National Technical University of Athens

School of Electrical and Computer Engineering
Division of Computer Science

Event-Driven Architectures using Apache Kafka

Diploma Thesis

of

Alexandros Aris Liarokapis

Supervisor: Vasileios Veskoukis
Professor NTUA

Software Engineering Lab

Athens - July 2022

Page intentionally left blank.

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science
Software Engineering Lab

Event-Driven Architectures using Apache Kafka

Diploma Thesis
of
Alexandros Aris Liarokapis

Supervisor: Vasileios Veskoukis
Professor NTUA

Approved by the examination committee on July 8, 2022.

(Signature)	(Signature)	(Signature)
.....	.....	.....
Vasileios Veskoukis	Georgios Goumas	Panayiotis Tsanakas
Professor NTUA	Assoc. Professor NTUA	Professor NTUA

Athens - July 2022

Page intentionally left blank.

(Signature)

.....

Alexandros Aris Liarokapis

Graduate of Electrical and Computer Engineering NTUA

Page intentionally left blank.

Abstract

Event-Driven architectures are a very useful approach to decoupling inter-service communication while enabling unique communication patterns. The specific resiliency, availability, and performance requirements of such architectures, warrants the usage of specialized Message-Broker services. Moreover, new real-time requirements of many event processing tasks, such as metrics and analytics, requires a new approach compared to legacy batch processing. Apache Kafka is an open-source event streaming platform that can accommodate both use cases. This thesis performs a succinct presentation of Apache Kafka's main architectural model, while focusing on its concrete resiliency and availability guarantees and providing guidelines for achieving desired operational characteristics. We also document and further enhance a mathematical model that permits the estimation and concrete sizing of cluster infrastructure, as well as express the link between topic sizing, maximum unavailability and end-to-end latency.

Keywords

events, event processing, event sourcing, event streaming, microservices, inter-service communication, asynchronous communication, asynchronous messaging, message brokers, message queues, event driven architecture, apache kafka, distributed log, topic, partition, replication, leader election, consumers, producers, consumer groups, high resiliency, high availability, throughput, latency, message ordering, design guarantees, infrastructure modeling

Page intentionally left blank.

Περίληψη

Οι αρχιτεκτονικές οδηγούμενες απο events αποτελούν μία πολύ χρήσιμη προσέγγιση ως προς την αποσύνδεση της επικοινωνίας μεταξύ υπηρεσιών, επιτρέποντας παράλληλα μοναδικά μοτίβα επικοινωνίας. Οι συγκεκριμένες απαιτήσεις για ανθεκτικότητα, διαθεσιμότητα και απόδοση αυτών των αρχιτεκτονικών, δικαιολογεί τη χρήση εξειδικευμένων υπηρεσιών μεταβίβασης events. Επιπλέον, νέες, πραγματικού χρόνου απαιτήσεις διαδικασιών επεξεργασίας events, όπως παραγωγή και επεξεργασία μετρήσεων και αναλυτικών στοιχείων, απαιτούν μια νέα προσέγγιση σε σύγκριση με την παραδοσιακού τύπου batch επεξεργασίας events. Το Apache Kafka είναι μία ανοιχτού κώδικα πλατφόρμα ροής events που μπορεί να επιτρέψει και τις δύο περιπτώσεις χρήσης. Αυτή η διατριβή εκτελεί μια συνοπτική παρουσίαση του κύριου αρχιτεκτονικού μοντέλου του Apache Kafka, ενώ εστιάζει στις εγγυήσεις που παρέχει ως προς την ανθεκτικότητά και διαθεσιμότητα του συστήματος, καθώς και παρέχει οδηγίες για την επίτευξη συγκεκριμένων επιθυμητών λειτουργικών χαρακτηριστικών. Επίσης, περιγράφει και ενισχύει περαιτέρω ένα μαθηματικό μοντέλο που επιτρέπει την εκτίμηση και τον καθορισμό μέγεθους της υποδομής ενός Apache Kafka σμήνους, αλλα και εκφράζει την σχέση μεταξύ του μεγέθους των Topics, το μέγιστο παράθηρο μη διαθεσιμότητας και την απο άκρο σε άκρο καθυστέρηση μεταφοράς.

Λέξεις Κλειδιά

events, event processing, event sourcing, event streaming, microservices, inter-service communication, asynchronous communication, asynchronous messaging, message brokers, message queues, event driven architecture, apache kafka, distributed log, topic, partition, replication, leader election, consumers, producers, consumer groups, high resiliency, high availability, throughput, latency, message ordering, design guarantees, infrastructure modeling

Page intentionally left blank.

Contents

1 Εκτενής Περίληψη

Τα σύγχρονα συστήματα λογισμικού ορίζονται από αυξημένη πολυπλοκότητα. Μεγάλοι όγκοι δεδομένων καταναλώνονται, μετασχηματίζονται, και μεταφέρονται σε warehouses και data lakes για περαιτέρω ανάλυση και επεξεργασία. Ανακάλυψη υπηρεσιών, υπηρεσίες διαχείρισης μυστικών, load-balancers, κρυφές μνήμες, γενική διαχείριση υποδομών, όλα απαιτούν μη τετριμμένες ρυθμίσεις και συντήρηση. Event brokers και ουρές, προγραμματισμός εργασιών, επεξεργασία ροής, επεξεργασία batch, όλα τα παραπάνω έχουν τη θέση τους σε μία σύγχρονη στοίβα ανάπτυξης λογισμικού.

Ως αποτέλεσμα, μικρές ομάδες δυσκολεύονται να διαχειριστούν αυτό το σημαντικό αριθμό διαφορετικών τεχνολογιών. Αυτό έχει οδηγήσει στη διάσπαση της λειτουργικότητας σε διαφορετικές υπηρεσίες, διαχειριζόμενες από διαφορετικές υποομάδες. Προκειμένου αυτές οι ομάδες να λειτουργούν αποδοτικά, κρίνεται απαραίτητο να μειωθεί η επικοινωνία και συγχρονισμός μεταξύ των διαφορετικών ομάδων. Αυτό συνεπάγεται στον περιορισμό των εξαρτήσεων μεταξύ των διαφορετικών υπηρεσιών, επιτρέποντας την ανεξάρτητη ανάπτυξη και διαχείριση τους. Η παραπάνω αρχιτεκτονική αρχή αποκτά συνέχεια έδαφος, με τη μεγαλύτερη έκφανση της να είναι καλά καθιερωμένη στη βιβλιογραφία ως αρχιτεκτονική Μικρο-υπηρεσιών.

Οι μικρο-υπηρεσίες συχνά επικοινωνούν εκτενώς με χρήση events. Οι αρχιτεκτονικές βασισμένες στη χρήση events, έχουν μία χαρακτηριστική προσέγγιση ως προς την ενδοϋπηρεσιακή επικοινωνία, επιτυγχάνοντας αυξημένη αποσύνδεση μεταξύ των υπηρεσιών και επιτρέποντας μοναδικά σχήματα επικοινωνίας που δεν είναι εφικτά με εναλλακτικές αρχιτεκτονικές. Τέτοιες αρχιτεκτονικές έχουν συγκεκριμένες μη-τετριμμένες απαιτήσεις ως προς την επικοινωνία μεταξύ των διαφορετικών δομικών στοιχείων, και πιο συγκεκριμένα ως προς τη διάταξη των μηνυμάτων, την ασφάλειά τους, τη συνοχή και τη διαθεσιμότητά τους.

Ως ακραία έκφανση αυτής της αρχιτεκτονικής έχουμε τις αρχιτεκτονικές βασισμένες στην προμήθεια events (Event-Sourcing). Αυτές βασίζονται σε ανθεκτικές ροές events που χρησιμοποιούνται προκειμένου να τροποποιείται και να εξάγεται η κατάσταση του συστήματος.

Προκειμένου να πληρούνται αυτές οι απαιτήσεις, τέτοια συστήματα συχνά χρησιμοποιούν εξειδικευμένες υπηρεσίες λογισμικού που ονομάζονται Message Brokers, οι οποίοι και λειτουργούν ως τον σκελετό αντίστοιχων αρχιτεκτονικών επιτρέποντας την ασφαλή και αποδοτική επικοινωνία με χρήση μηνυμάτων.

Ένα ακόμα σημαντικό θέμα είναι το γεγονός πως οι απαιτήσεις για αναλυτικά στοιχεία διαρκώς αυξάνονται. Αυτό έχει οδηγήσει στην υιοθεσία πλαισίων μαζικής επεξεργασίας (batch-processing frameworks), τα οποία επιτρέπουν την ασύγχρονη μεταφορά και επεξεργασία μεγάλων όγκων δεδομένων για μετέπεισα επεξεργασία και ανάλυση. Δυστυχώς, σύγχρονα συστήματα πολλές φορές απαιτούν επεξεργασία δεδομένων σε πραγματικό χρόνο που παραδοσιακά συστήματα μαζικής επεξεργασίας δεν μπορούν να παρέχουν αποδοτικά.

Ως αποτέλεσμα, μία πλατφόρμα που επιτρέπει την αποδοτική μεταφορά και επεξεργασία μεγάλου όγκου δεδομένων, ενώ επιτρέπει την εξυπηρέτηση των αναγκών τόσο για μαζική όσο και πραγματικού χρόνου επεξεργασία, είναι πολύ χρήσιμη για τις ανάγκες σύγχρονων αρχιτεκτονικών.

Επιπλέον, προκειμένου να αποτελέσουν τη βάση για αποδοτική ενδοϋπηρεσιακή επικοινωνία, τέτοιες πλατφόρμες πρέπει να είναι ευέλικτες και παραμετροποιήσιμες, υποστηρίζοντας τόσο υψηλή διαθεσιμότητα και ανθεκτικότητα όσο και χαμηλή καθυστέρηση μετάδοσης ή και υψηλής διαμεταγωγικής ικανότητας.

Μία τέτοια υπηρεσία είναι το Apache Kafka, ένα υψηλά επεκτάσιμο, διαθέσιμο και αποδοτικό κατανεμημένο σύστημα που λειτουργεί ως μία πλήρης πλατφόρμα ροής events. Το Apache Kafka δεν επιτρέπει μόνο την επικοινωνία με χρήση events μεταξύ των διαφορετικών υπηρεσιών αλλά ταυτόχρονα και την επεξεργασία μεγάλου όγκου δεδομένων σε πραγματικό χρόνο. Αυτή η πληθώρα δυνατοτήτων καθιστούν απότομη την καμπύλη εκμάθησης του συστήματος, καθώς και απαραίτητη την προσεκτική μελέτη διαφορετικών πηγών προκειμένου να μπορεί να ληφθούν εμπεριστατωμένες αποφάσεις ως προς την παραμετροποίηση, τον σχεδιασμό των Topics και την υποδομή καθώς μικρές αλλαγές μπορούν να αλλάξουν ραγδαία τη συμπεριφορά του συστήματος.

Παρόλο που υπάρχουν διαθέσιμες αναλυτικές μαθησιακές πηγές, ο στόχος της διατριβής είναι να αποτελέσει έναν πρακτικό μαθησιακό πόρο για την αποδοτική κατανόηση του Apache Kafka και την εφαρμογή του ως τη βασική υποδομή για κατανεμημένες αρχιτεκτονικές συμπεριλαμβανόμενης και αυτήν των μικρο-υπηρεσιών.

Βασικές έννοιες του Apache Kafka είναι τα Partitions τα οποία αποτελούν μία ακολουθία μηνυμάτων. Οι Producers είναι προγράμματα clients που προσθέτουν μηνύματα στο τέλος της ακολουθίας, ενώ Consumers είναι clients προγράμματα που διαβάζουν μηνύματα από την ακολουθία

Τα Partitions αποτελούνται από πολλά αντίγραφα (Replicas) για λόγους ασφάλειας και διαθεσιμότητας των μηνυμάτων. Τα αντίγραφα αυτά χρησιμοποιούν κατανεμημένους αλγορίθμους ομοφωνίας προκειμένου να επιτύχουν την ασφαλή αντιγραφή με ανοχή σε συνθήκες σφάλματος.

Μία ομάδα από Partitions που περιέχουν μηνύματα ίδιου τύπου αποτελούν ένα Topic. Ένας Producer μπορεί να εκμεταλλευτεί τα διαφορετικά Partitions προκειμένου να αυξήσει τον ρυθμό παραγωγής μηνυμάτων. Επίσης, καθώς ο ρυθμός επεξεργασίας των μηνυμάτων από την πλευρά των Consumers είναι περιορισμένος, διαφορετικοί Consumers μπορούν να αναλάβουν ξεχωριστά Partitions και έτσι με την εισαγωγή περισσότερων Consumers μπορούμε να αυξήσουμε και τον ρυθμό επεξεργασίας των μηνυμάτων. Η αυτόματη ανάθεση Partitions ενός Topic στους διαφορετικούς Consumers γίνεται με τον μηχανισμό των Consumer Groups.

Μερικά συγκεκριμένα θέματα που αφορούν τη διατριβή είναι τα εξής:

- Η διαδικασία αντιγραφής των Partition και η σημασία της ως προς την ασφάλεια, διαθεσιμότητα και καθυστέρηση μεταφοράς των μηνυμάτων. Θέλουμε να παρέχουμε συγκεκριμένες κατευθυντήριες γραμμές σχετικά με τις παραμέτρους `replication factor` και `min.insync.replicas` ενός Topic ανάλογα με τις απαιτούμενες εγγυήσεις ως προς την ασφάλεια μηνυμάτων σε περίπτωση σφάλματος καθώς και διαθεσιμότητάς. Αυτό μεταξύ άλλων μεταφράζεται στον ελάχιστο απαραίτητο αριθμό Servers εντός του Cluster.
- Η σημασία της παραμέτρου `acks` των Producers και οι συνέπειες του ως προς την ασφάλεια και ρυθμό διαμεταγωγής. Παρέχονται οδηγίες σχετικά με την επιλογή της παραμέτρους ανάλογα με τις απαιτήσεις ως προς την συνοχή των μηνυμάτων και ρυθμού διαμεταγωγής.
- Η επιλογή κλειδιού και η επίδραση του στη διάταξη μηνυμάτων.
- Η χρησιμότητα των Partitions και Consumers Groups για την κλιμάκωση του ρυθμού παραγωγής και επεξεργασίας των μηνυμάτων.
- Η έννοια του "effectively-once processing" και τη μορφή που υποστηρίζει το Apache Kafka με τη χρήση των transactions.

Επιπλέον, είναι απαραίτητη για λόγους κόστους και απόδοσης η λήψη εμπεριστατωμένων αποφάσεων σχετικά με την υποδομή του Cluster και τις λειτουργικές παραμέτρους των Topic:

- Η επιλογή του αριθμού των Servers ανάλογα με την απαιτούμενη ασφάλεια και διαμεταγωγική ικανότητα.
- Η εκτίμηση των απαιτούμενων υπολογιστικών πόρων και πόρων δικτύου για την επίτευξή του επιθυμητού ρυθμού διαμεταγωγής, αριθμού Servers και ρυθμίσεων αντιγραφής καθώς και συνολικό αριθμό Consumer Groups.
- Η εξαγωγή ελάχιστων και μέγιστων ορίων Partitions των Topic με βάση το επιθυμητό ρυθμό διαμεταγωγής, μέγιστο ανεκτό παράθυρο μη διαθεσιμότητας και από τέλος σε τέλος καθυστέρηση μεταφοράς.
- Η επιλογή συγκεκριμένου αριθμού Partitions σε ένα Topic προκειμένου να εξασφαλιστεί ομοιόμορφη κατανομή φορτίου κατά την διαδικασία κλιμάκωσης ενός Cluster.

Προκειμένου να παρουσιαστεί το λειτουργικό μοντέλο του Apache Kafka, αρχικά γίνεται μία εκτενής βιβλιογραφική ανασκόπηση των αρχιτεκτονικών βασισμένων σε events, δικαιολογώντας έτσι τη χρησιμότητα αντίστοιχων συστημάτων. Στην Ενότητα 3, γίνεται μία γενική ανάλυση σχετικά με τις αρχιτεκτονικές βασισμένων σε events, συζητώντας τα πλεονεκτήματα και μειονεκτήματα των διαφόρων μορφών ενδοϋπηρεσιακής επικοινωνίας. Εισάγεται η έννοια των Message Brokers και η χρήση τους ως θεμελιώδη συστήματα ενδοεπικοινωνίας. Τέλος, παρουσιάζεται η μορφή επικοινωνίας με χρήση events και τα μοναδικά μοτίβα επικοινωνίας που επιτρέπουν.

Στο υπόλοιπο κομμάτι της διατριβής, εφαρμόζεται μία πρακτική προσέγγιση στην περιγραφή του Apache Kafka και των διάφορων εννοιών του. Στο πρώτο κομμάτι της Ενότητας 4, χρησιμοποιείται ένα Kafka Cluster στημένο με χρήση Docker προκειμένου να παρουσιαστούν τα διάφορα παραδείγματα. Ξεκινάμε συζητώντας τα Partitions, τη θεμελιώδη έννοια του Apache Kafka και κοιτάμε σε βάθος πως αυτά αποθηκεύονται στα διάφορα Servers. Επιδεικνύεται βασική παραγωγή και κατανάλωσή μηνυμάτων από και σε απλά Partitions καθώς παράλληλα χτίζεται διαίσθηση ως προς τη διατεταγμένη φύση των event logs.

Επιπλέον, αναλύεται ο τρόπος που λειτουργεί η αντιγραφή των Partitions και πως μικρές αλλαγές σε ρυθμιστικές παραμέτρους τόσο στους Servers όσο και στους Producers μπορεί να έχουν μεγάλη επίπτωση στις συγκεκριμένες εγγυήσεις που δίνει το Apache Kafka ως προς την ασφάλεια και τη διαθεσιμότητα των μηνυμάτων.

Στην ενότητα των Topics, εδραιώνεται το μοτίβο διαχωρισμού μίας ροής μηνυμάτων σε διαφορετικά Partition προκειμένου να κλιμακωθεί ο συνολικός ρυθμός διαμεταγωγής. Στο Partitioning And Message Ordering, αναλύεται η επιλογή κλειδιών και την επιρροή τους στη διάταξη των μηνυμάτων. Στη συνέχεια εξετάζεται πως η χρήση Consumer Groups μπορεί να χρησιμοποιηθεί για την αυτόματη ανάθεση Partitions σε ένα σύνολο Consumers και πως μπορούν να χρησιμοποιηθούν για να κλιμακώσουν αποδοτικά τον συνολικό αριθμό επεξεργασίας ενός Topic.

Στο κεφάλαιο Processing Guarantees, εξετάζεται το θέμα διπλών και χαμένων μηνυμάτων. Εισάγεται η έννοια της ταυτοδυναμίας και πως το Kafka μπορεί να εξασφαλίσει τη μεταφορά μηνυμάτων χωρίς αυτά να χαθούν και χωρίς να υπάρχουν πολλαπλά μηνύματα σε διατάξεις επεξεργασίας/παραγωγής μηνυμάτων που έχουν τη μορφή κατευθυνόμενων άκυκλων γράφων.

Στην ενότητα 5, παρουσιάζεται ένα μαθηματικό μοντέλο του Apache Kafka και μέσω αυτού εξάγονται προσεγγίσεις σχετικά με τους πόρους της υποδομής καθώς και κατευθυντήριες γραμμές ως προς τον σχεδιασμό Topics ανάλογα με τις απαιτήσεις ως προς το μέγιστο ανεκτό παράθυρο διαθεσιμότητας και την άκρη προς άκρη καθυστέρηση μετάδοσης. Επίσης, παρουσιάζονται συμπληρωματικές κατευθυντήριες γραμμές σχετικά με τη συγκεκριμένη επιλογή Partitions προκειμένου να επιτευχθεί ομοιογένεια κατά τη διάρκεια κλιμάκωσης του Cluster.

Στην ενότητα 6 γίνεται μία περιπτωσιακή μελέτη της πλατφόρμας DIEM, η οποία είναι ένα οικοσύστημα που παρέχει πρόσβαση σε δεδομένα σχετικά με τον ενεργειακό τομέα της Ελλάδας και της γενικότερης περιοχής, και δίνεται βαρύτητα στον μετασχηματισμό του υποσυστήματος μετάδοσης πραγματικού χρόνου.

Τέλος, αξίζει να παρουσιαστούν συνοπτικά τα συγκεκριμένα αποτελέσματα και κατευθυντήριες γραμμές της διατριβής.

Ασφάλεια Μηνυμάτων

Προκειμένου να εξασφαλιστεί ασφάλεια μηνυμάτων, συνοχή και διαθεσιμότητα υπό την ταυτόχρονη βλάβη N Servers, πρέπει να τεθούν οι συγκεκριμένοι παράμετροι: - Οι Producers πρέπει να ρυθμιστούν με `acks=all`. - Το Topic πρέπει να ρυθμιστεί με $min.insync.replicas = N + 1$. Αυτό συνεπάγεται $replicationfactor \geq N + 1$ και επομένως θέτει ένα κάτω όριο στον απαιτούμενο αριθμό Servers. Περισσότεροι Servers θα επιτρέψουν να λειτουργήσει ο μηχανισμός των In-Sync Replicas και να μπορέσουν να μειώσουν την καθυστέρηση μετάδοσης και να αυξήσουν τη διαθεσιμότητα και ποιότητα υπηρεσίας υπό προσωρινές προβληματικές περιστάσεις. Αν επίσης απαιτείται διαθεσιμότητα παραγωγής υπό την ταυτόχρονη βλάβη N Servers: - Το Topic πρέπει να ρυθμιστεί με $replicationfactor \geq 2N + 1$. Τα παραπάνω ισχύουν και για εσωτερικής χρήσης Topics όπως τα `__consumer_offsets` και `__transaction_state`

Διάταξη Μηνυμάτων

Ιδιαίτερη προσοχή πρέπει να δοθεί στην επιλογή κλειδιού κατά την παραγωγή μηνυμάτων καθώς αυτό διαμορφώνει τις εγγυήσεις περί διάταξης των μηνυμάτων. Όλα τα μηνύματα που έχουν ίδιο κλειδί θα καταλήξουν στο ίδιο Partition και επομένως θα διαβαστούν και με τη σωστή σχετική σειρά από τους Consumers.

Οι Producers πρέπει να εξασφαλίσουν επίσης, πως τα μηνύματα θα φτάσουν στους Servers με την σωστή σειρά, επομένως: - Οι Producers πρέπει να ρυθμιστούν είτε με `enable.idempotence=true` ή `max.in.flight.requests.per.connection=1`. Το πρώτο είναι προτιμότερο. Επιπλέον, - Τα Topics πρέπει να φτιάχνονται με περισσότερα Partitions από ότι θεωρείται πως είναι απαραίτητα αρχικά έτσι ώστε να μη χρειαστεί να αυξήσουμε τον αριθμό τους στη συνέχεια. Η αλλαγή του αριθμό των Partitions ενός Topic χαλάει την αντιστοιχία κλειδιού-Partition καταστρέφοντας έτσι κάθε εγγύηση ως προς τη διάταξη των μηνυμάτων. Σχετικά με τη σύμπτυξη μηνυμάτων, - Τα κλειδιά, χρησιμοποιούνται κατά τη σύμπτυξη μηνυμάτων ως αναγνωριστικά ταυτότητας και αυτό πρέπει να λαμβάνεται υπόψη όταν διαλέγουμε το κλειδί ενός Topic. Μόνο τα τελευταία μηνύματα με το ίδιο κλειδί αποθηκεύονται κατά τη σύμπτυξη.

Εγγυήσεις ως προς τη διανομή Μηνυμάτων και Effectively-once delivery

- Προκειμένου να εξασφαλιστεί πως τα μηνύματα θα αποσταλούν τουλάχιστον μία φορά, όλες οι ρυθμίσεις περί ασφάλειας που αναφέρθηκαν παραπάνω πρέπει να ισχύουν. Επιπλέον, οι Consumers πρέπει να κάνουν χειροκίνητα commit τα offset των μηνυμάτων μετά την επεξεργασίας τους και όχι πριν, χωρίς να βασίζονται στην προκαθορισμένη auto-commit συμπεριφορά.
- Μεγάλες διατάξεις κατανάλωσης/επεξεργασίας/παραγωγής μηνυμάτων μπορούν να φτιαχτούν εξασφαλίζοντας πως για κάθε μήνυμα-είσοδο θα αντιστοιχεί ένα μήνυμα-εξόδου σε κάθε καταληκτική θέση. Αυτό αποτελεί το λεγόμενο "effectively-once delivery" που παρέχει το Kafka. Το κομμάτι της επεξεργασίας πρέπει να μην έχει εξωτερικά παρατηρήσιμες παρενέργειες καθώς θεμελιωδώς απαιτούνται κατανεμημένα transactions. Για αυτό το λόγο,

αυτή η δυνατότητα δεν είναι τόσο χρήσιμη στην ενδοεπικοινωνία μεταξύ των υπηρεσιών. Μία σημαντική λεπτομέρεια εδώ είναι η σωστή επιλογή του `transactional.id`. Μία καλή προσέγγιση είναι το σχήμα `<group id>.<topic>.<partition>`.

Υποδομή και Παραμετροποίηση

- Η προσθήκη Servers μειώνει τις απαιτήσεις δικτύου και απόδοσης τω αποθηκευτικών μέσων. Στα (4, 5) αναλύονται οι συγκεκριμένες απαιτήσεις δικτύου και αποθήκευσης. Στο (3) δίνονται εκτιμήσεις για τη μέγιστη εφικτή διαμεταγωγική ικανότητα των Topics.
- Αυξάνοντας τα Partitions μέχρι και τον αριθμό των αποθηκευτικών μέσων, αυξάνει τον ρυθμό παραγωγής.
- Αυξάνοντας τα Partitions επιτρέπει την εισαγωγή περισσότερων Consumers εντός ενός Consumer Group, επιτρέποντας την οριζόντια κλιμάκωση του ρυθμού επεξεργασίας ενός Topic. Στο (9) δίνονται εκτιμήσεις για τον ελάχιστο αριθμό Partitions που χρειάζονται για να επιτευχθεί ένας στόχος ρυθμού επεξεργασίας.
- Αυξάνοντας τα Partitions επηρεάζεται το μέγιστο παράθυρο μη-διαθεσιμότητας και η από άκρη σε άκρη καθυστέρηση αποστολής. Στα (10, 11) δίνονται σχέσεις για άνω όρια του αριθμού των Partitions με βάση αυτούς τους περιορισμούς.
- Διαλέγοντας ως αριθμό των Partitions ενός Topic πολλαπλάσιο του αριθμού των Server ενός Cluster, επιτυγχάνεται ομοιόμορφη κατανομή φορτίου. Στο κεφάλαιο keeping load uniform γίνεται σχετική ανάλυση.

2 Introduction

2.1 Background

Modern software systems are defined by increased complexity. Large volumes of data must be ingested, transformed, transported into data warehouses and data lakes and further analyzed. Service discovery, secret management facilities, load-balancers, caches, and general infrastructure management, all require non-trivial setups and maintenance. Message brokers and queues, task scheduling, stream processing, batch processing, all of the above have their place in a modern software development stack.

As a result, single teams struggle to manage a considerable number of different technologies. This has led into the segregation of functionality into different services, owned and managed by different sum-teams. In order to support these teams in operating efficiently, it is paramount to minimize the amount of inter-team communication overhead. This means minimizing the dependencies between services owned by different teams, thus enabling independent development and operations. The above architectural principle keeps gaining ground; one realization of this model is well established in the literature as Microservice Architecture.

Microservices are often a case of Event-Driven architecture, which takes an identifying approach to inter-service communications, by allowing for increased decoupling between services while also facilitating unique communication schemes that are not possible with other architectural patterns. Such architectures have specific non-trivial requirements regarding communication between components, namely message ordering, safety, consistency, and availability. At the extreme end, we have Event Sourcing architectures that rely on a resilient and persistent stream of events in order to track and derive the application's state.

To meet the above requirements, such systems often delegate functionality to specialized software services, called Message Brokers that operate as the backbone of such architectures, permitting safe and efficient communication, utilizing message passing.

Another important issue is that requirements for analytics have been constantly growing. This has led into the adoption of batch processing frameworks, able to asynchronously transfer and process large amounts of data for later processing and analysis. Unfortunately, it is frequently the case that modern system, also demand real-time processing of data that legacy batch processing frameworks cannot efficiently provide.

As a result, a platform that can tackle the task of efficiently moving and processing large volumes of data, while also being able to serve both batching and real-time processing, is very useful for modern software architectures. Additionally, in order to be able to efficiently provide the base for inter-service communications, such a platform must be flexible and configurable, supporting both high availability and resiliency as well as low latency and/or high throughput.

One such software service is Apache Kafka, a highly scalable, available and efficient distributed system that acts as a full-fledged Message Streaming Platform. Apache Kafka not only facilitates the asynchronous event-driven communication between services, but it also enables the real-time processing of large volumes of events such as user activity tracking, and analytics. Such use-cases have traditionally been implemented with batch processing, but were limited by the non-real-time nature of this approach.

2.2 Thesis Goals

In this context, we have been motivated by the possibilities of architecting applications using patterns such as Microservices, that make use of messaging to implement inter-process communications, explore the potentials and identify the drawbacks or risk that come with this decision. A mature and feature-rich messaging service that has been chosen for our approach is Apache Kafka, a large and complex piece of software with many configuration options in both servers and clients, which can largely affect its performance, high availability and safety guarantees. As such, its learning curve is steep, requiring careful studying of several resources before being able to make informed engineering decisions regarding configuration, topic design and infrastructure.

While detailed technical information and resources are available, our approach is focused on developing a practical learning resource for efficiently understanding and applying Kafka as an underlying infrastructure of messaging, for distributed software architectures including but not limited to Microservices.

Some specific topics worth analyzing are the following:

- Partition replication and it's consequences on safety, availability and latency. We want to provide concrete guidelines on how to derive the `replication factor` as well as `min.insyc.replicas` configurations of a Topic, depending on the required message safety in case of Server failures as well as worst-case read and write availability requirements. This translates to the minimum number of Kafka Servers required.
- The importance of the Producers' `acks` setting and its effect on message safety and throughput. We provide guidelines on how to pick the `acks` Producer setting according to the specific message consistency and throughput requirements.
- The importance of the concrete key-scheme that will be used, and it's effect on message ordering.
- How partitioning and consumer groups can be used to effectively increase read and write topic throughput.
- The exact meaning of Apache Kafka's transactions and so called "effectively-once processing" along with its practical consequences.

Moreover, it is paramount for proper cost estimation and cluster performance to be able to take concrete decisions regarding the cluster's infrastructure and topic operational parameters:

- Decide on the number of servers according to required safety guarantees and maximum topic write throughput.
- Estimate required maximum network and persistent storage throughput depending on target throughput, number of servers replication settings, and number of consumer groups.
- Derive lower and upper bounds on Topic partition count, according to target and Consumer throughput, maximum tolerable unavailability windows and end-to-end latency.
- Decide exact number of Partitions while ensuring uniform load across Servers when scaling.

2.3 Methodology and Structure

In order to help build intuition about Apache Kafka, we first provide sufficient background in event-driven architectures in order to justify them and also more clearly understand why systems such as Kafka are useful. Section 3 performs a general background analysis on event-driven architectures, discussing advantages and disadvantages of the various inter-service communication schemes. We present Message Brokers and their usage as an inter-service communication primitive. Finally, we focus on Event-based communication and the new communication patterns that it enables.

For the rest of the thesis, we take an applied approach to discussing Kafka and its various concepts.

In the first part of Section 4, we take advantage of a Dockerized Kafka cluster in order to showcase concrete examples of each concept. We start by discussing Partitions, the fundamental abstraction of Kafka, and we take a deep look into how they are stored in Kafka Servers. We demonstrate simple consumption and production to lone Partitions while also building intuition on the ordered nature of message logs. Furthermore, we analyze how replication of Partitions works and how small configuration changes in both Servers and Producers can have a big impact on the specific safety and availability guarantees that Kafka provides.

In Topics, we solidify the pattern of splitting a message stream to multiple Partitions in order to scale a message stream's total read and write throughput. In Partitioning And Message Ordering, we also discuss the consequences on message ordering and the specific message ordering guarantees that can be achieved by using keys.

Next, we discuss how Consumer Groups may be used for automatic assignment of Partitions. We then demonstrate an example of scaling a topic's read throughput by introducing multiple Consumers to a Consumer Group.

In Processing Guarantees, we address the issues of lost and duplicate messages. We discuss idempotence and how Kafka can achieve message de-duplication. We briefly present Apache Kafka's Transactions and how these can be used to avoid lost messages and duplicate message production in large message processing pipelines compromised of pure Apache Kafka consumers without observable side effects.

Section 5, presents a mathematical model of Apache Kafka, specific estimations for required cluster resources, as well as Topic sizing guidelines depending on maximum tolerable end-to-end latency and unavailability windows. It also provides a complimentary guideline for maintaining uniform load across brokers when scaling the cluster.

Section 6 puts the previous sections into practice by performing a concrete case study of the DIEM platform, an ecosystem that provides timely access to data for the energy Market in Greece and the wider area, focusing on the restructuring of its real-time subsystem.

3 Event-Driven Architectures

3.1 Microservices

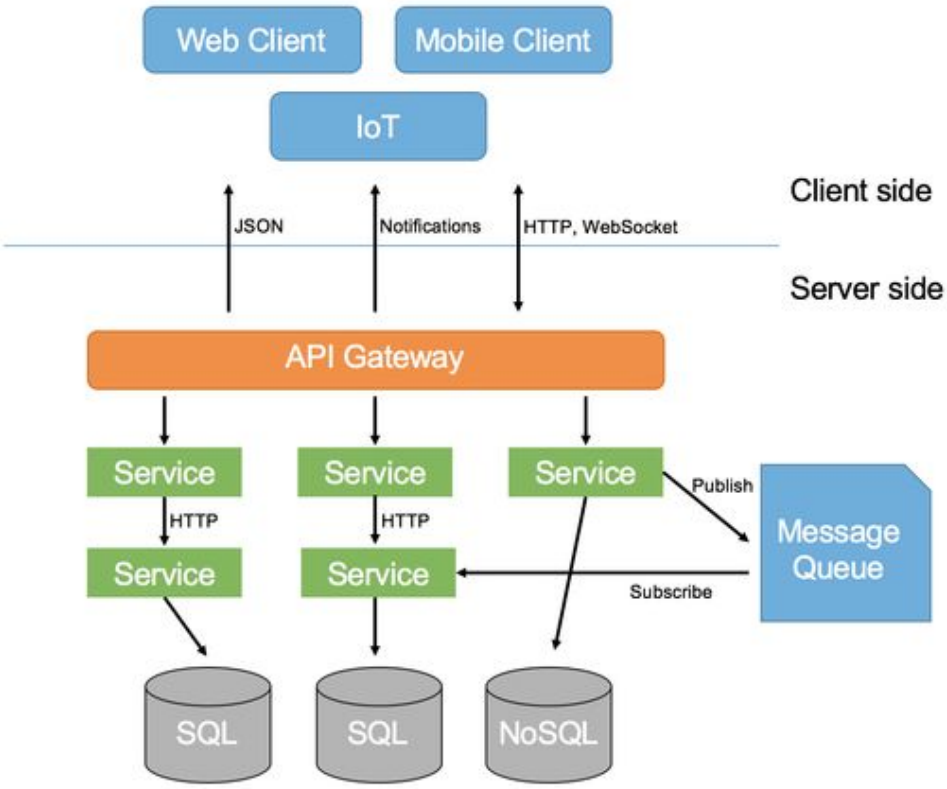


Figure 1: Example of Microservices architecture [1].

Microservices are independently releasable services that are modeled around a business domain [2, p. 3].

Such services encapsulate functionality and expose it to other services through the network using a public abstracted interface. These services are then composed together in order to construct the larger, more complex system [3] .

Microservices are treated as a black box, promoting the notion of information hiding. They maintain a public network endpoint (REST/GraphQL/Queue/RPC) along with an explicit or implicit schema. Other services communicate with them through the exposed endpoint, completely unaware of the internal implementation details. This information hiding acts as a barrier, defining which part of each Microservice is easy to change and what should stay backwards compatible [4, p. 36].

Each service is allowed to arbitrarily change any internal implementation functionality, as long as this change doesn't leak into the public interface. If it does, it does so in a backwards compatible manner. At the extreme end, this information hiding allows each team to work with its preferred and most productive technology stack. A consequence of this information hiding is the fact that Microservices tend to not share databases, opting to maintaining different databases for each service.

If shared state is required, it is often refactored to a separate service with an extra abstraction layer on top.

Microservice Architecture vs Service-Oriented Architecture

The above description of Microservice Architecture is reminiscent of the older Service-Oriented Architecture (SOA,[5]). The difference is not so much technical as cultural. Service-Oriented Architecture just like Microservice Architecture follows a generic service-based approach to monolithic decomposition. Due to historical reasons, Service-Oriented Architecture is a term associated to protocols like SOAP and the WS-* family of standards. Moreover, they are also associated to usage of heavyweight, vendor-specific, Enterprise Service Buses [3] .

Microservice architecture can be thought of as an opinionated version of Service-Oriented Architecture, favoring complete information hiding, lightweight inter-service communication protocols and maintaining independent service life cycles.

Independent Deployability

In order to achieve proper team independence, it is paramount to enable teams to have their own independent deployment schedules [2, p. 6] [4, p. 16] .

The alternative is to do system-wide microservice releases requiring complex synchronization between the different teams which is error-prone and time-consuming.

Modularity

While beneficial in itself, independent deployability also forces the services to stay decoupled. Synchronized releases can allow services to become tightly-coupled with all the disadvantages that this entails. In contrast, requiring each team to independently release their managed services acts as a modularity test, ensuring that each service can be changed without affecting other services.

Here we would like to stress that Microservice Architecture is not the only way to achieve modularity. Modularity can also be achieved in monolithic applications by properly defining module boundaries and forcing independent module releases [6] [7] .

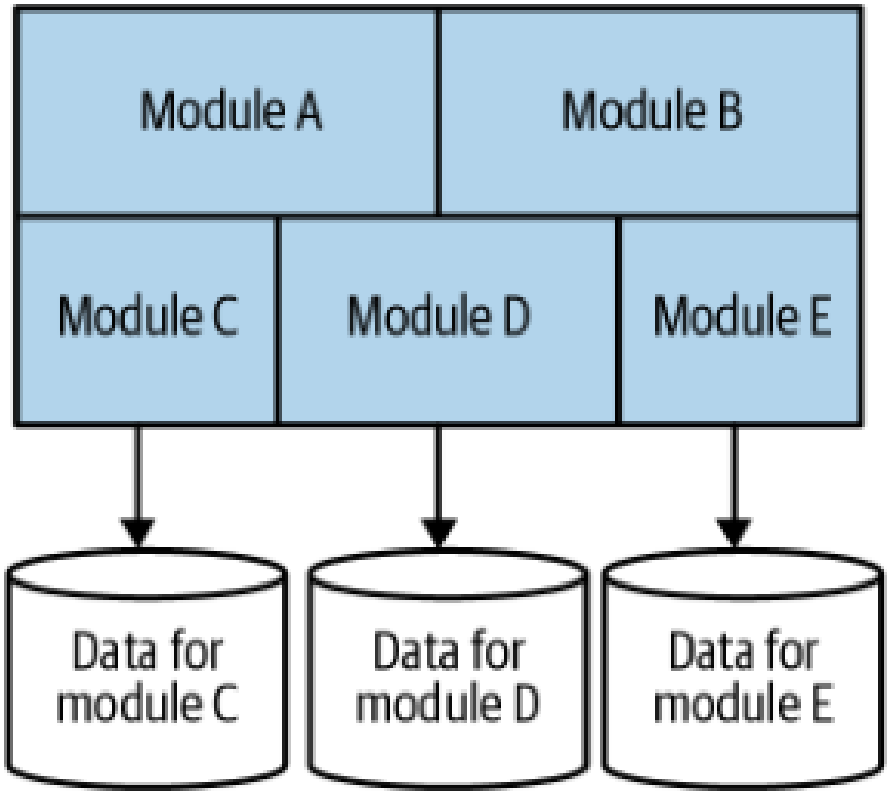


Figure 2: Modular Monolithic Architecture [2].

In contrast, we can also have service-oriented architectures where services are not actually decoupled.

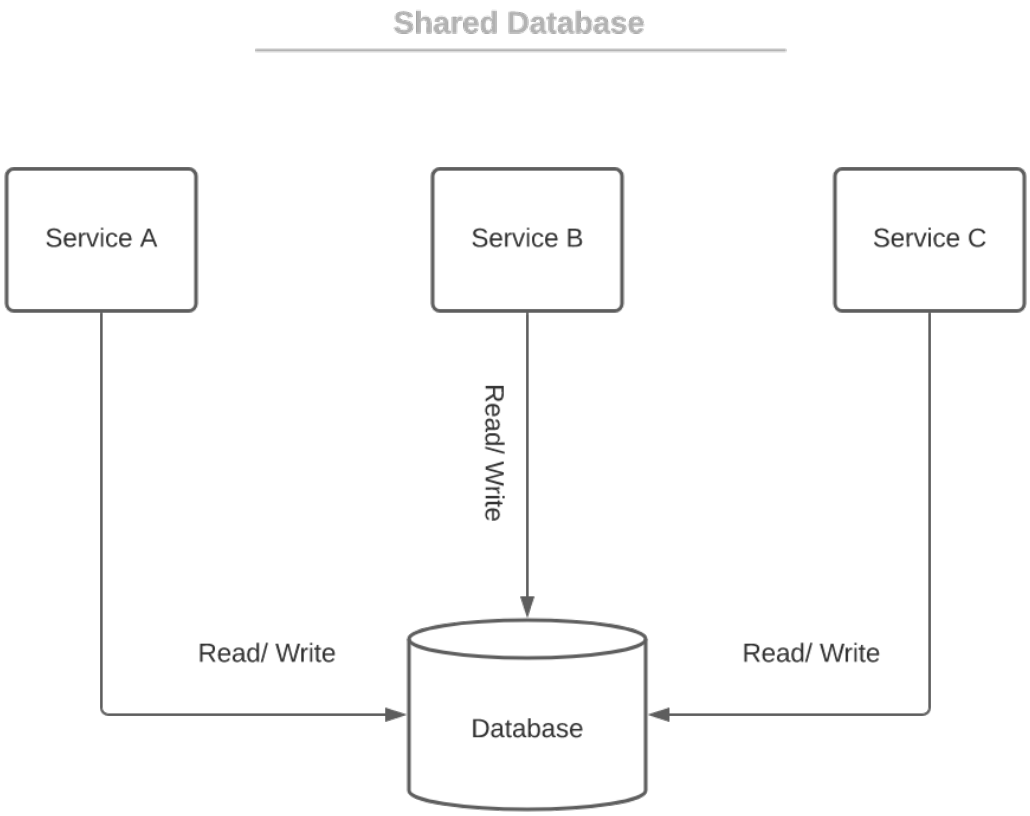


Figure 3: Services coupled to common database.

In the end, Microservice Architecture is a way to structure teams and keep their owned services decoupled in order to minimize change propagation. The overhead of maintaining clear public interfaces between services, and the networking nature of inter-service communications, introduces is non-trivial and should be taken into account.

Organized around Business Domains

> Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure. – Melvin Conway, 1968

It has traditionally been the case that team creation would be based on technology familiarity and expertise. This naturally leads to a componentization of system functionality that mimics this technology-based team specialization.

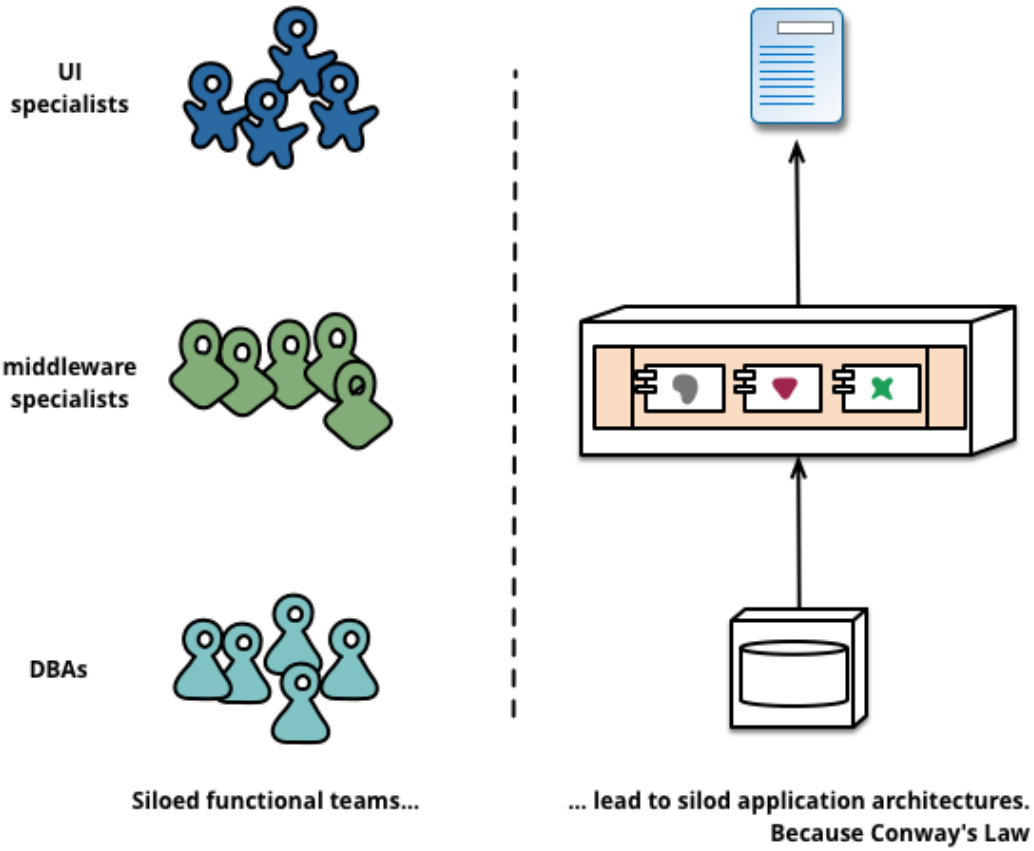


Figure 4: System architecture mirrors team structure [3].

Unfortunately, it is often the case that feature implementation spans across multiple technology layers. In a traditional team setting, this means that complex inter-team cooperation must take place, increasing communication and synchronization overhead.

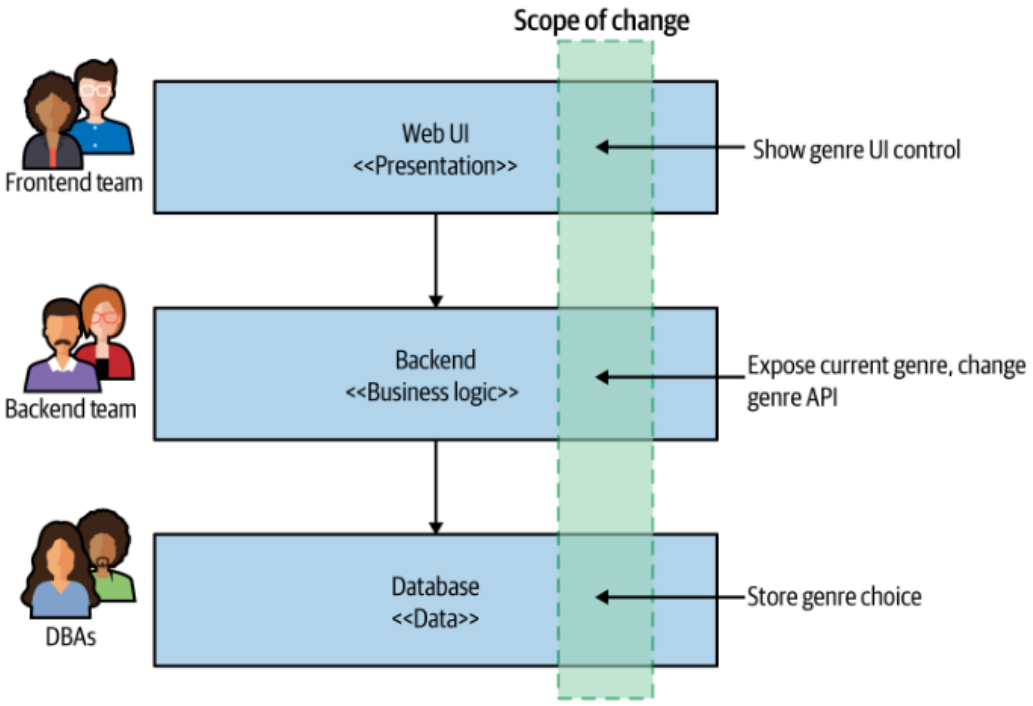


Figure 5: Features typically span multiple layers [2].

In a Microservice architecture, team independence is key, as such the focus is shifting towards cross-functional teams the structure of which mimics the business capabilities and therefore the communication patterns that are required when implementing business functionality [8] [2, p. 3] .

This ensures that each service can be independently enhanced and maintained without losing time and focus due to inter-team communication.

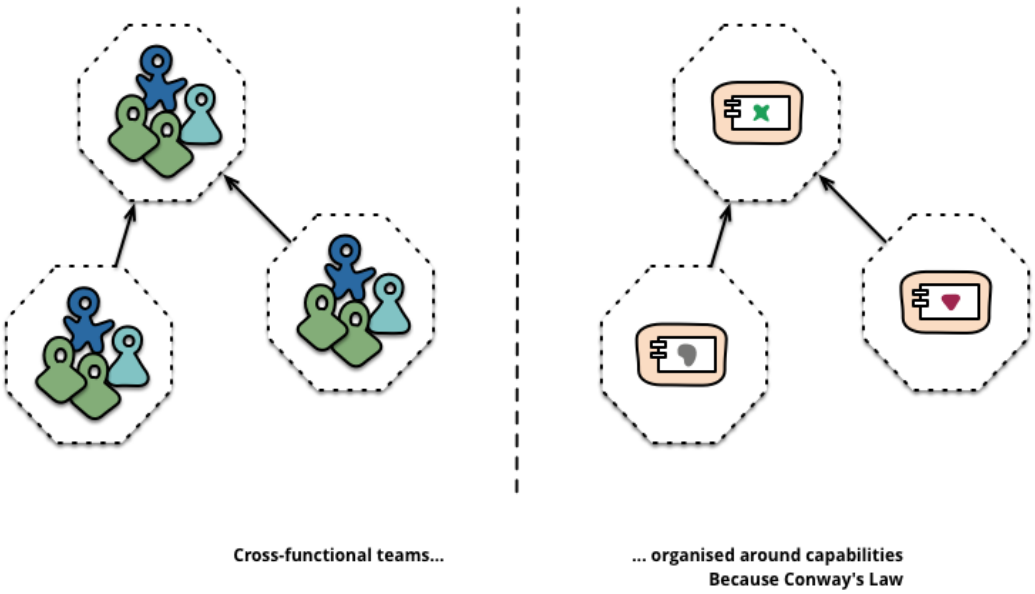


Figure 6: Cross-functional teams lead to independent services [3].

Technology Heterogeneity

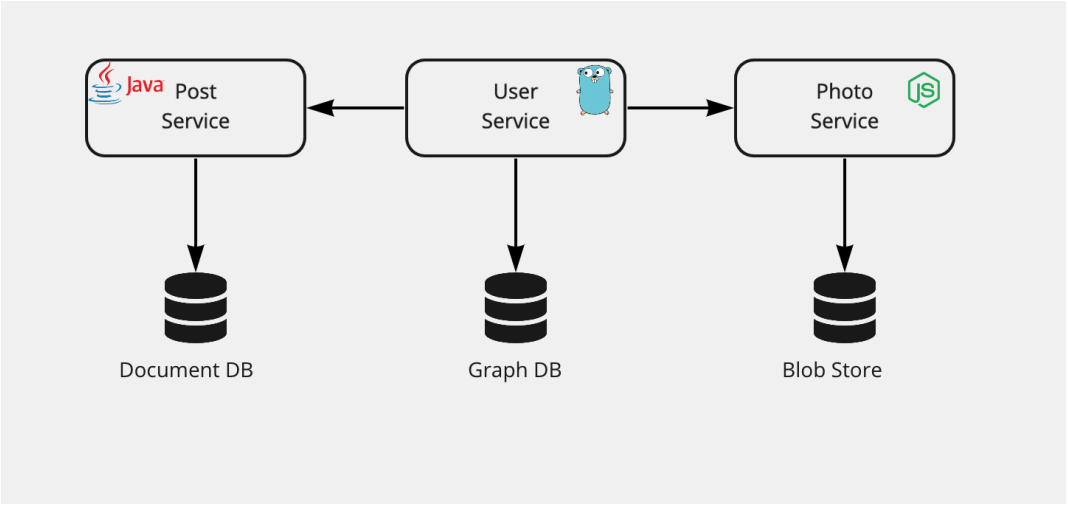


Figure 7: Each service can use its preferred technology stack [9].

Due to the information hiding nature of the Microservices Architecture, different services are free to use their preferred technology stack as an implementation detail. This flexibility allows using the best tools for a given job, without having to use a generic all-fits-one solution that ends up being the lowest common denominator.

Resilience

Microservices operate under the assumption that external service dependencies communicate via networking. As such, services must be able to handle all the different error conditions that occur due to the networking nature of inter-service communication. Services can tolerate service dependency unavailability by gracefully degrading service functionality. This, permits increased resiliency and robustness in the case of a subset of the system's services becoming unavailable.

Scalability

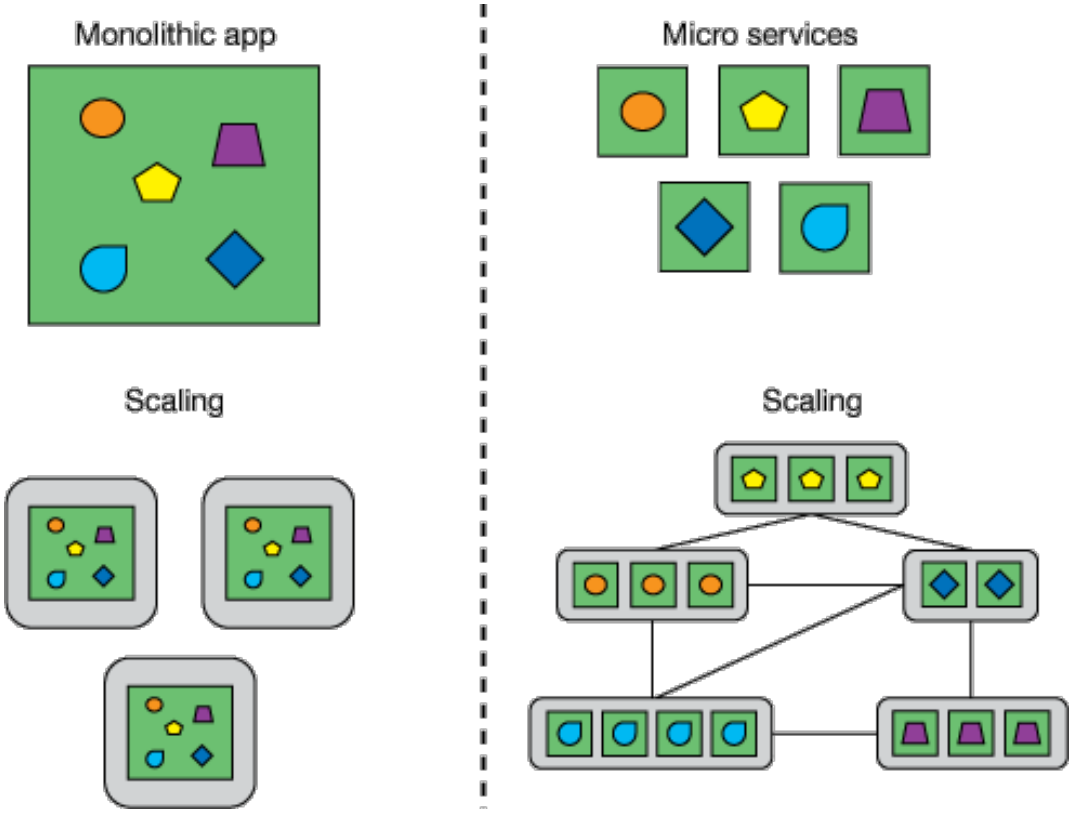


Figure 8: Microservices can be scaled independently [3].

Since each service can be independently deployed and communicate over the network, we are free to introduce multiple instances of specific services. We can then use these as backups in order to increase resiliency in the face of unavailability, or perhaps put them behind a load-balancer or some message broker in order to distribute load. More importantly, we have the freedom to scale each service independently, this is not the case with a monolithic application architecture. This is an important advantage and for this reason functional decomposition is considered one of the main axes of scaling (Y-Axis, [10]).

Organizational focus

As previously discussed, decomposing system functionality into components owned by independent teams, allows system functionality to be implemented without incurring heavy inter-team communication overhead. Since services are smaller in scope and more focused, the teams that own them tend to be smaller and more productive as well.

Disadvantages

Forcing communication through the network is a great way to create and maintain explicit communication interfaces and properly decouple services from each other.

Unfortunately, this decoupling can frequently have consequences.

Main disadvantages of the Microservices Architecture stem from the extra complexity involved, as well as issues inherent in distributed systems [11].

Technology overload

In order to enable Microservices, extra technology must be used. Service discovery mechanisms, instance orchestration, message queues, log aggregators and observability tracing frameworks must often be put into use in order to make Microservices viable. The technology flexibility that Microservices permit can also lead to complexity explosion, making transitions between teams hard and complicating the hiring process.

In-process vs Inter-process computational overhead

Microservices must run on difference processes, potentially on different hardware, and communicate to each other through non-native communication channels that can span programming languages and ecosystems.

Compared with communication between modules, inter-service communication is going to be slower and less robust.

Increased interface complexity

Calling a procedure from another module can't fail, due to the coupled nature of modules coexisting in the same process. In contrast, communication though the network can always fail, requiring countermeasures and introducing cognitive overhead that complicates the underlying application logic.

Maintaining non-native interfaces is also more complex. The interfaces and protocols involved must be communicated through an external mechanism and properly versioned. These are nonexistent problems in architectures relying on homogeneous technology stacks.

Observability and troubleshooting

No matter the testing coverage, some problems are bound to be manifested in production. In this case, being promptly notified and collecting enough data to be able to analyze and solve the problem is of paramount importance. Instrumenting code that operates across different technologies, ecosystems, runtimes, multiple instances and load-balancers is way harder compared to instrumenting more monolithic systems. Efforts like the Open Telemetry initiative are promising in this regard.

3.2 Inter-Service Communication Patterns

In this section, we attempt to provide a high-level overview of inter-service communication patterns. This overview is not meant to be exhaustive but instead serve as the basis for discussing event-based communication, queues, and message brokers later on.

Synchronous - Blocking

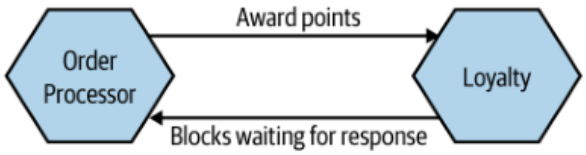


Figure 9: Service sends a message to another services and synchronously waits for the response [2].

This pattern consists of a service sending a request to another service, and blocking until it gets a request back before resuming execution. It is the default and most familiar style of inter-module communication in monolithic applications. Procedure and library calls constitute synchronous request-response communication. Relevant inter-service technologies include REST Apis and RPC frameworks [2, p. 95].

```
interface Dependency {
    public Response call(Request request);
};

...

void process(Dependency dependency) {

    // request-processing-response
    Response res = dependency.call(new Request(...));

    continuation(res);
}
```

Advantages of this communication pattern is the straightforward control flow and familiar mental model.

The main disadvantage of this pattern is the temporal coupling that it creates. In this case, request, processing, and response are treated as a single operation. **This means that the requesting context execution time is bounded from below according to the sum of the request, processing and response times**.

In order to further discuss this issue and see how this is a problem mostly relevant to distributed architectures, we will examine it from two different contexts. One is communication between modules and the other is communication between services across a network.

A good example of inter-module communication is native procedure calls.

We can make the following observations:

- Request time is negligible and bounded.
- Processing time is often measurable and bounded.
- Response time is negligible and bounded.

Furthermore, in monolithic scenarios, there are usually no strict execution upper-bound requirement. Of course, there are numerous counter-examples to this:

- Functions executed in GUI framework threads must have fast execution time in order to not affect responsiveness.
- Audio processing applications have hard upper-bound requirements in order to not create audible processing artifacts.
- Aviation System Control often have hard-real time requirements.

The above explain why synchronous request-response is such a good default communication style for inter-module communication. There are no real availability issues in this context, in order for a module to not be available the whole process must be down, making the requesting module unavailable too.

The situation is more complex when it comes to inter-service communication through the network. This is the typical Microservice communication style. Here the behavior is complicated by the network medium and the dependency-service's lifetime.

We can make the following observations:

- Request time depends on network and dependency lifetime and is therefore unbounded.
- Processing time is again dependent on dependency's lifetime. It is unbounded, and even if the service is highly available, it is still unwise to depend on its performance characteristics since it can end up being owned by a separate team.
- Response time again depends on the network and is therefore unbounded.

We can often improve our infrastructure in order to reduce the network effects and availability issues, but we don't really want to do the same with the dependency-service's lifetime.

Since it can be managed by an independent team, we would prefer to allow the team to manage that service's deployment, without having to worry about potential smart lived unavailability windows transitively affecting dependent services.

Of course, each service should strive to be highly-available but in any case we would prefer to decouple our service from the dependency-service's availability guarantees.

Using synchronous-blocking communication, we can't make easily achieve this goal. If a dependency becomes unavailable at any point, our blocking call won't be able to proceed making our service unavailable until the dependency becomes available again.

This can be very problematic in long chains of blocking inter-service calls.

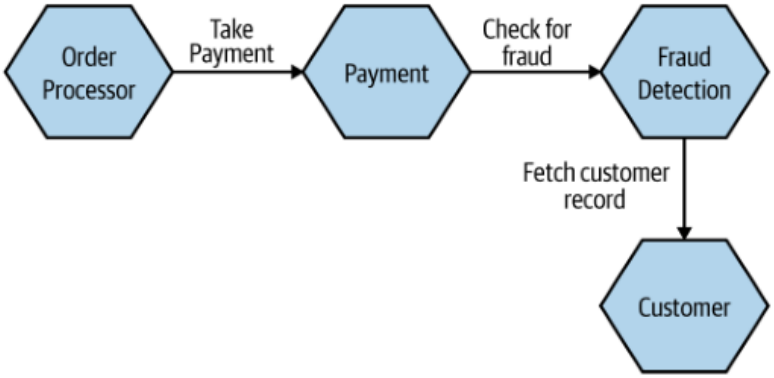


Figure 10: Example of long inter-service communication chain [2].

The transitive nature of blocking request-response means that if any service becomes unavailable, all services that are part of the chain become unavailable too.

The inherent issue of synchronous request-response is that request, processing, and response, are bundled into a single action and implicitly create a hard dependency on all subsequent lines of code.

```
...
void continuation(Response res) {
    handleResponse(res);
    otherOperations();
}

void process(Dependency dependency) {
    Response res = dependency.call(new Request(...));
    continuation(res);
}
```

Ideally, we would like to be able to remove this hard dependency and enable our service to execute all logic that is not dependent on the response. We will explore this decoupling in the subsequent section.

Asynchronous - Nonblocking

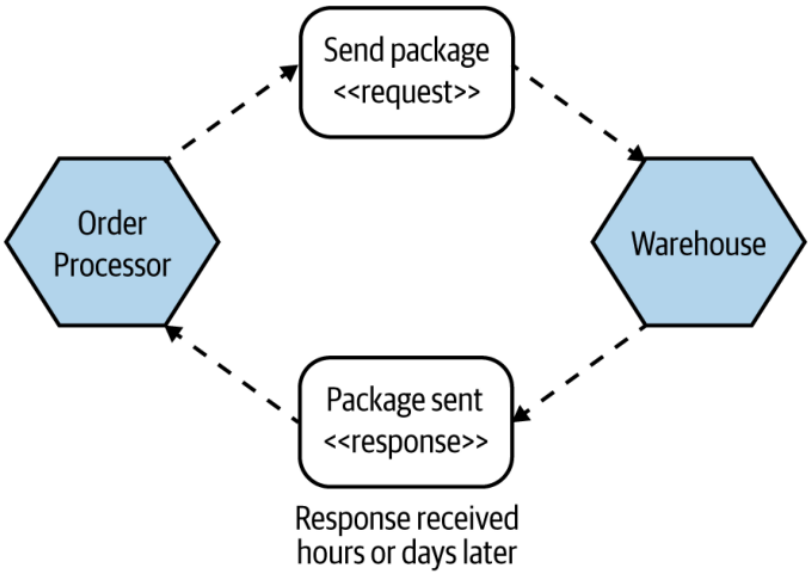


Figure 11: Service can resume processing until response asynchronously arrives. The response can arrive hours or days later without affecting the requesting service [2].

This communication pattern asynchronously handles the response of a request to the external service [2, p. 98]. This allows the issuing service to resume execution without having to wait for the response, reducing temporal coupling as a result .

Some types of asynchronous communication are:

- Communication through common data. In this case, services communicate through shared state either through the filesystem, or some shared data store.
- Asynchronous request-response. This is a variation of the typical blocking request-response pattern where requests are issued asynchronously. Communication occurs with a single instance of the service-dependency.
- Event-Driven. This is a variation of the asynchronous request-response pattern where the response is not required. Requests that don't require a response are called events. Since we don't require a single response per event, we can send our events to multiple interested instances and even to different services.

Communication through common data

This pattern consists of services sharing some sort of state. This state can consist of files within a shared filesystem, a shared database, or perhaps a distributed key-value store. Instead of directly depending on each other, these services implicitly communicate by observing the changes that occur to their shared state [12, p. 68].

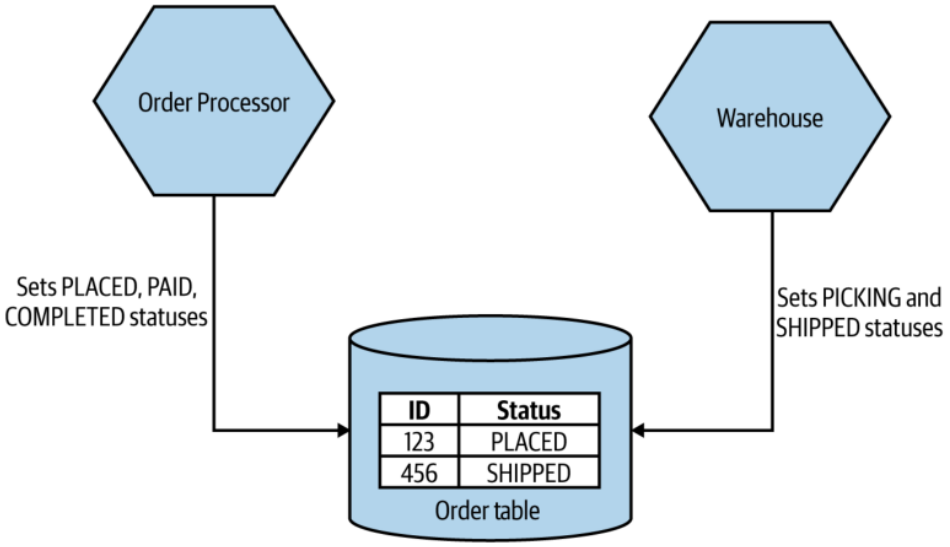


Figure 12: Two services communicating through a shared database [2].

The advantages of this method lie in that it can enable older technologies to be integrated into a Microservice architecture without having to introduce new technologies. It can also be especially useful for inherent large-volume data sharing situations. In this case, simply uploading a multi-gigabyte file to a shared filesystem, may be the best course of action [2, p. 101].

The main disadvantage, as discussed previously, is that shared state can lead to potential coupling between the two services. Any change in structure in the data store can lead to breaking the communication between the services.

Asynchronous Request-Response

This is an alternative to the blocking request-response pattern. Instead of executing and waiting for the remote call to finish, we can queue our request and asynchronously handle the response at some other point.

The way this is achieved is by explicitly changing the invocation interface so that we supply a response-dependent continuation when issuing the request:

```
interface Callback {
    public void execute(Response response);
}

interface Dependency {
    public void call(Request request, Callback callback);
};

...

void processOrder(Dependency dependency) {

    // request-processing-response
    dependency.call(new Request(...), new Callback {
        void execute(Response response)
        {
            handleResponse(response);
        }
    });

    otherOperations();
}
```

As a result of the above change, the service can resume doing `otherOperations` without having to wait for the response.

We can also queue the actual request and handle its transfer in the background while properly doing retries. This does not eliminate the coupling to the actual request time, but only mitigates it. If the dependency is not available for enough time, queuing requests can end up consuming a lot of memory and even lead to memory exhaustion. Practical implementations should make the queue's size bounded, this means that after some point the request would have to block. This means that the calling site's execution does not depend on processing and response time but is potentially coupled to the request time. In any case, the issuing service does not exhibit the same number of temporal coupling issues that it's blocking counterpart does.

Duality of Blocking and Asynchronous Interfaces

It is worth mentioning that any blocking interface can be trivially converted to an asynchronous interface and vice versa. In the case of the asynchronous interface, we can use shared state between the response continuation and the issuing context and

wait use it in order to block, waiting for the call to finish. In the case of a blocking interface, we can use threading in order to schedule the remote call for completion in the background. This of course does have performance consequences in the form of increased state bookkeeping, scheduling overhead and resource contention when parallel remote calls are active. A native asynchronous technology will naturally perform better in this regard. In any case, taking advantage of asynchronous operations requires careful software restructuring in order to benefit from removing the implicit dependency that the blocking interface imposes.

Another interesting problem of this approach is what happens when the issuing microservice and the dependency-service's request/response throughput differ. In this case, the issuing microservice can potentially overload the dependency-service. If the asynchronous interface is implemented by using threading to wrap blocking calls in the background, this means that the internal queue used to schedule the blocking calls will keep increasing until it reaches capacity, leading to a degradation of service quality.

A potential solution to this problem is the introduction of Queue middleware between the two services:

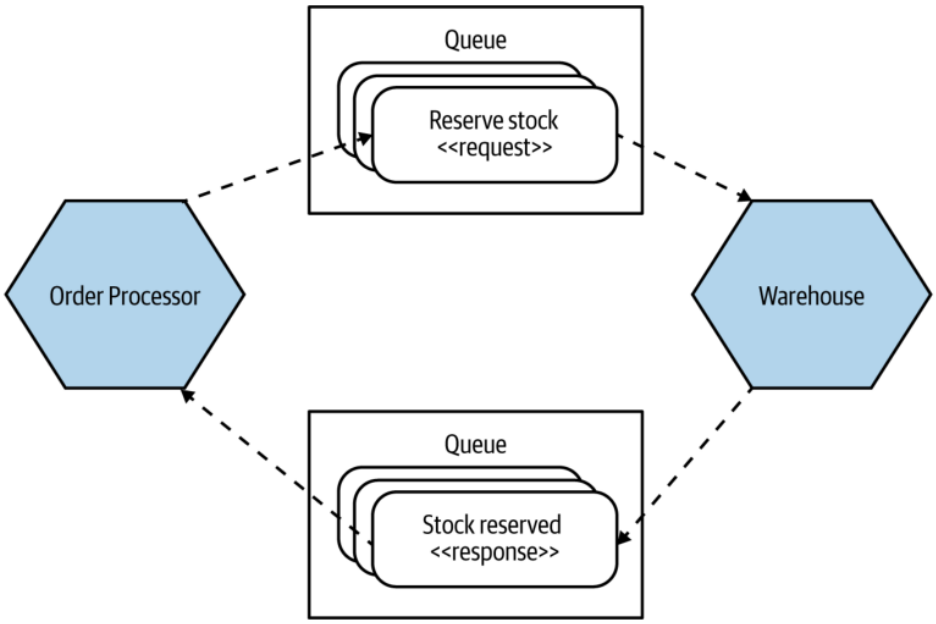


Figure 13: External queues can be used as buffers [2].

Using queues implies a native asynchronous interface. The issuing service does not communicate with its dependency directly, instead it pushes the request to an external queue. Later on, when the dependency service becomes available, it can read requests from the queue and then forward the response to another queue which will subsequently be read by the issuing service [4, p. 88] [2, p. 101]. [13]

The external queue can take advantage of disk storage in order to provide a better buffering service than the issuing service can achieve internally. Monitoring can be used to notify teams when lag accumulates noticeably. Other advantages of using queues, are increased message persistency in

case any service goes down, as well as a limited form of service discovery. Regarding the latter, the various services involved do not have to be aware of the actual services involved, they only have to be aware of the queue service. Finally, the queue service can concentrate on being highly available in contrast with the multiple other services. This alleviates the coupling issue of the issuing service to the request time.

A problem with this request-response approach is the fact that the request-response relationship is not natively maintained. One has to introduce specific metadata in order to link requests and responses and use a separate persistent store in order to maintain these links without losing the persistency advantages that queues provide.

In practice, this can be achieved by including correlation identifiers and ta separate database. Due to the extra complexity involved when having to maintain the request-response relationship, there is non-trivial cost in adopting this approach.

Event-Driven Communication

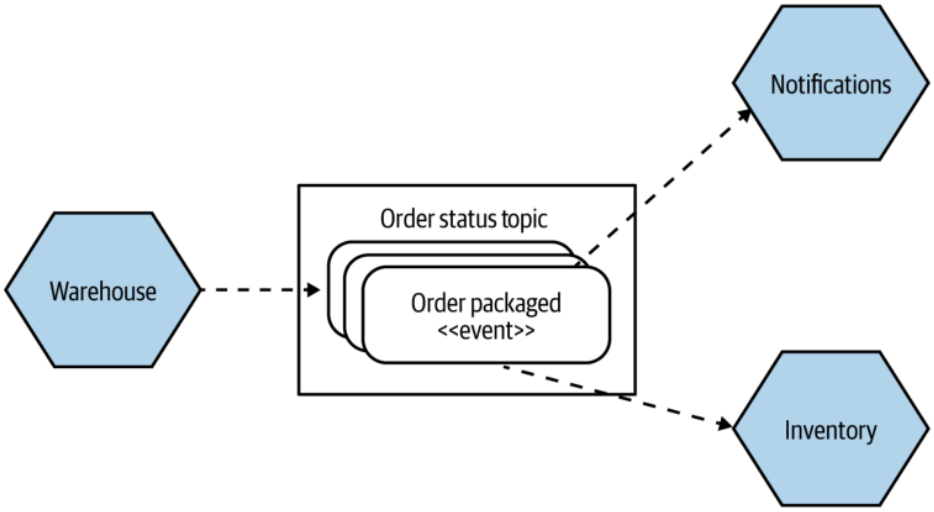


Figure 14: Example of event-based communication with multiple event subscribers [2].

This communication pattern involves sending messages to other services without expecting a response. The fact that the service does not require responses has a very important consequence. When requiring a response, there is an implicit association, where the receiving instance is also the responding instance. This means that since we expect a single response, we implicitly impose a one-to-one relationship between requests and receiving instances. If we don't require a response, then our messages can potentially reach multiple instances or even different services. Instead of reaching out and sending a message to a single service, we can broadcast a message to a broad topic abstraction where multiple listeners can subscribe to [14] [2, p. 108]. .

The interface as well as the code ends up being very simple without requiring complex control flow changes.

```
interface OrderPackagedTopic {
    public void publish(OrderPackaged event);
}

void processOrder(OrderPackagedTopic dependency)
{
    //...
    dependency.publish(new OrderPackaged{...});
}
```

Publish-Subscribe

This very powerful pattern consists of multiple listeners that subscribe to a type of published event [15] .

It is enabled when using event-driven communication, since the event can be sent to multiple listeners.

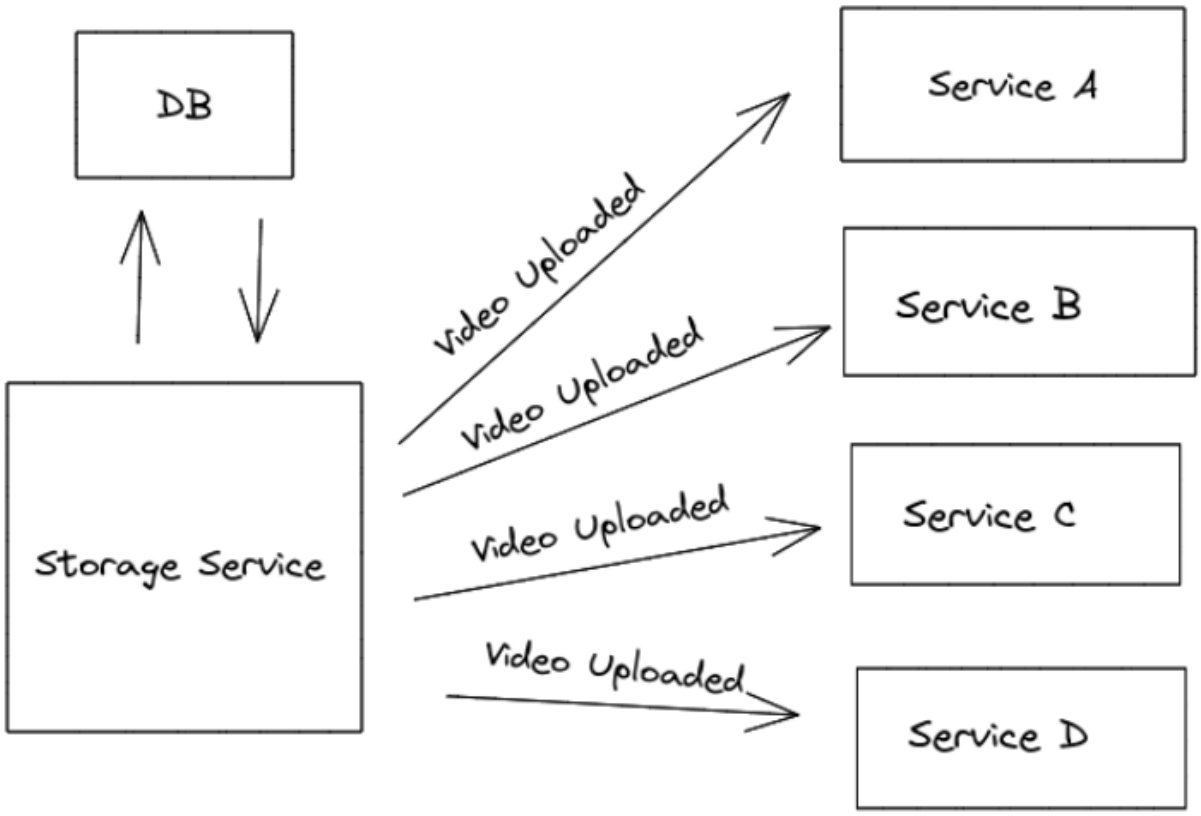


Figure 15: Storage service acts as a message broker.

In the above example, the storage service must maintain broker-like functionality integral to the service. External services subscribe to the events produced by the service by reaching out to the service and using its broker-specific protocol. This couples them to the service.

In addition, the storage service has to take care of internal queuing, retransmission of events in case of failures as well as event persistency in case they didn't manage to be transmitted in time.

This would also be the case for all similar use cases across different services.

Due to above reasons, it is worth splitting the broker functionality and package it as a separate service. This service can focus on proper retransmission, high availability and persistency [12, p. 287] [16, p. 3] [4, p. 86] .

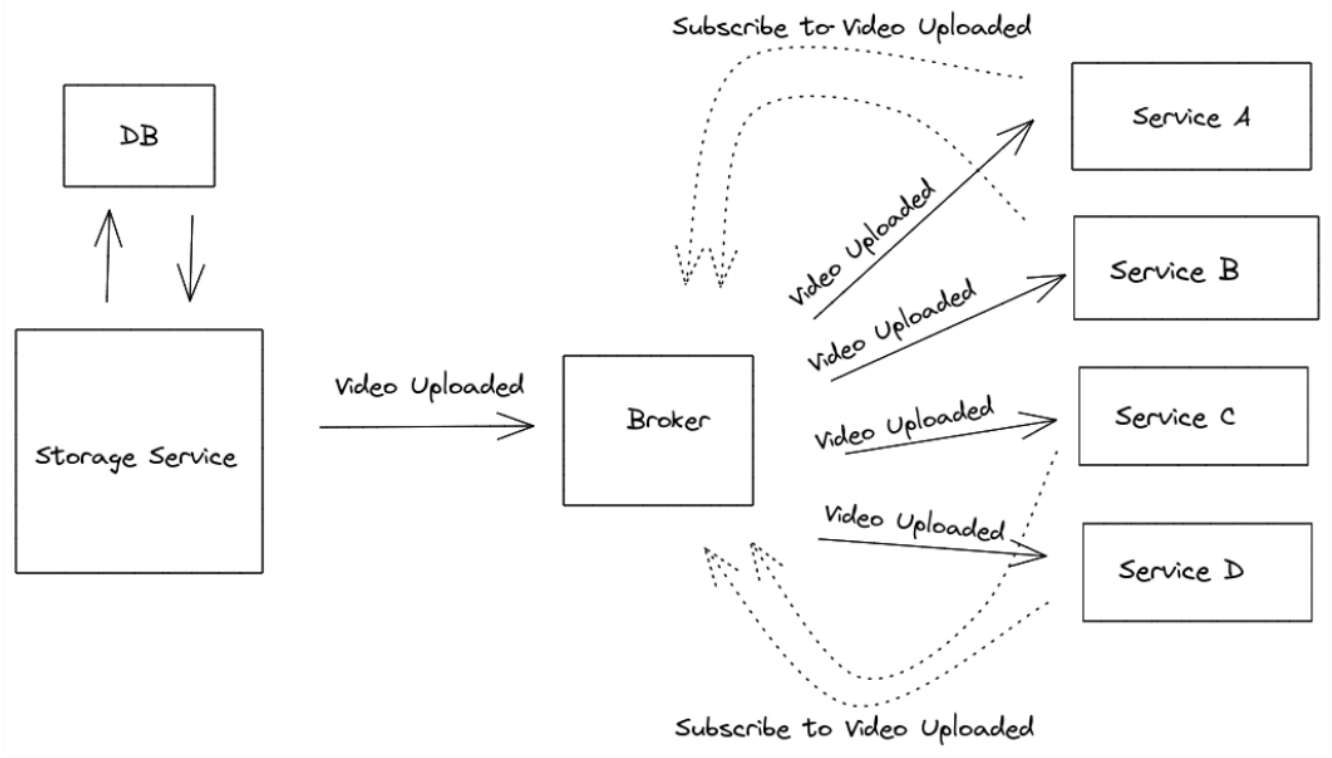


Figure 16: Event-driven architecture using a specialized message broker service.

Another big advantage of using a separate message broker service, is that we can use this service in order to break dependencies between other services. Instead of services reaching out to other services in order to subscribe to events, they can instead reach out to the specialized message broker service and subscribe to it. In the same spirit, any services that want to publish events, can contact the message broker and send the events to it.

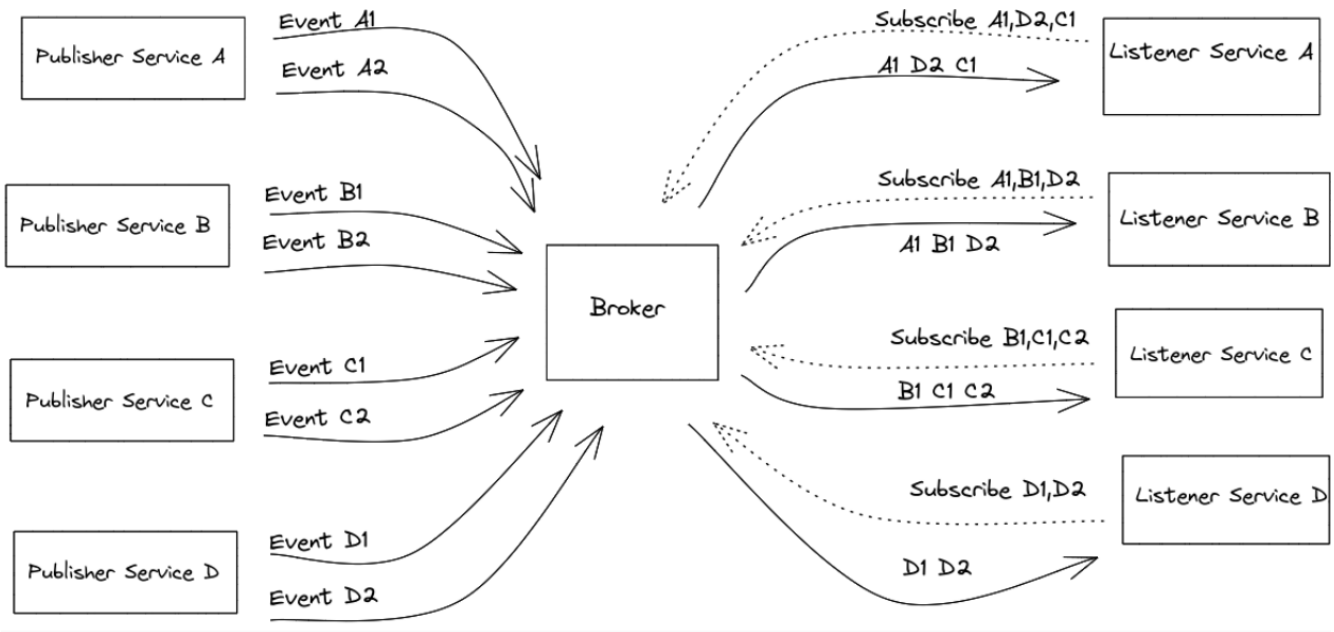


Figure 17: Broker as central dependency and message hub.

One potential issue of this approach is that while we manage to decouple the services from each other, we do however couple every service to the message broker.

This is often an acceptable compromise. In contrast with generic Microservices, the message broker can be specialized with the sole goal of supporting high-availability and convenient message semantics. This moves the burden of implementing such important functionality to a single service that can be properly tested in isolation.

Due to the message broker being central to event-driven infrastructures, it can frequently act as a single point of failure [17, p. 268] .

For this reason, understanding the exact safety and availability guarantees that a message broker provides is of extreme importance.

In order to be able to ensure that messages kept in message brokers are not lost, message brokers that attempt to take a central role in an event-driven architecture's infrastructure, often rely on redundancy in order to provide safety in the face of hardware and network issues.

Replicating a sequence of messages across different nodes fundamentally requires solving the distributed Consensus problem. This is non-trivial and relying on unproven custom algorithms can lead to destructive behavior during rare edge cases.

Using proven distributed Consensus algorithms such as Multi Paxos and Raft can help provide specific guarantees which can be relied upon when designing event-based architectures [18, p. 386] .

A great advantage of the lack of responses is that we can use queues and have all the advantages as discussed in the asynchronous request-response section without the large complexity that request-response bookkeeping introduces.

By changing the queue topology, we can often achieve various interesting communication schemes, allowing queues to act as message brokers [19] .

Publish-subscribe can be implemented by providing exchanges that can pull messages from a queue and send to multiple listeners. Not all queues provide exchanges-like functionality, however.

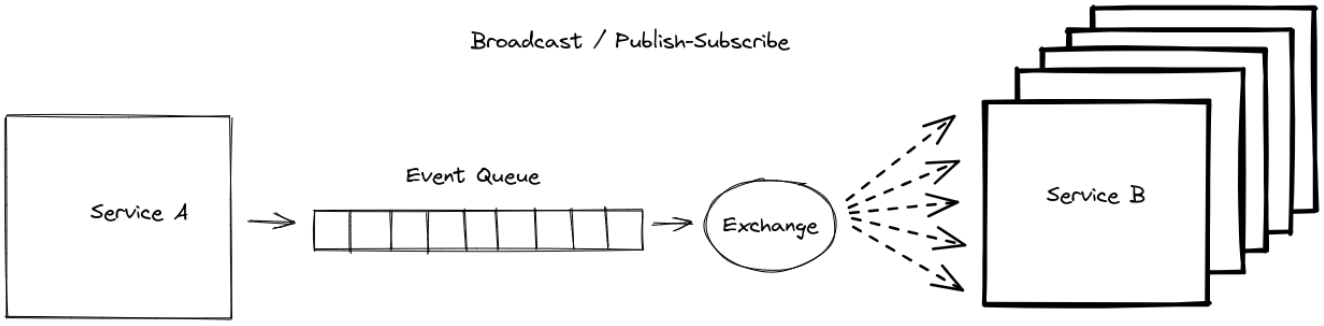


Figure 18: Queues along with exchanges can implement publish-subscribe.

Due to messages being extracted from the queue, Queue based message brokers can also trivially achieve task processing functionality by distributing tasks to a set of workers that compete to extract tasks from a queue [14] [20] .

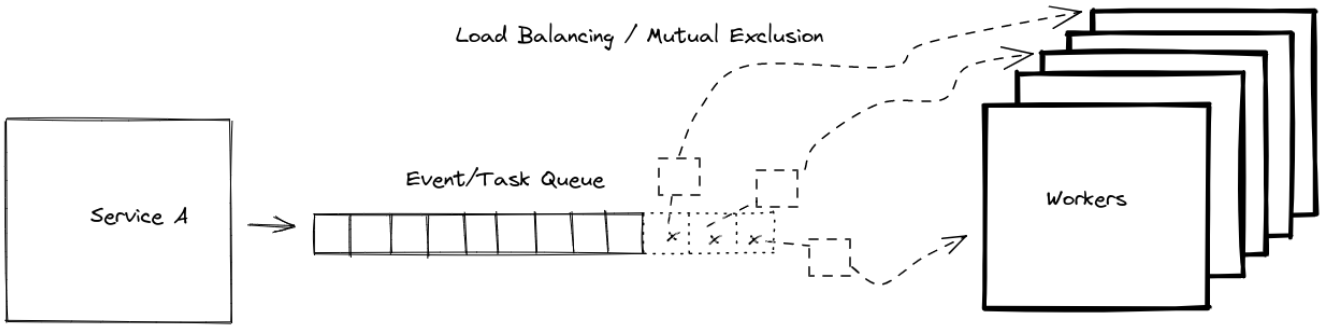


Figure 19: Worker instances compete for extracting messages from a single queue.

An alternative to queues are log-based message brokers . Fundamentally, such brokers use append-only logs in order to store published messages [18, p. 532]. Such logs abstractions can be mapped efficiently to disk-based persistent storage [21] .

Consumers can directly fetch messages from these logs, fetching messages from top to bottom, sallowing consumers to read messages in a first-in first-out fashion. This is sufficient for the basic *Publish-Subscribe* use case.

Since logs are persistent, such message brokers can persist messages and consumers can re-consume them if needed. This cannot be achieved by queues, which are fundamentally transient in nature.

A big advantage of log-based message brokers is the alignment of their fundamental log-based model with distributed log replication.

This is a well-studied problem, and known Consensus algorithms such as Raft can be used to provably provide replication guarantees [22] .

3.3 Command Query Responsibility Segregation and Event Sourcing

As we discussed in previous sections, sharing state between Microservices can create data coupling. For this reason, individual Microservices tend to use their own independent data stores and present a public API through which other Microservices, can issue commands and query state. This enables the Microservice to freely change the implementation of the underlying data store.

Within the context of a single Microservice, state management has interesting implications. It is often useful to present both Update and Query operations on the public API. These are typically implemented in the underlying Microservice using the same data model.

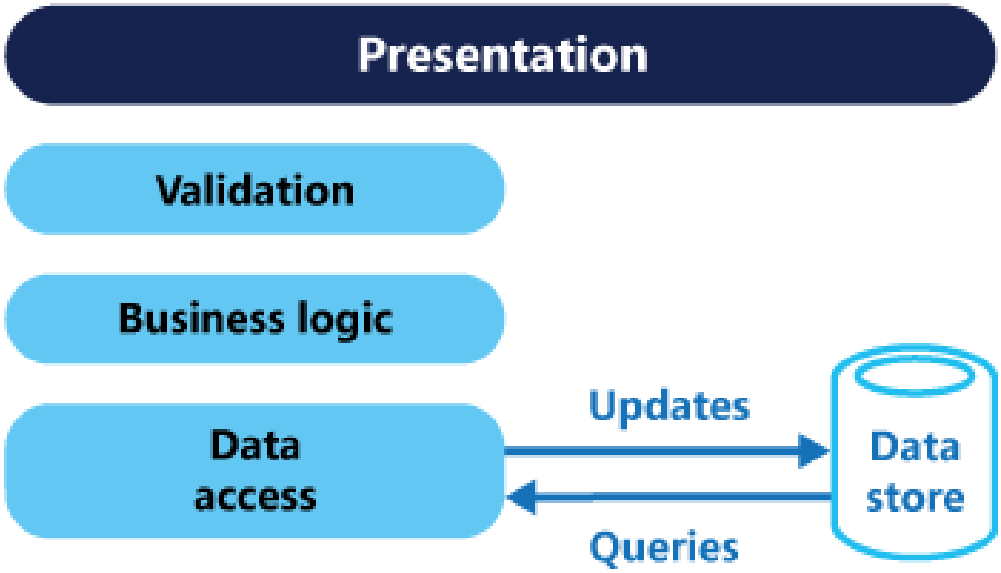


Figure 20: Updates and Queries are served from the same underlying model [23].

A problem with this approach is that the load of Queries compared to Updates is often different, with the Query endpoints usually receiving more traffic than the Update endpoints.

Furthermore, a Microservice may have to expose multiple Query interfaces that do not efficiently map to the underlying model. Often multiple Queries come at odds with each other, being limited by the single data model not being able to efficiently support all of them.

For the above reasons, it is frequently useful to separate the Update and Query data models .

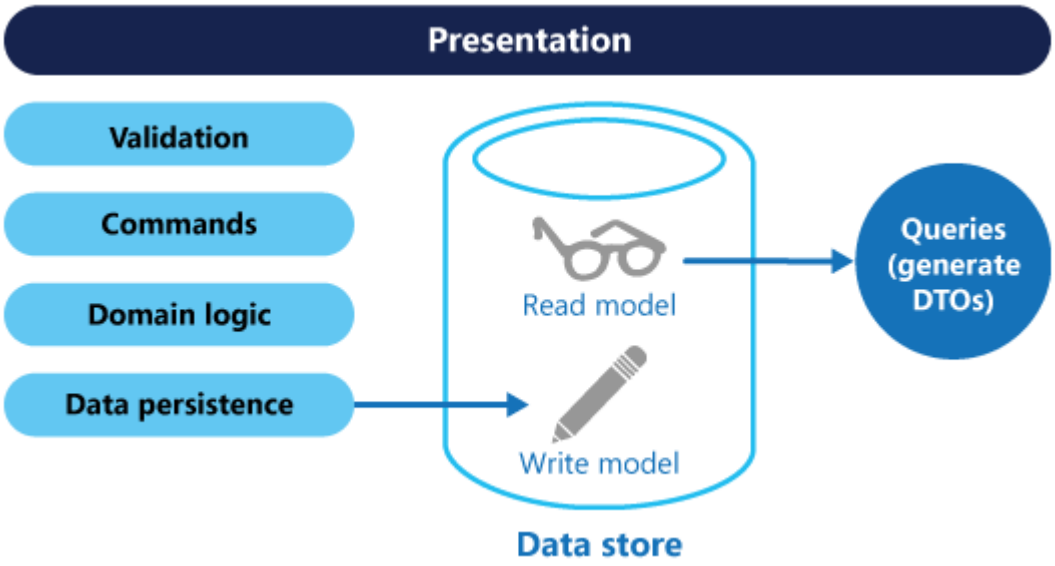


Figure 21: Updates and Queries are served from separate models [23].

This way, we can build the Query models in such a way that they efficiently map to their corresponding operations .

Going on step further, we can host the Query and Update models on separate data stores and perhaps on separate instances, allowing for independent scaling.

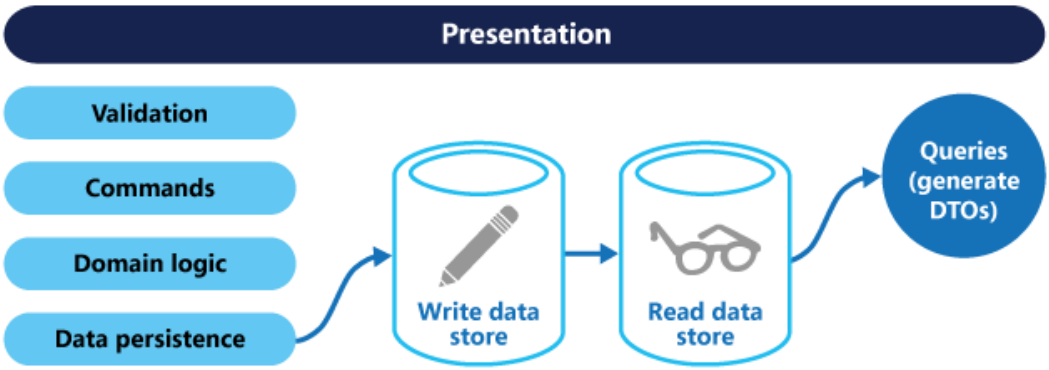


Figure 22: Updates and Queries models are hosted on different instances [23].

This pattern of separating the Update from the Query models is called *Command Query Responsibility Segregation* [24] [25] .

Keeping models in sync using event-based communication

At this point, the question arises of how we can keep the various Query models in sync with our main Update model.

As the Query models are increased, this situation starts taking the form of *Publish-Subscribe* communication.

Each verified Update must be transferred to multiple Query models, perhaps on separate instances, in order for them to keep their data in sync. This is usually achieved with event-driven communication, where Updates are transported as concrete events, to the various instances.

Event Sourcing

An interesting observation that we can make from the above discussion is that both the Update and Query data model states are derived from the underlying verified Update events that correspond to the commands issued through the Microservice's API.

At the extreme end, we can view databases as efficient projections of a given event-stream at a given point in time.

This view has led into the creation of a specific architectural pattern, that of *Event-Sourcing* [26] [24] [18, p. 543] [14, p. 43] .

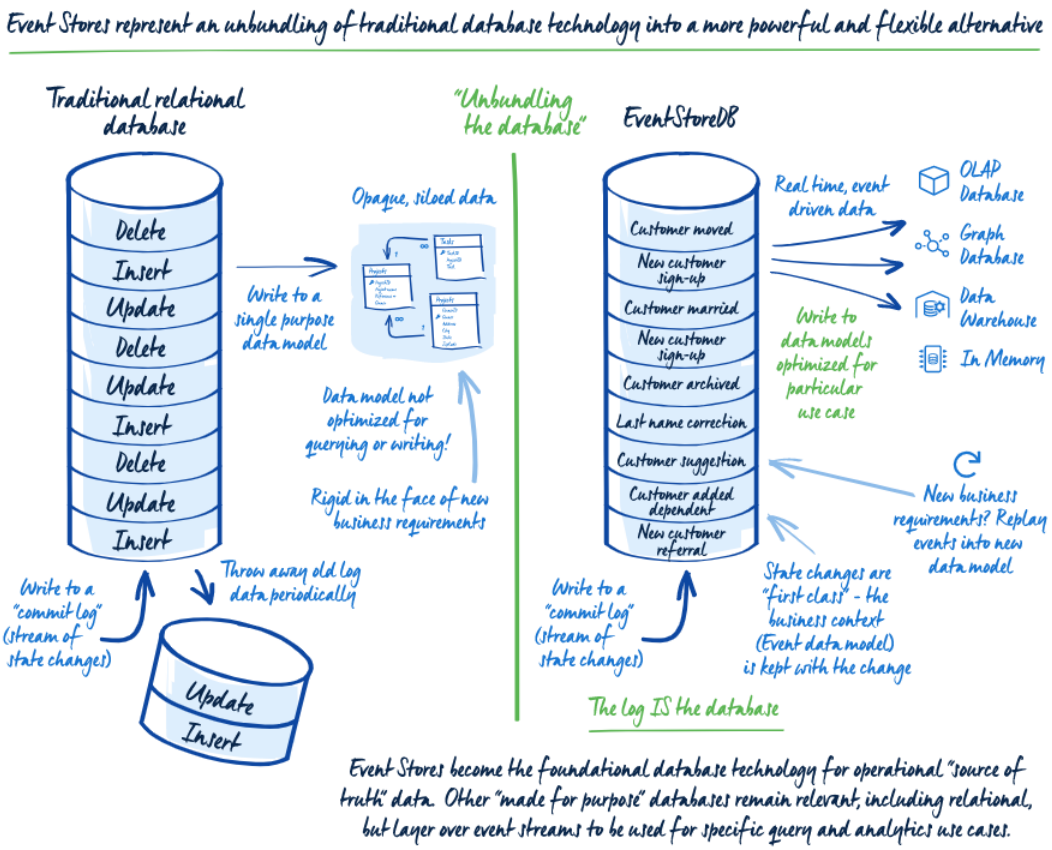


Figure 23: Event Sourcing as an architectural pattern [27].

Under *Event-Sourcing*, the fundamental source of truth is not the underlying database, but the

actual event stream.

This event stream is persisted into some sort of Event Store. Using the event stream, multiple projections can be derived from it, giving birth to aggregations, in-memory views, caches, concrete materialized views and, possibly, derived event streams.

In the case that a service becomes unavailable, it can then subsequently reconstruct all its relevant projections by replaying the events from the Event Store. In order for this to be done efficiently, the Event Store has to support efficient snapshot functionality.

Another advantage of this pattern is that since events are persistent, they can also serve auditing purposes. Furthermore, they can be used to fix incorrect derivations, perhaps in the case of implementation errors.

Strict safety and ordering guarantees from the side of the Event Store are paramount for the proper application of this architecture.

On-Demand Query models across Microservices

A big advantage of such event-based approaches is that we can transport the underlying high-level event streams across different services.

This permits various Microservices to derive projections for their own specialized usage [18, p. 549] [28] [29] .

Without permitting events to break the Microservice boundary, all dependent services would have to rely on the service's API supported Queries, putting the burden of maintaining them to the owning service.

Event-based communication can help decouple such services from a given service's API while permitting them to construct specialized projections of a given event stream themselves.

Eventual Consistency

A problem of applying CQRS and Event Sourcing is that due to different models consuming events at a different pace, such patterns can often only achieve eventual-persistency [30] [31] .

This in turn, creates complications for use cases requiring strict data consistency, limiting their usage as global architectural patterns.

4 Apache Kafka

4.1 History

Created initially by LinkedIn and subsequently open-sourced, Apache Kafka advertises itself as a modern event streaming platform, meaning a platform that facilitates the general management and high-throughput production, consumption, transfer, and processing of event-based data.

Kafka can act as a highly available and resilient publish/subscribe system while also providing persistence for event sourcing use cases and also covering real-time processing.

The term streaming is used to differentiate Kafka from other traditional Batch-processing frameworks, focusing on its real-time processing capabilities.

LinkedIn initially wrote Kafka [32] [17] in order to address various important data pipeline issues:

- Handle the transmission and processing of the growing amount of real-time events produced between its various services.
- Unify the different data pipelines within the LinkedIn infrastructure.

As a result, Kafka was built with specific attributes in mind:

Foremost, the biggest goal was throughput in order to support the growing number of data, as well as scalability in order to be able to scale for the future.

In order to properly decouple producers and consumers without introducing performance penalties, a mixed Push-Pull communication style was chosen Consumers initially poll, but also allow the server to subsequently push batched data within configurable limits.

Another important requirement was for Kafka to be flexible enough to cover multiple use case scenarios. As a result, Kafka is flexible towards its balance between consistency, throughput and availability, throughput and latency, persistence and infrastructure cost, parallelization and message ordering [33].

This makes Kafka a very configurable system, where different parameters can drastically affect its behavior and guarantees.

Kafka went open-source in 2010 and graduated from the Apache Software Foundation incubator project in 2012. It is currently used in some of the biggest data pipelines in the world [34] .

4.2 High Level Overview

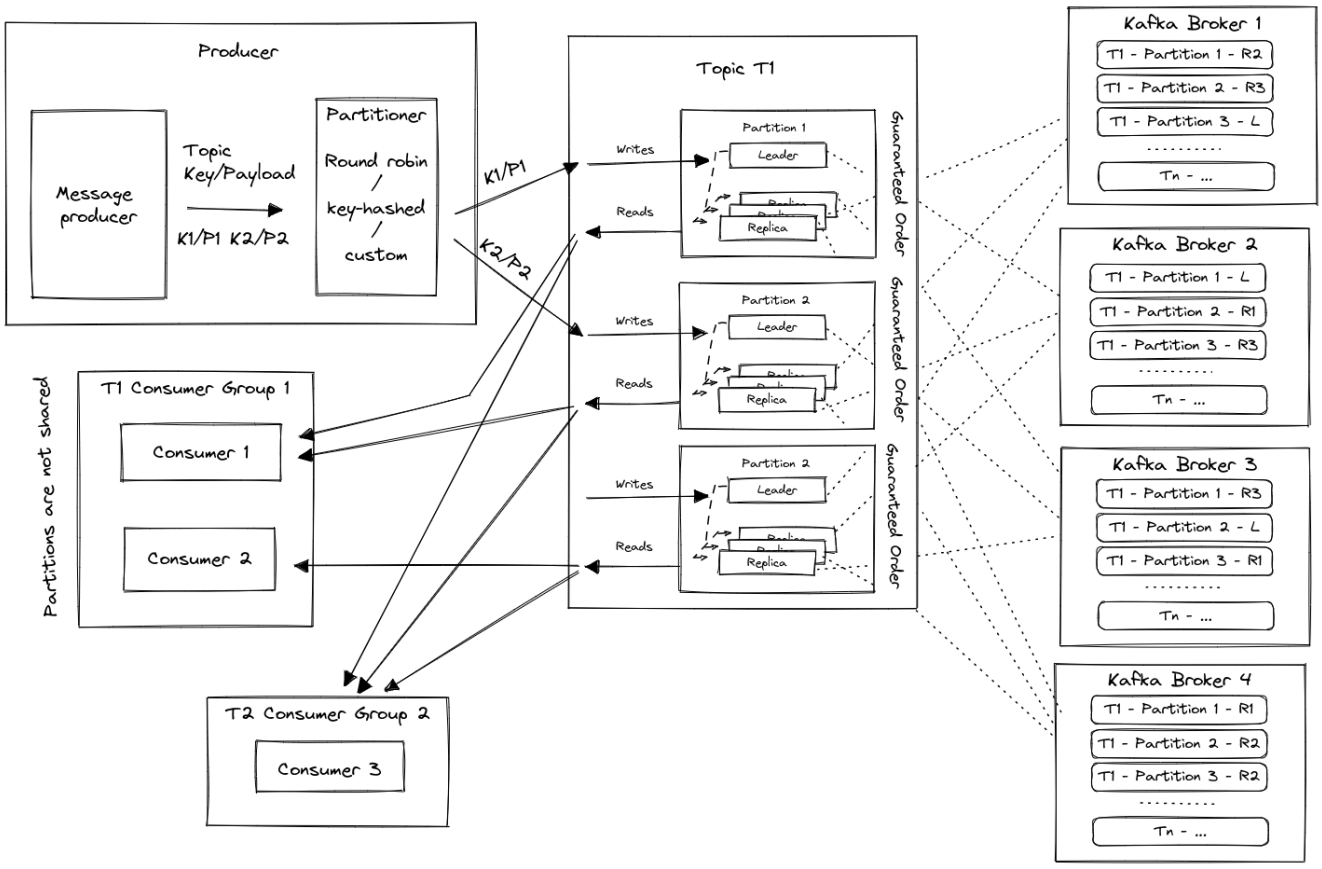


Figure 24: High-Level operational model of Apache Kafka.

This section gives a high-level overview of Kafka as well as introduce some minimal terminology that will be of help in the subsequent sections.

Partitions

Kafka at its core can be thought of as a distributed collection of logs. Each log exists on a potentially different server and consists of a sequence of key-value pairs. Each key-value pair has an associated position within the log called the message *offset*.

These logs in Kafka terminology are called *Partitions*

Consumers and Producers

Client applications speaking Kafka's binary communication protocol can query the cluster for metadata as well as fetch messages from each *partition* or produce messages to them by putting the messages at the end of the log.

Clients that consume messages are called *Consumers* and those that produce messages are called *Producers*

Producers chose the partition a message should go to and then contact the server responsible for

the partition, requesting that the message will be put at the end of the log, and getting a response regarding the request's status.

Consumers query messages by polling the server responsible for the partition and then specifying the message offset within the partition. They also specify the minimum and maximum data of the response and keep the communication channel open so that the server can immediately notify them when the data is available.

Consumers can query the messages in any order, but the most natural way is to read the partitions from top to bottom, effectively doing a First-In-First-Out traversal of the partition.

Due to the ordered nature of a partition, consumers reading the partition will receive messages in the same order as they were appended to the log.

Any number of consumers can read from the same partition, effectively allowing for Publish/Subscribe communication patterns.

Replicas

For availability and safety reasons, Kafka keeps potentially multiple copies of a simple *Partition*. These copies are called *Replicas* and will be discussed in detail along with Leader election.

Topics

In order to be able to scale throughput, Kafka uses collections of *Partitions* that contain the same type of messages. These collections of partitions are called *Topics*.

Producers send each message to a partition within a given *Topic*, while *Consumers* can read from any number of partitions within a *Topic*.

Consumer Groups

Finally, in order to be able to scale consumer throughput, Kafka uses the concept of *Consumer Groups*. These are groups of consumers that Kafka itself manages and assigns partitions to, such that each partition is only assigned to a single *Consumer*.

In the following sections, we will discuss each concept in detail and also look into the various configuration options of Kafka and how these affect the system's behavior as well as safety and availability guarantees.

4.3 Setup

Kafka is a distributed system in nature.

In order to be able to experiment with different Configurations and see how Kafka behaves in practice, we are going to need a proper cluster with multiple Kafka servers.

Kafka also currently requires Zookeeper as a dependency, which itself is a distributed service. At the time of writing an alternative feature that removes Zookeeper as a dependency, the Quorum-Based Controller, is announced as production ready.

All of this can non-trivial to set up and operate and for this reason we decided that for reproducibility reasons we utilize the Docker containerization framework and Docker Compose specifically.

Docker Compose allows us to compose multiple containerized services while declaratively changing their configuration.

The goal is to help anyone that wishes to reproduce our examples by automatically orchestrating the management, fetching and instantiation of the various required Docker images.

Docker will also be used for the tools that we will use in order to administer the cluster as well as produce and consume messages. This way, one doesn't need to download and install any tools in order to follow the examples.

All code samples used in this thesis, as well as all different Docker configurations, will be available in the accompanying GitHub repository

For demonstration purposes, one can initialize a Kafka cluster by going to the Docker folder on our repository and invoking *docker compose up*. The `-d` flag allows us to run the docker compose process in the background.

```
cd docker
docker compose up -d
```

This will instantiate a group of containers including a single *Zookeeper node*, 3 *Kafka servers* as well as various other tools for log collection and monitoring.

The Zookeeper server is exposed as a running service in our host server on port *2181*, while the different Kafka servers are exposed in the following way:

Docker Service	Host Port	Broker ID
kafka-0	9092	0
kafka-1	9093	1
kafka-2	9094	2

All other sections will assume an already initialized cluster.

In the following examples, we will be using some specific tools in order to perform basic administration and visualization tasks.

One useful Swiss-army-knife of Kafka tools is kcat. It can be used for basic cluster visualization, as well as consuming and producing to *Topics* and specific *Partitions*.

We will use edenhill/kcat docker image in order to invoke *kcat* without having to install it directly:

```
docker run -it --network=host edenhill/kcat:1.7.1
```

We will also take advantage of various native tools that Kafka itself provides. Kafka bundles these in the Kafka distribution's bin directory. We can use the bitnami/kafka that is also present in our Docker setup in order to access these tools without having to download the Kafka distribution and install the necessary Java libraries.

These tools can be invoked by running the `bitnami/kafka` image which includes the Kafka distribution's bin folder in its `PATH` variable, allowing us to run the tools directly:

```
docker run -it --network=host bitnami/kafka:3.1 KAFKA_TOOL.sh
```

In order to avoid specifying the docker part of all these tools repeatedly, we advise using command aliases:

```
alias docker-kcat="docker run -i --network=host edenhill/kcat:1.7.1"
alias docker-kafka="docker run -it --network=host bitnami/kafka:3.1"
docker-kcat --help
docker-kafka kafka-topics.sh --help
```

We include a file named `aliases.sh` in the `docker` folder containing such aliases, which can be directly sourced for convenience:

```
cd docker
source scripts/aliases.sh
```

In the subsequent sections, we will be using these aliased commands.

Finally, in our docker setup, we also include Kowl, a web application that can be used to visualize and administer a Kafka cluster. We will not discuss Kowl in depth in this thesis, but it is provided so that interested readers can use it as an alternative visualization source to the terminal based commands.

Kowl is exposed on port *8080*, so readers can point their browser to localhost:8080 in order to access it.

4.4 Partitions

At the heart of Kafka lies the concept *Partitions*.

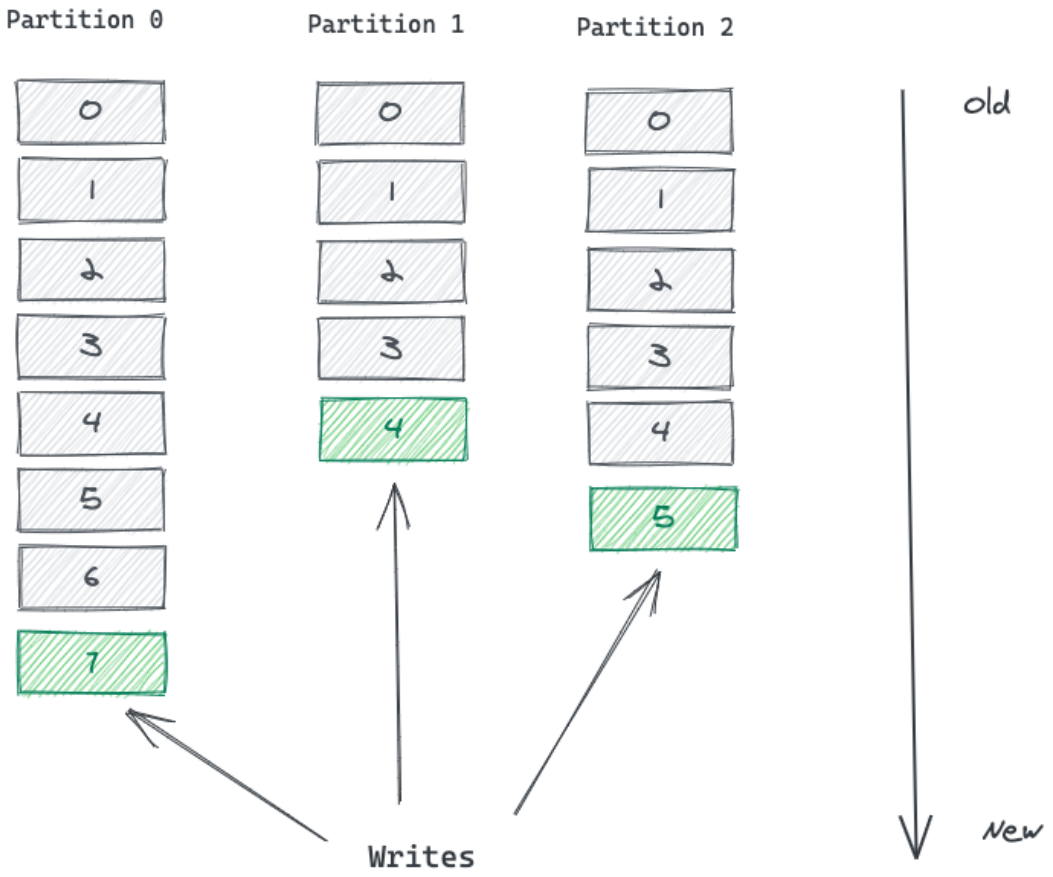


Figure 25: Messages being sent to multiple Partitions [35].

Each partition consists of a sequence of messages, where each message has an associated offset in the sequence [16, p. 33] . *Producers* append messages to the end of the sequence, while *Consumers* read messages by specifying their starting offset in the sequence.

Partitions in reality are concrete binary log files along with their indices and are hosted in *servers* [17, p. 207] .

The partitions use their associated *Topic* as a naming prefix. We will look into *Topics* in detail later on, for now topics can be thought of simply as prefixes when it comes to the partitions.

In order to examine *Partitions* we will start by using the native Kafka tool `kafka-topics.sh` in order to create a topic of a single partition.

As with every Kafka tool, we need to specify the *Bootstrap Servers* that the tool will attempt to initially contact in order to fetch metadata and retrieve information about the whole Cluster [16, p. 131] .

We remind readers that our docker setup exposes our Kafka servers on ports *9092, 9093, 9094* so we can use any subset of them as bootstrap servers.

```
docker-kafka kafka-topics.sh --bootstrap-server localhost:9092 \
                             --create --topic test --partitions 1
```

```
>>
Created topic test
```

At this point, we can either use `kafka-topics.sh` itself to retrieve partition information:

```
docker-kafka kafka-topics.sh --bootstrap-server localhost:9092 --describe
```

```
>>
Topic: test TopicId: 91r3O5DUQV2BWwBqHZUiIQ
PartitionCount: 1   ReplicationFactor: 3
Configs: min.insync.replicas=2,segment.bytes=1073741824
    Topic: test Partition: 0    Leader: 1   Replicas: 1, 2, 0   Isr: 2, 0, 1
```

Or use `kcat`:

```
docker-kcat -b localhost:9092 -L -u
```

```
>>
Metadata for all topics (from broker 0: localhost:9092/0):
 3 brokers:
  broker 0 at localhost:9092
  broker 2 at localhost:9094 (controller)
  broker 1 at localhost:9093
 1 topics:
  topic "test" with 1 partitions:
    partition 0,  leader 1,  replicas: 1, 2, 0,  isrs: 2, 0, 1
```

The same info can also be found in Kowl by browsing to Cluster > Topics > test > Partitions

At this point, I would like to advise readers to ignore the `replicas` and `isrs` portions of the command output. These will be explained in detail in Replication. In contrast, I would like to focus on the `leader` column, which specifies the Leader of the partition, meaning the Kafka server that is responsible for this partition [17, p. 16] .

We note that the `leader` for partition `test-0` is the server with ID `1`

We will now append some messages to the `test-0` partition by using `kcat`.

```
docker-kcat -b localhost:9092 -P -t test -p 0 -K:
```

```
>>
k1:msg1
k2:msg2
k3:msg3
^D
```

Flag `-P` specifies that we want to produce to the partition, flag `-t` specifies the topic, flag `-p` specifies the partition and `-K` specifies the key delimiter. Keys are part of Kafka messages and are used by Producers in order to choose the concrete partition that each message will be sent to.

They are also used in Log Compaction which we will discuss in detail later.

Under the covers, `kcat` uses the binary Kafka protocol to fetch metadata from the bootstrap servers, discover the leader of the specified partitions and then send *Produce* requests to them including the messages.

The leader server may use zero-copy techniques in order to immediately append the messages to the underlying partition log [21] .

We can now connect to the container of the server with ID `1` in order to see how Kafka actually stores partitions. All runtime data in Kafka by default is stored in the `/tmp/kafka-logs` folder.

```
cd docker
docker compose exec -it kafka-1 bash
cd /tmp/kafka-logs && ls
```

```
>>
cleaner-offset-checkpoint  meta.properties  replication-offset-checkpoint
log-start-offset-checkpoint  recovery-point-offset-checkpoint  test-0
```

A few notes about these files :

- `cleaner-offset-checkpoint` [36] Contains information regarding current *cleaning* progress.
- `meta.properties` [37] Contains simple metadata that Kafka reads on startup for restoration purposes.
- `replication-offset-checkpoint` [38] Is the file which Kafka uses to track which messages were successfully replicated to other servers. See Replication for more details.
- `log-start-offset-checkpoint` [39] Is a file containing the *low water mark* of assigned partitions. Basically the smallest offsets that consumers can consume from this server.
- `recovery-point-offset-checkpoint` [39] Is the file where Kafka tracks which messages were successfully checkpointed to disk. See Replication for details on what it means to checkpoint messages.
- `test-0` Is what interests us at the moment. It's a directory containing information about the partition `0` of topic `test`. A server may be assigned various partitions of different topics, in this case the server will contain multiple such directories.

We can now `cd` to the `test-0` directory and inspect it's contents:

```
$ cd test-0 && ls
00000000000000000000.index  00000000000000000000.timeindex  partition.metadata
00000000000000000000.log    leader-epoch-checkpoin
```

A few notes about these files:

- `partition.metadata` Is just a metadata file containing metadata meant to avoid split-brain.
- `00000000000000000000.log` Is the actual log file containing the messages of the partition.
- `00000000000000000000.index` Is an index file used to efficiently find the position of a message in the binary log given a message offset.
- `00000000000000000000.timeindex` Is an index file used to efficiently find the position of a message in the binary log given a timestamp.
- `leader-epoch-checkpoint` Is another file containing the last known leader of the partition. Used to avoid split-brain.

We can use the native `kafka-dump-log.sh` Kafka tool from within the server to visualize the contents of both the log and the index files:

```
$ kafka-dump-log.sh --files 00000000000000000000.index
>>
...
$ kafka-dump-log.sh --files 00000000000000000000.timeindex
>>
...
$ kafka-dump-log.sh --files 00000000000000000000.log --print-data-log
>>
Dumping 00000000000000000000.log
Starting offset: 0
baseOffset: 0 lastOffset: 2 count: 3 baseSequence: -1
lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
isTransactional: false isControl: false position: 0
CreateTime: 1652541195236 size: 100 magic: 2 compresscodec: none crc: 613208378
| offset: 0 CreateTime: 1652541195236 key: k1 payload: msg1
| offset: 1 CreateTime: 1652541195236 key: k2 payload: msg2
| offset: 2 CreateTime: 1652541195236 key: k3 payload: msg3
```

I would also like to acknowledge that all three binary files can actually be split into segments. This is discussed in Log Retention

As we can see, the log file contains the messages previously produced using `kcat` in the same order we specified them.

We can now exit from the Kafka server and use `kcat` as a consumer in order to read the messages in the `test-0` partition:

```
$ docker-kcat -b localhost:9092 -C -u -t test -p 0 -K:
k1:msg1
```

```
k2:msg2
k3:msg3
% Reached end of topic test [0] at offset 3
^C
```

The consumer under the covers, uses the Kafka protocol to fetch metadata, discover the server responsible for the partition (except when using rack-aware fetching, [40]) and then use *Fetch* requests to directly read messages from the underlying partition log [17, p. 194].

We notice that after the consumer consumes the last message, it keeps an open connection to the server. It specifies to the server the minimum and maximum amount of bytes required to notify it after getting new messages. So in essence Kafka consumers use a Push Pull model in order to communicate with the Server, allowing for decoupling but also reducing latency in case of new messages [33].

Each consumer is independent, and we can spawn multiple consumers reading from the same partition in parallel.

Since we have discussed the basics of operation, now it's a nice opportunity to also look into what happens when we have multiple partitions and a consumer fetches messages from both of them.

We create another topic named `test2` with 2 partitions:

```
docker-kafka kafka-topics.sh --bootstrap-server localhost:9092 \
                             --create --topic test2 --partitions 2
```

We then send a few messages to both of them:

```
docker-kcat -b localhost:9092 -P -t test2 -p 0 -K:
k1-1:msg1-1
k1-2:msg1-2
k1-3:msg1-3
^D
```

```
docker-kcat -b localhost:9092 -P -t test2 -p 1 -K:
k2-1:msg2-1
k2-2:msg2-2
k2-3:msg2-3
^D
```

Then a consumer may read messages from any of them, getting messages in the same order they were produced:

```
docker-kcat -b localhost:9092 -C -u -t test2 -p 0 -K:
```

```
>>
k1-1:msg1-1
k1-2:msg1-2
k1-3:msg1-3
```

```
docker-kcat -b localhost:9092 -C -u -t test2 -p 1 -K:
```

```
>>
k2-1:msg2-1
k2-2:msg2-2
k2-3:msg2-3
```

A consumer may also read from multiple partitions simultaneously. We will only specify the topic, forcing `kcat` to read from all partitions under the topic prefix:

```
docker-kcat -b localhost:9092 -C -u -t test2 -K:
```

```
>>
k2-1:msg2-1
k2-2:msg2-2
k2-3:msg2-3
k1-1:msg1-1
k1-2:msg1-2
k1-3:msg1-3
```

We get the messages of `test2-1` before the messages of `test1-1` even though we produced them in the reverse order. The consumer fetches messages in parallel from the separate partitions, which means that messages only maintain a strict ordering with respect to messages only in the same partition, total order of messages across partitions is not maintained [16, p. 33] .

It's very possible depending on the batching that Kafka performs to get a result such as:

```
>>
k2-1:msg2-1
k1-1:msg1-1
k1-2:msg1-2
k2-2:msg2-2
k2-3:msg2-3
k1-3:msg1-3
```

4.5 Log Retention

Kafka so far has been presented as an append-only distributed log service. This however, raises concerns about its storage requirements.

Retaining newer events is certainly very useful, Publish-Subscribe communication becomes trivial under this model and events can be audited and processed multiple times as needed.

After some point, however, some events will have been fully processed in all relevant services and their reprocessing will not be as relevant anymore. It would be a waste for Kafka to retain all such events forever. If Kafka did not have retention policies, logs would be unbounded and disk storage requirements would quickly become prohibitive.

For this reason, Kafka supports both time-based and storage-based retention policies, as well as two specific deletion policies that we will explore in detail.

In Partitions we discussed that Kafka keeps messages in binary `.log` files along with its indices. An extra detail that we skipped is that these log files can be segmented.

Each segment is named after the offset of the first message that it contains and also has its own index files. All appends affect only the latest *active* segment.

After either enough time passes, or after the *active* segment reaches a certain size, then the current active segment becomes *inactive*, and a new active segment is rolled out. Inactive segments can serve reads, but no writes.

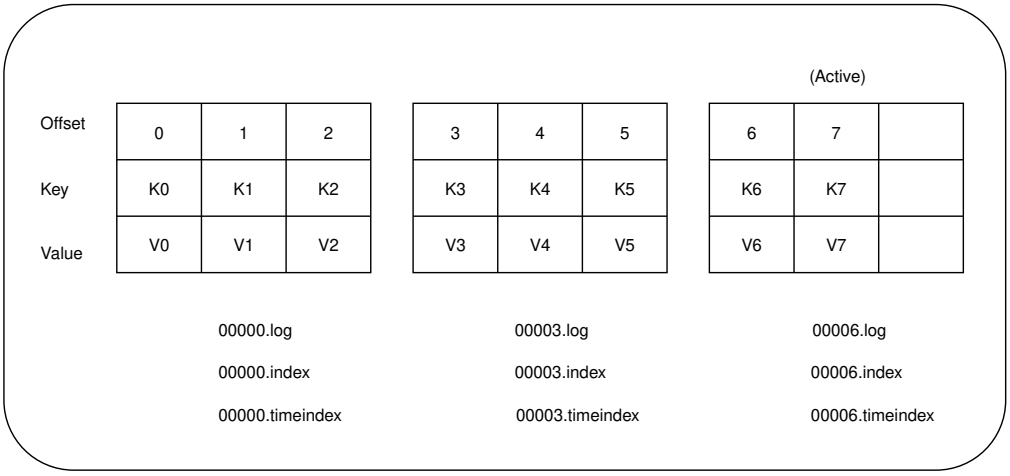


Figure 26: Partitions as multiple segments.

The time limit, after which a new active segment is rolled out, is controlled by the log.roll.hours and log.roll.ms configuration options.

The maximum segment size after which a new segment is rolled out, is controlled by the log.segment.bytes configuration option.

After an active segment becomes inactive, it is then subjected to the retention settings that are in place. Kafka supports both size and time based retention, and both apply independently. Both settings apply to segment files, not individual messages, marking them as expired.

With time-based retention, one can put an upper limit after which a segment file and its indices will be affected by the chosen *cleanup* policies.

Relevant configuration options are:

- log.retention.ms
- log.retention.minutes
- log.retention.hours

An important detail is that the settings for time-based retention like we mentioned apply only to inactive segments. Moreover, the settings apply to the last modified time of each segment, which is usually the time when a segment turns inactive. This means that actual retention time may be significantly larger than these settings, especially on low-throughput topics.

As an example, imagine having set `log.roll.hours = 168` which is also the default. Then we also set `log.retention.hours = 240` with the intention to expire messages after 10 days. If the per-partition throughput is low enough, a new active segment rollout, may not happen until the configurable setting of 168 hours 7 days is up. Only then will the retention setting kick in, resulting in messages expiring after 17 days instead of 10.

Alternatively, one can configure the maximum size of each **partition** in order to enforce a soft upper limit on disk storage. This can be hard to properly use since the topic-level behavior differs depending on the number of partitions and does not take peaking days/hours into account. It is disabled by default.

Relevant configuration option is: - log.retention.bytes

Cleanup Policies

These policies serve to define how Kafka will handle segments after the retention settings have marked them as expired. There are currently two policies, *Delete* and *Compact* and are controlled by the log.cleanup.policy configuration option.

Log cleaning is enabled by setting the log.cleaner.enable configuration option. If disabled, no cleanup takes place and nothing happens to expired segments. By default, cleanup is enabled.

Delete Policy

This policy is pretty straightforward, any expired segments are simply deleted. One can also add an extra delay after a segment becomes expired and before being deleted using the log.segment.delete.delay configuration value.

Compaction Policy

The compaction policy allows for a specific record-based retention behavior.

Instead of deleting whole segments, compaction goes through inactive segments and deletes "outdated" messages, keeping the latest message with a given key instead.

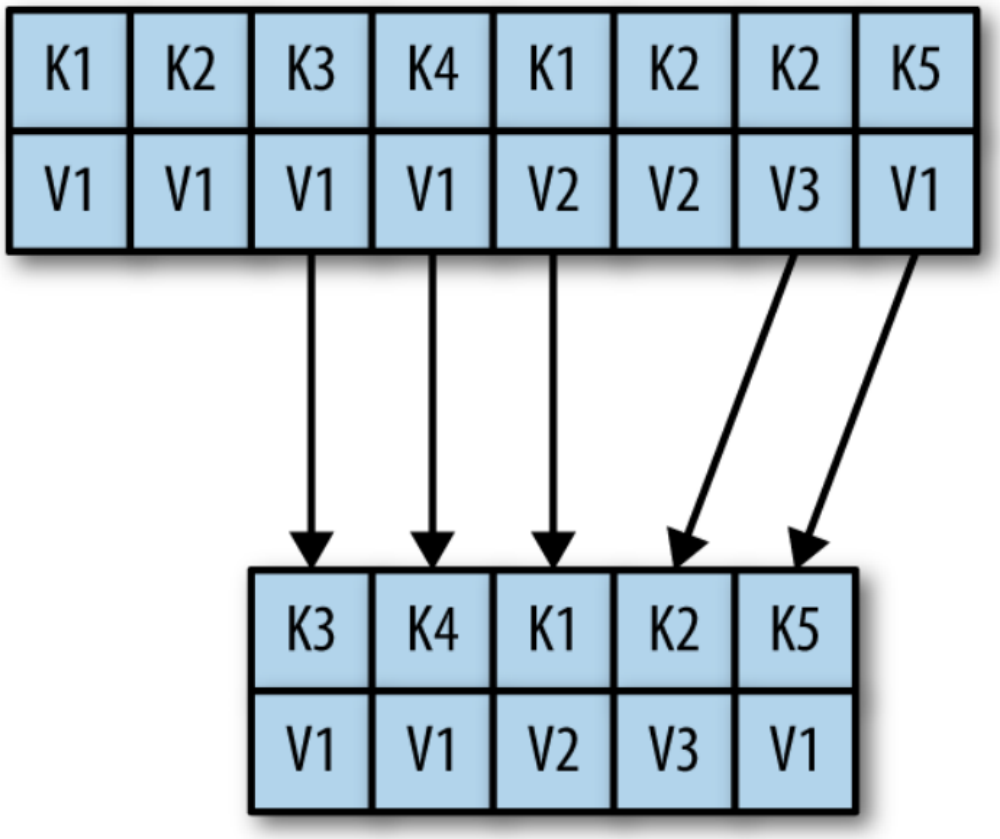


Figure 27: Compaction removes "outdated" entries [17].

Compaction also works across segments, merging them in the process.

Relevant configuration options are:

- log.cleaner.min.cleanable.ratio — the minimum ratio of the non-compacted partition size to the total partition size in order for a partition to be eligible for cleaning. It acts as a throttling mechanism due to compaction being resource-intensive.
- log.cleaner.min.compaction.lag.ms — prevents segments newer than a minimum duration from being compacted. The default value is 0, implying there is no minimum time that an inactive segment will persist before it may be compacted.
- log.cleaner.max.compaction.lag.ms — the max duration that a segment may idle before being compacted. Typically used with low-throughput topics.
- log.cleaner.delete.retention.ms — the extra duration that the tombstone records are persisted before being deleted. We will discuss tombstone records below.

Tombstone messages are messages with a key and a null value. They can be used to denote record deletions. When the compaction cleaner comes across a tombstone message and `log.cleaner.delete.retention.ms` time elapses, then the tombstone message is deleted, allowing from proper deletion of specific keys from the resulting compacted event stream. `log.cleaer.delete.retention.ms` will ensure that delayed Consumers will have an extra chance to notice the deletion event before the cleaner removes it completely.

This compaction policy is especially useful for record-based message streams where values correspond

to entities and keys correspond to primary keys. It is meant to serve Event Sourcing scenarios [27] , with compaction taking the place of snapshots. Given enough time, compaction will attempt to ensure that any given entity identifier used as key will correspond with a single / most recent value.

4.6 Replication

We briefly mentioned that when clients want to either consume or produce messages to a Partition, they use the metadata fetched using the Kafka protocol in order to find the server responsible for the given Partition and then send further requests to it. We remind readers that this server that is responsible for the Partition is called the Leader for that partition.

It is important to discuss what happens if, for any reason, a Kafka Server becomes unavailable. This could happen for a multitude of reasons, including hardware server problems, networking issues or even due to manual intervention, for example due to turning off the server for maintenance reasons.

In this case, a window of unavailability is created for all clients of any partitions of which the server is the Leader.

Another very significant issue is the fact that **Kafka does not immediately write received messages to the disk**, even if it acknowledges them.

For performance reasons, Kafka uses *memory-mapped* files under the scenes and periodically flushes them to disk. This means that if a Server goes down unexpectedly, there is a very real possibility that messages that were just received successfully from Producers, will not have the chance to be persisted. As a result, these messages will be completely lost when the Server goes down [16, p. 215] [17, p. 196] .

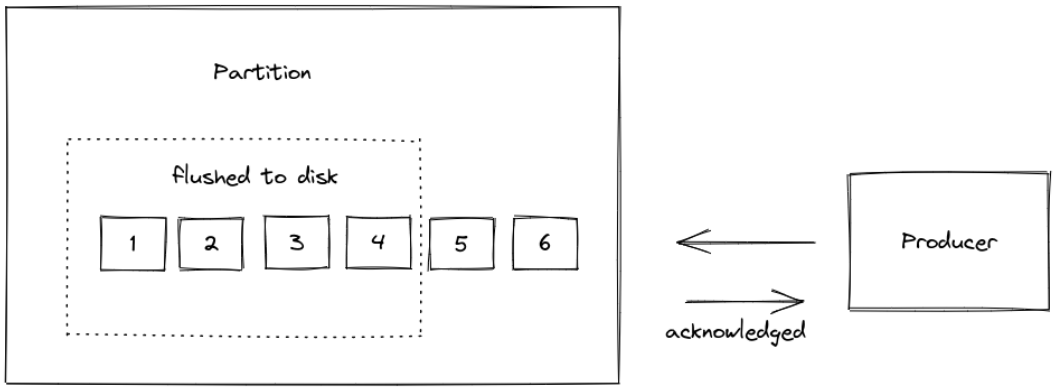


Figure 28: Acknowledged messages not flushed.

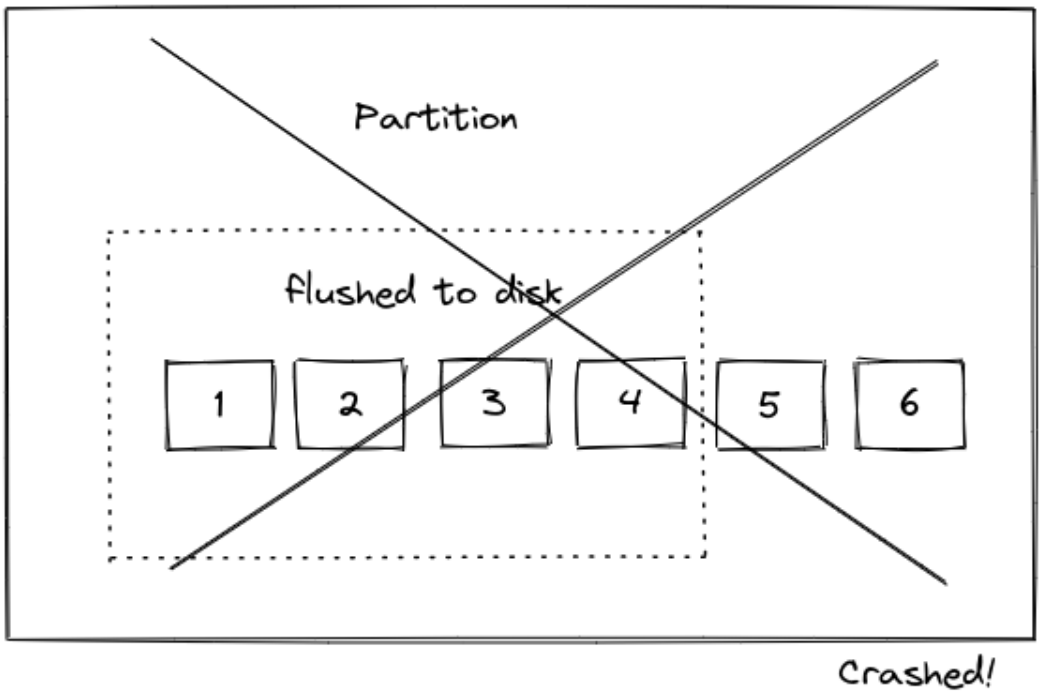


Figure 29: Server crashes losing non-flushed messages.

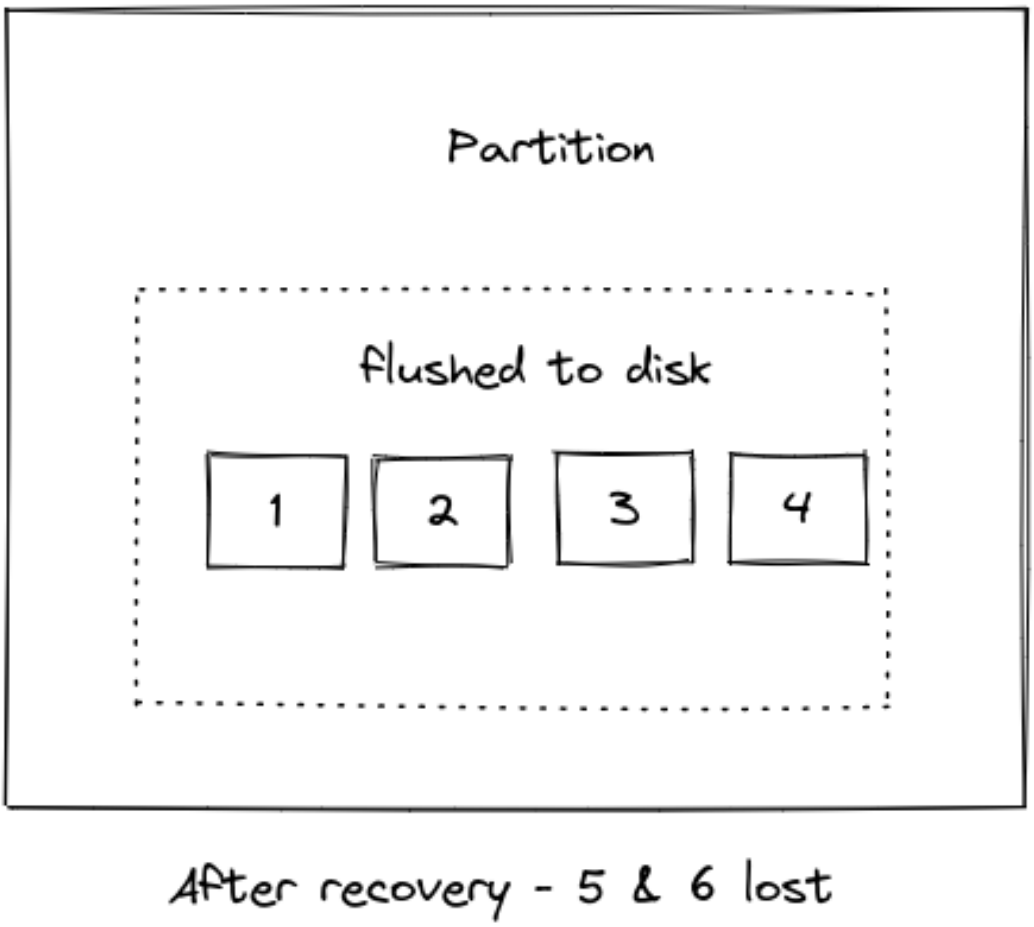


Figure 30: Server recovers with missing messages.

Kafka in order to provide message safety guarantees allows us to specify a number of copies of each partition. This is done on the Topics level – which we haven't discussed in detail yet. The copies of a partition are called *Replicas*, and the *Replica* in the Leader server is called the *Leader replica*.

We can think of the Leader replica as the main source of truth for a partition. Clients only communicate directly with the Leader replica. The rest of the replicas fetch messages from the Leader in the background using the same *Fetch requests* as normal Consumers, and then append these messages to the end of their own logs. These partition copies are called *Follower replicas* [17, p. 189].

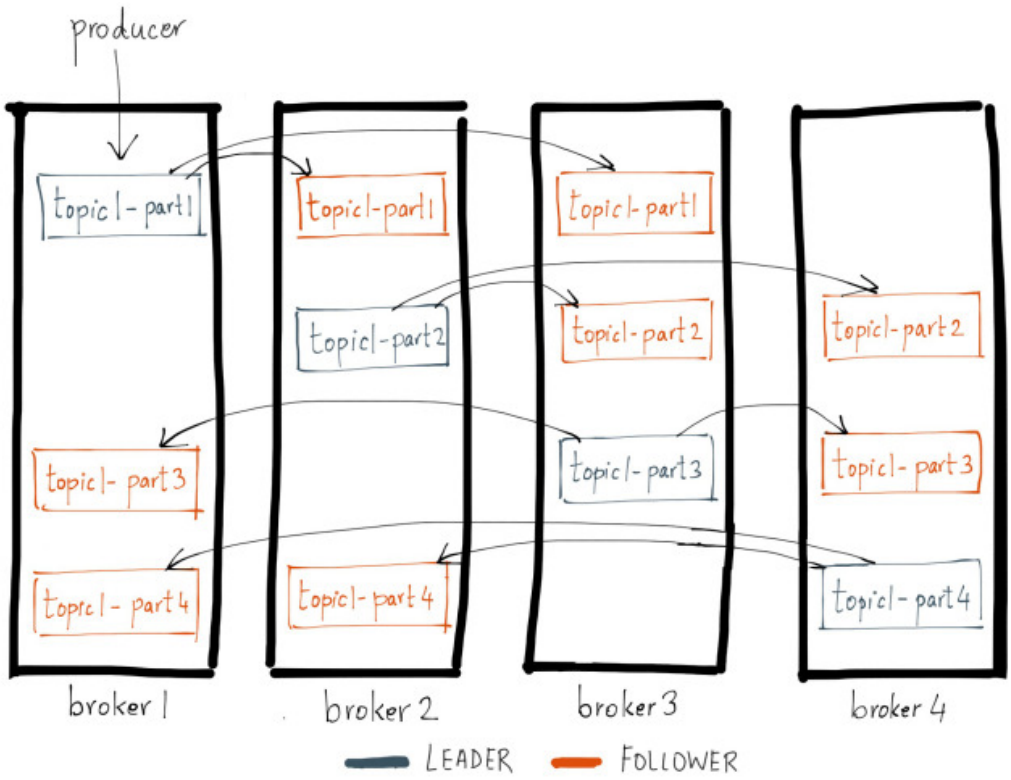


Figure 31: Example of Leader and Follower replicas [41].

Having discussed the basic mechanism of replication, we can now focus on how Kafka behaves when a server becomes unavailable.

When Kafka detects that the Leader of a partition does not communicate with the Cluster, it starts a *Leader election* process, where the Follower replicas synchronize and attempt to elect a new Leader between them. After a new Leader is elected, clients automatically redirect their traffic to it.

At this point an important question arises, what if the new Leader has not managed to copy all messages from the previous Leader? How does this affect the cohesion of the messages?

Initially, we discuss what happens from a Consumer's point of view:

The Leader replica keeps track of the latest message offset that was successfully replicated to **all** (*This is not completely true, see In-Sync-Replicas later on*) the *Follower* replicas. This is called the High Watermark.

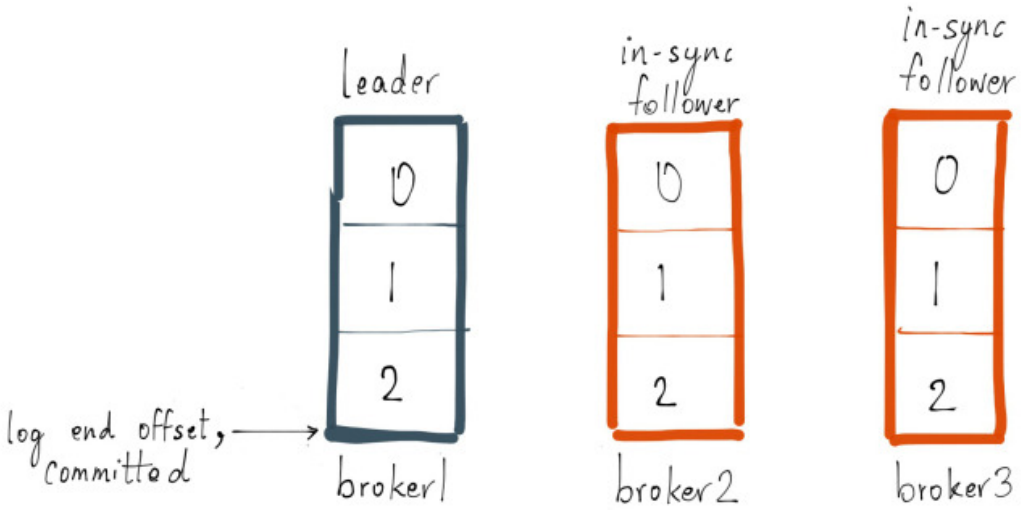


Figure 32: Leader and Followers all in-sync [41].

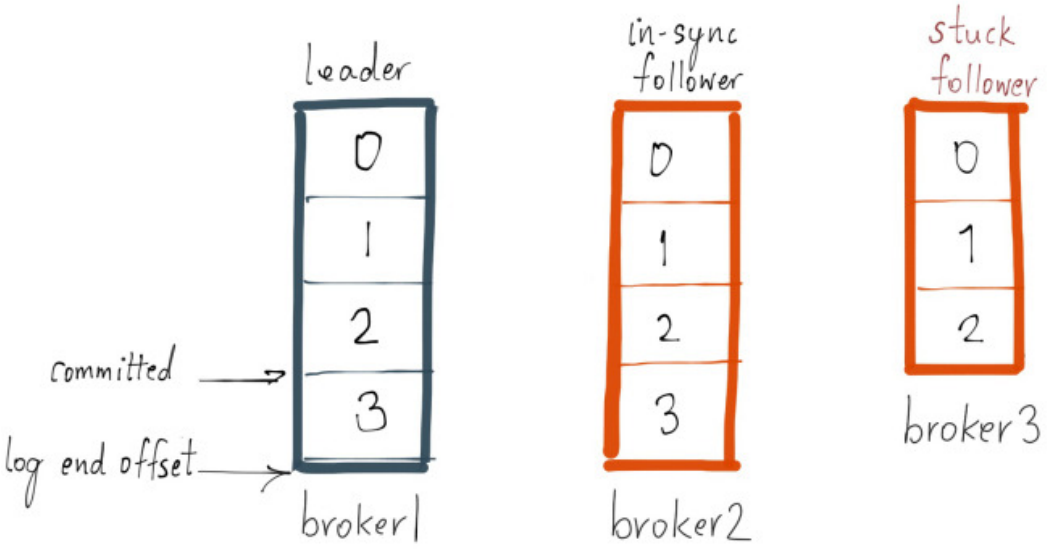


Figure 33: Not all messages are replicated, log end offset differs from High Water Mark [41].

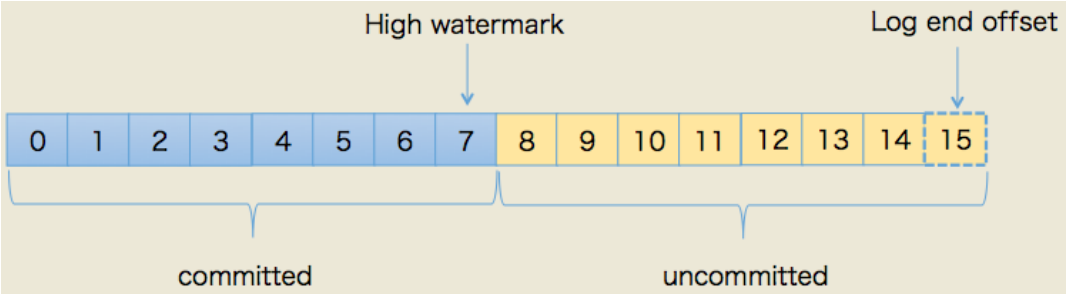


Figure 34: High water mark points to the latest committed message.

Only messages up to the *High Watermark* are available for consumers to fetch, messages that have not been fully replicated, are not visible to the Consumers.

This way cohesion from the point of Consumers is ensured, any messages not available to *all* Replicas, including the new Leader, would not be visible in the first place.

If any messages were fetched before the Leader election, then that means that they were fully replicated, so they will also be available in any new elected Leader.

Things from a Producer's point of view are more complicated.

Depending on the use case, a consumer may not be fully interested in the safety and/or cohesion of its messages.

For instance, we can think of a Producer publishing sensor measurements to Kafka.

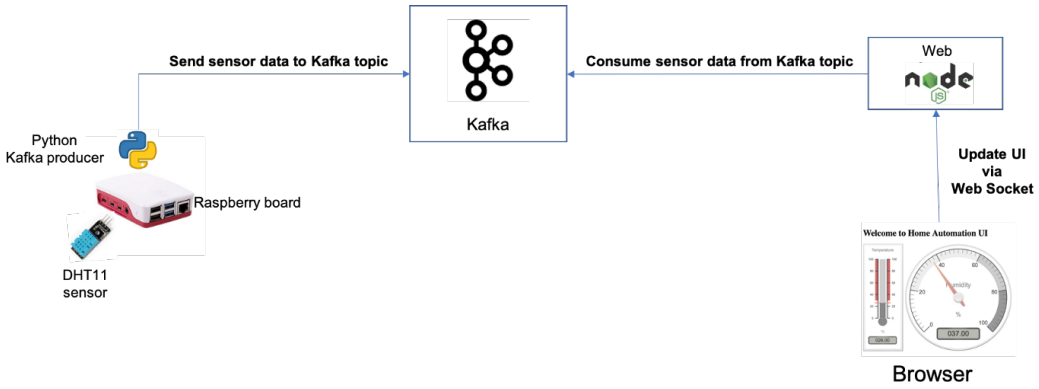


Figure 35: Sensor data is collected and transferred to a home automation UI [42].

In this case, losing messages is most likely not a big issue.

In cases where message delivery and safety is not essential, a producer may be more interested in total throughput and may not want to pay the price of waiting for an acknowledgment before sending a new message.

On the other hand, we can think of a video sharing application where videos are uploaded on a storage service.

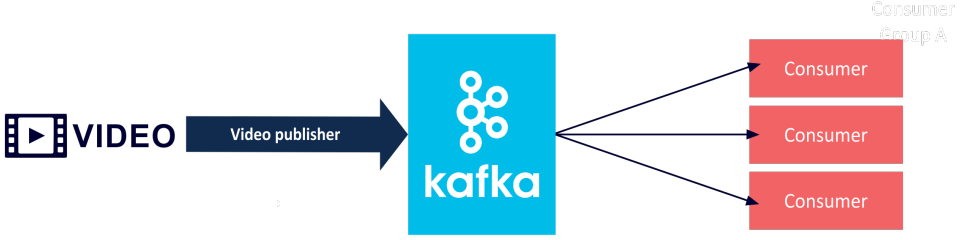


Figure 36: Service notifies other services that a video has been uploaded [43].

When a video is uploaded to the storage service, the service would publish an event notifying downstream services that a new video is available. Those in turn would fetch the new video and perform further processing, resuming the full upload process.

In this scenario, it would be extremely problematic if the storage service did not have any guarantees about the safety of it's transmitted events.

In order to be able to cover such different use cases, Kafka allows Producers to configure the acknowledgment behavior of the Cluster [16, p. 214] [17, p. 189].

This capability is exposed through the acks configuration parameter.

The *acks* configuration parameter has 3 possible values:

- *acks=0*

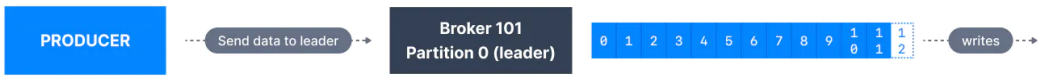
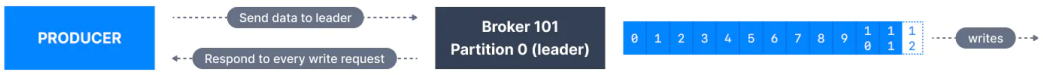


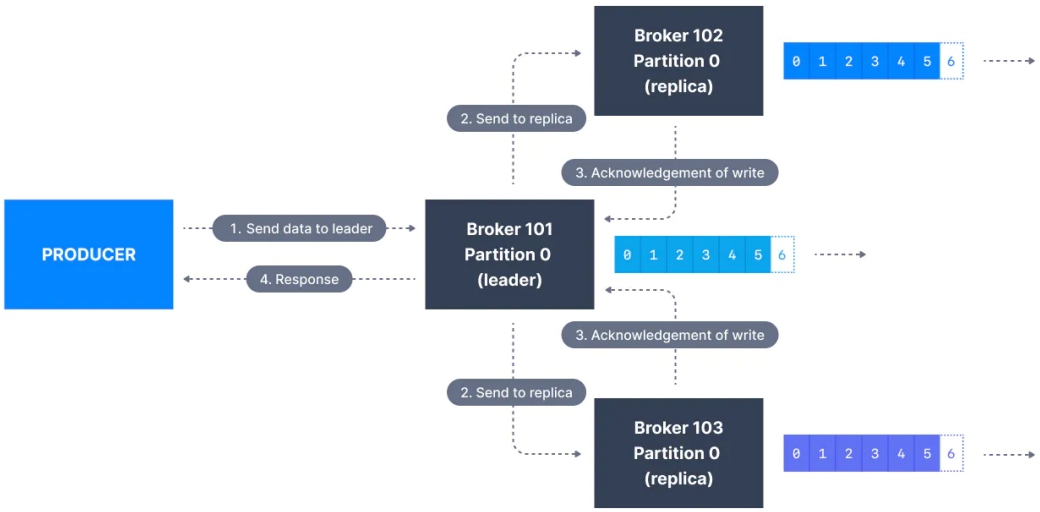
Figure 37: `acks=0` does not wait for acknowledgment [44].

In this case, the Producer is not interested in what happens to a sent message. It does not expect any acknowledgment at all from the Cluster, it doesn't even attempt to verify if the message has reached the Cluster. As such, it can resume sending messages immediately, achieving very high throughput. Of course, there are no guarantees about the safety of the messages.

- *acks=1*



This has traditionally been the default setting, creating issues for inexperience Kafka users (*This is no longer the case since Kafka 3*). With this setting, the Cluster sends an acknowledgment to the Producer as soon as the message arrives at the Leader replica of the message's target partition. We remind readers that **messages are not immediately persisted**. It is therefore possible that a message arrives at the Leader, the Leader sends the acknowledgment to the unsuspecting Producer client, but then the server goes down before the message has been persisted or replicated to another Replica, losing the acknowledged message in the process. The Producer will think that the message has been successfully persisted, but the message would be lost and unavailable to any Consumers. As such, we would like to stress that **this configuration parameter does not provide any real safety guarantees**. Its only advantage compared to the *acks=0* setting is that the Producer can get information about the *offset* that the message will be persisted, which can occasionally be useful. The producer achieves lower throughput compared to the *acks=0* setting since it has to wait for the acknowledgment message before resuming.



- *acks=all*

  With this setting, the Leader server will respond with acknowledgment to the Producer if and only if the message is successfully **transmitted** (*not persisted*) to **all** (*This is not completely true, see In-Sync-Replicas later on*) Replicas. This allows us to have some concrete safety guarantee about our messages. Specifically, if the Leader sends an acknowledgment back to the Producer, and we have $N + 1$ replicas for a given partition, then we can guarantee that our messages will persist even with up to N replica server losses (*again, this is not completely true, see the following In-Sync-Replicas section more details*). The downside of this setting is that the Producer will have to wait for the message to be successfully replicated to *all* replicas. This means that they have to wait for the slowest of all the replicas to catch up before getting an acknowledgment and resuming production, severely reducing its total throughput.

At this point, we would like to stress that the total end-to-end latency does not change with any of the different options, since Consumers in any case will not see any messages that are not fully replicated. What is affected is Producer throughput due to waiting for acknowledgments between sending different messages. Even this can be mitigated by waiting for acknowledgments asynchronously and retrying sending messages, this however is not always applicable depending on the message ordering and delivery requirements.

The fact that *acks=1* has traditionally been the default setting is an unfortunate historical artifact, it is important however to keep note of since it severely reduces the safety guarantees. After Kafka version 3, *acks=all* is the default Producer configuration, since message safety is usually more critical than Producer throughput and the end-to-end latency is not impacted.

At this point I would like to address all the notes acknowledging the inaccuracy of the above claims.

I want to start by making a few observations about the above specified behavior.

We discussed that Consumers will not be able to read messages as long as those have not been *fully* replicated. In the same spirit we said that *Producers*, when operating under the *acks=all* configuration setting, will get an acknowledgment from the Leader only after the message has been *fully* replicated.

Replica assignment to different Kafka servers, only occurs during topic creation (*or manual re-assignment*) and the set of replicas is fixed.

What changes afterwards, is their availability status as well as which of them is designated as the

Leader.

At this point, we have to discuss, *how is it possible that the partition stays available if* ***any*** *server goes down.* After all, as long as *full replication* implies replication to the whole initial replica set, then *full replication* cannot take place as long as one server is not available.

This means that *Producers* operating under *acks=all* would not get a reply, while the *High Watermark* would not be increased and the messages would not become visible to the consumers.

Another issue regarding the *acks=all* setting is how slow replicas affect acknowledgment latency. We can imagine that a single replica falls too far behind when it comes to replication. This may occur due because it just restarted after a big window of unavailability, due to temporal network issues, or perhaps due to operational load impacting its replication speed.

The mechanism that Kafka provides in order to address the above issues is that of the *In-Sync-Replicas* set. This is a dynamic subset of the initial full *replicas* set that keeps track of all the replicas that are *sufficiently* in-sync with the Leader replica of the partition [45] [46] [41].

What *sufficiently* means in this case is configurable. The relevant configuration parameter is replica.lag.time.max.ms which defines the upper bound on the time that a Follower replica has not reached the Leader partition's log end offset.

In the Partitions section, we advised readers to ignore the *replicas* and *isrs* initials in the `kafka-topics.sh`'s `--describe` as well as `kcat`'s output. These can now be understood as the initial full replicas set and the dynamic In-Sync-Replicas set, respectively.

It is now an appropriate time to address the inaccuracy notes of the previous section.

Full replication in the context of the *High Watermark* advancement and *acks=all* configuration setting refers to the *In-Sync-Replicas set* not the initial full replica set.

Let's examine *acks=all* in detail.

In the below example, the partition has a replication factor of 4. Only replicas in servers 1,2 and 3 are sufficiently in-sync. Server 4 is not part of the in-sync replica set. As a result, server 1 which is currently the Leader only waits for servers 2 and 3 to replicate the message 6 before responding with acknowledgment to the Producer.

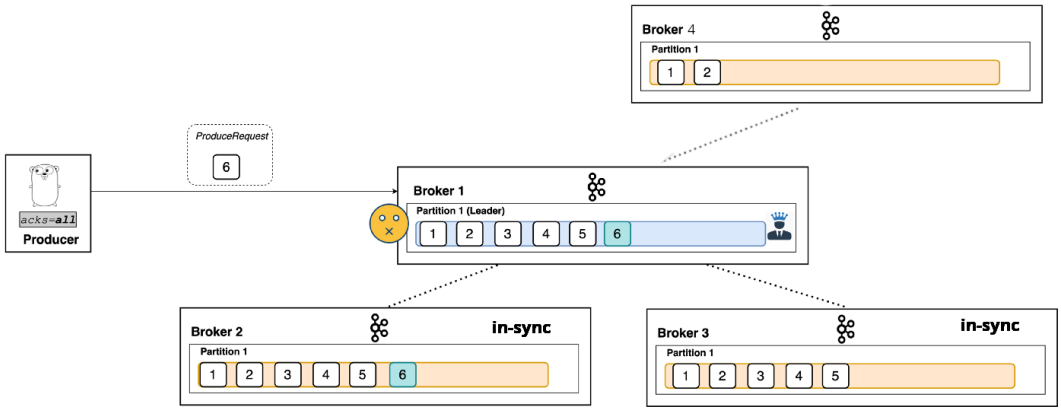


Figure 38: Message has been replicated in two out of three in-sync replicas [45].

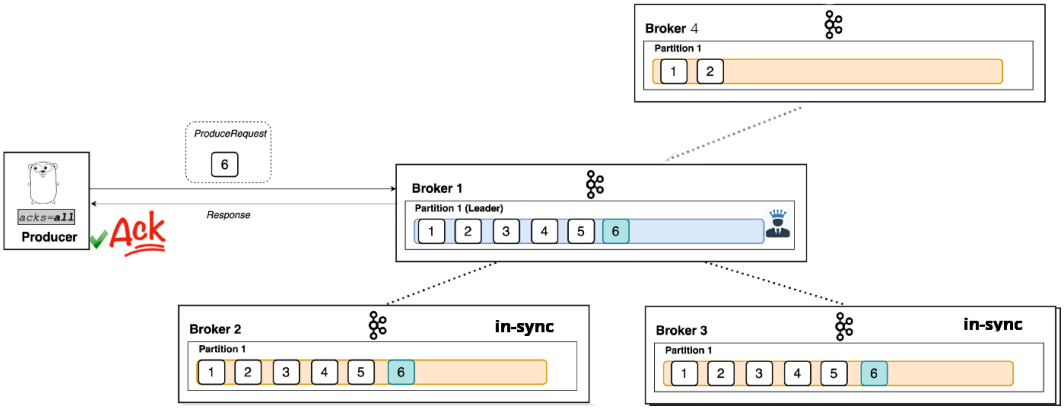


Figure 39: Message is replicated in three out of three in-sync replicas but has not been replicated in out-of-sync fourth replica. Producer receives acknowledgment [45].

In similar spirit, the leader replica waits only for the followers that are part of the current *In-Sync-Replicas* set to replicate a message before increasing its High Watermark.

The *In-Sync-Replicas* set basically allows Kafka to reduce the total acknowledgment latency and increase the availability of the partition. If a replica becomes unavailable, it will eventually be removed from the *In-Sync-Replicas* set and the Leader will resume sending acknowledgments and increasing the High Watermark, making the messages visible to the Consumers.

This behavior is in direct opposition to the simple safety guarantees that we discussed in the *acks=all* configuration option

We previously discussed that having a replication factor of $N + 1$ allows an acknowledged message to persist even with up to N replica losses. This is not entirely true in the presence of the *In-Sync-Replicas* set.

We revealed that with *acks=all* a message does not have to be replicated to all replicas, instead it only requires to be replicated to the replicas that are part of the current *In-Sync-Replicas* set.

With our current formulation of the *In-Sync-Replicas* set, we have no constraints on the number of its members. The set can be shrunk indefinitely, for example after a chaining reboot of the replica servers, in the worst case it can end up with just a single member – that of the current Leader. In this case, the *acks=all* behavior becomes effectively indistinguishable to the *acks=1* setting, losing all its safety guarantees in the process.

Below we show a concrete example, We have 3 servers and a replication factor of 3. Two of the servers have gone offline, ending up with a single member in the *In-Sync-Replicas* set – the Leader.

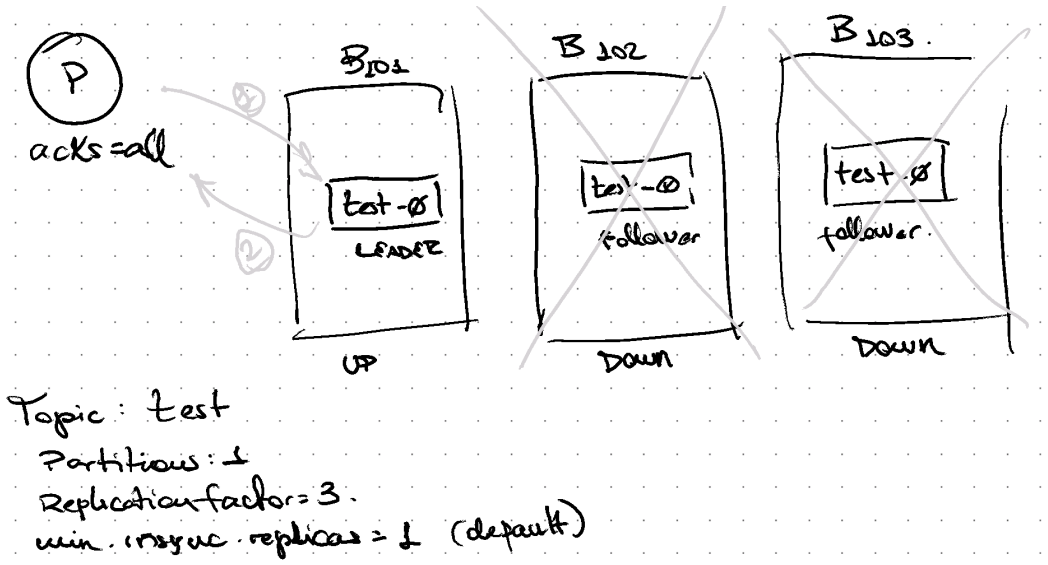


Figure 40: `min.insync.replicas=1` - no replication safety limit. Producer receives unsafe acknowledgment, due to the message being replicated to all replicas, in this case just the Leader [46].

In this case, the leader will immediately respond to the Producer with an acknowledgment, since it has successfully replicated the message to all servers in the current *In-Sync-Replicas* set – that is, itself. This however, just like in the case of *acks=1*, means that in the case of Leader failure, acknowledged messages will most likely be lost.

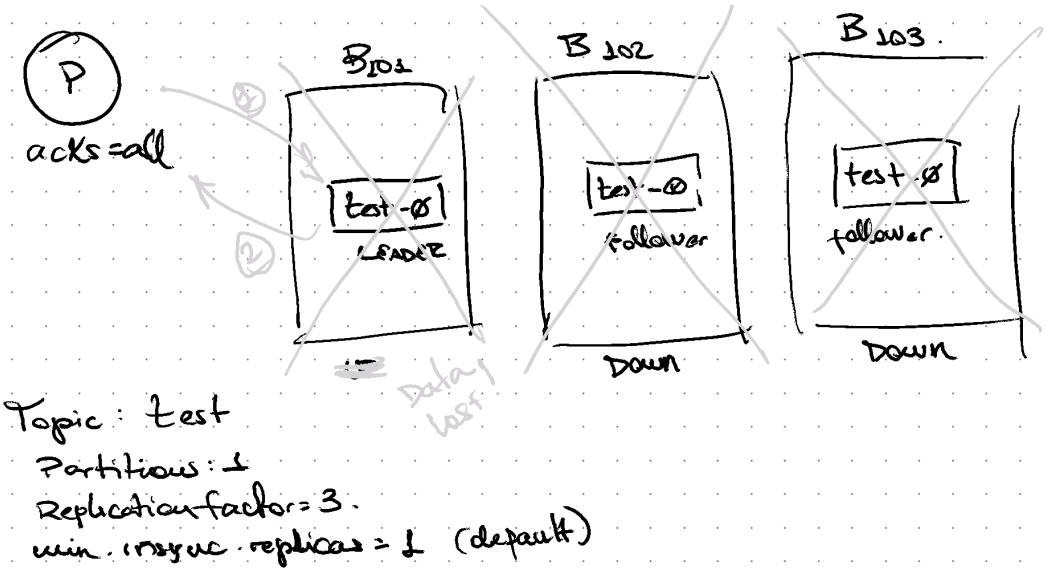


Figure 41: Leader crashes. Producer has received acknowledgment but message may be missing after recovery [46].

What is needed in this case is a mechanism that allows us to balance the availability and safety of messages. Kafka provides such a mechanism in the form of the min.insync.replicas configuration option.

This configuration parameter basically sets the minimum required members that the *In-Sync-Replicas* set should have before the Leader responds with an acknowledgment under the *acks=all* setting.

In the below example, we again have a replication factor of 3 and a *min.insync.replicas* value of 2. When only the Leader is available, it does not acknowledge the messages, since the current *In-Sync-Replicas* set does not have the required size and is therefore deemed unsafe to do so.

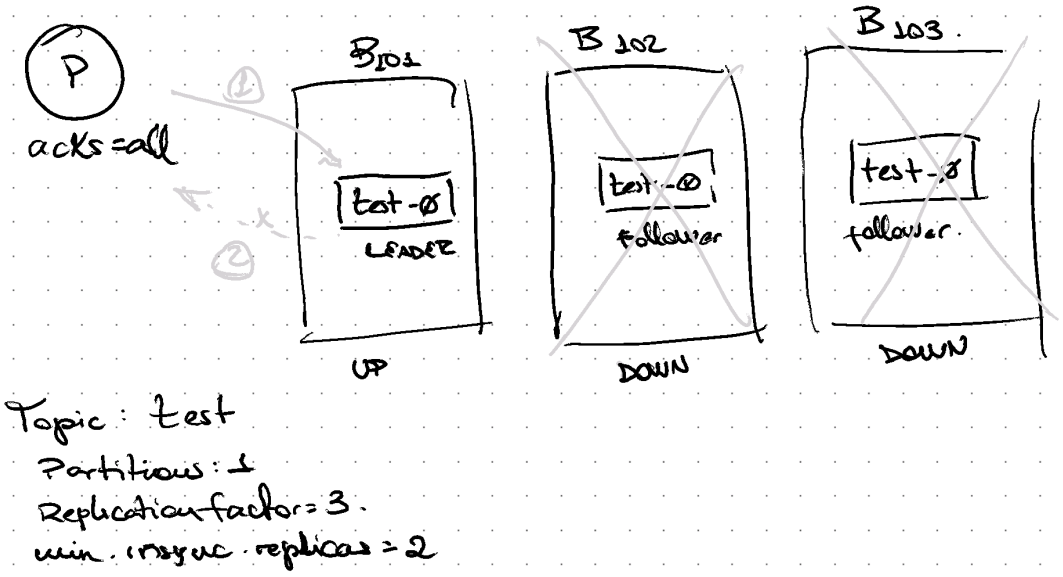


Figure 42: `min.insync.replicas=2` - requires at least 2 replications for acknowledgment. Avoids acknowledging in the unsafe situation of just the Leader being available [46].

The default value of *min.insync.replicas* is 1. This is done so that single server clusters can work out of the box. Unfortunately, this can hurt unsuspecting production configurations if not kept in mind, especially those that explicitly set the Producer *acks* configuration parameter to *all*, thinking that their message safety is guaranteed without this being the case like the above example demonstrates.

At this point it would be useful to properly recap the guarantees that Kafka actually provides taking into account the *In-Sync-Replicas* set along with the *min.insync.replicas* configuration parameter [17, p. 218] [47] .

- A producer cannot have any real guarantees about the safety and coherence of its sent messages unless it has been configured with *acks=all*
- In this case min.insync.replicas $= N + 1$, guarantees no message loss for up to N replica failures.
- If min.insync.replicas $= N + 1$ and $N \leq$ replication factor $\leq 2N$, Kafka can take advantage of the *In-Sync-Replicas* mechanism in order to remove the slow partitions from the set and successfully increase acknowledgment speed as well as read availability. In the case of N failures, message will survive and reads of existing messages will still be served, unfortunately there won't be enough replicas to left to meet the requirements of the *min.insync.replicas* parameter and therefore writes won't be acknowledged until enough partitions restart and join the *In-Sync-Replicas* set. We therefore achieve message safety and read availability, but no write availability in the worst case.

- If min.insync.replicas = $N + 1$ and replication factor $\geq 2N + 1$, Kafka provides message safety, as well as read and write availability even in the case of N replica failures. Even if N of the current replicas set go offline, there will still be enough replicas available to join the *In-Sync-Replicas* set.

From the above, it's obvious that we require at least 2 servers in order to guarantee any sort of message safety.

The takeaway is that if we want to be able to withstand N server failures we need min.insync.replicas = $N + 1$ which also implies at least $N + 1$ servers.

If we don't care for full write availability in the worst case, we can have any number of servers from $N + 1$ to $2N$ with more servers potentially permitting improving acknowledgment latency in erroneous scenarios.

If we want both safety and full write availability in the worst case, we require at least $2N + 1$ servers.

Finally, for completeness, it's worth mentioning another relevant configuration parameter – unclean.leader.election.enable

This parameter specifies what happens when all partitions part of the *In-Sync-Replicas* set become unavailable. If we are interested in availability then we can enable this parameter, in which case Kafka will attempt to elect a leader that is not part of the *In-Sync-Replicas* set, which will most likely result in data loss. If we care about message safety and consistency, then we disable this parameter which prevents Kafka from doing another Leader election, unless a member of the last *In-Sync-Replicas* set becomes available again. This provides safety but reduces availability of the partition in extreme cases. By default, this parameter is disabled.

4.7 Topics

Topics are a very central Kafka abstraction. While *Partitions* constitute the basic building block, Topics act as the basic operational primitive for most clients, meaning that most clients act at the Topic level instead of the Partition level. Topics sacrifice some of the ordering guarantees of Partitions in order to allow for both Producer and Consumer total throughput scaling.

In order to discuss Topics and the whole machinery around them, we first have to specify a throughput problem and see how a simple pattern involving multiple Partitions permits us to overcome it. Topics can be directly derived as a realization of this pattern, outsourcing the hard implementation part to Kafka itself.

The problematic scenario involves two entities. A single Producer operating at maximum speed and a single Partition. The write-throughput is limited not by the Producer but by the inherit network and more importantly, *Disk throughput*. The Producer waits for acknowledgment before producing new messages, effectively throttling its production rate. In the specific scenario, we can see that the Partition write-throughput is saturated at 500 Mb/s even though the Producer can operate up to 1 GB/s.

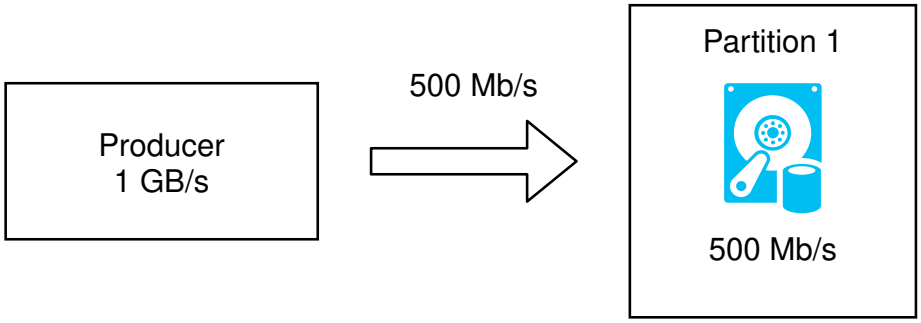


Figure 43: Throughput is limited due to storage throughput.

The main issue here is the fact that we have saturated the disk where the Partition is located. Scaling the server's disk is not sustainable, and we can't depend on optimizing the Producer further, since it is not operating at full capacity in any case. Our only way forward is through horizontal scaling.

A solution to this problem is to introduce another Partition, residing in a different Disk, and then have the Producer send half messages to the initial Partition and half to the new Partition. The new Partition can reside either on a different disk in the same Server if available, or within a different Kafka server.

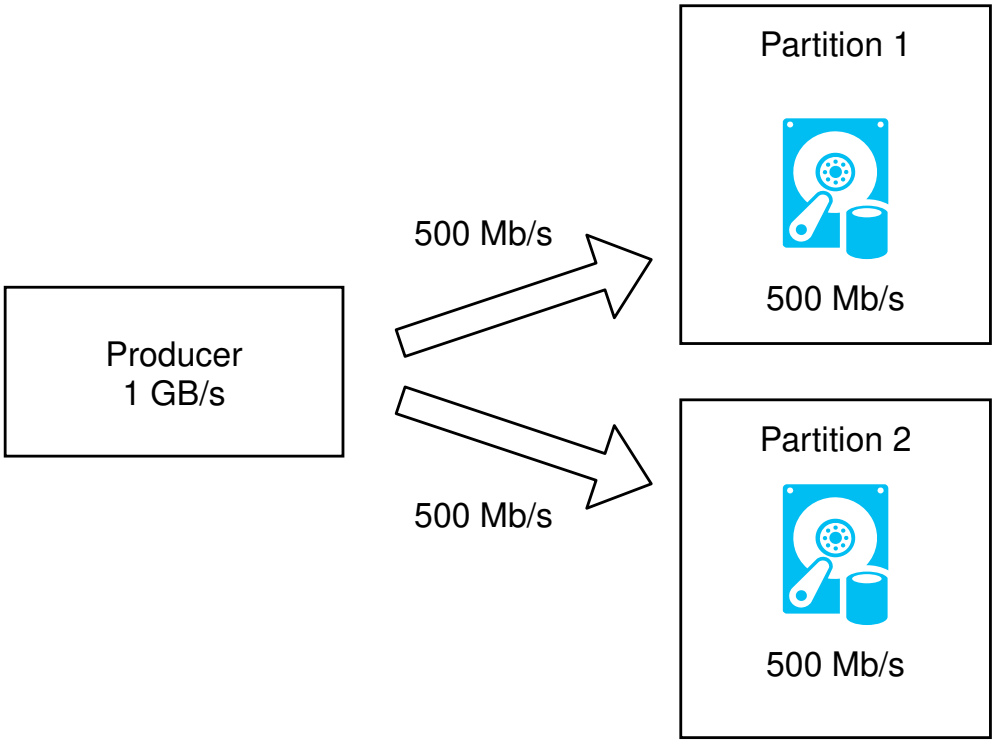


Figure 44: Throughput is maximized by splitting message stream to multiple storage.

By splitting the load to multiple disks, we can linearly scale our total write throughput. This approach of splitting/*partitioning* a single message stream to multiple partitions also gives us a great way to scale read-throughput, as we will discuss next.

Let's extend our initial problem by also including a single Consumer, fetching and processing messages from the single Partition:

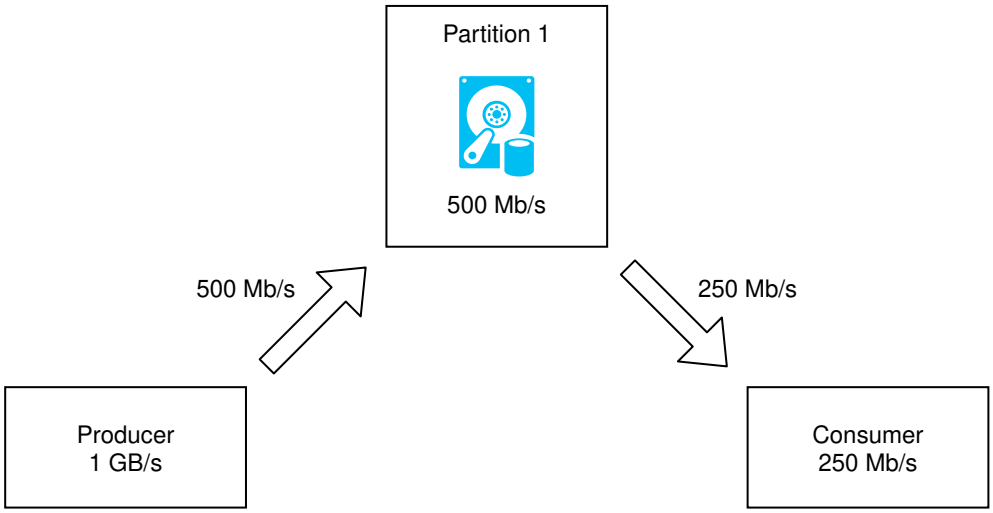


Figure 45: Message throughput limited by storage, lag is created due to slow consumer.

Due to inherent processing load, this Consumer can only achieve a processing and therefore read-throughput of 250 Mb/s. This means that the Consumer will start lagging behind the Producer, resulting in increasing lag, at the rate of 500 Mb/s - 250 Mb/s = 250 Mb/s.

After scaling to two partitions, write-throughput increases but read-throughput does not, resulting in even larger lag at the rate of 500 Mb/s

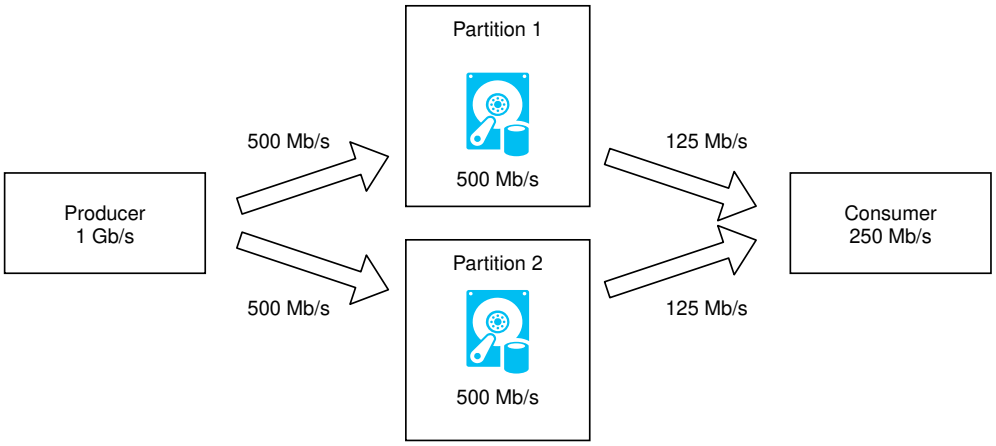


Figure 46: Write-throughput is maximized by utilizing multiple partitions. Consumer throughput is not increased due to inherent consumer processing power, lag is further increased.

In order to also scale read-throughput, we can use multiple instances of the Consumer node and assign different partitions to them. In the above example, we can increase read-throughput by a factor of 2 by introducing another Consumer node.

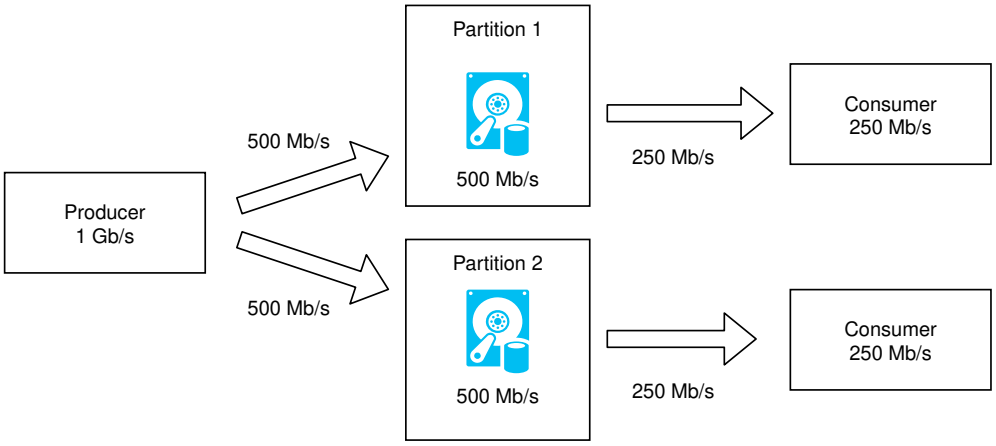


Figure 47: By assigning multiple partitions to multiple consumers, total read-throughput can be increased, lag is decreased.

The above unfortunately does not eliminate lag. In order to completely eliminate lag, we need another 500 Mb/s of read throughput. We can achieve this by using 2 more partitions as well as 2 more Consumer instances.

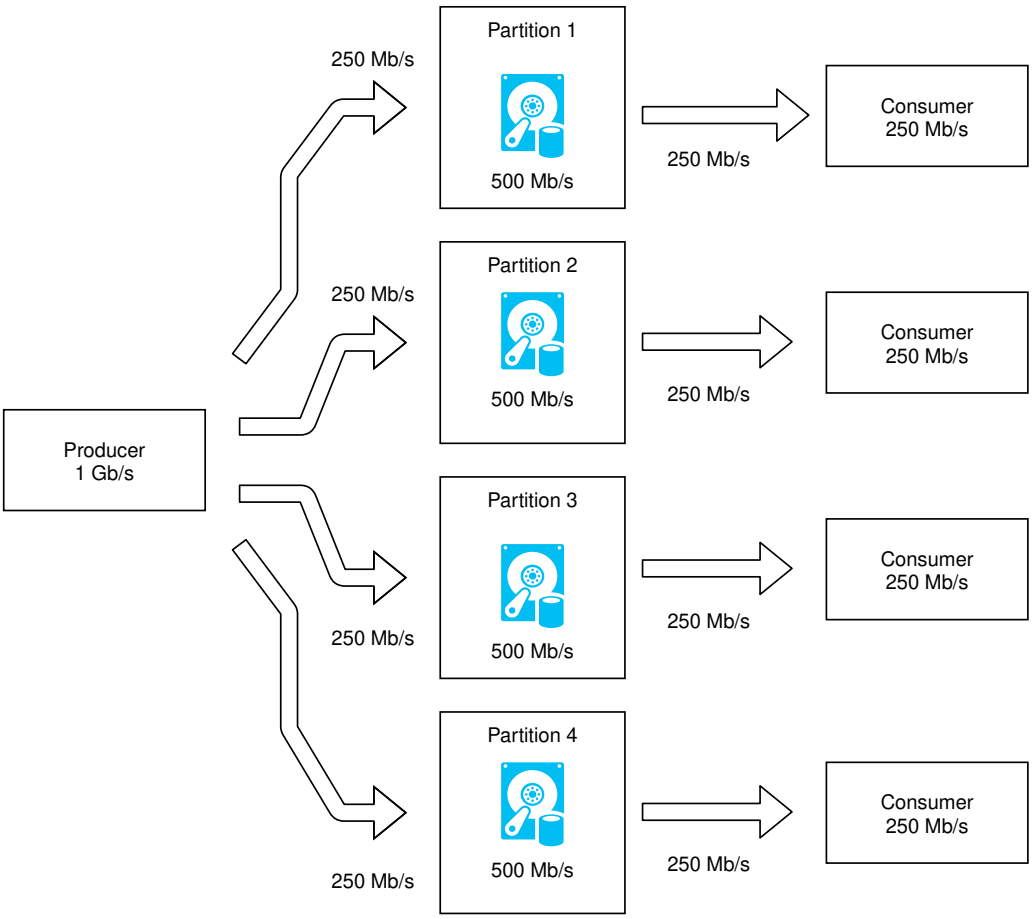


Figure 48: With enough partitions and consumers, topic throughput is maximized, read-throughput matches or exceeds write-throughput and lag is eliminated.

This pattern of using multiple Partitions to split a single message stream and assigning multiple Consumers to different Partitions, has first-class support from Kafka via the mechanism of *Topics*.

A *Topic* is a collection of Partitions that all keep messages of the same type. Producers split their message streams to these Partitions, increasing write-throughput, non-mutual subsets of these partitions can be assigned to multiple Consumers in order to increase read-throughput.

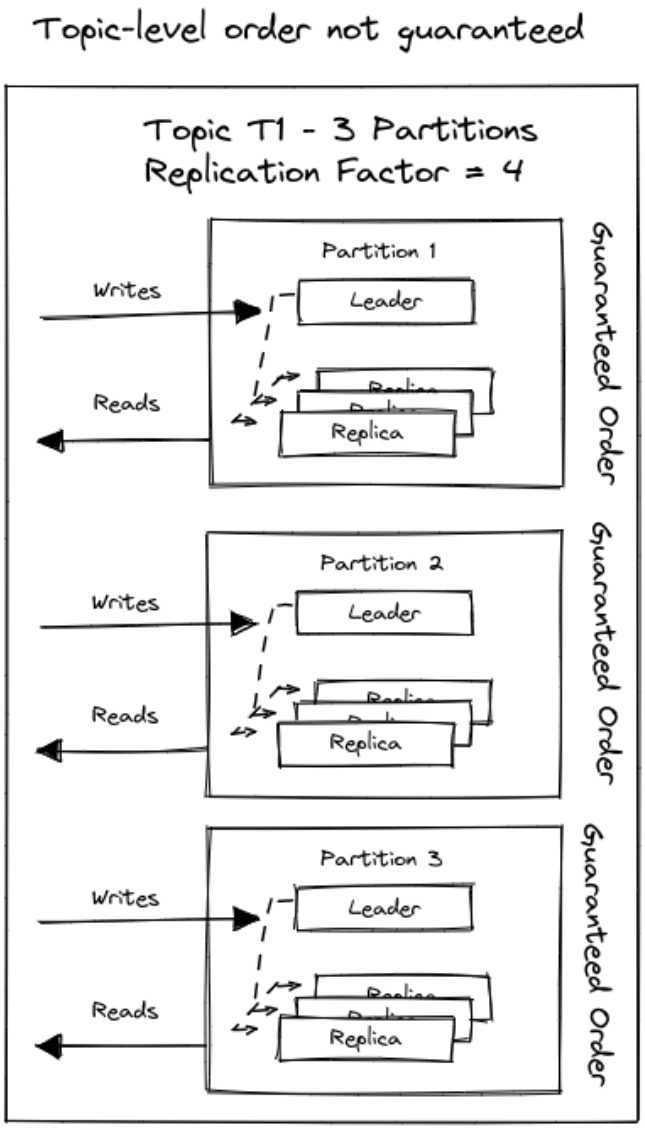


Figure 49: Topics as a collection of Partitions.

We discuss the ordering consequences of splitting a single message stream to multiple partitions in the Partitioning And Message Ordering section. The automatic assignment of topic-managed Partitions to Consumer instances is covered in Consumer Groups.

4.8 Partitioning And Message Ordering

In Topics we discussed how multiple Partitions can be used in order to increase total topic throughput. One side effect of this pattern is how it affects the total message ordering at the topic-level.

In Partitions we saw that ordering within a single Partition is naturally maintained. Consumers will retrieve messages in the same order they were stored in the underlying Partition log. As we briefly discussed, this total ordering is not maintained when consuming from multiple partitions. *Instead, messages across different partitions will be intermixed with each other, only maintaining the ordering across a single Partition.*

In this section, we take a closer look into the Producer partitioning mechanics and the message ordering guarantees that Kafka gives us at the topic level.

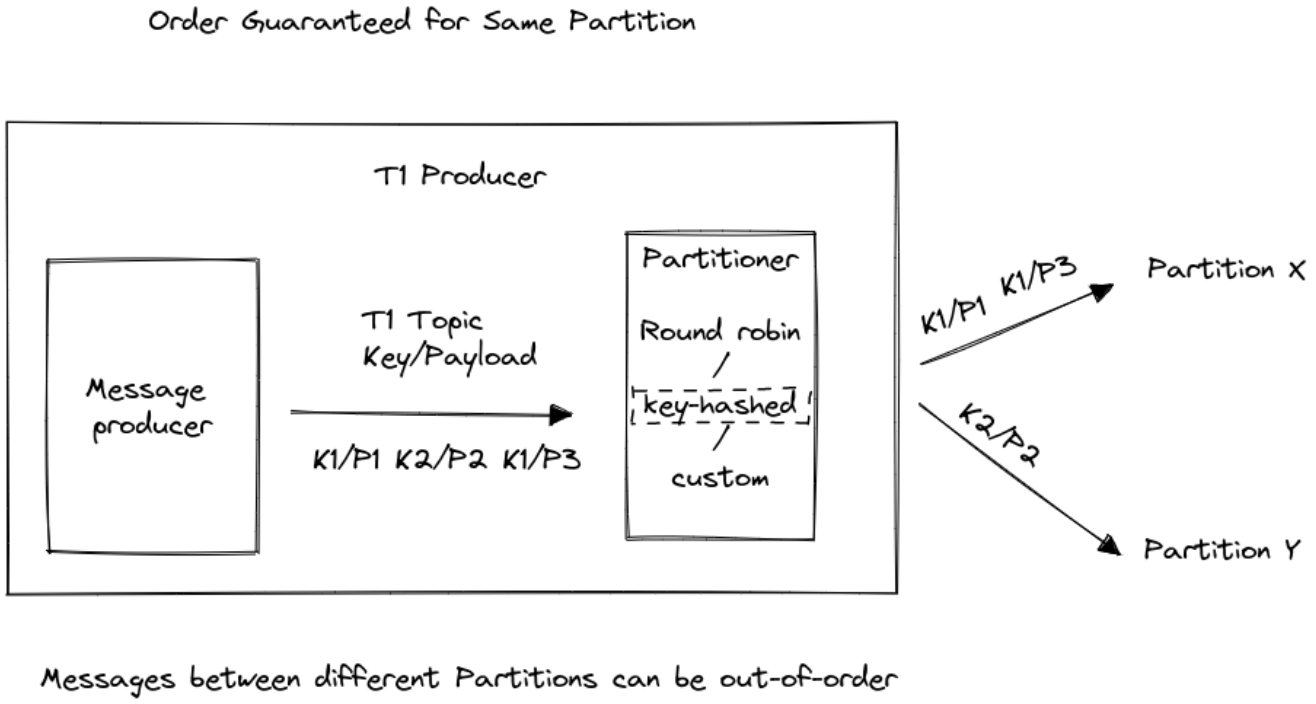


Figure 50: Operational model of Producer and Partitioner.

When Producers want to send a message to a topic, then they either chose the Partition directly or they use a *Partitioner* in order to decide on which topic-partition the message should be sent to. The default Partitioner uses the *key* part of a Kafka message in order to decide which partition the message should be sent to. The default behavior is to either assign messages in a Round-Robin fashion in case of a null key, or, hash the key and derive the partition number by using the modulo operation [17, p. 67] [16, p. 166] .

This implies that *given a single a key, changing the number of Partitions changes the concrete Partition that a message will be mapped to.* This will be important later on when we talk about Apache Kafka Modeling, Cluster and Topic Sizing.

Since, depending on the key, messages end up on different partitions, Consumers that fetch messages from these partitions in parallel will not receive the messages in the same order as they were originally produced.

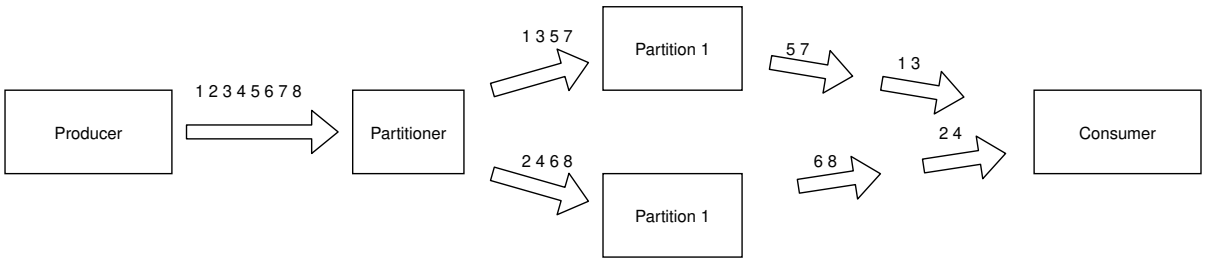


Figure 51: Messages that are sent to different partitions can be consumed out-of-order.

The only guarantee that Consumers have is that messages that ended up on the same partition will be read in order. As such, we can also conclude that Producers control message ordering by choosing the key for each message. Messages with the same key will end up on the same partition. Topics use this weaker ordering guarantee in order to be able to scale throughput.

We will use a concrete example in order to demonstrate this behavior and get intuition about how *keys* can be used to ensure message ordering.

Let's imagine that we want to create a system that provides sport event feeds. The events themselves for simplicity use the following naive schema:

```
{"tournament-id" : id, "match-id" : id, "event" : num }
```

If the producer uses the `match-id` as a *key*, then all messages with a given `match-id` will end up on the same partition. Therefore, Consumers can guarantee that all events regarding a single match will always be fetched in order.

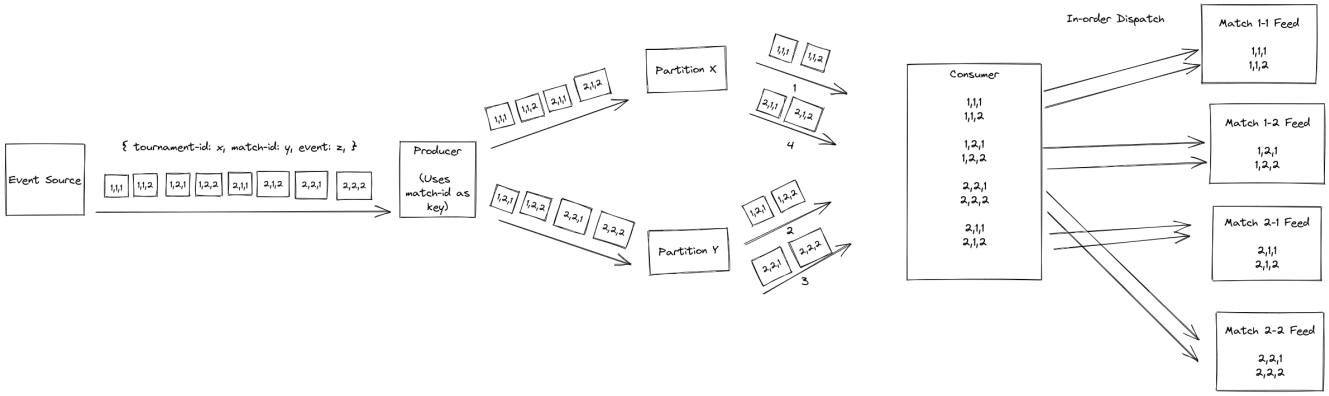


Figure 52: Using `match-id` as a key ensures proper ordering at the match level.

Let's now think what would happen if we also wanted to introduce a tournament-wide feed. Unfortunately, if we used the `match-id` as a key, then we would only guarantee ordering within a single match. Events of different matches would end up in different Partitions, and therefore, would lose their relative ordering.

In the below example, we can observe how the events of Tournament 2 arrive in the wrong order in the Tournament 2 feed. Specifically, the events of Tournament 2's match 2 arrived before those of match 1.

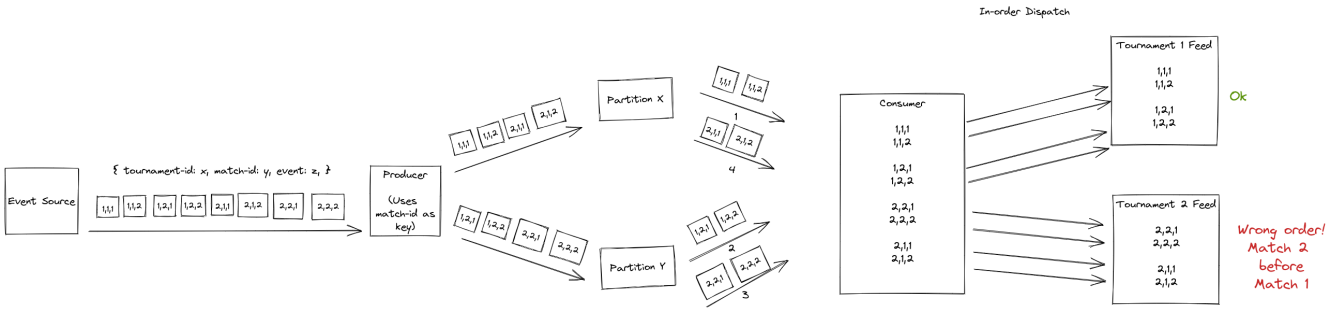


Figure 53: Using `match-id` as a key does not ensure ordering at the tournament level.

In order to tackle this problem, the Consumer could use some inherent information within the event in order to deduce the original ordering and manually reorder the messages. This, is not always applicable in practice due to missing information, but also due to performance reasons.

Another solution here is for the Producer to use the `tournament-id` field as a key instead.

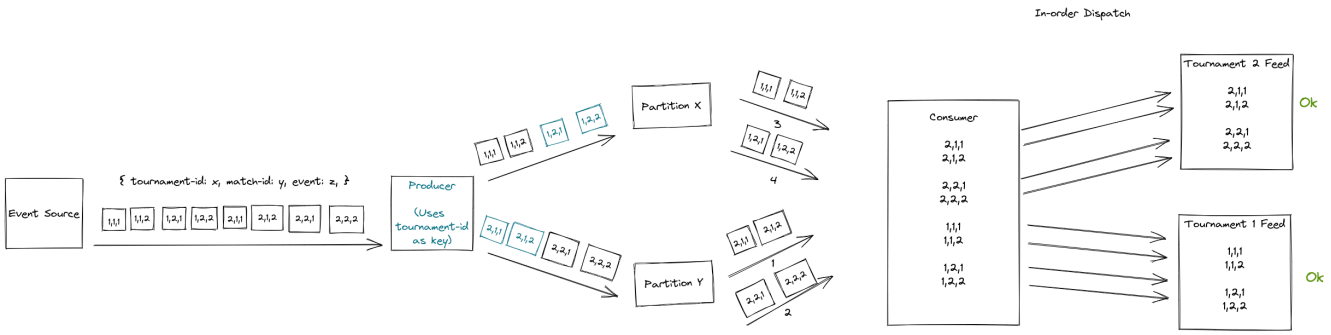


Figure 54: Using `tournament-id` as a key ensure proper ordering at both tournament and match level.

In this case, all messages regarding a tournament will end up in the same Partition, ensuring their relative ordering.

This property of message ordering according to their *key* as well as the deep importance of the *acks* parameter along with *replication factor* and *min.insync.replicas* parameters of the topic, leads us to the conclusion that the Producer should take ownership of the Topic.

At this point, we are equipped enough in order to put Topics and message ordering in practice. We will reproduce the above tournament events example using our setup.

We begin by creating two topics, *match-events* and *tournament-events*

```
docker-kafka kafka-topics.sh --bootstrap-server=localhost:9092 \
                             --create --topic match-events --partitions 6
docker-kafka kafka-topics.sh --bootstrap-server=localhost:9092 \
                             --create --topic tournament-events --partitions 6
```

We begin by creating a *producer-messages.txt* file which describes the events which we'll send.

```
{ "tournament-id": 1, "match-id": 1, "event": 1 }
{ "tournament-id": 1, "match-id": 1, "event": 2 }
{ "tournament-id": 1, "match-id": 2, "event": 1 }
{ "tournament-id": 1, "match-id": 2, "event": 2 }
{ "tournament-id": 1, "match-id": 3, "event": 1 }
{ "tournament-id": 1, "match-id": 3, "event": 2 }
{ "tournament-id": 1, "match-id": 4, "event": 1 }
{ "tournament-id": 1, "match-id": 4, "event": 2 }
{ "tournament-id": 1, "match-id": 5, "event": 1 }
{ "tournament-id": 1, "match-id": 5, "event": 2 }
{ "tournament-id": 1, "match-id": 6, "event": 1 }
{ "tournament-id": 1, "match-id": 6, "event": 2 }

{ "tournament-id": 2, "match-id": 1, "event": 1 }
{ "tournament-id": 2, "match-id": 1, "event": 2 }
{ "tournament-id": 2, "match-id": 2, "event": 1 }
{ "tournament-id": 2, "match-id": 2, "event": 2 }
{ "tournament-id": 2, "match-id": 3, "event": 1 }
{ "tournament-id": 2, "match-id": 3, "event": 2 }
{ "tournament-id": 2, "match-id": 4, "event": 1 }
{ "tournament-id": 2, "match-id": 4, "event": 2 }
{ "tournament-id": 2, "match-id": 5, "event": 1 }
{ "tournament-id": 2, "match-id": 5, "event": 2 }
{ "tournament-id": 2, "match-id": 6, "event": 1 }
{ "tournament-id": 2, "match-id": 6, "event": 2 }
```

Using the above file as a template, we create two other files, *match-keyed-producer-messages.txt* and *tournament-keyed-producer-messages.txt*

These will include the above messages along with their associated keys, which will use `=` as a separator.

For *match-keyed-producer-messages.txt* we use the `match-id` field as a key while for *tournament-keyed-producer-messages.txt* we use the `tournamnet-id` field.

We begin with *match-keyed-producer-messages.txt*:

```
1={ "tournament-id": 1, "match-id": 1, "event": 1 }
1={ "tournament-id": 1, "match-id": 1, "event": 2 }
2={ "tournament-id": 1, "match-id": 2, "event": 1 }
2={ "tournament-id": 1, "match-id": 2, "event": 2 }
3={ "tournament-id": 1, "match-id": 3, "event": 1 }
3={ "tournament-id": 1, "match-id": 3, "event": 2 }
4={ "tournament-id": 1, "match-id": 4, "event": 1 }
4={ "tournament-id": 1, "match-id": 4, "event": 2 }
5={ "tournament-id": 1, "match-id": 5, "event": 1 }
5={ "tournament-id": 1, "match-id": 5, "event": 2 }
6={ "tournament-id": 1, "match-id": 6, "event": 1 }
6={ "tournament-id": 1, "match-id": 6, "event": 2 }

1={ "tournament-id": 2, "match-id": 1, "event": 1 }
1={ "tournament-id": 2, "match-id": 1, "event": 2 }
2={ "tournament-id": 2, "match-id": 2, "event": 1 }
2={ "tournament-id": 2, "match-id": 2, "event": 2 }
3={ "tournament-id": 2, "match-id": 3, "event": 1 }
3={ "tournament-id": 2, "match-id": 3, "event": 2 }
4={ "tournament-id": 2, "match-id": 4, "event": 1 }
4={ "tournament-id": 2, "match-id": 4, "event": 2 }
5={ "tournament-id": 2, "match-id": 5, "event": 1 }
5={ "tournament-id": 2, "match-id": 5, "event": 2 }
6={ "tournament-id": 2, "match-id": 6, "event": 1 }
6={ "tournament-id": 2, "match-id": 6, "event": 2 }
```

And we send it using the Producer mode of `kcat`:

```
cat match-keyed-producer-messages.txt \
    | docker-kcat -b localhost:9092 -t match-events -K=
```

We can imagine how `match-events` consumers may continuously fetch the messages and immediately update the feeds that correspond to each match.

We use `kcat` as a consumer in order to see the order of received messages. The output is not deterministic, since it depends on which servers each partition resides in, as well as network latencies and batching of produced messages.

We use the `-f` format string flag in order to print partition and keys along with the consumed messaged.

```
docker-kcat -b localhost:9092 -C -u -t match-events \
            -f "partition: %p, key: %k - %s\n"
```

```
>>>

partition: 1, key: 2 - { "tournament-id": 1, "match-id": 2, "event": 1 }
partition: 1, key: 2 - { "tournament-id": 1, "match-id": 2, "event": 2 }
partition: 1, key: 3 - { "tournament-id": 1, "match-id": 3, "event": 1 }
partition: 1, key: 3 - { "tournament-id": 1, "match-id": 3, "event": 2 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 2, "event": 1 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 2, "event": 2 }
partition: 1, key: 3 - { "tournament-id": 2, "match-id": 3, "event": 1 }
partition: 1, key: 3 - { "tournament-id": 2, "match-id": 3, "event": 2 }
% Reached end of topic match-events [1] at offset 8
partition: 4, key: 4 - { "tournament-id": 1, "match-id": 4, "event": 1 }
partition: 4, key: 4 - { "tournament-id": 1, "match-id": 4, "event": 2 }
partition: 4, key: 5 - { "tournament-id": 1, "match-id": 5, "event": 1 }
partition: 4, key: 5 - { "tournament-id": 1, "match-id": 5, "event": 2 }
partition: 4, key: 6 - { "tournament-id": 1, "match-id": 6, "event": 1 }
partition: 4, key: 6 - { "tournament-id": 1, "match-id": 6, "event": 2 }
partition: 4, key: 4 - { "tournament-id": 2, "match-id": 4, "event": 1 }
partition: 4, key: 4 - { "tournament-id": 2, "match-id": 4, "event": 2 }
partition: 4, key: 5 - { "tournament-id": 2, "match-id": 5, "event": 1 }
partition: 4, key: 5 - { "tournament-id": 2, "match-id": 5, "event": 2 }
partition: 4, key: 6 - { "tournament-id": 2, "match-id": 6, "event": 1 }
partition: 4, key: 6 - { "tournament-id": 2, "match-id": 6, "event": 2 }
% Reached end of topic match-events [4] at offset 12
% Reached end of topic match-events [2] at offset 0
% Reached end of topic match-events [0] at offset 0
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 1, "event": 1 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 1, "event": 2 }
partition: 5, key: 1 - { "tournament-id": 2, "match-id": 1, "event": 1 }
partition: 5, key: 1 - { "tournament-id": 2, "match-id": 1, "event": 2 }
% Reached end of topic match-events [3] at offset 0
% Reached end of topic match-events [5] at offset 4
```

We see that after hashing all messages with keys 2 and 3 were sent to partition 1, those with keys 4,5,6 were sent to partition 2 and those with key 1 were sent to partition 5.

Likewise, we also observe that the total order of messages is not preserved, only the messages within each partition keep their relative ordering.

Since we used `match-id` as the key, all events of a single match will end up on the same partition, keeping the relevant ordering. Therefore, the consumers can just immediately update the relevant feeds without having ordering issues.

If a consumer wanted to provide a tournament-wide event feed, this keying scheme will not be good enough. In the above example we can observe that for both tournaments 1 and 2, events with `match-id` arrive after all other events even though they were produced first.

Another option is to use `tournament-id` as a key. We create *tournament-keyed-producer-*

messages.txt:

```
1={ "tournament-id": 1, "match-id": 1, "event": 1 }
1={ "tournament-id": 1, "match-id": 1, "event": 2 }
1={ "tournament-id": 1, "match-id": 2, "event": 1 }
1={ "tournament-id": 1, "match-id": 2, "event": 2 }
1={ "tournament-id": 1, "match-id": 3, "event": 1 }
1={ "tournament-id": 1, "match-id": 3, "event": 2 }
1={ "tournament-id": 1, "match-id": 4, "event": 1 }
1={ "tournament-id": 1, "match-id": 4, "event": 2 }
1={ "tournament-id": 1, "match-id": 5, "event": 1 }
1={ "tournament-id": 1, "match-id": 5, "event": 2 }
1={ "tournament-id": 1, "match-id": 6, "event": 1 }
1={ "tournament-id": 1, "match-id": 6, "event": 2 }

2={ "tournament-id": 2, "match-id": 1, "event": 1 }
2={ "tournament-id": 2, "match-id": 1, "event": 2 }
2={ "tournament-id": 2, "match-id": 2, "event": 1 }
2={ "tournament-id": 2, "match-id": 2, "event": 2 }
2={ "tournament-id": 2, "match-id": 3, "event": 1 }
2={ "tournament-id": 2, "match-id": 3, "event": 2 }
2={ "tournament-id": 2, "match-id": 4, "event": 1 }
2={ "tournament-id": 2, "match-id": 4, "event": 2 }
2={ "tournament-id": 2, "match-id": 5, "event": 1 }
2={ "tournament-id": 2, "match-id": 5, "event": 2 }
2={ "tournament-id": 2, "match-id": 6, "event": 1 }
2={ "tournament-id": 2, "match-id": 6, "event": 2 }
```

And send them to the *tournament-events* topic.

```
cat tournament-keyed-producer-messages.txt \
    | docker-kcat -b localhost:9092 -t tournament-events -K=
```

Then we read the messages.

```
docker-kcat -b localhost:9092 -C -u -t tournament-events \
            -f "partition: %p, key: %k - %s\n"
```

```
>>>

% Auto-selecting Consumer mode (use -P or -C to override)
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 1, "event": 1 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 1, "event": 2 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 2, "event": 1 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 2, "event": 2 }
```

```
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 3, "event": 1 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 3, "event": 2 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 4, "event": 1 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 4, "event": 2 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 5, "event": 1 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 5, "event": 2 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 6, "event": 1 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 6, "event": 2 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 1, "event": 1 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 1, "event": 2 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 2, "event": 1 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 2, "event": 2 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 3, "event": 1 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 3, "event": 2 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 4, "event": 1 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 4, "event": 2 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 5, "event": 1 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 5, "event": 2 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 6, "event": 1 }
partition: 1, key: 2 - { "tournament-id": 2, "match-id": 6, "event": 2 }
% Reached end of topic tournament-events [2] at offset 0
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 1, "event": 1 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 1, "event": 2 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 2, "event": 1 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 2, "event": 2 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 3, "event": 1 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 3, "event": 2 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 4, "event": 1 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 4, "event": 2 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 5, "event": 1 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 5, "event": 2 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 6, "event": 1 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 6, "event": 2 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 1, "event": 1 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 1, "event": 2 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 2, "event": 1 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 2, "event": 2 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 3, "event": 1 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 3, "event": 2 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 4, "event": 1 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 4, "event": 2 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 5, "event": 1 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 5, "event": 2 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 6, "event": 1 }
partition: 5, key: 1 - { "tournament-id": 1, "match-id": 6, "event": 2 }
% Reached end of topic tournament-events [0] at offset 0
```

```
% Reached end of topic tournament-events [1] at offset 24
% Reached end of topic tournament-events [4] at offset 0
% Reached end of topic tournament-events [5] at offset 24
% Reached end of topic tournament-events [3] at offset 0
```

Here we can see that due to hashing, all messages with key `tournament-id=1` were sent to partition 5 while all messages with key `tournament-id=2` were sent to partition 1.

We observe again that total event ordering is not maintained. Events of tournament 2 arrived before events of tournament 1 even though they were produced in the reverse order.

Because we used `tournament-id` as a key, all events of a given tournament will end up on the same partition, keeping their relative ordering. This scheme would support both the single match and tournament-wide feeds.

Finally, we note the different approaches that we can use to observe the above messages.

First, we can use the native Kafka tool, `kafka-console-consumer.sh`

```
docker-kafka kafka-console-consumer.sh --bootstrap-server=localhost:9092 \
             --topic tournament-events --from-beginning
docker-kafka kafka-console-consumer.sh --bootstrap-server=localhost:9092 \
                  --topic match-events --from-beginning
```

Or by using Kowl to query the messages

Finally, we can also directly use `kafka-dump-logs.sh` to directly read the binary log of the given partitions directly.

We will use `match-events` Partition 1 as an example.

We use `kcat`'s query mode to find the leader of Partition 1.

```
docker-kcat -b localhost:9092 -L -u -t match-events
```

```
>>>

Metadata for match-events (from broker 0: localhost:9092/0):
 3 brokers:
  broker 0 at localhost:9092
  broker 2 at localhost:9094 (controller)
  broker 1 at localhost:9093
 1 topics:
  topic "match-events" with 6 partitions:
    partition 0, leader 2, replicas: 2, 0, 1, isrs: 2, 0, 1
    partition 1, leader 1, replicas: 1, 2, 0, isrs: 1, 2, 0
    partition 2, leader 0, replicas: 0, 1, 2, isrs: 0, 1, 2
    partition 3, leader 2, replicas: 2, 1, 0, isrs: 2, 1, 0
```

```
    partition 4, leader 1, replicas: 1, 0, 2, isrs: 1, 0, 2
    partition 5, leader 0, replicas: 0, 2, 1, isrs: 0, 2, 1
```

We can see that the Leader of Partition 1 is the Kafka Server with `id=1`. Looking at our Setup this corresponds to docker instance `kafka-1`.

We connect to it using

```
docker compose exec -it kafka-1 /bin/bash
```

We then `cd` to the `/tmp/kafka-logs/match-events-1` directory.

```
cd /tmp/kafka-logs/match-events-1
ls
```

```
>>>
00000000000000000000.index  00000000000000000000.timeindex  partition.metadata
00000000000000000000.log    leader-epoch-checkpoint
```

And then use `kafka-dump-log.sh` in order to inspect the partition.

```
kafka-dump-log.sh --print-data-log --skip-record-metadata \
                  --files 00000000000000000000.log
```

```
>>>

Dumping 00000000000000000000.log
Starting offset: 0
baseOffset: 0 lastOffset: 7 count: 8 baseSequence: -1 lastSequence: -1
producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false
isControl: false position: 0 CreateTime: 1651419238773 size: 517 magic: 2
compresscodec: none crc: 3756381328 isvalid: true
| key: 2 payload: { "tournament-id": 1, "match-id": 2, "event": 1 }
| key: 2 payload: { "tournament-id": 1, "match-id": 2, "event": 2 }
| key: 3 payload: { "tournament-id": 1, "match-id": 3, "event": 1 }
| key: 3 payload: { "tournament-id": 1, "match-id": 3, "event": 2 }
| key: 2 payload: { "tournament-id": 2, "match-id": 2, "event": 1 }
| key: 2 payload: { "tournament-id": 2, "match-id": 2, "event": 2 }
| key: 3 payload: { "tournament-id": 2, "match-id": 3, "event": 1 }
| key: 3 payload: { "tournament-id": 2, "match-id": 3, "event": 2 }
```

Before we conclude this section, we deem important to discuss a small edge case regarding the ordering of Producer messages sent synchronously to a single Partition.

When producing messages, Kafka internally queues the messages and sends them in batches for performance reasons. [33].

In order to increase throughput, Kafka Producers may send multiple batches back-to-back before waiting for their individual acknowledgment.

The number of batches that Kafka may wait for acknowledgment in parallel, is controlled by the max.in.flight.requests.per.connection configuration parameter.

If the parameter is larger than 1 (the default is 5), this can lead to the following scenario in the case of **synchronous** acknowledgment.

- The producer sends two batches, batch 1 and batch 2.
- Batch 1 for some reason fails to be delivered, batch 2 is delivered successfully.
- Batch 1 is automatically retried, resulting in batch 1 being delivered to the server after batch 2.
- Batch 1 will end up being appended to the Partition after Batch 2, resulting in the wrong message ordering.

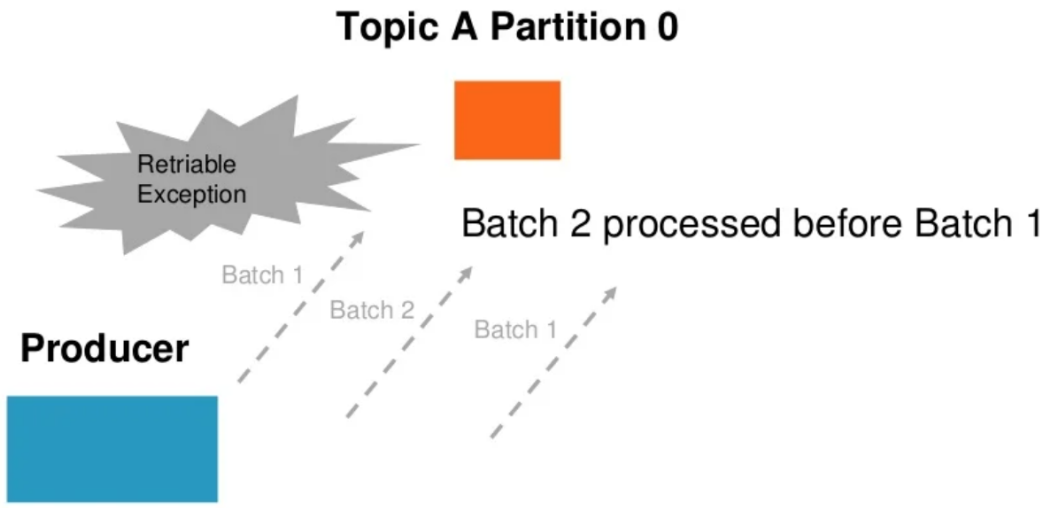


Figure 55: Parallel in-flight requests may result in incorrect message ordering.

There are two solutions for the above issue, either limit *max.in.flight.requests.per.connection* to 1, or enable Kafka's **idempotence semantics**. The latter is the preferred approach. We will discuss more about idempotency in a later section.

Concluding this section, Topics allows us to take advantage of multiple partitions in order to increase throughput horizontally. In order to do so, we have to sacrifice some of our total ordering guarantees. Keys can be used in order to specify which messages should end up on the same Partition, ensuring that messages with the same key, will always be consumed in the correct order among each other. As a result, choosing the correct key-scheme is very important, since it defines the ordering guarantees of the whole topic. Moreover, changing the key-scheme is not recommended, since already produced messages will not be converted to the new key-scheme, leading to message ordering issues. This makes selecting the proper key-scheme essential to get right from the start.

4.9 Kafka Consumer Groups

In the Topics section, we introduced a throughput problem and we saw how using multiple Partitions can help increase throughput.

The final solution we ended up with had the following topology:

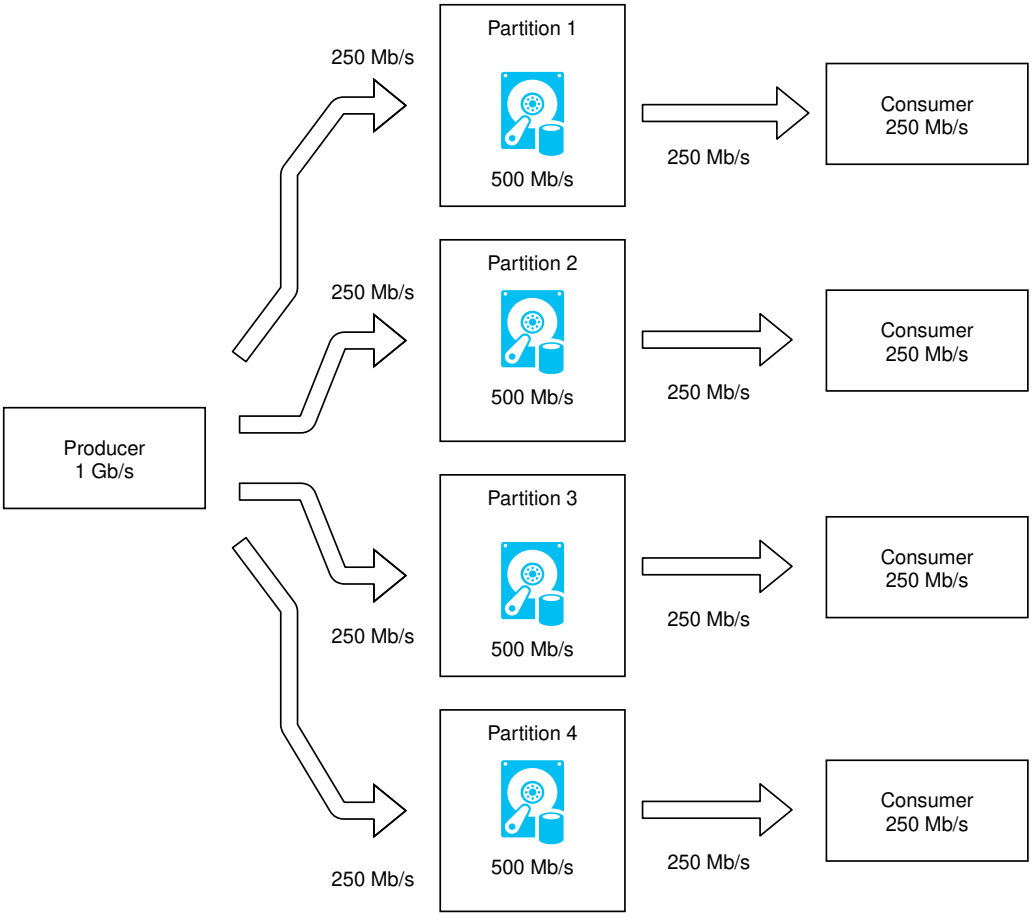


Figure 56: Throughput properly scaled using 4 partitions.

In Partitioning And Message Ordering we looked at how Producers can produce messages at the Topic level and discussed the ordering guarantees when Consuming messages. In this section we will concentrate on the Consumers part of the above topology and specifically the Partition assignment.

In theory, we could create Consumers and assign them specific Partitions manually. While possible, this approach is very error prone and requires a lot of machinery in order to work correctly. Let's discuss some of the design issues in this domain.

First of all, like we will see at Apache Kafka Modeling, Cluster and Topic Sizing, due to ordering restrictions it is best to not change the initial amounts of Partitions within a Topic. Because having a lot of Partitions is not very problematic until we get to larger numbers, it is often a good idea to over-commit when creating Partitions with topics ending up with tens or even hundreds of Partitions. In this case we rarely need to introduce as many Consumers as Partitions in order to provide the necessary read-throughput. Most pipelines use just a few Consumers that are assigned

multiple Partitions and then more are added as read-throughput requirements are increased, rarely reaching capacity.

As a result, when more Consumers are added, Partitions have to be re-assigned to all Consumers. This means that we have to somehow keep track of the progress across all the assigned partitions for each Consumer, then after a re-assignment occurs all Consumers read the previous progress of their newly assigned Partitions and resume from that Point. Thus we have to introduce some sort of global state were all Consumers can keep track and read each others' progress.

The same should happen when removing Consumers. This includes error cases where some Consumer is stalled or unavailable. It would be a good idea to detect such cases, and automatically revoke the assigned Partitions from these Consumers, evenly re-assigning them to the remaining Consumers.

All of the above functionality is provided by Kafka via the concept of Consumer Groups [17, p. 103] [16, p. 31] .

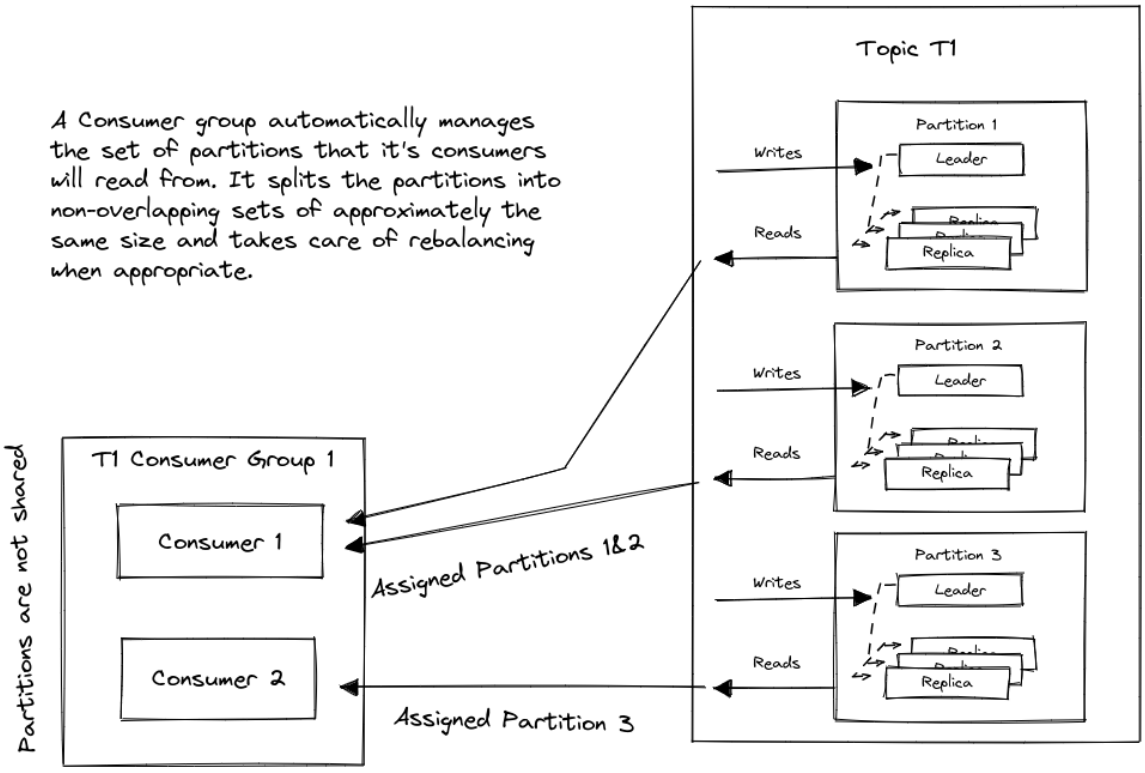


Figure 57: Consumers that are part of a consumer group are exclusively assigned partitions.

A Consumer Group is a Kafka-managed group of Consumers. Kafka keeps track of every Consumer in the group and administrates the automatic assignment of Partitions to them, properly taking care of Consumer additions/removals including automatic removals due to Consumers stalling or

becoming unavailable. Each Partition and therefore each of their messages is only assigned to a single Consumer in the whole group.

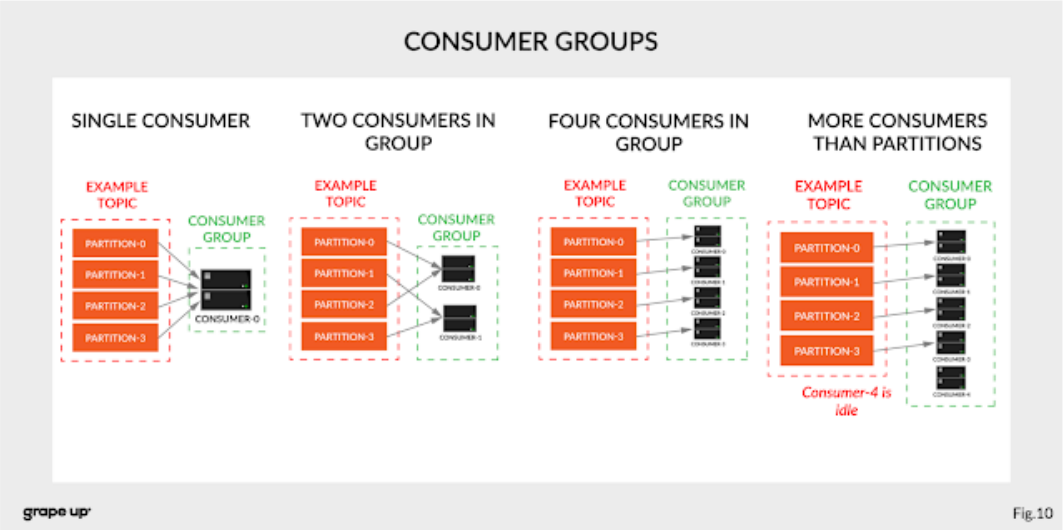


Figure 58: Showcase of consumer group assignments at different group sizes [48].

This means that the maximum amount of Consumers within a Consumer group is equal to the number of Partitions of the given Topic. In this case any excess Consumers will not be assigned any partitions while all the others will be assigned a single Partition. This case is usually rare since Topics should be over-partitioned at creation.

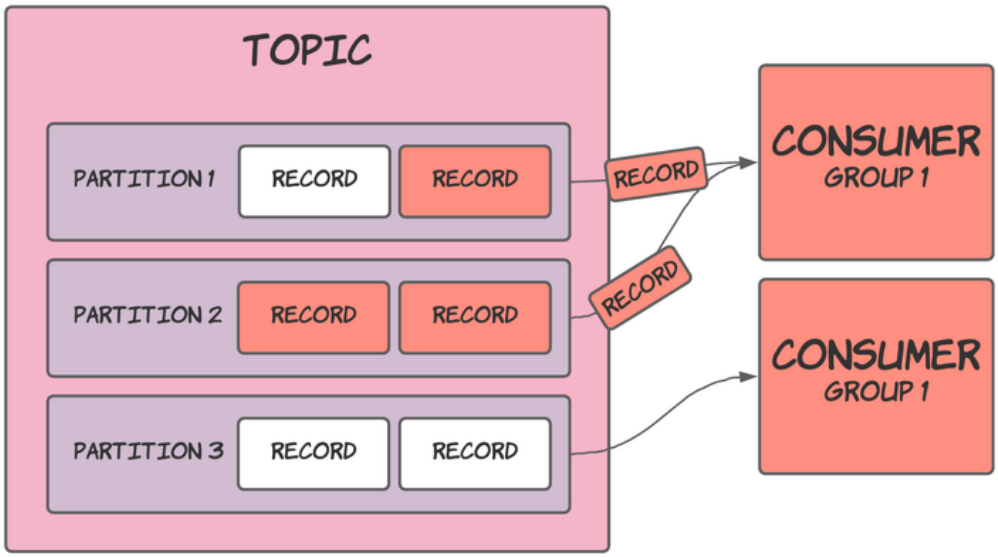


Figure 59: Partitions being assigned to single Consumer Group of two Consumers [49].

Different consumer groups act independently of each other.

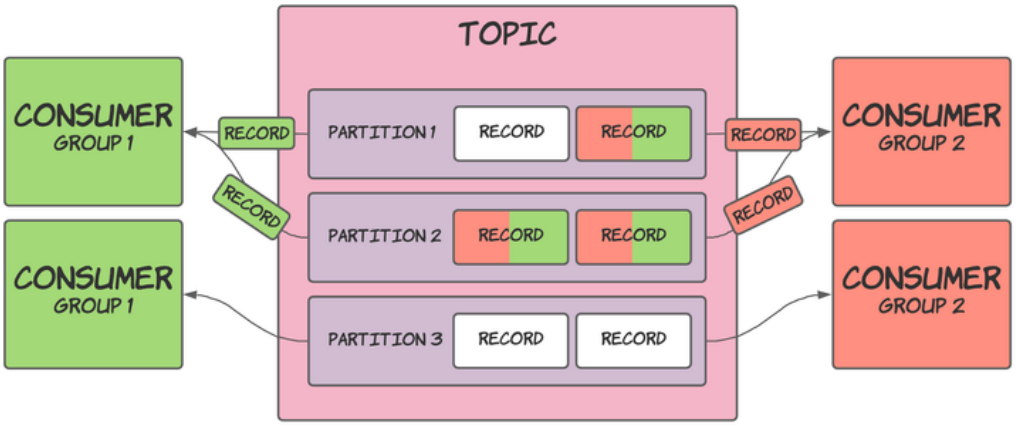


Figure 60: Partitions being assigned to multiple Consumer Groups [49].

When a Consumer is either added or removed from the Consumer Group (due to unavailability, stalling or more often manually), then Kafka re-assigns the unavailable Consumer's Partitions to the other Consumers that are part of the group. This is done in order to not stall these Partitions' processing and evenly redistribute the load.

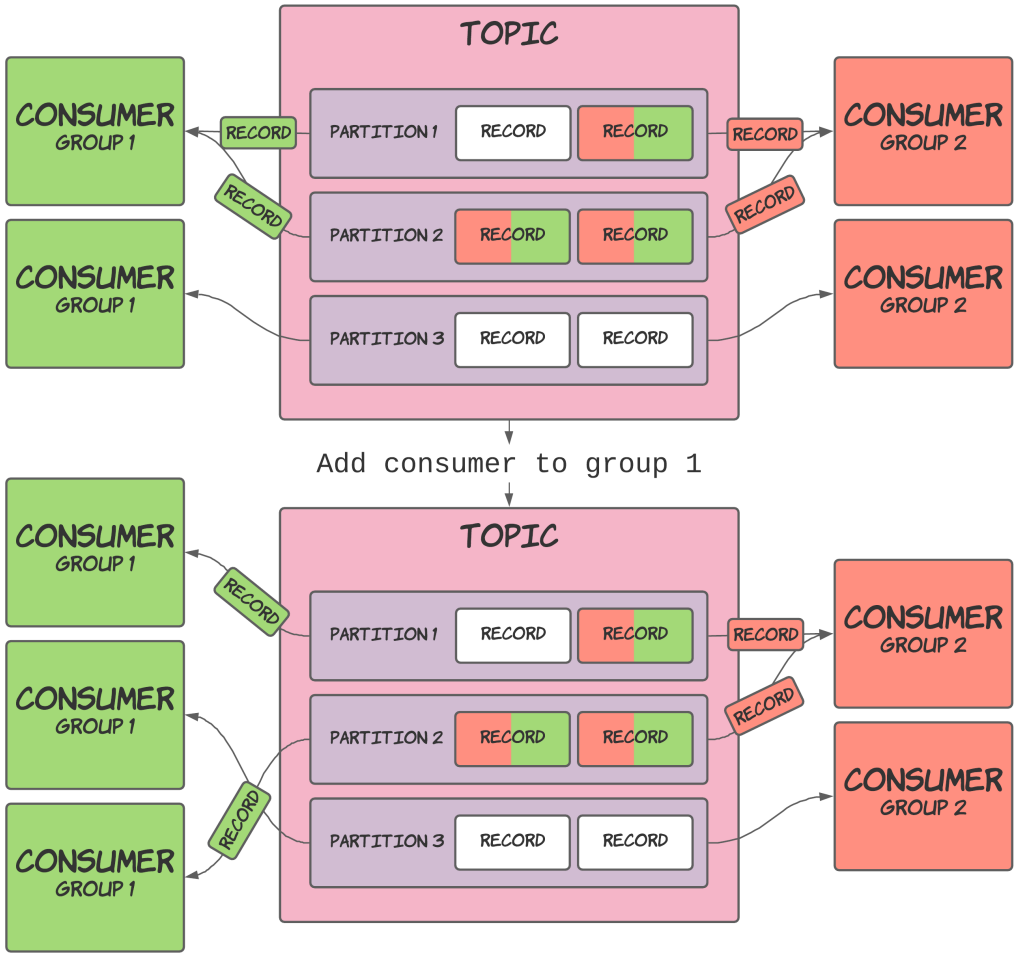


Figure 61: Assignment changes after a Consumer is added to a Consumer Group [49].

This procedure is called Rebalancing and requires complex synchronization between the different consumers. More specifically, since Partitions can be revoked from a Consumer and assigned to another, some form of progress tracking is required in order for the new Consumer to know where to resume the processing, without reprocessing or skipping messages. This implicitly requires some form of shared state between the Consumers.

Kafka achieves this by using an offset commitment mechanism from the Consumers' side.

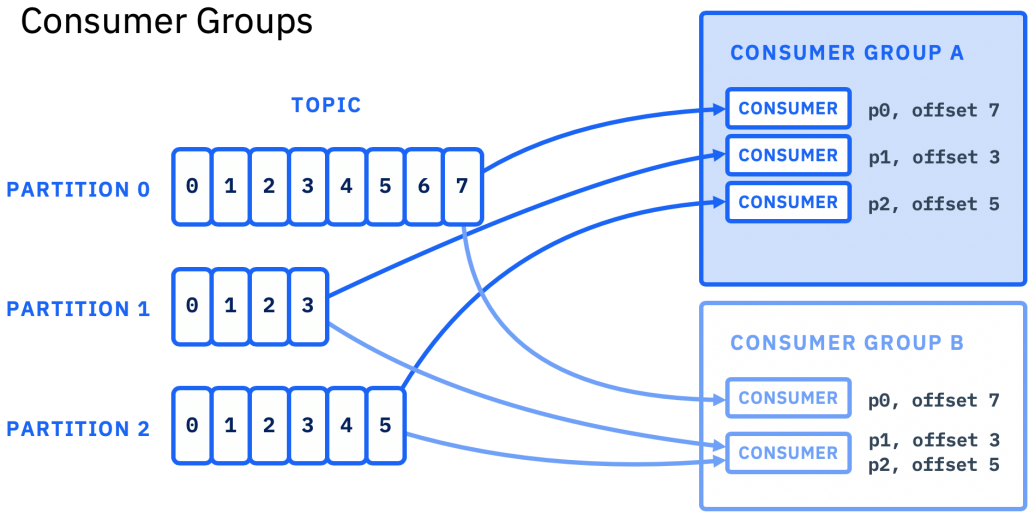


Figure 62: Committed partition offsets of each consumer group are tracked independently [50].

Each time a consumer processes a message of a partition, it commits the message's offset to Kafka. Kafka uses an internal topic in order to provide the shared state.

When a partition is newly assigned to a Consumer, then the Consumer reads the last committed offset that corresponds to the Partition, and resumes fetching and processing the messages from that offset onward.

This mechanism can also be useful in the case of a single Consumer. In that case, the Consumer does not have to use local state in order to keep track of it's processing and can instead piggyback on the Consumer Group's offset commitment functionality. This way, it can connect and disconnect without losing track of the processing progress.

At this point, we can connect to our cluster in order to build intuiting. We begin by creating a topic with name test and 10 partitions.

```
docker-kafka kafka-topics.sh --bootstrap-server localhost:9092 \
                             --create --topic test --partitions 10
```

At this point we can also use a separate native Kafka utility, `kafka-verifiable-producer.sh`, that allows us to produce messages to a given topic with a specific message throughput.

```
docker-kafka kafka-verifiable-producer.sh --bootstrap-server localhost:9092 \
                                          --topic test --throughput 3
```

This specific producer will send 3 messages each second to the topic test, randomly choosing the partition that each message will be sent to. Since the topic has 10 Partitions, then the per-Partition throughput will be 0.3 msg/s.

For this example, we have manually implemented a `rate-limited-consumer` client using the *librdkafka* bindings for `Node.js`. This consumer explicitly consumes messages at a specified rate.

The code is provided in the accompanying code repository.

```
cd simple-consumers/node/rate-limited-consumer
BUILD_LIBRDKAFKA=0 npm install
npx ts-node src/main.ts --help
```

We can imagine that the emulated consumption rate is actually due to the inherent processing load of the consumer.

```
npx ts-node src/main.ts -b localhost:9092 -t test -g test-group -r 1
```

With the above snippet, we create a rate limited consumer, we add it to the new consumer group with name `test-group` and we specify a total read-throughput of 1 msg/s from the topic test.

As a result, it won't be able to meet the total topic throughput requirements, and it will start to lag behind the topic.

At this point, the consumer is the sole member of the consumer group and will therefore be assigned all of its Partitions.

We can observe this by using the native Kafka utility `kafka-consumer-groups.sh`

We can list the different groups available with

```
docker-kafka kafka-consumer-groups.sh --bootstrap-server localhost:9092 --list
```

```
>>>

test-group
```

And we can observe details about the different groups by using the `--describe` flag,

```
docker-kafka kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
                                      --describe --group test-group
```

```
>>>

GROUP       TOPIC  PARTITION  OFFSET  END  LAG  CONSUMER-ID HOST
test-group  test   6          71      83   12   4bdd..7cdc  /172.22.0.1
test-group  test   0          75      93   18   4bdd..7cdc  /172.22.0.1
test-group  test   7          60      73   13   4bdd..7cdc  /172.22.0.1
test-group  test   5          68      85   17   4bdd..7cdc  /172.22.0.1
test-group  test   8          72      87   15   4bdd..7cdc  /172.22.0.1
test-group  test   1          53      62   9    4bdd..7cdc  /172.22.0.1
test-group  test   4          64      80   16   4bdd..7cdc  /172.22.0.1
```

```
test-group  test   9          58      67   9    4bdd..7cdc  /172.22.0.1
test-group  test   3          72      88   16   4bdd..7cdc  /172.22.0.1
test-group  test   2          65      80   15   4bdd..7cdc  /172.22.0.1
```

Another alternative is to use Kowl

The above output specifies which consumer instance each partition is assigned to along with the `log-end-offset`, basically the latest available offset of the partition, as well as the `current-offset` which is the latest Consumer committed offset. It also directly shows us the Partition Lag, which is derived by subtracting the two values.

Since the single consumer has been assigned all 10 partitions, and since it has a total read-throughput of 1 msg/s then we can expect a per-Partition read-throughput of 0.1 msg/s. The per-Partition write-throughput is 0.3 and therefore we expect that the Lag will keep increasing.

By running the above command a few times, we see that Lag keeps increasing like we predicted.

We will now insert another consumer instance to the same consumer group.

```
npx ts-node src/main.ts -b localhost:9092 -t test -g test-group -r 1
```

Since the consumer group now has 2 members, these will share the partitions, ending up with 5 partitions per consumer.

```
docker-kafka kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
                                      --describe --group test-group
```

```
>>>

GROUP       TOPIC  PARTITION  OFFSET  END  LAG  CONSUMER-ID HOST
test-group  test   0          152     324  172  4bdd..7cdc  /172.22.0.1
test-group  test   1          123     295  172  4bdd..7cdc  /172.22.0.1
test-group  test   4          138     302  164  4bdd..7cdc  /172.22.0.1
test-group  test   3          149     310  161  4bdd..7cdc  /172.22.0.1
test-group  test   2          155     325  170  4bdd..7cdc  /172.22.0.1
test-group  test   6          161     328  167  4e35..ffac  /172.22.0.1
test-group  test   7          136     305  169  4e35..ffac  /172.22.0.1
test-group  test   5          139     292  153  4e35..ffac  /172.22.0.1
test-group  test   8          162     335  173  4e35..ffac  /172.22.0.1
test-group  test   9          119     284  165  4e35..ffac  /172.22.0.1
```

Since the total throughput of each consumer is 1 msg/s and since each consumer is currently assigned 5 partitions, we can deduce that the per-Partition read-throughput of each consumer will be 0.2 msg/s.

This is better but still not enough to sustain the current per-Partition write-throughput of 0.3 msg/s.

We can fix this by adding another 2 consumers to the consumer group. This will bring us to a per-Partition read-throughput of 0.4 msg/s which can sustain the per-Partition write-throughput.

We can do this by running the following command twice.

```
npx ts-node src/main.ts -b localhost:9092 -t test -g test-group -r 1
```

We can observe the decreasing lag either by running `kafka-consumer-groups.sh`, or by looking at the Total Lag in Kowl

At this point we have a complete picture of *Kafka's architectural model*

4.10 Processing Guarantees

As we have seen in previous sections, produced messages can be lost depending on multiple factors such as whether the Producer looks at acknowledgments at all, the `acks` configuration variable, the replication factor and the in-sync replicas and whether the Producer keeps track of produced messages in the case of unexpected failures. This may result in consumers not processing certain messages. Moreover, even if we ensure that messages are not lost, retries at the producer side may result in duplicate messages being published. Finally, processing consumers may unexpectedly fail before committing their offsets, resulting in potential duplicate processing.

In this section, we will discuss the different processing guarantees of Kafka, looking at the issue holistically, following the complete chain of publication, consumption, and processing of messages.

At-Most Once Processing

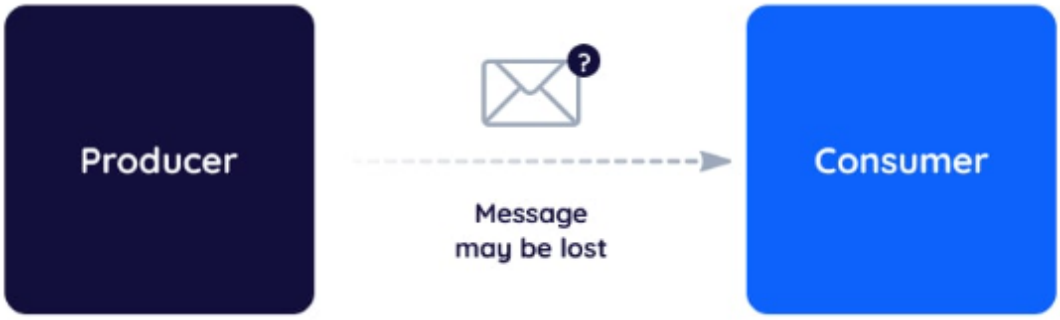


Figure 63: Messages may be lost [51].

There are multiple ways for a message to be lost or potentially simply not processed.

From the Producer side, any Kafka configuration not following the conditions for message safety, may result in lost messages under certain situations.

Another potential source of lost messages, is the Producer not properly keeping track of acknowledged produced messages. This may occur if the Producer persistently stores its progress before messages are acknowledged. In this case and under unexpected failure scenarios, after a message is marked as sent, but before it is actually sent or acknowledged, the Producer will skip the message, resulting in it not being stored in the Cluster.

Not performing retries after acknowledgment time-outs will also result in lost messages.

From the Consumer side, committing offsets before the processing of messages may result in skipping the processing of messages. Specifically, unexpected partition rebalancing or rebooting after unexpected failure, will result in messages being skipped.

Properly configuring the Cluster's size, replication factor, minimum in-sync replicas and Producer acknowledgment behavior, while also properly tracking produced messages and committing after processing in the Consumer side, will result in messages being safely stored in the Cluster and processed at-least once.

At-Least-Once Processing

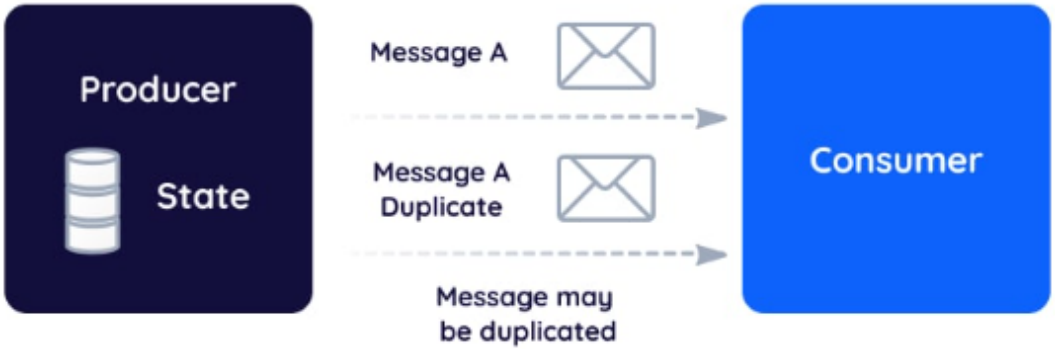


Figure 64: Messages can be duplicated, especially in node recovery situations [51].

A properly configured Kafka Cluster with accordingly configured Producers, following the conditions for maximum message safety, can guarantee that all messages will be successfully and safely stored.

Unfortunately, this can often result in duplicate messages being published.

As an example, we can imagine a Producer publishing a single message and subsequently waiting for an acknowledgment. The Cluster stores the message, safely replicates it, and then sends an acknowledgment back to the Producer. Unfortunately, either due to the Producer crashing or perhaps due to some transient network error, the acknowledgment never makes it to the Producer. As a result, the Producer will retry sending the message resulting in the message being stored twice in the server.

From the Consumer's side, lack of atomicity between message processing and message commitment means that a message may be consumed, but the Consumer crashes before the offset is finished being committed. Any Consumer that picks up processing for that partition, will subsequently re-process the message. We will revisit this concept of atomicity a bit later when discussing **Transactions**.

Idempotence

Processing a message multiple times is not always an issue. This largely depends on what are the side effects of processing a message. If the message models a concrete entity's state and the side effect means updating the corresponding row of a database, then performing the same operation multiple times in succession doesn't matter. The row will end up with the same state in any case. If the message instead models an incremental update to a given entity's state, then processing the same message multiple times will result in the wrong state for the given row.

Consumers that can process the messages multiple times without problems are called **idempotent**. Some consumers are naturally idempotent, while others are not.

It is often possible to make non-idempotent Consumers idempotent by including a unique identifier with each message, effectively giving them an identity. The Consumer can then keep a persistent record of consumed messages, and explicitly avoid reprocessing messages that use the same identifier.

Unfortunately, there is a fundamental issue with this approach. It is possible for a consumer to fail before persisting the message identifier. After resuming operations and receiving a message with

the same identifier, the Consumer will end up processing the message and therefore applying the side effect multiple times.

In order for a Consumer with side effects to effectively make use of this pattern, it requires atomicity between processing a message and persisting its identifier. This fundamentally requires transaction support across the persistency system and the systems involved in the processing side effects.

A trivial application of this, is when both processing and persistency occur within the bounds of a single system with transaction support. An example is when the processing side effect means performing a database operation while the message identifiers are stored in the same database.

Another applicable case of this pattern is when both systems support some common protocol of distributed transactions.

In general, this restriction severely limits the applicability of this pattern and is fundamental to messaging across different services.

Idempotent Producer

Kafka has gradually added support for idempotent message production. This means that even if message publications are retried, they will be stored to the server exactly once. This way, message batches that are sent multiple times due to retries will not be duplicated. The feature is only effective during a single Producer session and does not carry over to Producer restarts. We will discuss how Kafka supports idempotence across different sessions in the **Transactions** section.

Idempotence is enabled using Producer's enable.idempotence configuration parameter.

When a Producer initially connects to a server, it will be assigned a unique Producer Identifier (PID). After a Producer is assigned a PID, it will also initialize a monotonically increasing Sequencer Number, which will be included in all subsequent message batch publications.

Servers keep an internal mapping of the latest sequence numbers sent for each PID per partition. They can then use this mapping in order to avoid appending duplicate message batches.

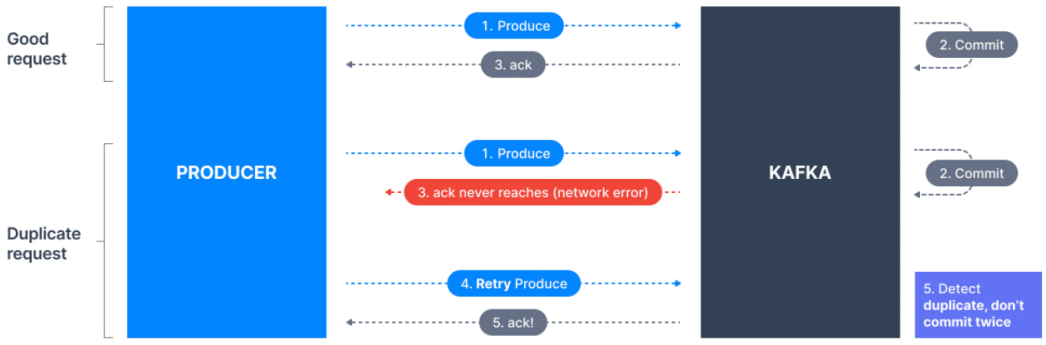


Figure 65: Servers can avoid re-appending messages by keeping track of sequence numbers [52].

Moreover, since the specific sequence numbers are maintained, servers can re-order incoming batches according to their included Sequence Numbers. This avoids the issue specified in Single Partition Ordering and In-flight batches permitting multiple batches to be sent per connection without affecting the message ordering. This behavior is only guaranteed as long as max.in.flight.requests.per.connection is less than or equal to 5.

Transactions

Kafka has a very useful feature that permits Producers to transactionally write messages across different partitions. This feature compliments the previously discussed Idempotent Producer behavior by ensuring that idempotence behavior is maintained across Producer sessions. In this section, we have a brief look into how this feature works. In the next section, we will also discuss how this feature permits effectively-once processing for pure data transformation Kafka-only pipelines.

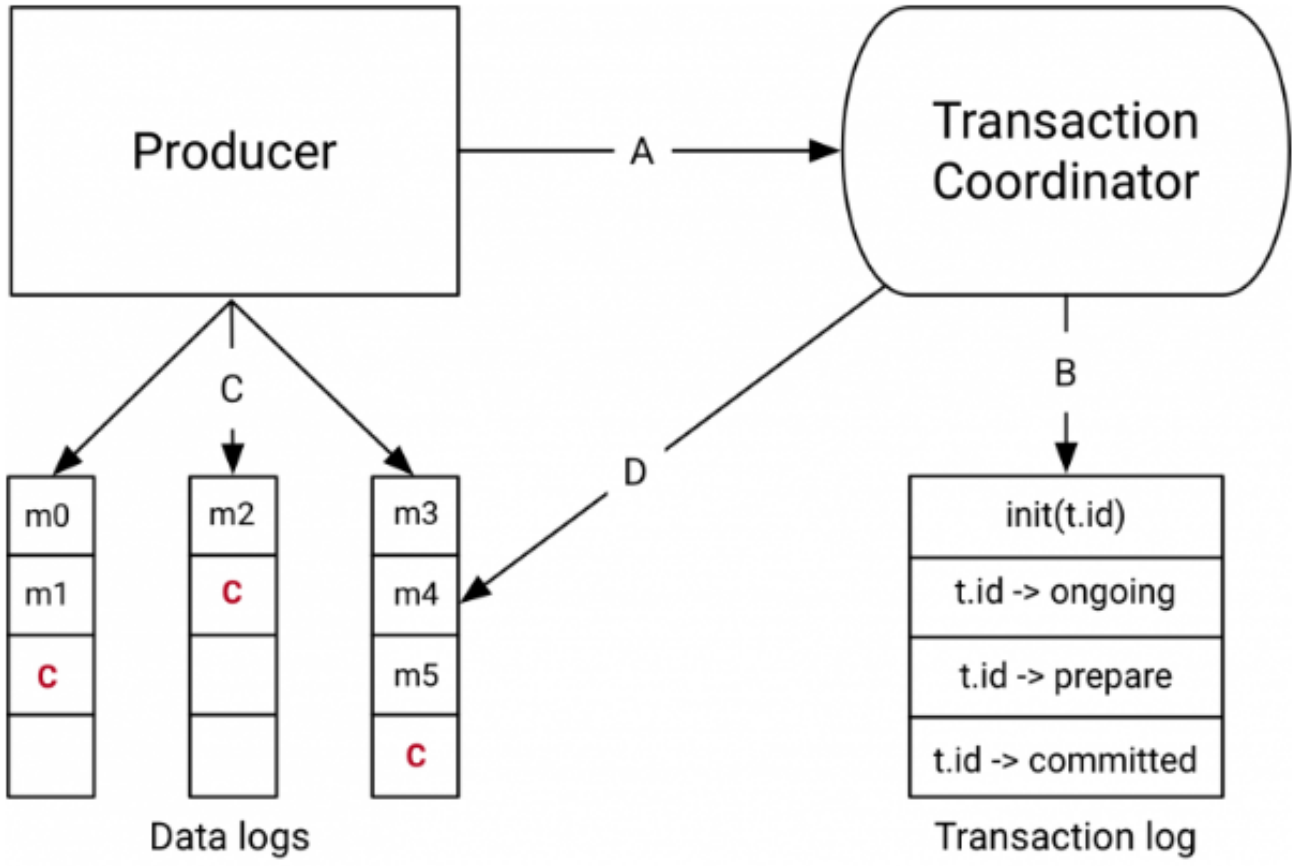


Figure 66: Transactions high-level overview [53].

Transactional Producers begin a transaction, perform multi-partition writes, and afterwards they either Commit or Abort the transaction. These Commit and Abort operations will be persisted in the form of transaction markers within the relevant partitions. A transaction coordinator delegate is used to facilitate the commits and aborts of a given Producer. This transaction coordinator also persists the *state* of the transactions in an internal `__transaction_state` topic, which can be used in sudden failure cases to resume or abort a given transaction. The transaction coordinator uses two-phase commit in order to properly ensure that the markers will be properly persisted. The servers keep track of running and aborted transactions, and also maintain the last message offset that is not part of a running transaction (Last Stable Offset - LSO). Consumers that want to take advantage of the transactional mechanics, can set their isolation level configuration parameter to `read_commited` in order to only fetch messages up to the LSO as well as skip messages of aborted transactions.

transactional.id enables a Producer to set a persistent identifier that will be used to uniquely determine its identity across multiple Sessions. Configuring a `transactional.id` also implies that Producer idempotence is enabled. The server, instead of issuing a new PID, will instead use the `transactional.id` that has been configured for a given producer. The server will also issue a new Epoch number that can be used to fence against zombie producer instances. When a new session is detecting using an existing `transactional.id`, any undergoing transactions are completed.

A producer can persist its progress inside a Kafka topic and then use transactions in order to atomically record both its progress and also produce to a given Kafka topic. Retries will be handed using the normal idempotence semantics. Under restarts, the Producer can read its progress from the corresponding topic and resume from where it stopped. Due to progress and produced messages being part of the same transaction, Kafka can guarantee that either both were properly recorded or none were. This way, we can make sure that produced messages will only be stored in the server once.

Effectively Once Processing

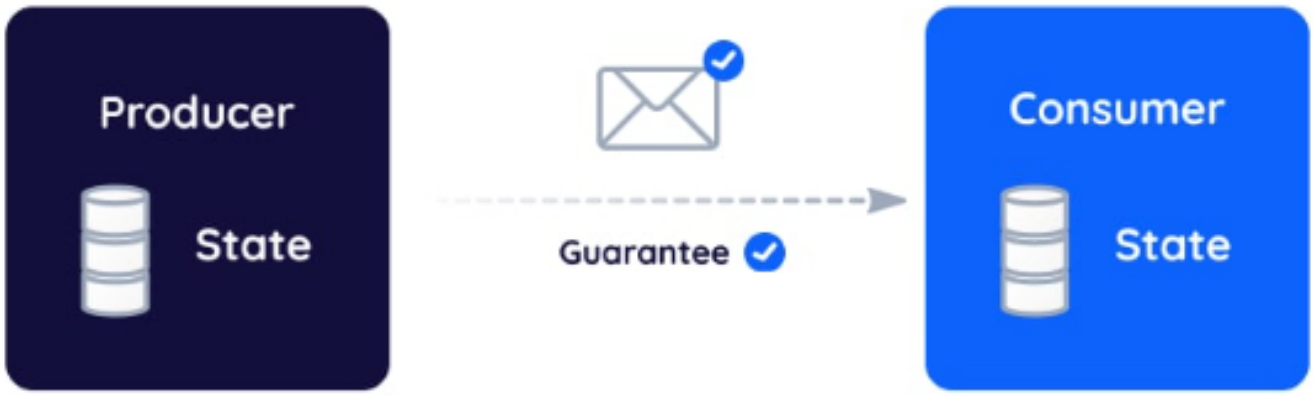


Figure 67: Effectively once processing using idempotence and transactions [51].

All Consumers that are part of a consumer group, do in fact keep their progress inside Kafka using the internal `__consumer_offsets` topic. Consumers can piggyback on an ongoing Producer transaction in order to commit them as part of the transaction. As we discussed in Idempotence, if the processing part of the Consumer has side effects within the bounds of the same transaction-able system, then we can use transactions to ensure that processing occurs only once. In the case of Kafka Consumers, if their processing side effects consist of performing pure transformations and producing to other Topics, then we can make committed offsets and produced topics part of the same transaction in order to achieve effectively once semantics. Effectively once in this case means that the underlying pure data transformations may have to be re-applied, but that won't result in externally visible state changes.

This whole Consume-Transform-Produce behavior is transitive in nature, which implies that we can use transactions in order to guarantee effectively-once processing in pure data transformation pipelines that form directed acyclic graphs.

Partition rebalancing between different Consumers that are part of a Consumer Group makes choosing `transactional.id` complicated. Any assigned partition that is undergoing processing

may at any point be revoked from the given Consumer and assigned to another one. The new Consumer will have to take ownership of transactions that will be used to persist that partitions' offsets as well as produced records. This behavior implies that any given partition that is part of a Consumer group will require its own assigned Producer with a unique - for that partition - `transactional.id`.

Using this information, we can derive a simple Consumer-Transform-Producer architecture for any given transformation node. Each Consumer, associates a unique Producer instance with each assigned Partition. The `transactional.id` that the Producers use should carry over when partitions are revoked and moved to other Consumers. As a result, the `transactional.id` of each Producer should be a derivation of the Consumer Group, Input topic and assigned Partition number. A good choice is `<group id>.<topic>.<partition>`

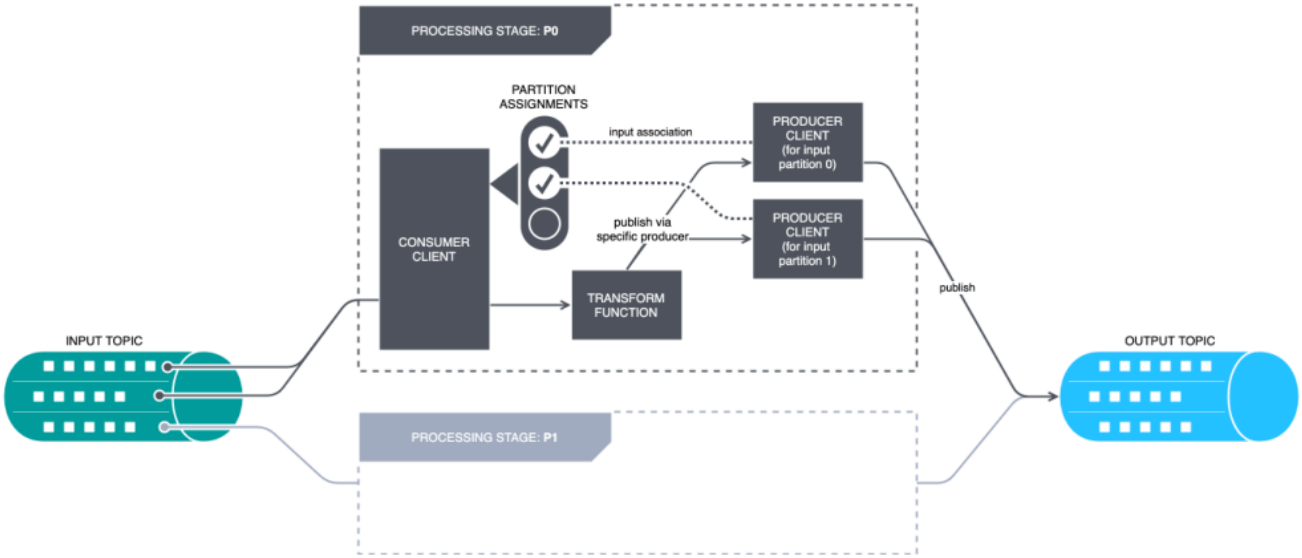


Figure 68: Each partition is assigned it's own Producer [16].

We finally want to stress that effectively once processing only applies to pure data transformation pipelines that produce only Kafka-specific side effects. Any produced external side effects such as sending an email or sending a command to an external service, can potentially be executed multiple times. The only reason that we can get away with Kafka-specific side effects, such as appending to a Topic, is due to being able to take advantage of Kafka-specific transactions.

5 Apache Kafka Modeling, Cluster and Topic Sizing

This section presents a mathematical model first introduced by Amazon [54] and further enhanced by us in order to derive estimations for infrastructure resource requirements as well as specific topic sizing guidelines.

We begin with some definitions, the definitions apply for a single topic:

- T_T is the Total Topic Throughput
- T_S is the Single server Throughput
- T_N is the per-server Network Throughput
- n, the total number of servers
- r, the replication factor of a given topic
- g, the number of independent consumer groups of a given topic

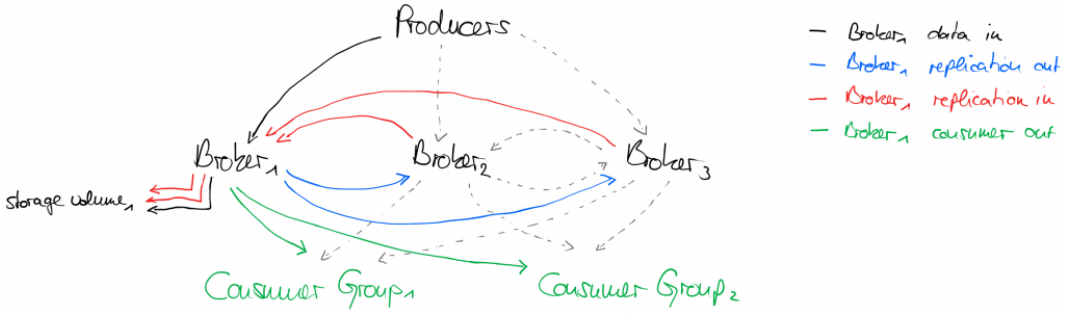


Figure 69: Kafka Infrastructure Model [54].

As we can see above, the Producers send a total of T_T throughput into the cluster. Due to *ideal* even distribution of load, servers share the total amount of traffic from the Producers. Each server also receives additional traffic due to replication, which is transported through the local network. Traffic will be persisted to the server-specific storage, limited by its associated storage throughput limit.

As a result, maximum possible throughput is tightly coupled to the number of servers, their storage, the network throughput and the replication factor
[54] [55] .

5.1 Infrastructure Requirements and Estimations

Each server will receive approximately $\frac{T_T}{n}$ throughput from the Producers.

Total extra replication traffic in the cluster is $T_T \cdot (r-1)$.

This means that server receives approximately $\frac{T_T}{n} \cdot (r-1)$ extra traffic due to replication.

As a result, each server will receive a total throughput of $\frac{T_T}{n} \cdot r$.

We conclude,

$$\begin{aligned} max(T_S) &\geq \frac{max(T_T)}{n} \cdot r \Rightarrow \\ max(T_T) &\leq \frac{max(T_S) \cdot n}{r} \end{aligned} \tag{1}$$

As discussed above, each server receives $\frac{T_T}{n} \cdot r$ traffic due to Producers and replication. They also have to be able to send the Producer traffic to each independent *Consumer Group*.

As a result, we get,

$$\begin{aligned} max(T_N) &\geq \frac{max(T_T)}{n} \cdot r + \frac{max(T_T)}{n} \cdot g \Rightarrow \\ max(T_N) &\geq \frac{max(T_T)}{n} \cdot (r+g) \Rightarrow \\ max(T_T) &\leq \frac{max(T_N) \cdot n}{r+g} \end{aligned} \tag{2}$$

Taking all of the above relations into account, we end up with the following formula,

$$\begin{aligned} max(T_T) \leq min \{ & \\ & \frac{max(T_S) \cdot n}{r}, \\ & \frac{max(T_N) \cdot n}{r+g} \\ \} & \end{aligned} \tag{3}$$

The above can also be expressed in a more practical format that allows us to derive the required Storage and Network throughput for each Topic. This depends on the required total throughput, number of servers, consumer groups and replication factor. The latter depends on the safety guarantees as discussed in Replication. Increasing the number of servers lessens the requirements.

- $$max(T_S) \geq \frac{T_T \cdot r}{n} \tag{4}$$
- $$max(T_N) \geq \frac{T_T \cdot (r+g)}{n} \tag{5}$$

5.2 Topic Sizing

An important aspect of managing Topics is choosing the number of Partitions. With more Partitions, we permit more consumers per Consumer group, increasing read-throughput. We also increase write-throughput up to the number of total servers available. [56] [57] .

We define,

- p, Number of partitions
- T_C, Single-consumer throughput

In order for a topic to maximize write throughput and properly balance load through the cluster, it should be able to utilize all servers. RAID setups can allow for further improvement of throughput when increasing partitions further.

As such,

$$p \geq n \tag{6}$$

In order for servers to uniformly distribute a topic's load without overloading specific servers, the total partition count including the replicas should be a multiple of the number of servers. As such,

$$p \cdot r = 0 \mod n \tag{7}$$

An easy way to achieve this is by forcing p to be a multiple of n,

$$\begin{aligned} p &= 0 \mod n \Rightarrow \\ p \cdot r &= 0 \mod n \end{aligned} \tag{8}$$

In order for consumers to be able to meet total topic throughput, we have to ensure that in the extreme case where each Consumer is assigned a separate partition, then each consumer can meet the partition throughput. As such,

$$\begin{aligned} T_C &\geq \frac{T_T}{p} \Rightarrow \\ p &\geq \frac{T_T}{T_C} \end{aligned} \tag{9}$$

This will usually be the biggest contributing factor to choosing the number of partitions.

5.3 Upper Partition Bounds - Original Work

This section introduces the well-known approach of *over-partioning* and uses the previous model along with details about Kafka's operational behavior in order to link partitions with maximum unavailability windows and end-to-end latency. These can then be used to derive upper partition bounds.

Over-partitioning

While it is possible to increase the number of partitions after topic creation, this has two very critical consequences.

First, as briefly mentioned in Partitioning And Message Ordering, changing the number of partitions implicitly changes the resulting mapping of *keys* to Partitions. As a result, any topics that rely on maintaining message ordering through keys, will loss all ordering guarantees. In order to handle this case, one has to create a separate topic with enough partitions and assign a consumer to clone the existing topic to it. Then one can do a rolling redirection of the clients to the new topic.

Another issue is the extra traffic that assigning new partitions creates. Ideally, for availability and performance reasons, we would like our servers to have an equal distribution of Partitions. In this case we need to move whole Partitions across the server which is a very bandwidth expensive operation.

For the above reasons, it is advised that we avoid Partition resizing by initially over-partitioning our Topics so that we don't ever require additional Partitions.

It is worth noting however that there are a few negative consequences of over-partitioning topics.

More open file-handles

Since each partitioning maps to a directory along with the relevant log files, having more partitions increases the total number of open handles within a server, potentially reaching the maximum size of open file handles of the operating system. This is not a big issue and can be overcome by proper configuration.

Increased Memory Requirement for Servers and Clients.

Both servers and clients use buffering in order to increase throughput. Often buffering occurs at the partition level, so having more partitions requires more Memory due to buffering. This can be an issue when the number of partitions is very large.

Potential Unavailability Increase

In the case of manual server termination, partitions are iteratively moved one at a time. As a result, a client will observe the partition unavailability only for the duration of a single Leader election cycle.

In the case of unclean server termination, all partitions become unavailable at the same time. As a result, the same number of Leader elections as partitions will take place. The time will scale linearly with the number of pending leader elections due to them taking place in the Controller which must coordinate multiple phases for each election [56] .

As an example, t denoting total topics, if the leader election takes t_{elect} for a single partition, then a server will contain approximately $\frac{t \cdot p \cdot r}{n}$ partitions of which it will be the leader of $\frac{t \cdot p}{n}$ partitions. This will result in a total of $t_{elect} \cdot \frac{t \cdot p}{n}$ seconds of unavailability. As an example, given $t_{elect} = 5ms$, $t = 50$, $n = 5$, $p = 50$, we have an unavailability window of $2.5s$ due to leader elections.

> Important note!
>
> The below sub-section only applies for the traditional controller implementation that uses Zookeeper. At the time of writing, the new Quorum Controller implementation has been announced as production-ready. Due to the new Raft-like implementation, metadata will always be available on voter nodes, so the below unavailability issue does not occur [58] .

If the failed server is also the controller responsible for Leader elections, then all partition metadata will also have to be fetched from Zookeeper to a the new controller, increasing the unavailability window further. The new controller will have to fetch metadata for all partitions in the cluster. If t_{meta} is the time required to fetch a partition's metadata from Zookeeper, then the new controller will require $t \cdot p \cdot r \cdot t_{meta}$ seconds. Given the previous parameters and $t_{meta} = 2ms$, this will result in an extra $15s$ of unavailability.

End-to-end latency increase

Due to the nature of the inter-server replication model, partition replication use a single thread per peer server connection. [17, p. 86]

We define as t_{repl} the time required to replicate a single partition to a server. Leading partitions on a given server are approximately $\frac{t \cdot p}{n}$. This means that the server will have to replicate them $r - 1$ times to the other $n - 1$ servers. As a result, it will have to replicate $\frac{(r-1) \cdot t \cdot p}{n(n-1)}$ partitions to each peer, which results in any given partition requiring $t_{repl} \cdot \frac{(r-1) \cdot t \cdot p}{n(n-1)}$ time before being replicated.

Deriving Partition Count Upper Limits

Here we will illustrate how we may derive an upper partition limit.

This upper limit depends on the cluster size, the topic count, the average replication factor and partition count as well as the maximum tolerable unavailability window in case of unclean leader failures and maximum tolerable end-to-end latency.

We define:

- $max\_unavailability$, The maximum tolerable unavailability window in seconds
- $max\_latency$, The maximum tolerable end-to-end latency

Then we derive,

- End-to-end latency restriction

$$\begin{aligned} t_{repl} \cdot \frac{(r-1) \cdot t \cdot p}{n(n-1)} &\leq max\_latency \\ p &\leq \frac{n(n-1) \cdot max\_latency}{(r-1) \cdot t \cdot t_{repl}} \end{aligned} \tag{10}$$

- Unavailability restriction (For the new Quorum Controller we can assume $t_{meta} = 0$)

$$\begin{aligned} t_{elect} \cdot \frac{t \cdot p}{n} + t \cdot p \cdot r \cdot t_{meta} &\leq max\_unavailability \\ p \cdot t \cdot \Big(\frac{t_{elect}}{n} + r \cdot t_{meta}\Big) &\leq max\_unavailability \\ p \cdot t \cdot \Big(\frac{t_{elect} + r \cdot n \cdot t_{meta}}{n}\Big) &\leq max\_unavailability \\ p &\leq \frac{n \cdot max\_unavailability}{t \cdot (t_{elect} + r \cdot n \cdot t_{meta})} \end{aligned} \tag{11}$$

We can now illustrate how we can derive an upper partition bound.

We make the following realistic assumptions:

- $t_{elect} = 0.005s$
- $t_{meta} = 0.002s$
- $t_{repl} = 0.00002s$
- $r = 3$
- $n = 5$

Then we can apply the above restrictions with different scenarios.

Traditional Controller

Total Topics	Max unavailability (s)	Partition Count Limit
100	10	14
100	100	142
100	1000	1428
1000	10	1
1000	100	14
1000	1000	142

Total Topics	Max latency (s)	Partition Count Limit
100	0.001	5
100	0.010	50
100	0.100	500
100	1.000	5000
1000	0.001	0
1000	0.010	5
1000	0.100	50
1000	1.000	500

Quorum Controller ($t_{meta} = 0$))

Total Topics	Max unavailability (s)	Partition Count Limit
100	1	10
100	10	100
100	100	1000
100	1000	10000
1000	1	1
1000	10	10
1000	100	100
1000	1000	1000

We observe that end-to-end latency affects the upper bound the most. Max allowed unavailability window also has a large effect when using the traditional zookeeper-based controller, but has almost an order of magnitude smaller effect when using the new Quorum controller.

In order to keep end-to-end latency acceptable, especially on clusters that host many topics, it is a good idea to maintain a smaller partition count, most likely smaller than 50.

As we will see in the next section, this restricts our choices, which in turns influences uniformity when scaling the cluster.

Keeping load uniform

As we previously discussed, in order to ensure uniform server load, the total partition count should be a multiple of the number of servers. When scaling the cluster, ideally we would like to re-distribute the partitions while maintaining the uniform load distribution. In order to do that, we have to ensure that our partition count is a multiple of all cluster sizes that we want to support.

In order to pick the optimal partition count, we can count the cluster sizes that divide a given partition count. Partition counts with the most divisors up to a given cluster size are very good candidates since they ensure the least forced cluster upgrades, minimizing the up front cost when gradually scaling up our cluster.

Below, we make provide heatmaps for partition counts up to 120 and cluster sizes up to 12.

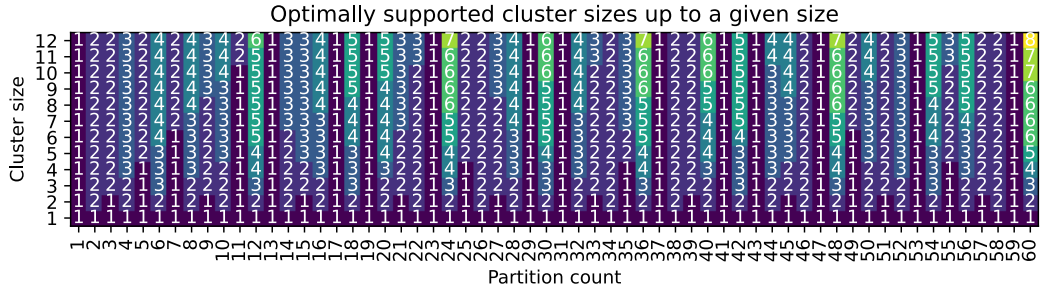


Figure 70: Kafka optimal partition counts 1-60.

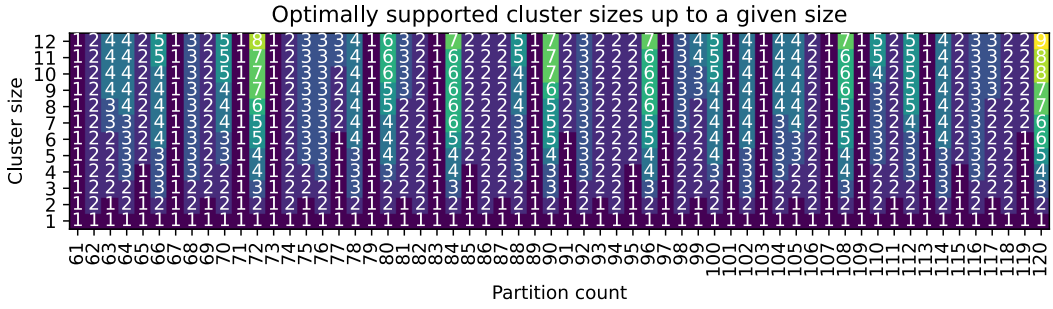


Figure 71: Kafka optimal partition counts 61-120.

One can pick a partition count range according to the upper limit restrictions and throughput requirements, and then search for a partition that is divided by the most cluster sizes up to a target cluster size.

Some partition counts that stand out from the above heatmaps are:

- 24: Ensures uniform load on cluster sizes 1,2,3,4,6,8 and 12.
- 36: Ensures uniform load on cluster sizes 1,2,3,4,6,9 and 12. 24 is most likely a better option due to supporting the same number of cluster sizes, earlier (8 vs 9).
- 48: Has the exact same behavior with 24 but is larger. Could possibly be handy for high throughput scenarios but leans on the upper end when it comes to latency.
- 60: Ensures uniform load on cluster sizes 1,2,3,4,5,6,10,12. Most supported sizes so far but also the largest count and therefore more vulnerable to restrictions. Forces large jump for cluster sizes 7-9.
- 72: Ensures uniform load on cluster sizes 1,2,3,4,6,8,9,12. Comparable to 60. It forces a jump earlier on (5), but forces jumps more uniformly after that.
- 120: Most optimal up to this size. Ensures uniform load on cluster sizes 1,2,3,4,5,6,8,10,12. Also spreads jumps very uniformly, ensuring least up front costs when scaling up. Would be the ideal option if not for its larger size, which may be at odds with the restrictions put into place due to latency and worst case unavailability windows.

6 Case Study - DIEM Platform

SMART RUE (Smart grids Research Unit of the Electrical and Computer Engineering School) is one of the Research Groups of the Institute of Communication and Computer Systems (ICCS).

It belongs to the Electric Energy Systems Laboratory (EESL) of the School of Electrical and Computer Engineering of the National Technical University of Athens.

SmartRUE is leveraging on its expertise in gathering large amounts of market-relevant data and structuring it in a consistent manner. In collaboration with Enargia A.G., have developed an ecosystem of tools (DIEM Platform) to actively support energy market participants.

DIEM platform provides timely access to all relevant data for the energy Market in Greece and the wider area. The core of this ecosystem is a central repository that stores semantically aligned data from different sources. The participant can access the information via an API or a user-friendly web page.

This case study focuses on restructuring the web-page architecture in order to better facilitate real-time updates of relevant data by improving communication overhead and enabling scaling of the associated services.

High-Level Architectural Overview

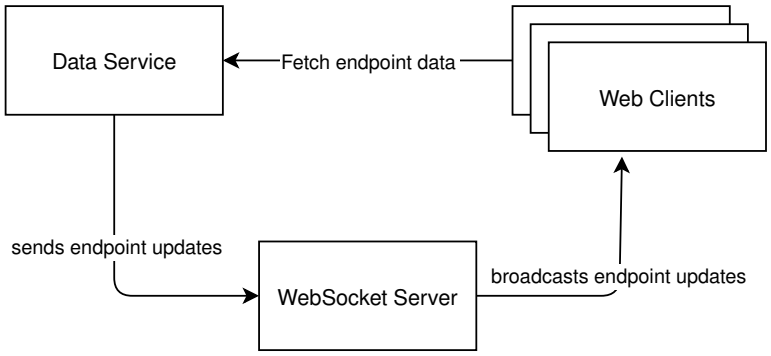


Figure 72: Simplified Overview.

- The *Data Service* is responsible for observing curated sources, and then fetching the relevant data in order to make it available through a public facing REST API. The API is used for metrics and enforcing pricing plans. The various data sources are exposed by different API endpoints.
- The *clients*, consist of the browsers that service the DIEM web-page in order to provide user-friendly views of the data that is provided by the Data Service. The data is fetched using the REST API that is provided by the Data Service. It is deemed important that the various views are updated in soft-real time, permitting users to be notified of data changes without having to refresh their page manually. Moreover, due to the large number of datasets and API endpoints, the results of the API requests should be cached in local storage in order to minimize the traffic load of the Data Service. For these reasons, clients need a way to be notified of underlying endpoint data changes. The current implementation uses WebSockets in order to establish a two-way communication channel between each client and the underlying *WebSocket Server*. The server can efficiently notify clients of multiple endpoint updates, and clients can then make the corresponding API requests, updating their local storage in the process.
- The *WebSocket Server* keeps a persistent connection with each connected client. It communicates with the Data Service and is notified by the latter when the datasets that are exposed from the Rest API endpoints are updated. The server then relays this info to all connected clients, permitting them to more efficiently fetch new data from the Data Service.

The original architecture has some implementation issues that increase infrastructure requirements. The goal of our thesis is to restructure the communication flow in order to fix these issues, as well as provide a way to scale the infrastructure in order to meet future requirements if needed.

- **Lack of Endpoint Updates Granularity**

The WebSocket server, currently, forwards all endpoint updates to all Clients. This creates unnecessary load to both the WebSocket server, since it has to a forward all update events to all clients, and to the Clients themselves, which have to filter all incoming update event and dispatch them to the corresponding Views.

Fundamentally, the front-end knows which views need to be updated at any given time. This means that it can communicate to the WebSocket server, the corresponding endpoints that it is interested in. The server can then perform some extra bookkeeping and dispatch the update events only to the interested clients. This considerably reduces the total number of update events sent to clients.

- **Lossy poll-based front-end handling of update events**

The original version of the Front-End uses a singleton *WebSocket Service* component that communicates with the WebSocket Server, and is used by the various Views in order to observe new endpoint updates received. The component observes an incoming stream of endpoint updates and only keeps the latest one, clearing it after a few seconds. The various views then poll the Service at specific intervals, expecting that they will observe all updates. This is obviously not always the case. Specifically, in the scenario where the update events arrive faster than the polling frequency of the views, the views can and will lose update events. The solution we used was to use the Observer pattern, effectively providing a subscription-based interface to the WebSocket Service component, and allow the various views to subscribe to the endpoints they are interested in. On each endpoint update, the WebSocket Service can notify all interested views, ensuring that they will not miss the update.

6.1 Frontend

DIEM Platform's frontend, is an Angular application. Our use case deals with the various Views available throughout the application. The Views fetch data by using a REST API and subsequently render the data in a view-specific format.

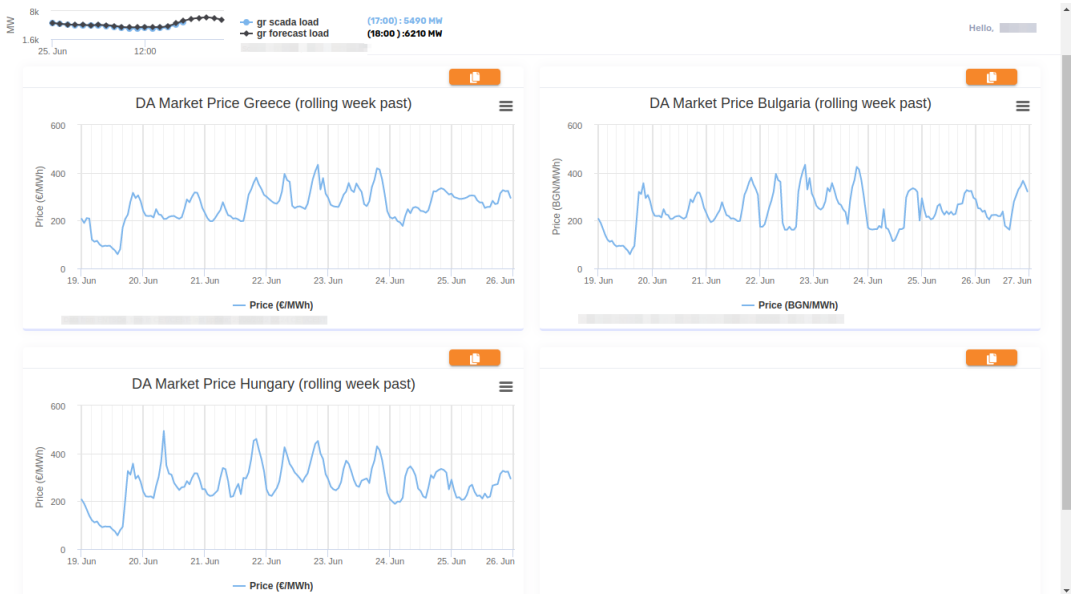


Figure 73: Example DIEM platform views.

The Views use an injected *DataService* provider in order to fetch the data by specifying dataset-specific Endpoints. They also use an *UpdatesService* provider in order to be notified of Endpoint dataset changes in order to re-fetch the data and subsequently render it.

Below, we provide an approximate UML class diagram. We took the liberty of using an equivalent version of the actual class hierarchy in order to provide a clearer high-level view and facilitate the analysis.

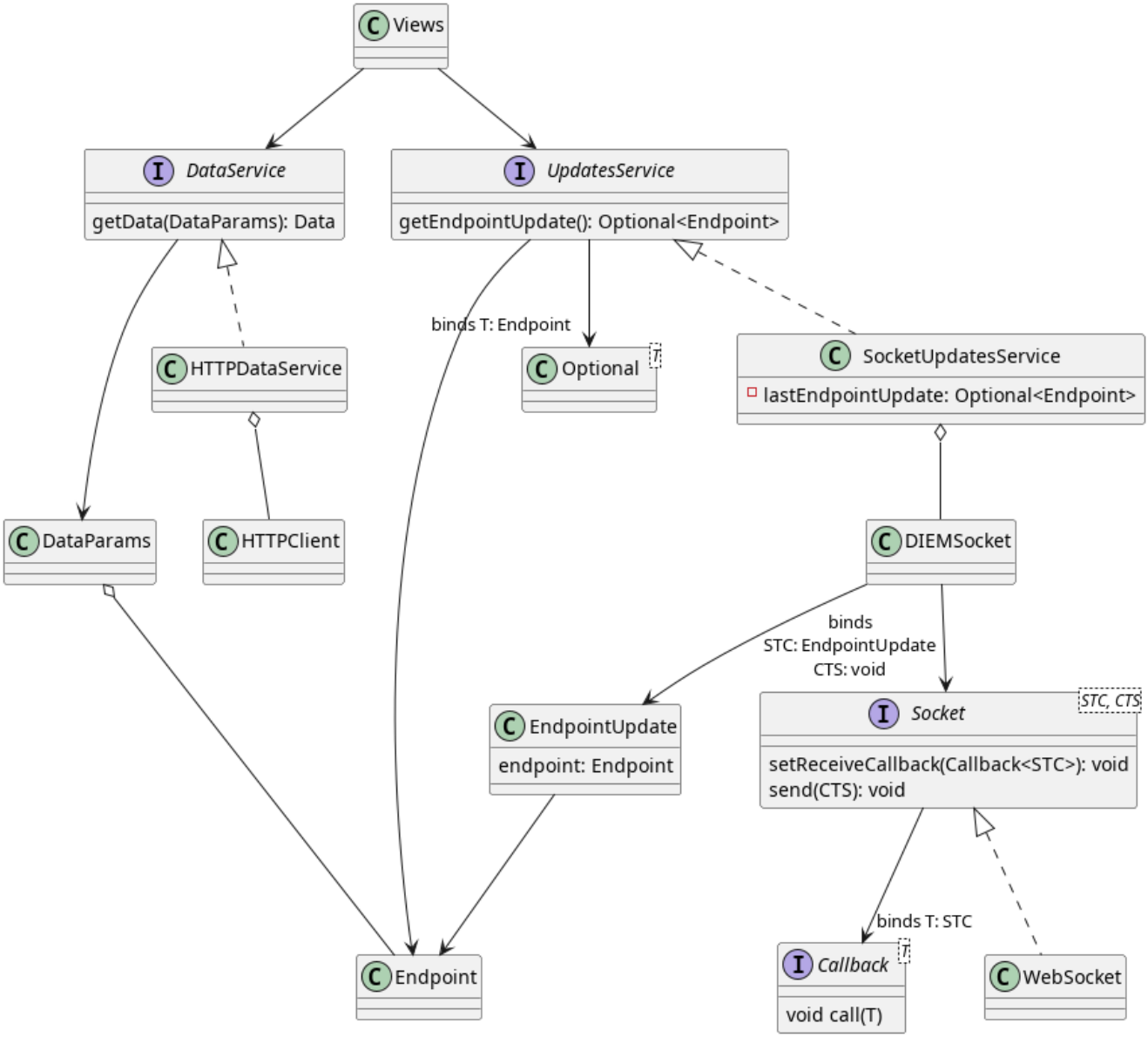


Figure 74: Initial frontend architecture.

From the above diagram, we can notice two problems:

1. Looking at the `DIEMSocket` class, we can observe that there is no communication from the Client to the underlying WebSocket Server. Because of this, the frontend cannot send any info about the `Views`' interested `Endpoints`. As such, the server has to forward all `Endpoint` updates to the frontend, instead of only forwarding the `Endpoint` updates that the `Views` are interested in.
2. The `UpdatesService` provides a single method that retrieves the latest `Endpoint` that got updated. The `Views` perform periodic polling on this method in order to detect any changes to the `Endpoints` that they care about. If the polling frequency is not fast enough, the `Views` can fail to observe corresponding `Endpoint` updates in high-load situations.

We can solve these problems by performing a series of enhancements to the `UpdatesService`:

1. We add dedicated `Subscribe/Unsubscribe` messages to the communication protocol between the frontend and the WebSocket Server. This allows the frontend to relay information to the backend, so that it is able to send only the relevant `Endpoint` updates to the frontend, subsequently reducing the client-server communication traffic.
2. We add the corresponding `subscribe/unsubscribe` methods to the `UpdatesService` provider. Views can then provide `Callbacks` at subscription time. The `UpdatesService` can maintain a mapping of `Endpoints` to listening `Views` and then notify them, when the appropriate `Endpoint` updates arrive.

We provide a new class diagram with the revised interactions. The Visitor pattern is used in order to express `ClientToServerMessage` as a sum type within the bounds of the UML language.

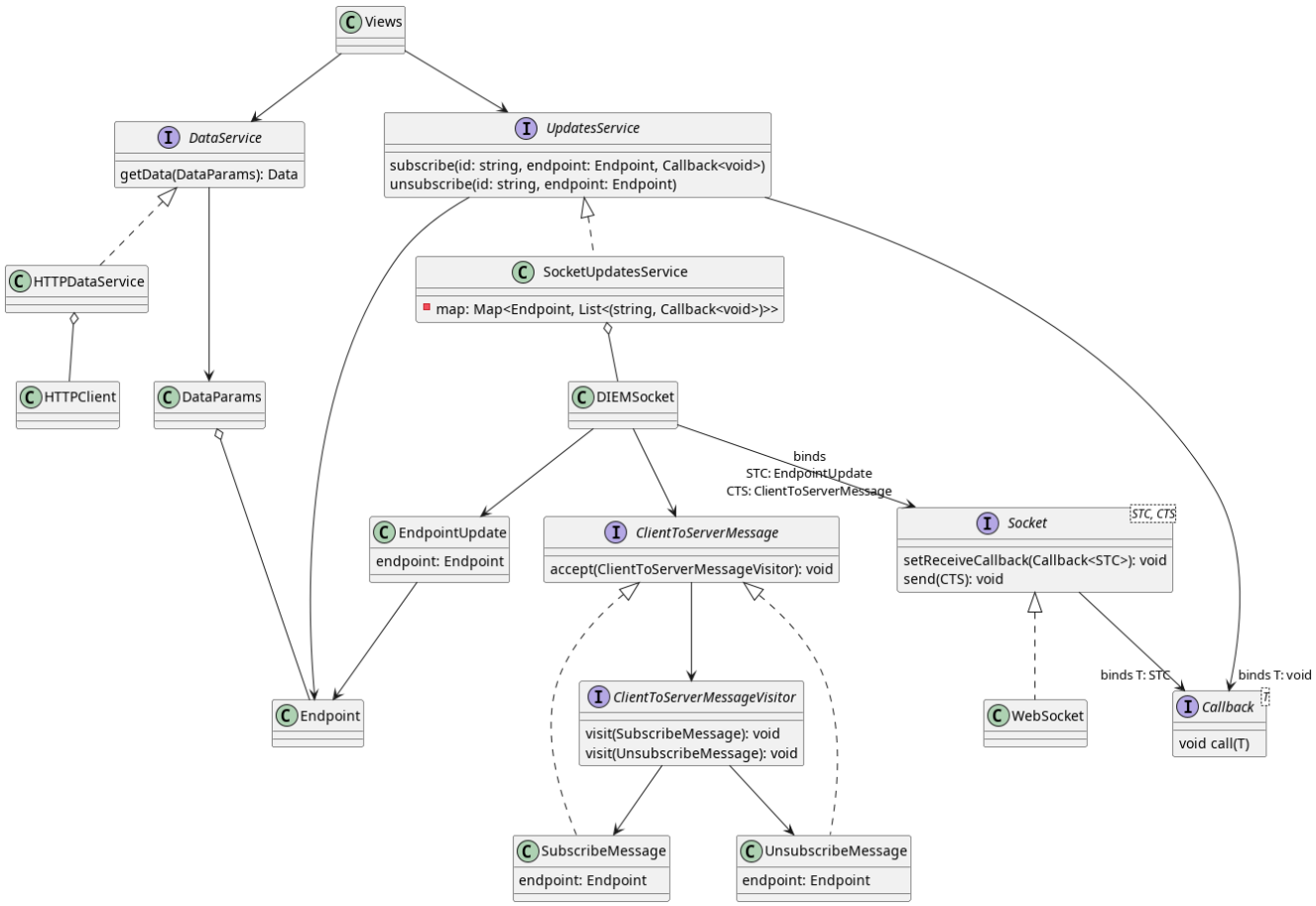


Figure 75: Enhanced frontend architecture.

6.2 Backend

DIEM platform's backend part is responsible for observing sources, downloading any new data, unifying them under a common format and then make them available through a Rest API that both the frontend but also external services can use. Finally, it is also responsible for caching data and transmitting endpoint update events to the frontend for real-time updates of relevant views.

Below, we provide a sequence diagram of the original implementation:

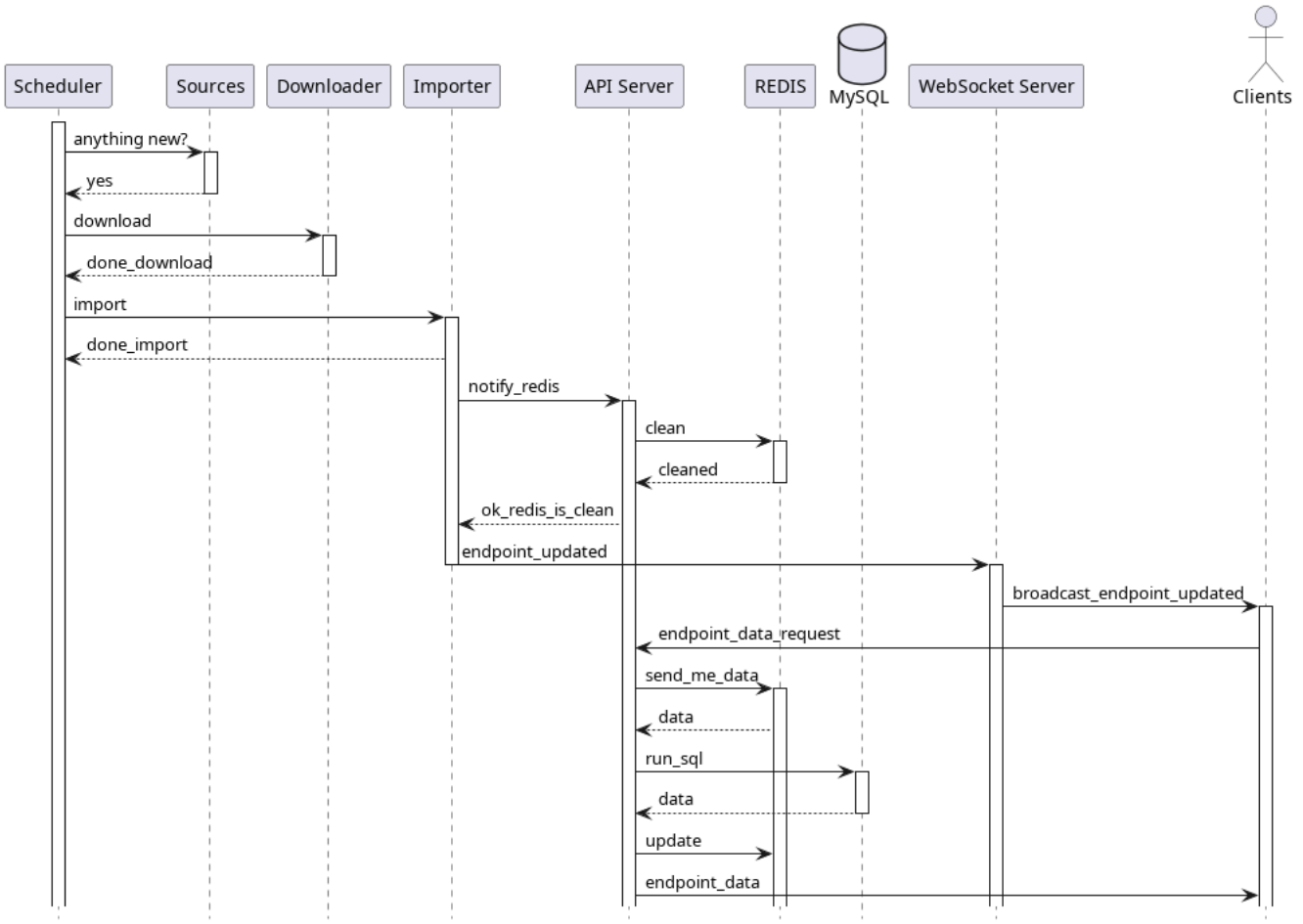


Figure 76: Original sequence diagram.

In order to facilitate analysis, it will be helpful to consider most backend participants as a singular Data Service unit. We can see that the resulting diagram is greatly simplified, allowing us to focus on the interaction between the updates, the clients and the WebSocket server:

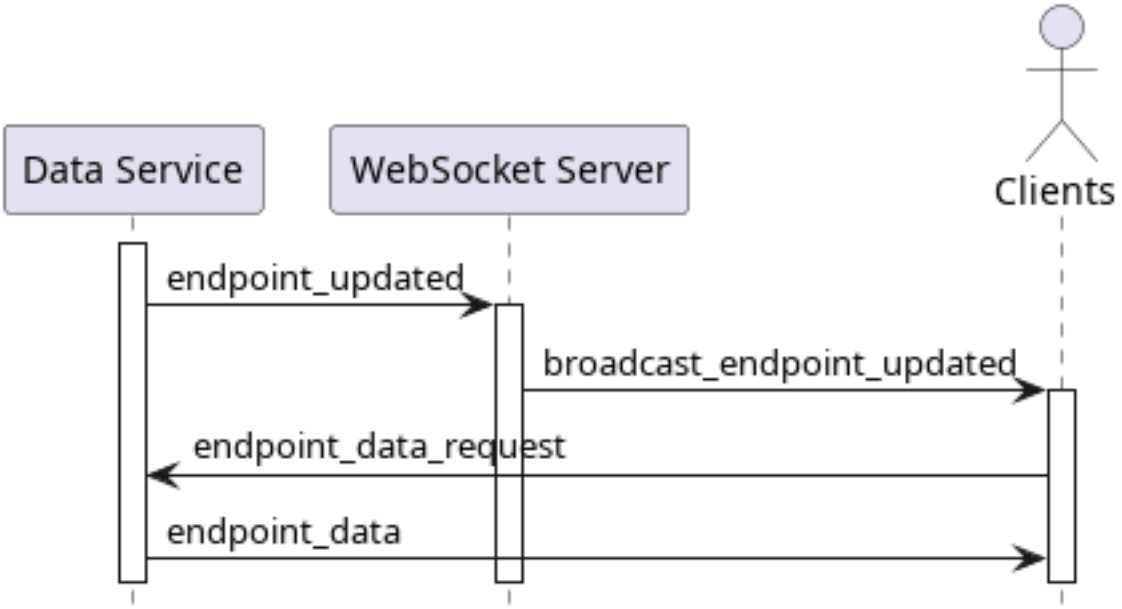


Figure 77: Simplified original sequence diagram.

Relaying endpoint interest using subscription messages

In the Frontend section, we added subscription messages to the client-server communication protocol, allowing clients to relay information to the WebSocket Server regarding the endpoints that they are interested in. This allows the WebSocket Server to add clients to a notification group on an endpoint basis. When an endpoint update arrives, the server can notify only the relevant clients.

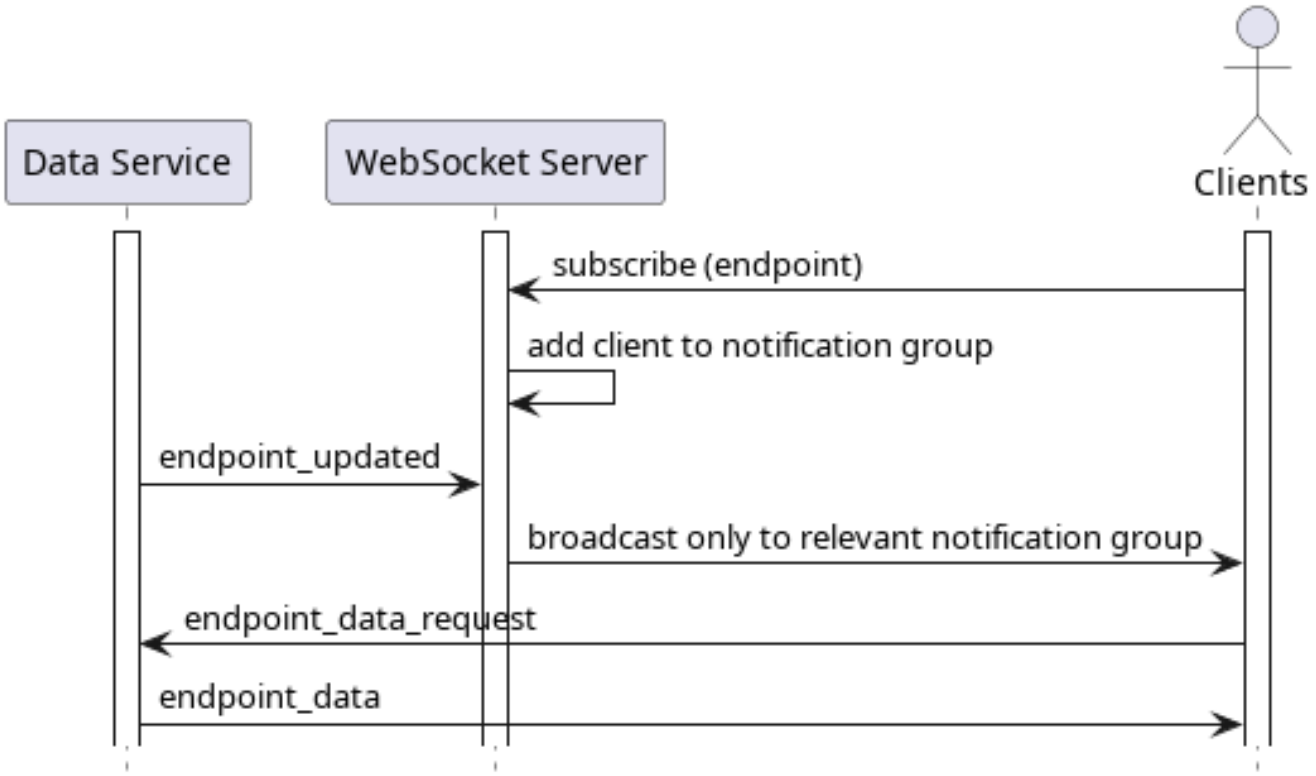


Figure 78: Enhanced sequence diagram.

6.3 Scaling

As the user base of the platform grows, it is very important to devise a scaling strategy in order to independently scale the various services in and meet the increased load requirements. For this case study, we decided to focus on the scaling of the WebSocket server. We present both X and Z-axis scaling [10], one utilizes Publish-Subscribe along with a client-facing load balancer, while the other takes advantage of data partitioning, in order to reduce the load of each individual instance, without increasing inter-service network traffic.

X-Axis Scaling – Scaling through cloning

This is the easiest approach to scaling our WebSocket server. Our end goal is to distribute the Clients to independent WebSocket Servers. In order to do this, we first have to create multiple WebSocket Server instances. Then we can introduce a reverse proxy between the Clients and the instances, whose sole purpose is to connect each client to a separate server. In order for each instance to be independent and able to serve any clients, it needs to receive its own copy of endpoint updates from the Data Service. This requires some form of Publish-Subscribe communication between the Data Service and the various instances. Then architecture is illustrated below:

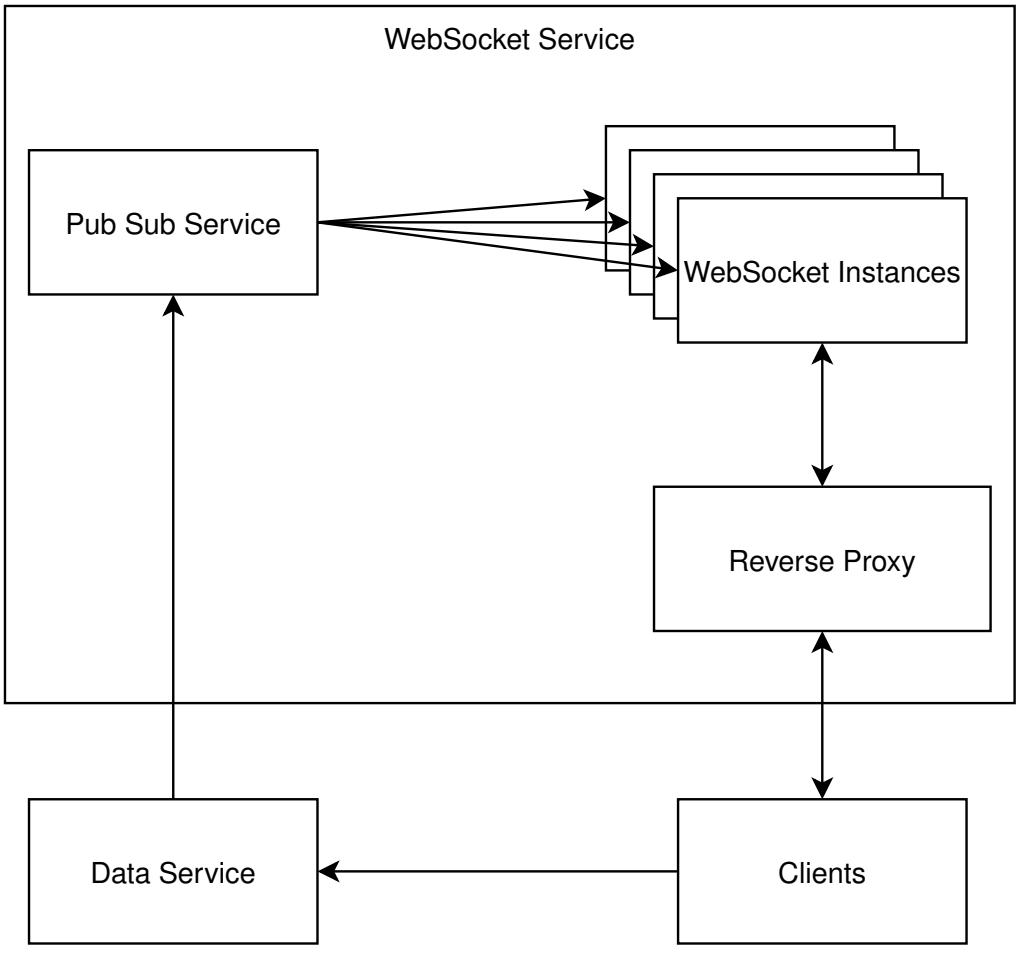


Figure 79: High-level X Axis scaling.

The above scaling scheme, does not require any modifications to the frontend. For the Reverse

Proxy, we used nginx along with it's WebSocket support. For the Pub/Sub service, we decided to go with Apache Kafka. At this point, we have to stress that Apache Kafka is definitely overkill for this use case due to multiple reasons:

- Messages are idempotent in nature, Kafka's partition model and ordering guarantees are not very useful.
- Even with increased traffic, the update events are not likely to reach disk-saturation throughput numbers. As a result, Kafka's binary log zero-copy architecture will be underutilized.
- Messages are ephemeral in nature, only required for updating user-views. Replaying endpoint update messages would result in clients performing multiple data fetches for the same underlying data. The log-based model of Apache Kafka is not very useful in this use case.
- The server instances do not currently perform heavy per-message processing, so scaling throughput through Consumer groups is not as useful. As a result, consumer group partitioning, a very important part of Apache Kafka, is not utilized.
- Kafka in general sacrifices latency in order to achieve higher throughput. While it is possible to configure the system for low latency, this will go against the default intended behavior. Other services can provide better latencies when throughput is low and safety is not as important.
- Kafka is a relatively heavyweight system, requiring multiple servers in order to provide its guarantees, and also traditionally requiring the additional setup and operation of a separate Zookeeper cluster for proper operation. We can use a smaller cluster, but this will undermine Kafka's high availability and safety guarantees. Thankfully, due to the new Quorum Controller, Zookeeper will soon no longer be necessary, resulting in easier and more cost-effective cluster deployments.
- It doesn't make a lot of sense to introduce a complete event streaming platform for a single and very simplistic pub-sub scenario between a few nodes.

A more sane option would most likely be to maintain WebSocket connections between the WebSocket Server instances and a dedicated Broadcast node. The Broadcast Node would act as a fanout exchange, sending the incoming endpoint update events to all connected WebSocket service instances. This way, the platform could be scaled without having to introduce a new heavyweight technology. Any Broadcast Node failure would of course result in an unavailability window until the issue got resolved. This would not be the case with Kafka, a small advantage in the large scheme of things.

With that in mind, the choice was made with the intention that Apache Kafka will slowly be used for more tasks within the DIEM platform, taking part in data collection, transformation pipelines, activity tracking and more while also facilitating an event-based communication architecture.

For implementing the above Pub/Sub part with Kafka, we wrote a Kafka / WebSocket bridge in the form of a Kafka consumer. The consumer subscribes to a topic and forwards the data to a WebSocket server. Each bridge uses a unique consumer group identifier in order to receive all topic messages. Each WebSocket server, is paired with a Kafka bridge. A producer bridges the Data Service with Kafka. The system is illustrated below,

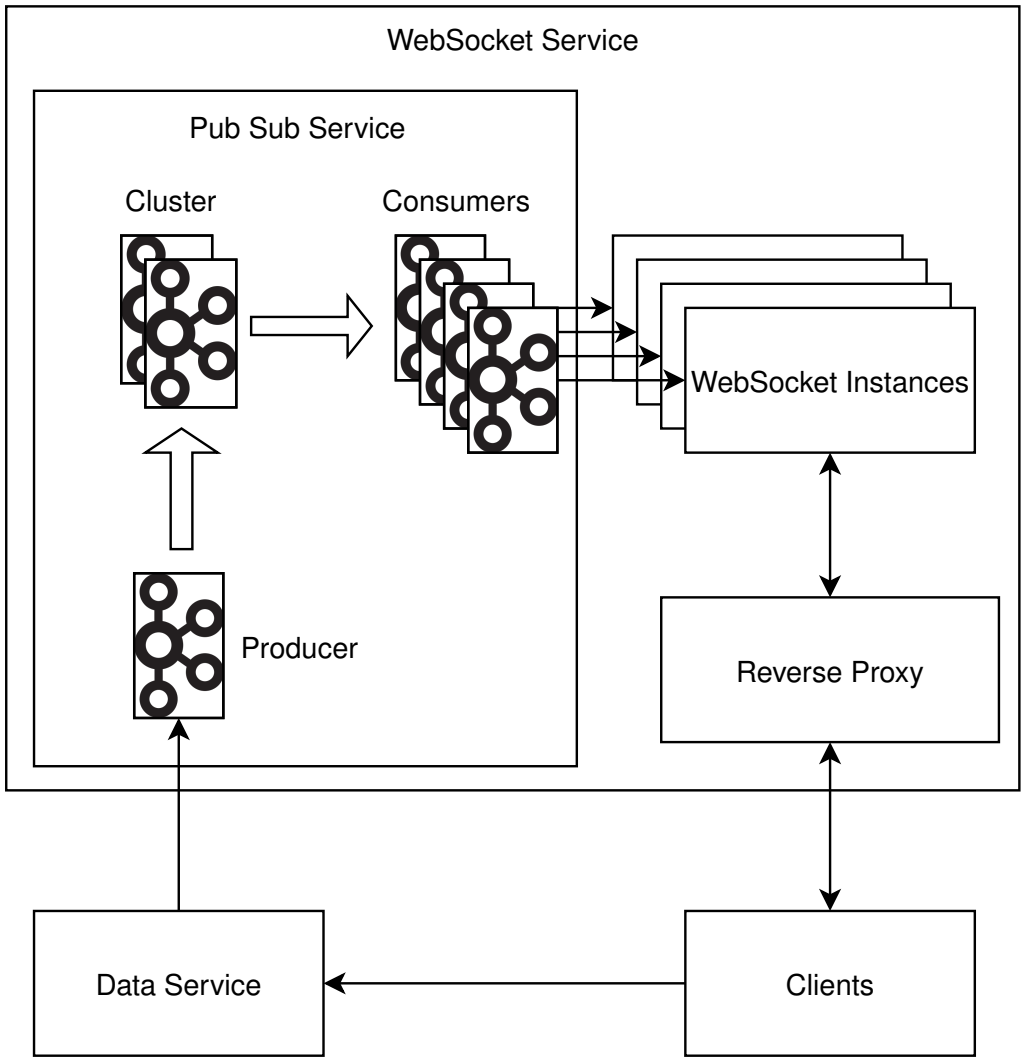


Figure 80: X Axis scaling using Kafka.

The relevant code is included in the support repository.

Z-Axis Scaling - Scaling through partitioning

Another approach to scaling is through data partitioning.

Instead of each server being responsible for all endpoint updates, they instead manage only the updates relevant to a subset of all endpoints. This in turn makes the WebSocket Servers more efficient by limiting the number of messages that they need to process before dispatching. It is especially useful when the event processing itself is more expensive, allowing the Server to significantly reduce its processing load. For our concrete case study, the event processing includes extracting the messages, deserializing it and forwarding it to all relevant WebSocket connections. Another advantage is that by partitioning the data, we reduce the size of the bookkeeping data structures that are used for associating connected clients with endpoint updates.

Under this model, WebSocket servers manage different subsets of updates, therefore, clients will need to somehow communicate with more than one server. It would be useful to provide a

partitioning-aware node that knows which WebSocket server manages what endpoints. The clients can then either explicitly query this node in order to discover the relevant severs, or we can create a smart reverse proxy that will multiplex the connections using the aforementioned partition-aware node.

Kafka's consumer groups are very useful for implementing this scaling approach, permitting for dynamic addition of new WebSocket servers and facilitating service discovery through the usage of its admin client.

Another approach puts the burden of maintaining multiple connections to the client. The client first contacts a Partitioning Metadata service in order to discover the relevant WebSocket servers and then subsequently connects to all of them or perhaps only to the ones that manage the endpoints that it is interested in.

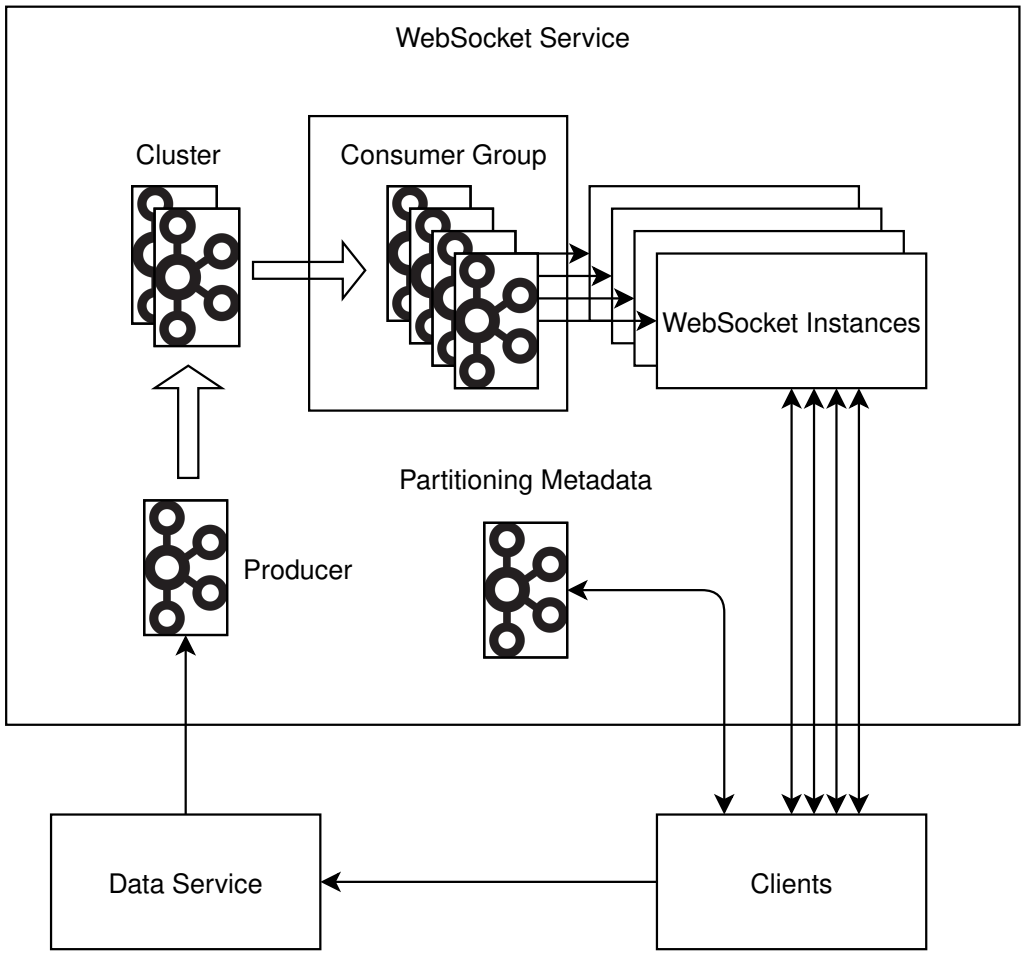


Figure 81: Client-based Z Axis scaling using Kafka.

A nice advantage of doing things this way is that we can then also scale by cloning without introducing a WebSocket-aware reverse proxy. Instead, a simple load balancer to the appropriate Partitioning Metadata service will suffice, each client will end up connecting to a different set of WebSocket instances.

The disadvantage is that it requires Client code modifications in order to work.

We can avoid modifying clients by implementing a multiplexing reverse proxy. The client will connect to the proxy and the proxy will make sure to query the Metadata service and multiplex the resulting WebSocket server connections using TCP pass-through.

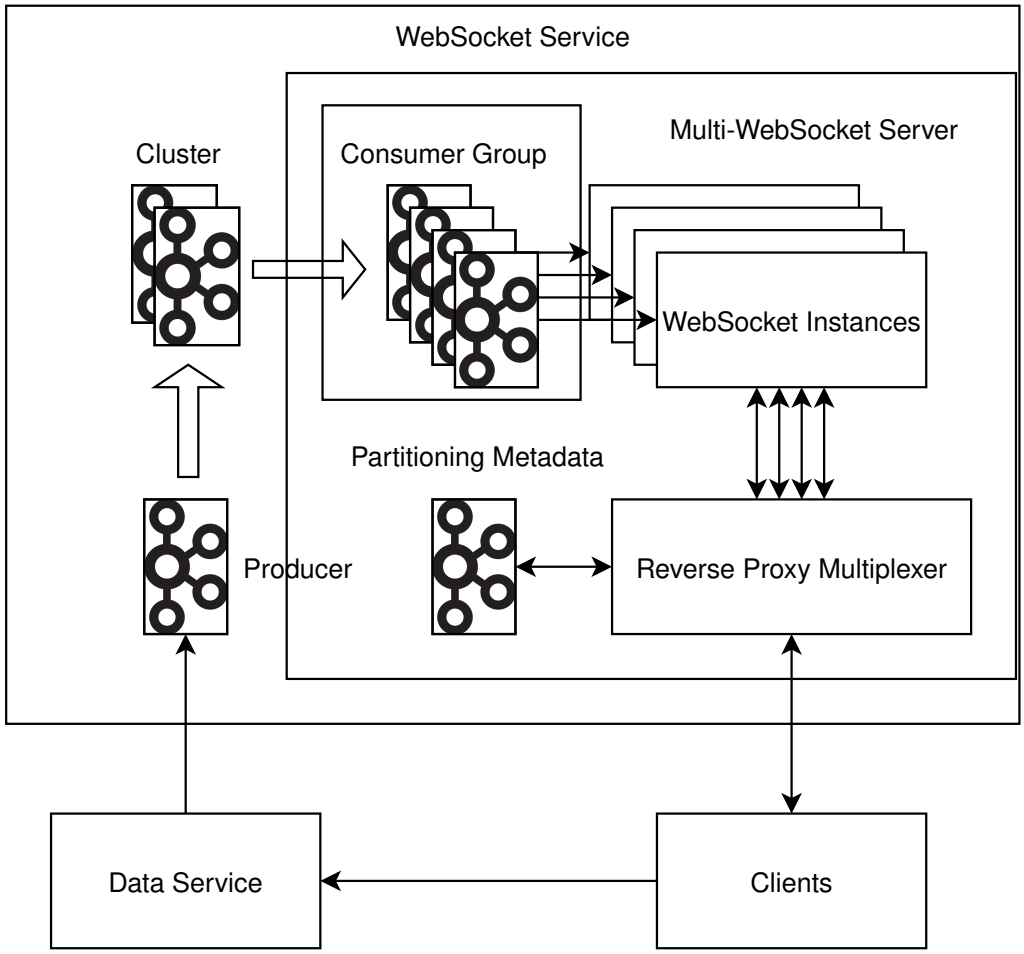


Figure 82: Backend-based Z-Axis scaling using Kafka.

We can then split the bridge consumer group, the WebSocket Server instances, the Partitioning Metadata and the Reverse Proxy Multiplexer into a single Multi-WebSocket composite service:

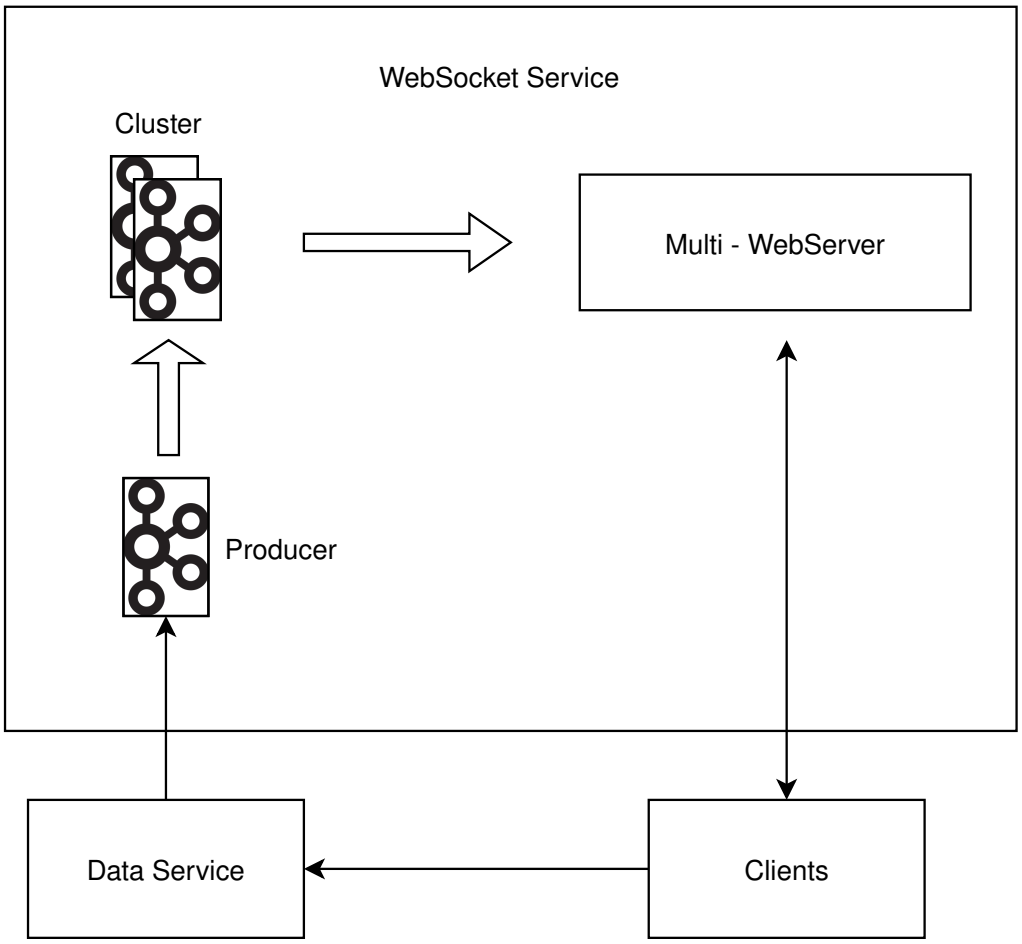


Figure 83: Collapsed Z-Axis scaling using Kafka.

Finally, we can also scale the resulting Multi-WebSocket Servers by cloning:

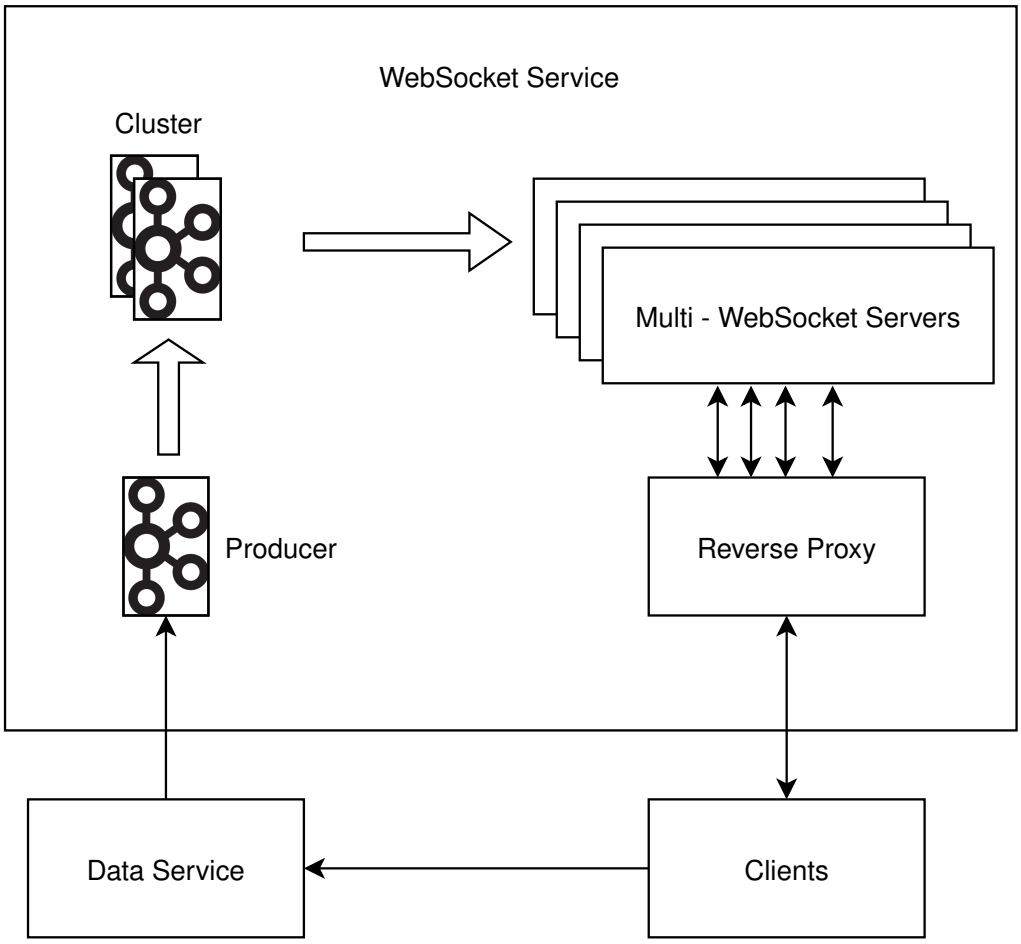


Figure 84: Backend-based XZ-Axis scaling using Kafka.

7 Conclusions and Further Work

7.1 Summary

This thesis presents a useful resource for getting acquainted with event-based architectures and Apache Kafka while presenting a simplified but useful model that can facilitate the taking of real-world engineering decisions.

We explicitly discuss the guarantees that Kafka provides through different configuration options and derive guidelines for proper infrastructure and topic sizing by presenting a simple mathematical model of Apache Kafka's data flow.

From the mathematical model, we derive relations between total number of partitions and end-to-end latency and unavailability windows, giving us a guideline on how to optimize Kafka Clusters for low-latency high-throughput use-cases. We also discuss how careful partition count choices can help achieve uniform load when scaling Kafka clusters.

7.2 Derived Guidelines Summary

We reiterate the various guidelines discussed throughout the thesis so far.

Message safety

In order to ensure message safety, consistency and availability without message loss even in the case of N servers going down simultaneously, the following must bet set:

- Producers must be configured using `acks=all`
- The Topic must be configured with min.insync.replicas $= N + 1$, this implies replication factor $\geq N + 1$ and therefore sets a lower bound on the number of servers. More servers will allow the in-sync replica mechanisms to improve end-to-end latency in case of slow or unresponsive replica servers.

If we also require write-availability in the case of N servers going down simultaneously:

- The Topic must be configured with replication factor $\geq 2N + 1$.

These also apply for internal topics such as `__consumer_offsets` and `__transaction_state`.

- For total safety, it must be specified what to do when all servers responsible for a given Partition become unavailable. Setting *unclean.leader.election.enable = false* is a must for deployments prioritizing safety over availability.

Message Ordering

Care must be taken to pick a key-scheme that will ensure proper ordering. All messages using the same key will end up on the same Partition and will therefore be consumed in the same relative order.

Producers must ensure that the messages arrive at the Servers in the intended order, for that reason:

- Producers must be configured using `enable.idempotence=true` or `max.in.flight.requests.per.connection=1`. The former is preferred.

Additionally:

- Topics should be over-partitioned in order to not have to change the number of Partitions and therefore break the existing Key-Partition mapping.

Regarding compaction,

- Keys are used in order to perform compaction, they effectively act as entity identifiers and this should be taken into account when deciding on the key-scheme that will be used.

Delivery Guarantees and Effectively-Once-Semantics

- In order to ensure that messages will be delivered at least once, all safety-related configurations specified above must be set. Furthermore, consumers must manually commit their offsets after they process messages, without relying on default auto-commit behavior.
- Large consume-transform-produce pipelines can be built with the guarantee that each input message will only result in a single consumable output message per pipeline endpoint, and therefore achieve so called "Effectively-Once Processing" semantics. Transformation steps should be free of observational side effects since they may run multiple times, the only guarantee is regarding Kafka-based message consumption and production. Due to this fact, this capability is not as useful for inter-service communication as much as general event-based-processing. Care should be taken to properly choose `transactional.id`. A good approach is '`<group id>.<topic>.<partition>`. The mechanism requires enabling transactions in both Consumers and Producers. Consumers must also piggyback on Producers' transactions in order to commit offsets. Kafka Streams is a useful tool for implementing such pipelines while coordinating all these details.

Infrastructure and Topic Sizing

- Adding Servers reduces per-server network and storage requirements. See (4, 5) for concrete network and storage requirements. Also see (3) for estimating maximum topic throughput.
- Increasing Partitions up to the number of Servers, increases write-throughput. RAID setups allow for increasing number of partitions further while still being able to see write-throughput improvements.
- Increasing Partitions allows for the introduction of Consumers within a Consumer Group, enabling read-throughput scaling. See (9) for estimating the minimum Partitions required for a target throughput.
- Increasing Partitions affects worst-case unavailability and end-to-end latency. See (10, 11) for upper-bound estimations.
- Choosing a multiple of cluster sizes as the initial Partition count allows for uniform distribution of Partitions while scaling. See keeping load uniform for details. 36 or 60 are good default Partition counts.

7.3 Further Work

While the generic mathematical model presented in Apache Kafka Modeling, Cluster and Topic Sizing is not unique to this thesis, we are not aware of any experimental verification of it in available literature.

The derived upper partition limits and their relation to end-to-end latency and worst case unavailability, while straightforwardly derived from the aforementioned model and known operational behavior of Apache Kafka, have also not been experimentally verified.

The partition count guidelines, regarding uniform load while scaling, have also not been verified as being significant for real world use cases. We can imagine how imbalances can cancel out with a large amount of Topics. There is also the issue of partitions being inherently unbalanced due to data skewing. A dynamic partition monitoring and reassignment approach may be more effective in pursuing.

List of Figures

References

[1] "An Overview of Microservices Architecture." [Online]. Available: https://khoadinh.github.io/2015/05/01/microservices-architecture-overview.html. [Accessed: 03-Jul-2022].

[2] S. Newman, *Building Microservices*. "O'Reilly Media, Inc.", 2021.

[3] Martin Fowler, "Microservices." [Online]. Available: https://martinfowler.com/articles/microservices.html. [Accessed: 11-Jun-2022].

[4] C. Richardson, *Microservices Patterns: With examples in Java*. Simon and Schuster, 2018.

[5] N. M. Josuttis, *SOA in Practice: The Art of Distributed System Design*. "O'Reilly Media, Inc.", 2007.

[6] Martin Fowler, "MonolithFirst." [Online]. Available: https://martinfowler.com/bliki/MonolithFirst.html. [Accessed: 17-Jun-2022].

[7] Thomas Betts, "Modular Monolithic Architecture, Microservices and Architectural Drivers." [Online]. Available: https://www.infoq.com/news/2020/01/monolith-architectural-drivers/. [Accessed: 17-Jun-2022].

[8] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," *IEEE Softw.*, vol. 33, no. 3, pp. 42–52, May 2016.

[9] "Getting Started with Microservice Architecture – Challenges and Considerations – Learn With Mainul." [Online]. Available: https://learnwithmainul.com/2021/02/21/getting-started-with-microservice-architecture/. [Accessed: 03-Jul-2022].

[10] M. L. Abbott and M. T. Fisher, *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. Addison-Wesley Professional, 2015.

[11] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov, "Microservices: The Journey So Far and Challenges Ahead," *IEEE Softw.*, vol. 35, no. 3, pp. 24–35, May 2018.

[12] G. Hohpe, B. Woolf, K. Brown, J. Crupi, and M. Fowler, *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley, 2004.

[13] EdPrice-MSFT, "Asynchronous Request-Reply Pattern - Azure Architecture Center." [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/async-request-reply. [Accessed: 01-Jul-2022].

[14] B. Stopford, *Designing Event-Driven Systems*. O'Reilly Media, Incorporated, 2018.

[15] alexbuckgit, "Publisher-Subscriber pattern - Azure Architecture Center." [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber. [Accessed: 01-Jul-2022].

[16] E. Koutanov, *Effective Kafka: A Hands-On Guide to Building Robust and Scalable Event-Driven Applications with Code Examples in Java*. Amazon Digital Services LLC - KDP Print US, 2020.

[17] G. Shapira, T. Palino, R. Sivaram, and K. Petty, *Kafka: The Definitive Guide*. "O'Reilly Media, Inc.", 2021.

[18] "Designing Data-Intensive Applications [Book]." [Online]. Available: https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/. [Accessed: 11-Jun-2022].

[19] "AMQP 0-9-1 Model Explained — RabbitMQ." [Online]. Available: https://www.rabbitmq.com/tutorials/amqp-concepts.html. [Accessed: 01-Jul-2022].

[20] "RabbitMQ tutorial - Work Queues — RabbitMQ." [Online]. Available: https://www.rabbitmq.com/tutorials/tutorial-two-python.html. [Accessed: 01-Jul-2022].

[21] "Notes from the Architect — Varnish version trunk documentation." [Online]. Available: https://varnish-cache.org/docs/trunk/phk/notes.html. [Accessed: 01-Jul-2022].

[22] "Replicated Log." [Online]. Available: https://martinfowler.com/articles/patterns-of-distributed-systems/replicated-log.html. [Accessed: 01-Jul-2022].

[23] "Command and Query Responsibility Segregation (CQRS)." [Online]. Available: https://gowie.eu/index.php/patterns/32-architecture/patterns/105-command-and-query-responsibility-segregation-cqrs. [Accessed: 03-Jul-2022].

[24] "Bliki: CQRS." [Online]. Available: https://martinfowler.com/bliki/CQRS.html. [Accessed: 13-Jun-2022].

[25] R. Abdullin, "CQRS Documents by Greg Young," p. 56.

[26] "Event Sourcing." [Online]. Available: https://martinfowler.com/eaaDev/EventSourcing.html. [Accessed: 13-Jun-2022].

[27] "Event sourcing, CQRS, stream processing and Apache Kafka: What's the connection? | Confluent." [Online]. Available: https://www.confluent.io/blog/event-sourcing-cqrs-stream-processing-apache-kafka-whats-connection/. [Accessed: 01-Jul-2022].

[28] P. Helland, "Data on the outside versus data on the inside," *Commun. ACM*, vol. 63, no. 11, pp. 111–118, Oct. 2020.

[29] "Turning the database inside-out with Apache Samza." [Online]. Available: https://www.confluent.io/blog/turning-the-database-inside-out-with-apache-samza/. [Accessed: 01-Jul-2022].

[30] P. Mahajan, L. Alvisi, and M. Dahlin, "Consistency, Availability, and Convergence," p. 31.

[31] "Eventually Consistent - ACM Queue." [Online]. Available: https://queue.acm.org/detail.cfm?id=1466448. [Accessed: 13-Jun-2022].

[32] "A Brief History of Kafka, LinkedIn's Messaging Platform - insideBIGDATA." [Online]. Available: https://insidebigdata.com/2016/04/28/a-brief-history-of-kafka-linkedins-messaging-platform/. [Accessed: 01-Jul-2022].

[33] "Kafka Design | Confluent Documentation." [Online]. Available: https://docs.confluent.io/platform/current/kafka/design.html#push-vs-pull. [Accessed: 01-Jul-2022].

[34] "Apache Kafka - PoweredBy." [Online]. Available: https://kafka.apache.org/powered-by. [Accessed: 01-Jul-2022].

[35] "Kafka Security Best Practices Checklist to Securing your Server," 30-May-2022. [Online]. Available: https://cloudinfrastructureservices.co.uk/kafka-security-best-practices-checklist-to-securing-your-server/. [Accessed: 03-Jul-2022].

[36] *Apache Kafka - LogCleanerManager*. The Apache Software Foundation, 2022.

[37] "Kafka/KafkaServer.scala at 6bf5bfc2982158c3a1bfff4a0f65ea901ea84e7a · apache/kafka." [Online]. Available: https://github.com/apache/kafka/blob/6bf5bfc2982158c3a1bfff4a0f65ea901ea84e7a/core/src/main/scala/kafka/server/KafkaServer.scala. [Accessed: 01-Jul-2022].

[38] "Kafka/ReplicaManager.scala at 6bf5bfc2982158c3a1bfff4a0f65ea901ea84e7a · apache/kafka." [Online]. Available: https://github.com/apache/kafka/blob/6bf5bfc2982158c3a1bfff4a0f65ea901ea84e7a/core/src/main/scala/kafka/server/ReplicaManager.scala. [Accessed: 01-Jul-2022].

[39] "Kafka/LogManager.scala at 6bf5bfc2982158c3a1bfff4a0f65ea901ea84e7a · apache/kafka." [Online]. Available: https://github.com/apache/kafka/blob/6bf5bfc2982158c3a1bfff4a0f65ea901ea84e7a/core/src/main/scala/kafka/log/LogManager.scala. [Accessed: 01-Jul-2022].

[40] "KIP-392: Allow consumers to fetch from closest replica - Apache Kafka - Apache Software Foundation." [Online]. Available: https://cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replica. [Accessed: 01-Jul-2022].

[41] "Hands-Free Kafka Replication: A Lesson in Operational Simplicity." [Online]. Available: https://www.confluent.io/blog/hands-free-kafka-replication-a-lesson-in-operational-simplicity/. [Accessed: 01-Jul-2022].

[42] R. Pozzi, "Using sensors with Raspberry boards Part 2: Connecting and sending data from a DHT11 sensor," 08-Sep-2021. [Online]. Available: https://robertopozzi.medium.com/using-sensors-with-raspberry-boards-part-2-connecting-and-sending-data-from-a-dht11-sensor-14691b70bab1. [Accessed: 03-Jul-2022].

[43] P. Brebner, "Exploring the Apache Kafka® 'Castle' Part B: Event Reprocessing," 18-Jan-2018. [Online]. Available: https://www.instaclustr.com/blog/exploring-apache-kafka-castle-event-reprocessing/. [Accessed: 03-Jul-2022].

[44] "Kafka Topic Replication | Learn Apache Kafka with Conduktor," 14-Jan-2022. [Online]. Available: https://www.conduktor.io/kafka/kafka-topic-replication. [Accessed: 03-Jul-2022].

[45] ACCU, "Kafka Acks Explained." [Online]. Available: https://accu.org/journals/overload/28/159/kozlovski/. [Accessed: 01-Jul-2022].

[46] J. E. Q. Otoya, "Use min.insync.replicas for fault-tolerance." [Online]. Available: https://jeqo.github.io/til/2021-12-02-kafka-min-isr/. [Accessed: 01-Jul-2022].

[47] "Jepsen: Kafka." [Online]. Available: https://aphyr.com/posts/293-jepsen-kafka. [Accessed: 01-Jul-2022].

[48] "Apache Kafka Fundamentals," 18-Nov-2020. [Online]. Available: https://grapeup.com/blog/apache-kafka-fundamentals/. [Accessed: 03-Jul-2022].

[49] D. Newton, "Intro to Kafka - Consumer groups," 20-Jun-2021. [Online]. Available: https://lankydan.dev/intro-to-kafka-consumer-groups. [Accessed: 03-Jul-2022].

[50] G. Jansen, K. S. P. April 21, and 2020, "Configuring Kafka for reactive systems." [Online]. Available: https://developer.ibm.com/articles/configuring-kafka-for-reactive-applications/. [Accessed: 03-Jul-2022].

[51] "Apache Kafka's Delivery Guarantees." [Online]. Available: https://adservio.fr/post/apache-kafkas-delivery-guarantees. [Accessed: 03-Jul-2022].

[52] "Idempotent Kafka Producer | Learn Apache Kafka with Conduktor," 17-Jan-2022. [Online]. Available: https://www.conduktor.io/kafka/idempotent-kafka-producer. [Accessed: 03-Jul-2022].

[53] "Transactions in Apache Kafka | Confluent." [Online]. Available: https://www.confluent.io/blog/transactions-apache-kafka/. [Accessed: 03-Jul-2022].

[54] "Best practices for right-sizing your Apache Kafka clusters to optimize performance and cost | AWS Big Data Blog." [Online]. Available: https://aws.amazon.com/blogs/big-data/best-practices-for-right-sizing-your-apache-kafka-clusters-to-optimize-performance-and-cost/. [Accessed: 01-Jul-2022].

[55] ©. 2021. Cloudera, I. A. rights reserved A. Hadoop, associated open source project names are trademarks of the A. S. F. F. a complete list of trademarks, and C. Here, "Kafka Cluster Sizing | 6.3.x | Cloudera Documentation." [Online]. Available: https://docs.cloudera.com/documentation/enterprise/6/6.3/topics/kafka_performance_cluster_sizing.html. [Accessed: 01-Jul-2022].

[56] "How to Choose the Number of Topics/Partitions in a Kafka Cluster? | Confluent." [Online]. Available: https://www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster/. [Accessed: 01-Jul-2022].

[57] "Choosing the number of partitions for a topic | CDP Public Cloud." [Online]. Available: https://docs.cloudera.com/runtime/7.2.10/kafka-performance-tuning/topics/kafka-tune-sizing-partition-number.html. [Accessed: 01-Jul-2022].

[58] "Why ZooKeeper Was Replaced with KRaft - The Log of All Logs | Confluent Why Replace ZooKeeper with Kafka Raft - The Log of All Logs." [Online]. Available: https://www.confluent.io/blog/why-replace-zookeeper-with-kafka-raft-the-log-of-all-logs/. [Accessed: 01-Jul-2022].