Load balancing and early fault detection in Apache Kafka clusters

Università Ca' Foscari Venezia, Italy

Software Dependability and Cyber Security

Dario Burato 843238

Supervisor Prof.Andrea Marin

Academic year 2018/2019

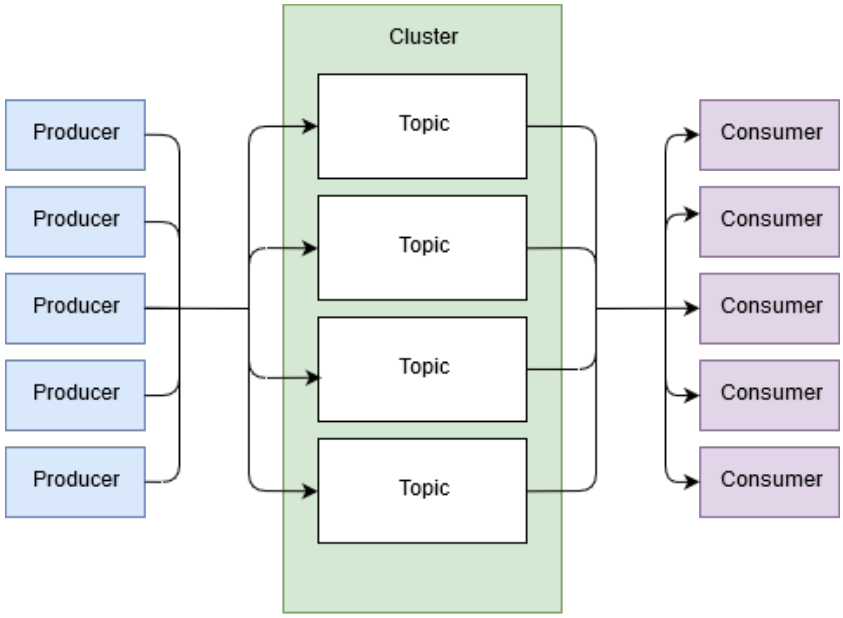
Abstract

Apache Kafka is a publish-subscribe message system, producers publish data on a cluster, from which clients subscribe to receive data. The messages are sent by their producers and stored in partitions. The load balancing is performed thanks to the data distribution among each cluster's node. The component that assigns a message to a partition is called partitioner, and every producer contains one partitioner. When partitions lack intrinsic meaning, and are used purely for load-balancing purposes, the default partitioner available with Apache Kafka aims only to get the same amount of messages shared among partitions. The most common Apache Kafka cluster configuration is based on multiple identical systems that can be changed, even at run-time, on purpose or by faults. Even if re-balancing tools exist, it would take time to properly adapt to an heterogeneous cluster configuration. The balancing issue is caused by the partitioners focus on partition's data amount, rather than node's. The problem could be solved by changing the amount of partitions in each node, to level the data/node ratio, thus tricking the default partitioner logic, however this may actually hurt client performance. A proper partitioner that infers the performance of each node is the desirable solution. This work presents an algorithm to detect problematic scenarios and customs partitioners that adapt to them.

Contents

Introduction

The core function of Apache Kafka is very "simple": it handles receiving, storing and delivering messages from multiple producers to multiple consumers [17]. The data which transit through Apache Kafka, is saved within topics (logical feed name/labels). The producer decides the target topic, and the consumer decides which to read from [10]. Producers see the cluster running Apache Kafka as queues to append message. The consumers, on the other hand, see queues to pull data from. Indeed this is the reason why software with these functionalities are commonly called Message Queuing Services [36].



In software architecture this communication model is called publish-subscribe pattern, referring to the act of topic selection performed by the entities at both ends of the system.

Even for a relatively young software, eight years (first release January 2011) Apache Kafka is widely used, even by big names of IT, like Cisco [27], Linkedin [1], Twitter [8], Cloudfare [5], Netflix [25] [38], the New York Times [33], Apple [22], PayPal, Spotify [3] and Uber [41]. Apache Kafka has rapidly defeated the competitors thanks to its simplicity, scaling possibilities and ease to adopt. The high adaptability comes with a cumbersome drawback, working out-of-the-box in almost any situation force the existence of a tuning phase to configure every installation (or expect non-optimal performance).

Objective of the thesis

This thesis will discuss how Apache Kafka, a software written by The Apache Software Foundation, works, how it performs under high stress or partial failure, and how to improve it. Edge use cases [6] reveal problems on how Kafka perform data load balancing, analysing its internal structure and design will make clear where the issue is located. The only available solution to data distribution problems is to setup correctly a cluster from the start, planning how it will react to topology changes.

The Partitioner, the key component in charge of splitting data between each node, will be isolated, studied and compared with new implementations. The comparison of each solution is done by benchmarking, with a test cluster of dedicated machines, by code-logic reviews and asymptotic analysis (classification by Big O notation). The final objective is to show how more complex partitioners, and so producers, can solve the balancing problem without losing the configuration "simplicity" that Apache Kafka is known for.

Structure of the thesis

This thesis is split in three sections:

- Chapter 1, 2 and 3, explain what Apache Kafka is, how it works, which problems it has and how to detect them in the early configuration stage of a project.
- Chapter 4, 5 and 6, explore the state of the art regarding resource management, scheduling and load balancing and ways to apply bits of it to Apache Kafka specific case.
- Chapter 7 will compare the performance of the default settings found in the Apache Kafka software with two new policies implemented on the basis laid in Chapter 6.

In specific, the first Chapter explains the Apache Kafka features, its design structure and introduces the data distribution problem. The second Chapter is focused on the only partitioner shipped by default within Apache Kafka, and analyses its behaviour in unhealthy clusters. The third Chapter shows how set theory formalizes the problematic cluster topologies found in the second Chapter, and how to detect them before a cluster goes live.

The final chapter compares the current Apache Kafka software with each new policy proposed. A test cluster was set up, used as benchmarking to drawn parallels to real world bigger scenarios. The results will confirm the hypothesis formulated in Chapter 6, having advanced and smarter producers ease the difficult configuration process of clusters and improves the overall Apache Kafka performance, especially when faults occur.

Chapter 1

The Kakfa Infrastructure

1.1 Introduction

This chapter will explore how Apache Kafka has been implemented, and for which reasons it has been designed in that way. To satisfy the high demand of premiues on load balancing and performance problems, only the basic Producer/Consumer paradigm will be explored and discussed. More advanced APIs [31] exposed/offered by Kafka are aimed for specific programs/cases, but use the same base structure and core-concepts, so they will not be taken into consideration.

A specific component, the partitioner, reveals that Apache Kafka has a load balancing problem rooted in the overly simplistic nature of its structure.

1.2 Features

Apache Kafka provides all enterprise level features [26] [37] [12] expected in any modern message system solution:

- **Redundancy**. Data replications capabilities for handling failures, even on a big scale if the cluster is configured to expect them.
- **High-Volume data handling**. There is no difference between small data bursts and long big streams, the software has been designed to cover any kind of data flow. Hardware bottlenecks should be hit before software ones.
- **Communication Scalability**. The cluster, as single entity, manage to have a PTMP (Point to Multi-Point) communication with both producers and consumers, this is possible since multiple node resides in the cluster, sharing a split of the workload.
- **Durability**. The usage of distributed, and replicated, commit logs ensure persistence of data on disks.
- **Fault Tolerance**. When faults occur, designated backup nodes take the role of leaders getting zero, or negligible, downtime, invisible to users.

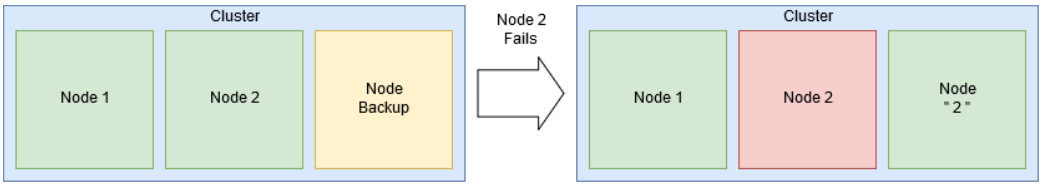


Figure 1.1: Cluster with a faulty node recovers thanks to a backup node

- **Event Replication**. A cluster can act as producer itself, to collect, replicate, modify and/or create new events.

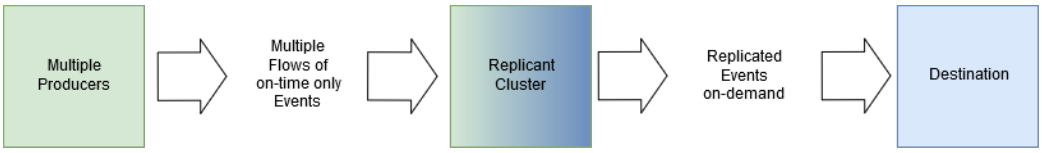


Figure 1.2: Events replication basic schema

- **Dynamic topology updates**. Any change in the cluster topology causes updates to be forwarded live to each entity of the system. Hence the system automatically adapt to them.
- **Work Auto-Save**. Producers and consumers remember how far in the log they went. In the unlucky case of a crash, the position information is not lost since is also tracked by the cluster.

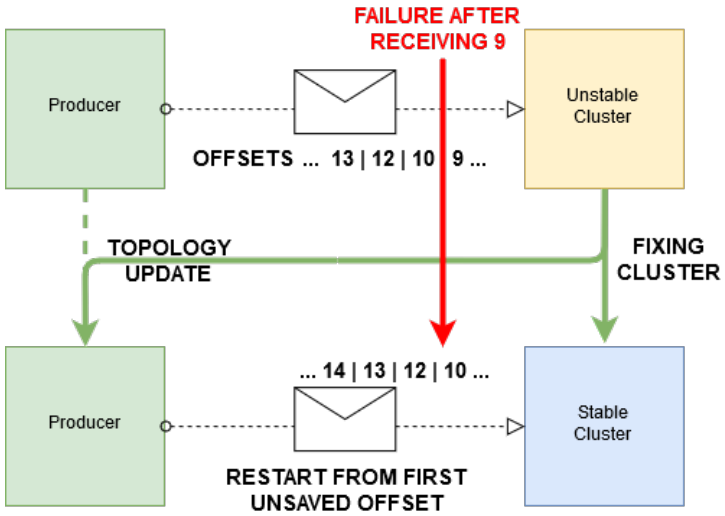


Figure 1.3: A producer waits cluster stability before trying to re-send data

- **Reliability** comes as consequence of all the previously listed features.

1.3 Use Cases

Covering all the possible ways in which Apache Kafka can be, and is, used is not the aim of this thesis, but it is useful to know how is deployed in production environments, to better understand how it works internally.

- **Message Delivery** from multiple sources to multiple destinations [36], the core functionality of Apache Kafka. Common alternative solutions in this field are RabbitMQ [16] [11] or Active MQ (MQ is an acronym for Message Queue).
- **Metrics collection** of multiple monitoring apps, aggregated in a single coherent feed [14].
- **Stream processing**. The act of transforming incoming data, making Apache Kafka acting as aggregator and translator of information, later forwarded into other applications for further processing [16].
- **Commits/Events log**. The ability to store any step of a process to later recreate the same event for replay purposes. Important where the state of an application is stored as a chronologically ordered sequence of records [16], like a text editor "undo" and "redo" functions or a DBMS diff. history after each database update.

1.4 Code Design

A big set of feature is not a synonym with "code heaviness" (and with "code bloated performance"), Apache Kafka code provides each promised functionality without renouncing to high performance. Aside from the simple data replication and distribution, good performance is possible thanks to few key design choices, not always implemented by the competitors [16] [11]:

- **Zero-Copy**, a programming technique used to avoid useless middle buffers in application memory, when the work is already backed by kernel/OS level buffers [32]. It must be supported by the full stack of technology were the code runs (hardware, kernel, OS, API, ...), but it drastically improves latency and throughput in both local I/O (to ram, to disk, ...) and network communications. This approach is also capable of DMA-esque (Direct Memory Access) features, getting the data delivered with no CPU interaction, eliminating context switches and additional data copies [28].

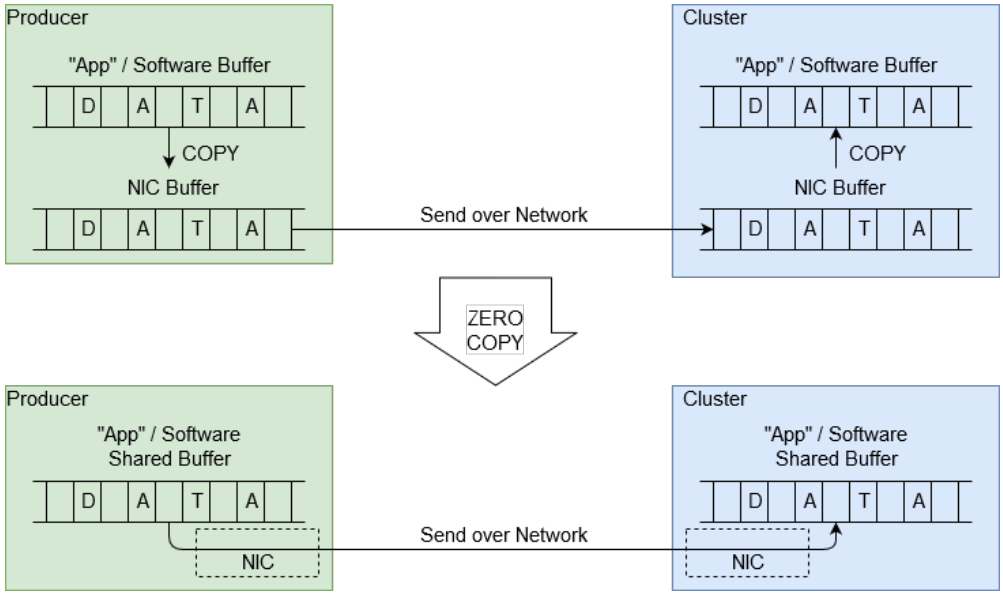


Figure 1.4: Usage of buffers comparison with and without Zero-Copy

- **Sequential operation** pervades the whole system, negating the real need of dedicated coded cache mechanism, relying on the built-in OS cache features, more tested, robust, highly configurable and with exploitable features in ordered reading scenarios (like read/cache-ahead).
- **Queue data structure**, used to represent the logs that store data in Apache Kafka. Everything new is always appended.
- **Batch processing** of multiple data, lowering the overall number of network requests. Batch compression can achieve good compression ratios without hurting latency if the data split sweet spot is found.

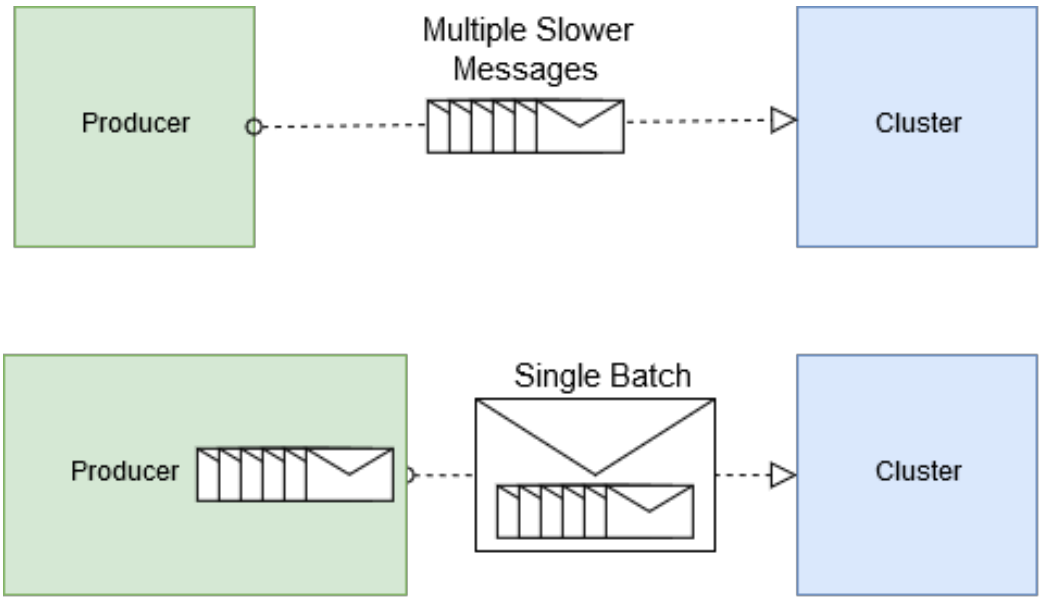


Figure 1.5: Single batch of multiple message example

- **Standard binary format** perfectly fitted for batch processing and sequential I/O operation, leveraging completely the Zero-Copy approach. No transformation needed when switching communication channel/media (to network, to disk, to memory).

1.5 System Design and Structure

Now that we know the basic entities that compose an Apache Kafka system, we must understand how they interact among each other. Three concepts should be kept in mind to understand why Apache Kafka is built in this way:

- The common and ideal configuration of a cluster assumes that each node has the same performance (same machine). Heterogeneous configuration are allowed but require some tuning.
- Producers and consumers are part of the system, designed together with it, and not external foreign entities that interacts with unpredictable behaviors.
- The cluster topology is shared and known by each entity (and dynamic updates to it, both by legit changes or random failures).

1.5.1 Storing Data

A topic is a logical label or feed name used to group records, and are divided in one or more partitions, that is why Apache Kafka is said to maintain a so-called partitioned log [39]. Records are published to a selected partition, appended to it and marked with an incremental offset.

Topic	Logs with #offset					
Partition 0	0	1	2	3	4	
Partition 1	0	1	2	3		
Partition 2	0	1	2	5	6	7
Partition 3	0	1	2			
	Older		→			Newer

Apache Kafka has two policy for log retention, deletion and compaction, but they work with the same base parameter, a configurable maximum age that records can have (by default set to two days). The retention policies tackles storage capacity problems that each node may have, but they are in place mainly because there is no consume-on-read action by any consumer.

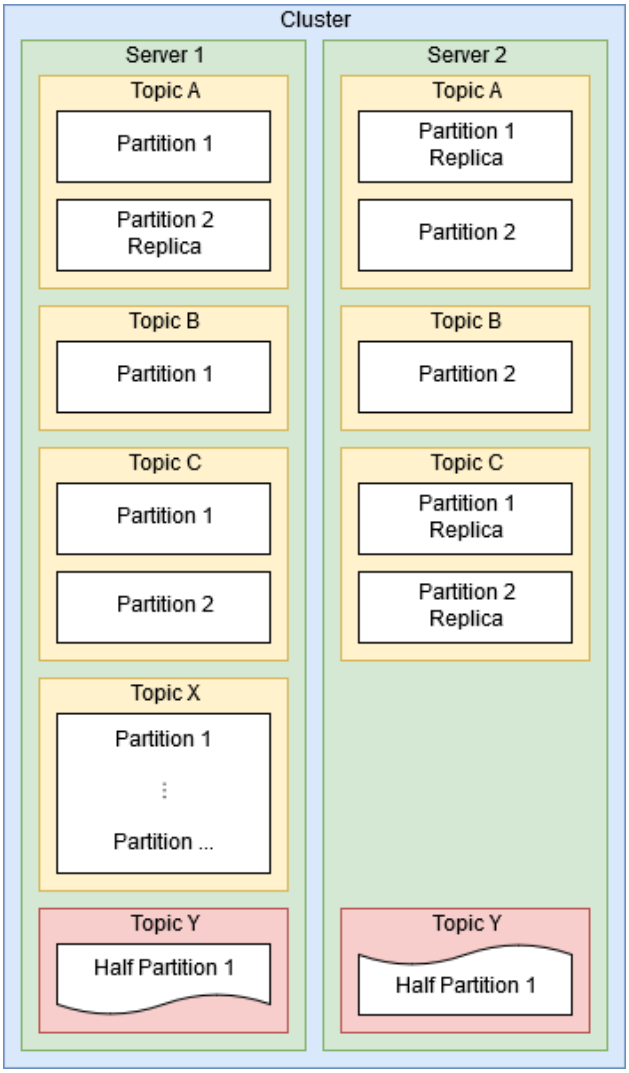
Topics are multi-subscriber, and so partitions, which are defined to be immutable. This is why consumers only read and never delete afterwards, since it would cause conflicts with other consumers reading next/previous records.

Apache Kafka keeps track of each consumer offset (with an internal topic hidden by the user), but it is the consumer itself controlling it. Thus jumps between records are permitted.

1.5.2 Distributed Data Store

While the topics can be, and commonly are, assigned/shared by multiple nodes, partitions cannot. Partitions can be entirely copied/replicated between nodes (with the same topic), but any single partition cannot be split between different nodes. Per se data distribution (usage of partitions) achieve consumer and producer load balancing, while redundant data distribution (partition replication) gives fault tolerance.

The following figure helps to visualize different cluster configurations (with only two servers for drawing simplicity). Not all combinations are shown, but most can be derived by mixing the first examples.



Focusing on each single topic configuration:

Topic A - 2 servers, 2 partitions, 1 replica per partition

Topic B - 2 servers, 2 partitions, no replicas (no fault tolerance)

Topic C - 2 servers, 2 partitions, originals all on same server (no load balancing)

Topic X - Assigned only to one server, one or more partitions and no replicas (B and C cons. combined)

Topic Y - This cannot be done

In production environments we rarely see B, and almost never C or X, but nevertheless those are still valid configurations.

1.5.3 Fault Tolerance

As previously stated, in Apache Kafka partitions replication on multiple different nodes is possible and achieve fault tolerance. Each partition has, at any given time, only one leader node (the node holding the original partition) and as many followers (the nodes holding a replica) as specified by the replication factor. The replication factor of a cluster cannot be greater than the total number of nodes assigned to a single topic (it makes no sense to replicate a partition X times with less than X nodes).

When a cluster is healthy and all nodes are properly working, a leader is the only handler of read/write requests coming from consumers/producers. Followers communicate with the leader to update their content to the last available, follower which have already completed this task are called ISR (In-Sync Replica).

A producer, depending on its configuration, considers a record to be successfully sent only when it has received one, some or all (another cluster configurable variable) write confirmations from the leader and the followers.

When a node stops working the whole cluster is notified, for each partition he was leader of a new election is made between its followers. Any topology change is communicated also to each consumer and producer, so they know where to send requests.

1.5.4 Load Balancing

In Apache Kafka the data generated by the producers affect the whole system in two phases. First the servers receive the workload from the producers, and later the consumers request it from the servers. Analyzing the full life flow of data helps understand where the hard work is being done. Partitions are the key to achieve a good load balancing [20].

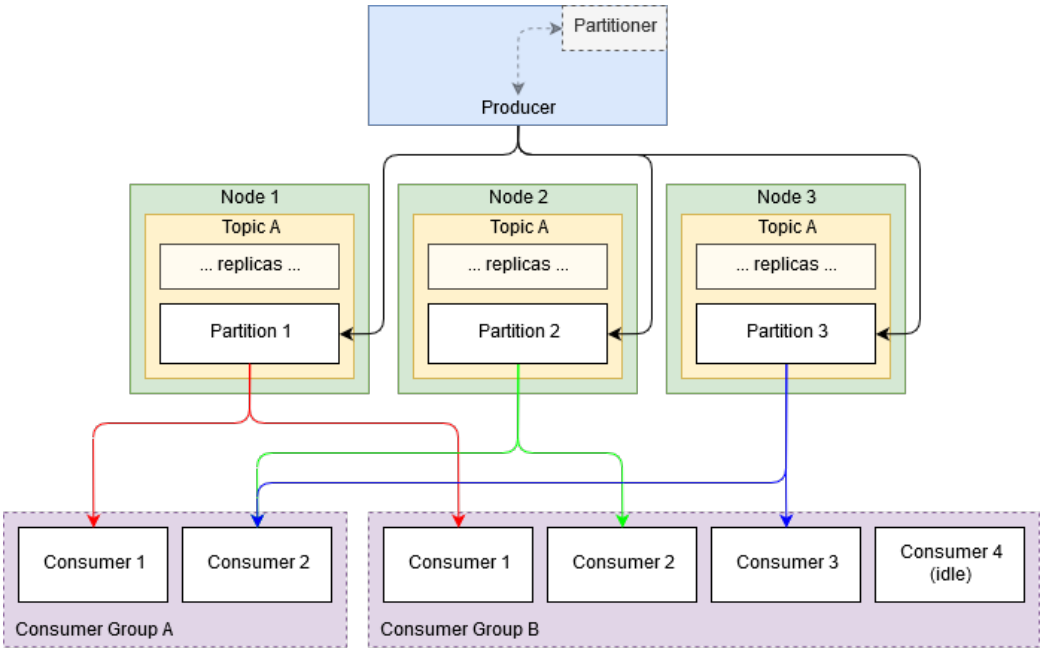


Figure 1.6: Complete view of a simple Apache Kafka system

The simple cluster shown in the figure has only three nodes with just one topic assigned to all of them. Each server is leader (p.10) of just one partition of the only topic.

The producer, knowing the whole cluster topology, asks its partitioner in which partition to write. Then, it contacts only the leader node of the selected partition. The leader node does the first phase of work by receiving data (from multiple sources concurrently), writing it to disk and handling replicas updates (as illustrated in previous paragraphs).

Before sending new data, the producer waits for a configurable number of successful write acknowledgments from the replicas (by default all).

The second work phase is done by each consumer inside its consumer group, by receiving only a split of the requested data from the server. While reading from any topic, groups with less consumer than partition will have some consumer more loaded (in the image "Group A - Consumer 2"), while on the other hand surplus of consumer leads to partial group idling, but provides consumer fault tolerance (in the image "Group B - Consumer 4").

Consumption is balanced among the consumers assuming that a "fair share" of data reach each consumer, this is possible only with the "design everything together" mentality; producers, consumers, nodes and partitions are engineered together, by design highly unbalanced scenarios should not be conceived in production environments.

To get the best performance the data distribution fairness assumption must be granted (between partitions), the component responsible for it is the partitioner.

1.6 Assumption on the Data Distribution

The official guide on Apache Kafka does not have any peculiar focus on how to obtain the perfect balance, even if it is implicitly intended that has to be pursued. Thanks to its simplicity and easiness of use and code for, Apache Kafka has the advantage to have been quickly adopted and adjusted in any production scenario, but, on the other hand, leaves uncovered areas on how tuning its performance for various situations.

There is no AI or statistical process that automate the value selection for every possible variable which controls how Apache Kafka behaves. Manual testing is needed to perfectly fit a cluster to any specific circumstance, and this can be seen by the wide broad of powerful testing tools provided out-of-the-box with any Apache Kafka installation.

Perfect adaptation must be discussed on case-by-case basis, but this should not lead to believe that there is no common ground for improvements. A shared problematic reality comes from failures and recovery from it. The assumption that data is evenly distributed between each node is often not true in reality, and not by having unpredictable producer behavior (which is excluded a priori from the official guide), but by not taking in consideration topology updates, mainly from failures.

Even the most distributed cluster can be greatly affected by small random failures. A shift in the topology could cause a node to handle more partitions and work than what can handle, possibly reaching its hardware limits and bottlenecking every topics touched by it.

How can an advanced system, with shared known topology, fail to balance data distribution? Performance issues related to wrong balancing can be originated by the following conditions:

- The node which will receive the unbalanced quantity of work is not properly scaled to handle fault scenarios. This could reveal a flaw in the initial design phase, but actually is only a matter of scale. How big a fault has to be to cripple a topic performance?
- The partitioning policy does not take in consideration any metric from which can infer single node performance.

The first condition could be solved by adding more nodes to a cluster or improving the power of each one, but this solution may be completely wrong (and possibly have a considerable financial impact). It is imperative to remember that performance does not always scales linearly with the unit of parallelism in a system, or its power. Doubling the power of each node does not ensure that a cluster can handle double the work with the same results, for the same reason there is no certainty on being able to handle half the load when half the nodes go offline.

The second condition is always met. The only partitioning policy shipped with Apache Kafka, has the benefit to be fast and has close to zero memory impact, but does not consider any performance metrics, relies only on assumed good health of the cluster and fast recovery from faults. The next chapter (p.17) is dedicated to the Apache Kafka partitioner and deeply illustrates its caveats.

1.7 Keywords

Working with Kafka require a basic knowledge of its most common terms. The following list help the comprehension of this document.

- **Cluster** - Group of one or more **Nodes** running an instance of Apache Kafka.
- **Node** - One server/machine running an instance of Apache Kafka and participating in a **Cluster**. When all nodes are online and no faults occur the **Cluster** is said to be "healthy".
- **Record** - Data/Message consisting of key, value and timestamp, that will be sent, received and stored in a **Cluster**.
- **Topic** - Category/Label/Feed/Name to which **Records** are published. Stored in a **Cluster**.
- **Partition** - Partition of a **Topic**.
- **Consumer Group** - Who will subscribe to one or more **Topics** in a **Cluster**. Even if this entity is a group of **Consumers**, in the "common client-server logic" this is seen as a single client, with each **Consumer** acting like a separate worker/thread/process. A group is composed of one or more machines running one ore more threads, joined together by the same group ID.
- **Consumer** - Each entity of a **Consumer Group**. It receives a split of the data (and so work/load) coming to its **Consumer Group**.
- **Producer** - Sends **Records** to a specific **Topic** inside a **Cluster**.
- **Partitioner** - Tells the **Producer** in which partition each **Record** should be sent.

1.8 Conclusion

Apache Kafka fundamentals have been explored, it is clear what this software has to offer and why is appealing to big companies in the IT filed, richness of features, fast adaptability and outstanding performance makes anything a top tier choice.

The main weakness that pervades the whole approach to Apache Kafka, is to put too much faith in good and proper configuration of each installation. Operating on real systems may reveal new previously unknown problems that the theory would have never encountered.

The logic which controls were to send data is called Partitioner, is a component inside each Producer and is queried by them before sending any records. Assuming that the data is always well distributed and balanced is wrong as assuming that faults can always be controlled no matter the scale and impact.

Chapter 2

Apache Kafka Partitioner

2.1 Introduction

This chapter will explain how the only partitioner shipped by default in Kafka distribute data within a cluster. Analyzing the default policy reveals why some early code design choices were made and which configurations suffers from said choices.

Faults are the big overlooked problematic that can drastically change the distribution of data within a cluster, and the default partitioner does not take them into consideration [40].

2.2 The Default Partitioner

Any test tool, app, code or, more in general, producer can control which partitioner to use, no value is present it fallback to the default partitioning logic. When a producer asks to a partitioner which partition to send a specific record to, it provides a fixed set of elements:

- The data to be sent.
- The key associated with that record (if any, could also be null).
- To which topic the data belongs to.
- To which cluster the data must be sent. From a coding point of view, not having a partitioner chained to a specific cluster (and producer) permits more advanced programs to be develop, and allows a single producer to connect to different clusters and/or to have multiple producers sharing the same partitioner instance.

The default partitioner handles records with key differently form key-less ones, but the aim is the same, spreading them between partitions as equally as possible. The logic path/sequence of operations executed by the default partitioner to select a partition (based on its index, 0, 1, ...) is rather simple:

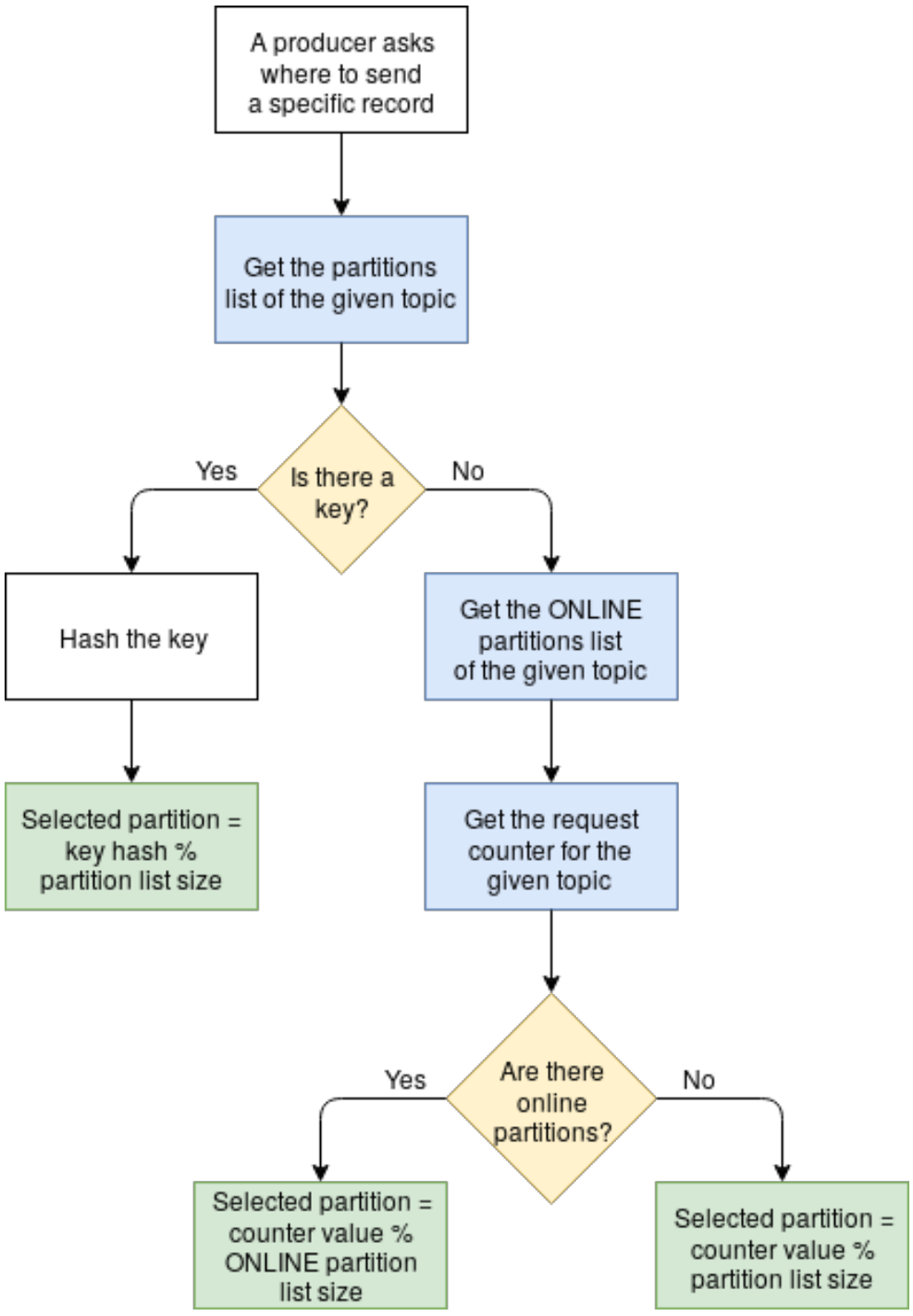


Figure 2.1: Flow chart representing the logic of the default partitioner

With an available key, its hash decides which partition to select, the pseudo-randomicity of the hashing result helps the algorithm to achieve a good data per-partition distribution (for the given topic).

If no key is available, a counter is retrieved, which tracks how many records have been sent by this partitioner to a specific topic, and its value is used to select a partition, based only on the available ones if possible. In the diagram, all the final (green) processes of selection require basic modulo arithmetic, but any action that relies on gathering data regarding a topic (blue with asterisks) requires the usage of lookup tables (implemented with hash-maps).

The counter for each partition is updated after every request. Its value changes with modular arithmetic selects every partition in order to achieve a maximum difference in the total number of records sent to the topic partitions of 1 (assuming working healthy cluster). This implements the Round Robin discipline (p.51) for the distribution of the records among the partitions.

2.3 Failing Data Distribution Assumption

Aside from code logic optimizations, the paritioner needs to be improved, it fails to perfectly complete its job by not covering fault scenarios, assuming that recovery from them would be fast enough. Faults happens at node level but the paritioner distribute data at partition level.

The easiest way to visualize where the Default Partitioner fails, is to set up a three node cluster with only one topic and three partitions perfectly split between nodes with one of them going offline.

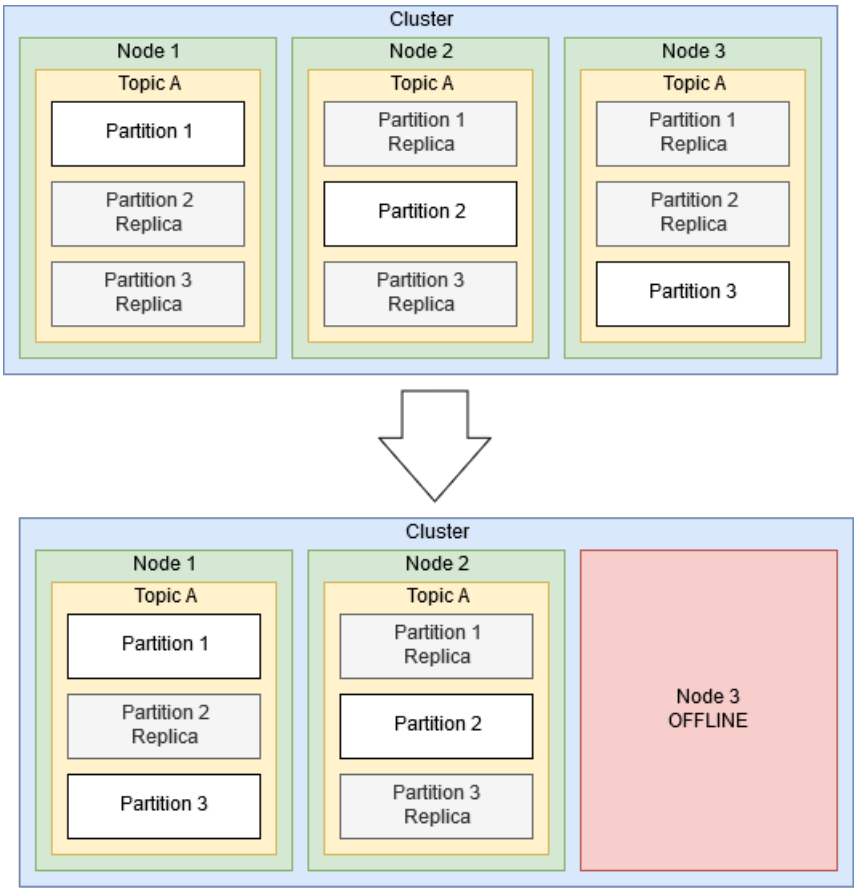


Figure 2.2: Partition 3 leader election result after a node went offline

Node 3 fails and **Node 1** becomes the new leader for **Partition 3**, different problematic scenarios emerges depending on were the system hardware design has been scaled incorrectly:

- Producer network line overload when sending record to **Node 1**.
- Consumer network line overload when requesting record from **Node 1**.
- Both the above if producers and consumers shares the channel to communicate with the cluster.
- Disk on **Node 1** may become overloaded, depending on sync policies (waiting for every replica to be in-sync or carry-on with new data send as soon the leader has confirmed a correct write).

The first consequence is to slow down the incoming requests to other nodes, since both producers and consumers are maybe waiting response from overloaded nodes. A secondary issue is harder overload detection if poor metrics are used to judge the cluster health, failure could be detected but overall metrics (average) may fail to show the single node overload in action.

Trying to keep equals per-partition data distribution unbalance per-node request distribution in fault scenarios. A simple approach would be to have one (or more) idling nodes dedicated to fast swapping with crashing one.

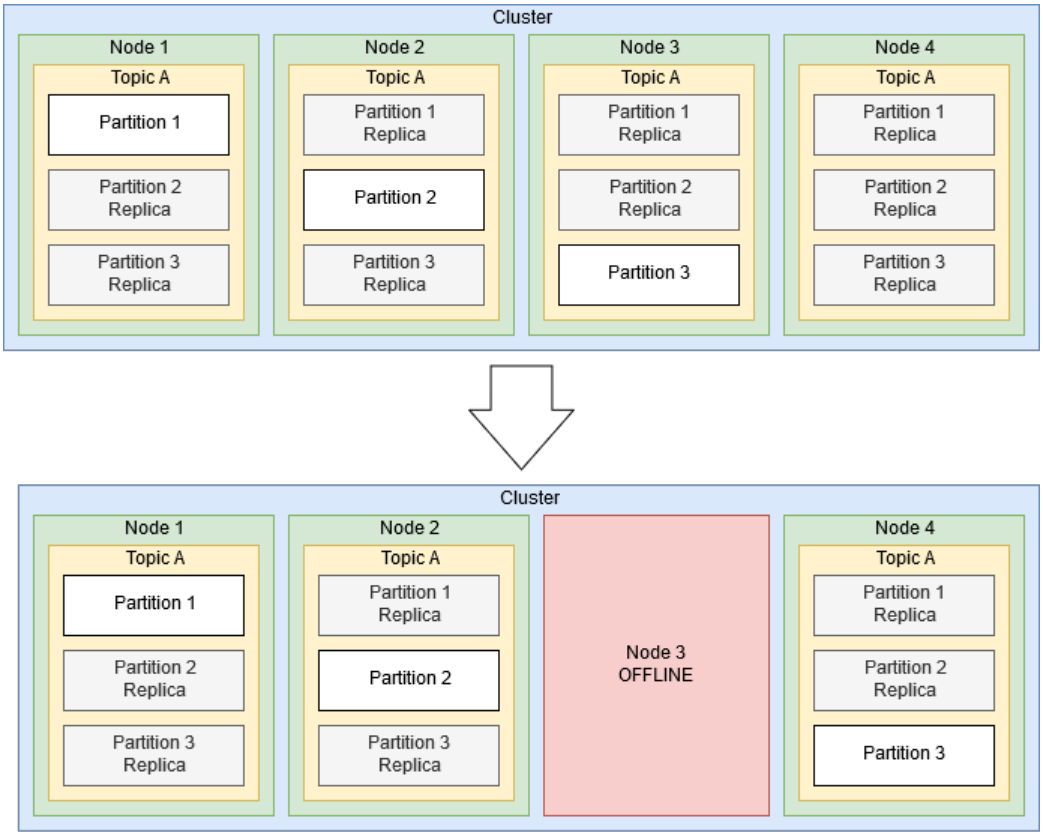


Figure 2.3: What a backup node would have done in 2.3

Dedicated failover servers are costly and may introduce new challenges.

- A new server is needed for every new grade of failure that is want to be avoided, multiplied for each topic touched by each faulty node.
- Failover server should have the same computational power as the other nodes.
- There is no complete idling since replicas must be kept in-sync.
- May require a more powerful network, caused by the increased replica synchronization traffic

2.3.1 Bad Configurations

A fault in not necessary the only way to get bad data distribution. Especially with multiple independent producers, mixing node assignments, unevenly among topics can easily results in unbalanced data amounts. The next chapter explains what a bad configurations is, and will show a common point with faults. The issues of both unhealthy and wrongly configured clusters are rooted in the cluster topology. When we speak about cluster topology we meant the whole structure, the assignments and the dependencies between nodes, topics and partitions.

2.4 Conclusion

Good balancing in Kafka is rather simple when dealing with easily predictable workloads and ideally reliable nodes. In practice, these conditions are hardly met. The fault tolerance of Kafka assumes the functionality of the system, but we have shown that there is not any mechanism to handle the unbalancing caused by the failures.

The scale of a fault is not the issue of bad partitioner performance and data balancing, disastrous events can put offline even the best configured cluster. The problem is how partitioners react to faults, which is completely overlooked. Too much faith is put into believing that every fault can be solved in reasonable time, and that any instantaneous sporadic minor impact on the distribution will gradually decrease its severeness with data aging.

Feature marketing could have been done differently to account for the simple partitioning policy. Being capable to continue to work even after major faults does not comes with a minimum QoS level. Apache Kafka, as is now, does not always have high fault tolerance, but instead it always has high fault survivability.

Chapter 3

Set Theory Prospective

3.1 Introduction

In the previous chapter we saw how failures are not the main reason that degrades the performance of any Apache Kafka system, the resulting cluster topologies (nodes, topics and partitions arrangements) actually are.

A cluster can be configured from start with a failing topology [2], even on purpose or by human inexperience [42]. Having an automatic detection mechanism helps the user to avoid such scenarios [15].

This chapter approaches Apache Kafka clusters with set theory. Labels are given to nodes, topics and partitions, and rules are set to relate them. A parallel is drawn between bad cluster topologies and specifics ways to group topics and nodes together. We also show an algorithm that can detect those scenarios.

Bad Cluster Configuration Example

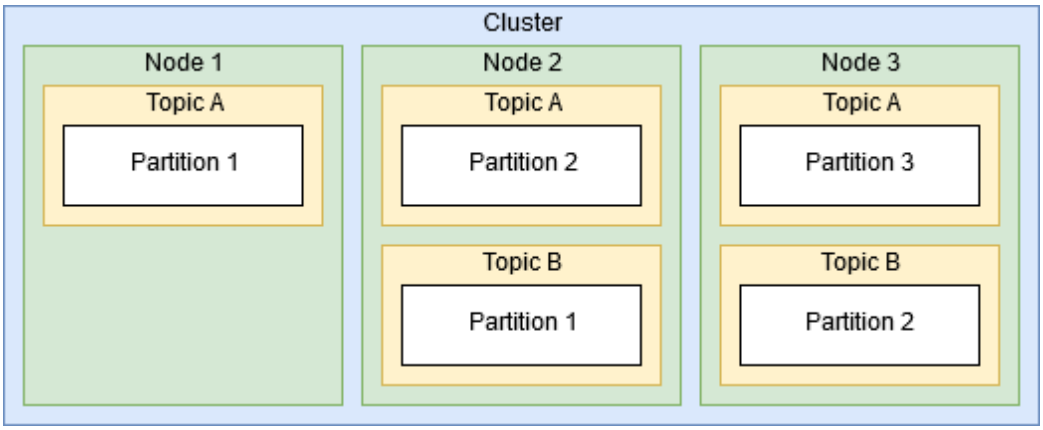


Figure 3.1: Cluster with an obvious asymmetry (thus unbalance, thus issue)

In this test cluster, it is unknown how the shown topology has been formed but, assuming equal distribution of load per topic, the first node is indeed much more relaxed compared to the others. It is not important to know if the bad cluster topology is caused by failures or bad configurations. In both cases thanks to a formal method these situations can be promptly detected.

Uneven topics/partitions count per node is not a prerogative of bad configurations. Complex dependencies between topics, nodes and partitions may create situations where the producers data flows can affect multiple nodes even if not directly touched by those partitioners. This other cluster (Fig.3.1) have even distribution of partitions and topics per nodes but suffers from data distribution unbalancing if the rate of each producers is not perfectly equal.

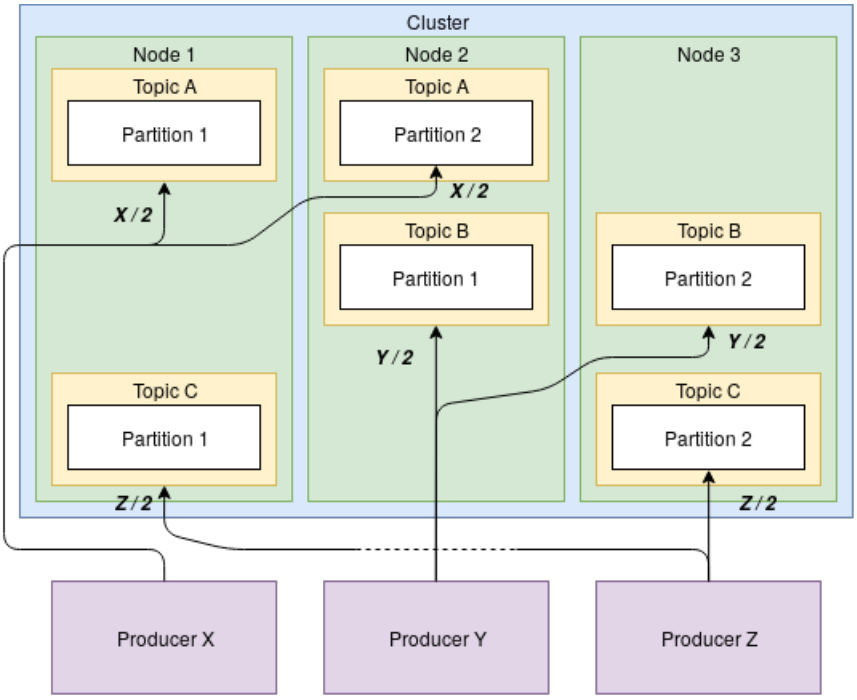


Figure 3.2: Bad data distribution if producers do not send at equals rates

If just one producer send less or more data than the others, its nodes, where it writes to, will experience less or more load. As example if $\lambda_X = \lambda_Y = 1$ (send rate of producers X and Y) and $\lambda_Z = 2$ (send rate of producer Z), Node 1 and 3 will see an incoming rate of 1.5 while Node 2 only 1. As result both Node 1 and Node 3 experience 50% more write requests from the producers and 50% more read request from the consumers (compared to Node 2).

This document gives an set theory approach to detect faults and avoid some lazy and bad Apache Kafka installations, while the official guide simply discourage it and favours manual tweaking. Obvious bad configurations of single topics, assigning an amount of partitions of the same topic to less nodes of said amount, are not explored until the end of this chapter.

3.2 Clusters as Sets

We introduce some mathematical objects that will be used in this chapter.

C_T set of all topics in a given Apache Kafka cluster.

C_N set of all nodes in a given Apache Kafka cluster.

C_P set of all partitions, no replicas, in a given Apache Kafka cluster. Replica information is only considered later in this chapter.

t_α represents **Topic** α and $t_\alpha \in C_T$.

n_β represents **Node** β and $n_\beta \in C_N$.

$p^\gamma_{\alpha\beta}$ γ-th partition of t_α with n_β as leader
$where\ \ p^\gamma_{\alpha\beta} \in C_P \wedge t_\alpha \in C_T \wedge n_\beta \in C_N$.

$N(t_\alpha)$ set of nodes, leaders of at least one partition of t_α.
$N(t_\alpha) = \{n_\beta \in C_N | \ \exists\ p^\gamma_{\alpha\beta} \in C_P\}$

$N(T)$ set of nodes, leaders of at least one partition of any topics in T.
$N(T) = \bigcup_{t_\alpha \in T} N(t_\alpha)\ \ where\ \ T \subseteq C_T$

Definition 1. Node-Topic Groups
A set of topics $T \subseteq C_T$ is a Node-Topic Group, written as T^*, if and only if every topic in it has the same set of nodes. All writes requests to topics on the same group, are sent to the same set of nodes. Hence data distribution within a group is ensured even if producers have different send rates when the Default Partitioner is used (when producers perfectly split data among the partitions of a topic).

$$T = T^* \iff N(t_\alpha) = N(t_\beta)\ \forall\ t_\alpha, t_\beta \in\ T$$

From which follows

$$\begin{aligned} t \in T \wedge T = T^* &\implies t \in T^* \\ t \in T^* &\implies N(t) = N(T^*)\ \forall\ t \in\ T^* \\ \#T \leq 1 &\implies T = T^* \\ S \subseteq T^* &\implies S = S^* \\ N(T^*) = N(t) &\implies T^* \cup \{t\} = U^* \\ t\ \in S^* \wedge N(T^*) = N(t) &\implies T^* \cup S^* = U^* \end{aligned}$$

Definition 2. Node-Topic Max Group
T^* is a Node-Topic Max Group, T^M, of C_T, if and only if, any topic in $C_T \setminus T^*$ merged with T^* would not form a new NTG.

$$T^* = T^M \iff T^* \cup \{t_\alpha\} \neq S^*\ \forall\ t_\alpha \in C_T \setminus T^*$$

Which is the impossibility to find a new topic assigned to any nodes of T^*.

$$T^* = T^M \iff N(T^*) \neq N(t_\alpha)\ \forall\ t_\alpha \in C_T \setminus T^*$$

This concept is used to identify those groups where we are sure that adding any other topic may break the assumption of correct data distribution when the Default Partitioner is used (when producers perfectly split data among the partitions of a topic).

Example: Basic cluster with three max-groups (Fig.3.2).

$$C_T = \{t_A,\ t_B,\ t_C,\ t_D\} \quad C_N = \{1,\ 2,\ 3\}$$

$$C_P = \{p^1_{A1},\ p^2_{A2},\ p^3_{A3},\ p^1_{B2},\ p^2_{B3},\ p^1_{C2}, p^2_{C3},\ p^1_{D1}\}$$

$$N(t_A) = \{n_1,\ n_2,\ n_3\} \quad N(t_B) = N(t_C) = \{n_2,\ n_3\} \quad N(t_D) = \{n_1\}$$

$$T^M_1 = \{t_A\} \quad T^M_2 = \{t_B,\ t_C\} \quad T^M_3 = \{t_D\}$$

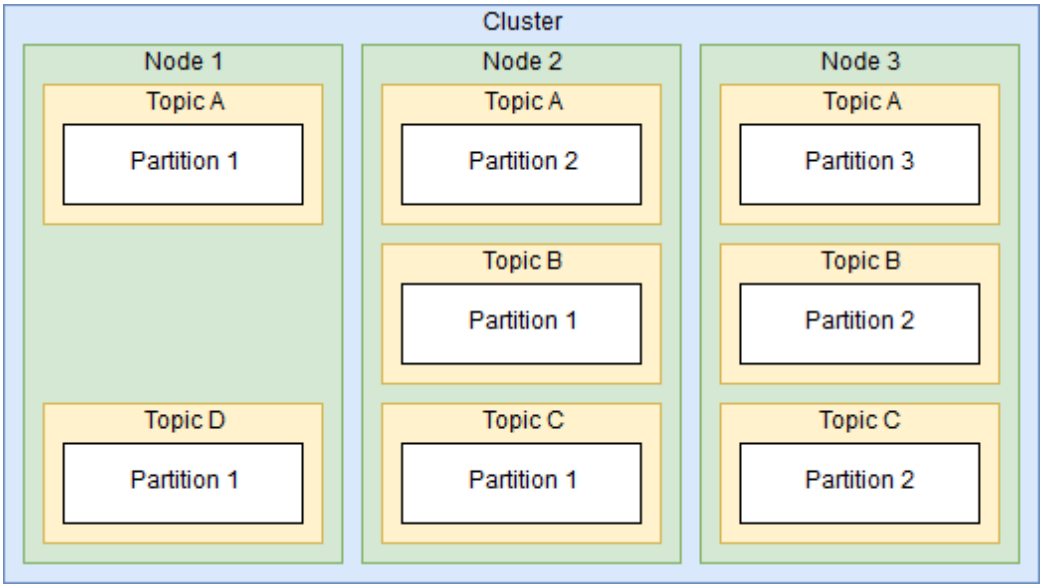


Figure 3.3: Visual representation of the example

Definition 3. Node-Topic Group Overlapping
Overlapping defined between two NTG sets, $T^* \circledcirc S^*$, occurs when the leader node set of T^* shares some elements with the one of S^*.

$$T^* \circledcirc S^* \iff N(T^*) \cap N(S^*) \neq \emptyset$$

No overlapping is easily defined by negating the previous statement.

$$T^* \not\circledcirc S^* \iff N(T^*) \cap N(S^*) = \emptyset$$

The overlapping combined with the definition of max-group, identifies entire groups of topics that may (or may not) have distribution problems once the cluster goes live.

3.3 Overlapping and Cluster Topology

In any given Apache Kafka cluster, all node-topic max groups can be in only two overlapping statuses:

Cluster No overlapping.
All max groups do not overlap between themselves.

$$T^M \not\oslash S^M \ \forall \ T^M \neq S^M \subseteq C_T$$

This covers all clusters where every max group has its own set of nodes assigned and does not share it, even just partially, with any other group.

Cluster Partial overlapping.
At least one non-empty max group overlap with a different one.

$$\exists \, T^M \neq S^M \subseteq C_T \wedge T^M \neq \emptyset \wedge S^M \neq \emptyset \mid T^M \circledcirc \ S^M$$

Detecting partial overlapping not only tells if a cluster have topology problems but also can pin-point where they are located, if a program is designed to keep an history of every pair (T^M, S^M) that satisfied the partial overlapping relation.

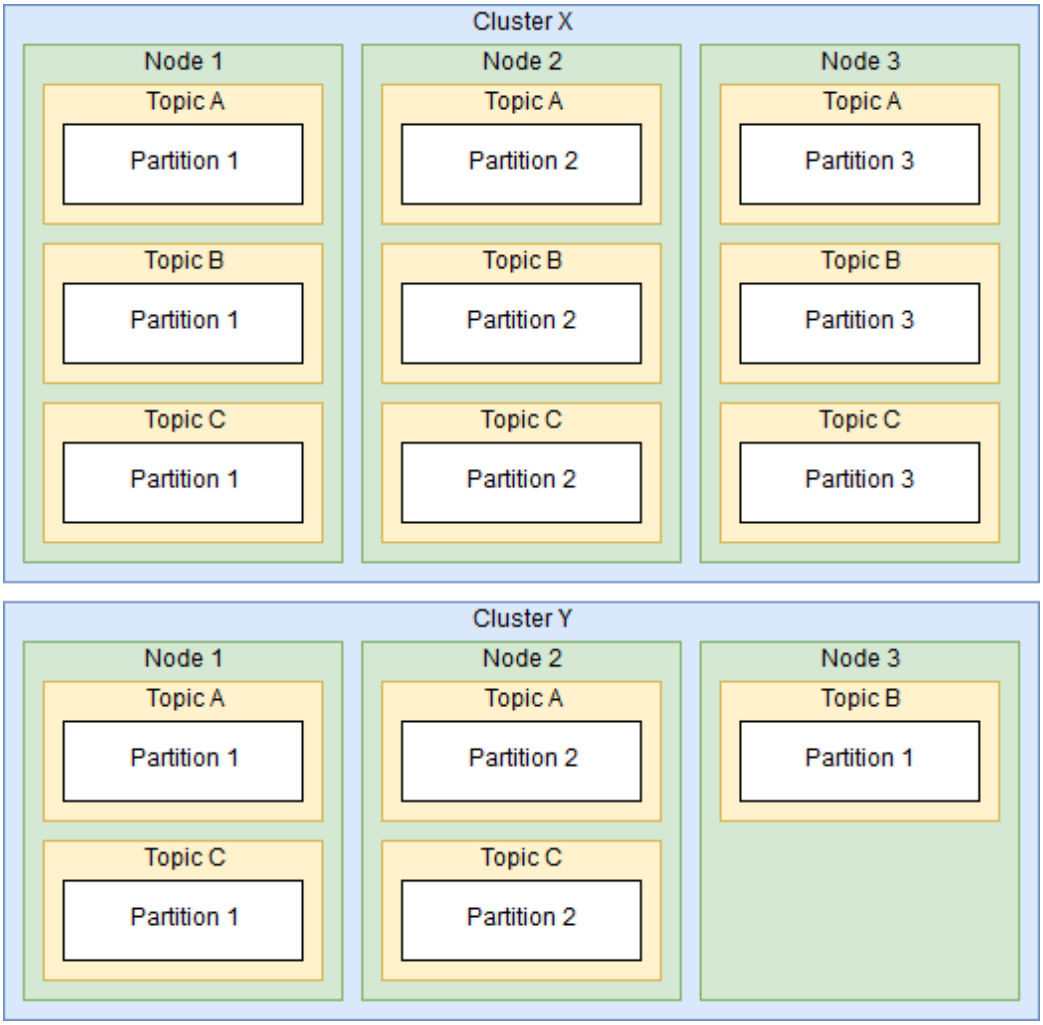


Figure 3.4: Clusters with no overlapping issues

Cluster X and Y does not suffers from partial overlapping, X has only one big max-group ($T^M = \{t_A,\ t_B,\ t_C\}$), Y has two, dividing t_B from the other topics ($T_1^M = \{t_A,\ t_C\}$ and $T_2^M = \{t_B\}$).

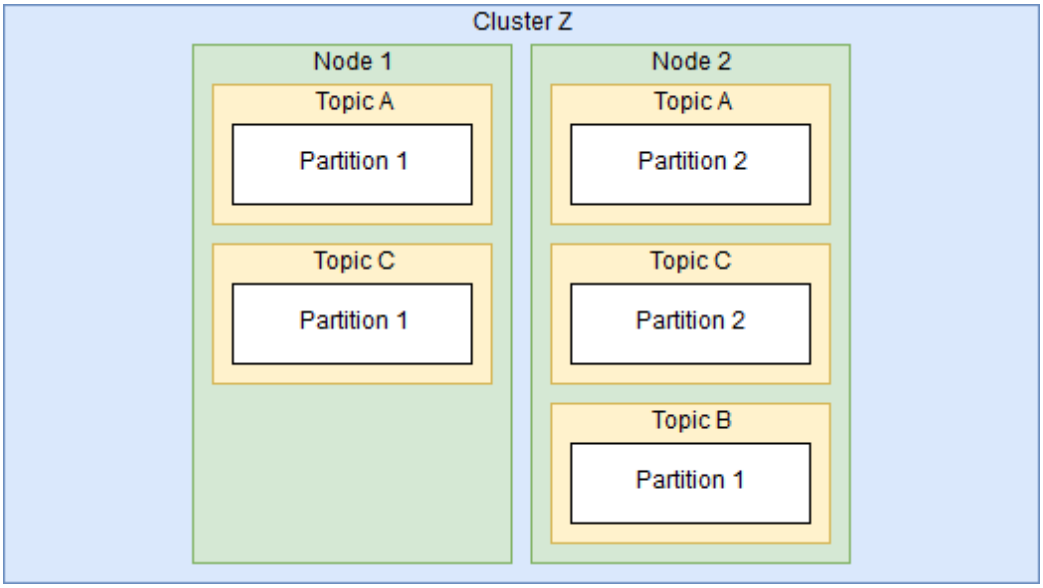


Figure 3.5: Cluster with clear overlapping issues

Cluster Z has the same max group configuration of cluster Y, only two max-groups $T_1^M = \{t_A,\ t_C\}$ and $T_2^M = \{t_B\}$, but the overlapping problem is obvious. Following the overlapping definition (p. 30):

$$N(T_1^M) = \{1,\ 2\}$$

$$N(T_2^M) = \{2\}$$

$$N(T_1^M) \cap N(T_2^M) = \{2\} \neq \emptyset \implies T_1^M \circledcirc T_2^M$$

Which satisfies the definition of cluster partial overlapping (p. 31), since exist a pair of distincts not-empty overlapping max-groups.

3.4 Cluster Worst Case Scenarios

Before moving to algorithms that handle overlapping is imperative to early distinguish which theoretical cluster will be the benchmarks of said test. Sticking to an undefined number of nodes C_N, the only variable that needs more clarity is the number of topic inside any given cluster.

Each cluster defining overlapping has its own worst case scenarios, which should be used has lower and upper bounds to evaluate the complexity of those algorithms that works with set theory applied to Apache Kafka clusters. Both situations just helps estimating the possible number of max-groups in a cluster, but there is no information about topic per-group which vary based on each application specs.

On next worst case analysis , to "keep them real" (and meaningful), topic distribution will be assumed to be evenly spitted between max-groups

$$\#N(T^M) \propto \underbrace{\frac{\#C_N}{\#M}}_{\text{set of all max-groups} \subseteq C_T} \quad \forall\, T^M \subseteq C_T \wedge M = \{\bigcup_{S^M}^{\subseteq C_T} \{S^M\}\}$$

or to have just one topic per max-group $\#N(T^M) = 1 \quad \forall\, T^M \subseteq C_T$.

Cluster No overlapping.

Every max group can not overlap with any other group. Regardless the overall amount of max group in the cluster, which can be just one and tops out at C_N, every node has at most one max group covering it. In a cluster with no overlapping N^{-1} exists (the inverse function of N, mapping nodes to sets of topics) and is always equals to a max-group

$$\exists\, N^{-1} \wedge N^{-1}(n) = T^M \subseteq C_T \quad \forall\, n \in C_N$$

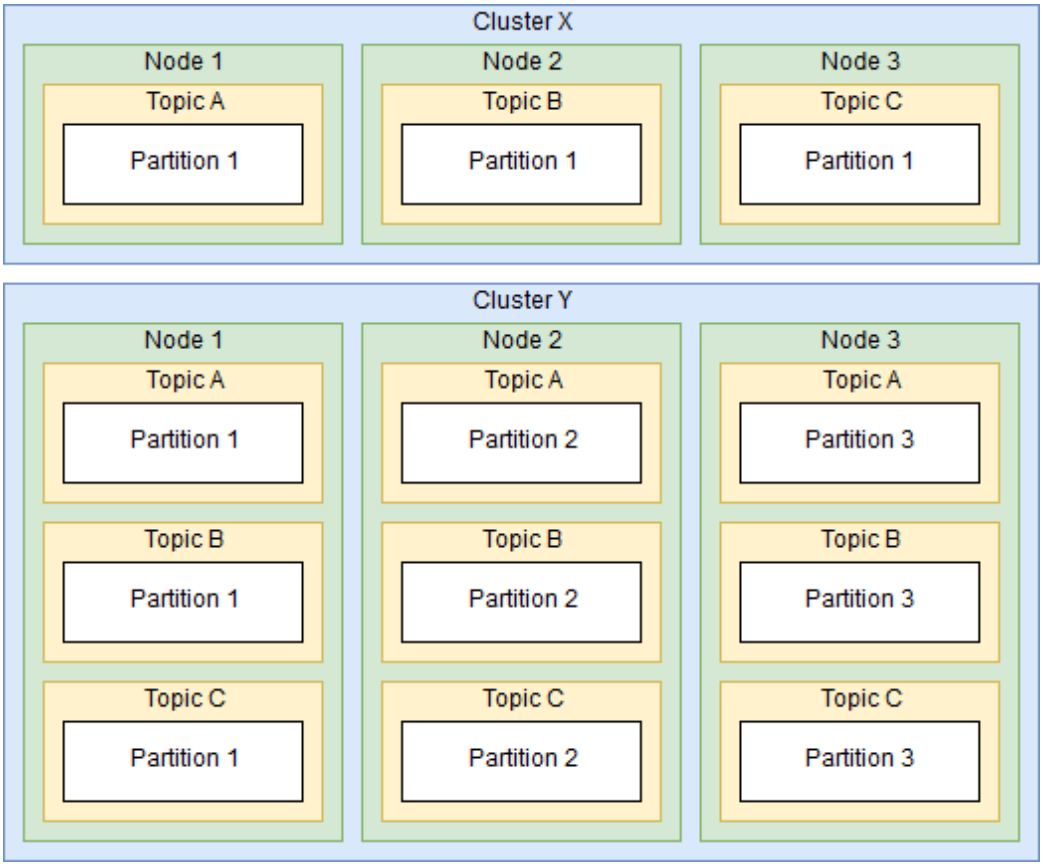


Figure 3.6: Clusters with one max-group

Cluster X has one max-group for every node, Y have one big max-group covering the whole cluster, both have no overlapping issues and, as expected, both have just one max-group assigned to each node.

Cluster Max Overlapping.

This case require more explanation. With no overlapping constraints, and aiming to achieve as much overlapping sets as possible, the journey to find out how many max-groups we can fit in a cluster becomes more conflicted. Since a node-topic group to be a max-group, by definition (p.29), must have at least one different assigned node from every other max-group, computing the amount of max-groups is counting all possible different un-ordered sets, with cardinality going from 1 to C_N, that can be formed with C_N nodes. By directly translating words to formulas, the amount of all combinations of sets with size k in a pool of C_N elements is the definition of binomial coefficient, making k range from 1 to C_N gives the final result.

$$\sum_{k=1}^{\#C_N} \binom{\#C_N}{k} = 2^{\#C_N} - 1$$

That is a specific use of the famous [9] binomial expansion

$$\sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k = (x+y)^n$$

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n \qquad x = y = 1$$

$$\sum_{k=1}^{n} \binom{n}{k} + \binom{n}{0} = 2^n \qquad k \text{ starts from 1}$$

$$\sum_{k=1}^{n} \binom{n}{k} = 2^n - 1 \qquad \text{since } \binom{n}{0} = 1 \; \forall \; n \in \mathbb{N}$$

An easier way to think about this is to count how many binary numbers can be written with C_N bits (-1 to not count 000...000, a topic with no nodes assigned), the i-th bit tells if the i-th node is assigned or not to the related max-group of the binary number taken into exam.

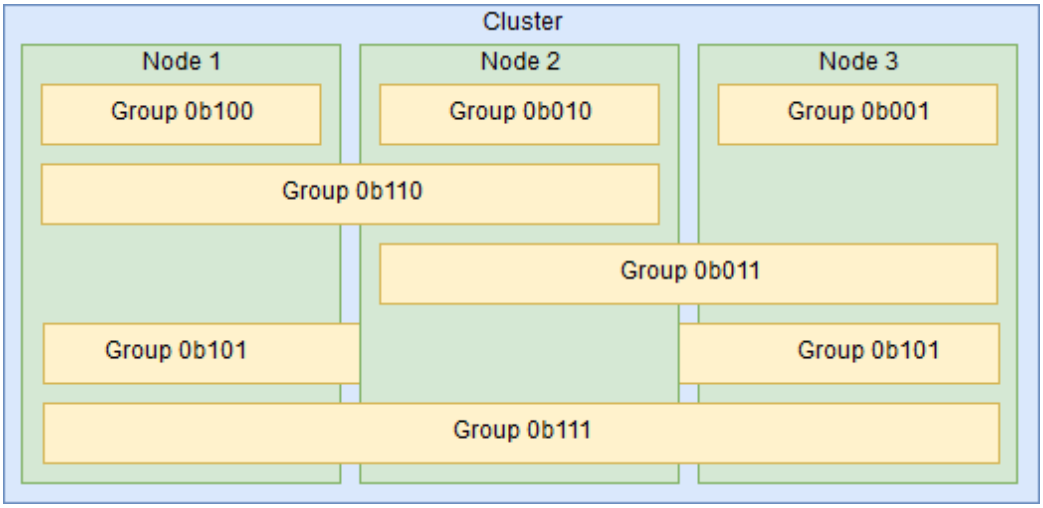


Figure 3.7: Cluster with 3 nodes and max overlapping

This is the visual representation of how a cluster with maximum overlapping looks like, only max-groups are kept and topics in them are removed for image clarity. Each group has been labelled with a binary number to highlight which nodes is assigned to.

As expected with 3 nodes, $\#C_N = 3$, the cluster has 7 max-groups

$$\sum_{k=1}^{3} \binom{3}{k} = 2^3 - 1 = 8 - 1 = 7$$

Complexity of function $N()$

A simplification can be introduced to more easily estimate the topic population while running the function $N()$, which maps sets of topics to sets of nodes (p. 28). $N()$ retrieve contents in constant time while dealing with a node-topic group, since every topic in it has the same set of nodes assigned (in Apache Kafka basic topology information are mapped so it is fair to assume this). Having infinite number of topics inside every node-topic group, or just one, makes no difference while executing $N()$ (if it is aware that it has been executed on a node-topic group) because $N(T^*) = N(t)\ \forall\ t \in T^*$ (by p. 28).

$N()$ returns a set of nodes, if represented as list of integers exploring it can be easily done in linear time compared to the size of the set itself. Since nodes are either assigned or not to a topic (set of topics, node-topic group, max-group, ...) representing the mapping from topic to nodes with bit-masks instead of lists of integers greatly reduce the overall number of operations. On a cluster with 2^n nodes and 2^x-bits arch. the list of bit-masks will be $\lceil 2^{n-x} \rceil$ long ($\lceil 2^n/32 \rceil = \lceil 2^{n-5} \rceil$ with 32-bits, $\lceil 2^{n-6} \rceil$ on 64-bits, ...).

3.5 Overlapping Detection

Overlapping detection can be implemented by first gradually building each node-topic max groups, and later find which ones overlaps.

3.5.1 Gather max-groups information

Algorithm 1 Builds M, the sets of every node-topic max groups.

```
procedure getMaxGroups(C_T)                    ▷ C_T set of topics in a cluster
    M ← ∅                                      ▷ Starts with no max groups found
    for t ∈ C_T do                             ▷ For every topic
        for T ∈ M do                           ▷ For every max group found until now
            if N(T) = N(t) then                ▷ T and t are in the same max group
                T ← T ∪ {t}
                continue outer for-loop        ▷ Directly goes to fetch next t
            end if
        end for                                ▷ Next instruction is skipped if inner if is triggered
        M ← M ∪ {{t}}                          ▷ Add a new max group
    end for
    return M
end procedure
```

Complexity analysis

		One topic per max-group $\#T = 1 \; \forall \; T \in M$	Even topic dist. $\#T = \#C_T/\#M$ $\forall \; T \in M$
No Overlapping	Single Max-Group $\#M = 1$	Case 1. $\mathcal{O}(1)$	Case 2. $\mathcal{O}(\#C_T \#C_N)$
	$\#C_N$ Max-Groups $\#M = \#C_N$	Case 3. $\mathcal{O}(\#C_N^2)$	Case 4. $\mathcal{O}(\#C_T \#C_N)$
Max Overlapping $\#M = 2^{\#C_N} - 1$ Max-Groups		Case 5. $\mathcal{O}(4^{\#C_N} \#C_N)$	Case 6. $\mathcal{O}(\#C_T 2^{\#C_N} \#C_N)$

Case 1 - $\mathcal{O}(1)$

One max-group with just one topic leads to $\#C_T = 1$, makes the first loop to be executed only once, and the second to be skipped to populate M with just the only available max-group.

Case 2 - $\mathcal{O}(\#C_T \#C_N)$

All topics are placed inside the same, and only, max-group, $\#C_T$ dictates how many times the first loop is executed and second run only once for every topic. The inner, and only, if statement will always compare to identical list of nodes, the whole set C_N, since the only max-group, and so every topic, spans on the entire set.

Case 3 - $\mathcal{O}(\#C_T \#C_N) = \mathcal{O}(\#C_N^2)$ **since** $\#C_T = \#C_N$

Same as **Case 1**, one topic per max group forces a fixed number of topics, but this time having $\#C_N$ max groups leads to $\#C_T = \#C_N$ topics. The second loop will gradually cycles more for every new topic which will form its own max-group, since no overlapping will be found with the already existing max-groups. The comparison is $N(T) = N(t)$ linear on the number of nodes per max-group so its $\mathcal{O}(1)$ in this case.

Case 4 - $\mathcal{O}(\#C_T = \#C_N)$

Having more topics per max-groups does not change the performance of $N(T) = N(t)$ comparison. The only difference with **Case 3** is that $\#C_T$ is a free variable and not depends on $\#C_N$.

Case 5 - $\mathcal{O}(\#C_T 2^{\#C_N} \#C_N) = \mathcal{O}(4^{\#C_N} \#C_N)$ **since** $\#C_T = 2^{\#C_N}$

$\#C_T$ depends on $\#C_N$, by its chain to $\#M$, with one topic for each of the $2^{\#C_N}$ max-groups, $\#C_T = 2^{\#C_N} - 1$. Both for-loops are executed $2^{\#C_N} - 1$ times, the if check needs at most $\#C_N$ operations.

Case 6 - $\mathcal{O}(\#C_T 2^{\#C_N} \#C_N)$

Same as **Case 5**, but C_T can not be derived from C_N, so it is left as free variable.

3.5.2 Detect Partially Overlapping Max-Groups

Algorithm 2 Detect **P**artial **O**verlapping by filling the set P with max groups pairs that causes topological problems.

procedure DPO(M) ▷ M Sets of max groups $\{M_1^M, M_2^M, ...\}$
  $P \leftarrow \emptyset$ ▷ Begins with no conflicts found
  for $i \leftarrow 1$ to $\#M$ **do**
    for $j \leftarrow i+1$ to $\#M$ **do**
      if $N(M_i^M) \cap N(M_j^M) \neq \emptyset$ **then** ▷ if $M_i^M \circledcirc M_j^M$
        $P \leftarrow P \cup \{(M_i^M, M_j^M)\}$ ▷ Records partial overlap
      end if
    end for
  end for
  return P
end procedure

Complexity analysis

		One topic per max-group $\#T = 1 \; \forall \; T \in M$	Even topic dist. $\#T = \#C_T/\#M$ $\forall \; T \in M$
No Overlapping	Single Max-Group $\#M = 1$	Case 1. $\mathcal{O}(1)$	
	$\#C_N$ Max-Groups $\#M = \#C_N$	Case 3. $\mathcal{O}(\#C_N^2)$	
Max Overlapping $\#M = 2^{\#C_N} - 1$ Max-Groups		Case 5. $\mathcal{O}(4^{\#C_N}\#C_N)$	

Writing an algorithm that does not consider single topic exploration, as expected, does not expose $\#C_T$ in the final complexity computation. Having one or an infinite amount of topics does not affect performance since the algorithm handle only max-groups and max-group aware $N()$ function (p. 38).

Saving only the pair (M_i^M, M_j^M) does not tells directly which nodes are shared between M_i^M and M_j^M, changing the inner loop body solves the issue but also require P to not store pairs but triples, adding $N(M_i^M) \cap N(M_j^M)$.

...
$I = N(M_i^M) \cap N(M_j^M)$ ▷ Saves possible overlap information
if $I \neq \emptyset$ **then** ▷ if $M_i^M \circledcirc M_j^M$
    $P \leftarrow P \cup \{(M_i^M, M_j^M, I)\}$ ▷ Records partial overlap
end if
...

This small change may actually significantly impact performance because just telling if two sets completely differs needs less memory and is faster in few lucky cases, while, instead, recording each conflict must always fully scan both sets. A possible smarter real world implementation could rely on the basic code and fully compute the set intersection $N(M_i^M) \cap N(M_j^M)$ only on user request.

3.5.3 Improving Detection Results Readability

The basic DPO algorithm produce a list/set of max groups pairs, it may be hard to pin-point where the issues is located on complex clusters. This new detection algorithm returns a list of pairs, matching each node with the set of max groups that touch it, if the sets contains more than one element, all those max group overlaps on at least that node.

Algorithm 3 Detect **P**artial **O**verlapping with improved readability

procedure DPO2($M, \#C_N$) ▷ M Sets of max groups $\{M_1^M, M_2^M, ...\}$
▷ $\#C_N$ number of nodes in the cluster
  $R \leftarrow \{R_1 = \{\} \; ... \; R_{\#C_N} = \{\}\}$ ▷ Init. R, maps nodes to max groups
  for $T^M \in M$ **do** ▷ Fill R
    for $n \in N(T^M)$ **do**
      $R_n \leftarrow R_n \cup \{T^M\}$
    end for
  end for
  $P \leftarrow \emptyset$ ▷ Init. P, starts with no conflicts found
  for $R_n \in R$ **do** ▷ Fill P
    if $\#R_n > 1$ **then** ▷ Keep only pairs with overl. max groups
      $P = P \cup \{(n, R_n)\}$
    end if
  end for
  return P
end procedure

This more readable output needs more data structures and memory but is not slower to obtain. The algorithm is divided in three sections, initializing R, filling R and, at the end, striping R of useless information (nodes with no overlapping issues) to obtain P. Both building of P and R require $\mathcal{O}(\#C_N)$ operations, filling R needs $\mathcal{O}(\#M\#C_N)$ operations.

3.5.4 Early Overlapping Detection

This whole chapter has been focused only on examine a given cluster topology and test for overlapping issues, while taxing performance-wise these operations may also fall short to give the complete evaluation since replicas are not considered (definition of set C_P p. 28). Replicas are not considered in overlapping detection since the worst case scenario is meaningless, with no real limit on replica creation, and even "real"-worst case scenarios require to consider some degree of exponential growth crippling algorithms performance. An advanced take on overlapping detection, that also considers replicas information, could be implemented as aid during cluster configuration, to limit computation only each new topology change. Fault simulation may also be considered but could result unfeasible on large clusters. New, but naive, algorithms are also necessary to detect all those cases were no overlapping occurs but partitions-per-node counter differs too much.

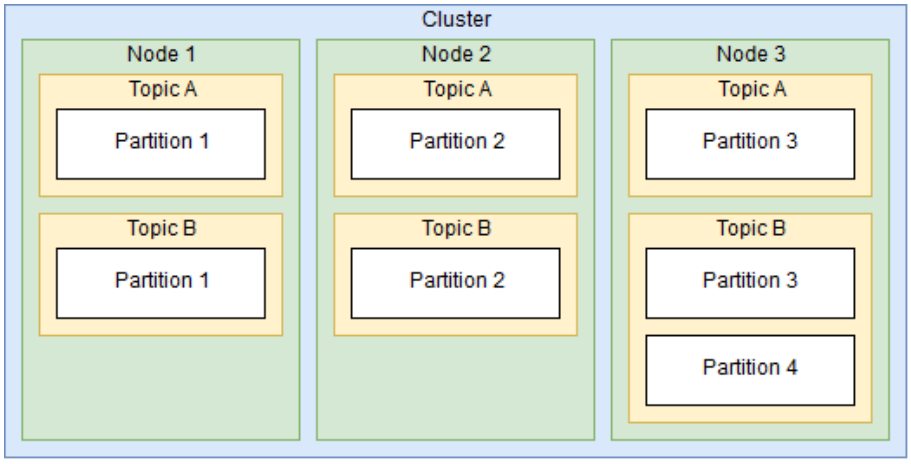


Figure 3.8: Uneven partitions amount per topic in the same max group

This cluster has no overlapping issue but Topic B has four partitions split in three nodes leaving Node 3 more loaded than the others. A simple automatic check can notify the cluster admin about this kind of issues.

3.6 Conclusion

Cluster Partial Overlapping Detection require two algorithms with similar execution time, both achieve $\mathcal{O}(k\#C_N)$, where k can be either $\#C_T$, $\#C_N$ or $\#M$. In any given $\#C_T$ and $\#C_N$ are known values, $\#M$ instead, the number of max-groups, must be computed (with the first presented algorithm) and is not always derivable directly from $\#C_T$ or $\#C_N$, as happened in few peculiar cases.

The proposed algorithms have the advantage of being simple but at the same time, looking at their raw complexity analysis, are bad performing. Smart programs can be implemented to delay computation only when is necessary and/or on user request, but the undeniable worst case scenario casts its shadow of exponential complexity growth. While it is still correct to assert that even real cluster topologies can range from trivial to incredible exponential structures, obtaining these last ones for meaningful dimension on $\#C_N$ can not be done without the help of automatic procedures conceived exactly to build bad performing clusters.

Replicas were not considered, doing so results in less demanding analysis but also leaves unchecked simpler topologies with easier issues, like having more partitions than node assigned. Overlapping detection helps to reconfigure clusters to remove conflicting node dependencies between multiple nodes, but everything can still fails with no advanced logic in load distribution. The default partitioner shipped with Kafka has already shown its problems handling whatever topology issue emerged in these first chapters, in the next ones more advanced partitioner will be presented to gradually tackle each challenge.

Chapter 4

Scheduling Theory Disciplines

4.1 Introduction

Before diving into advanced implementations that solve Apache Kafka problems, it is imperative to explore the theory behind the partitioning mechanism. It is possible to draw a parallel from the act of deciding to which partition to send a specific message and scheduling theory [20]. Therefore a fast direct conversion can not be applied, units of measure change and active/passive components may switch while adapting an algorithm.

Any scheduling algorithm allocates resources to an incoming list of tasks while trying to achieve the same goals [35] [21] [40]:

- **No starvation**. No job should be left in the queue without never being processed.
- **High fairness/average-utilization**. If jobs are available no process unit should be left idling, and the load between them should be balanced.
- **Low latency**. Jobs waiting in queue should be scheduled promptly.

Apache Kafka can be seen as a relaxation of the scheduling problem where the jobs are represented by the records in queue and the shared resource is the cluster.

- No starvation can happen. Records are always sent in order, they can be batched together, but ultimately only when the write acknowledgment is received the next records can be served. Once a record has been successfully sent to the cluster and written, no re-queue happen.
- High utilization and fairness (between nodes) is accomplished by satisfying the data distribution assumption.
- Latency is how much time a record is expected to wait to see its state transitioning from "queued" to "written-in-the-cluster". Obviously, from the partitioner prospective the overall waiting time can be tackled only with a swift decision in assigning a record to a partition, other tasks, like network and disk I/O, cannot be controlled directly. Here it returns one of the oldest problem programming ever had, and always will have, trading accuracy, precision and/or grades of correctness with computing time. Apache Kafka achieve extremely low latency by having, as this thesis shows, an almost decision-less default partitioner [40], which by design does not cover all use cases publicized by Apache Kafka feature-marketing (high fault tolerance).

This chapter will analyse the most common algorithms that can be found in literature, often categorized by separating the ones which works independently from the jobs size and the ones that do not.

4.2 Preemption and Time Slots

Before diving into some different scheduling policies, other concepts must also be known. Preemption and time slots are basic well established scheduling techniques usually boundled together, one often implies the other but they are completely different.

Preemption
The act of temporally removing a job from its assigned processing unit, putting it back to whatever data structure, if any, were used to group waiting processes, and continuing the scheduling operations with the newly freed resource. It can be a voluntary choice of a process, to suspend itself, useful when the waiting is necessary and/or known to happen, really helps only if an high portion of running applications were implemented with this feature in mind.

Time slots
Represented as a simple limit which dictates the maximum continuous time a task can spend been processed, when the virtual timer expires the preempt mechanism takes place.

Both are not one-size-fits-all algorithms, with special pre-computed fixed values that solve any issue, they can and must be tweaked to perfectly adapt to the wide range of systems which require scheduling practices. Most advanced implementations may include non-static parameters that change by learning from past behaviors. Modern systems mostly exploit these features with the wide presence of multicore hardware and thread-oriented software.

4.3 Size Independent

Often labelled as the easiest to implement, the decision making process do not take into account one particular metric. Ease in programming should not be confused with overall algorithm complexity, as we will see, even this category of algorithms may require some degree of data structure and/or have at least linear time complexity.

4.3.1 FIFO

Its acronym stands for First In First Out, which underlines the simplistic nature of this policy, a fact also notable with its I/O scheduler counterpart called No-Op, literally a short version of "No Operations". Its extended name tells how following the FIFO policy to schedule jobs does not need advanced decision making, it simply puts each incoming job in a queue data structure in the exact order of arrival.

4.3.2 Priority Queue

Evolution of the FIFO policy keeps the simple data structure but improves the logic and so performance. Serving jobs in the incoming order is still the basic logic but there is also a declared or assigned priority that dictates its placement inside the queue, this is done to ensure a minimum QoS to any high-priority task. Starvation is a big risk every time priority-ordering is involved, more important jobs could continue to come, forcing the low priority ones to never leave the queue. To avoid starvation there might be implemented mechanisms of modifiable soft-priorities, jobs that spend too much time in the queue can increase or decrease their virtual priority, permanently or not, to move their location inside the queue itself. Of course reordering, also called Priority Demotion/Promotion, increases the overall algorithm complexity.

This simple mutation of the base algorithm does not fit the Apache Kafka use case. All records have the same importance and are sensitive to order, this may seems to contradicts the case where multiple producer are acting on the same Kafka cluster, but this tempting intuition is false since the initial statement has to be true only within a single partitioner. It is imperative to remember while adapting already existing algorithms from the theory to Apache Kafka, that its specific use cases require specific solutions or ad-hoc modifications (that could also simplify the overall problem).

Multi-Queue

The multi-queue variant is a much more widely adopted version, with a queue for each priority level available. Other non-priority based grouping classes/criteria could be used, but the basic idea of having different funnel/buckets to categorize jobs still applies. Having multiple data structures facilitates and speeds up all operations needed to manage priorities changes. This solution naturally allows advanced parallelism mechanisms, but must keep in check synchronization between working active entities/units in the system.

In Apache Kafka all records are considered equal, there is no priority, their categorization is based on their intrinsic meaning. Nodes in the cluster can be seen as multiple queues structures by grouping up producers and destination nodes based on data type handled, and, for the sake of simplicity, assuming that no entity handle multiple distinct types of data.

4.3.3 Round Robin

This policy directly tackles fairness by cycling/assigning jobs through the available processing unit. The limit of maximum continuous processing time, called quantum, is applied to each task being actively processed, once the timer expires the job is suspended and puts back at the end of the queue. Time slots and circular queues are the reasons why this algorithm is so widely used, it also solves starvation problems since every job is bound to be completed even if it would take multiple time slots.

The length of the quantum is variable [24] inside the Round Robin scheduler:

- Set big enough to cause a major portion of jobs to finish within one or few time slots. No starvation/locks occurs but the system may feel unresponsive when a long process is been served.
- Setting the quantum so big, to a virtual infinity, that any process is going to end its computation without letting the timers run out, will emulate the FIFO policy.
- If set too low, timers would run out too often and, by that, swapping out jobs from processing state would be too aggressive, the processing unit will spend more time switching context instead of computing. Changing task is not instantaneous, it requires handling data structures, context switching and the next to-be-processed jobs could also have warm-up times that may be smaller than the slot length chosen.
- Can not be set to 0, its the degenerate case of the "too short time slot", fascinating to be considered only in theory. The processing unit will only perform swap process operations without actually letting any job to advance to its computation.

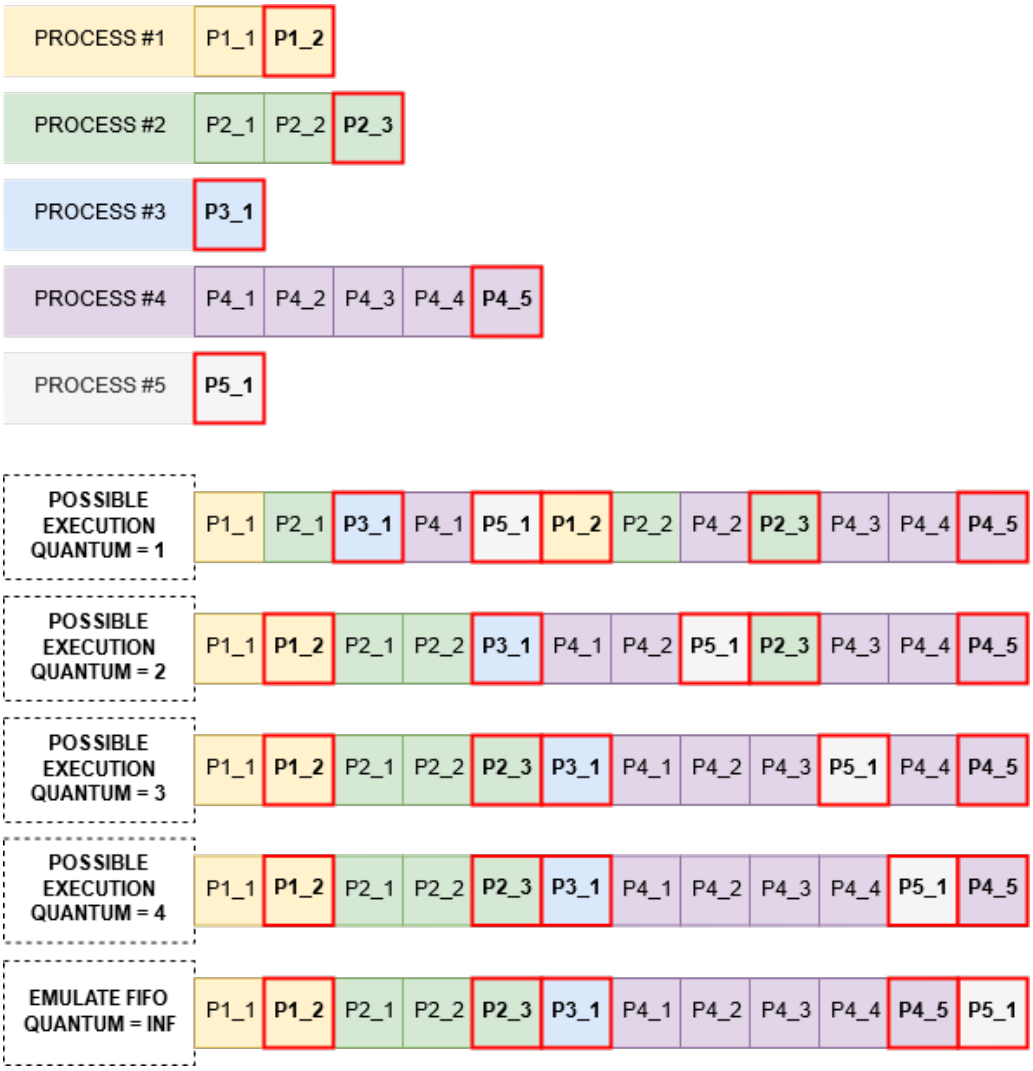


Figure 4.1: Quantum length effect on different task sizes

This example shows how the quantom choice has an impact the final result, with a large quantum the 3rd and 5th process may have to wait a lot to reach their ends, instead the 1st and 2nd would terminate earlier. Continuous, or periodic, performance sampling is an efficient widely adopted solution to tweak the quantum length, not only to pursue an "overall best" but also to follow the jobs flow evolution.

4.4 Size Dependent

Previously explored policies have shown some degree of metrics collection to improve the resulting scheduling, why grouping together all schedulers that consider jobs length in a different category? The size of any given task is not known before the end of the task itself. Any algorithm that takes decisions based directly on size, or other metrics derived from it, must implement dynamic assessment techniques and can not use only hard-coded constant evaluations.

4.4.1 Shortest Job First

A priority queue where the priority is decided based on the time required for each jobs to end. This policy maintains the starvation problems that priority queue had and amplifies it, since jobs in real-time systems does not have a declared process time. Even on non-realtime system (like batch systems), were the users declare the amount of time in which theirs jobs can operate, the starvation problem is still relevant when a never ending stream of incoming new short jobs is present [30].

This scheduler does not contemplate preemption, the only basic improvement available, without deep changes in the schedule logic, is a new pointer to the rear of the queue. Having direct access to both the head and the tail of an ordered queue, allows the scheduler to periodically change the location to from where to pick jobs. The small enchantment does not solves starvation, combined burst of big and small jobs can still starve middle sized ones, and arguably defeat the motives behind the name chosen for this policy.

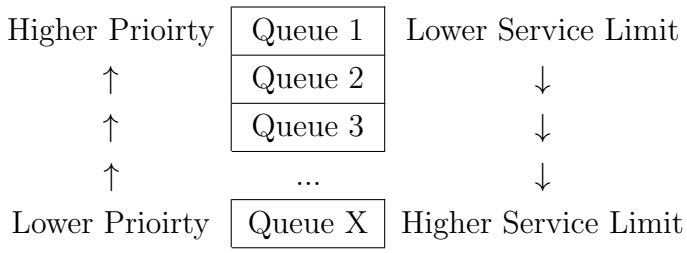
By the nature of the parameter we need to analyze, jobs length, any implementation based on its estimation, either by system average, history or any metrics, is, and will always be, an approximation of this policy.

4.4.2 Shortest Remaining Time First

Adding preemption capabilities to the SJF algorithms, partially solves latency related problems. SJF fails in all those timing sensitive scenarios, where a long task arrives first before a long burst of small ones, lacking the ability to see the future and evaluate perfectly the process time tremendously aggravate the average jobs latency. While still relying mainly on an accurate jobs size estimation [4], preemption helps to adapt to the evolving state of the incoming jobs flow, at any given time the running task is the one with less remaining process time [29].

4.4.3 Size Based Priority Multi-Queue

Multiple queues are used in this policy, ordered by decreasing priority level and increasing threshold to job demotion, also called service limit.



This policy moves a job to the next structure with lower priority based on the amount of service provided, the complementary criteria of SRTF. To select the next job to process, the highest not-empty priority queue is taken and the control is given to its assigned scheduling policy. Each queue can have any scheduling policy, it only must be modified to include the preemption mechanism once a job has received enough service and need to be demoted to a lower queue.

4.5 Conclusion

Apache Kafka does not directly implement its own version of any scheduling policy, modifications are required to fulfil the specification of its use case. The scheduling to Apache Kafka translation follow these mappings:

- Jobs are represented by records collected by the produces.
- The nodes are the resource to be shared, both for producers and consumers.
- Nodes do not perform computation related to records, only to handle and store them. The service time can be represented by the summation of network latencies and time to be handled by the server.
- Since records are order sensitive within the same partition and sizes should not vary within the same topic, records based priority policies are not meaningful.

Regardless of the policy chosen, in scheduling theory the performance are mainly dictated by the incoming flow of jobs and theirs sizes, in Apache Kafka this is not applicable, performance depends on the load acting on each Kafka node. Exploring the basics of scheduling and few simple partitioners gives the needed preparation for the next chapter: load balancing multiple resources represented as queues.

Chapter 5

Multiple-Queue Load Balancing

5.1 Introduction

This chapter deals with the state of the art regarding load balancing in queues, this differs from the simple scheduling theory and shifts the focus from the jobs size to the queue length.

Multiple queues is the fastest representation of modern CPUs and software. With multiple processing cores available and programs built more and more around parallelism, the job size becomes less important (thought still is) and the average queue length diminish.

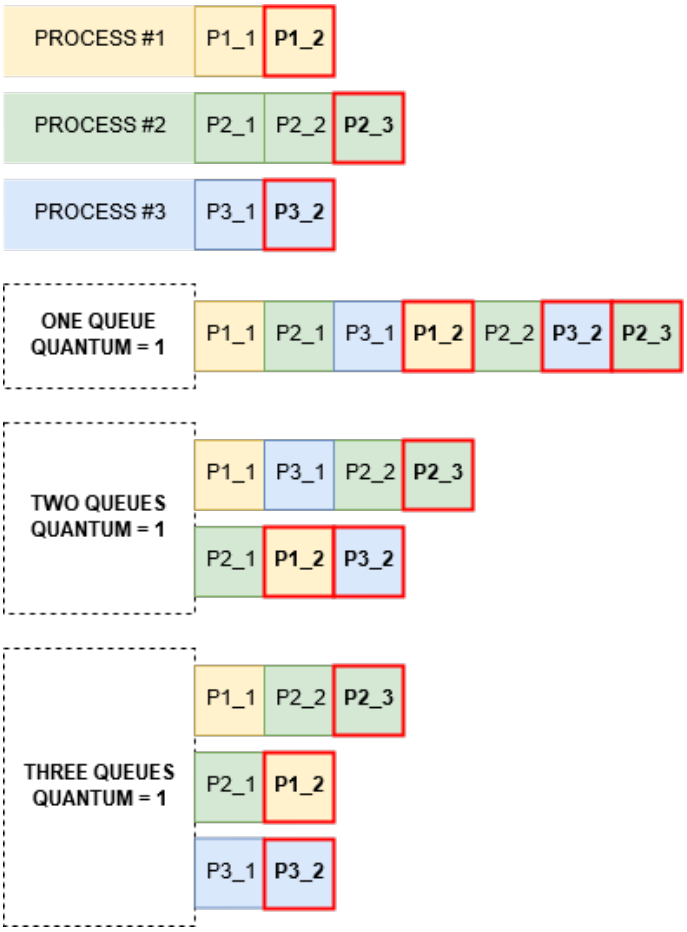


Figure 5.1: Effect of Multiple Queues Round Robin policy

Making the Round Robin policy multi-queue aware is a simple and fast solution to achieve better performance and overall lower job waiting time to be served.

Showing that single core CPUs are slower than multiple cores ones may seems obvious, but evaluating what "speed" is, may not be so trivial and is not the only metric that justifies or negates a possible money investment on a system hardware upgrade.

5.2 Single vs. Multiple Dispatcher Systems

Before introducing different balancing policies applied to multi-queues systems it is imperative to define a model of those system and locate a real world counterpart. There are two groups of multi-queue systems

- Single Dispatcher
- Multiple Dispatcher

As the names suggest, the categorization is based on the number of structures that handle the load before queues, those are the active component which enacts the balancing policy. Apache Kafka is by nature a multiple dispatcher system, where producers are dispatchers, and nodes are queues.

5.3 Parallelism Common Issues

Parallelism adoption requires a deep analysis of the system, there are multiple common problems that can arise apart from achieving proper load balancing [13]:

- Overall Performance, with stress on "overall". Not all systems need the same treatment, judging a system speed must not boil down to evaluating only one variable. The most common performance indexes are throughput and latency, but, as most network related tests teach, there may be considered also packet loss, synchronization delays, resend ratios, overhead-vs-real data ratios ...
- Complexity in implementation. Most parallel systems are modelled by the same theoretical model, load balancing algorithms follows the same principle but, at the end of the line, will always require some adjustment to adapt to every situation. More complex an algorithm is, more difference may be observed between real and theoretical performance.

Assuming instantaneous information propagation, disregarding I/O operations times and data structure memory impact are the most common practices that, as we will see, condemned perfect/top-performing algorithms to be declared as theory-only.

5.4 Dispatching Policies

Load balancing between multiple queues is often addressed as dispatching or routing problem to move the spotlight from each single message to the decision of where to move it.

5.5 Round-Robin & Random Selections

This algorithm differs from the scheduling counterpart introduced in the previous chapter (p. 51), each job must be considered atomically executable, no re-queueing process takes place once the job reaches its destination. The simplest implementation just needs to cycle between each queue, this is typically done by using the incremental jobs counter in conjunction with modulo arithmetic on queues number. The analysis on the default partitioner clearly revealed that this policy, mixed with some node/topic code handling, is the chosen one by Apache Kafka, obtaining an almost instant decision-less dispatching but sacrificing performance in not optimal situation exposing its limitation:

- Trust on the well being of the system, faults may and will result in unpredictable behaviour.
- Trust on good configuration from the system is applied to (each queue is independent from the other and it is run on a different machine).
- The only additional knowledge external to the jobs list is the number of queues.

A different approach to round-robin polices is to rely on random selection, the better the randomness used, the fairer the load distribution is. The final goal is the same, obtaining a equal split of jobs solely based on the number of queues. While random selection may not obtain optimal distribution (true perfect randomness does not exist) the main advantage on the naive rotation of selected queue is to avoid issues with localized sequential failures (assuming queue index/label is also related to topological and/or functional proximity, ence justifying that failures on queue X may also affect queues in the interval of X).

5.5.1 Expected Waiting Time

Sticking to simple systems, and emulating an Apache Kafka cluser, the ideal situation would require N equals queues and perfect distribution of the incoming jobs. Modelling the incoming flow of jobs with a Poisson process with parameter $N\lambda$, the "seen" arrival rate of each queue, assuming N total queues, is λ. Choosing a Poisson process also for the outgoing flow, rate μ, let us consider each queue as on $M/M/1$ queue [18] [34].

Analyzing the stability scenarios, were the departure rate is enough to avoid infinite queue length (arrival rate is less than service rate), the estimated waiting time of each single queue is

$$E[W] = \frac{\lambda}{\mu(\mu - \lambda)}$$

Considering the whole system almost as a black box, thus observing only the incoming $N\lambda$ and outgoing $N\mu$ flows, a new task approaching the system will see that its expected service time is, as trivially expected, inversely proportional to the number of queues (nodes/servers/...).

$$E[W] = \frac{N\lambda}{N\mu(N\mu - N\lambda)} = \frac{N\lambda}{N^2\mu(\mu - \lambda)} = \frac{1}{N}\frac{\lambda}{\mu(\mu - \lambda)}$$

5.5.2 More Complex Systems

When this policy is applied in multiple dispatcher scenarios, it strongly relies on each specific system configuration. In all those situations (Chapter 3, p. 25) where the queues are partially shared between multiple dispatchers, the system performance is at risk when the incoming flow was not designed to accommodate that specific topology, or viceversa.

5.6 Join Shortest Queue

To solve the adaptation problems of the previous policy, the pool of information used for decision making has to be changed, instead on internal metrics now each dispatcher has also access to the current state of each queue. Knowing exactly the current load of each queue, ence the current waiting time, allows this algorithm to select the best location to where send the next message, even where multiple dispatcher are present and working concurrently.

JSQ effectively minimizes the average queue length, since it chooses always the shortest one, which means that in a sustainable scenarios, faster service (out flow) than incoming rate (in flow), it optimally minimizes the overall mean delay.

5.6.1 Message Exchange

JSQ is the algorithm which achieve the optimal performance, but why it is not widely adapted, being the way to go when tackling a scheduling problem? JSQ needs the constantly updated information about the waiting time and/or length of each queue, which can not be implemented as simple in-memory sent-packets counters array, since, to work it would require

- Knowing exactly how each queue will react to any kind of incoming task
- Multiple dispatchers systems would have to work at unison

For this to work, without sharing any information between queues and dispatchers, the behavior of each queue and dispatcher to every possible flow of jobs must be predictable. In most real systems, the status of the queues or dispatchers can not be known at any given time without first querying the single component. The final real implementations use one of these two techniques, or a mix of them:

- Each dispatcher has to periodically share is own metrics array describing the status of each queue, saw from its own prospective. $\mathcal{O}(d^2)$ messages (number of dispatchers squared) sent at every periodic update. The status arrays from other dispatchers are used to fine tune the queue selection until the next update.
- Each dispatcher periodically query each queue, tracking directly the load status. $\mathcal{O}(d * q)$.

Information is not shared instantaneously, the metrics array update frequency is what mainly governs the effectiveness of this policy. Too few updates would obtain an algorithm always late in adapting to systems hiccups and changes, too much updates would waste more time communicating state vectors instead of actual data. Dynamically adapting the message frequency is a nice idea that might work in some systems, but it may result in a infinite search for a sweet spot that may not exist, and if it does, the performance are still affected by the presence of message exchange.

5.6.2 Join Shortest of Queue Selection - $JSQ(d)$

This policy keeps the JSQ performance in non-limit scenarios (when the queue is sable, the number of job arrival is lower than the queue serving time) while lowering the overhead impact [23]. The continuous exchange of messages in the basic JSQ policy is avoided by considering only a restricted number of queues to determine the less loaded one. When a message has to be sent only d queues will be queried instead of the whole set.

5.7 Join Idle Queue

Another theoretical policy that achieve optimal performance, but still later requires reality checks, is the JIQ (Join Idle Queue), which use a notify system to tell the dispatcher which queue is currently idling. Each dispatcher will hold the chronologically ordered list of incoming messages until a queue is flagged to be free, the notify process is handled by the queues themselves which, once idling, notify each dispatcher of their status.

This policy improves the overall message utility, each message has more importance and will change the state of at least one entity in the system, compared to pull-based polices where a queue may periodically ask an empty dispatcher if it has available data (worse message utility, messages may be sent even if no data is available).

As in JSQ policy there is a message overhead to be paid, but the load created is handled by the dispatcher (by being the receiving end of the notify system) and not the queues.

5.8 Conclusion

This chapter explains the theory behind the basic and only partitioner implemented in Kafka, a simple cycling selection on the available partitions. The negative sides of the simplest policy are made less impactful by the special use case represented by an Apache Kafka cluster, which, when properly configured, expects to have:

- message size variability per topic is zero or virtually zero.
- balanced ratios of partitions per topic/nodes, weighted by the expected incoming traffic, through all node
- less conflicting topics as possible, assigned to the same nodes
- dispatchers acting on the same topic as equal as possible

While not binding these rules are neither impossibly strict or extremely permissive, and when not all are met the default partitioning policy may still perform badly.

All the shown solutions rely on status information from each queues. Those policies are prohibitive to implement, most of the times, considering the overhead that comes with moving that information.

The first real fix to the paritioner will be, as shown in the next chapter, to move its focus from partitions to nodes, and once that is done, adopting a restricted version of the JSQ policy. The final goal is to achieve similar performance without tampering with the adaptability of Apache Kafka by imposing stricter configuration rules than needed.

Chapter 6

Design of new Partitioners

6.1 Introduction

Implementing an in-house partitioner is the common solution adapted by advanced users and companies with specific needs. Ad-hoc implementations can also depend on specific data or configurations with their own particularities and issues, not the scope of this document which, instead, wants have a broader approach by proposing partitioners which adapts to most cases.

Three new partitioner have been implemented, which step by step stray away from the original concept proposed by The Apache Software Foundation to tackle the data distribution assumption introduced in chapter 2 (p.17). Comparison with both healthy cluster situations and edge cases are necessary to understand the impact of the current available solution compared to new proposed policies.

Balancing records with a given key will not be considered, since in real production environments when a key is present is also already combined with a purposely built partitioner with specific balancing needs. Relying on the default partitioner omitting record keys also avoid additional computations.

6.2 New Partitioners

6.2.1 Improved Default Partitioner

The first step is to improve the basic partitioner without changing its overall policy. There are three areas where adjustments can be made:

- Code paths logic
- Data structure used
- Thread synchronization mechanisms

These changes will not solve the data distribution assumption problem but makes its effect slightly less performance taxing on the producer side.

By recalling how the default partitioner works (Fig. 2.2 p. 19) it is clear that only three cases has to be handled:

- The record to be sent has a key specified.
- The record has no key, the destination topic have at least one partition online.
- The record has no key, the destination topic have no partition available. This is an indicator of severe cluster faults.

The common path that most records will go through require no key to be specified and the cluster not to be completely offline. Since the central path does not need to know the complete list of available partitions which is retrieved anyway, re-arranging instructions gets a bigger code but requires one less use of lookup tables (blue in the image) in the most common case.

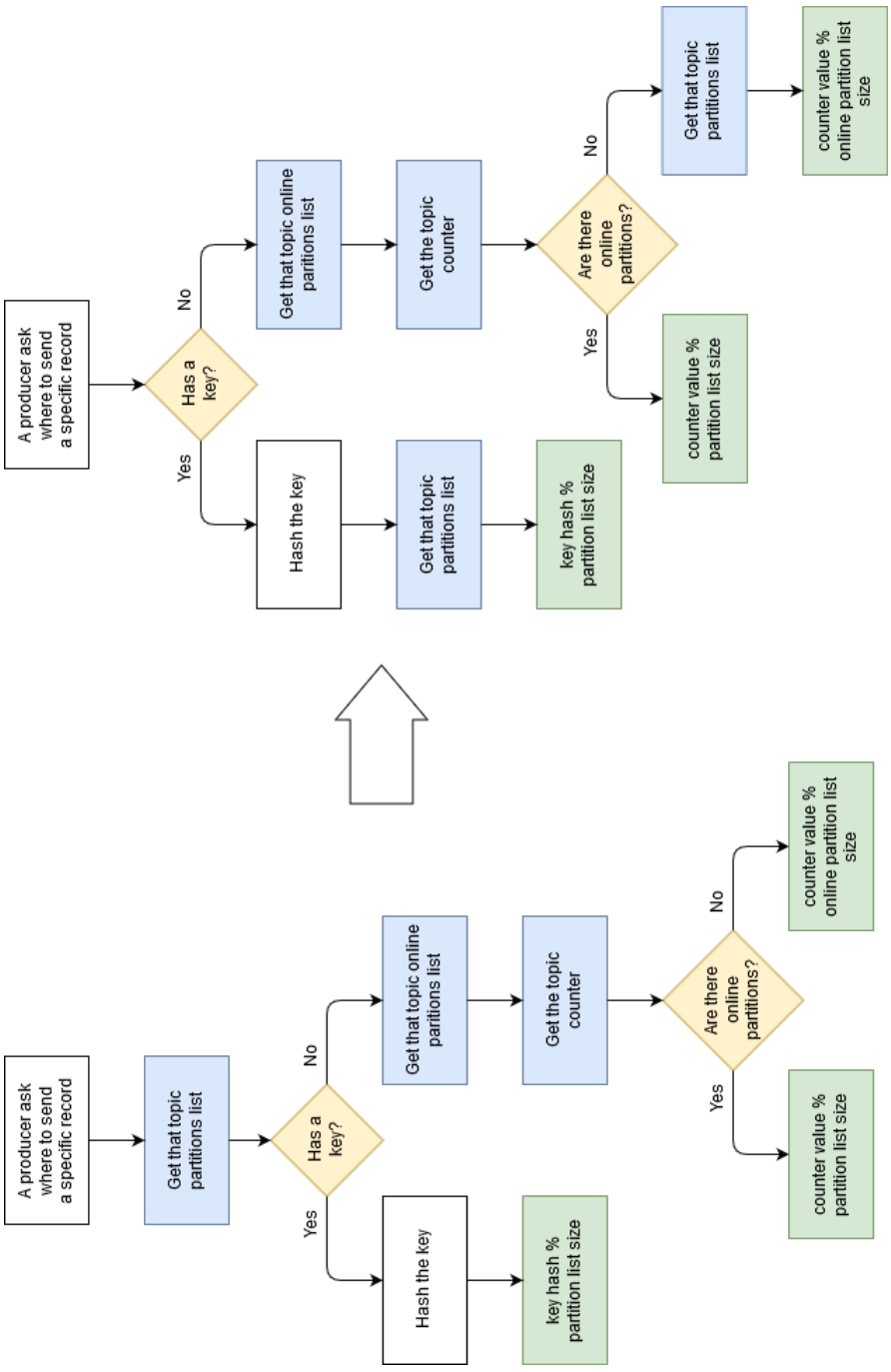


Figure 6.1: Default Partitioner logic compared to reorganized logic

The look-up table, which gets the record counter for each topic, is the only data structure located inside the partitioner. A special hash map (*ConcurrentHashMap*, built-in Java SDK 8) is used to support concurrent operations, since the same partitioner instance is designed to be called concurrently by different producers. The only concurrency sensitive code section in the original partitioner implementation is:

```
// Hash map that can be accessed/updated atomically.
ConcurrentMap<String, AtomicInteger> topicToCounter
    = new ConcurrentHashMap();

// Get the new counter value for given topic
int nextValue(String topic) {
    // Try to get counter ...
    AtomicInteger counter
        = topicToCounter.get(topic);
    // ... if still does not exist ...
    if (null == counter) {
        // ... create a new one with a random value
        counter = new AtomicInteger(
            ThreadLocalRandom.current().nextInt()
        );
        // Try to put the new counter in the map
        AtomicInteger currentCounter =
            topicToCounter.putIfAbsent(
                topic, counter
            );
        // If someone did it before me ...
        if (currentCounter != null) {
            // ... save the correct counter (theirs)
            counter = currentCounter;
        }
    }
    // get the counter value and increment it
    return counter.getAndIncrement();
}
```

Without diving to deep on code design and meaning, everything is set up to handle rare, but possible, cases where two (or more) producer query the same partitioner instance, passing the same new topic name never used before as parameter. The whole process can be speed up with low level thread synchronization primitives and much simpler data structures. The final updated code will be:

```
// Simple Hash Map
HashMap<String, AtomicInteger> topicToCounter =
    new HashMap();

int nextValue(String topic) {
    // Fast Weak-get. Do not care about sync.
    AtomicInteger counter =
        topicToCounter.get(topic);
    if (null == counter) {
```

As before the code checks if the counter does not exist or the weak-get failed (tried to get the counter just before someone else put it there). The operations on the synchronization-less data structure are said to be "weak", but are faster.

The next block is executed only if no counter is retrieved, the whole block is synchronized on the same object, so there will always be at most one thread running it. The only set operation can be found in this block, making all writes to the hash-map concurrency free. Reads are not always concurrency safe, but null-safe checks are in place to avoid issues and to keep the code fast.

Another null-check is performed to guess if the thread running this block is the first to reach that instruction (given the same topic). The first if is needed only to avoid synchronization all together when the hash-map filled.

```
        synchronized (topicToCounter) {
            // Perform another weak-get
            counter = topicToCounter.get(topic);
            // if still does not exist...
            if (null == counter) {
                // ... I'm the first to get here
                // create a new counter
                counter = new AtomicInteger(
                    ThreadLocalRandom.current()
                        .nextInt()
                );
                // add it in the map
                topicToCounter.put(topic, counter);
            }
        }
```

Now that a counter has been finally retrieved, increment it and continue with the partitioning policy.

```
    }
    // get the counter value and increment it
    return counter.getAndIncrement();
}
```

6.2.2 Node Partitioner

The first substantial modification that transition from partition balancing to node balancing, is to adapt the default partitioner to keep the same Round Robin policy but making it act on nodes rather than partitions. Missing data structures must to be implemented in the Apache Kafka codebase. Continuing with the lookup-table approach:

- A main hash map is needed to pair each topic to the list of nodes which handles at least one partition of that specific topic
- An hash map for each node, that pair each topic to a partition list only for those partitions present in that node.

The first hash-map is used to select which node to send the request to, rotation between nodes is achieved with the same counter-based Round Robin technique as the improved partitioner. Once the node is selected, its representation in partitioner memory is accessed to retrieve the second level hash-map, which will finally select the partition within the selected node.

Unseen by the partitioner the node object representation has another per-topic counter selection, it helps the node internal rotation of selected partition, but it does not need another dedicated lookup table since this special counter can be saved together with its own partition list.

Keeping the Round Robin selection policy for partitions with the same leader node is extremely important, when a fault is resolved some of those partitions will return to its original owner.

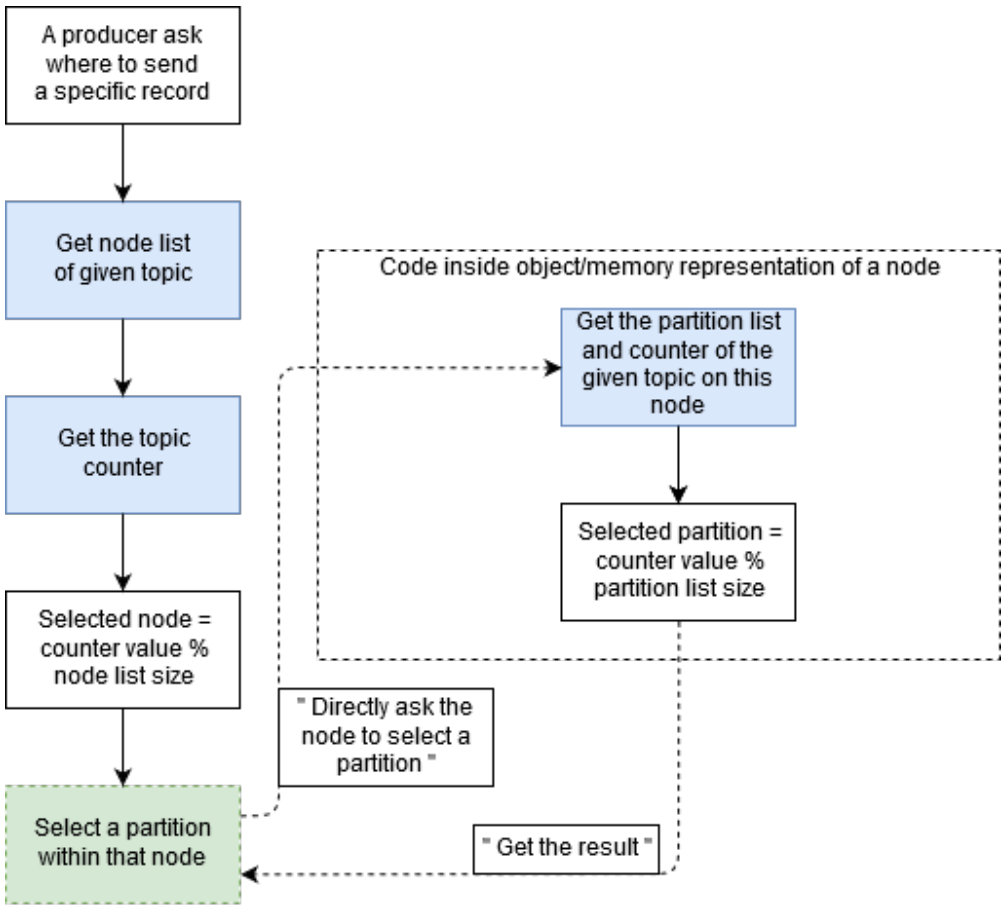


Figure 6.2: Code exectuion flow of the Node Partitioner

This new partitioner implements also the first simple workaround to lower hashing operations, double valued maps. Is pointless to have two maps with the same sets of keys and with the read/write operations always happening on the same key. Having a structure mapping topics to pairs $(nodes\ list, counter)$ reduce the needed hashing computation by 1. In this work the main hash-maps used by the original partitionier have been kept also in the improved partitioners, because the final performance comparison would have been more about coding tricks and not about meaningful algorithms changes.

6.2.3 SQF Partitioner

The name is the acronym of "Shortest Queue First", hinting that, when a record has to be sent, the node selection is based on usage metrics, trying to choose the most unused node which will take less time to compute our request. The final partitioner evolution to achieve per-node data distribution balancing requires to ditch the Round Robin policy and implement a tracking mechanism on node usage. When a producer want to send data to a topic the less used node will be found with a simple linear search.

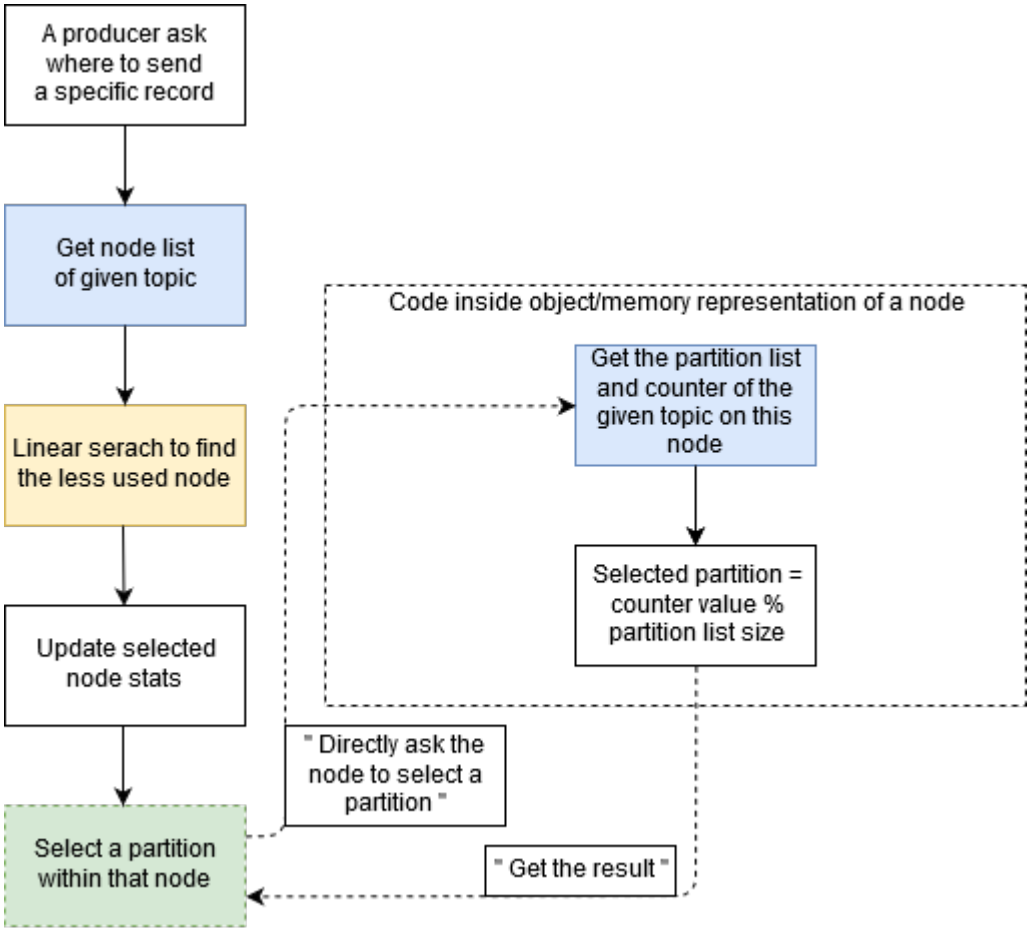


Figure 6.3: Code exectuion flow of the SQF Partitioner

Integer overflow protections are in place to avoid wrong selection of nodes once binary representation limits are hit. Since both the linear search and the overflow checks are bound by the number of nodes in the cluster, the overall performance is not affected as common logic suggests.

The node population in real environments is never huge enough to raise concerns on linear operations performances, and the topics spreading between nodes makes it even less meaningful.

Trading simplicity with more advanced programming may actual results in speed losses caused by cache misses and/or directly handling improved data structures.

6.3 Advanced Proposed Improvements

In both Node and SQF partitioner the hash key is ignored because it defeats the whole node rotation policy, it could be used to rotate partition selection within a node, but it may have negative impacts when a cluster is healthy and well distributed. Applying hash-based node internal partition rotation selection is useless when the number of available partition per-node per-topic is close to one, which is the expected and suggested configuration of any healthy cluster.

Apache Kafka code maintains a coherent memory representation of the cluster topology, when it changes no operation is performed until the representation is properly up to date. Having a reliable, trustworthy and unchangeable representation of the cluster can be used to improve the overall usage of that information:

- Most hash map describing topology could be pre-computed only when a cluster changes, not while the partitioner is running (thus removing any need of synchronization).
- Some hash maps are created knowing already which elements will contains and be never changed. With some minimum hash key collision checks perfect hash maps could be built, when possible, wasting less memory and improving performances
- Double value hash-maps (Node and SQF partitioner use a basic implementation of that concept) would lower the amount of hashing computation by one.
- Currently topics are represented as simple string objects, making them more complex and capable to store topology information, related to themselves, could eliminate the need of some look-ups mechanism entirely.

All of these proposed code innovations need deep changes on how Apache Kafka works, requiring big code restructuring and possibly switch currently adopted programming standards.

More reasons that condemned the implementation of more advanced programming techniques are:

- Not focused on performance, but more on code utility. Complete implementation dependent on single company choice.
- That requires a small overhaul of many different sections of Apache Kafka code. Correct and proper performance evaluation needs at least a small programming team with deep knowledge of Apache Kafka code standards.
- That requires Advanced data structures purposely built for Apache Kafka specific problems.
- The resulting code would be visibly different, making the comparisons more like a code design competition and not a performance evaluation.
- Pre-computing lots of information will result in faster code but will slow down topology change adaptation, since its where pre-computing would take place.
- Extensive tests must be performed on both small and big scale failures.

6.4 Comparisons between Default and New Partitioners

Data Distribution Comparison

At first both Node and SQF partitioner does not solve the data distribution assumption problem, emerged with the default partitioner. A producer which simply split data between nodes does not always achieve the perfect distribution when other producers are in place, asymmetrically assigning multiple topics between nodes is enough to void most improvements provided by the advanced partitioners.

To better understand the common partitioning issue shared between all partitioner, imagine this prototype Apache Kafka cluster with an evident Node/Topic-Group overlapping problem:

- Two topics, **Topic A** and **Topic B**.
- Three nodes, **Node 1**, **Node 2** and **Node 3**.
- **Topic A** has one partition in **Node 1** and two in **Node 2**.
- **Topic B** is evenly split between **Node 2** and **Node 3**
- Two producers, **Producer X** and **Producer Y**, adopts the same node based partitioning policy but are executed by different machines.
- **Producer X** sends α records to **Topic A**
- **Producer Y** sends β records to **Topic B**

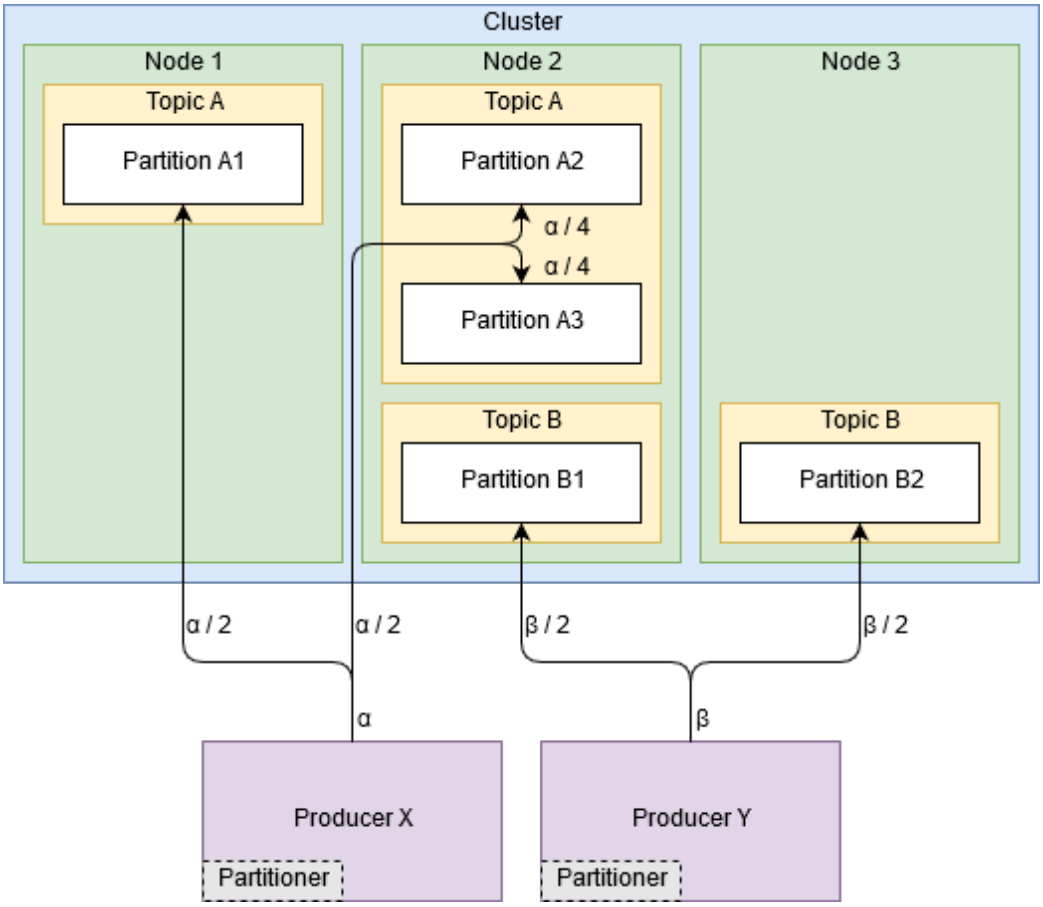


Figure 6.4: Prototype Cluster with any node based partitioner

The first Node/Topic-Group, made by **Topic A** with **Node 1** and **Node 2**, clearly overlaps on **Node 2** with the second group, made by **Topic B** with **Node 2** and **Node 3**.

Both producers (and their partitioners) are running on different machines, thus not sharing any node metrics data, by their point of view the data is perfectly split, but the real data count differs from node to node. Switching to the default partitioner will get even more unbalanced partitions.

		Node 1	Node 1 Sum	Node 2	Node 2 Sum	Node 3	Node 3 Sum
Node / SQF	Prod. X	$\frac{\alpha}{2}$	$\frac{\alpha}{2}$	$\frac{\alpha}{2}$	$\frac{\alpha+\beta}{2}$	-	$\frac{\beta}{2}$
	Prod. Y	-		$\frac{\beta}{2}$		$\frac{\beta}{2}$	
Default	Prod. X	$\frac{\alpha}{3}$	$\frac{\alpha}{3}$	$\alpha\frac{2}{3}$	$\alpha\frac{2}{3}+\frac{\beta}{2}$	-	$\frac{\beta}{2}$
	Prod. Y	-		$\frac{\beta}{2}$		$\frac{\beta}{2}$	

This test cluster has been designed to easily expose two main caveats that real systems could have:

1. It is the resulting configuration after a fault or a bad initial setup. Either the node dedicated to handing **Partition A3** crashed and **Node 2** became the new leader of it, or the whole **Topic A** has been badly designed from the beginning, with three partitions assigned to only two nodes.

2. Producers of different data have, as expected, different topic to write to, but the sets of assigned nodes of each topic partially overlaps.

Both conditions may come from bad cluster design, resulting in partially overlapping of the assigned nodes sets of different topics, complete or no overlapping would have solved the issue.

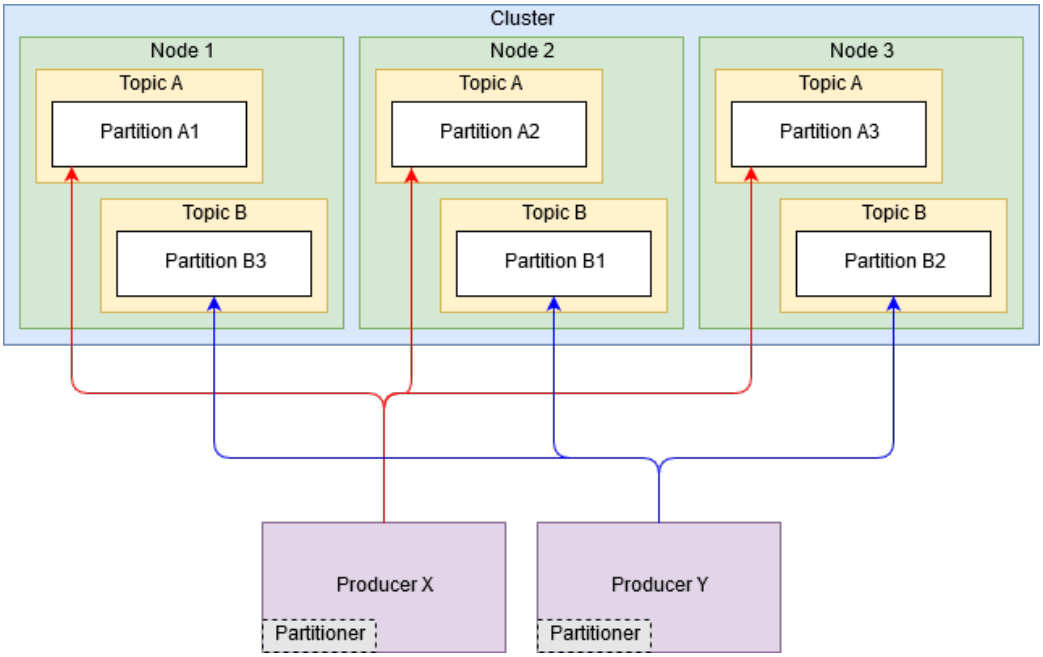


Figure 6.5: Prototype cluster with no partial overlapping

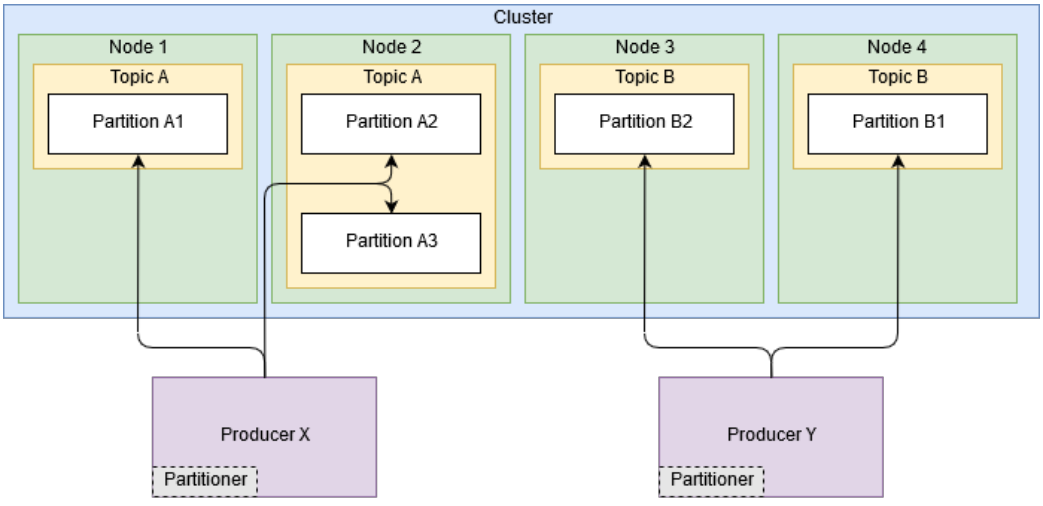


Figure 6.6: Prototype cluster without node-topic overlap

Both solutions require changes in cluster topology by creating new partitions, changing partitions assignments or adding new nodes to the cluster. The complete overlapping may reduce the performance of those topics which were previously handled by dedicated nodes, they would have to share compute power, I/O and network time with other topics. When possible, complete node separation of topics is the suggested Apache Kafka configuration, and if not applicable cluster-wide to each topic and least try with sub-sets. Configurations built purposely wrong to get bad performance are allowed but not contemplated in the official documentation, and so neither in this thesis.

It has to be noted that the no-overlap configuration (p.80) did not required a complete partition reassessment. **Topic A** still have three partitions on just two nodes, assuming this situation to be the result of a fault, only by using the Node or SQF partitioner the data balance can been kept. The default partitioner would not have adapted to the topology change and wrongly balanced **Node 1** and **Node 2** with a 1:2 ratio.

Any partitioner that relies on any kind of metrics suffers from the same structural limit, the memory is not shared with other instances running on different machines. Having a global state shared between each separated partitioner instance would be the perfect theoretical solution, but impossible since the overall system would waste more time sharing updated state data than working on the data itself.

When partial overlapping can not be eliminated, if all the producers which are sending data to those partially overlapping topics sub-sets run on the same machines, the SQF partitioner still solves the distribution problem. The SQF partitioner is less performing, since it needs a linear search to obtain results, but allow more flexible cluster configurations.

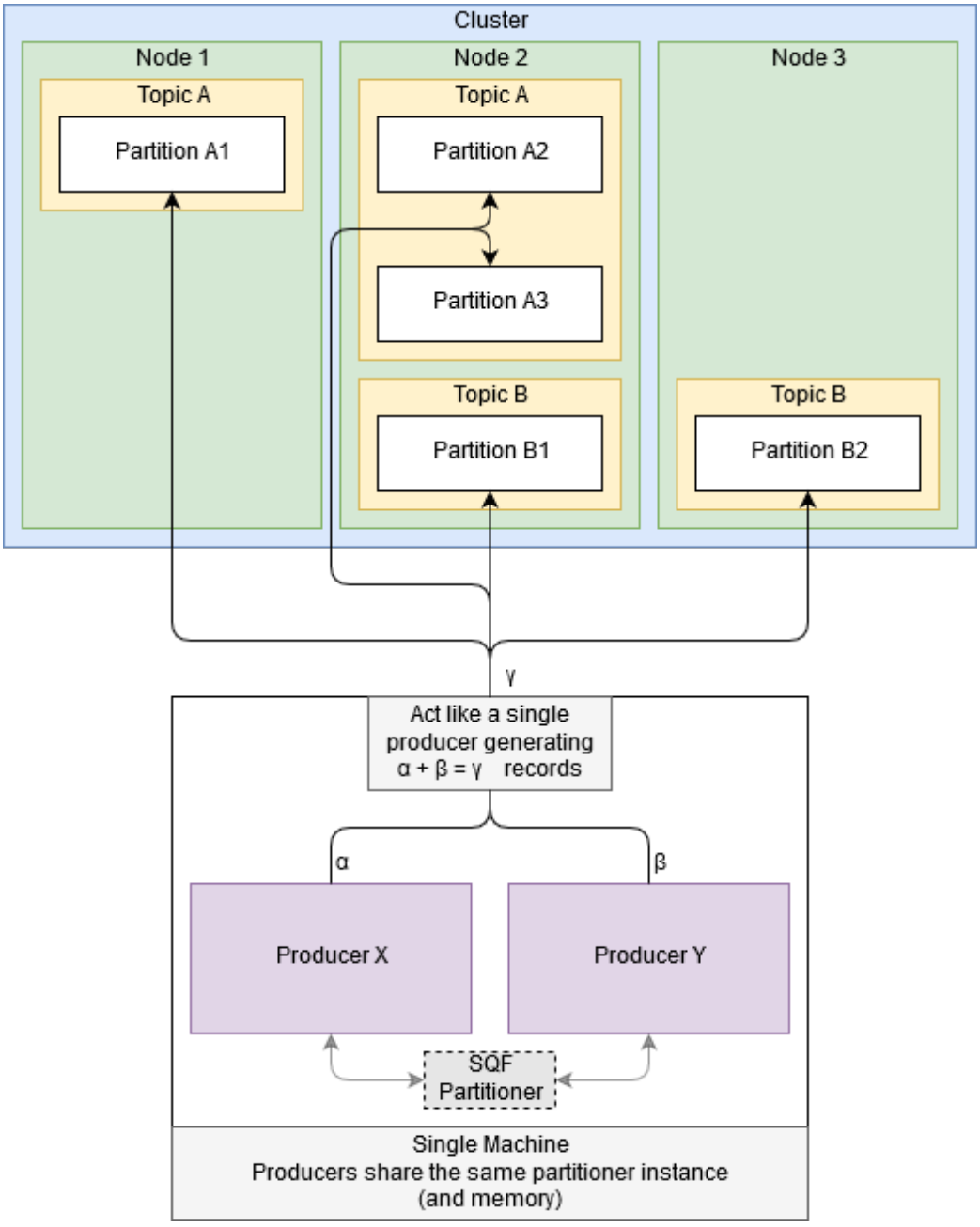


Figure 6.7: An SQF partitioner instance shared between two producers

Keeping the bad cluster topology, but forcing the producers to share the SQF partition instance, still helps distribution. The flow outgoing from both producers is combined, but the data is still correctly dispatched, records coming from **Producer X** still goes only to **Topic A**, same goes for **Producer Y** and **Topic B**, the single flow to each node is simply weighted to keep the overall balance assumption valid. Partition rotation within a specific topic and node is still in place, whatever amount of data is sent to **Topic A** inside **Node 2** is perfectly split between **Partition A2** and **Partition A3**.

Complexity Comparison

Hash maps are the ever-presents data structure in the Apache Kafka code, the used implementation is the default one built-in inside Java SDK, with worst case scenarios requiring $\mathcal{O}(\log n)$ operations to access the data, but in most real applications this is not the case. The amount of elements in any map is bounded by overall number of nodes, topics or partitions, and usually only subsets of them, a collision is really unlikely to happen, multiple collisions on the same values even less, making the theoretical worst case scenarios not meaningful.

Focusing only on the Apache Kafka use case, hash maps access time can be considered to be $\mathcal{O}(1)$, leading to this time complexity comparison:

	Default	Improved		Node	SQF
		Some partitions available	All partitions OFFLINE		
Hash Map #access	3	2	3	3	2
Linear search	-	-	-	-	$\mathcal{O}(n)$

- n is the maximum number of nodes acting as leaders of any partition belonging to any given topic. In a perfect cluster n equals p (partitions number of any topic), because any node is leader of at most one partition of any given topic.

- Each hash map access counter can be reduced by at least one unit, either by implementing the proposed solutions described in section 6.3 (p.75), or by replacing the partition Round Robing policy with records hash key based selection.

Space complexity is not as concerning as time complexity, the explosion of asymptotic behaviors is possible in theory but impossible in real world applications. The total number of topics, nodes and partitions is bounded, usually by financial costs associated with designing overpowered systems for a given tasks, nevertheless the complete space complexity analysis is:

- **Default and Improved Partitioner** $\mathcal{O}(t)+2*\mathcal{O}(t*p) = \mathcal{O}(t*p)$. One mapping topics to counter, and two mapping topics to list of partitions.

- **Node Partitioner** $\mathcal{O}(t) + \mathcal{O}(t) + n * \mathcal{O}(t * (p + 1)) = \mathcal{O}(n * t * p)$. One mapping topics to nodes, one mapping topic to counter, and n mapping topics to pair (list of partitions, counter).

- **SQF Partitioner** $\mathcal{O}(t) + n * \mathcal{O}(t * (p + 1)) = \mathcal{O}(n * t * p)$. One mapping topics to nodes, and n mapping topics to pair (list of partitions, counter).

A similar approach used to improve the time complexity can be applied also here, implementing advanced data structure proposed in previous sections (2.3.4) or replacing the partition Round Robing policy with records hash key based selection, remove the need of counters (and the space used by to store them).

6.5 Halfway Cluster Collectors

The message life cycle in Apache Kafka is the chain of process that start with the data generated from a producer, the data sent to the cluster, stored in the cluster and finally received by a consumer.

Trying to balance data distribution early in the producer side of the message life cycle, permanently improves performance regardless of how many consumers will to access that data and it require only one computation per message. In those systems where modifying the cluster structure is prohibitive and multiple numerous producers do not have additional computational power, switching partitioner may not be possible, but adding collectors that implements advanced partitioning may help. This solution is adopted by Netflix to provide its service [38].

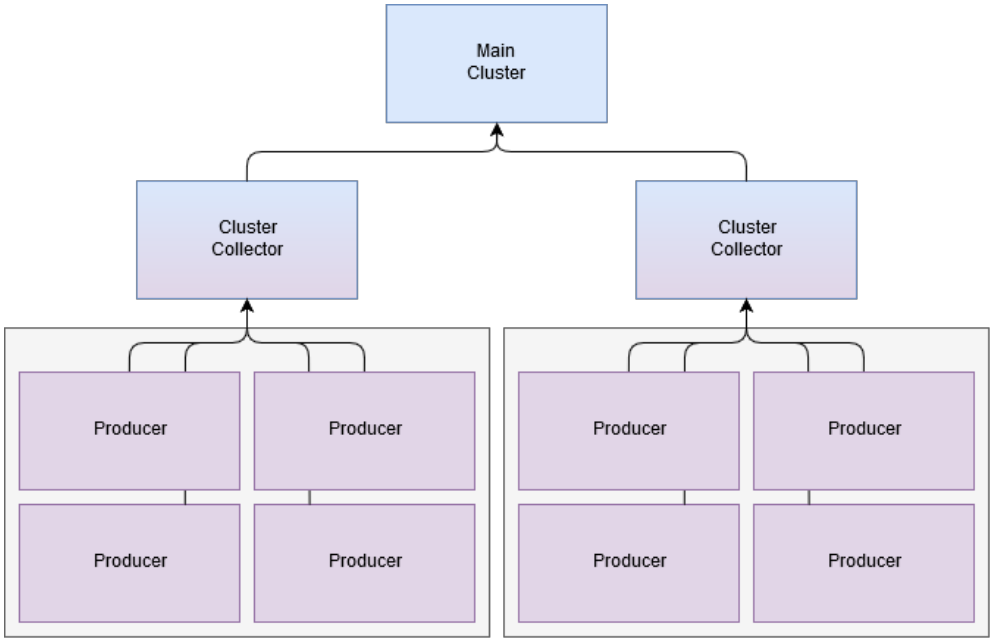


Figure 6.8: One main cluster that sees only two producer which are actually another two clusters handling four producers each.

A collector is small cluster placed between the real cluster and a subset of its producers, it needs more power than a singles producer but not as much as the main cluster. The role of a collector is to gather a fraction of the messages that would have been sent to the cluster, apply the advanced partitioning policies and forward the data to the main cluster. If possible the usage of a cluster comes with lots of benefits:

- Minimal configuration on the producers side (they have to point to a collector instead of the main cluster).
- Plugging a new machines into a system is cheaper than renewal of already existing components.
- No need to change the cluster node/topics/partitions structure if there are enough collectors to balance the previously unbalanced incoming message flow.
- No consumer logic is present. The halfway cluster is seen as a normal cluster from its producers, and acts as one while sending data to the main cluster.
- The subset of producers assigned to a collector may be grouped together following a geographical location criteria.
- Having to deal with less producers, the collector cluster structure configuration is much more simple compared to the main cluster (and must be, to successfully fulfill its role).

Using additional clusters also comes with its drawbacks:

- Single point of failure, losing a collector will cut the communication with the main cluster (if no redundancy or fallback mechanics are in place).
- The time, for a message, to be generated from the producer and stored into the main cluster is inevitability increased. It may be impossible to use collector in all those realities were absurdly strict timings are necessary, like when Apache Kafka is used to replicate or react to events.
- Steepest learning curve to adopt it properly, require precise and specific tweaking to each scenario.

6.6 Conclusion

The Apache Kafka codebase has lot of room for improvements, the implemented proposed partitioners discussed in this thesis are just an initial approach to optimizing the whole project.

The default partitioner can not adapt to every topology changes caused by faults or bad cluster configurations, making time spent recovering from those unbalanced typologies even more critical. Node and SQF partitioners solve the fair data distribution assumption for different fault realities, but still relies on good cluster configuration.

The perfect solution with a global shared state between different instances, running on different machines and thus memory, of the same partitioner policy is not possible, it would require too messages exchange. To get closer to perfection, a good cluster design has to be studied early with no partial node-topic overlapping issues, and to avoid crippled distribution a node based partitioner must be used. If node-topic overlapping can not be avoided but the producers acting on the same colliding sets run on a single machine, the SQF partitioner is a good choice to properly distribute data. In the extreme cases where big changes cannot be made to a running Kafka system, adding midway clusters, with an advanced partitioner, that group producers and act as funnels to the main cluster is another possible solution.

Chapter 7

Performance Evaluation

7.1 Introduction

Testing every possible scenarios and use case is, of course, impossible. Finding a common ground is the necessary step to properly evaluate any new partitioning policy. The partitioner is the key component to be tested, directly or indirectly the whole system performance depends on it.

The publicly distributed release package of Apache Kafka provides a built-in tool suite with a wide range of tests that evaluate any system component. A test cluster should switch between different configuration to simulate fault scenarios, where the common partitioner does not perform at its best.

The results does not crown a partitioner as absolute best, differences are actually expected, the final choice should be weighted by what kind of usage and configuration the real target cluster will have.

7.2 Focusing on the partitioner

The final user is only interested in the metrics focused on the consumers, the first component in a performance dependency chain which connect, in sequence, the consumer, the cluster, the data distribution assumption and the partitioner policy.

1. The consumers get data from the cluster.
2. The power of the cluster depends on the data-node distribution quality.
3. The data is distributed by following a specific partitioning policy.

The partitioner is the key component that drives how the whole cluster will perform.

Regardless of the macro system configuration, bottlenecks and hardware performance, the only common problem between any real production environments is how the partitioner works in healthy cluster situation and how it reacts to big faults.

It should also be noted that the fault state of a cluster should be transient, any serious company is expected to aid any problematic offline node in short time and/or provide enough backup nodes ready. Even if speed and latency will be our main final yardstick, to evaluate when and which partitioner satisfy our needs, the results should be not be read just as raw performance indexes but more as overhead to pay for features we want to have.

7.3 Cluster Configuration

Since a change in behavior has to be tested, a multiple-scenarios model must be followed. No artificial "fault" event will be created, instead different configurations will represent different states of an hypothetical cluster, this avoid any data contamination coming from the fault-reassessment process. Reactions to faults are influenced by the partitioning policies chosen but mainly dependent by disks I/O and internal network performances, this is not true in intentionally highly unbalanced clusters or in catastrophic fault scenarios, that is why the fault recovery process will not be simulated or tested.

To avoid any interference external to the mere partitioning process, the most basics cluster configurations, which still take advantage of parallelism, are the only possible choices. Two main configuration will be set up, balanced and unbalanced, both with only one topic and two nodes, but different partitions amount.

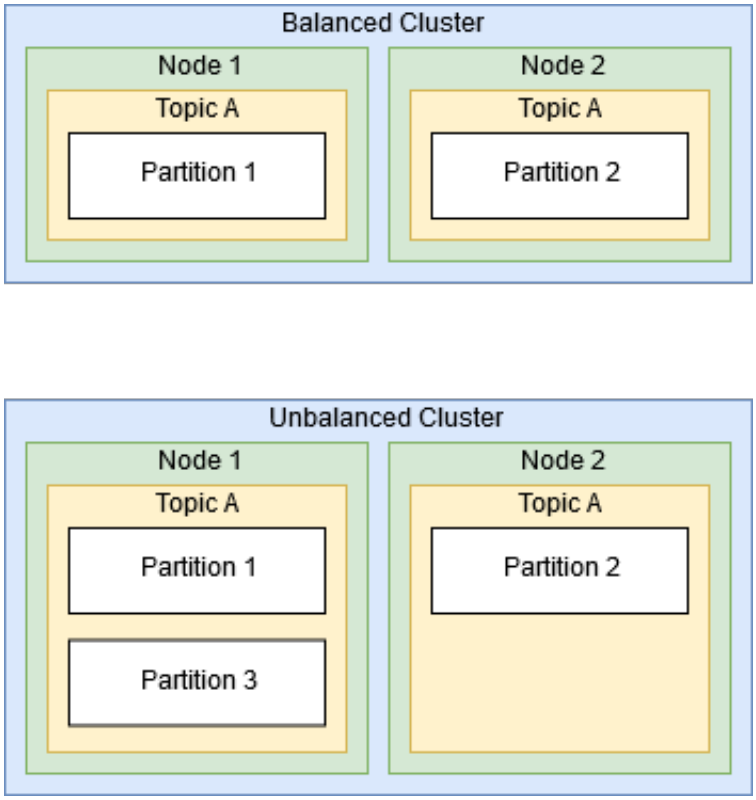


Figure 7.1: Balanced and Unbalanced replication-less clusters

These base cases, while helpful to clearly compare the raw overhead a policy might have, does not shed enough light on how real clusters are impacted. Real working environments make use of replication, one of the flag features of Apache Kafka. Two new test clusters need to be created with replication active.

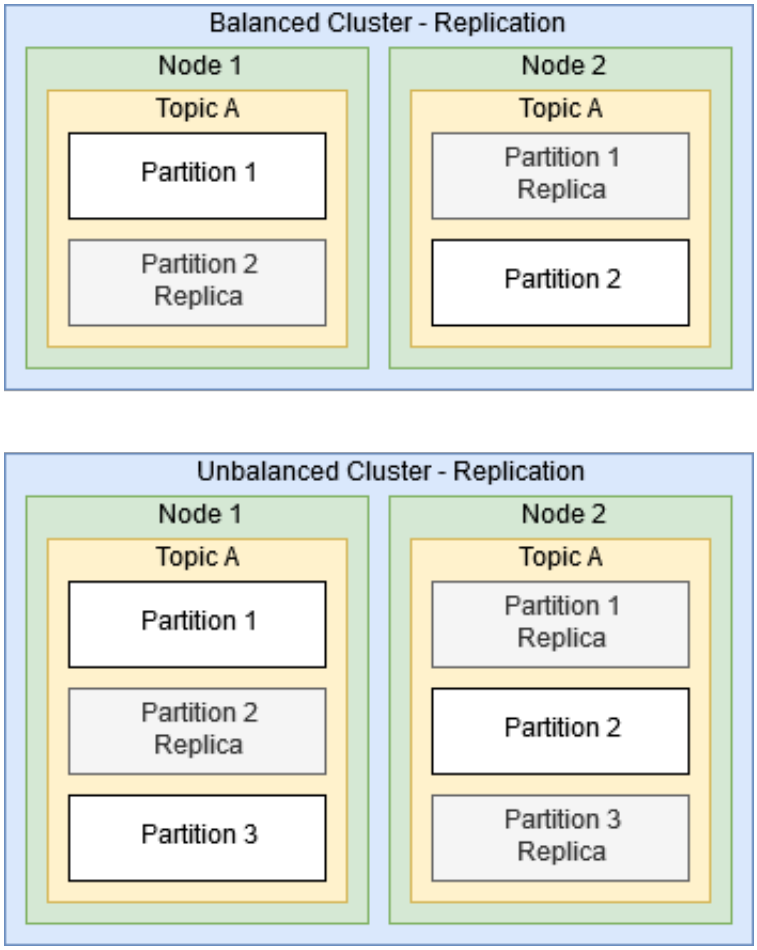


Figure 7.2: Balanced and Unbalanced cluster with replication

Replication is considered off only when the replication factor is sets to one, its minimum. Assign only one node to each partition means allowing only the leader node to be created. The first two simple test cluster configurations have a replication factor of 1, while the others use the biggest value available, 2.

Switching between every different configuration for every test, which the cluster will go through, can results in time loss, to avoid it, a single configuration was chosen with multiple topics representing all the four possible distinct clusters.

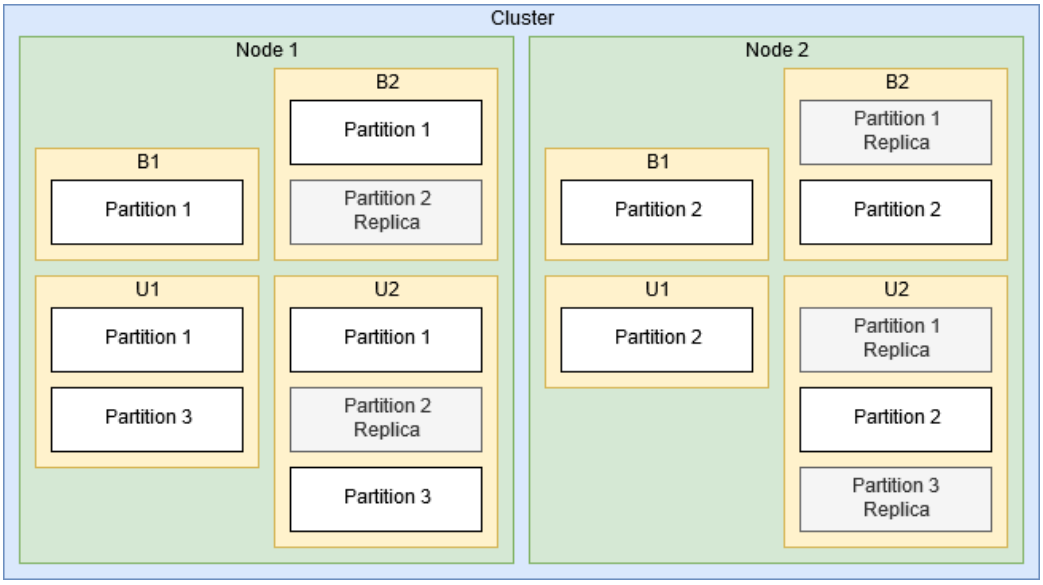


Figure 7.3: How the test cluster has been configured

Each topic represent a different simulated scenario:

- **B1** - Balanced partitions/node ratios with no replication.
- **B2** - Balanced partitions/node ratios with full replication.
- **U1** - Unbalanced partitions/node ratios with no replication.
- **U2** - Unbalanced partitions/node ratios with full replication.

The unbalanced topics represent a possible failure on larger clusters, both **U1** and **U2** act as cluster where the node leader of partition 3 failed and everything fell back on the only available replica.

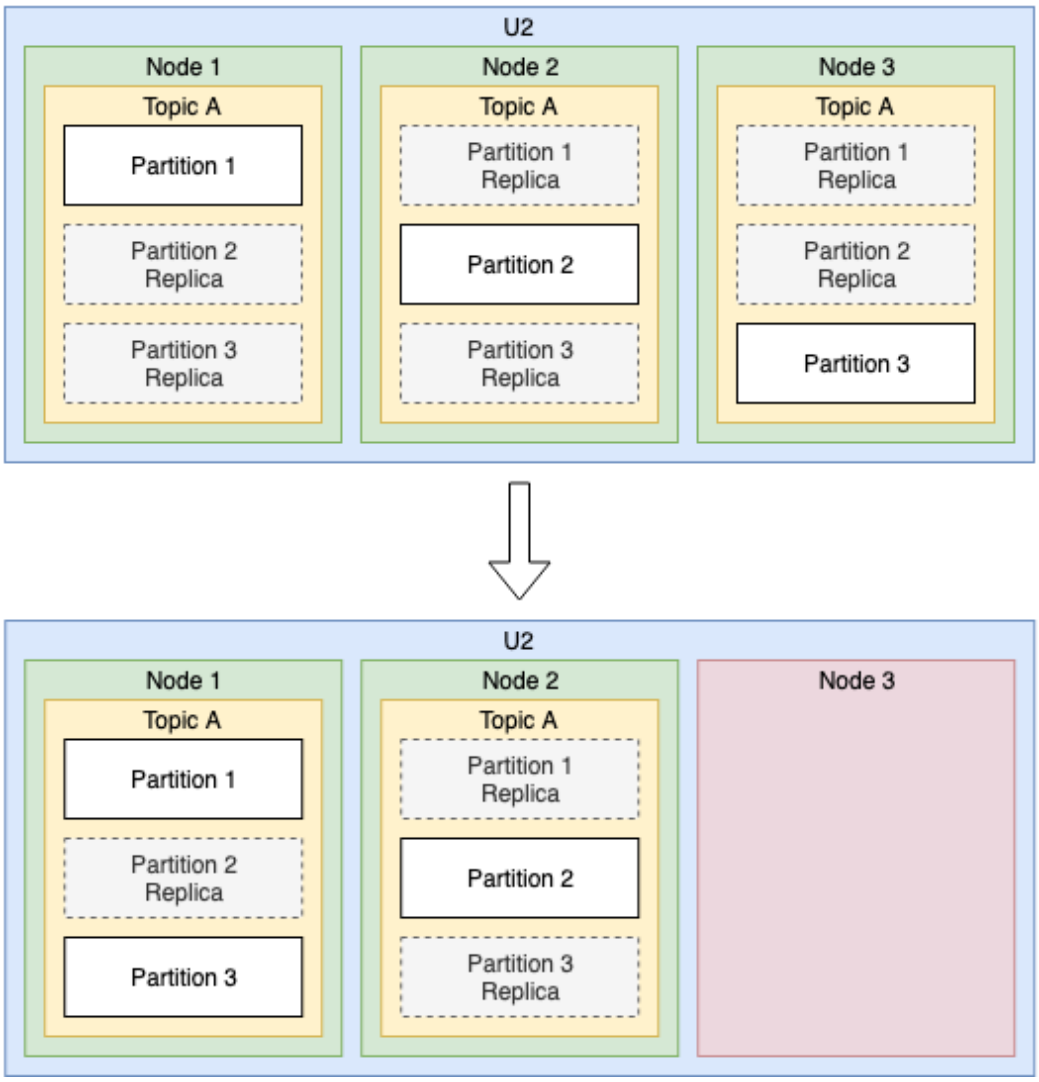


Figure 7.4: What real world case U2 is ment to represents

The **U2** case carry the most significance because its the closer to the real world context, a cluster with some degree of redundancy and replicas is having an offline node.

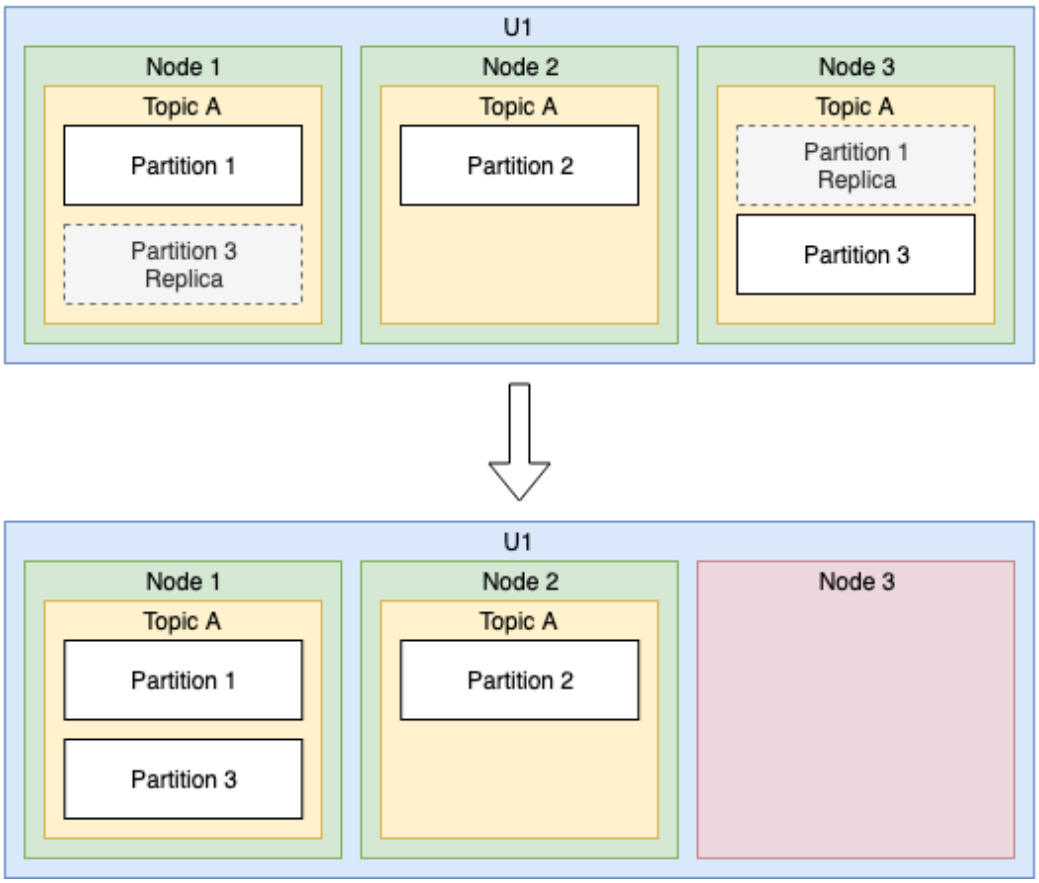


Figure 7.5: What real world case U1 is ment to represents

U1 is almost never present in real world applications, but is useful to repeat the test **U2** without the possibility to tamper the results with the impact of replicated partitions.

The remaining balanced topics, **B1** and **B2**, are used to test algorithms overhead when the system is healthy. **B1** is not meaningful for real implementations, since removing replication and renouncing to fault tolerance is foolish, but is useful to compare it with **B2**, like **U1** for **U2**.

7.4 Test Software Configuration

When using the full suite, included with any of Apache Kafka package, there is no need to build ad-hoc tools or simulation for testing any major components. The tool "`kafka-run-class.sh org.apache.kafka.tools.ProducerPerformance`" became a de facto standard to benchmark producer performance, that command, executed within the Apache Kafka binary directory, dynamically start a program which acts as producer and stress a specified topic with custom parameters that shape the generated load. Properly use the available test tools is the main focus of this section, a correct parameter selection is the key cover as much production environment realities as possible.

Mocking real environments does not just needs partitions and replicas, it should also mimics possible synchronization functionalities. The two replication enabled configurations could be split again to take into account blocking synchronization or not, this is not needed since is the single producer decide whether or not wait for the replica synchronization process to finish. Testing clusters with only two nodes does not allows to evaluate the impact of partitioning policy on gradually increasing blocking synchronization, the producer either waits for all nodes, two, to acknowledge the successful write to a partition or waits only for its leader.

Aside from the acknowledgment rate no other exotic settings are needed to test Apache Kafka, as Jay Kreps, one of the main developers, wrote [19]:

> *[...] For benchmarks that are going to be publicly reported, I like to follow a style I call "lazy benchmarking". When you work on a system, you generally have the know-how to tune it to perfection for any particular use case. This leads to a kind of benchmarketing where you heavily tune your configuration to your benchmark or worse have a different tuning for each scenario you test. I think the real test of a system is not how it performs when perfectly tuned, but rather how it performs "off the shelf". This is particularly true for systems that run in a multi-tenant setup with dozens or hundreds of use cases where tuning for each use case would be not only impractical but impossible. As a result, I have pretty much stuck with default settings [...]*

The tests performed in that blog article were used as base for the tests in this document, keeping most settings (changing only cluster configurations and amount of packet sent, 100 million instead of 50).

The single prototype command that will launch a specific instance of out battery of tests is:

```
<KAFKA_DIR>/bin/kafka-producer-perf-test.sh          \
    --num-records 100000000 --throughput -1          \
    --record-size 100 --producer-props               \
    partitioner.class=<POLICY_NAME> acks=<ACKS>      \
    --topic <TOPIC_NAME>
```

Those commands describe a continuous flow of 100 million packets of 100 bytes each with no set cap on the throughput (mimicking the original Kafka benchmark settings), limited only by the hardware and the policy chosen. The placeholder values in the previously shown code are:

$KAFKA\_DIR$ - Install location of any Apache Kafka suite.

$POLICY\_NAME$ - Selected partitioner.

$ACKS$ - Wait for replica sync (1 = OFF or -1 = ON).

$TOPIC\_NAME$ - Topic name (U1, U2, B1 or B2).

As presented, the previous command do nothing by itself and needs other parameters to properly function (cluster IPs, JVM args, ...), manual configuration is needed to to replicate the tests.

7.5 Hardware

Two identical machines have been setup to represent each node.

CPU - Intel Pentium Dual-Core E6300, clocked at 2.8 GHz, NO Hyper-threading, Cache 64 KB L1, 2 MB L2.

HDD - 500 GB, 5400 RPM, 16 MB cache.

Motherboard - FUJITSU D3011-A1 (Chipset Q43).

RAM - 2 x 2 GB of DDR3 RAM, clocked at 1067 MHz (Dual Channel enabled).

SO - Customized Debian based Linux distribution, tuned to focus on network data transfer and with a minimal software installation impact, just the basics to run Apache Kafka and diagnostics tools.

The individual performance of each single node is not important, the components list reveals multiple possible bottlenecks, which is actually the whole point. Changing partitioner is not the universal solution to avoid any kind of bottleneck, but rather a way to move said bottleneck closer to the real hardware limits.

Without a bigger amount of machines more complex clusters were impossible to setup, but the low specs compensate for it, making it easy to rapidly approach the cluster breakpoints, and possible to analyze various policy impacts even on small sets of tests. An excessively powerful cluster would have required a proportionate amount of producer power, to avoid overshadowing the partitioning policy differences by perfectly handling everything thrown at the test cluster.

7.6 Results

To evaluate each partitioner five main metrics are recorded during each session:

- **Throughput**, MB per second sent by the producer.
- **Average Network Latency** (milliseconds), experienced by the producer for a single packet. The averages considering only the 50th, 95th, 99th and 99.9th percentiles are also shown.
- **Maximum Network Latency** (milliseconds), not excessively meaningful or accurate, gives a quick hint on spikes or hiccups issues.
- **Average I/O Latency** (nanoseconds), caused by the cluster storage solution.
- **Time to completion** (TTC) (seconds) a single test on the selected topic.

Each test is run multiple times to compute a 99% confidence interval, which can tell how stable a partitioner is (considering a specific metric).

7.6.1 Default Partitioner

This is the out-of-the-box experience given by the software that comes with the Apache Kafka suite. In the launch command line the partitioner parameter can be omitted and this policy will be selected as fallback.

Throughput	62.48	[	60.56,	64.41]	MB/s
Average Net. Latency	116.99	[	94.67,	139.32]	ms
Latency 50th Percentile	1.78	[	1.53,	2.03]	ms
Latency 95th Percentile	688.67	[	556.15,	821.18]	ms
Latency 99th Percentile	1347.44	[	1162.41,	1532.48]	ms
Latency 99.9th Percentile	1732.39	[	1504.69,	1960.09]	ms
Max Net. Latency	2571.00	[	,	]	ms
Average I/O Latency	12 008.24	[	11 078.40,	12 938.08]	ns
TTC	154.50	[	149.55,	159.45]	sec.

Table 7.1: Default Partitioner - Topic B1 - Balanced cluster without replication

Throughput	15.79	[	13.01,	18.56]	MB/s
Average Net. Latency	1917.82	[	1730.55,	2105.09]	ms
Latency 50th Percentile	1617.50	[	1048.11,	2186.89]	ms
Latency 95th Percentile	2667.90	[	2094.32,	3241.48]	ms
Latency 99th Percentile	3418.95	[	2745.13,	4092.77]	ms
Latency 99.9th Percentile	4356.25	[	3666.40,	5046.10]	ms
Max Net. Latency	6624.00	[	,	]	ms
Average I/O Latency	23 077.06	[	21 610.39,	24 543.73]	ns
TTC	638.20	[	576.43,	699.97]	sec.

Table 7.2: Default Partitioner - Topic B2 - Balanced cluster with replication where producers wait for replica synchronization

Throughput	37.61	[	35.88,	39.34]	MB/s
Average Net. Latency	641.77	[	560.39,	723.15]	ms
Latency 50th Percentile	425.65	[	220.63,	630.67]	ms
Latency 95th Percentile	1294.15	[	1195.98,	1392.32]	ms
Latency 99th Percentile	1872.45	[	1593.22,	2151.68]	ms
Latency 99.9th Percentile	2431.15	[	2087.50,	2774.80]	ms
Max Net. Latency	3736.00	[	,	]	ms
Average I/O Latency	17 516.85	[	15 830.56,	19 203.15]	ns
TTC	256.60	[	243.32,	269.88]	sec.

Table 7.3: Default Partitioner - Topic B2 - Balanced cluster with replication where producers do not wait for replica synchronization

Throughput	63.17	[	60.79,	65.56]	MB/s
Average Net. Latency	116.70	[	88.33,	145.08]	ms
Latency 50th Percentile	3.44	[	0.96,	5.93]	ms
Latency 95th Percentile	685.83	[	578.35,	793.32]	ms
Latency 99th Percentile	1163.50	[	1024.00,	1303.00]	ms
Latency 99.9th Percentile	1531.56	[	1307.44,	1755.67]	ms
Max Net. Latency	2725.00	[	,	]	ms
Average I/O Latency	11 683.05	[	10 071.40,	13 294.71]	ns
TTC	152.72	[	146.64,	158.81]	sec.

Table 7.4: Default Partitioner - Topic B2 - Unbalanced cluster without replication

Throughput	13.61	[	13.08,	14.13]	MB/s
Average Net. Latency	2117.20	[	2039.17,	2195.24]	ms
Latency 50th Percentile	2992.94	[	2724.68,	3261.21]	ms
Latency 95th Percentile	4070.72	[	3958.65,	4182.79]	ms
Latency 99th Percentile	4833.50	[	4708.17,	4958.83]	ms
Latency 99.9th Percentile	5667.39	[	5273.73,	6061.05]	ms
Max Net. Latency	7780.00	[	,	]	ms
Average I/O Latency	41 934.19	[	40 519.12,	43 349.26]	ns
TTC	704.94	[	679.63,	730.26]	sec.

Table 7.5: Default Partitioner - Topic U2 - Unbalanced cluster with replication where producers wait for replica synchronization

Throughput	44.82	[	40.90,	48.74]	MB/s
Average Net. Latency	402.23	[	270.76,	533.70]	ms
Latency 50th Percentile	255.42	[	82.16,	428.68]	ms
Latency 95th Percentile	1079.32	[	987.08,	1171.55]	ms
Latency 99th Percentile	1633.32	[	1461.28,	1805.35]	ms
Latency 99.9th Percentile	3605.47	[	2488.80,	4722.15]	ms
Max Net. Latency	8243.00	[	,	]	ms
Average I/O Latency	16 402.79	[	13 213.75,	19 591.84]	ns
TTC	219.74	[	197.15,	242.32]	sec.

Table 7.6: Default Partitioner - Topic U2 - Unbalanced cluster with replication where producers do not wait for replica synchronization

7.6.2 Improved Partitioner

This partitioner does not change the algorithm logic at the base of the default policy, but it uses localized low-level synchronization and less complex data structures, compared to concurrency-ready ones with always-preset synchronization. The less sophisticated code is actually an advantage, since, after an initial warm up, its easier for the JVM (Java Virtual Machine) to improve it at run-time if needed [7].

The results for this partitioner are not shown since they follow the values of the default policy, with just stricter confidence intervals, this was caused by limited complexity of the simulated clusters, which did not stress enough the intrersted sections in the improved code.

7.6.3 Node Partitioner

The focus on nodes rather than partitions favors this new policy while dealing with the unbalanced tests, turning synchronization on or off accentuate the impact. Looking at the balanced scenarios sheds light on the overhead when working in normal/common situations (no faults or bad cluster topology configuration).

Throughput	60.69	[	59.10,	62.28]	MB/s
Average Net. Latency	131.79	[	110.26,	153.31]	ms
Latency 50th Percentile	2.10	[	1.50,	2.70]	ms
Latency 95th Percentile	765.20	[	640.87,	889.53]	ms
Latency 99th Percentile	1411.60	[	1226.70,	1596.50]	ms
Latency 99.9th Percentile	1844.95	[	1644.08,	2045.82]	ms
Max Net. Latency	2641.00	[	,	]	ms
Average I/O Latency	12 461.02	[	11 527.49,	13 394.55]	ns
TTC	158.90	[	154.52,	163.28]	sec.

Table 7.7: Node Partitioner - Topic B1 - Balanced cluster without replication

Throughput	14.17	[	14.07,	14.27]	MB/s
Average Net. Latency	2027.04	[	2012.76,	2041.32]	ms
Latency 50th Percentile	1690.00	[	1169.33,	2210.67]	ms
Latency 95th Percentile	2707.60	[	2163.74,	3251.46]	ms
Latency 99th Percentile	3630.25	[	3004.46,	4256.04]	ms
Latency 99.9th Percentile	4610.50	[	4025.32,	5195.68]	ms
Max Net. Latency	7197.00	[	,	]	ms
Average I/O Latency	24 560.92	[	23 675.61,	25 446.23]	ns
TTC	674.40	[	669.83,	678.97]	sec.

Table 7.8: Node Partitioner - Topic B2 - Balanced cluster with replication where producers wait for replica synchronization

Throughput	39.01	[	37.20,	40.83]	MB/s
Average Net. Latency	577.65	[	494.84,	660.46]	ms
Latency 50th Percentile	489.75	[	282.76,	696.74]	ms
Latency 95th Percentile	1356.25	[	1215.68,	1496.82]	ms
Latency 99th Percentile	1965.45	[	1696.11,	2234.79]	ms
Latency 99.9th Percentile	2557.40	[	2230.67,	2884.13]	ms
Max Net. Latency	4273.00	[	,	]	ms
Average I/O Latency	16 930.01	[	14 581.28,	19 278.75]	ns
TTC	247.20	[	236.96,	257.44]	sec.

Table 7.9: Node Partitioner - Topic B2 - Balanced cluster with replication where producers do not wait for replica synchronization

Throughput	59.95	[	56.04,	63.86]	MB/s
Average Net. Latency	161.30	[	97.25,	225.36]	ms
Latency 50th Percentile	2.20	[	1.84,	2.56]	ms
Latency 95th Percentile	435.20	[	220.97,	649.43]	ms
Latency 99th Percentile	1194.47	[	796.53,	1592.40]	ms
Latency 99.9th Percentile	1483.60	[	1113.20,	1854.00]	ms
Max Net. Latency	2564.00	[	,	]	ms
Average I/O Latency	11 465.13	[	9669.87,	13 260.39]	ns
TTC	162.00	[	150.51,	173.49]	sec.

Table 7.10: Node Partitioner - Topic U1 - Unbalanced cluster without replication

Throughput	15.52	[	15.45,	15.59]	MB/s
Average Net. Latency	1847.48	[	1839.22,	1855.75]	ms
Latency 50th Percentile	16.40	[	16.00,	16.80]	ms
Latency 95th Percentile	406.70	[	316.13,	497.27]	ms
Latency 99th Percentile	1159.90	[	1085.72,	1234.08]	ms
Latency 99.9th Percentile	1641.30	[	1513.96,	1768.64]	ms
Max Net. Latency	6442.00	[	,	]	ms
Average I/O Latency	29 475.75	[	27 259.62,	31 691.87]	ns
TTC	615.80	[	612.96,	618.64]	sec.

Table 7.11: Node Partitioner - Topic U2 - Unbalanced cluster with replication where producers wait for replica synchronization

Throughput	41.67	[	41.31,	42.02]	MB/s
Average Net. Latency	651.79	[	637.81,	665.77]	ms
Latency 50th Percentile	3.00	[	3.00,	3.00]	ms
Latency 95th Percentile	218.00	[	176.83,	259.17]	ms
Latency 99th Percentile	706.33	[	591.21,	821.45]	ms
Latency 99.9th Percentile	1045.11	[	891.76,	1198.46]	ms
Max Net. Latency	3400.00	[	,	]	ms
Average I/O Latency	18 045.37	[	16 456.60,	19 634.15]	ns
TTC	230.33	[	228.48,	232.19]	sec.

Table 7.12: Node Partitioner - Topic U2 - Unbalanced cluster with replication where producers do not wait for replica synchronization

7.6.4 SQF Partitioner

The available cluster used as test was not complex enough to completely justify the use of an SQF partitioner, multiple producers should have been used, with varying topic arrangements and possibly node failure at runtime (caused on purpose). As with the improved vs. default partitioner comparison, this policy does not drastically drift from its counterpart, the Node partitioner, and the tests just show the impact of a slightly different code that pursues the same goal.

Throughput	61.99	[	60.28,	63.70]	MB/s
Average Net. Latency	122.42	[	103.97,	140.87]	ms
Latency 50th Percentile	2.00	[	1.64,	2.36]	ms
Latency 95th Percentile	741.70	[	649.56,	833.84]	ms
Latency 99th Percentile	1347.25	[	1189.37,	1505.13]	ms
Latency 99.9th Percentile	1832.50	[	1625.77,	2039.23]	ms
Max Net. Latency	2996.00	[	,	]	ms
Average I/O Latency	11 491.45	[	10 478.43,	12 504.47]	ns
TTC	155.60	[	151.12,	160.08]	sec.

Table 7.13: SQF Partitioner - Topic B1 - Balanced cluster without replication

Throughput	14.12	[	14.04,	14.19]	MB/s
Average Net. Latency	2035.37	[	2023.80,	2046.95]	ms
Latency 50th Percentile	1923.15	[	1438.47,	2407.83]	ms
Latency 95th Percentile	3428.55	[	2957.90,	3899.20]	ms
Latency 99th Percentile	4300.20	[	3705.37,	4895.03]	ms
Latency 99.9th Percentile	5176.35	[	4597.25,	5755.45]	ms
Max Net. Latency	7543.00	[	,	]	ms
Average I/O Latency	24 120.92	[	23 116.33,	25 125.50]	ns
TTC	676.90	[	673.11,	680.69]	sec.

Table 7.14: SQF Partitioner - Topic B2 - Balanced cluster with replication where producers wait for replica synchronization

Throughput	38.48	[	37.18,	39.79]	MB/s
Average Net. Latency	610.00	[	529.06,	690.95]	ms
Latency 50th Percentile	488.60	[	308.71,	668.49]	ms
Latency 95th Percentile	1324.00	[	1250.66,	1397.34]	ms
Latency 99th Percentile	2090.50	[	1892.43,	2288.57]	ms
Latency 99.9th Percentile	2522.70	[	2310.40,	2735.00]	ms
Max Net. Latency	3581.00	[	,	]	ms
Average I/O Latency	17 204.29	[	14 937.87,	19 470.71]	ns
TTC	250.00	[	241.86,	258.14]	sec.

Table 7.15: SQF Partitioner - Topic B2 - Balanced cluster with replication where producers do not wait for replica synchronization

Throughput	54.65	[	51.39,	57.91]	MB/s
Average Net. Latency	291.27	[	201.19,	381.36]	ms
Latency 50th Percentile	2.00	[	2.00,	2.00]	ms
Latency 95th Percentile	228.40	[	152.94,	303.86]	ms
Latency 99th Percentile	749.05	[	566.03,	932.07]	ms
Latency 99.9th Percentile	1213.80	[	935.90,	1491.70]	ms
Max Net. Latency	3104.00	[	,	]	ms
Average I/O Latency	18 139.45	[	13 250.12,	23 028.78]	ns
TTC	177.75	[	166.96,	188.54]	sec.

Table 7.16: SQF Partitioner - Topic U1 - Unbalanced cluster without replication

Throughput	15.45	[	15.19,	15.70]	MB/s
Average Net. Latency	1857.56	[	1827.09,	1888.03]	ms
Latency 50th Percentile	16.75	[	15.99,	17.51]	ms
Latency 95th Percentile	2397.25	[	1054.02,	3740.48]	ms
Latency 99th Percentile	3068.00	[	1738.28,	4397.72]	ms
Latency 99.9th Percentile	3759.88	[	2218.49,	5301.26]	ms
Max Net. Latency	6040.00	[	,	]	ms
Average I/O Latency	29 973.12	[	27 326.99,	32 619.25]	ns
TTC	618.75	[	608.67,	628.83]	sec.

Table 7.17: SQF Partitioner - Topic U2 - Unbalanced cluster with replication where producers wait for replica synchronization

Throughput	42.06	[	41.25,	42.88]	MB/s
Average Net. Latency	630.82	[	584.78,	676.85]	ms
Latency 50th Percentile	3.00	[	3.00,	3.00]	ms
Latency 95th Percentile	258.25	[	172.73,	343.77]	ms
Latency 99th Percentile	684.85	[	534.92,	834.78]	ms
Latency 99.9th Percentile	1312.85	[	735.59,	1890.11]	ms
Max Net. Latency	5715.00	[	,	]	ms
Average I/O Latency	17 948.38	[	16 760.71,	19 136.04]	ns
TTC	228.25	[	224.22,	232.28]	sec.

Table 7.18: SQF Partitioner - Topic U2 - Unbalanced cluster with replication where producers do not wait for replica synchronization

7.7 Conclusions

Consumers do not have any advanced logic and simply request data when they needed it, sequentially based on the record index. If the records are evenly distributed between partitions and nodes the resulting performance of both producers and cluster will improve, the nodes in the cluster will be equally stressed (and equally far from the overloading point) and consumers will experience less latency.

Until now, the producer performance were out of the equation while looking at any Kafka system, some focus was put into latency, and, from there, the default partitioner was implemented with almost no advanced logic whatsoever. Adding more load balancing logic into the partitioner is an efficient way to improve the overall cluster performance.

The most advanced new partitoner, the SQF (p.73), may not be seen as really applicable since it require more computational power, but nowadays even the smallest embedded devices can handle simple linear searches if the data collection is small enough, and this is exactly the case of Apache Kafka. The theory allows for indefinitely populated clusters, with the highest imaginable topics and partitions density. In the real world, not only it will be hard to find a scenario where the numbers were high enough to matter, but having also an impactful amount of nodes, topics-per-nodes and nodes-per-topic all at the same time is unpractical and unreal.

I would be easy to think that the tested scenarios may have been too simple too properly evaluate each partitioner (actually true only for the improved partitioner), instead they are perfectly placed in a sweetspot where:

- **B1** and **B2** with sync. does not care about the selected partitioner, the below average hardware is enough to handle a simple good configured cluster regardless of the chosen partitioning policy.

- **B2** without synchronization is the only balanced scenarios where the node-oriented partitioners slightly improve the average latency, thanks to simpler code paths and less hash-maps look-ups, which hurts the unstoppable continuous data streams. The absence of a partitioner-to-cluster synchronization did not made the producer wait a replica cluster-wide write acknowledgment, thus stressing the code more. **B1** did not have replicas, ence had less stress on the cluster, and **B2** with sync. lets the producers "relax" while the replicas aligned themselves with their leaders.

- **U1** puts the default partitioner under the spotlight with a substantial gain in throughput, but it must be remembered that a cluster with no replicas partitions and unbalanced node-topic/partition ratio is one of the worst possible configuration, no fault-tolerance and prone to higher latency spikes (has shown bye the 99th and 99.9th latency percentiles). A cluster configuration like this would be risky and unpredictable to actually use.

- **U2** without sync. falls in the same case of B2 without sync., where the absence of pauses lets the default partitioner gain an edge on average latency and throughput, but it experiences high latency spikes compared to node based partitioners, as happend with **U1**.

- **U2** with synchronization is the main point of this document, representing a fault-tolerant cluster that just experienced one, the simulation of a real world case. Both node-based partitioners perform almost the same (SQF have slightly higher latency spikes) and beat the default partitioner in throughput and average latency. The difference in speed is also noticeable by the big difference in I/O latency experienced by the cluster.

All proposed new partitioners in this thesis gradually improve the performance metrics in the simulated real world scenarios, **U2** and **B2** with synchronization, and fails in degenerate cases that should never be deployed in productions environments (no fault tolerance whatsoever).

Choosing a node-based partitioner is a place-and-forgot choice in all those cluster that can withhold the negligible performance impact and needs more robust fault tolerance. Using multiple-level sub-clusters can also help the adaptation of more advanced partitioners, but the key feature, that any Apache Kakfa cluster needs, is good node/topic/partition configuration which is done in the initial setup of the cluster itself. The cluster admins are not left alone and can be helped with possible-fault detection algorithms during the topology planning process.

The proposed partitioners bring an impactful but basic approach on improving Apache Kafka performance by having powerful producers. As explored in chapter 6.3 (p.75) advanced techniques can be used further evolve the current code-base, which can later be used as starting point to implement even smarter producers.

Bibliography

[1] Shadi A. Noghabi et al. "Samza: stateful scalable stream processing at LinkedIn". In: *Proceedings of the VLDB Endowment* 10 (Aug. 2017), pp. 1634–1645. DOI: 10.14778/3137765.3137770.

[2] Polak. A. *What to consider for painless Apache Kafka integration*. Blog Post. Jan. 2019. URL: https://www.freecodecamp.org/news/what-to-consider-for-painless-apache-kafka-integration-df559e828876/.

[3] J Baer. *How Apache Drives Spotify's Music Recommendations*. Presentation. Sept. 2015. URL: https://apachebigdata2015.sched.com/event/400d/keynote-how-apache-drives-spotifys-music-recommendations-josh-baer-spotify.

[4] N. Bansal and M. Harchol-Balter. "Analysis of SRPT Scheduling: Investigating Unfairness". In: *SIGMETRICS Perform. Eval. Rev.* 29.1 (June 2001), pp. 279–290. ISSN: 0163-5999. DOI: 10.1145/384268.378792. URL: http://doi.acm.org/10.1145/384268.378792.

[5] H. Blanks. *More data, more data*. Blog Post. July 2016. URL: https://blog.cloudflare.com/more-data-more-data/.

[6] G. Bouloukakis et al. "Queueing Network Modeling Patterns for Reliable and Unreliable Publish/Subscribe Protocols". In: Nov. 2018, pp. 176–186. DOI: 10.1145/3286978.3287002.

[7] J. Adam Butts and Guri Sohi. "Dynamic Dead-instruction Detection and Elimination". In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS X. San Jose, California: ACM, 2002, pp. 199–210. ISBN: 1-58113-574-2. DOI: 10.1145/605397.605419. URL: http://doi.acm.org/10.1145/605397.605419.

[8] A. Chaffai, L. Hassouni, and H. Anoun. "Informal Learning in Twitter: Architecture of Data Analysis Workflow and Extraction of Top Group of Connected Hashtags: Third International Conference, BDCA 2018, Kenitra, Morocco, April 4–5, 2018, Revised Selected Papers". In: vol. 872. Aug. 2018, pp. 3–15. ISBN: 978-3-319-96291-7. DOI: 10.1007/978-3-319-96292-4_1.

[9] J Coolidge. "The story of the binomial theorem". In: 56 (June 2019), pp. 147–157. DOI: 10.2307/2305028.

[10] D. Dedousis, N. Zacheilas, and V. Kalogeraki. "On the Fly Load Balancing to Address Hot Topics in Topic-Based Pub/Sub Systems". In: July 2018. DOI: 10.1109/ICDCS.2018.00018.

[11] P. Dobbelaere and K. Sheykh Esmaili. "Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper". In: June 2017, pp. 227–238. DOI: 10.1145/3093742.3093908.

[12] H. Isah and F. Zulkernine. "A Scalable and Robust Framework for Data Stream Ingestion". In: (Dec. 2018).

[13] S. Islam. "Network Load Balancing Methods: Experimental Comparisons and Improvement". In: *CoRR* abs/1710.06957 (2017).

[14] S Jaloudi. "Communication Protocols of an Industrial Internet of Things Environment: A Comparative Study". In: *Future Internet* 11 (Mar. 2019). DOI: 10.3390/fi11030066.

[15] A. Javed et al. "CEFIoT: A fault-tolerant IoT architecture for edge and cloud". In: Mar. 2018. DOI: 10.1109/WF-IoT.2018.8355149.

[16] K. Jay, N. Neha, and R. Jun. "Kafka: a Distributed Messaging System for Log Processing". In: *ACM* 19 (2011), p. 6. DOI: 10.3390/s19010134. URL: http://notes.stephenholiday.com/Kafka.pdf.

[17] K. Kato et al. "A Study of a Scalable Distributed Stream Processing Infrastructure Using Ray and Apache Kafka". In: Dec. 2018, pp. 5351–5353. DOI: 10.1109/BigData.2018.8622415.

[18] L. Kleinrock. *Theory, Volume 1, Queueing Systems*. New York, NY, USA: Wiley-Interscience, 1975. ISBN: 0471491101.

[19] J. Kreps. *Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines)*. Blog Post. Apr. 2014. URL: https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines.

[20] B. Leang et al. "Improvement of Kafka Streaming Using Partition and Multi-Threading in Big Data Environment". In: *Sensors* 19 (Jan. 2019), p. 134. DOI: 10.3390/s19010134.

[21] A. Miklos et al. "Fairness in scheduling". In: *Journal of Algorithms* 29.2 (Nov. 1998), pp. 306–357.

[22] C. Molter and P. Vijay. *Kafka as a Service: A Tale of Security and Multi-Tenancy*. Presentation. May 2018. URL: https://kafka-summit.org/sessions/kafka-service-offering-tale-security-multi-tenancy/.

[23] D. Mukherjee et al. "Universality of Power-of-d Load Balancing Schemes". In: *SIGMETRICS Perform. Eval. Rev.* 44.2 (Sept. 2016), pp. 36–38. ISSN: 0163-5999. DOI: 10.1145/3003977.3003990. URL: http://doi.acm.org/10.1145/3003977.3003990.

[24] A. Noon, A. Kalakech, and S. Kadry. "A New Round Robin Based Scheduling Algorithm for Operating Systems: Dynamic Quantum Using the Mean Average". In: *International Journal of Computer Science Issues* 8 (Nov. 2011).

[25] C. Park and A. Shankar. *Apache: Big Data*. Presentation. Sept. 2015. URL: https://apachebigdata2015.sched.com/event/3ztw/netflix-integrating-spark-at-petabyte-scale-cheolsoo-park-netflix-and-ashwin-shankar-netflix.

[26] M. Rostanski, K. Grochla, and A. Seman. "Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ". In: Sept. 2014. DOI: 10.15439/978-83-60810-58-3.

[27] P. Salazar. *OpenSOC: An Open Commitment to Security*. Blog Post. Nov. 2014. URL: https://blogs.cisco.com/security/opensoc-an-open-commitment-to-security.

[28] P. Sathish and N. Pramod. "Efficient data transfer through zero copy". In: (Sept. 2008). URL: https://developer.ibm.com/articles/j-zerocopy/.

[29] L. SCHRAGE. "A proof of the optimality of the shortest remaining processing time discipline". In: *Operations Research* 16 (1968), pp. 687–690. DOI: 10.1287/opre.16.3.687. URL: https://ci.nii.ac.jp/naid/30041575850/en/.

[30] Basit Shahzad and Muhammad Afzal. "OPTIMIZED SOLUTION TO SHORTEST JOB FIRST BY ELIMINATING THE STARVATION". In: Nov. 2005.

[31] G. Shapira, N. Narkhede, and T. Palino. *Kafka: The Definitive Guide*. O'Reilly Media, Sept. 2017.

[32] D. Stancevic. *Zero Copy I: User-Mode Perspective*. Blog Post. Jan. 2003. URL: https://www.linuxjournal.com/article/6345?page=0,0.

[33] B. Svingen. *Publishing with Apache Kafka at The New York Times*. Blog Post. Sept. 2017. URL: https://www.confluent.io/blog/publishing-apache-kafka-new-york-times/.

[34] J. Sztrik. "Basic queueing theory". In: *University of Debrecen, Faculty of Informatics* 193 (2012).

[35] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014. ISBN: 013359162X, 9780133591620.

[36] Ervin Varga. "Apache Kafka as a Messaging Hub". In: Nov. 2016. DOI: 10.1007/978-1-4842-2196-9_12.

[37] D. Vohra. "Using Apache Kafka". In: Jan. 2016, pp. 185–194. ISBN: 978-1-4842-1829-7. DOI: 10.1007/978-1-4842-1830-3_12.

[38] A. Wang. *Multi-Tenant, Multi-Cluster and Hierarchical Kafka Messaging Service*. Presentation. Aug. 2017. URL: https://apachebigdata2015.sched.com/event/3ztw/netflix-integrating-spark-at-petabyte-scale-cheolsoo-park-netflix-and-ashwin-shankar-netflix.

[39] G. Wang et al. "Building a replicated logging system with Apache Kafka". In: *Proceedings of the VLDB Endowment* 8 (Aug. 2015), pp. 1654–1655. DOI: 10.14778/2824032.2824063.

[40] R. Wiatr, R. Slota, and J. Kitowski. "Optimising Kafka for stream processing in latency sensitive systems". In: July 2018.

[41] D Yuan. *Stream Processing in Uber*. Presentation. Dec. 2015. URL: https://www.infoq.com/presentations/uber-stream-processing.

[42] S. Zhao et al. “Real-time network anomaly detection system using machine learning”. In: *2015 11th International Conference on the Design of Reliable Communication Networks, DRCN 2015* (July 2015), pp. 267–270. DOI: 10.1109/DRCN.2015.7149025.