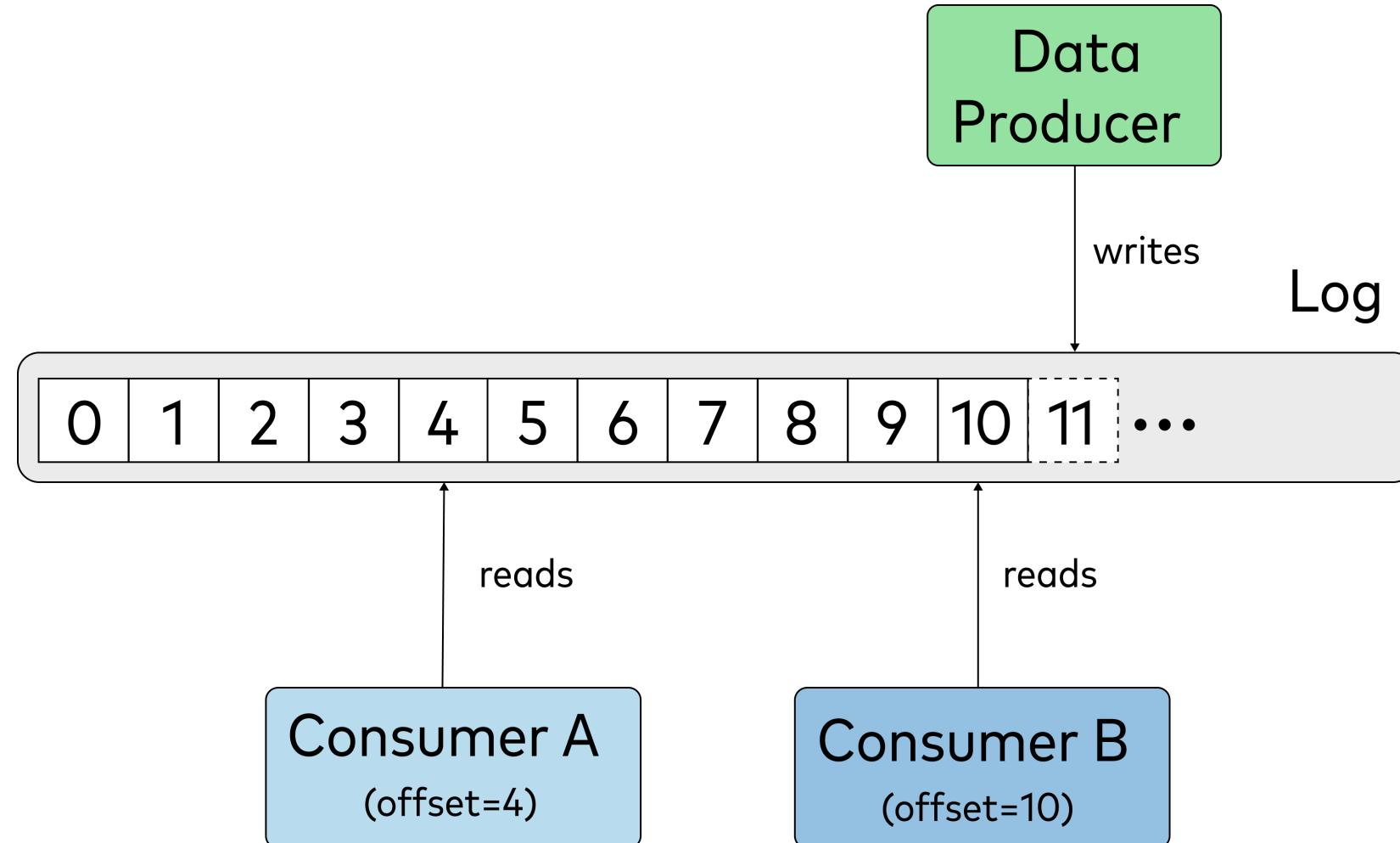


The Kafka Commit Log

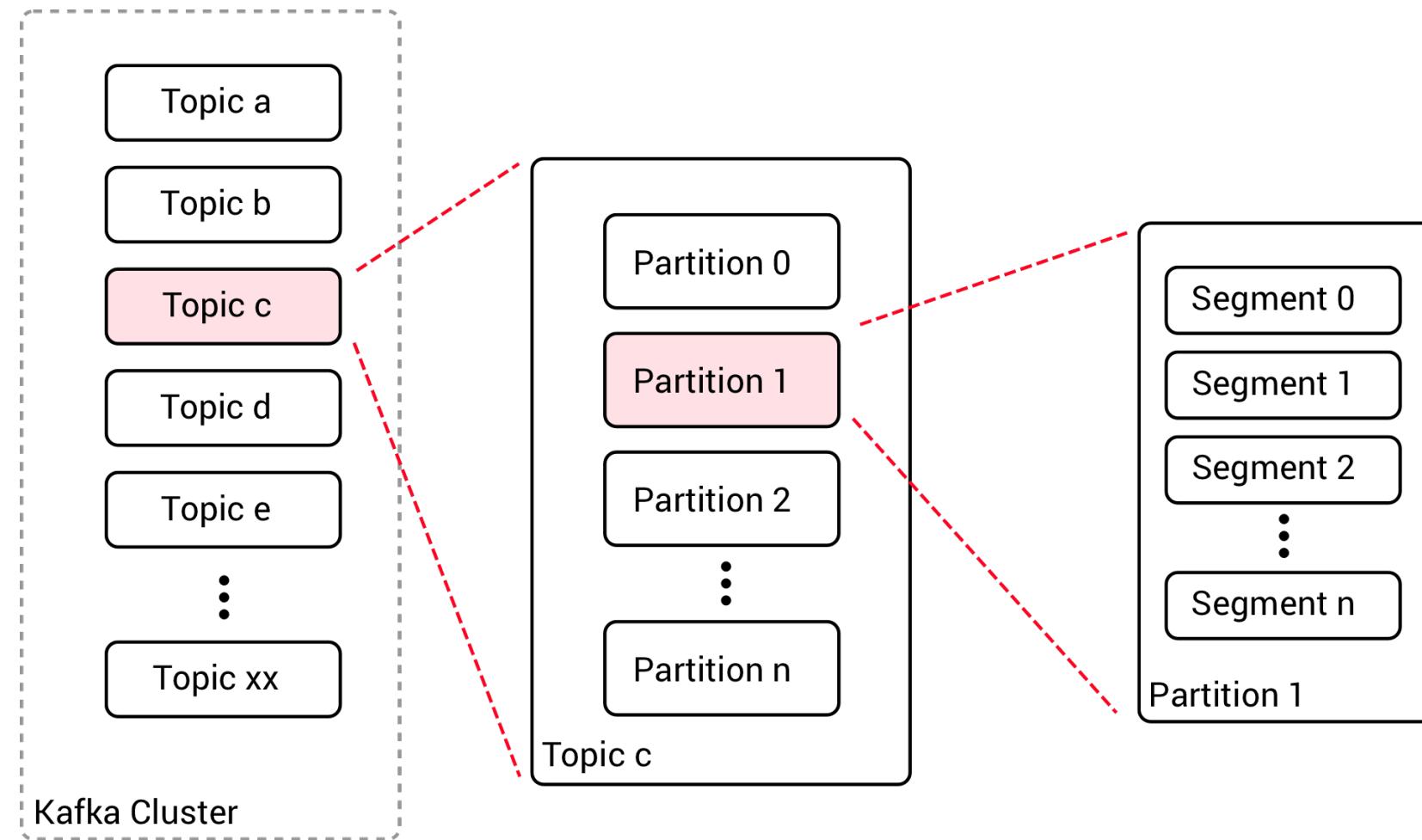


A position in the log is called an "offset." Here we see offsets 0 through 11.

Decoupling Data Producers from Data Consumers

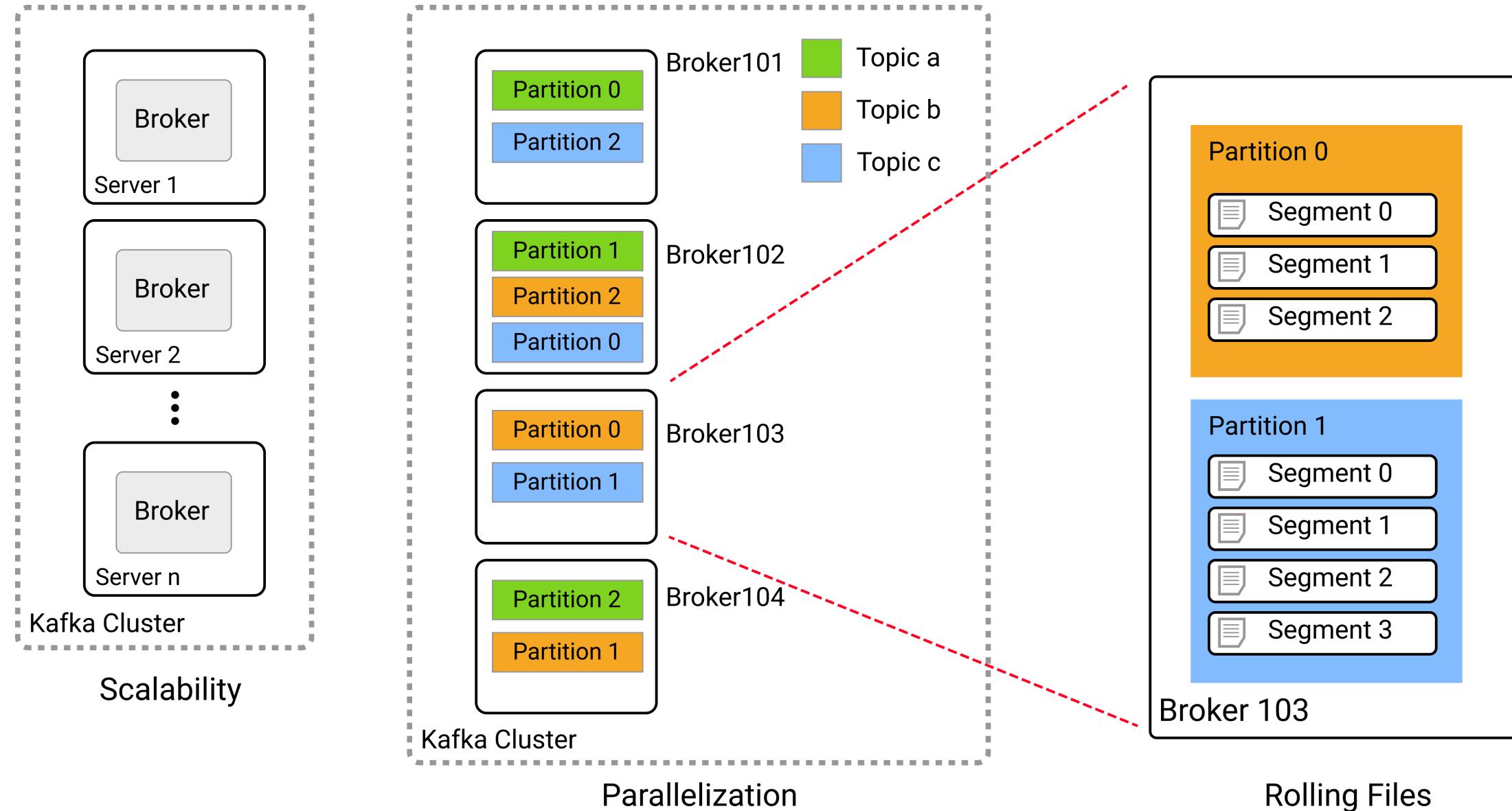


Kafka—Logical View of Topics, Partitions, and Segments

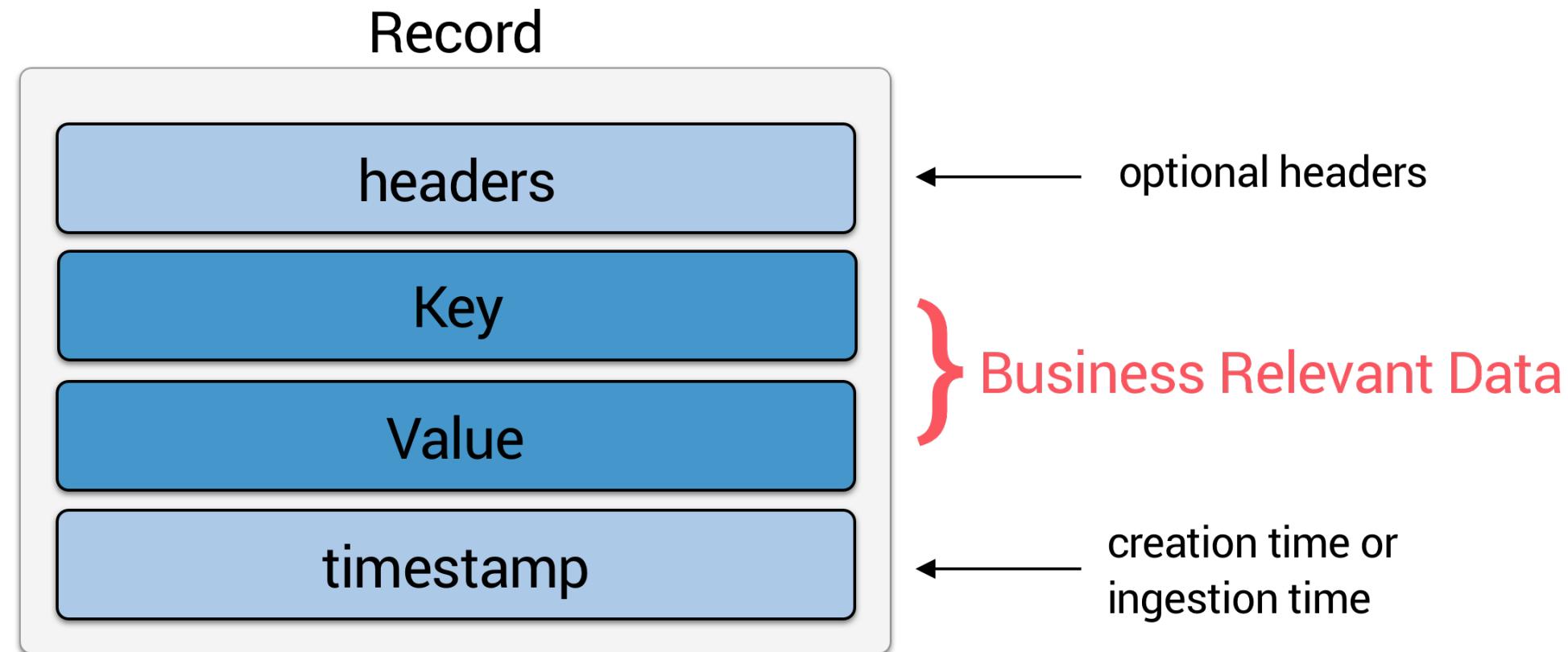


Each partition is a commit log. We often use "partition" and "log" interchangeably.

Kafka—Physical View of Topics, Partitions and Segments



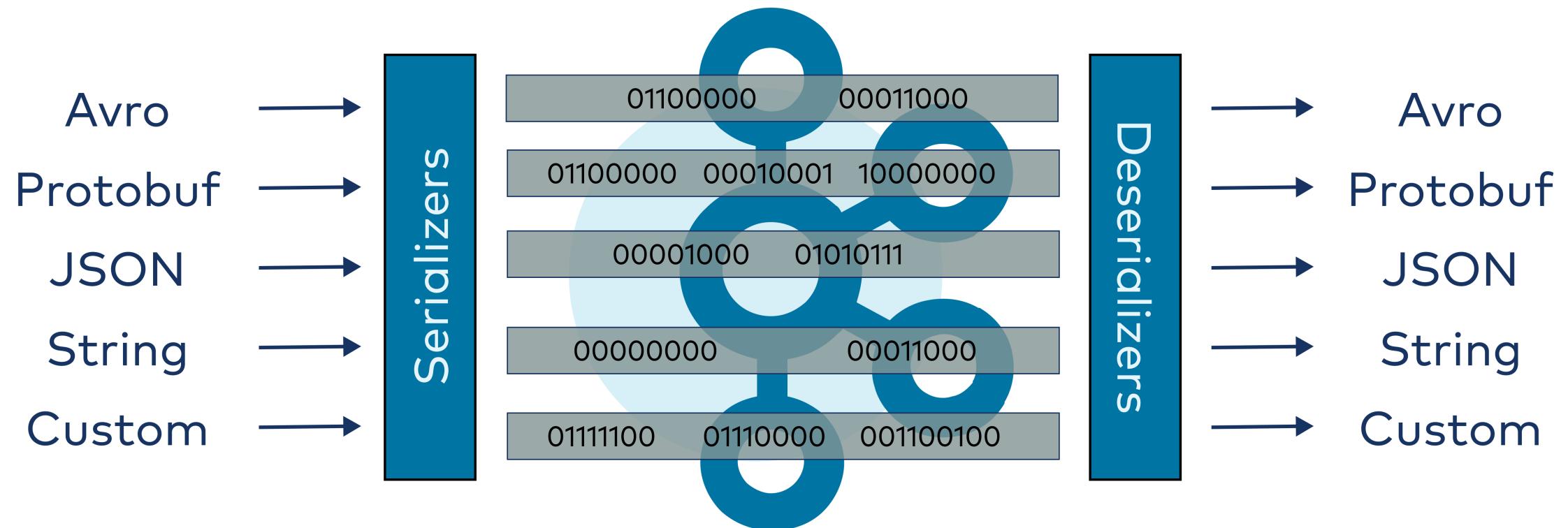
The Record—The Atomic Unit of Kafka



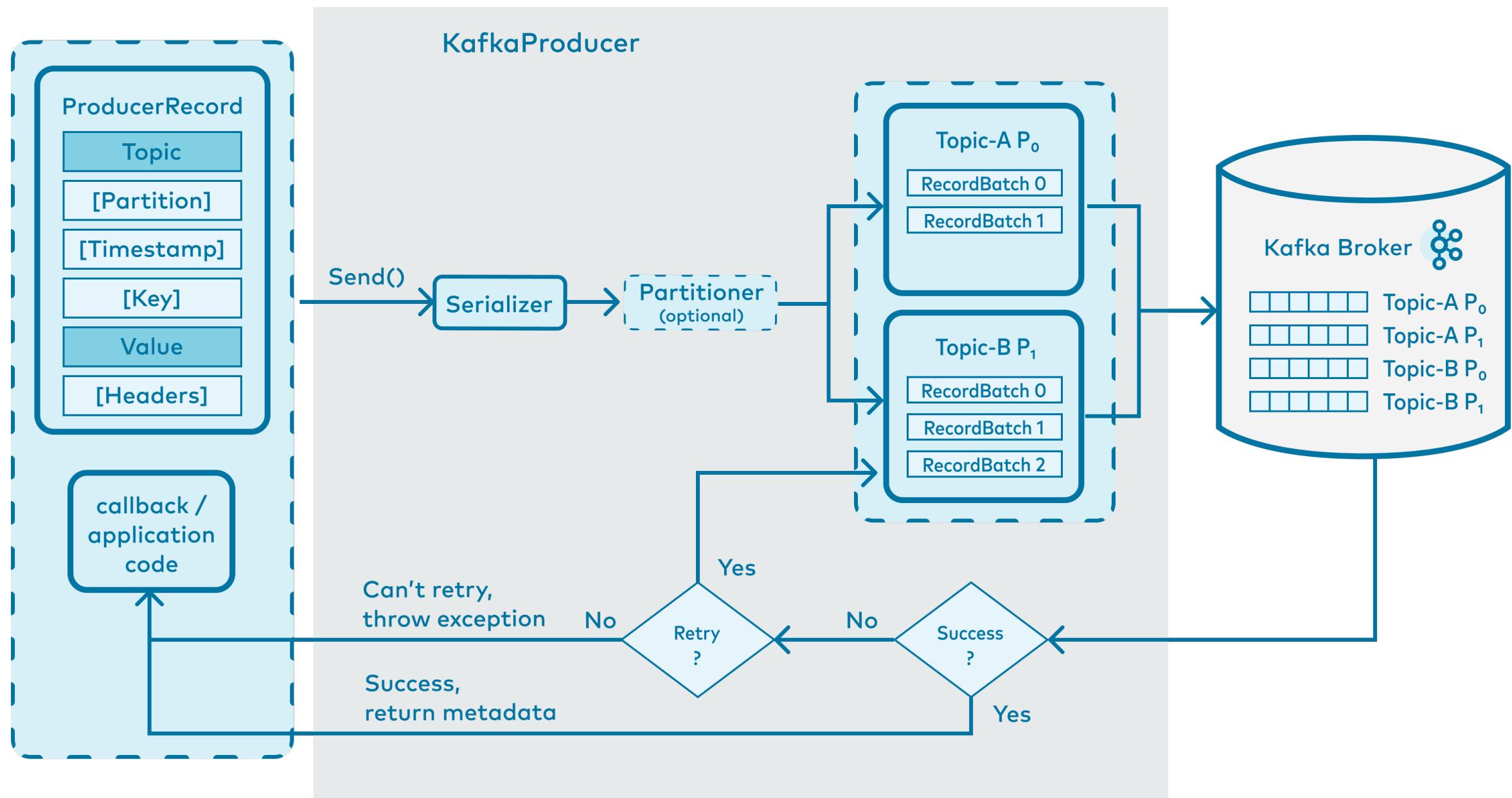
Kafka Records are also known as "Messages" or "Events"

Serialization

- Kafka stores byte arrays

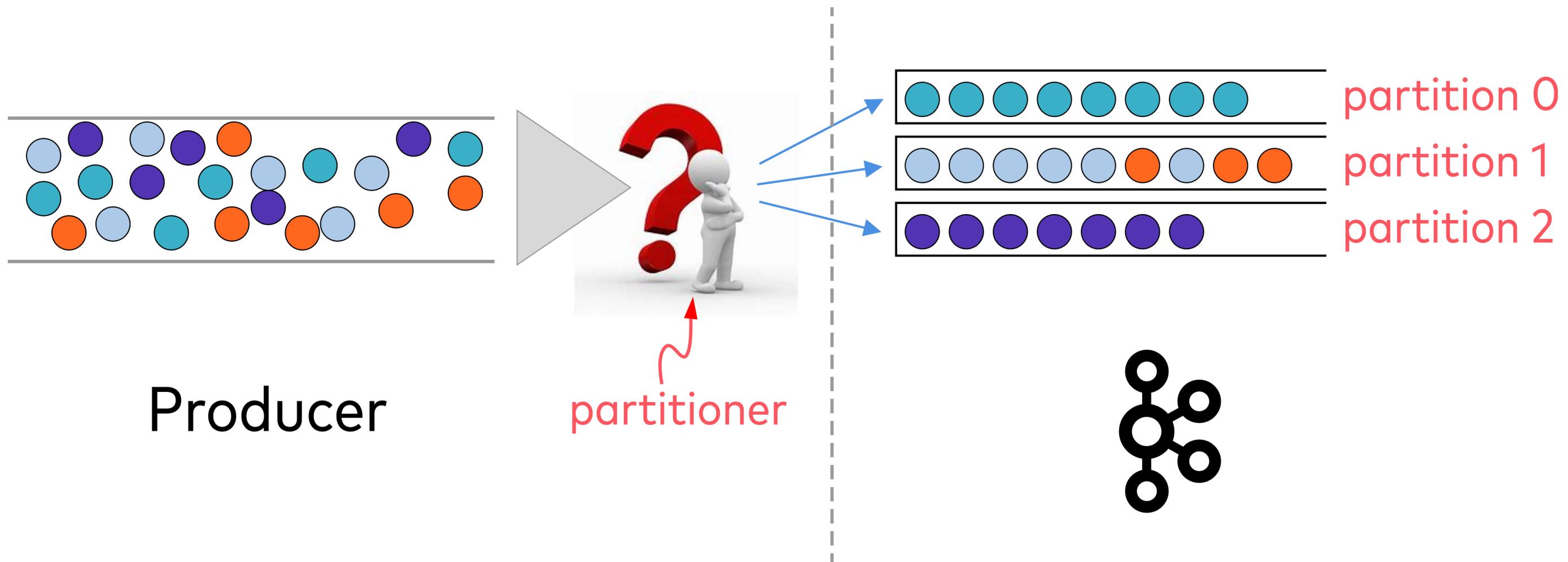


Producer Design



Partitioning

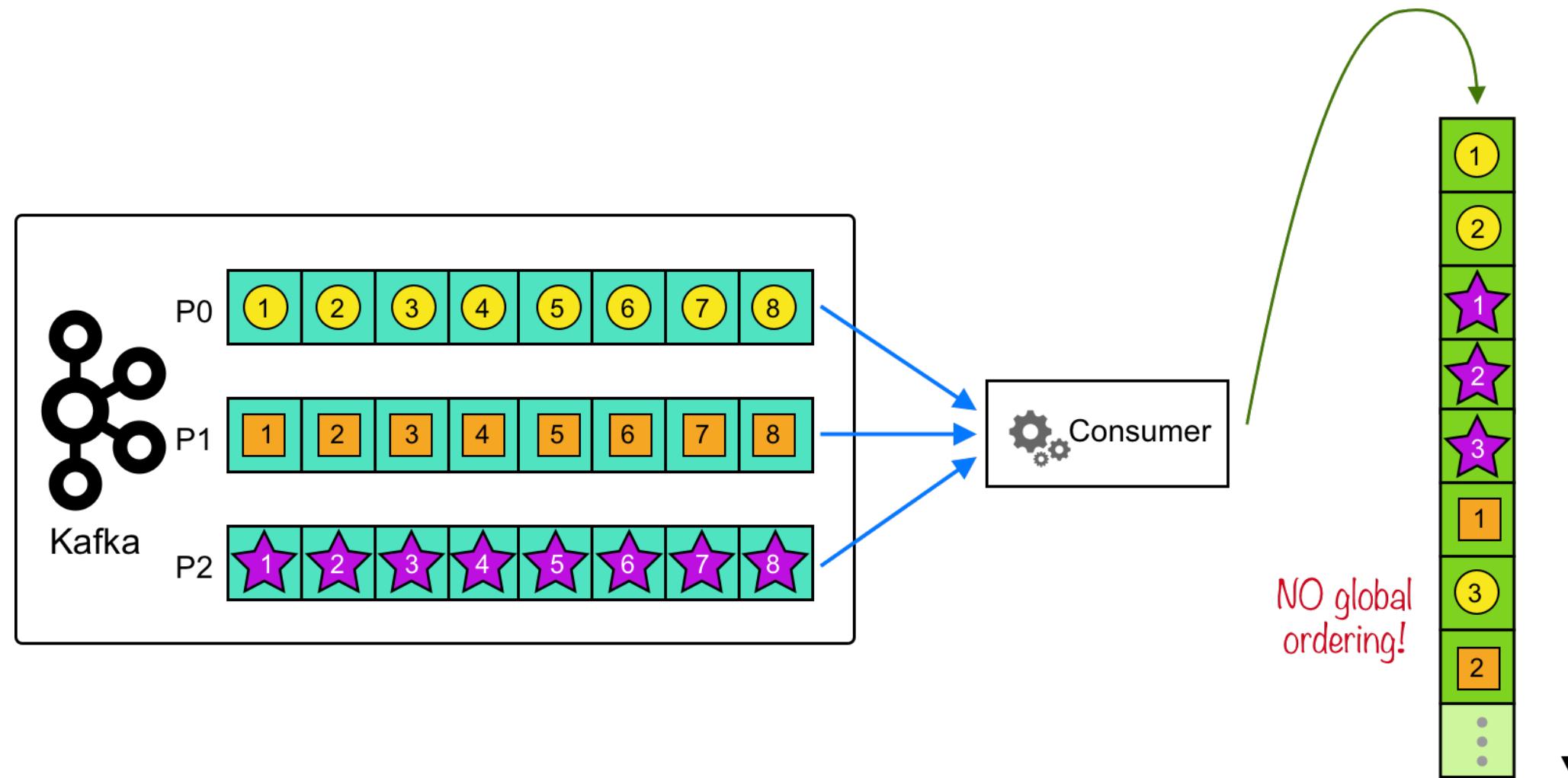
$\text{Partition} = \text{hash(key)} \% \text{Number of Partitions}$



Preserve Message Ordering

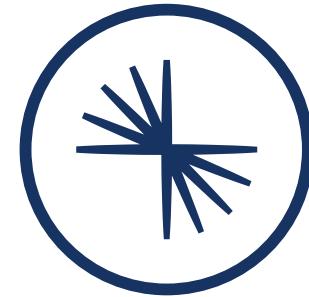
- Messages with the same key, from the same Producer, are delivered to the Consumer in order
 - Kafka hashes the key and uses the result to map the message to a specific Partition
 - Data within a Partition is stored in the order in which it is written
 - Therefore, data read from a Partition is read in order *for that partition*
- If the key is null and the default Partitioner is used, the record is sent to a random Partition

An Important Note About Ordering



- **Question:** How can you preserve message order if the application requires it?

Appendix: Some Alternative Slides



CONFIDENT
Global Education

Overview

This appendix, appearing only in the slide deck, contains a few alternate slides that build up to concepts otherwise presented all at once.

Alt Slides: How Does a Produce Request Get Processed on a Broker?

Description

Thread pools. Queues. Purgatory. The path of a produce request from broker receipt to acknowledgement.

Review

We know:

- Producers prepare messages to send.
- Producer settings like `linger.ms` and `batch.size` control how messages get grouped in batches to send.
- Batches of messages may be compressed, according to producer setting `compression.type`.
- We can configure producer property `acks` to have producers request to hear back from Kafka when messages are successfully written.

When a producer sends a batch, we say it is creating a **produce request**. So...

- What does a broker do when it receives a produce request?
- How exactly does a broker satisfy a producer's `acks` request?

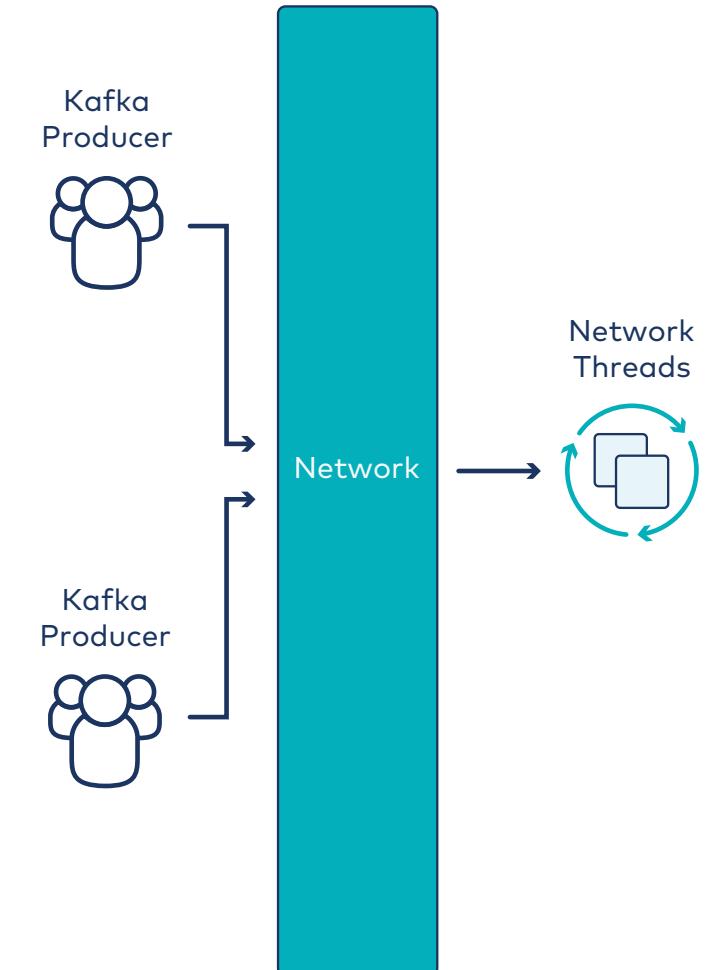
Let's find out...

Network Threads Receive Requests

Brokers have **network threads** to receive incoming requests.



- The number of threads is configurable in broker setting `num.network.threads`.
- Default: 3.



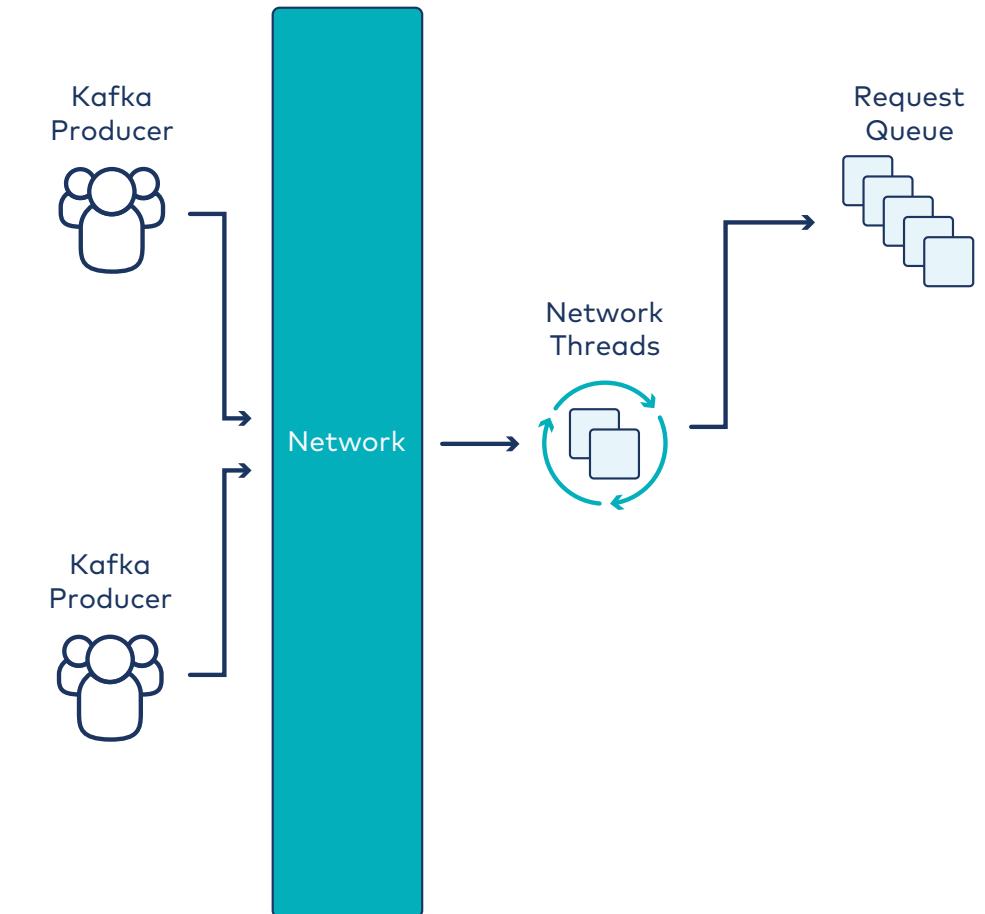
Where Do Requests Go?

The broker may not be able to process requests the moment they are received.

Network threads write requests into a **request queue**.

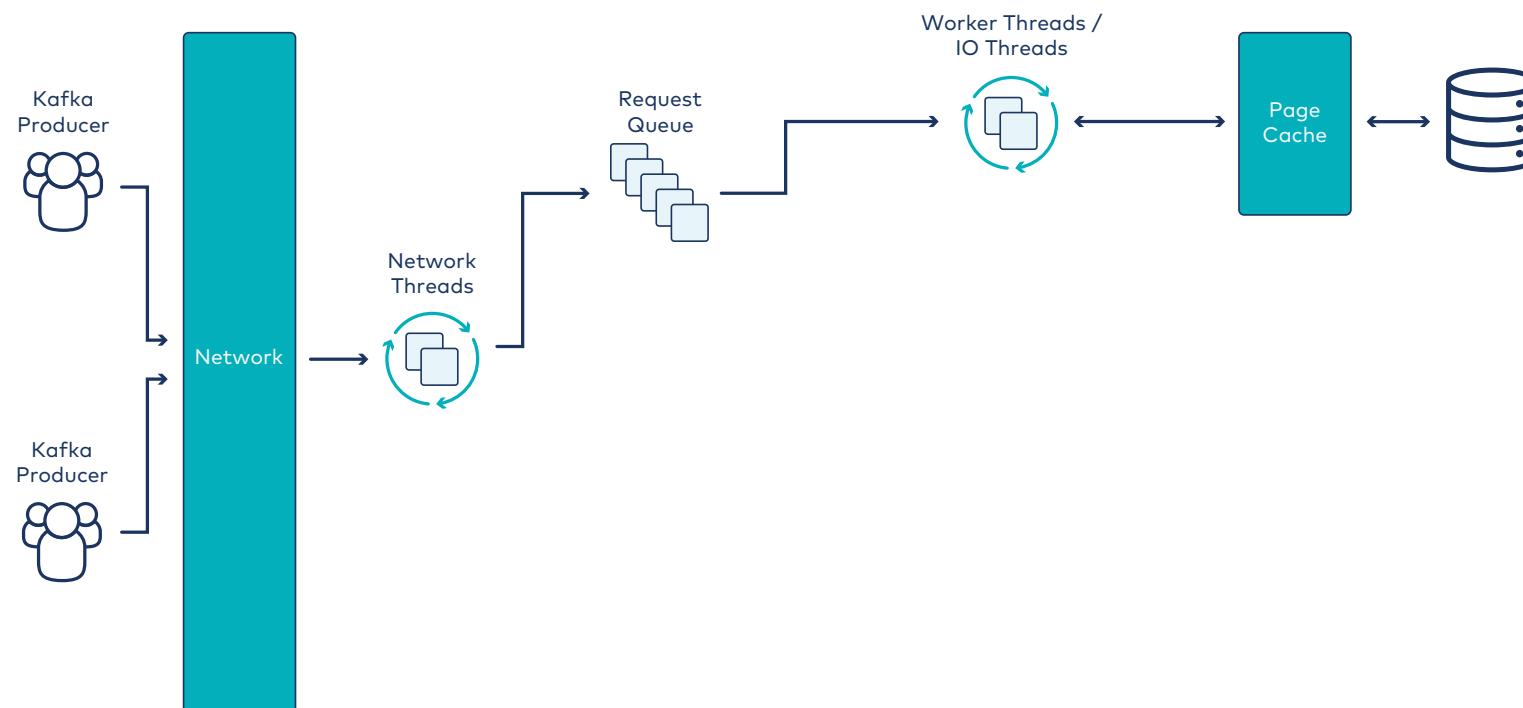


- The request queue size is `queued.max.requests`.
- Default: 500.



Worker Threads Process the Request Queue...

Another thread pool, **worker threads** or **IO threads**, contains threads to process the request queue and write messages to the page cache.



Configurable in `num.io.threads`. Default 8.

Are We Done?

Maybe.

Under what condition?

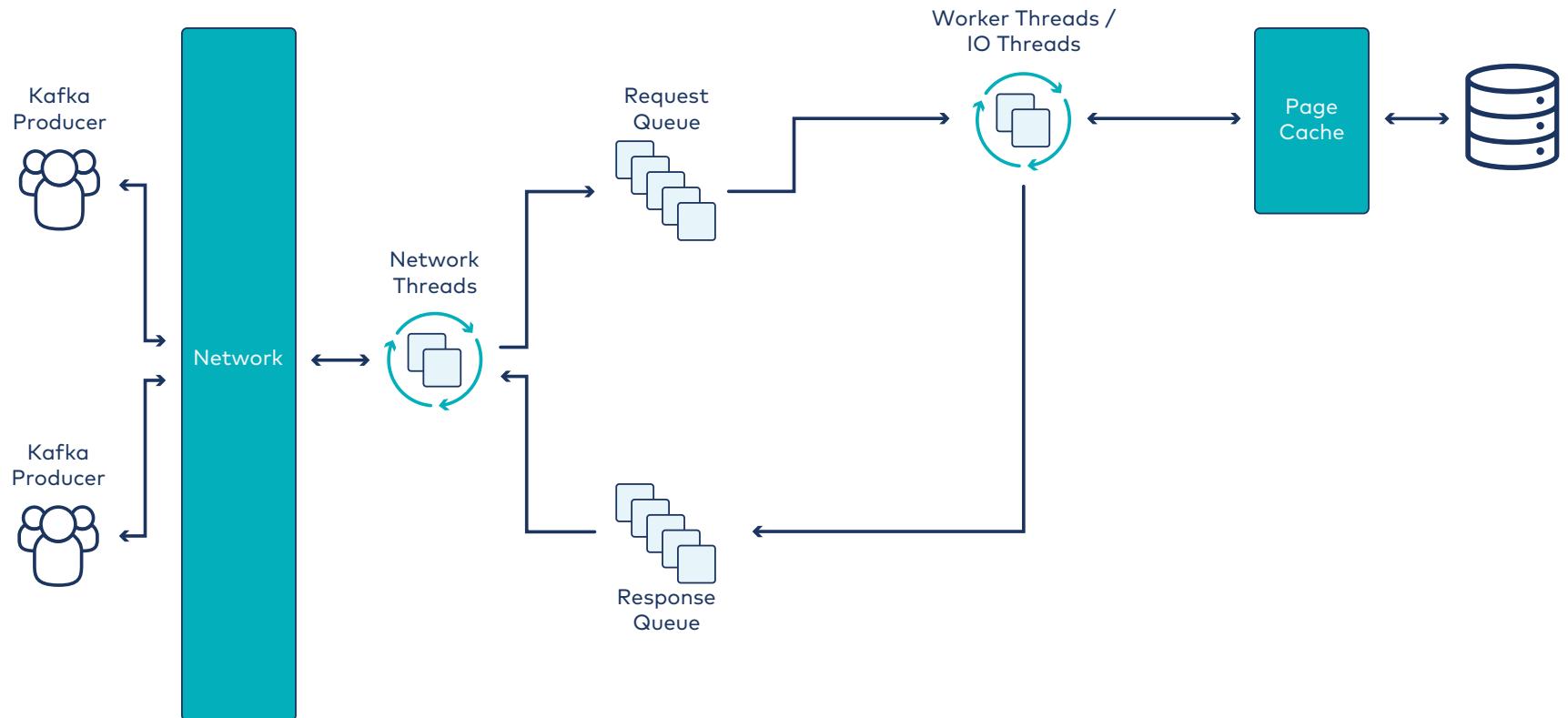
What else might matter?



What About Acknowledgements?

Let's start with `acks = 1`.

We now add a **response queue**.



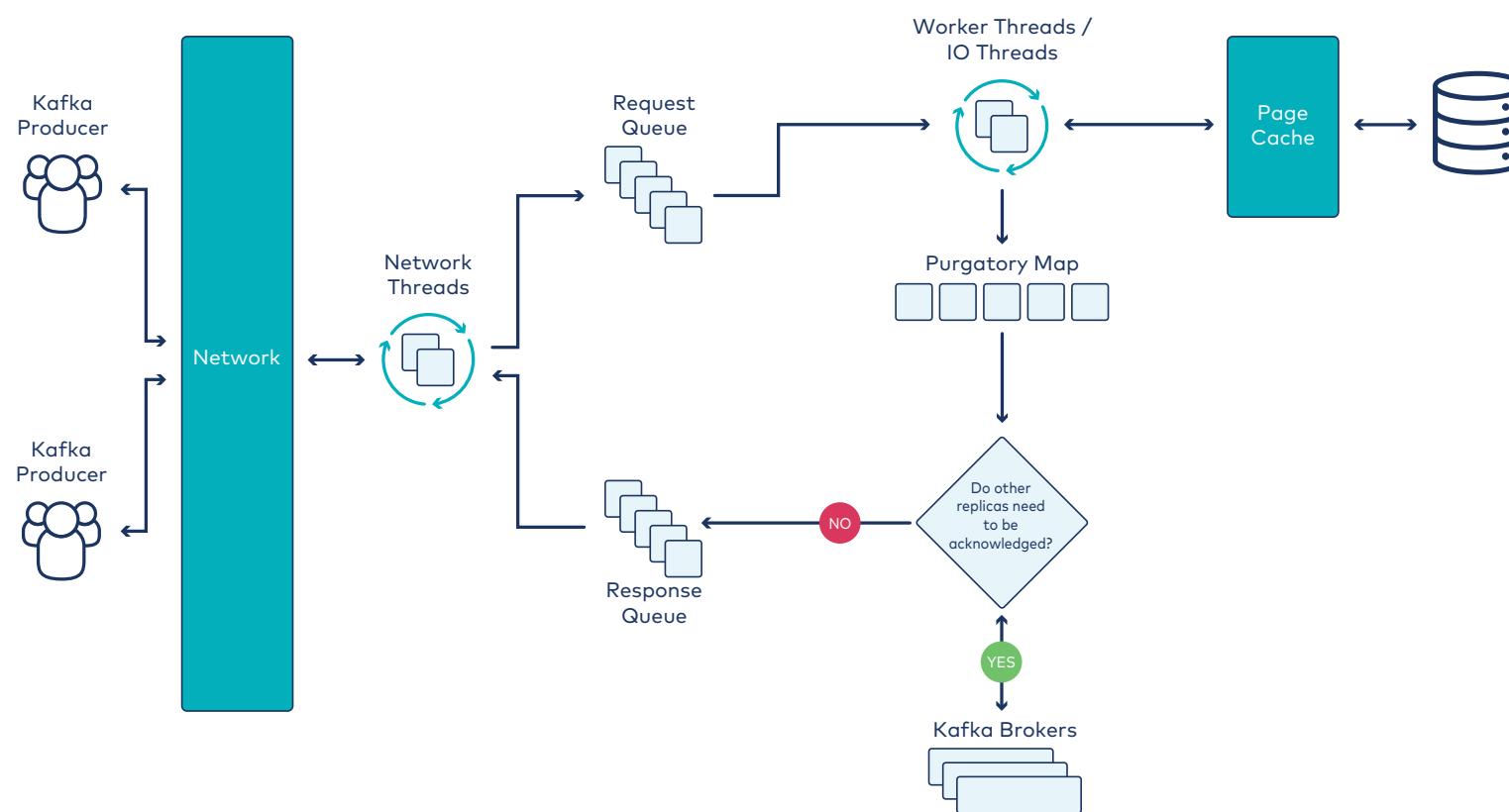
Network threads handle the response queue and the request queue.
Notice some arrows are now two-way.

Handling Replication and **acks = all**: Purgatory

Purgatory

Structure in memory for holding produce requests that are not yet complete

Complete version of the diagram:



Activity



Discuss:

1. You might hear someone say or read that one should pay attention to garbage collection on Kafka brokers. Why do you think this is a concern?
2. Study the anatomy of a produce request firugre on the previous page. Where do you see there being potential problems/bottlenecks?

Alt Slides: How Does a Fetch Request Get Processed on a Broker?

Description

Extending the anatomy of produce requests on a broker to fetch requests.

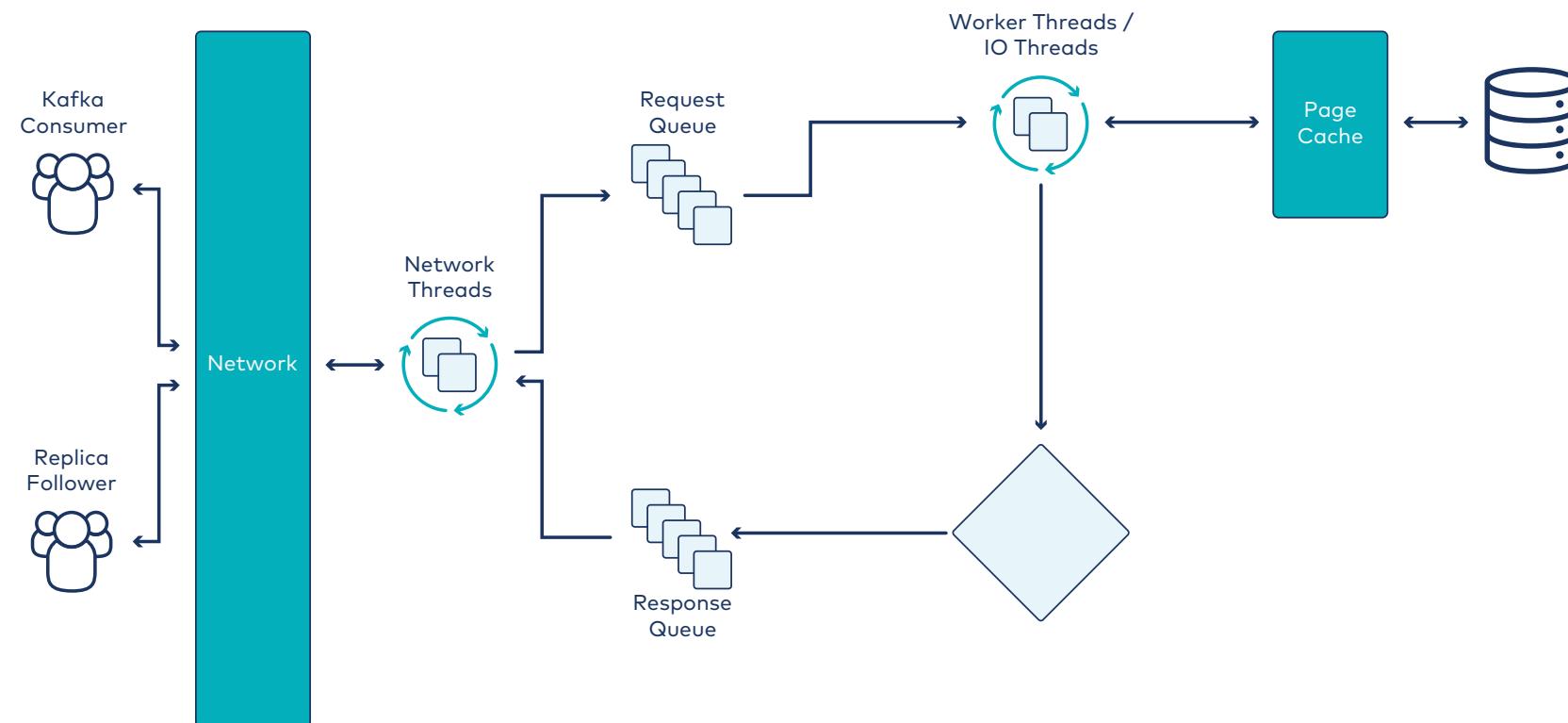
Questions & Segue

We've seen how brokers handle produce requests. Now, what about fetch requests?

Question: What Kafka entities would make a fetch request?

A Familiar Picture

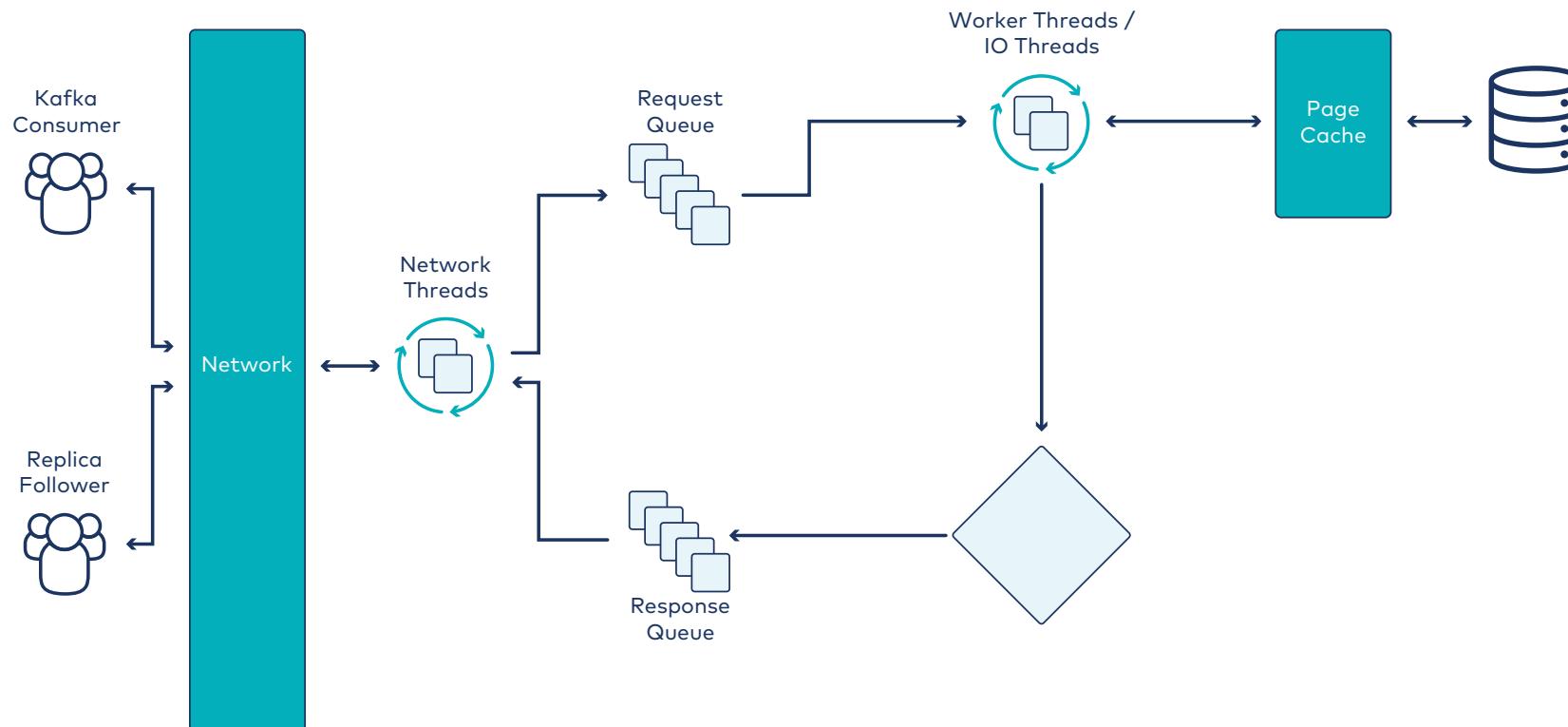
Let's start with a picture very similar to what we saw for produce requests...



The queues and thread pools here are exactly the same as we saw before.

What's Missing?

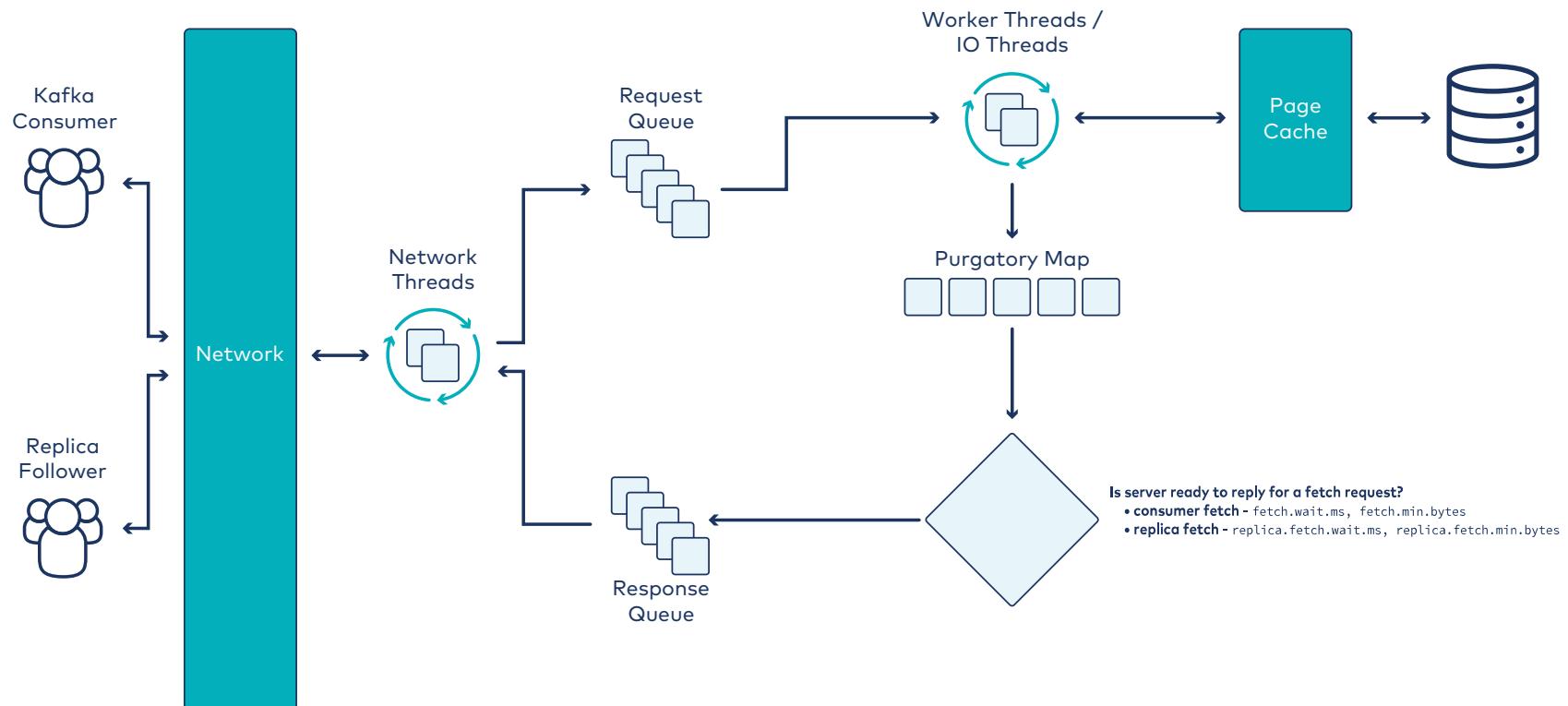
We just saw this picture:



Question: What's missing?

Fetch Purgatory

We need a purgatory for fetch requests as well...



The produce purgatory and fetch purgatory are separate.