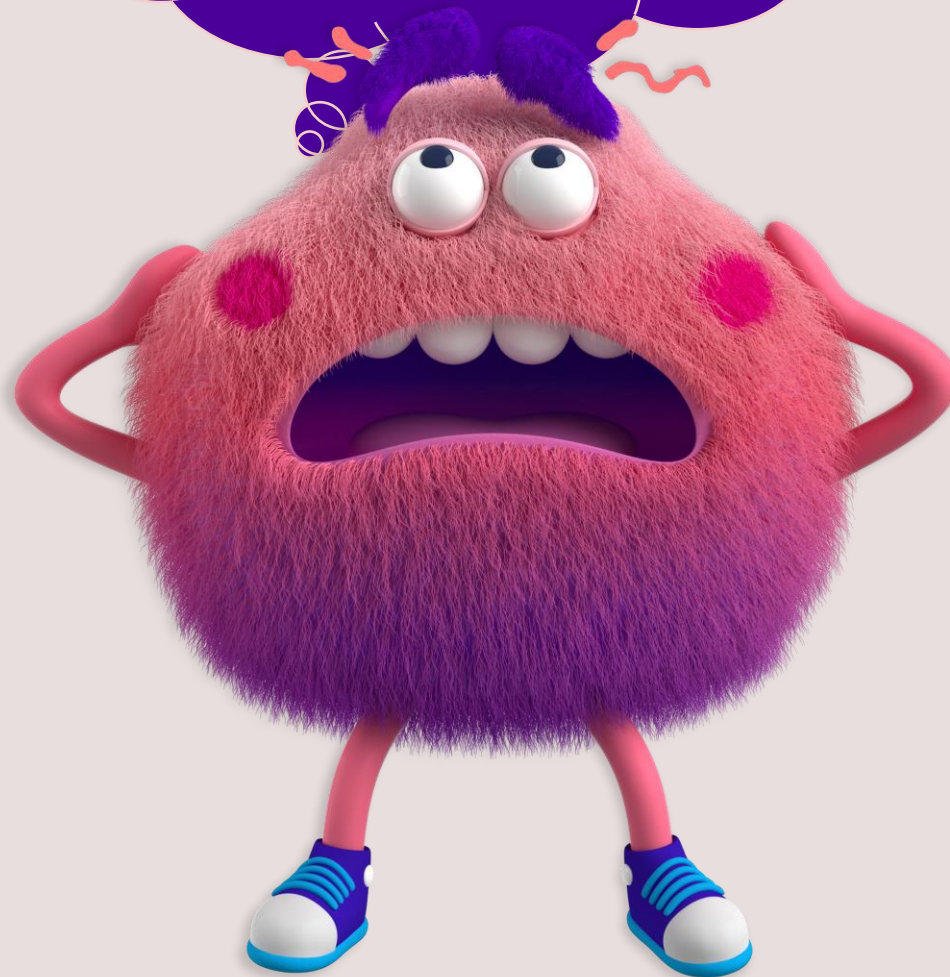


Kafka





Apache Kafka is an **open source**
distributed streaming platform
designed for :

- building real-time
- data pipelines
- streaming applications

Kafka introduction

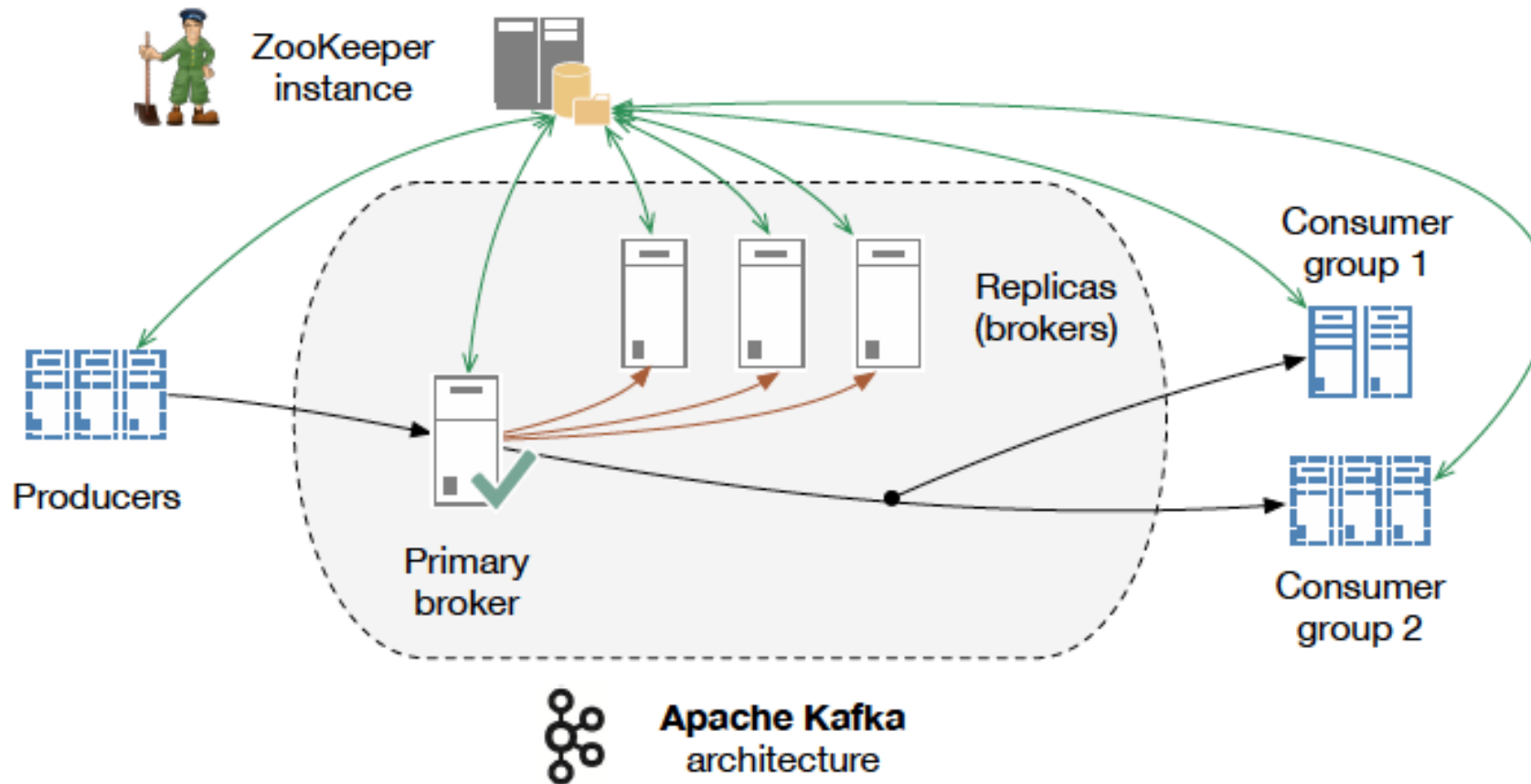
Apache kafka is a **distributed publish-subscribe messaging system**

It operates on a pub-sub model, where producers publish messages and consumers subscribe to topics to receive these messages.

Multiple consumers from a Consumer Group subscribe to topics allowing for load-balancing and parallel data consumption.

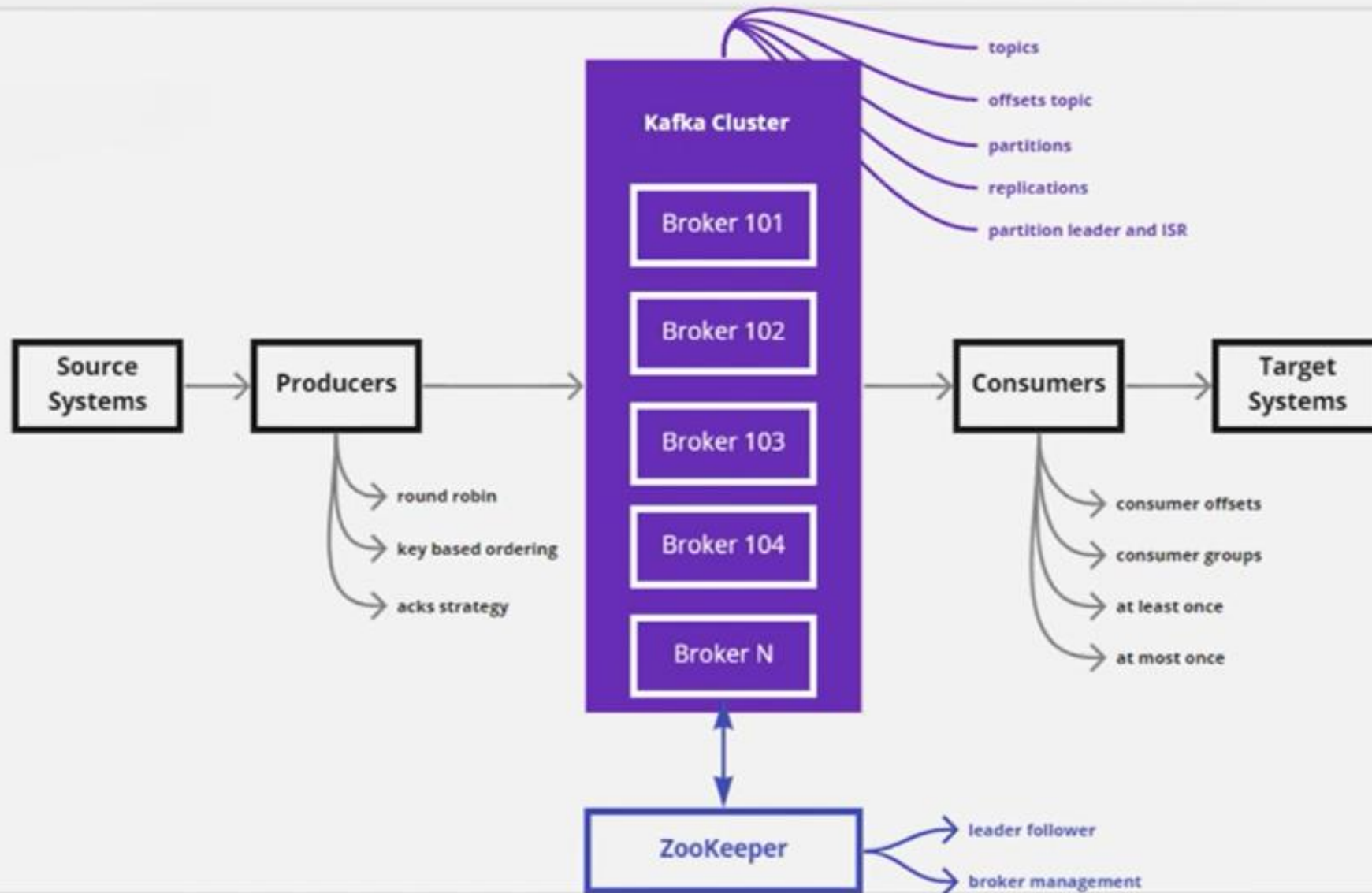
Kafka introduction

Apache Kafka که اولین بار برای linkedin پیاده‌سازی و در سال ۲۰۱۱ به صورت open source ارائه شد، امروزه بزرگترین data pipeline ها در جهان از کافکا استفاده می کنند. شکل زیر معماری این نرم‌افزار را نشان می‌دهد:



این نرم‌افزار که با زبان جاوا و اسکالا نوشته شده است دارای تکنولوژی فوق‌العاده‌ای است

Concepts in Kafka



Kafka Features

distributed log-based model

Fast

scalable

durable

fault-tolerant

supports automatic recovery

Dumb broker / smart consumer model

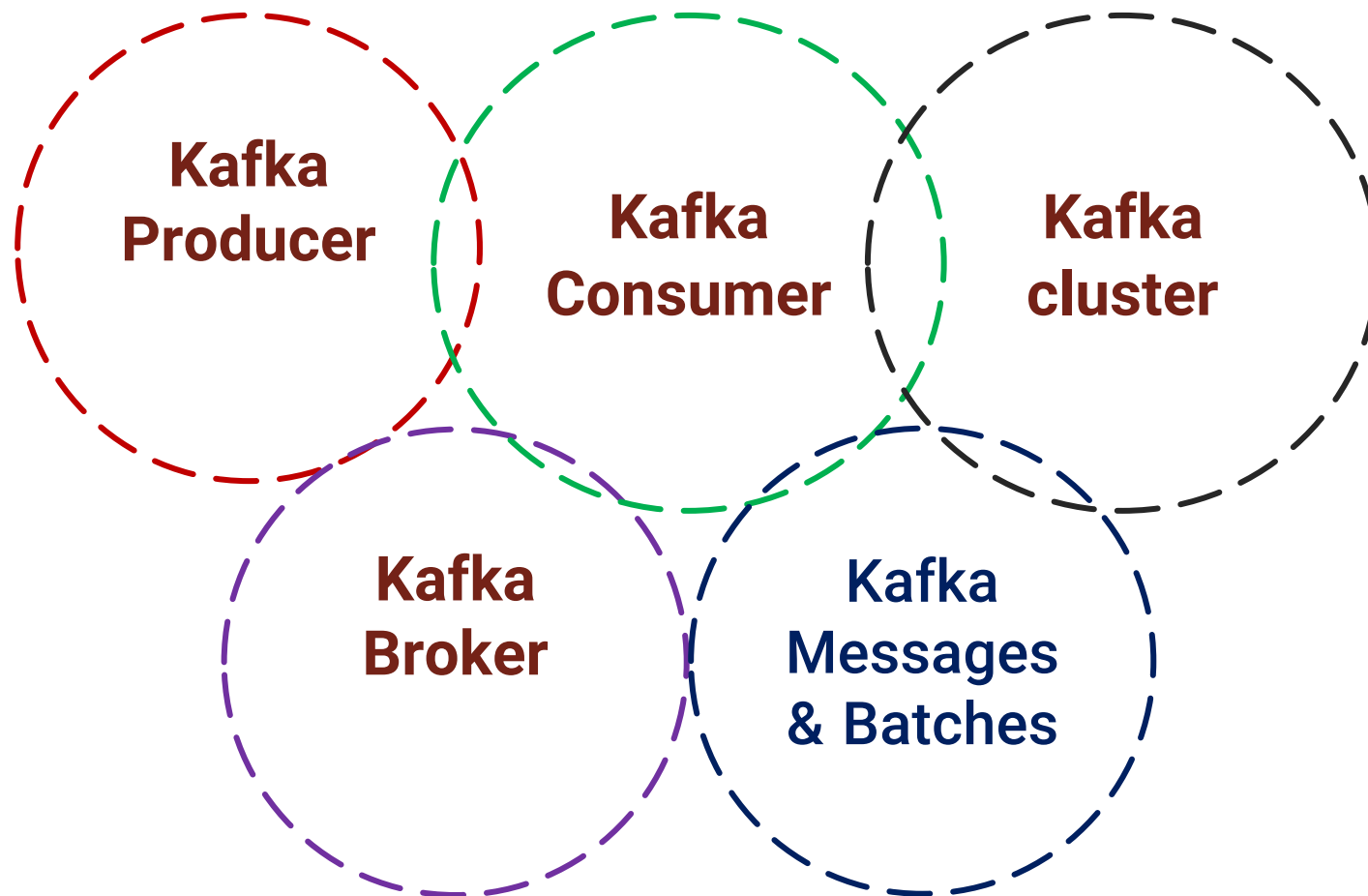
Security

It offers durable storage with a write-ahead log, which ensures that data is not lost if a server crashes

Kafka supports SSL and Kerberos for secure communication

Kafka به طور ذاتی distributed طراحی شده است تا با توزیع داده ها در چندین گره، مقیاس پذیری و تحمل خطا (fault tolerance) را فراهم کند.

عملکرد کلی Kafka شامل مفاهیمی از قبیل:



Kafka Messages & Batches

واحد اصلی داده در کافکا یک پیام (Message) است

A Message is like a record in a DB table

به صورت آرایه ای از byte ها ارسال می شود و پیام ها را می توان فشرده (compress) کرد. کافکا از فشرده سازی پیام برای کاهش ترافیک شبکه پشتیبانی می کند.

Messages are grouped into batches

کافکا از یک پروتکل binary TCP-based استفاده می کند که برای کارایی بهینه شده است و متکی بر یک انتزاع "message set" متکی است که به طور طبیعی پیام ها را گروه بندی می کند تا سربار رفت و برگشت شبکه را کاهش دهد.

رویکرد دسته ای = افزایش توان عملیاتی

لایه ذخیره سازی کافکا با استفاده از **partitioned transaction log** پیاده سازی شده

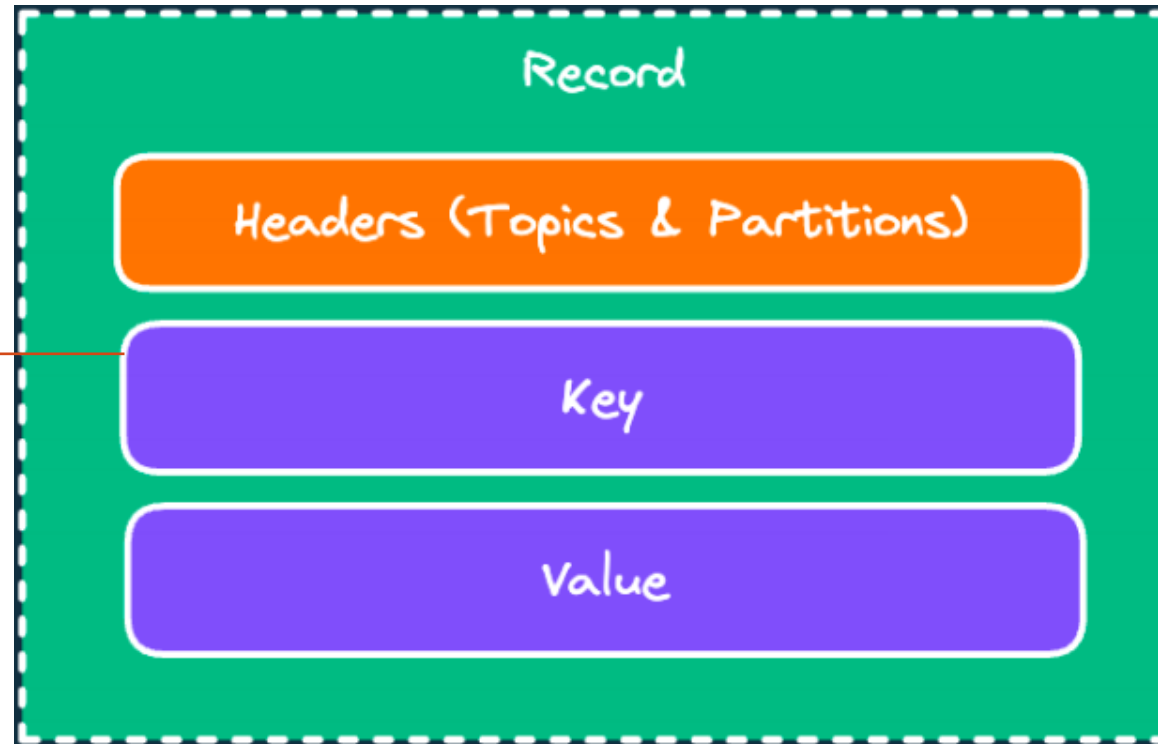
Once a batch is ready, it's sent to a **Topic & Partition**

Kafka Messages & Batches

کافکا جریان داده ها و رکورد ها را در ساختارهایی به نام topic ذخیره می کند.

هر رکوردی (Message) دارای یک **کلید**، یک **مقدار** و یک **برچسب زمانی** می باشد تا بصورت مجزا از سایر رکورد ها مشخص باشد.

Type key: int, string ,
...



KAFKA TOPICS & PARTITIONS

Topics مانند یک جدول DB یا یک پوشه هستند

هر پیامی (Message) به Topic خاصی می رود

Topics نیز از چند پارتیشن (partition) تشکیل شده اند

هر پارتیشن از یک سری پیام‌های **متوالی مرتب سازی شده**، **غیر قابل تغییر** (زمانی که دیتا رو داخل آن می نویسیم غیر قابل تغییر) ، تشکیل یافته که هر پیامی بصورت پیوسته بهش الحاق میشه.

پارتیشن ها (partitions) به معنای افزونگی (redundancy) هستند و Topic را به صورت افقی (horizontally) مقیاس پذیر می کنند

پیام ها توسط producer (تولید کننده پیام) به یک پارتیشن (partition) append می شوند ، یعنی به انتهای لیست می رود.

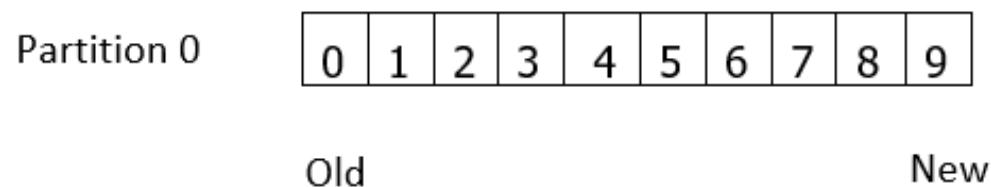
نکته ای که باید بهش توجه کرد اینه که Kafka بدون در نظر گرفتن اینکه آیا پیام ها consume شدن یا نه اونارو تا یک مدت زمانی (به صورت پیش فرض یک هفته) نگه میداره.

کافکا بصورت پیش فرض به منظور **توزیع پیامها بصورت یکنواخت** بین پارتیشن ها از **round-robin** **partitioner** استفاده میکنه.

KAFKA TOPICS & PARTITIONS

گفتیم که هر Topic از یک یا چندین Partition برای ذخیره داده‌ها استفاده می‌کند. تعریف درست تعداد Partition‌ها در یک Topic، تاثیر مستقیمی بر **درجه همزمانی و کارایی** در آن Topic و کل سیستم دارد. در Kafka تمامی پیام‌ها به همان ترتیبی که وارد شده‌اند، در Partition‌های یک Topic ذخیره می‌شوند و به همان **ترتیب** نیز توسط **Consumer** ها **pull** می‌شوند. **یعنی کافکا ترتیب پیام را در یک پارتیشن تضمین می‌کند، اما نه در بین پارتیشن‌های یک موضوع.**

بطور مثال فرض کنید تعداد Partition های Topic ، یک می باشد در این صورت تمامی پیامهای دریافتی تنها در یک Partition ذخیره می شوند.



هر خانه از یک Partition، توسط یک شناسه از نوع **int** و با نام **offset** در دسترس است. تمامی پیام‌های جدید ارسالی توسط Producer با **offset** بزرگتر از **offset** موجود در این Partition ذخیره می‌شوند؛ یعنی در انتهای آن قرار می‌گیرند. در مثال فوق در صورت دریافت پیام جدید، **offset** آن با عدد 10 مقداردهی می‌شود. همچنین عملیات خواندن نیز از **کوچکترین offset** که هنوز مقدار آن توسط **Consumer** خوانده نشده است، انجام می‌شود. همانطور که مشخص است، بدلیل اینکه تعداد Partition‌های این مثال عدد یک می‌باشد، تمامی درخواست‌های Producer‌ها در یک Partition قرار می‌گیرند و تمامی Consumer‌ها نیز از طریق یک Partition به پیام‌ها دسترسی دارند؛ یعنی در صورت بالا بردن تعداد Producer‌ها یا Consumer‌ها، کارآیی بالا نمی‌رود.

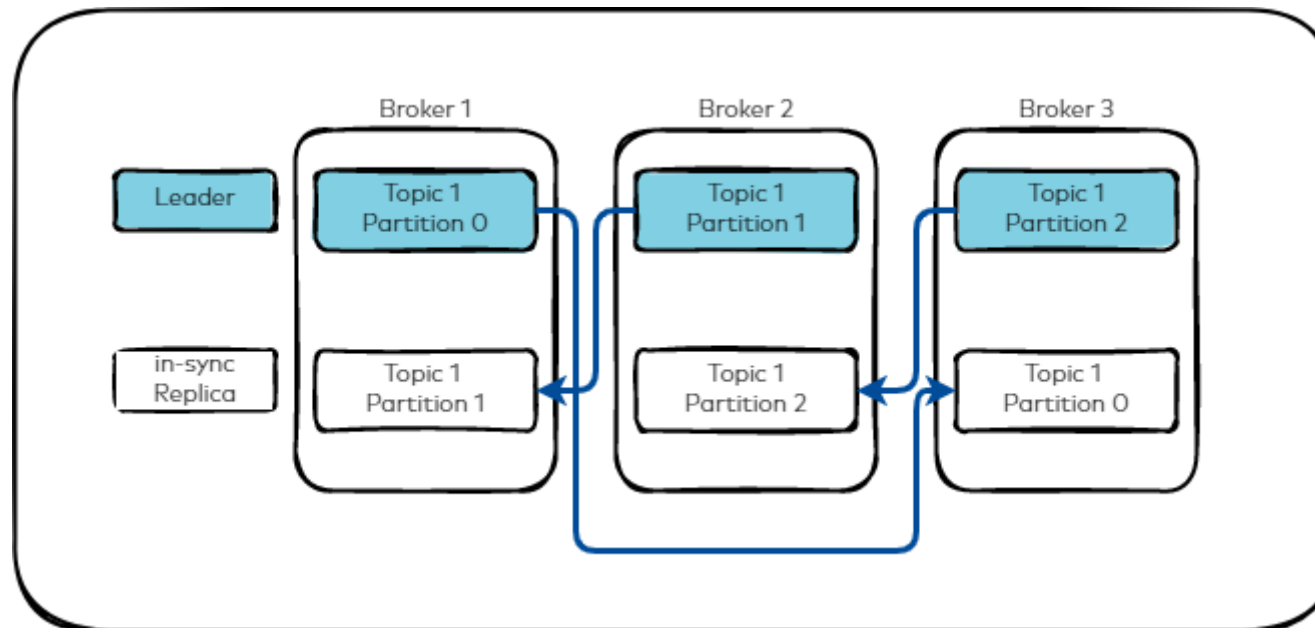
البته با اینکه کنترل مقدار اولیه offset برای شروع یک Consumer به دست خود Consumer و Zookeeper است، اما در اکثر موارد تمامی Consumer های یک Topic باید از یک نقطه، شروع به خواندن داده ها کنند. **در این حالت تا زمانیکه پیام با 1 offset، توسط Consumer ای خوانده نشود، هیچ Consumer ای نمی تواند پیام شماره 2 را بخواند.**

KAFKA TOPICS & PARTITIONS

کافکا built-in durability را از طریق log replication ارائه می‌کند.

پارتیشن‌های یک موضوع در میان broker های Kafka cluster توزیع می‌شوند و هر broker یک یا چند پارتیشن را ذخیره می‌کند. کافکا به شما این امکان را می‌دهد که ضریب replication را برای یک Topic پیکربندی کنید تا کنترل کنید چند سرور دارای کپی از داده‌ها برای fault tolerance هستند. به عنوان مثال، اگر شما سه broker و ضریب تکرار 2 داشته باشید، حداکثر دو تا از آن broker می‌توانند بدون از دست دادن داده‌ها شکست بخورند.

ضریب replication باید حداقل دو باشد تا در صورت خرابی سرور، fault tolerance فراهم شود.

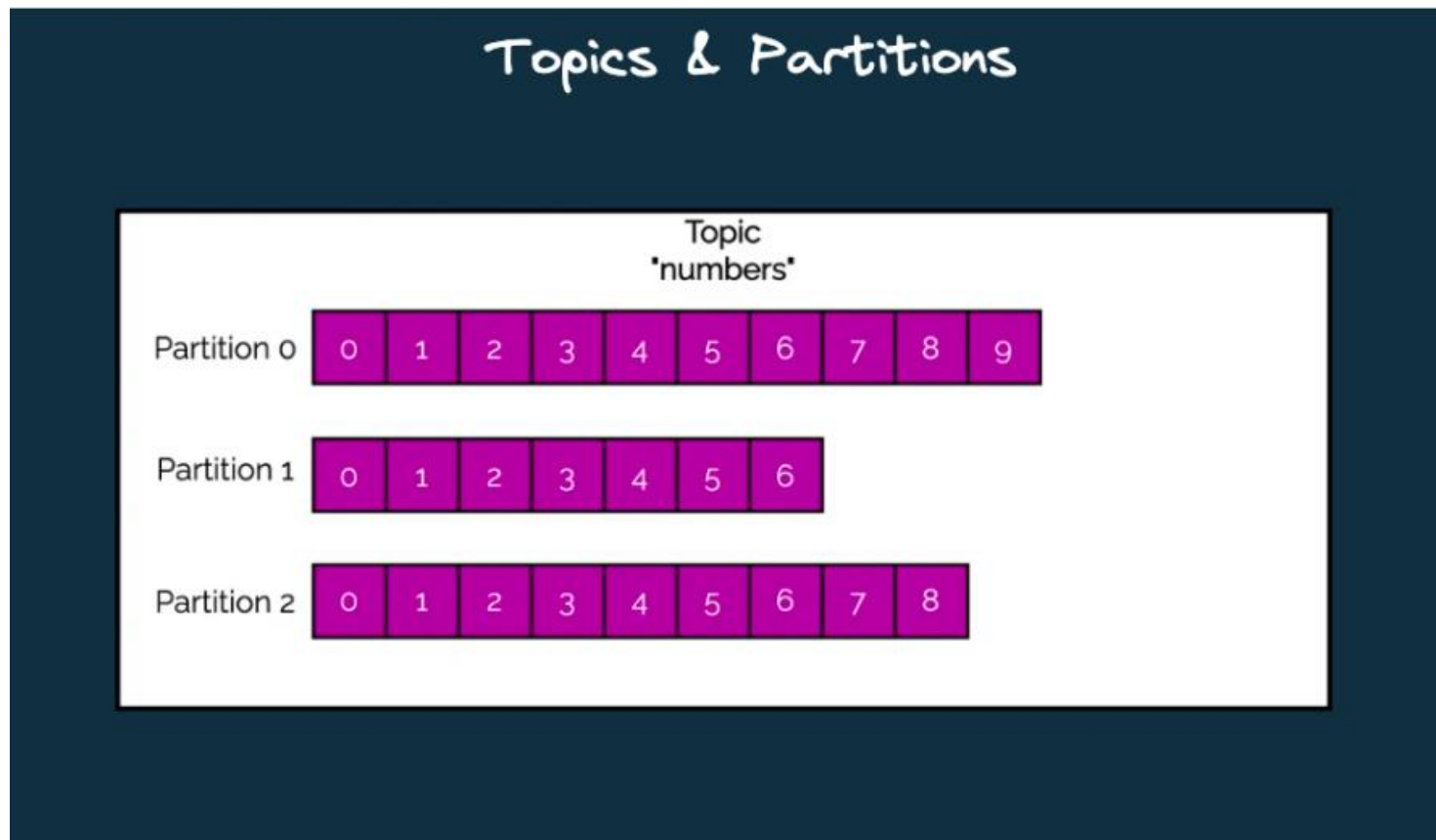


علاوه بر آن باید بدانید که هر Partition در هر زمان تنها توسط یک Primary Broker (leader) می‌تواند در دسترس سایر قسمت‌ها قرار بگیرد و تمامی عملیات خواندن و نوشتن در Partition توسط Kafka Server انجام می‌شود و سایر broker ها به صورت in-sync دیتا ها را خواهند داشت.

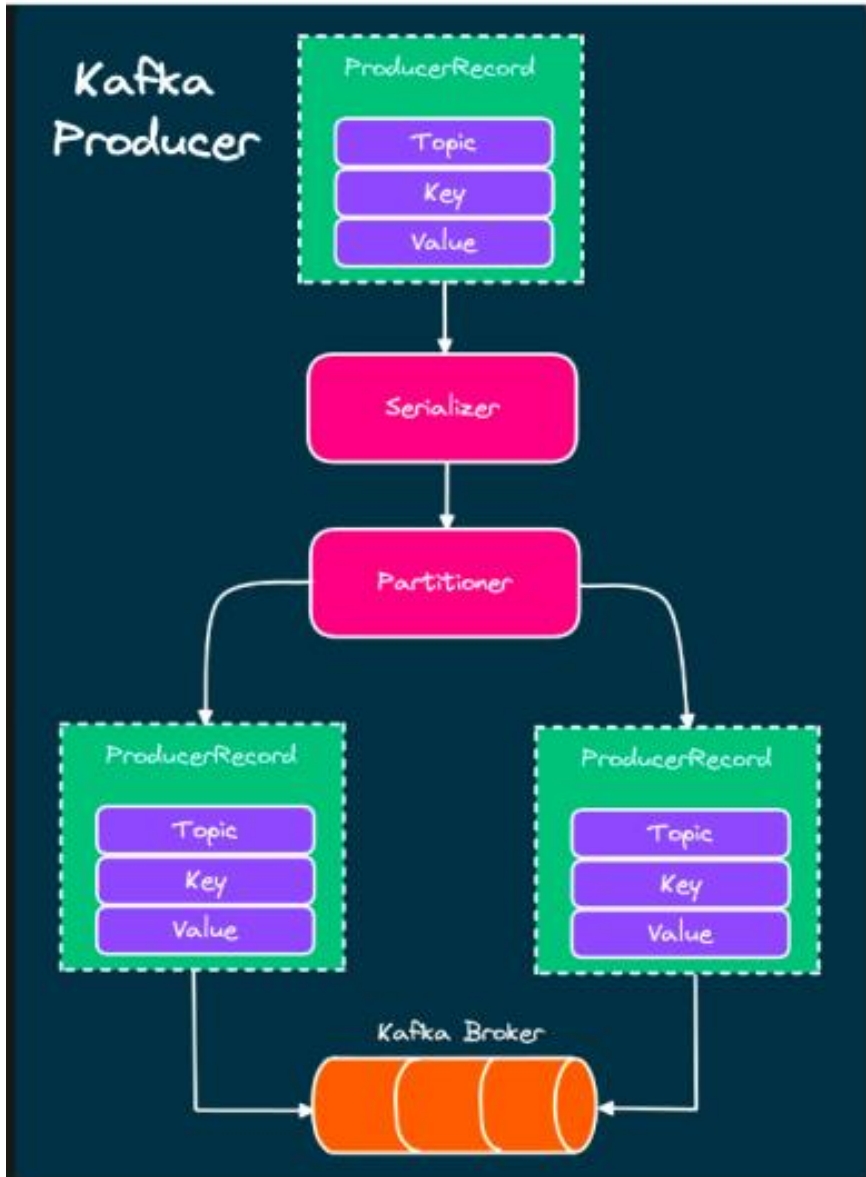
و در صورتیکه به هر دلیلی سرور leader از دسترس خارج شود، مدیریت این Partition به سرورهای دیگر (replica broker) داده می‌شود.

KAFKA TOPICS & PARTITIONS

باید در تعریف تعداد Partition های یک Topic این نکته را در نظر بگیرید که این تعداد کاملاً به نیازمندی شما و کارایی که شما مد نظر دارید، بستگی دارد.
مسیریابی پیام کافکا نسبتاً اساسی است و بر پارتیشن بندی مبتنی بر topic متکی است.



Producer



Producer می تواند یک اپلیکیشن ، سیستم یا زیرسیستم باشد.

Producers پیام های جدیدی ایجاد می کنند، آن ها را دسته بندی می کنند و به یک موضوع Topic می فرستند.

با استفاده از TCP connection دو طرفه با broker حرف می زنند

A Producer also **balances** messages across all partitions of a topic.

شما همچنین می توانید یک استراتژی پارتیشن بندی سفارشی ارائه دهید. اگر message ها key=null باشد دیتا ها (message ها) به صورت round robin ارسال می شود اما اگر key بزاریم Kafka تضمین می کند همیشه داخل یک partition ذخیره بکنیم. اما key اساسا زمانی ارسال میشه که ما احتیاج داریم message ها ترتیب داشته باشند.

✓ زمانی که broker جدید start می شود همه producer ها به طور خودکار شروع به جست و جو broker جدید کرده و پیام ها را به آن ارسال می کنند

Producer won't wait for acknowledgment
(possible data loss)

acks = 0



log collection/metric collection



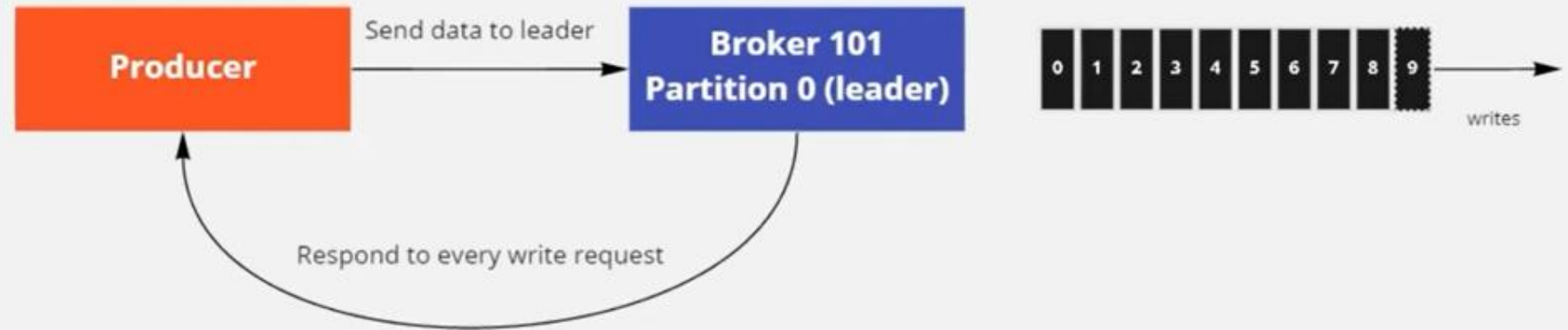
Broker 101
Partition 0 (leader)



writes

Default producer
will wait for leader
acknowledgment
(limited data loss)

acks=1



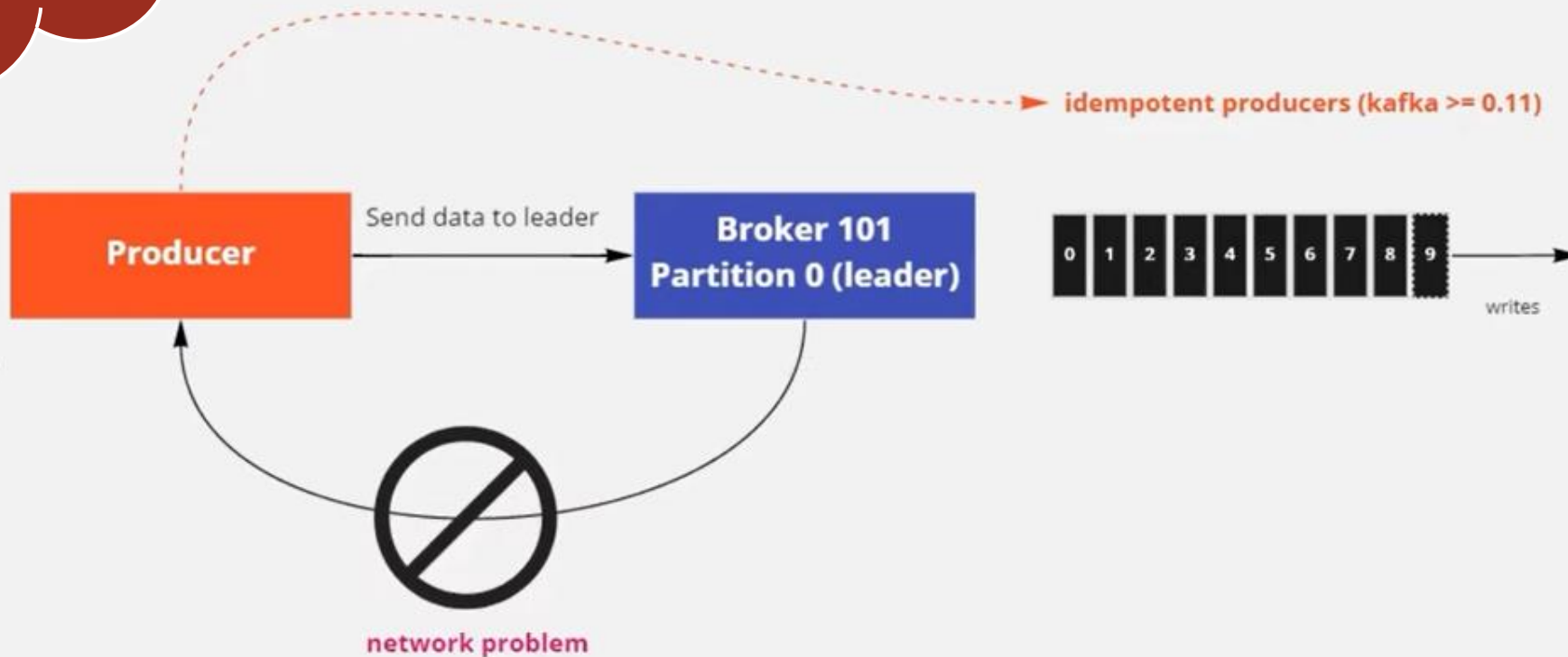
وقتی producer دوباره
message ارسال می کنه در
صورت عدم موفقیت در ارسال
پیام میاد نگاه می کنه آیدی
پیام اگر یک پیام duplicate
بشه رد می کنه و فقط acks
می کنه

acks=1

Property in
level producer

retries

- retry.backoff.ms
- delivery.timeout.ms

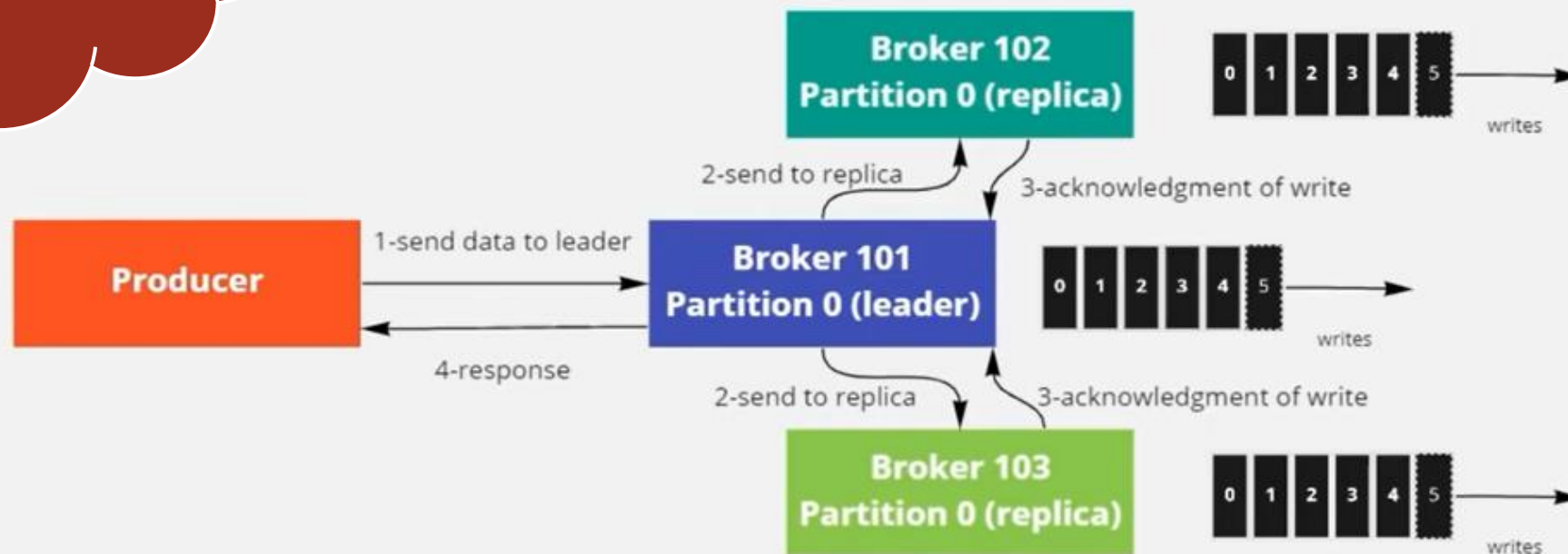


idempotent producers (kafka >= 0.11)

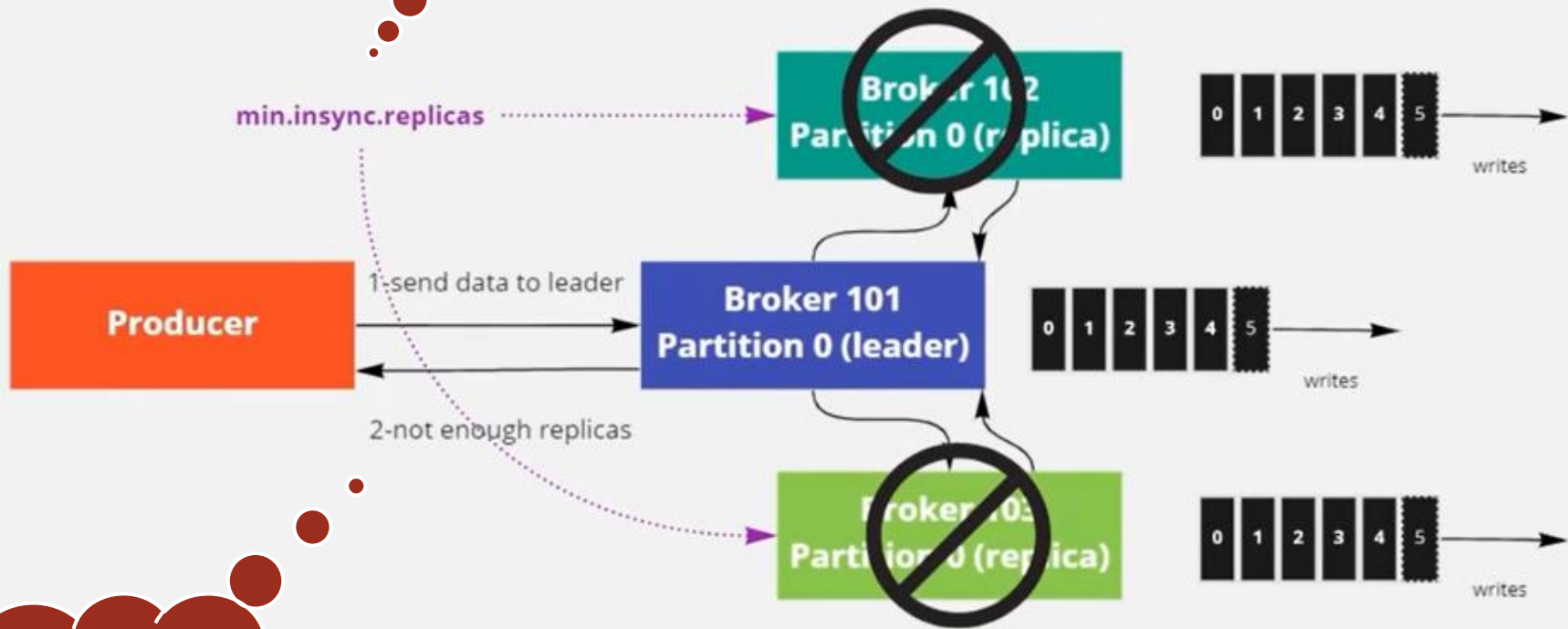
writes

Leader & replica
acknowledgment

acks=all



Property in
level producer
Value-> int
If value=2

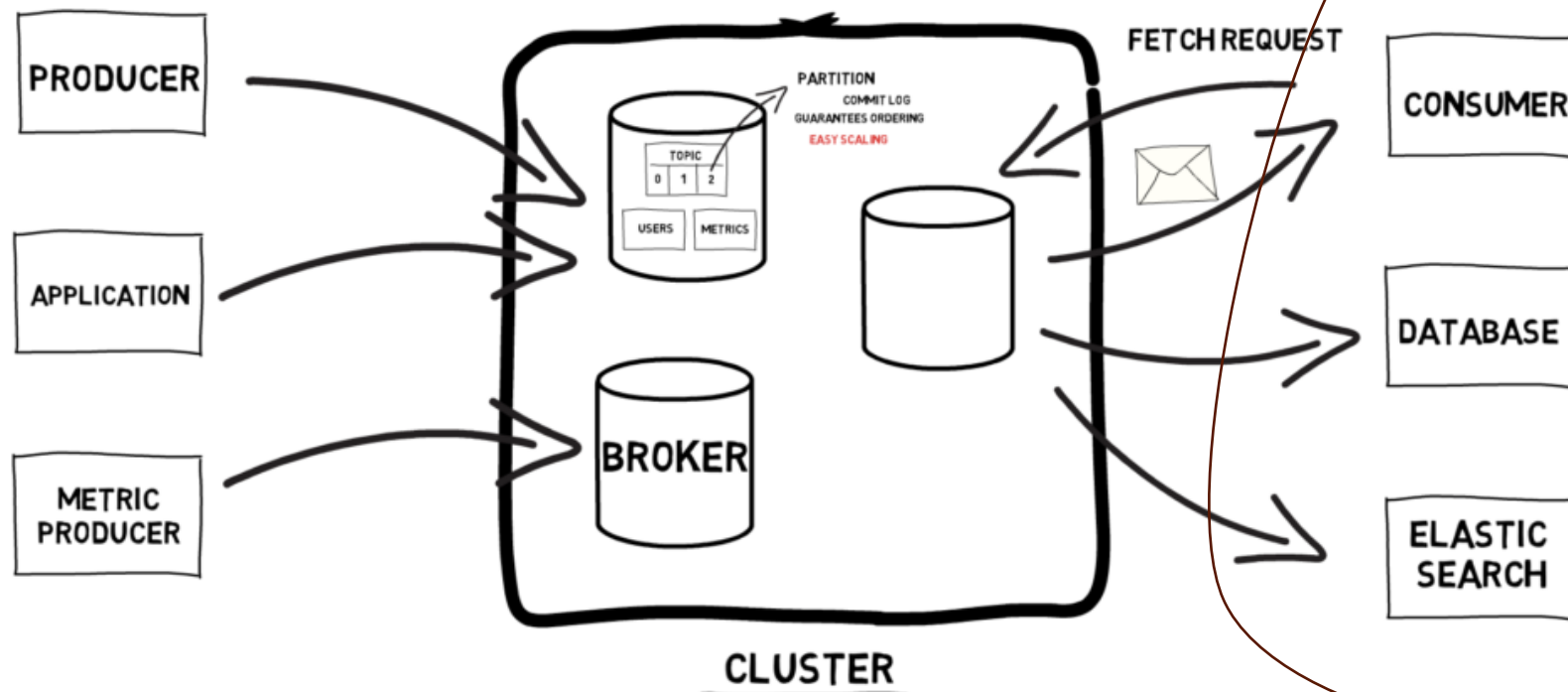


Leader return
exception
2-not....

Consumer

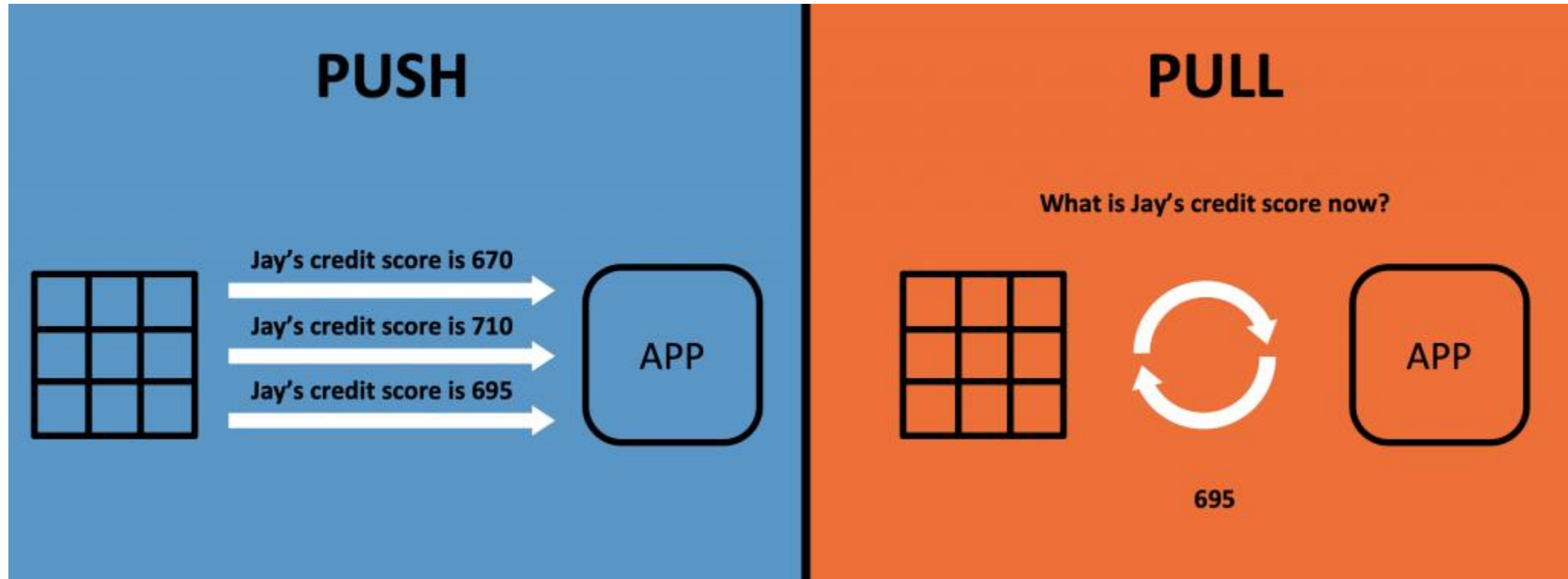
✓ دریافت کننده پیام یا رابط مصرف کننده
✓ Application، سیستم یا زیرسیستمی که بر روی یک یا چند Topic خاص، Subscribe کرده است به عبارتی دیگر به یک یا چند تاپیک متصل شده و به پردازش رکوردها می پردازد.
✓ با استفاده از TCP connection دو طرفه با broker حرف می زنه.

KAFKA ARCHITECTURE



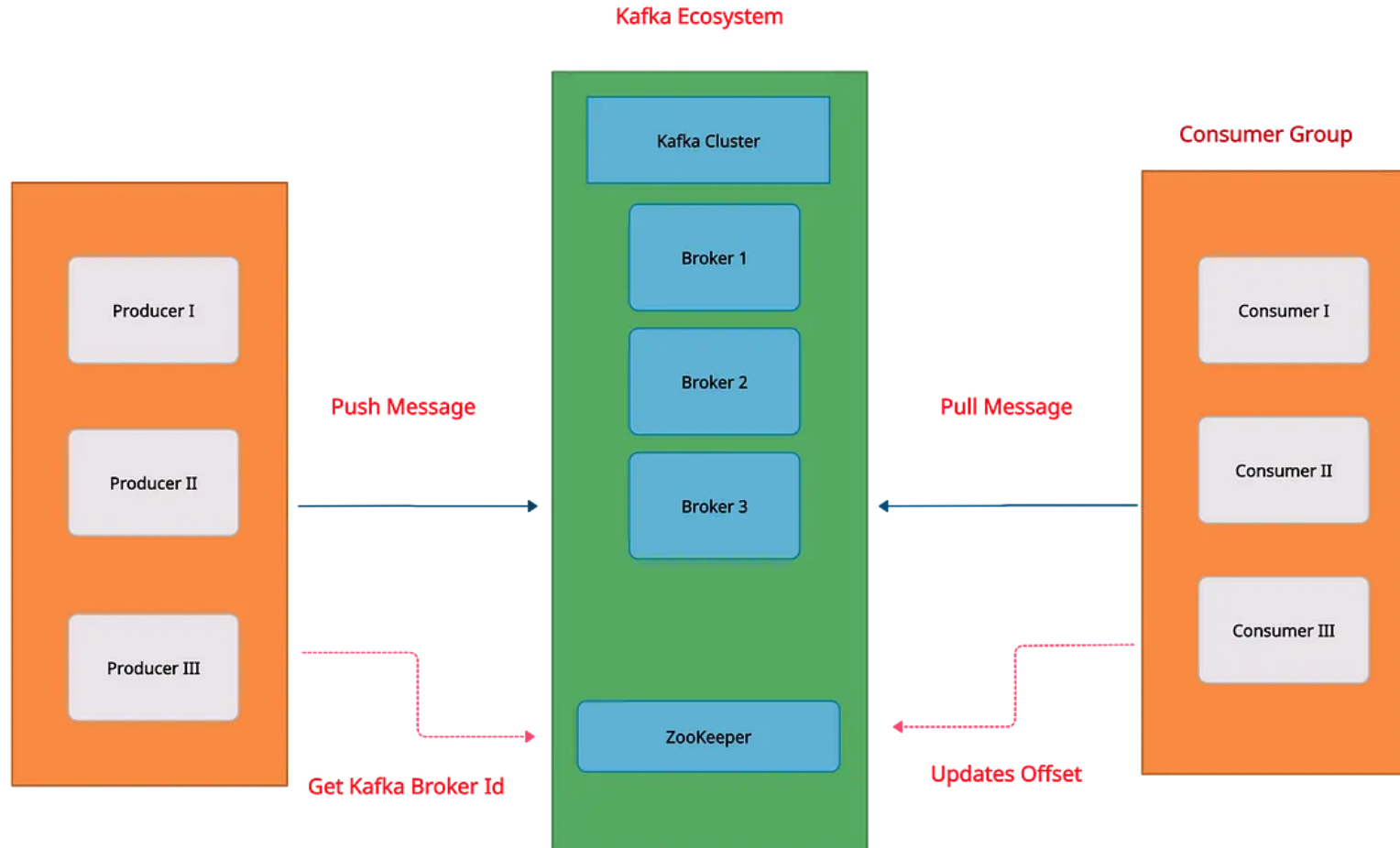
Pull-based message/data consumption

مصرف پیام Pull-based یک مدل مصرف داده است که در آن consumer به جای ارسال خودکار پیام ها توسط broker به consumer ها پیام ها را pull می کند (به طور فعال داده ها را درخواست می کند) در این مدل، consumer زمانی که آماده پردازش داده ها است، بازیابی داده ها را آغاز می کند و می تواند نرخ مصرف داده ها را کنترل کند.

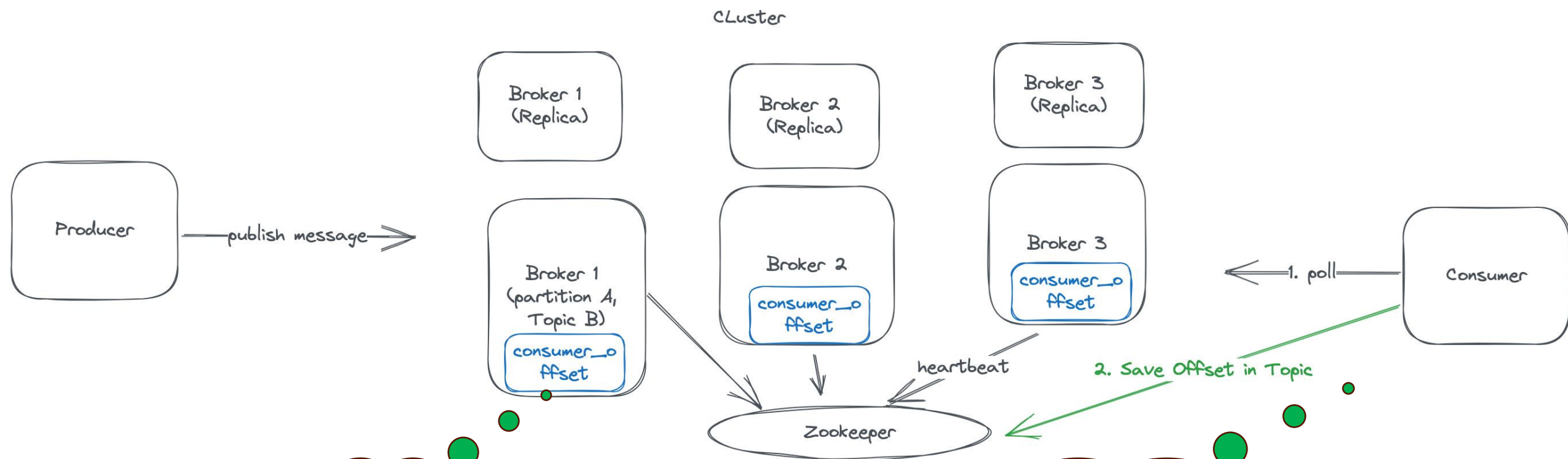


Apache Kafka: Pull-based method

در سیستم‌های pull-based، broker ها منتظر می‌مانند تا consumer اطلاعات را بخواهد ("pull"). اگر consumer دیر کند، می‌تواند بعداً می‌تواند پیام‌ها را مصرف کند (pull).
در این سیستم‌ها Dumb broker / smart consumer مدل خواهیم داشت broker پیام‌هایی را که توسط مصرف‌کنندگان خوانده می‌شود ردیابی نمی‌کند و فقط پیام‌های خوانده نشده را نگه می‌دارد. (broker های کافکا چیزی در مورد offsets های مصرف‌کننده نمی‌دانند).
Consumer ها ایندکس پارتیشن‌ها و نگه‌میدارن و بصورت پی‌درپی پیام‌ها رو می‌خوانن



Consumers keep track of their offsets in a special topic called __consumer_offsets.



consumer به offset کمک می‌کند تا در صورت مرگ از جایی که ترک کرده است ادامه دهد.

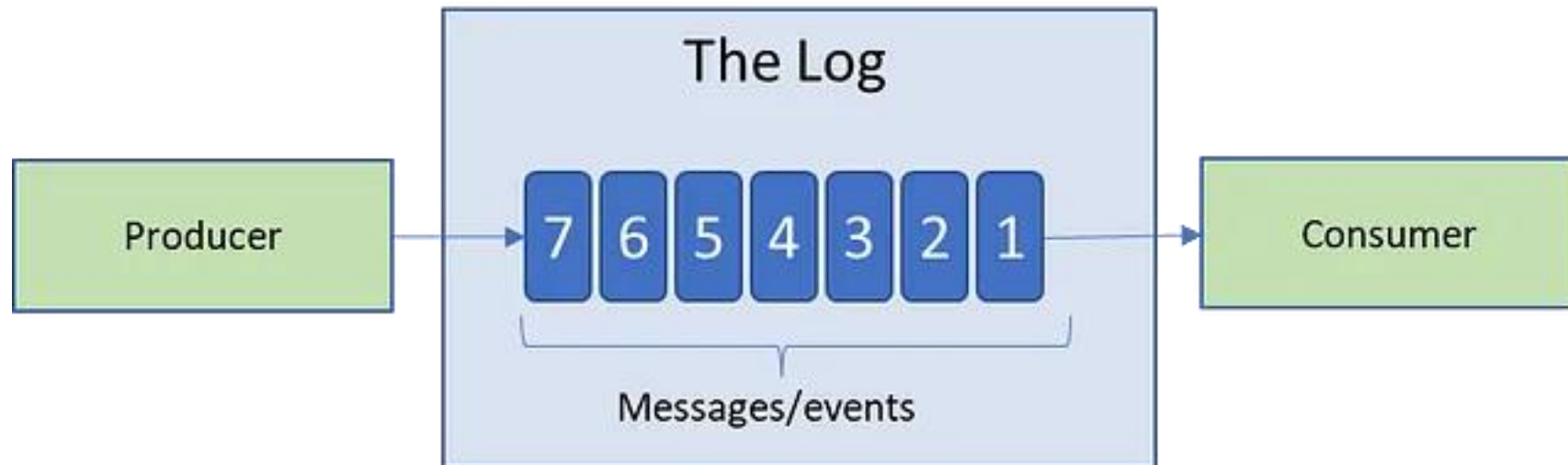
از اونجایی که consumerها ایندکس پارتیشن رو نگه میدارن پس می‌تونن مشخص کنن که subscriptionشون بصورت **durable** باشه که بعد از هر بار ریستارت ادامه پیام‌ها رو پردازش کنن یا بصورت **ephemeral** باشه و با هر بار ریستارت از آخرین پیامی که وارد پارتیشن شده کارشونو شروع کنن.

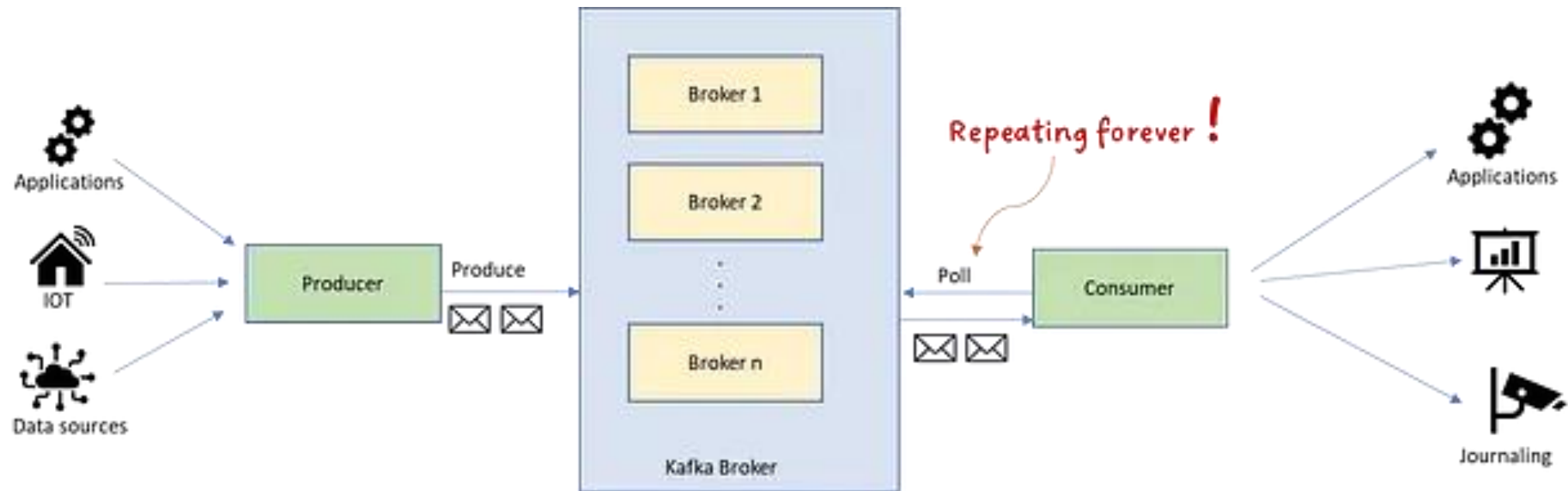


Why does Kafka use a pull-based message consumer model?

کافکا یک queue نیست، یک log است - log از پیام ها/رویدادها. Producer ها به log اضافه می کنند و مصرف کنندگان از log می خوانند.

log تغییر ناپذیر است اما نمی تواند تعداد بی نهایت داده را ذخیره کند، بنابراین یک زمان پیکربندی شده برای زنده ماندن سوابق در log وجود دارد. از سوی دیگر در یک queue، پیام ها پس از پردازش/دریافت توسط مصرف کننده حذف می شوند.

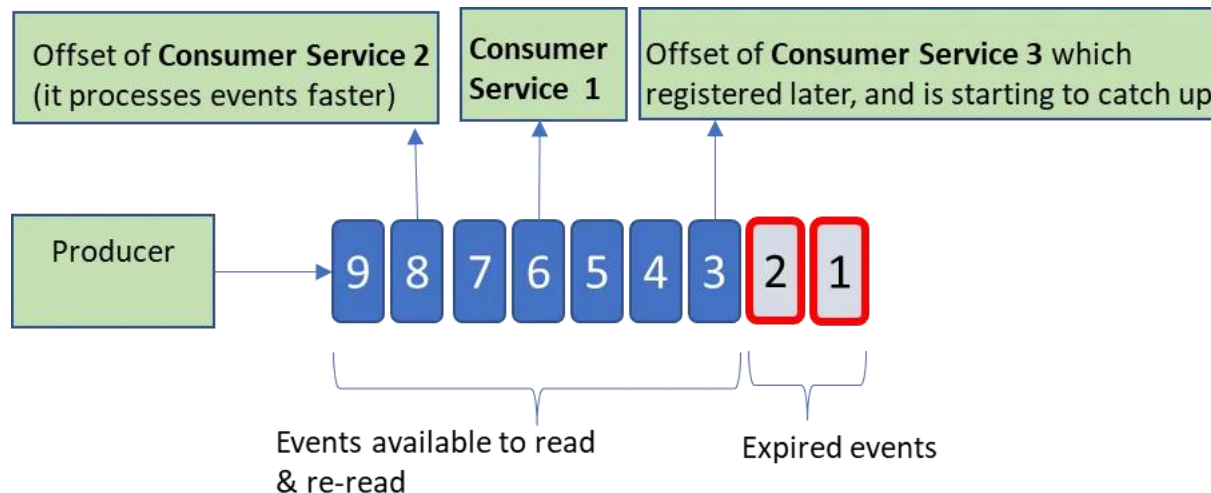




Producer ها یک درخواست تولید همراه با رکوردها را به log ارسال می کنند و به هر رکورد، با رسیدن، همان طور که گفتیم شماره خاصی به نام offset داده می شود که موقعیت logical آن رکورد در log است. مصرف کنندگان یک درخواست واکشی برای خواندن سوابق ارسال می کنند، و آنها از offsetها برای bookmark، مانند متغیرهای مکان، استفاده می کنند.

Now let's see how the pull model helps Kafka.

Kafka is deployed with consumers varying in processing capacities & requirements. For simplicity concept of partition is not shown in this image.



1. Diverse consumers

Schema های log به چندین گروه consumer مستقل اجازه می‌دهد تا با سرعتی که برای گروه consumer امکان‌پذیر است، از log بخوانند. همانطور که در تصویر بالا نشان داده شده است، مدل consume Pull-based consumer به consumer های متنوع با نرخ های مصرف متفاوت اجازه می‌دهد. برای مصرف رکورد ها.

2 - مدل pull-based consumption کافکا کنترل دقیق تری بر پردازش پیام، مدیریت خطا، acknowledgement و مدیریت offset و امکان مصرف مجدد داده ها در صورت نیاز به consumers می‌دهد.

مزیت دیگر یک سیستم pull-based این است که به دسته بندی کارآمد داده های ارسال شده به consumer کمک می‌کند. در حالی که یک سیستم push-based باید انتخاب کند که یا درخواستی را فوراً ارسال کند یا داده های بیشتری را جمع آوری کند و بعداً بدون اطلاع از اینکه downstream consumer قادر به پردازش فوری آن خواهد بود یا خیر ارسال کند.

نقص کلی یک سیستم pull-based ساده این است که اگر broker داده ای نداشته باشد،

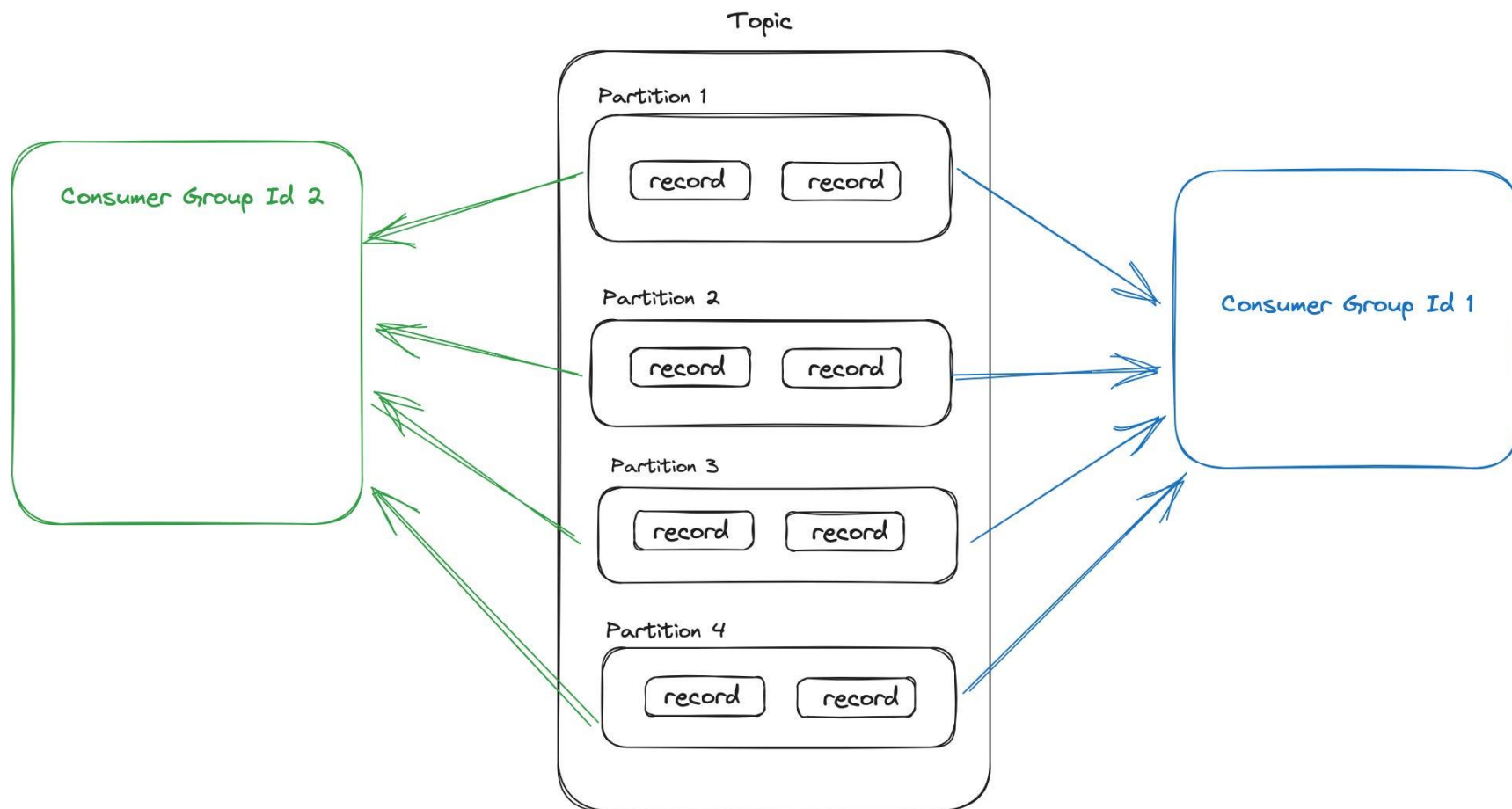
the consumer may end up polling in a tight loop wasting resources

منابع را در حالی که منتظر رسیدن داده است هدر دهد.

برای جلوگیری از این امر، کافکا پارامترهایی در درخواست واکشی خود دارد که به درخواست consumer اجازه می‌دهد تا در یک «long poll»، منتظر بمانند تا داده‌ها در دسترس باشند یا تا زمان انقضا، مسدود شود.

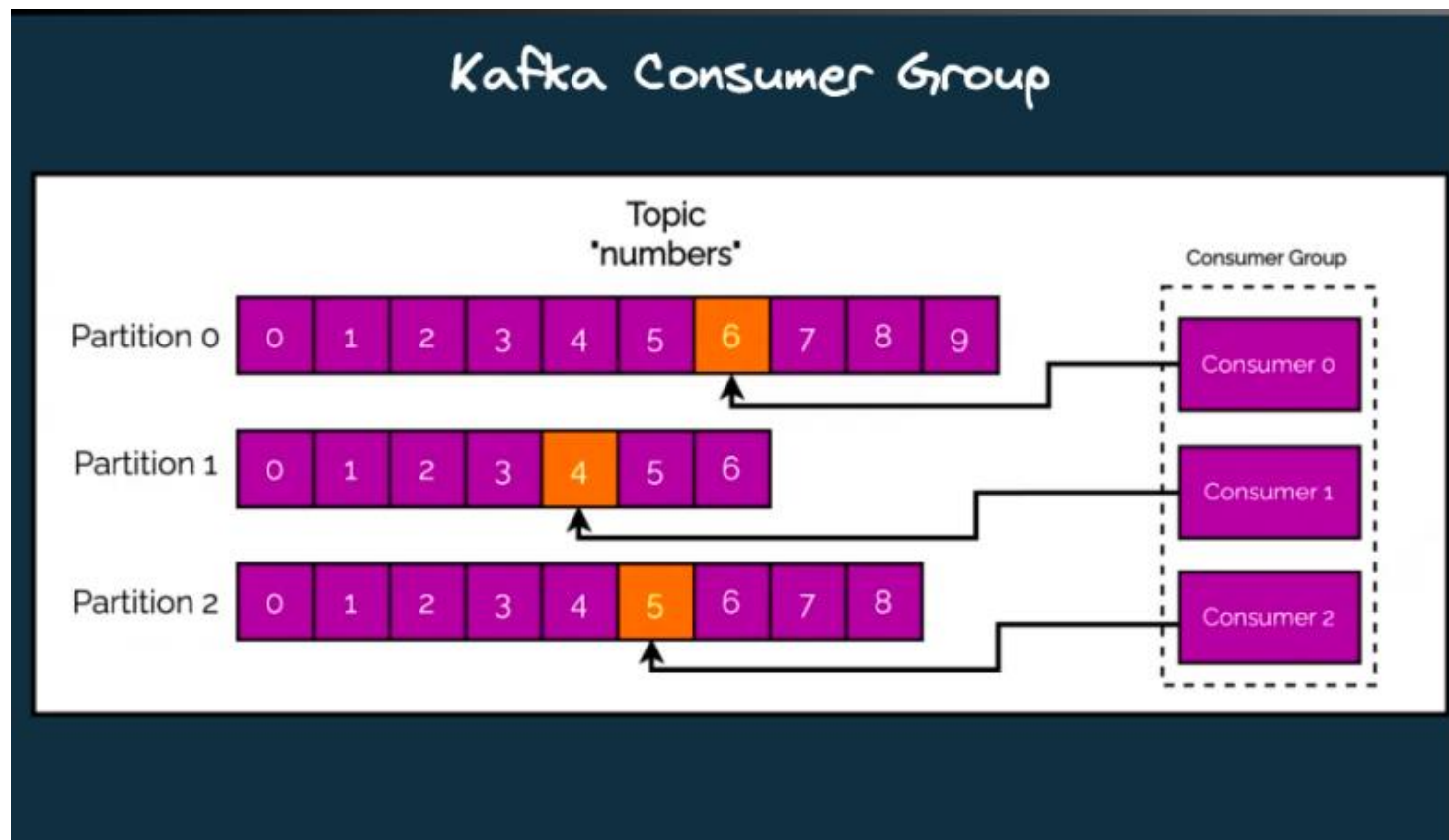
What is the Consumer Group in Kafka?

وقتی سیستمی دارید که از producer های زیادی تا consumer های زیادی تشکیل شده است، به راهی نیاز خواهید داشت که یک موضوع توسط چندین consumer مصرف شود. که کافکا با گروه بندی کردن consumer ها به آن دست یافت. کافکا از شناسه گروه برای شناسایی همه consumer ها در یک گروه استفاده می کند.



consumer group: یک یا چند consumer به عنوان یک گروه consumer برای مصرف پیام های یک Topic کار می کنند. عموماً این گروه شامل یک Replicate از یک Application است.

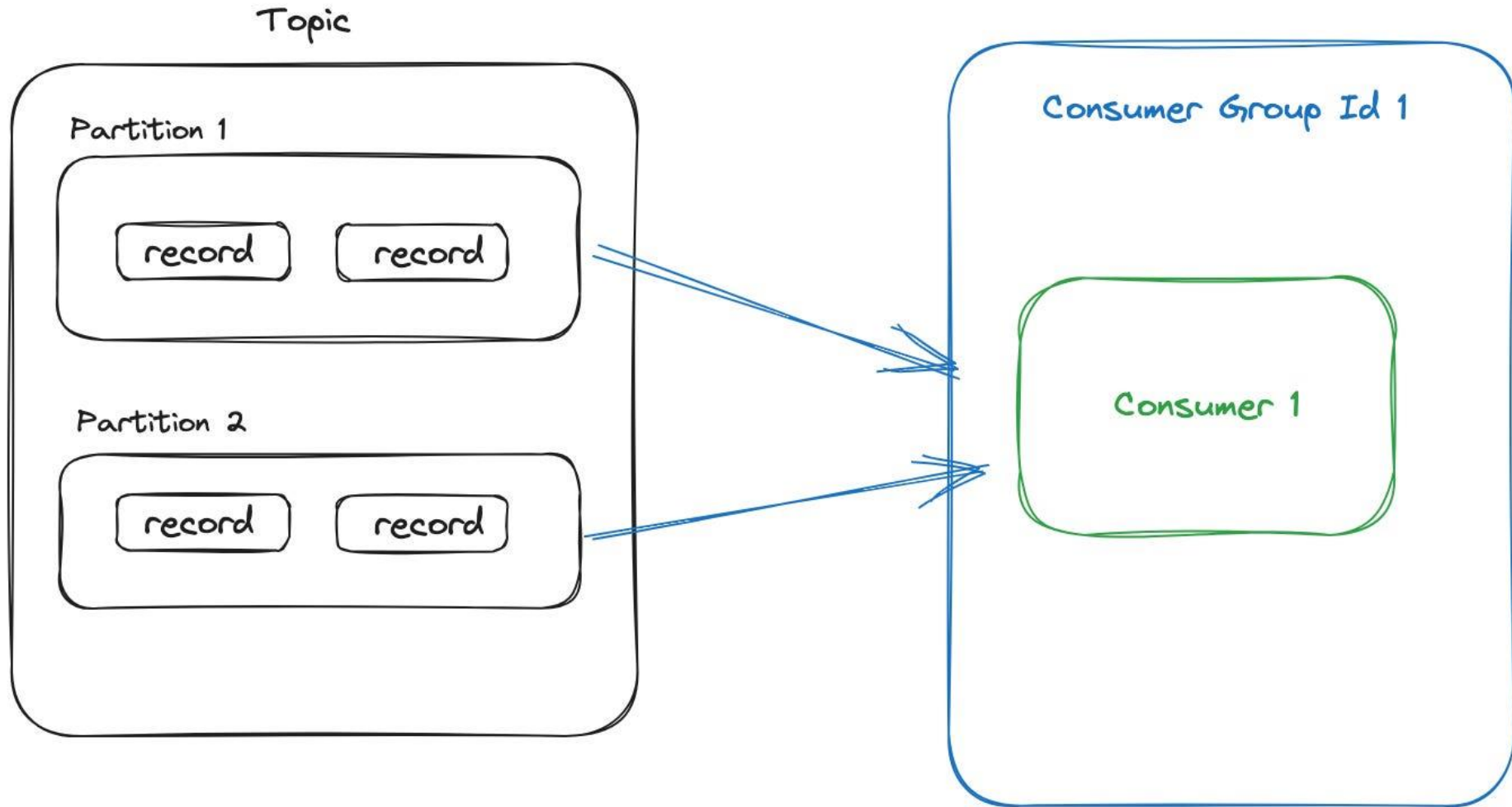




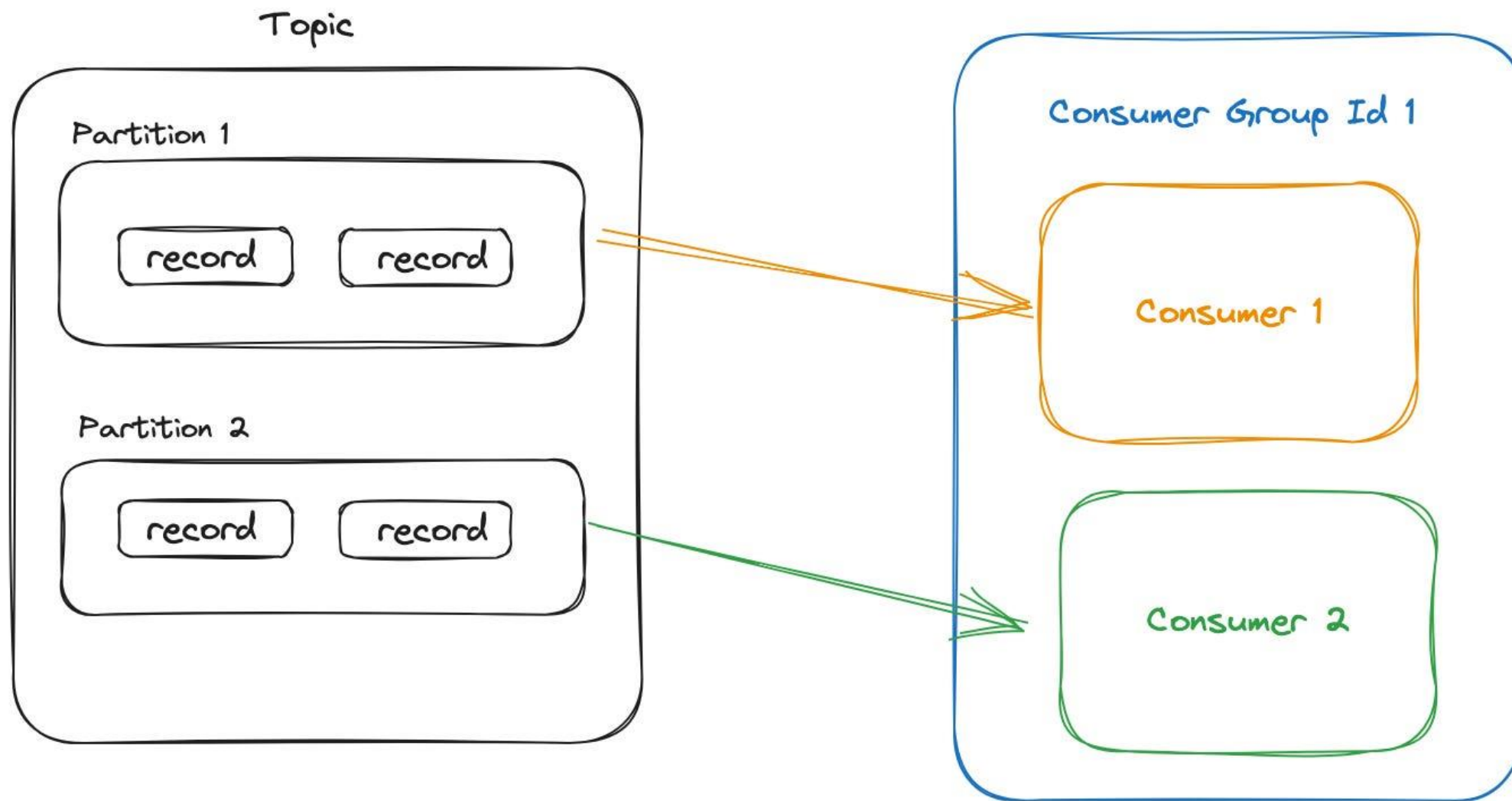
به یاد دارید که topic از یک یا چند پارتیشن تشکیل شده است؟ جدا از هدف افزایش توان عملیاتی و در دسترس بودن، کافکا همچنین تضمین می کند که تنها یک consumer هر پارتیشن را در آن گروه مصرف کند.

کافکا همچنین دارای پشتیبانی داخلی برای فیلتر کردن پیام است. این به consumer ها اجازه می دهد فقط پیام هایی که به آن ها علاقه دارند subscribe کنند. برای مثال، یک consumer می تواند فقط پیام هایی با یک کلید یا مقدار خاص را subscribe کنند.

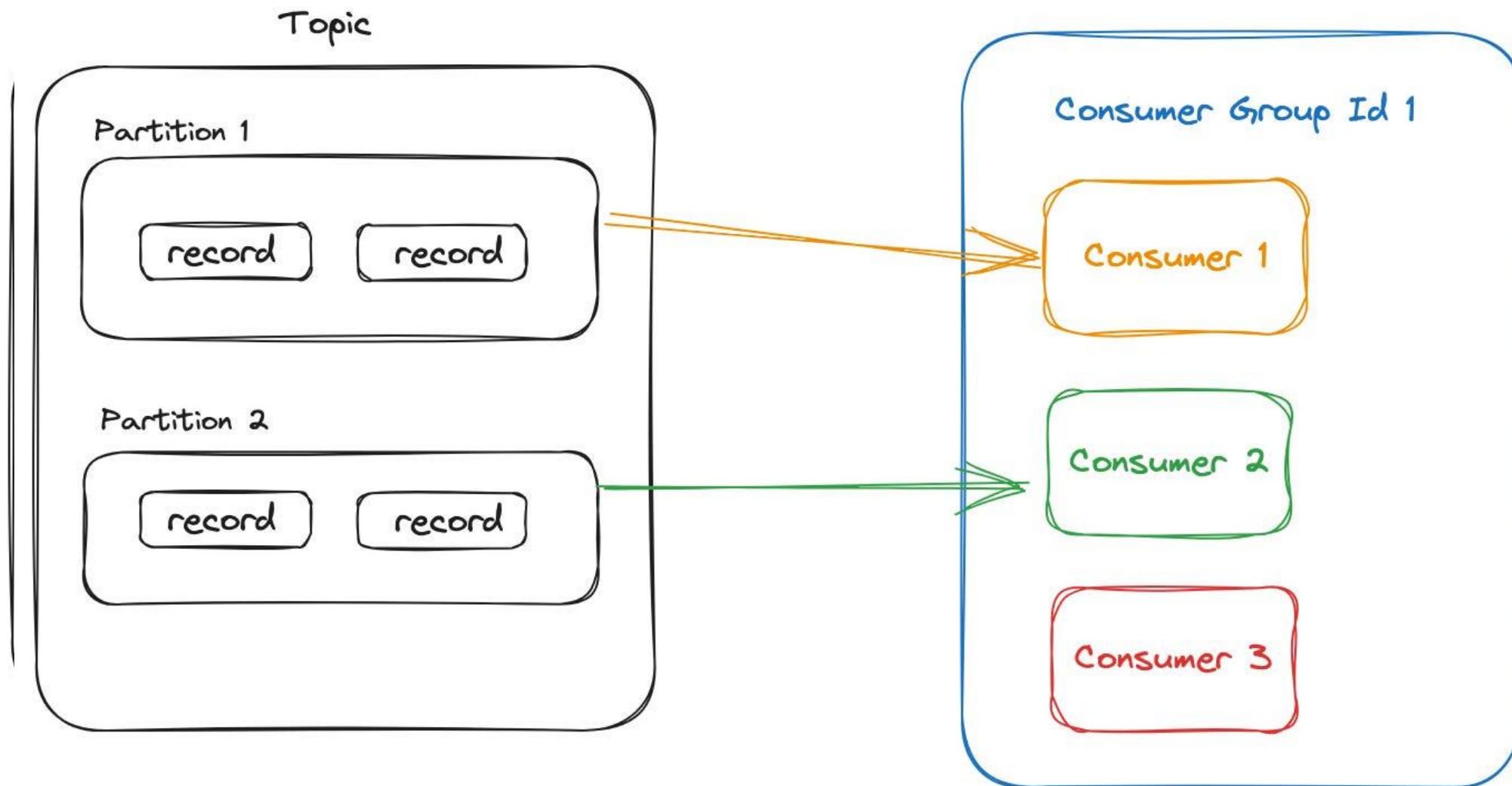
هنگامی که یک consumer راه اندازی می شود، به یک گروه consumer ملحق می شود



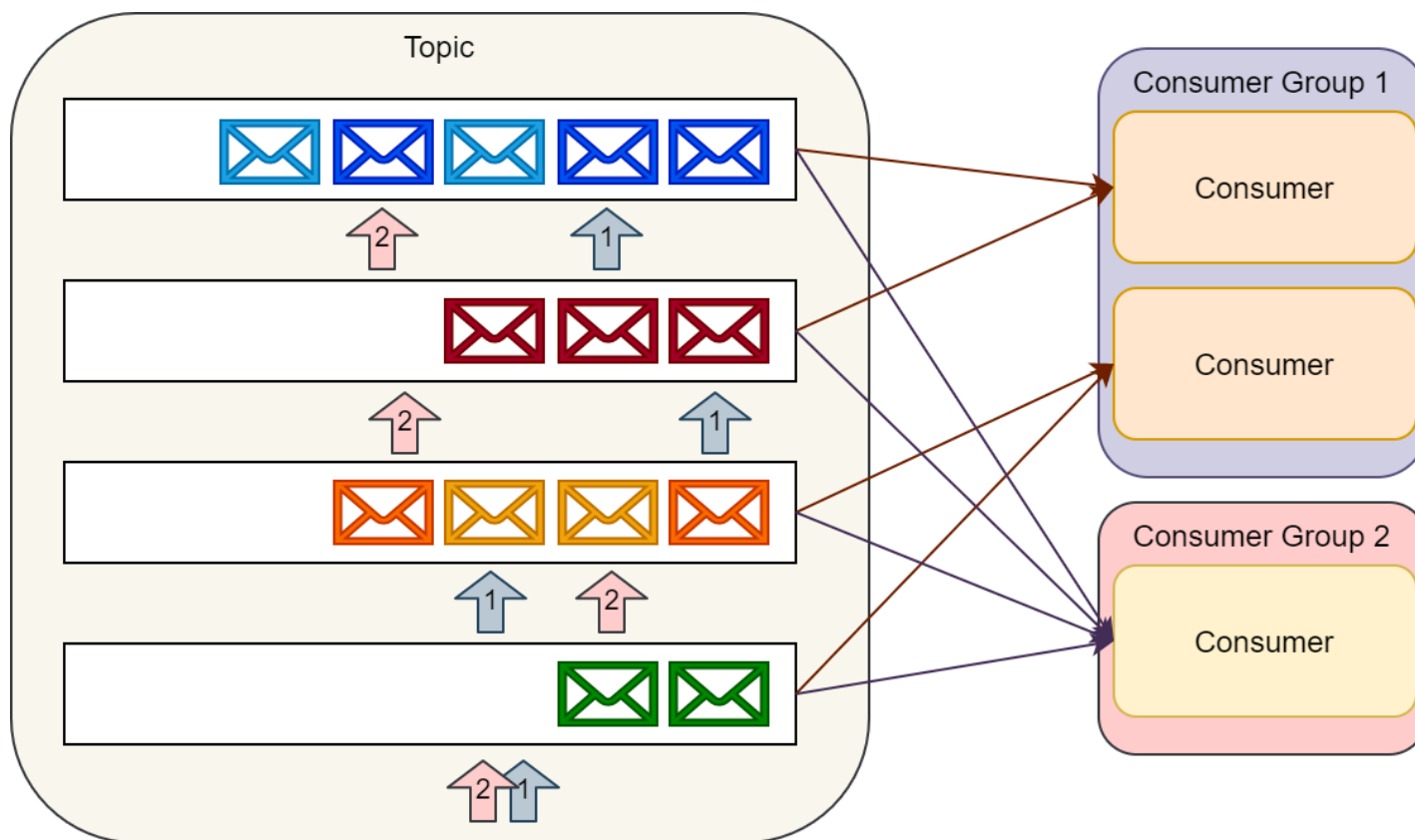
حال، اگر consumer را به دو برسائید، چه اتفاقی می افتد؟ هر consumer یک پارتیشن مصرف خواهد کرد.



دوباره، چه اتفاقی می‌افتد اگر تعداد consumer های بیشتری در گروه نسبت به پارتیشن ارائه شده توسط کارگزار داشته باشید؟



ان یک مصرف کننده بیکار انجا خواهد نشست. اگر می‌خواهید تعداد مصرف‌کنندگان را افزایش دهید، باید با افزایش تعداد پارتیشن‌ها بر این اساس برنامه‌ریزی کنید. بنابراین، پارتیشن یک واحد موازی از دیدگاه مصرف‌کنندگان است.



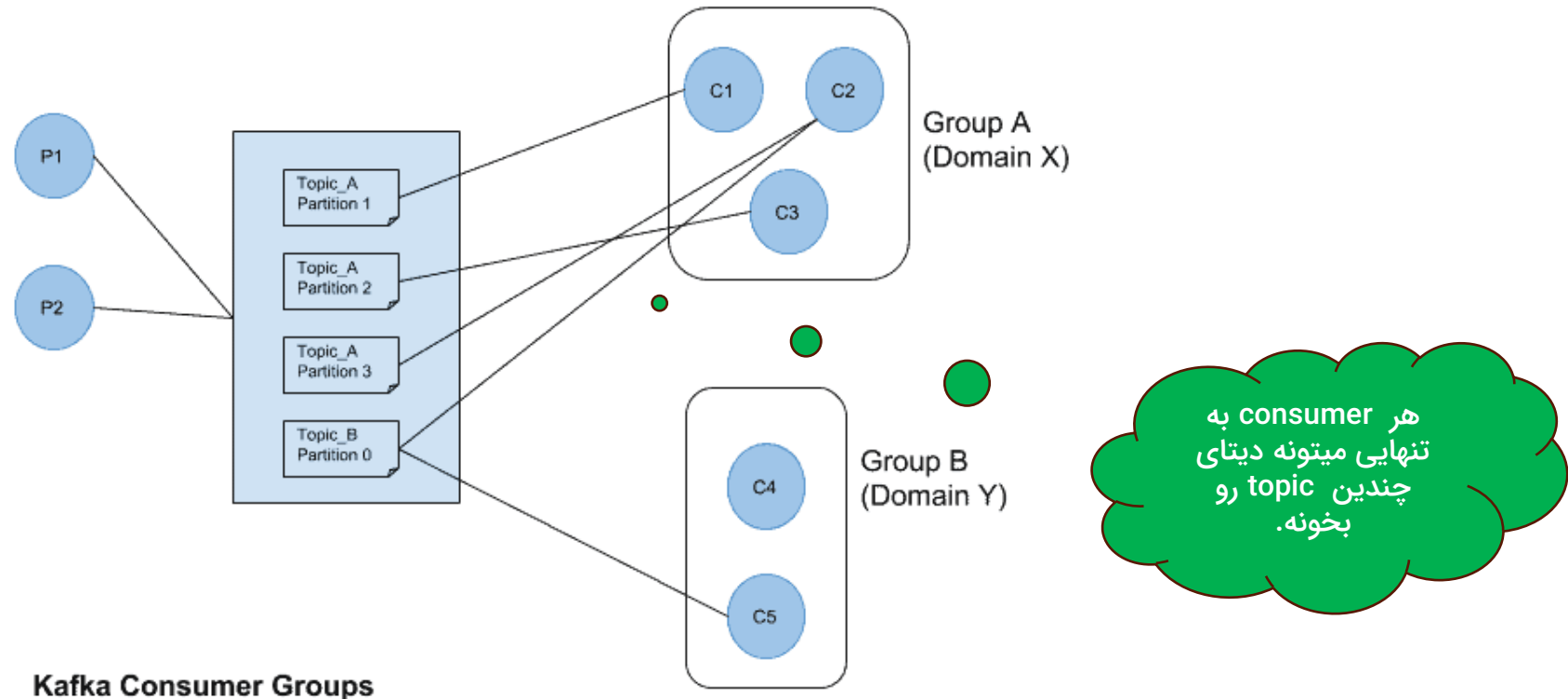
✓ پارتیشن و consumer در داخل گروه consumer یک رابطه 1:1 است. بنابراین، اگر می خواهید consumer خود را مقیاس کنید، باید پارتیشن های خود را نیز مقیاس کنید. **در نتیجه همیشه تعداد consumer ها رو به تعداد پارتیشن های موجود scale کرد.**

✓ پارتیشن یک واحد parallelism در کافکا است. هر consumer می تواند در داخل گروه consumer به صورت موازی پردازش کند.



هر partition حداکثر توسط یک consumer از یک گروه **consume** **میشه** یعنی یک partition نمی تواند همزمان توسط دو consumer از یک گروه consume بشه.

دو معماری message-queue و pub/sub قابل پیاده سازی در kafka هستند
زمانی که بخواهیم پترن Queue داشته باشیم همه consumer ها را داخل یک گروه قرار می دهیم و هر consumer وظیفه consume هر partition داره ولی خب این پترن می تونه یه سری مشکلات داشته باشه.
وقتی که پترن pub/sub می خواهیم داشته باشیم هر consume داخل گروه unique قرار می دهیم در نتیجه partition توسط چندین consumer در گروه های مختلف می تواند consume شود.



Kafka Consumer Groups

همان طور که گفتیم consumer ها توانایی دارن که offset ها تا جایی که read کردن restore کنن.

kafka topic name_consumer_offsets

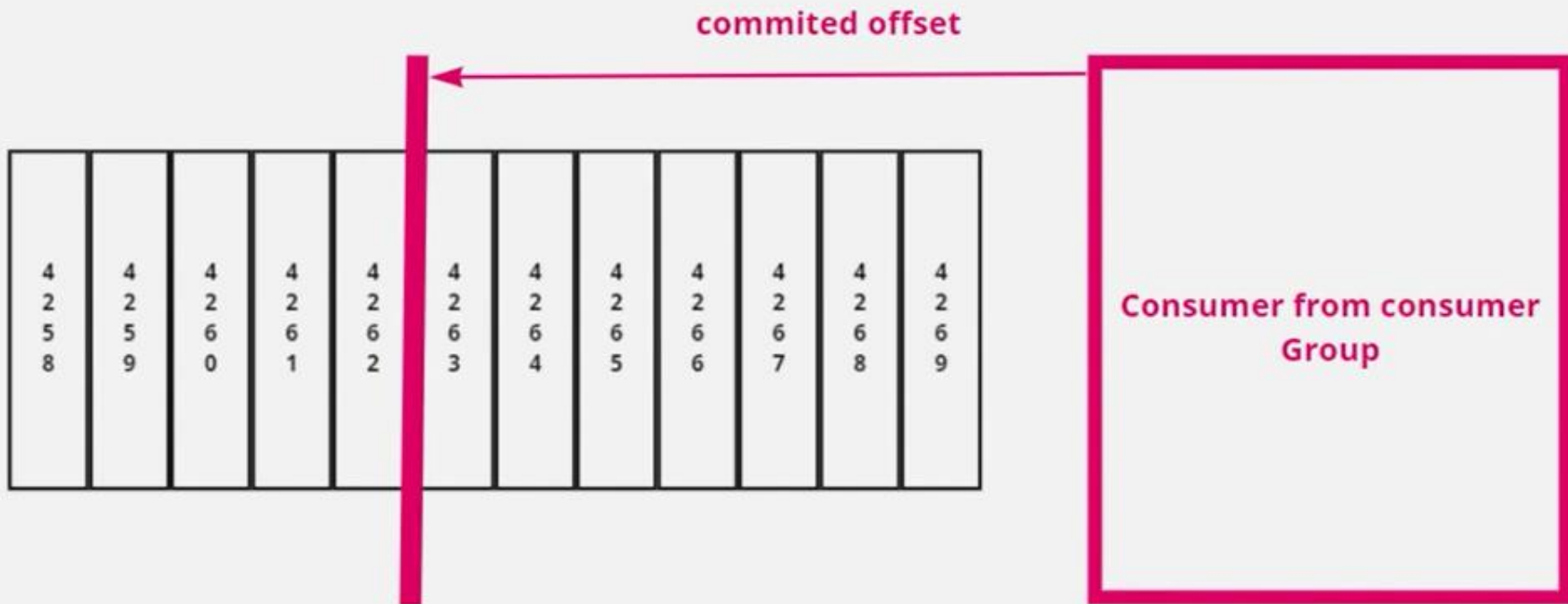
Consumer Offset

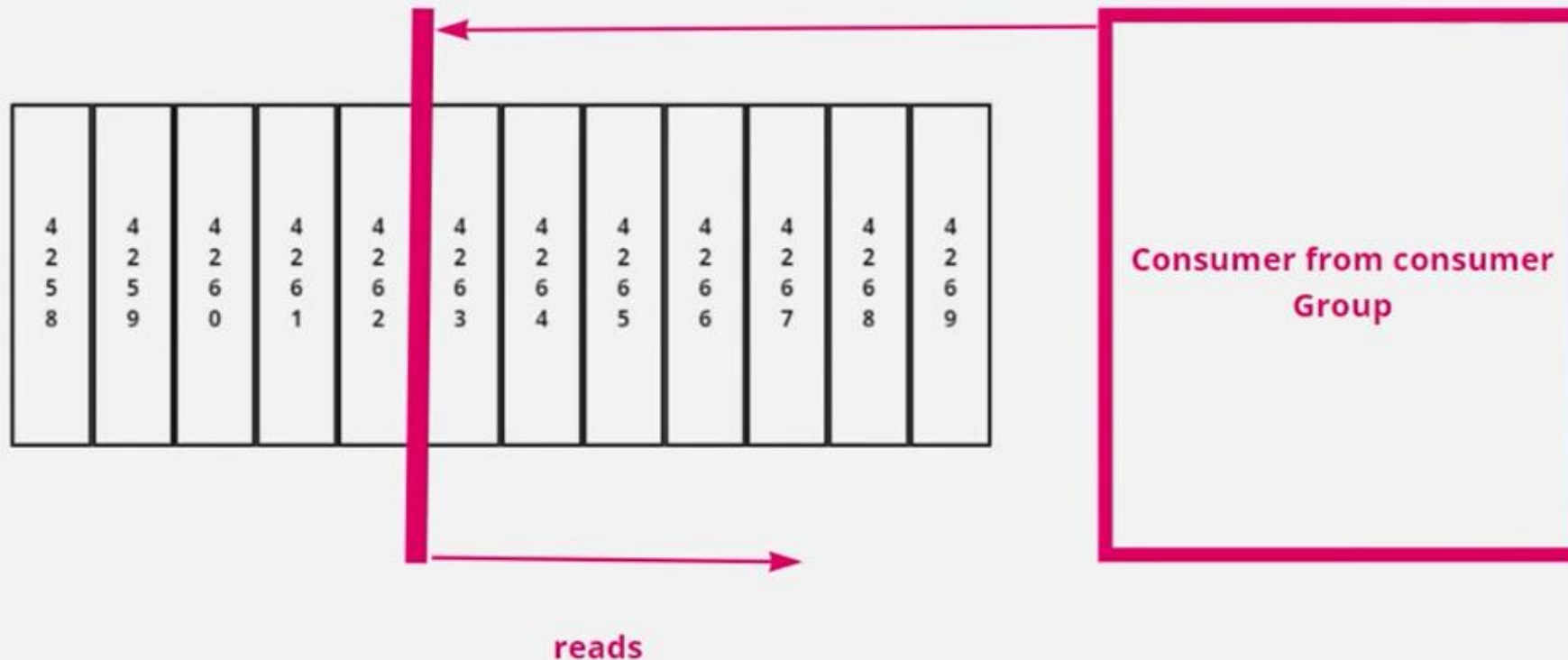


4	4	4	4	4	4	4	4	4	4	4	4
2	2	2	2	2	2	2	2	2	2	2	2
5	5	6	6	6	6	6	6	6	6	6	6
8	9	0	1	2	3	4	5	6	7	8	9

Consumer from consumer
Group

زمانی که consumer داره پیام ها رو process می کنه می آید offset را commit می کنه یعنی action صدا می زنه.





زمانی که consumer (مصرف کننده) down بشه این اتفاق باعث میشه که بتوانیم Read back کنیم از جایی که Left off شده.

Commit offset تاثیر داره روی Deliver semantics

به محض رسیدن پیام به consumer
Offset = committed
این خوب نیست چون اگر processing پیام به هر دلیلی به مشکل بر بخوره پیام و دیتا ما loss میشه

Deliver semantics

At most once



At least once



Exactly once

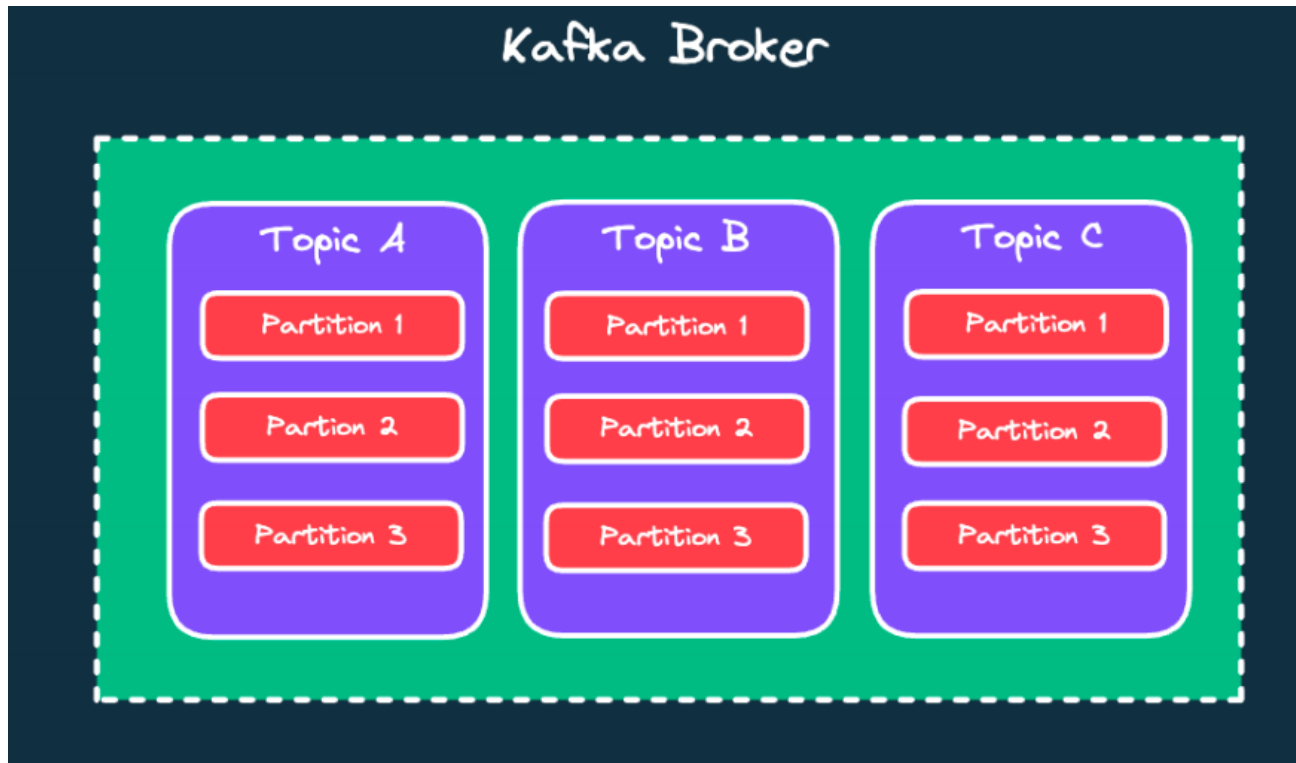


Offset زمانی commit میشه که process پیام تمام بشه در نتیجه پیام از بین نمی ره فقط مشکلی که پیش میاد idempotent

Idempotent -> برای مدیریت پردازش پیام تکراری بهتره این رویکرد را در نظر بگیریم
پردازش یک پیام تکراری نباید تاثیری توی سیستم ما داشته باشد.

Broker

A single Kafka server is called a Broker
A broker is part of a Kafka cluster
Port default : 9092



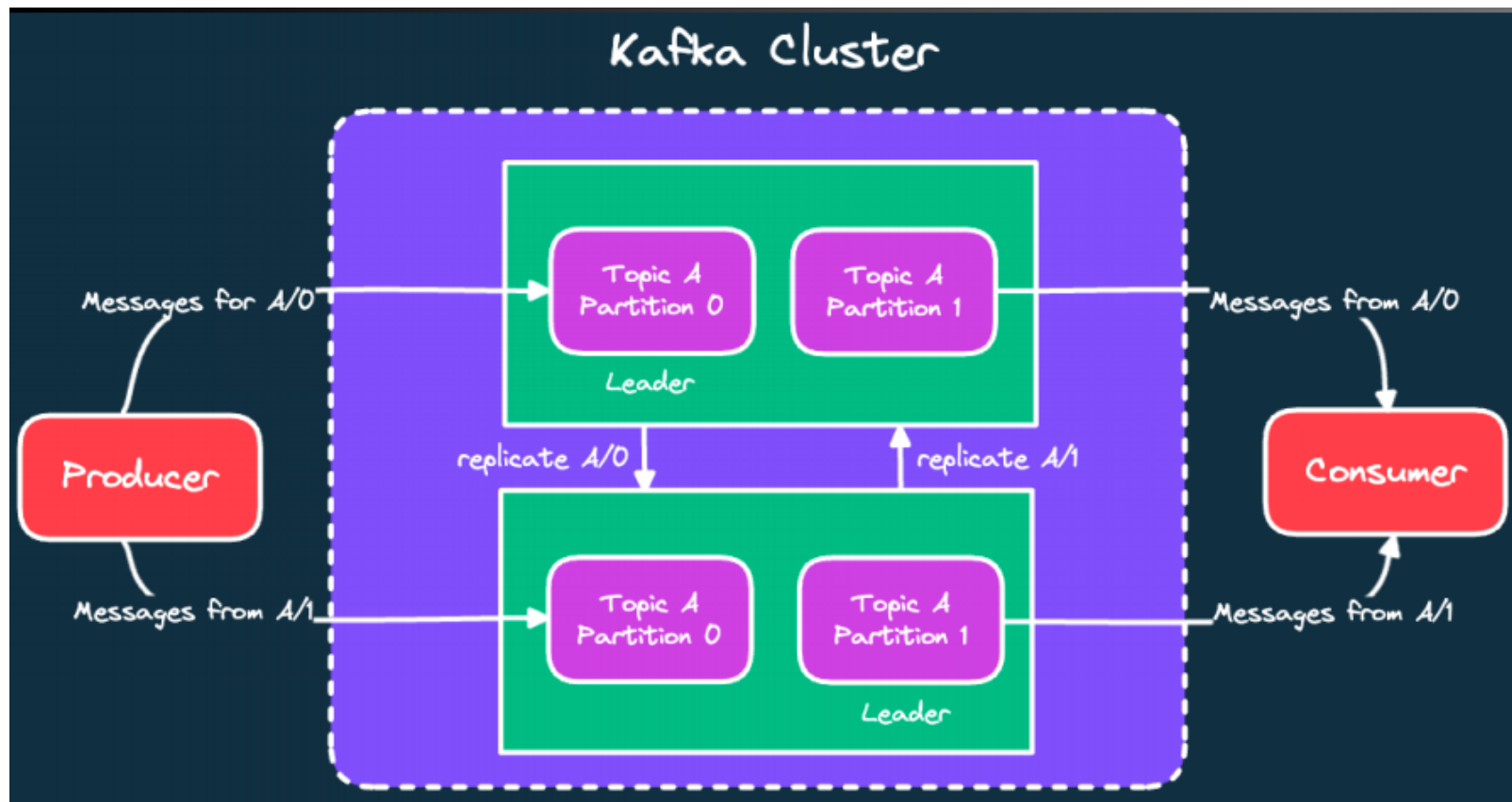
یک **Broker** می تواند هزاران پارتیشن و میلیون ها پیام را در هر ثانیه مدیریت کند

یک **Broker** برای برقراری ارتباط روی TCP ها listen می کنه

Broker مانند یک پل است چون قسمتی که تمامی پیام ها را از Producer دریافت می کند، سپس آن ها را در Log مربوط به Topic مشخص شده ذخیره می کند

Cluster

مجموعه ای از Broker ها می باشد که بصورت یک Cluster اجرا شده اند. این کار باعث بالا رفتن کارایی و تحمل خطا می شود.



Kafka broker ها stateless هستند

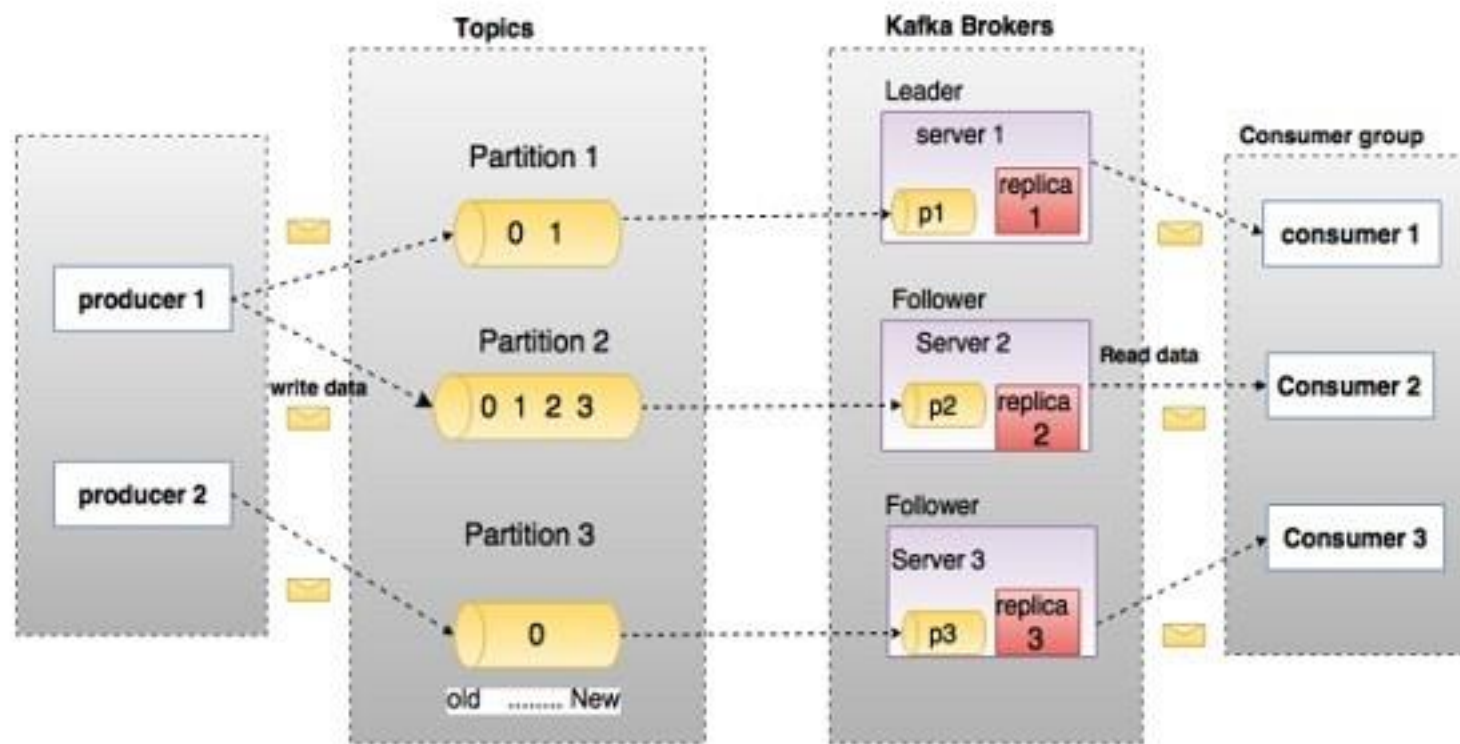
یک Kafka Cluster به شما امکان می دهد پیام ها را در یک پارتیشن مشخص تکرار کنید و در نتیجه این پارتیشن در چندین broker تکرار می شود

یک Broker کنترل کننده Cluster است و به صورت خودکار (توسط zookeeper) انتخاب می شود

Primary Broker

Kafka Server یک که مسئول خواندن و نوشتن در یک Partition است. در یک Cluster هر Partition در یک زمان تنها یک Primary Broker دارد. این Broker همزمان می تواند برای Partition های دیگر نقش Replicas را بازی کند. انتخاب یک Primary Broker برای یک Partition توسط ZooKeeper انجام می شود.

انتخاب Kafka broker leader توسط zookeeper انجام می شود.



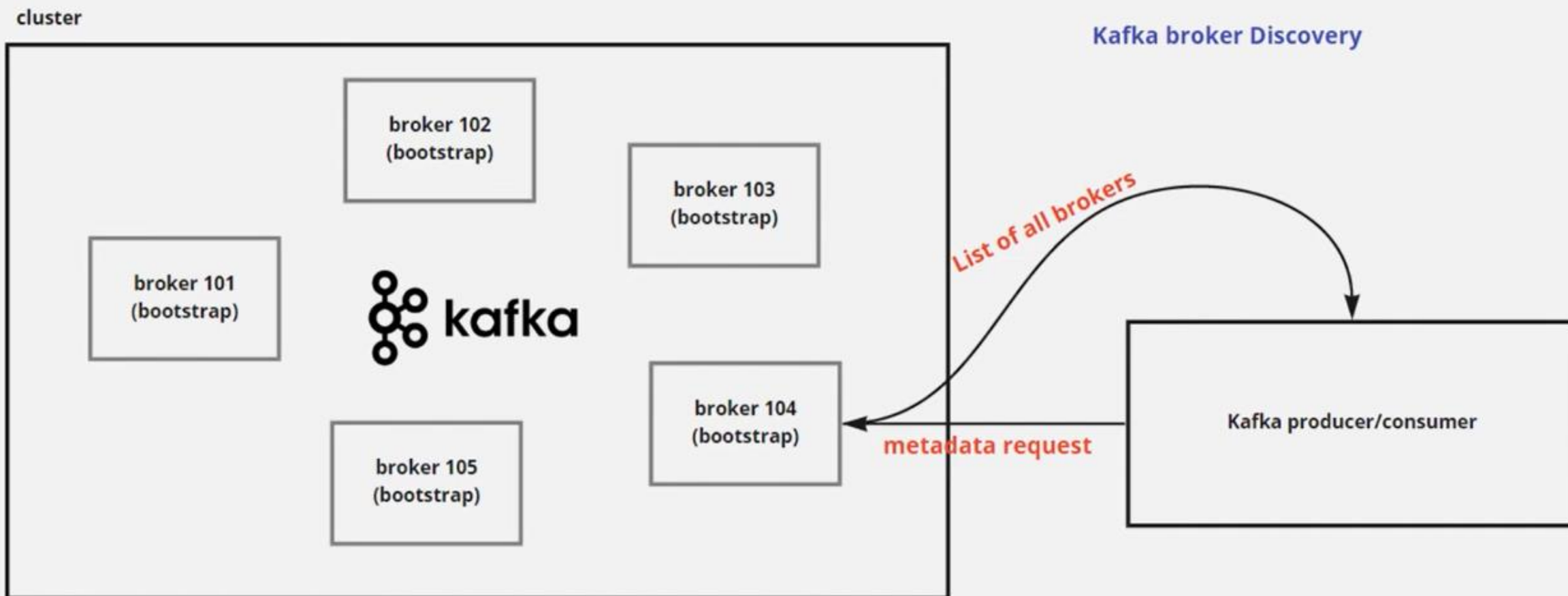
Replication Factor

این خصوصیت احتمال از دست دادن داده های یک Topic را به حداقل می رساند؛ به این صورت که هر پیام از یک Topic، در چندین سرور مختلف که تعداد آنها توسط این خصوصیت مشخص می شود، نگهداری می شود.

Replicas Brokers

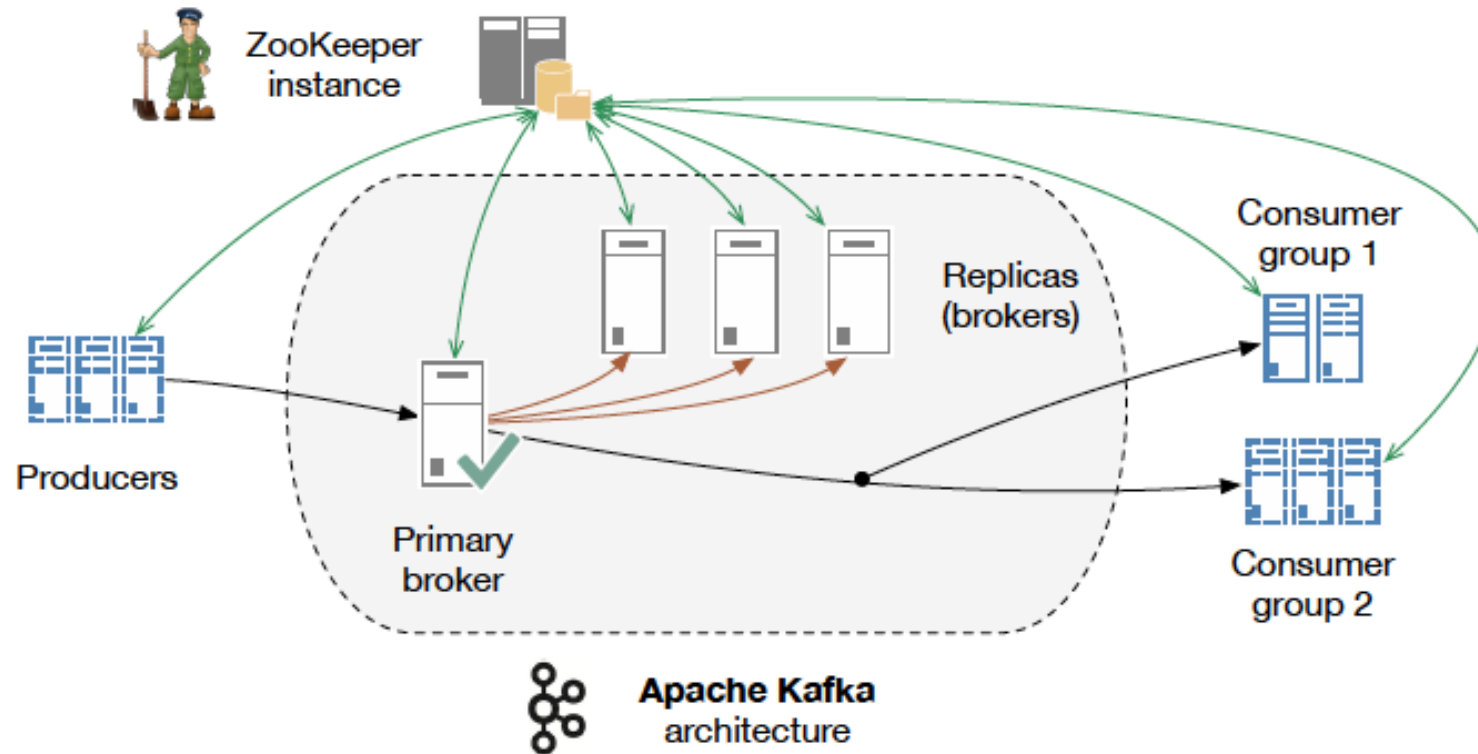
Kafka Server هایی هستند که شامل یک کپی از Partition می باشند. عملیات خواندن و نوشتن در Partition توسط Primary انجام می شود. در صورتیکه Primary از دسترس خارج شود، ZooKeeper یکی از Replicas Broker ها را بعنوان Primary در نظر می گیرد. همچنین این نکته را باید در نظر بگیرید که هر Replicate همزمان می تواند Primary پارتیشن های دیگر باشد.

در cluster هر broker در مورد سایر broker ها و topic ها و partition ها اطلاعات دارد که بهش می گوییم Meta data

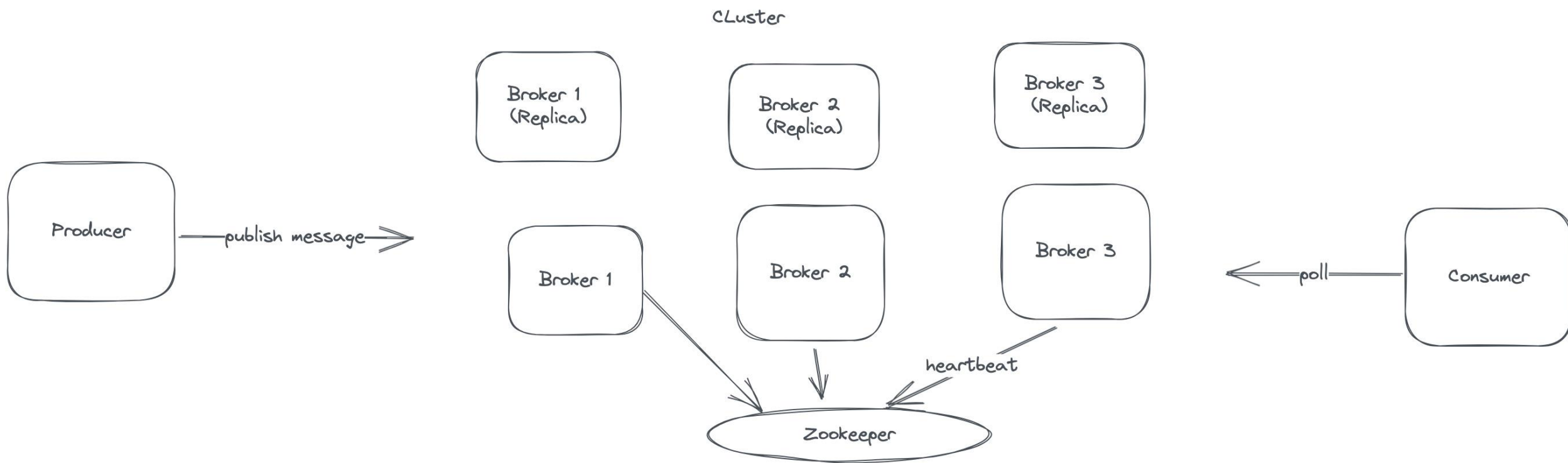


Apache ZooKeeper

Kafka هیچ State ای را نگه نمی‌دارد (اصطلاحاً stateless می‌باشد). برای ذخیره کردن و مدیریت تمامی State ها از جمله اینکه در حال حاضر Primary Broker برای یک Partition چه سروری است، یا اینکه پیام‌های یک Partition تا کدام Consumer offset ها خوانده شده‌اند یا اینکه کدام Consumer در حال حاضر در یک Consumer Group مسئول یک Partition می‌باشد، از یک سیستم پیکربندی توزیع شده به نام Apache Zookeeper استفاده می‌کند.



ZooKeeper به leader در cluster اطلاع می دهد تا مطمئن شود که مرده نیست. اگر leader بمیرد، ZooKeeper به انتخاب leader جدید کمک می کند تا اطمینان حاصل شود که broker بسیار در دسترس است.



Reliability Guarantees

کافکا 5 ضمانت برای کار خودش می‌دهد که عبارت اند از :

- 1- تمامی پیام‌های دریافتی در یک Partition از یک Topic، **به همان ترتیبی** که **دریافت** می‌شوند ذخیره می‌شوند. اگر پیام A قبل از پیام B نوشته شده باشد، با استفاده از همان تولید کننده در همان پارتیشن، کافکا تضمین می‌کند که offset پیام B بالاتر از پیام A خواهد بود. این بدان معنی است که consumer ها پیام A را قبل از پیام B خواهند خواند.
- 2- در یک Partition فعال، consumer ها تمامی پیام‌ها را **به همان ترتیبی که ذخیره** شده‌اند، دریافت می‌کنند.
- 3- در یک Topic با Replication Factor ای با **مقدار N**، **درجه تحمل خطا N - 1** می‌باشد.
- 4 - زمانی که پیام‌ها برای leader و همه نسخه‌های همگام‌سازی شده نوشته می‌شوند، «**committed**» در نظر گرفته می‌شوند
- 5 - consumer ها فقط می‌توانند پیام‌های **committed** را بخوانند

درک این ضمانت‌ها برای هر کسی که سعی در ایجاد برنامه‌های کاربردی قابل اعتماد با استفاده از کافکا را دارد، حیاتی است.

Advantages of Kafka

- Tackles integration complexity
- Great tool for ETL or CDC
- Great for big data ingestion
- High throughput
- Disk-based retention
- Supports multiple producers/consumers
- Highly scalable
- Fault-tolerant
- Fairly low-latency
- Highly configurable
- Provides backpressure

Disadvantages of Kafka

- Overwhelming number of configuration options
- Lack of mature client libraries other than Java or C (but things are improving)

Why is Kafka So Fast?

Kafka is known for its high performance and speed due to several reasons:

The two most important reasons are its low latency message delivery through Sequential I/O and Zero Copy Principle.

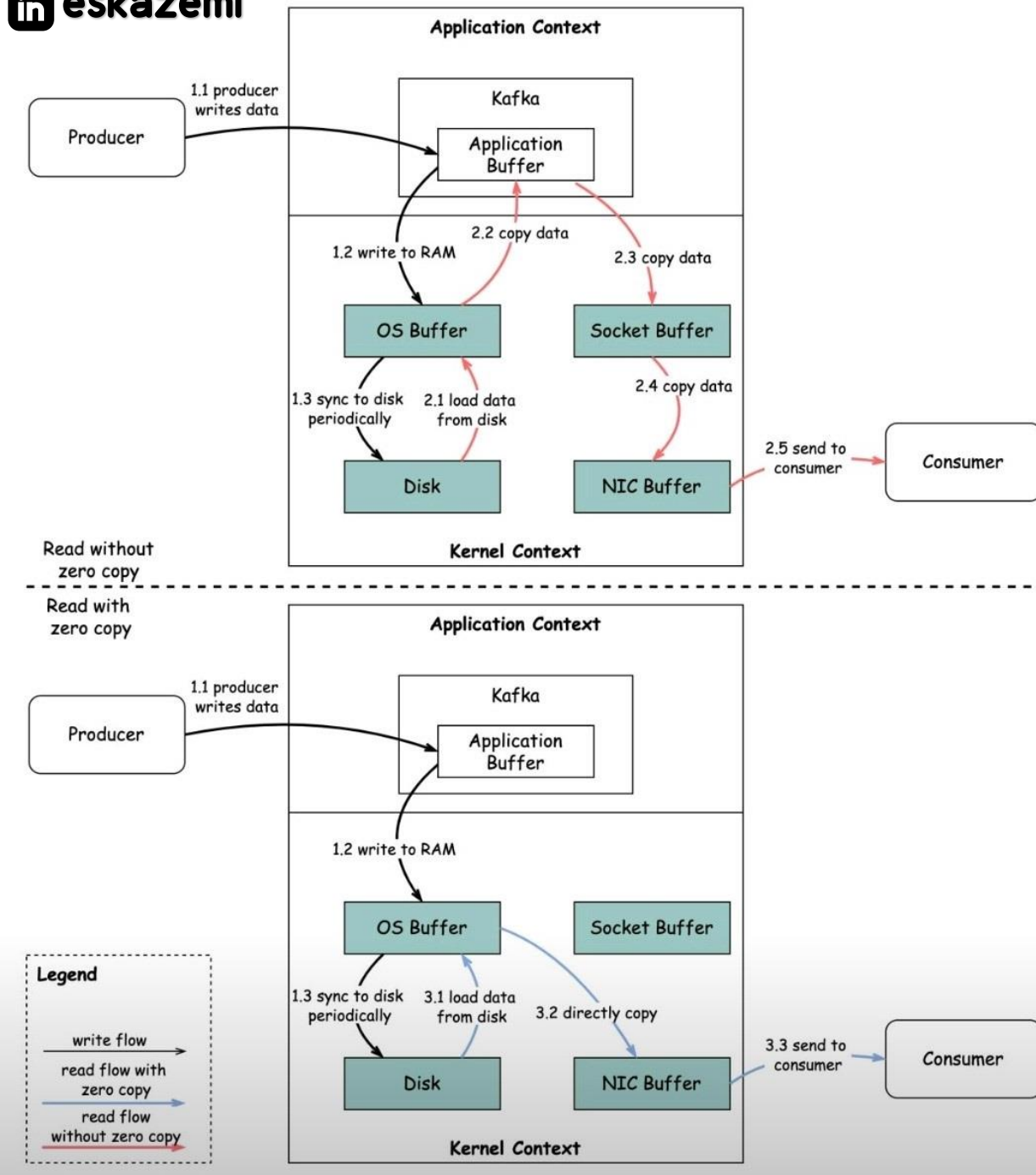
Sequential IO

Kafka relies heavily on the filesystem to store and cache messages. The general perception is that "disks are slow," meaning high seek time. Imagine if we can avoid seeking time. We can achieve low latency as low as RAM here. Kafka does this through Sequential I/O. Kafka efficiently uses a log, an append-only, totally ordered data structure.

ZERO COPY

To read a file from a disk, and then send it over the network, a traditional data transfer would require four context switches between user and kernel modes, making the data copied four times.

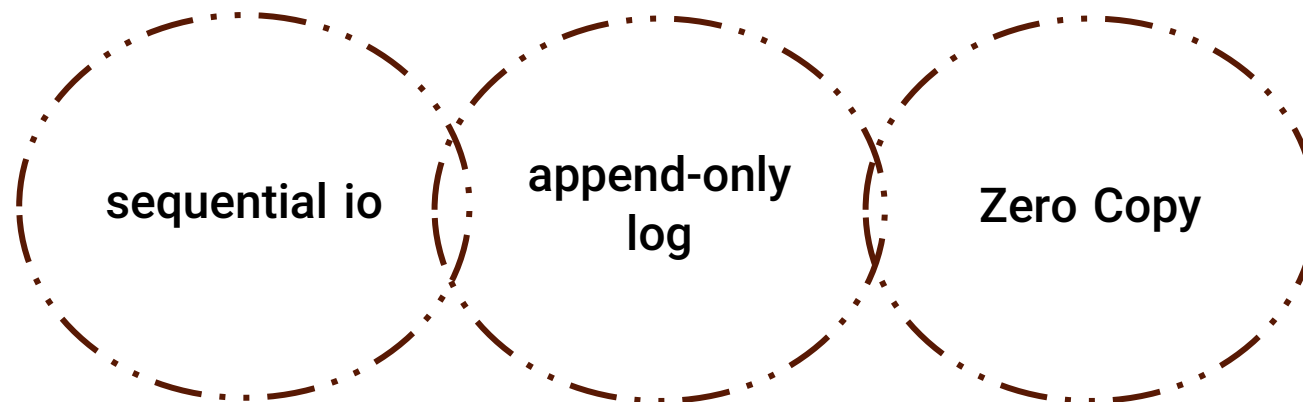
Kafka uses the zero-copy principle by requesting the kernel to move the data to nic rather than via the application. (Zero-copy optimization consists of removing the second and third data copies and transferring the data directly from the OS buffer to the nic buffer.)



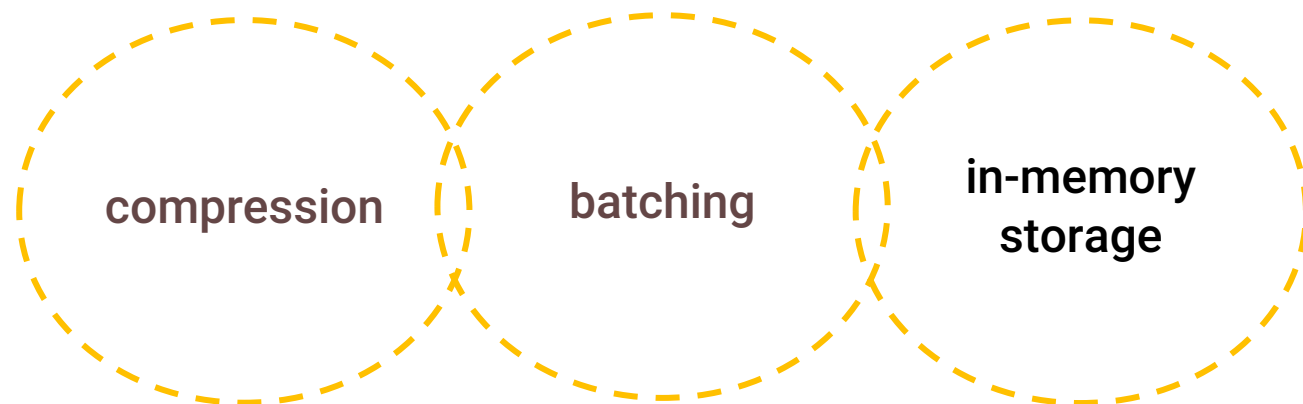
Message Compression

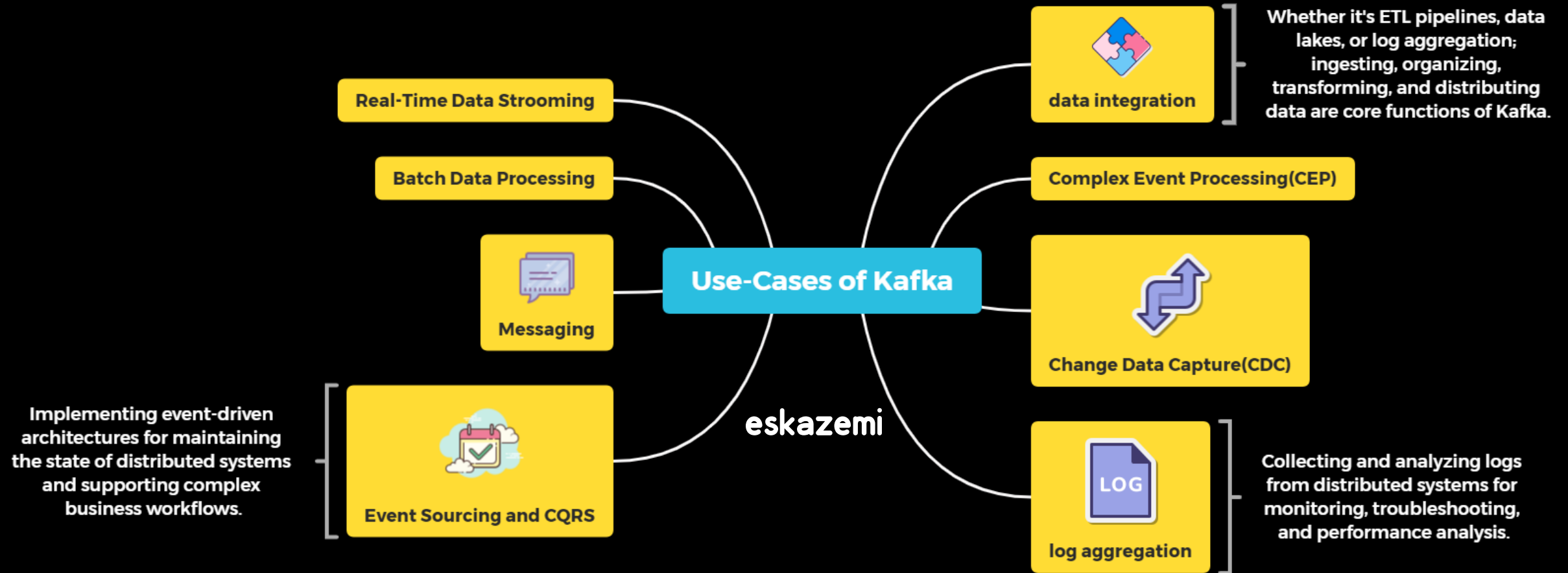
Kafka allows for message compression, which reduces messages' size and enables faster data transmission and processing. Message Batching: Kafka can batch messages together, which reduces the overhead of individual message processing and improves throughput.

به طور کلی، سریع بودن کافکا ناشی از موارد زیر
1- معماری بهینه شده آن شامل 3 مورد زیر



2- فشرده سازی 3 - دسته بندی 4 - ذخیره سازی





در مباحث **Big data** معمولا با دو چالش اصلی مواجهیم

1- جمع آوری حجم زیادی از داده ها

داده ها شامل :

داده های مربوط به رفتار user ها

Event messages

و...

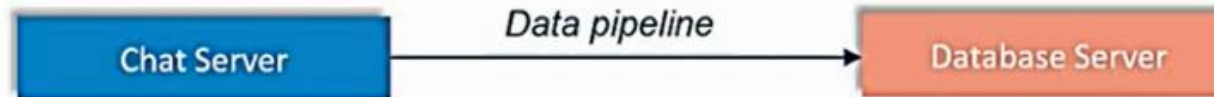
2- تحلیل داده های جمع آوری شده به صورت real-time می باشد.

لازمه ی غلبه بر این دو چالش استفاده از یک **messaging system** همچون **kafka** است.

Real-time streaming data pipelines

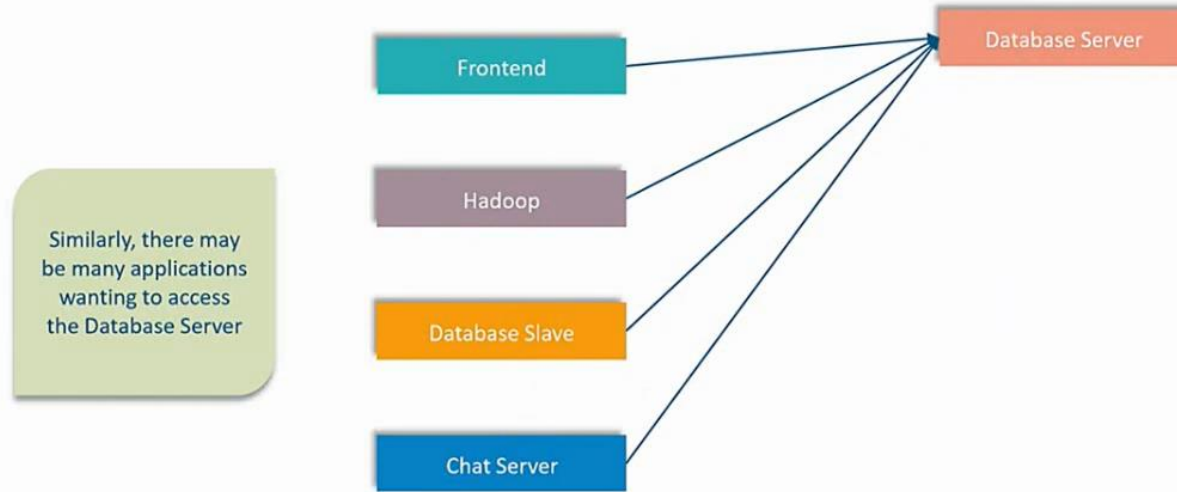
پایپ لاین داده

Communication is required between different systems in the real-time scenario, which is done by using data pipelines.



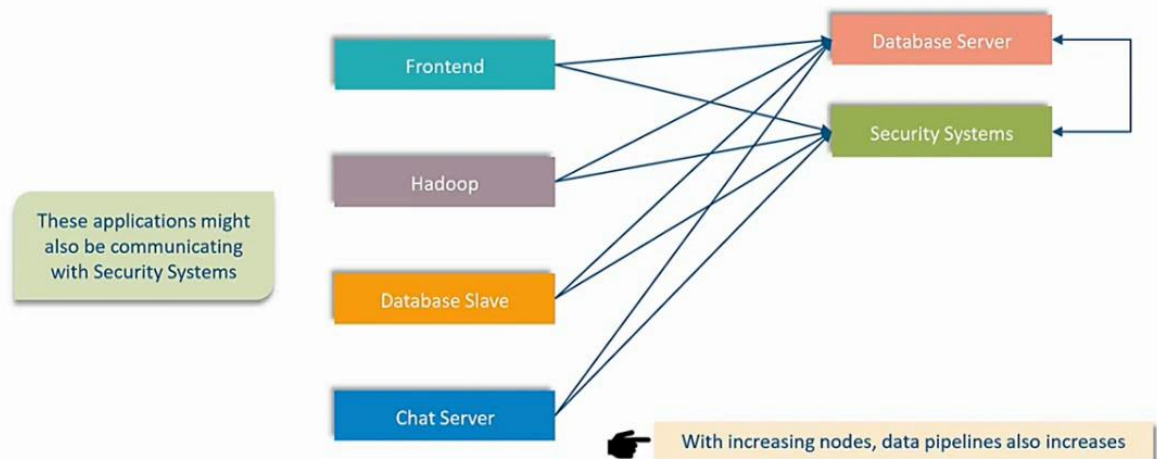
For Example: Chat Server needs to communicate with Database Server for storing messages

افزایش در تعداد نودها



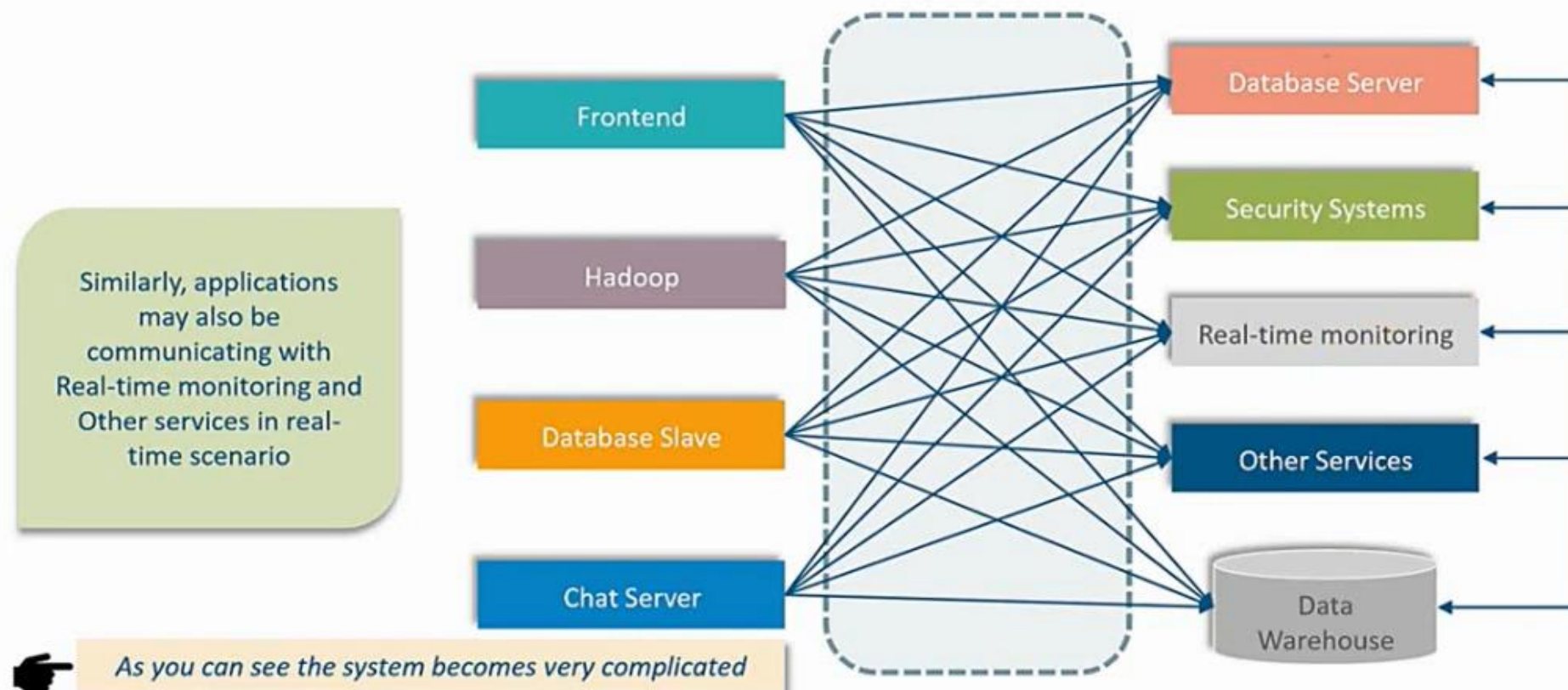
فرض کنید شما تعداد زیادی سیستم و زیرسیستم مختلف را داشته باشید که هر کدام از آنها نیازمند ارتباط با برخی از قسمت‌های دیگر است. در این صورت شما دو راه دارید: اول اینکه در هر قسمت سرویس‌هایی را برای ارتباط با سایر قسمت‌ها پیاده‌سازی کنید یا اینکه هر قسمت بصورت مستقیم با سایر قسمت‌ها در ارتباط باشد.

افزایش در تعداد نودها

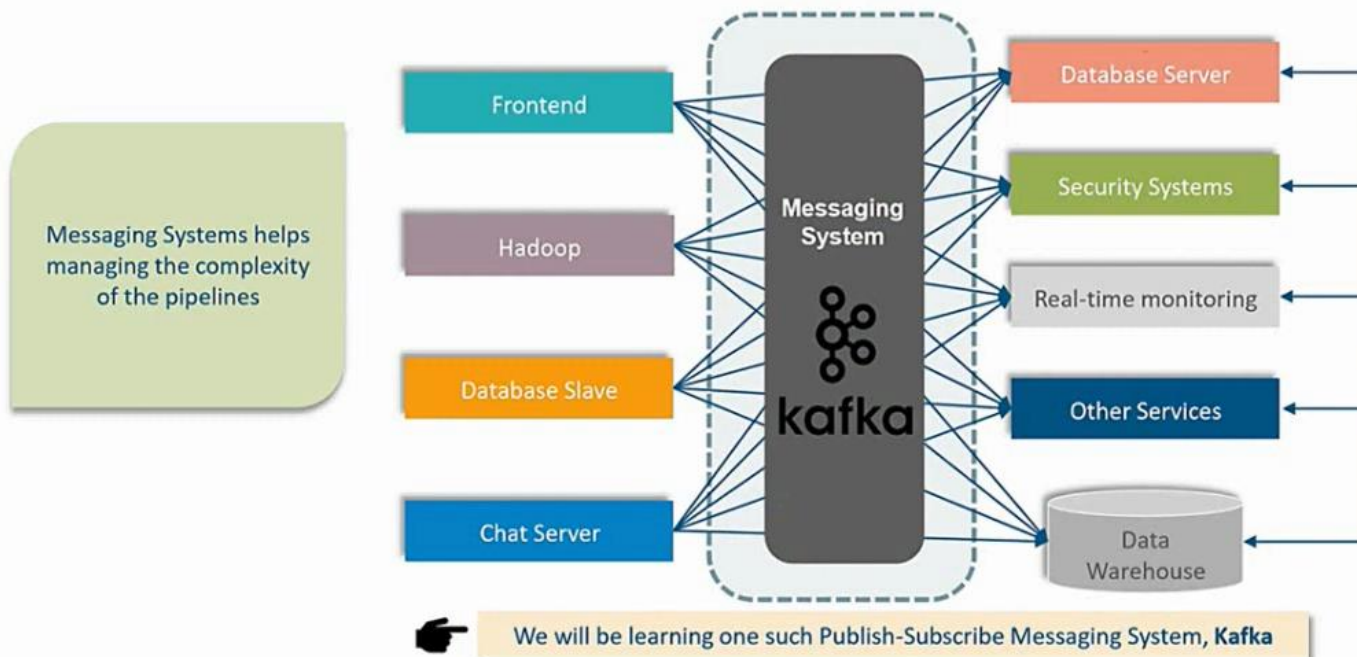


مشخصاً کنترل و مدیریت جریان اطلاعاتی در این پیاده‌سازی کار بسیار دشواری است. تغییر هر قسمت، تاثیر مستقیمی بر روی سایر قسمت‌ها دارد و در صورتی که هر یک از قسمت‌ها با مشکلی روبرو شوند، سایر قسمت‌های مرتبط نیز با مشکل روبرو می‌شوند. این مشکل زمانی بسیار نمایان می‌شود که در معماری‌هایی مانند میکروسرویس، بدلیل بالا رفتن تعداد زیرسیستم‌ها و ارتباطات آنها، مدیریت این ارتباطات کار بسیار دشوار، پرهزینه و پیچیده‌ای می‌شود.

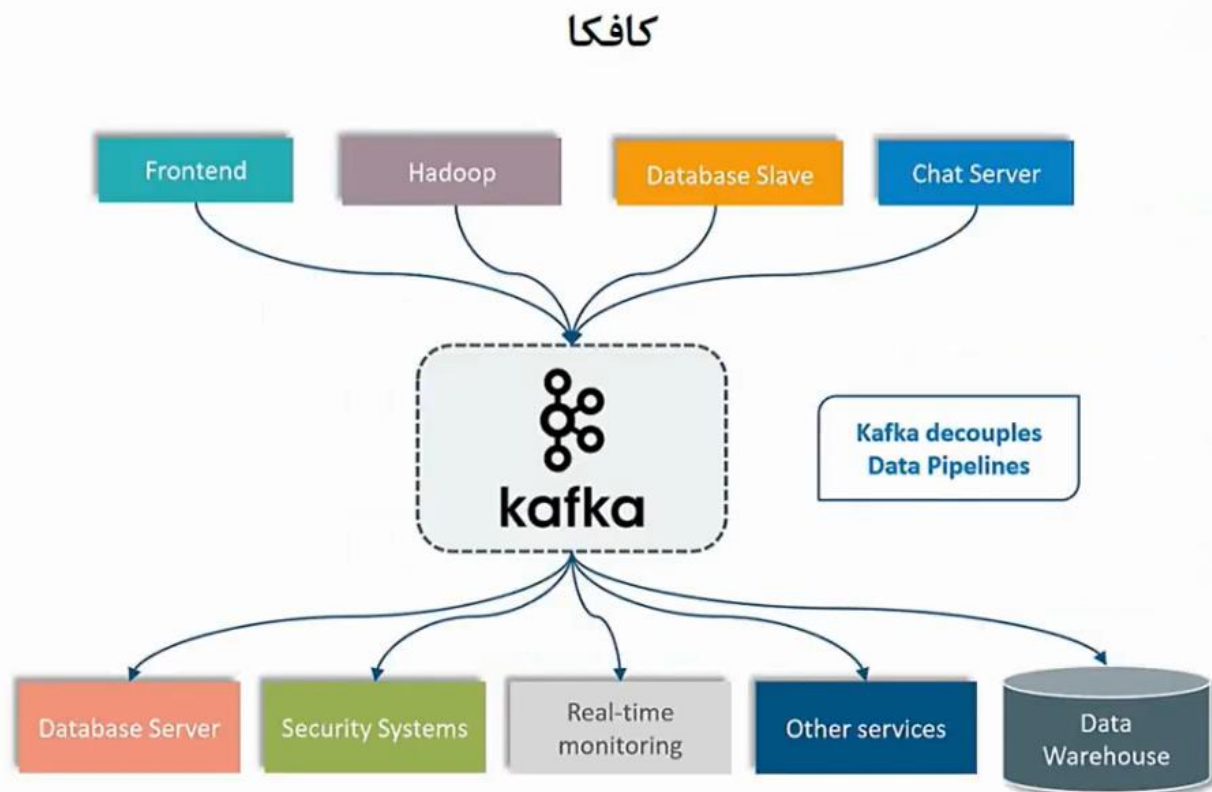
پایپ لاین‌های پیچیده



راهکار پایپ لاین‌های پیچیده



روش Apache Kafka برای رفع مشکل پایپ لاین‌های پیچیده به این صورت است که ارائه یک بستر برای مدیریت و کنترل جریان‌های اطلاعاتی با کارایی بسیار بالا، در سیستم‌ها و زیرسیستم‌های مختلف است. یعنی شما می‌توانید با ایجاد کردن یک Pipeline برای جریان اطلاعات خود، وابستگی مستقیم سیستم‌ها و زیرسیستم‌ها را از بین ببرید؛ آن هم به صورتی که بروز مشکلی در هر قسمت، کمترین میزان تاثیر را در سایر قسمت‌ها داشته باشد.



همانطور که می بینید دیگر نیازی نیست تا قسمتهای مختلف بصورت مستقیم با یکدیگر در ارتباط باشند؛ تمامی ارتباطات از طریق Kafka انجام می شود. تغییر یک قسمت، تاثیر زیادی بر روی سایر قسمتها ندارد یعنی دسترس خارج شدن یا بروز هر گونه مشکلی در یک قسمت، بر روی کل سیستم تاثیر زیادی ندارد. پیامهای مربوط به یک قسمت تا زمانی که پردازش نشده اند از بین نمی روند؛ پس سیستمها می توانند در حالت Offline نیز به کار خود ادامه دهند. شما می توانید در این روش تمامی قسمتهای سیستم را بصورت یک Cluster پیاده سازی کنید. در اینصورت احتمال از دسترس خارج شدن هر قسمت به کمترین میزان می رسد. حتی در صورتی که یک قسمت بصورت موقت از دسترس خارج شود، پیامهای مرتبط با آن قسمت تا زمانی که دوباره به جریان پردازش بازگردد، از بین نمی روند بلکه پس از اضافه شدن قسمت از دسترس خارج شده، بلافاصله تمامی پیامهای مرتبط با آن قسمت برایش ارسال می شوند.



 eskazemi