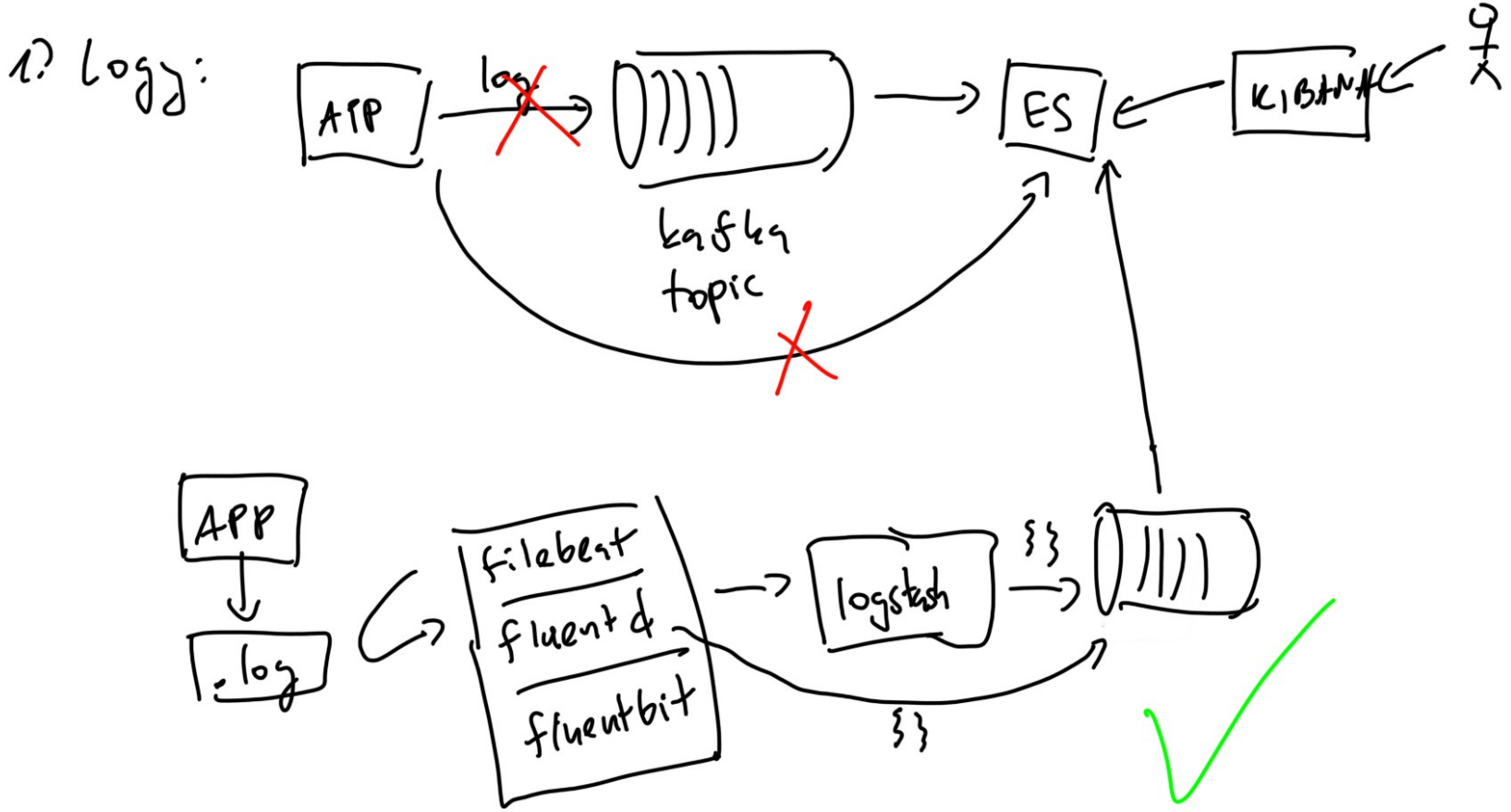
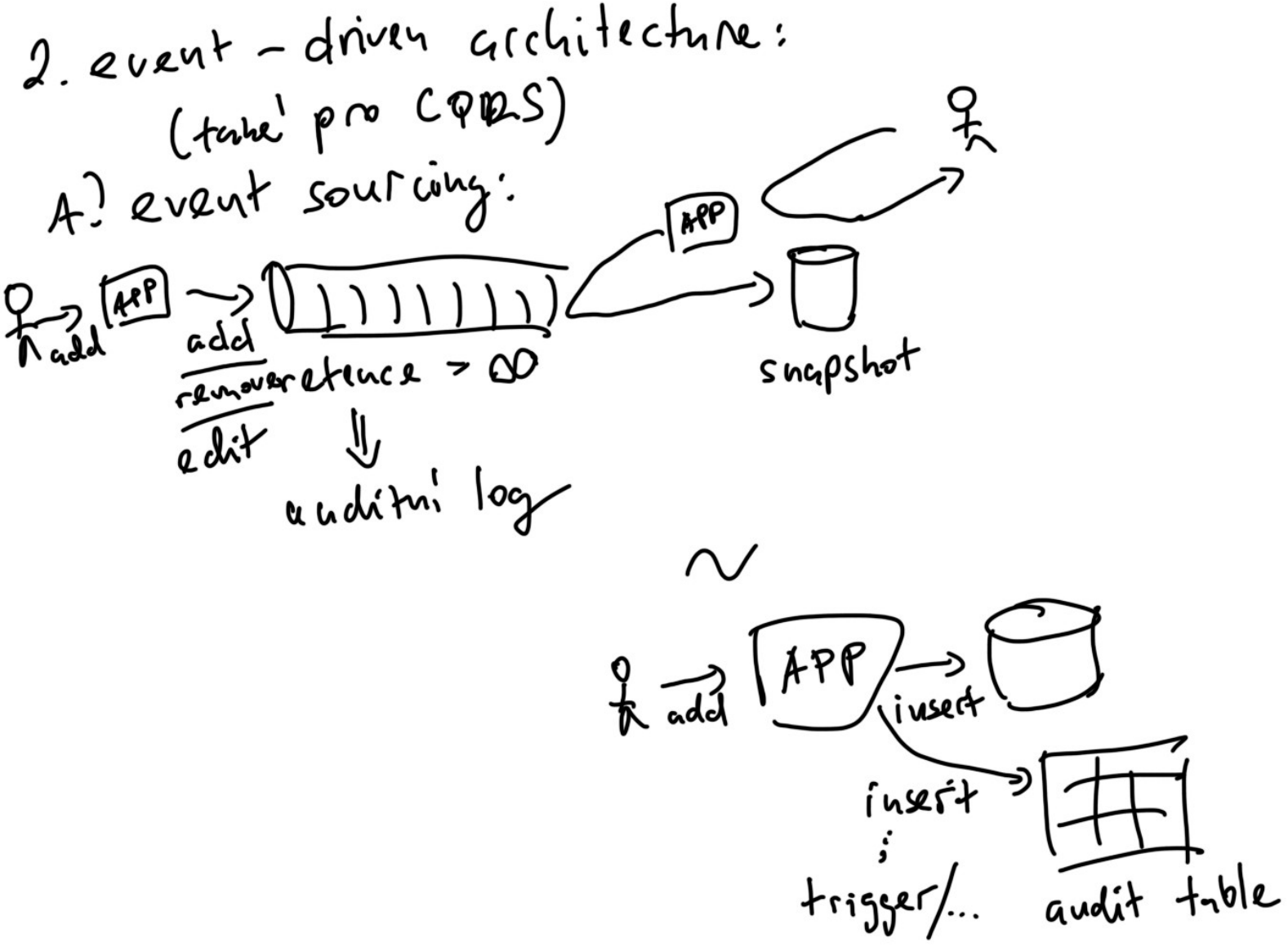
Apache Kafka

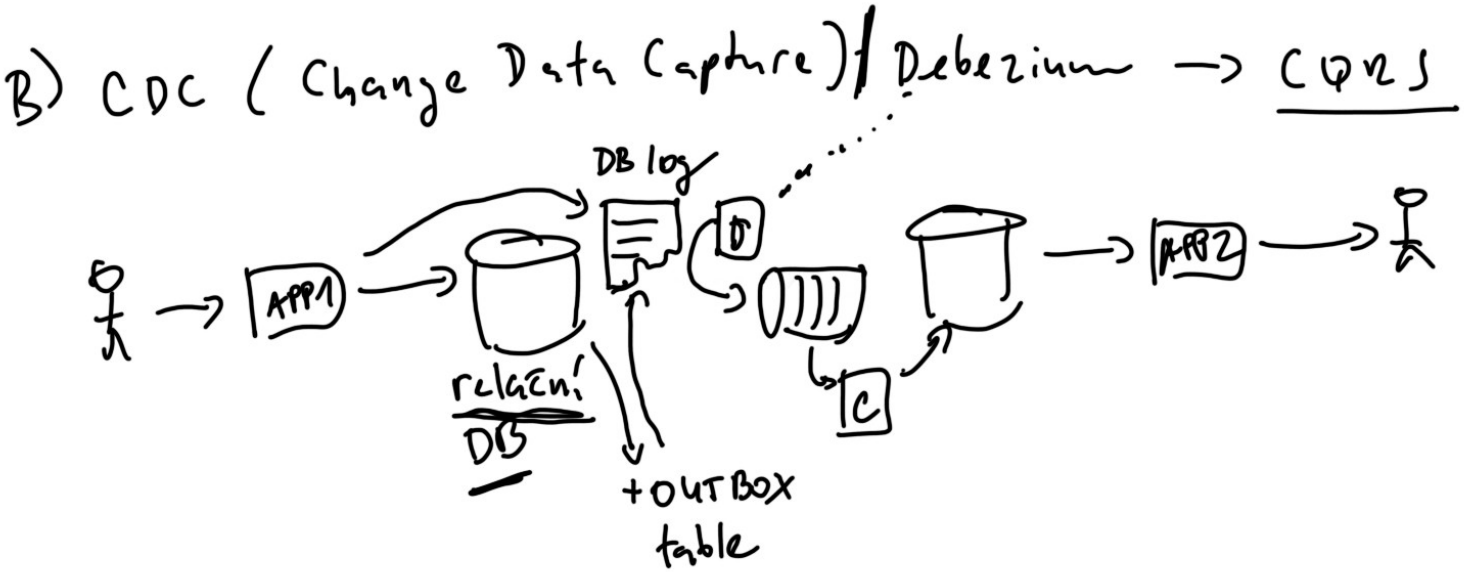
Kafka: use-cases I.: logy

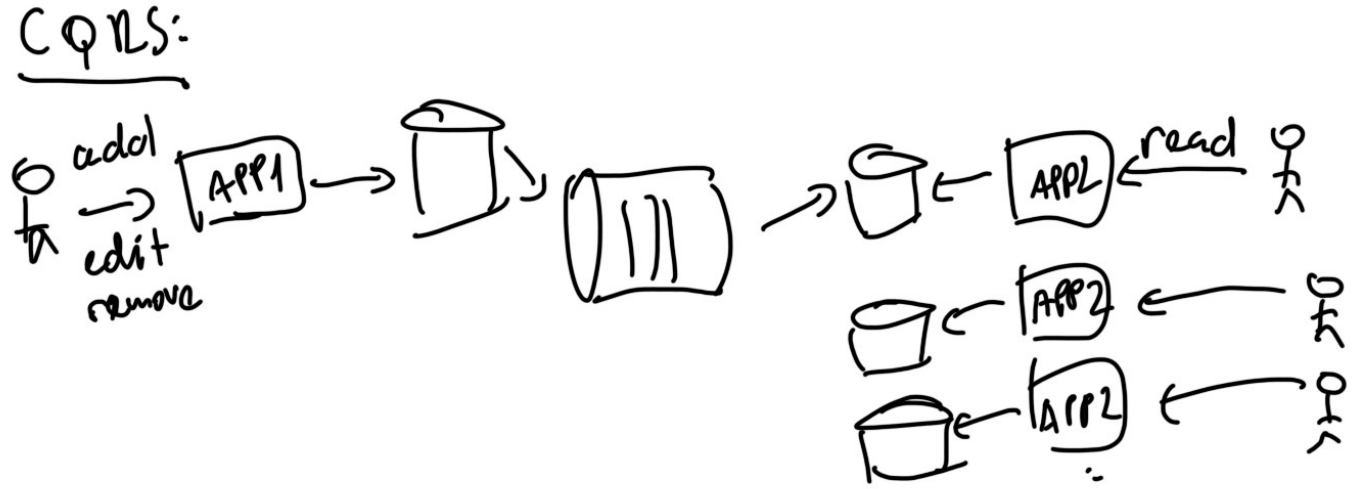


Kafka: use-cases II.: event sourcing

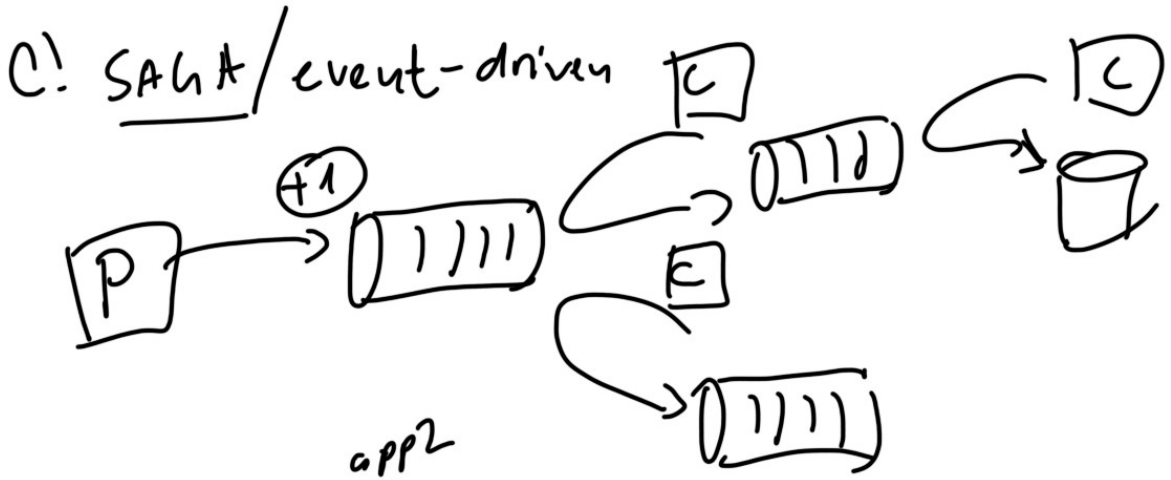


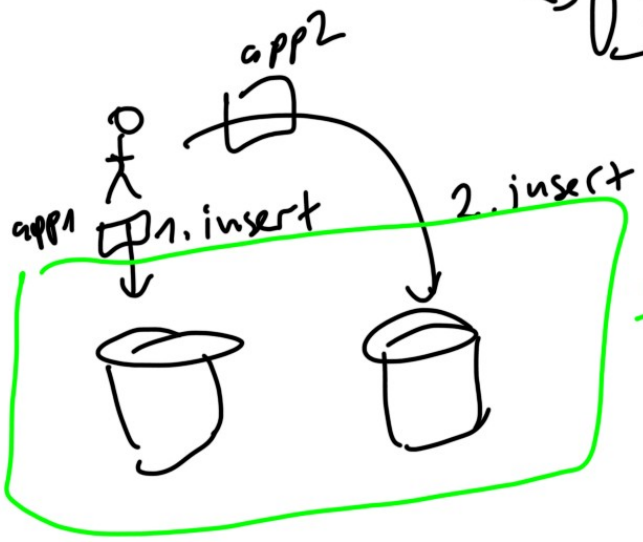
Kafka: use-cases III.: CDC & CQRS





Kafka: use-cases IV.: SAGA

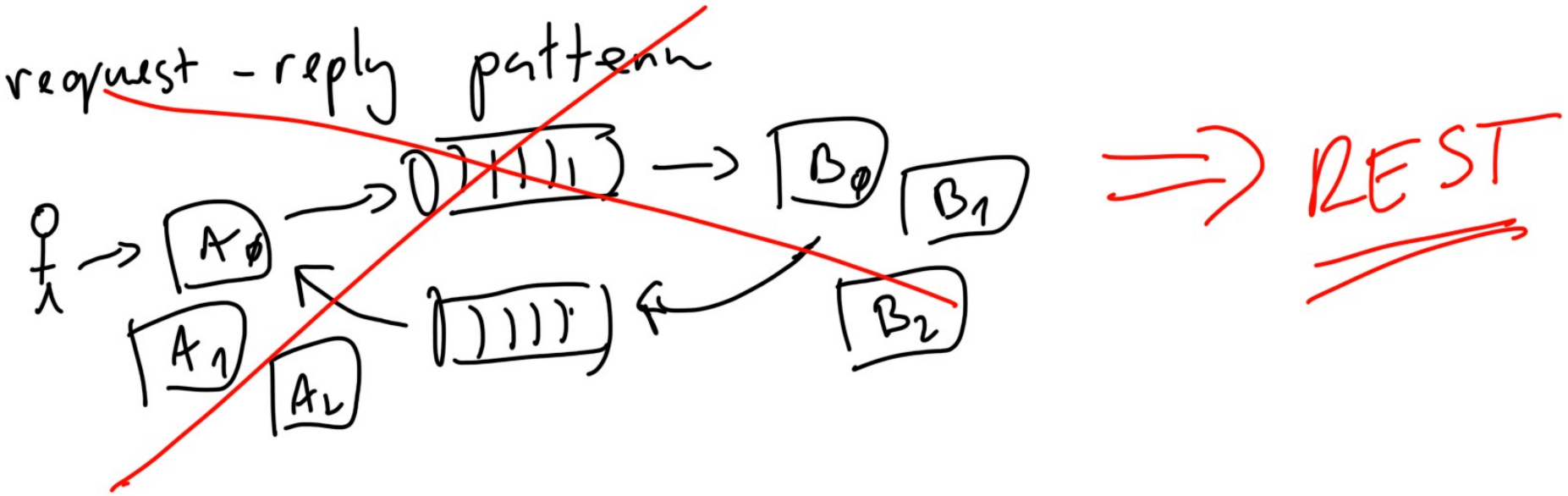




transakce → "distribuovaná transakce"

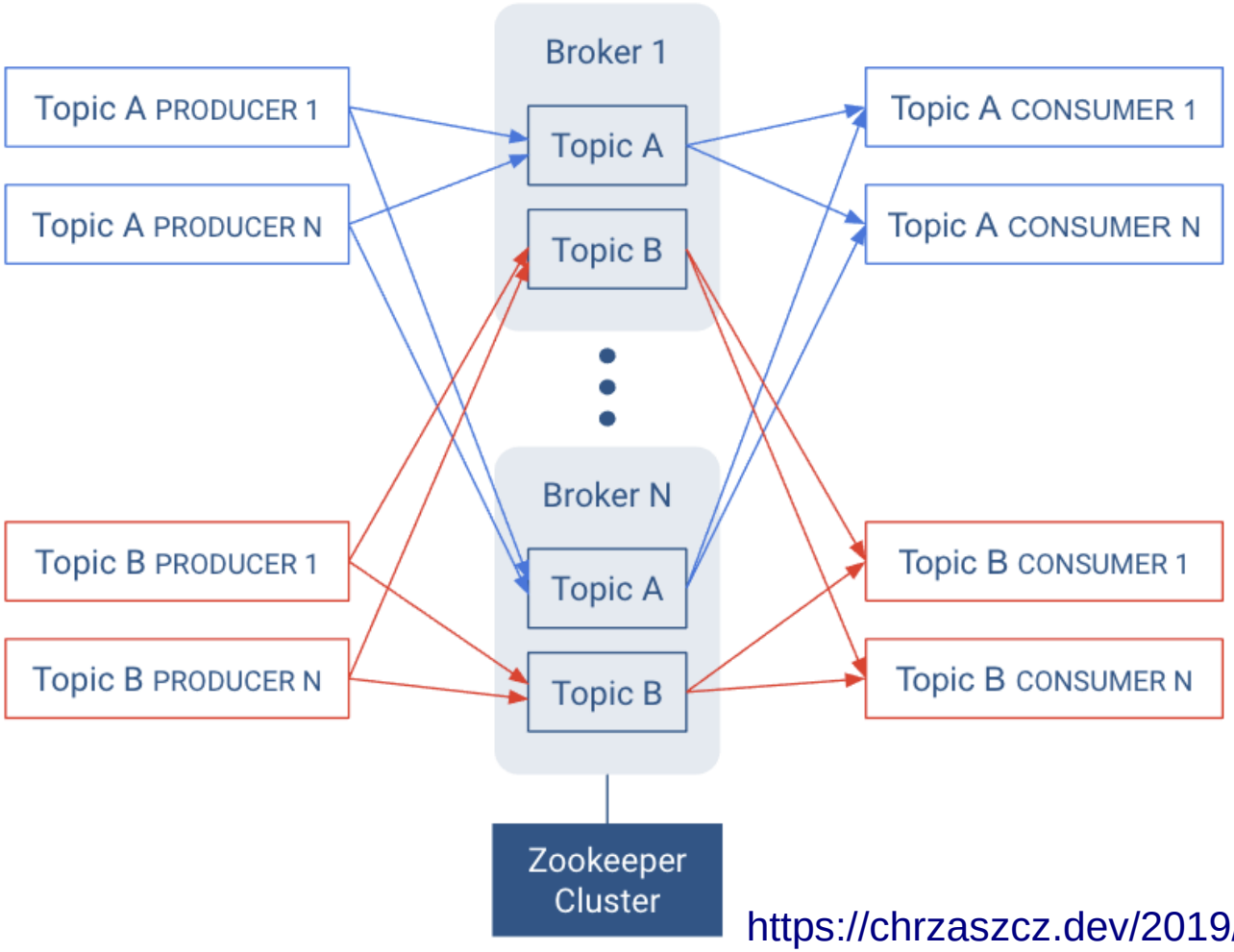
FyI: SAGA!

Kafka se nehodí na request-reply!!!



Kafka 101

- Kafka je distribuovaný systém, který umožňuje implementovat asynchronní zprocesování událostí, publish-subscribe mechanismus, nebo asynchronní distribuci práce mezi workery.



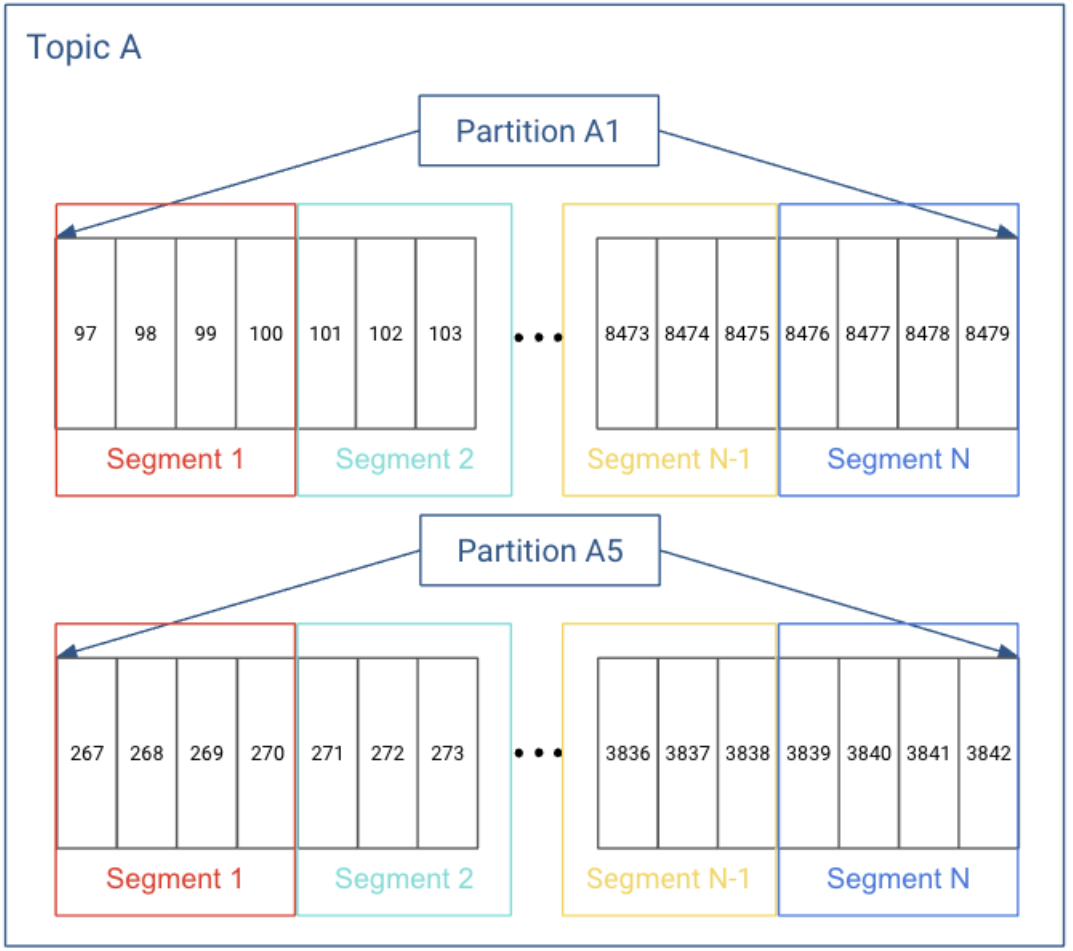
https://chrzaszcz.dev/2019/05/26/kafka-101/

Základní pojmy

- Broker
  - server (onprem, virtual, kontejner), na kterém běží Kafka proces
- Message (Event)
  - Zpráva obsahuje pár (klíč – hodnota)
    - klíč i hodnota mohou být libovolného datového typu – skalární hodnota i složitější objekt jako například JSON.
    - Klíč není něco jako primární klíč u relační databáze, ale například název senzoru, ID / název business operace apod. Může být NULL, ale z řady důvodů je dobrý nápad aby byl klíč přítomen.
    - Zpráva je immutable

Základní pojmy

- Topic
  - Zprávy jsou ukládány do „topic“ (něco jako log)
- Partition
  - Topic se skládá z „partitions“
    - Pořadí zpráv má význam pouze v rámci jedné partition!
      - Každá partition má offsety číslované od nuly!
- Segment
  - Data se neukládají do jednoho velkého souboru, ale jsou uložené v log segmentech (to je na disku soubor)



Pěkně zpracované kam přesně na disku Kafka ukládá data: https://rohithsankepally.github.io/Kafka-Storage-Internals/

Základní pojmy

- Když je key == null, pak se message vkládají do partitions pomocí round robin algoritmu. Pokud message mají key, pak je garantované, že message se stejným key budou ve stejné partition a také budou seřazené.
  - Interně se z klíče získá hash a na jeho základě se ta zpráva ukládá do příslušné partition. Hashování a zařazení do příslušné partition má na starosti producer.
- Topic is append only, can only seek by offset. Topics are durable, retention is configurable.
  - Výchozí retence Topicu je časová (7 dní), dá se změnit. Retence může být také na základě velikosti Topicu
    - https://medium.com/@sunny_81705/kafka-log-retention-and-cleanup-policies-c8d9cb7e09f8
- Replication
  - jedna partition je lead replica, ostatní jsou follow repliky. Replication factor má na starosti počet replik, výchozí je 1 (tzn. vypnutá replikace, data jsou jenom v lead replice)

Cluster

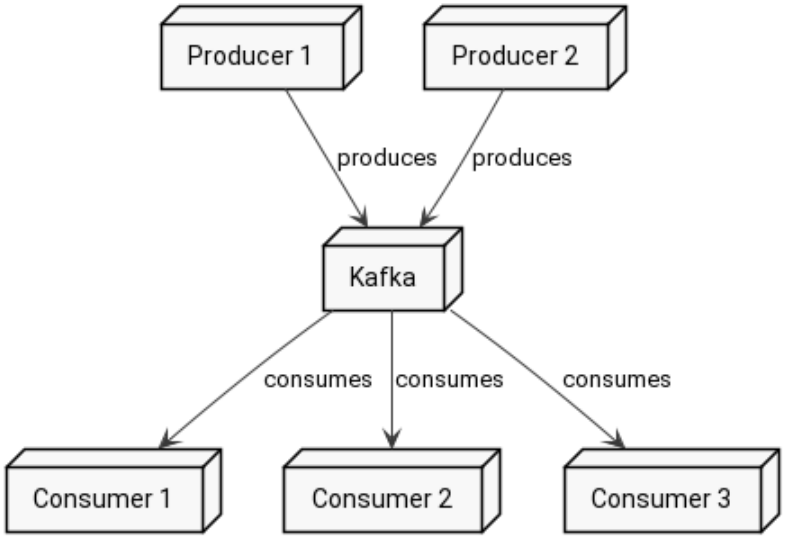
- Kafka používá Apache Zookeeper pro udržování listu brokerů, kteří jsou aktuálně členy clusteru. Každý broker má unikátní ID, které když není uvedené, tak se při startu brokera automaticky vygeneruje.
- Jak získat seznam všech aktivních brokerů v clusteru:
  - zookeeper-shell localhost:2181 ls /brokers/ids
- Informace o brokeru s KAFKA_BROKER_ID = 10 v clusteru:
  - zookeeper-shell localhost:2181 get /brokers/ids/10
- https://www.baeldung.com/ops/kafka-list-active-brokers-in-cluster
- Controller
  - První z Kafka brokerů, který se připojí do clusteru je tzv. Controller a má na starosti přidělování partition leaderů. Když Controller přestane fungovat, pak se jiný fungující broker stane Controllerem.

Replikace

- Jsou dva typy replik:
  - Leader replica
    - Každá partition má jednu repliku, která je typu leader replica. Všechny požadavky producerů i consumerů jdou zkrz leadera kvůli zajištění konzistence dat.
  - Follower replica
    - Všechny ostatní repliky jsou typu followeři. Followeři nepracují s požadavky klientů, jejich jediná práce je replikovat zprávy od leadera a zůstat up-to-date se stavem leadera. Když leader replica crashne, pak se jedna z follower replik stane novým leaderem.
- Leader replica má ještě jednu roli a to je kontrolovat jestli follower repliky jsou in-sync. Repliky nemusí být in-sync například když je problém na síti, nebo když vypadne broker a data ještě nejsou zreplikovány na jiné brokery.
- Kafka garantuje, že každá replika je na jiném brokerovi, tudíž maximální hodnota replication factoru se rovná počtu brokerů.

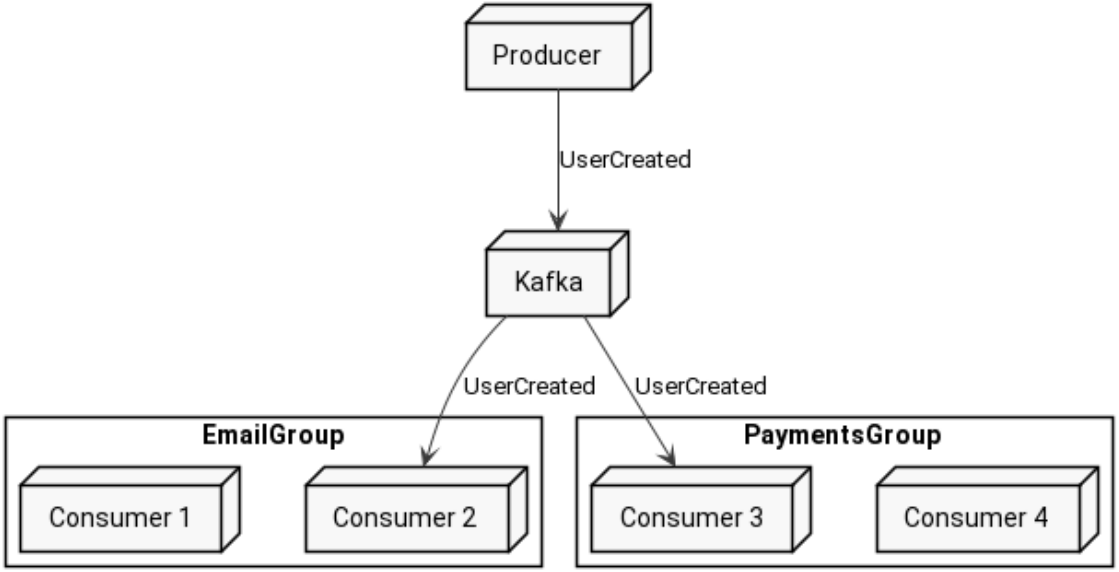
Základní pojmy

- Producer
  - Producer slouží k produkování zpráv do topiců
- Consumer
  - Consumer slouží k získávání zpráv z topicu
    - Consumers mohou být sdruženi do skupin (pomocí group.id).
    - Consumer neodebírá zprávu z topicu (velký rozdíl oproti message-queue), ale udržuje si offset, pomocí kterého ví, od jaké zprávy má číst dál.
      - Offsety jsou v dnešních verzích Kafky uložené v topicu __consumer_offsets

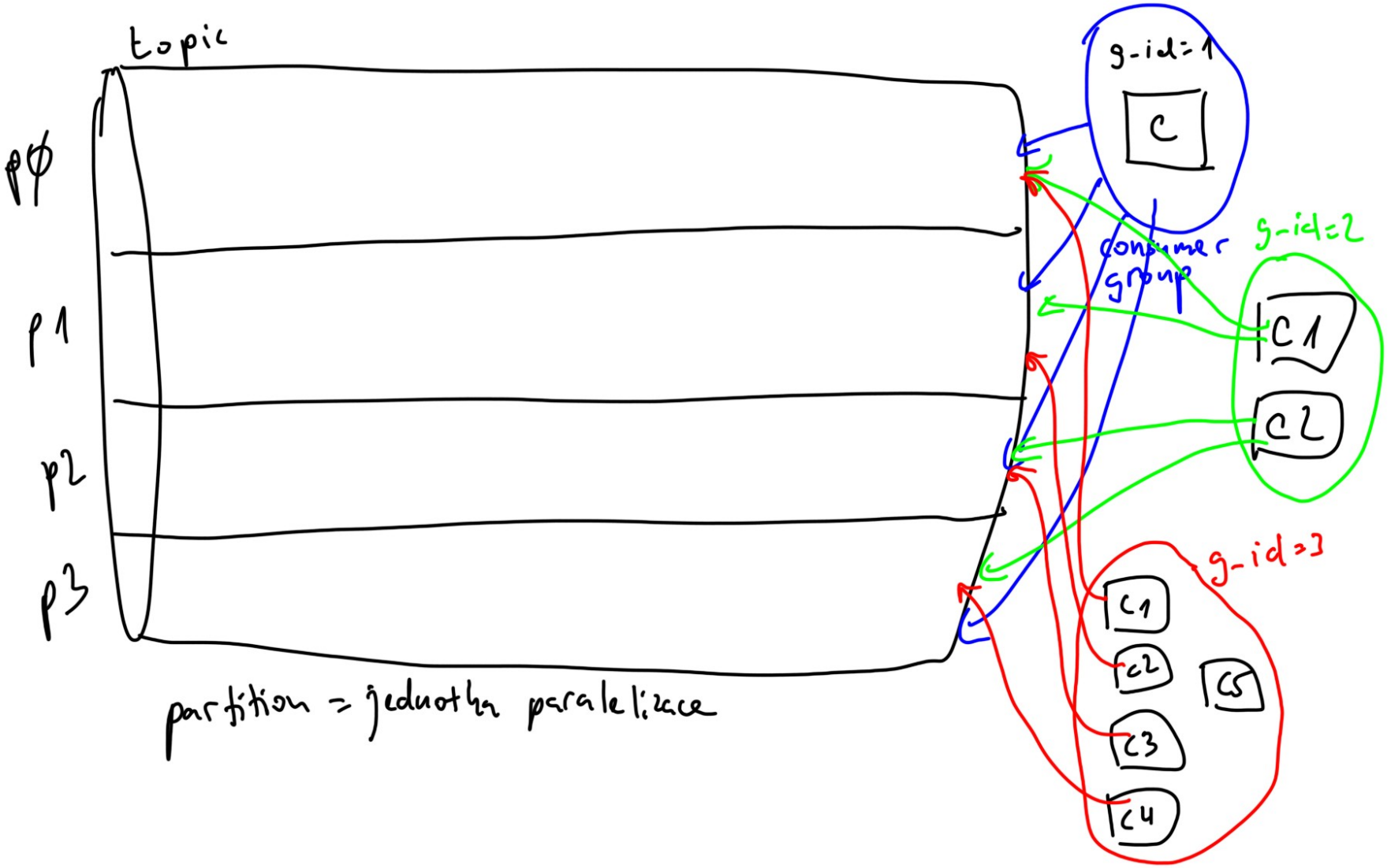


Consumer Group

- Consumer Group
  - Může být víc consumerů, kteří mají stejné group.id (to je pak tzv. consumer group) a tak funguje horizontální škálování consumerů. Kafka bude automaticky rovnoměrně distribuovat zátěž mezi tyto consumery. Consumerů se stejným group.id nemůže být víc než partition instancí. Z jednoho topicu může číst data X consumerů, kteří mají rozdílná group.id.
    - Každá zpráva bude poslána do každé Consumer group, ale uvnitř té Consumer group bude poslána pouze jednomu Consumeru.

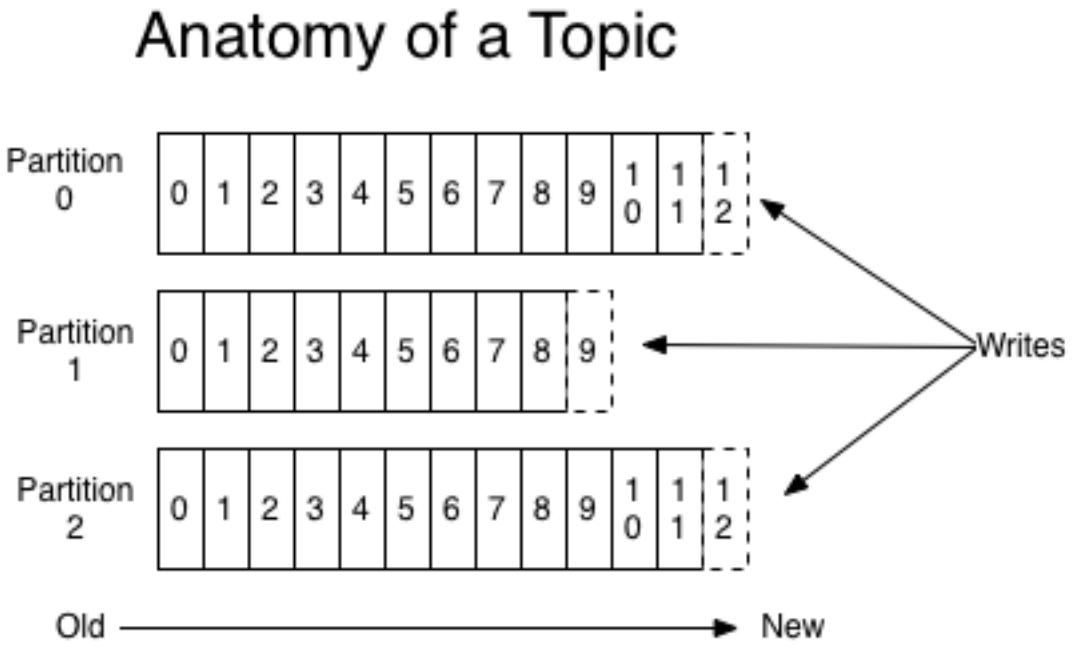


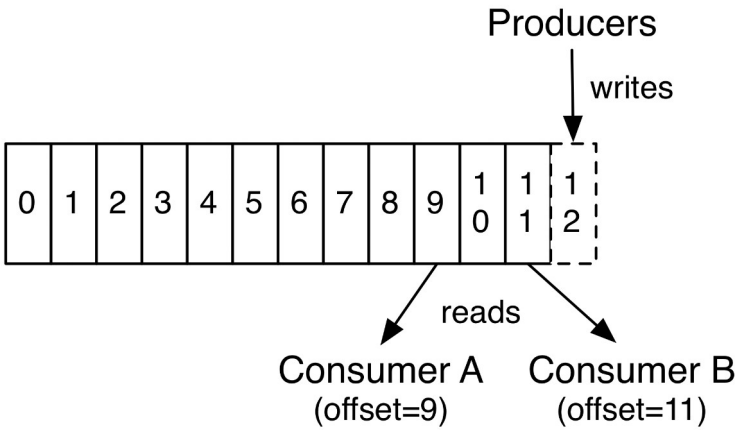
Consumer Group & Partition



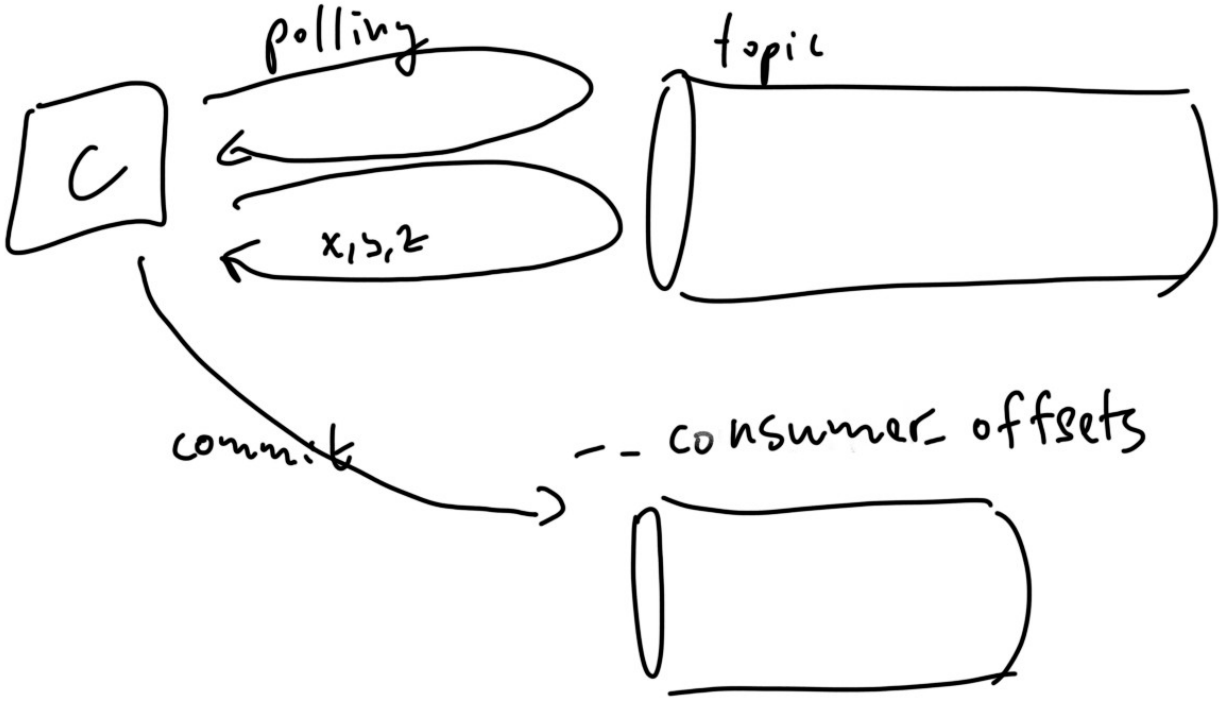
Offset

- Každá zpráva při uložení do Partition dostane offset. První zpráva má offset 0, druhá 1 atd.
- Každá zpráva může být unikátně identifikována pomocí kombinace: (topic_name, partition_number, offset)
- Consumer používá offset pro specifikování pozice v logu. Jsou dva druhy offsetů. Jeden je uložen v Kafce (commited offset) a druhý je lokální a consumer ho používá pro polling (consumer position).





At-least-once delivery



V kafce je at-least-once delivery!

Commited Offset vs. Consumer Position

- Commited Offset
  - Ukládá se do Kafky při commitu v Consumeru.
  - Využije se když Consumer crashne.
  - Příklad:
    - Consumer C1 právě začal konzumovat nový topic. Provedl fetch deseti záznamů od offsetu 0. Po jejich zprocesování chce dát vědět Kafce informaci, že je zprocesoval. Tento proces se nazývá „commitování“. Takže provede commit. Poté C1 crashne. Následně se nastartuje Consumer C2, který se podívá do Kafky co bylo naposledy zprocesováno a začne procesovat záznamy od desátého záznamu.

Commited Offset vs. Consumer Position

- Consumer Position
  - Consumer normálně provádí polling (to je jeho činnost). Při pollingu se ale automaticky neprovádí commit. Ten se provádí buď periodicky při Xtém zavolání poll metody, nebo manuálně zavoláním metody commit.
  - Na pozadí si během pollingu Consumer pamatuje aktuální offset posledního zprocesovaného záznamu v něčem, co má název Consumer Position.
  - Consumer Position se dá změnit pomocí volání metod:
    - consumer.seek(somePartition, newOffset)
    - consumer.seekToBeginning(somePartitions)
    - consumer.seekToEnd(somePartitions)

Initial Offset

- Když se přidá úplně nová Consumer Group, tak nemá žádný commited offset. Jaký se použije výchozí? Musíme ho zvolit:
  - Earliest
    - Použije se nejmenší (nejstarší) dostupný offset
    - V Consumeru je nutné nastavit auto.offset.reset=earliest
  - Latest
    - Použije se největší (nejnovější) dostupný offset
    - Tohle je default
  - None
    - Konkrétní offset musíme specifikovat manuálně
- Pozor! V Brokeru je nastavení offsets.retention.minutes (ve výchozím nastavení 24h), což znamená, že když Consumer Group vypadne na víc jak 24h a je nastaven latest offset, pak bude zprocesovávat záznamy, které přišly až po spuštění Consumera (čili se vytvoří initial offset) a tudíž dojde ke ztrátě dat.
  - https://dzone.com/articles/apache-kafka-consumer-group-offset-retention

Kafka vs. RabbitMQ

- Tradičně RabbitMQ je message broker, který slouží pro doručování zpráv (takový pošťák), zatímco Kafka je distribuovaný log. V dnešní době nicméně nabízí víceméně to samé (zejména od vydání RabbitMQ 3.9, od kdy obsahuje novou datovou strukturu Streams).
- V současnosti bych viděl největší rozdíly zejména v ekosystému okolo (v Kafce jsou navíc Kafka Streams, Kafka Connect a Debezium, ksqlDB atd.).
- Z pohledu výkonu má Kafka vyšší throughput, zatímco RabbitMQ má nižší latenci doručení zpráv.
- https://qr.ae/pG0g2J
- https://tanzu.vmware.com/developer/blog/understanding-the-differences-between-rabbitmq-vs-kafka/

Hello World Kafka (landoop)

- Spuštění Apache Kafka:

```
docker run --rm -it -p 2181:2181 -p 3030:3030 \
-p 8081:8081 -p 8082:8082 -p 8083:8083 -p 9092:9092 \
-e ADV_HOST=127.0.0.1 landoop/fast-data-dev
```

- Spuštění command-line nástrojů:

```
docker run --rm -it --net=host landoop/fast-data-dev bash
```

- Dashboard:
  - http://localhost:3030/

Hello World Kafka (confluent)

- Spuštění Apache Kafka:

```
git clone https://github.com/confluentinc/cp-all-in-one
cd cp-all-in-one/cp-all-in-one
docker compose up -d
```

- Spuštění command-line nástrojů:

```
docker run -it --rm --net=host \
confluentinc/cp-zookeeper:5.0.0-beta30 bash
```

- Dashboard:
  - http://localhost:9021

- https://docs.confluent.io/platform/current/quickstart/ce-docker-quickstart.html

Security

- Confluent UI servery (ZooKeeper, Kafka brokers, Kafka Connect, …) je možné zabezpečit několika způsoby (SSL, autorizace & autentizace):
  - https://docs.confluent.io/platform/current/security/security_tutorial.html

kafka-topics

- Vytvoří topic:

```
kafka-topics --zookeeper localhost:2181 --create \
--topic first_topic --partitions 3 --replication-factor 1
```

- Zobrazí seznam topiců:

```
kafka-topics --zookeeper localhost:2181 --list
```

- Podrobnější informace o vybraném topicu:

```
kafka-topics --zookeeper localhost:2181 --describe --topic first_topic
```

- Smaže topic:

```
kafka-topics --zookeeper localhost:2181 --delete --topic first_topic
```

Kafka Broker Configuration

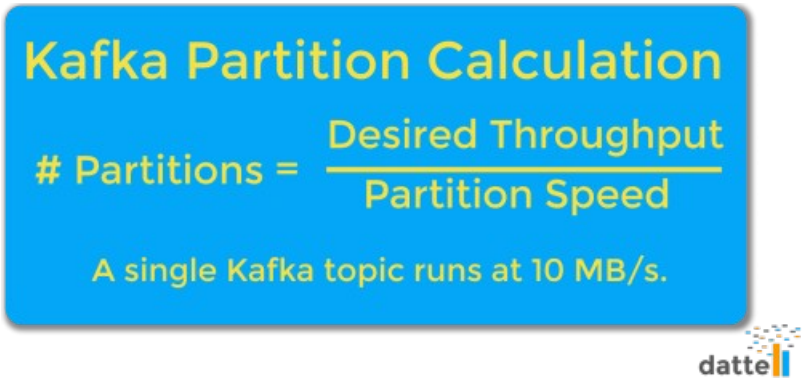
- Je dobré nastavit / změnit následující výchozí konfiguraci Kafka brokeru:
  - auto.create.topics.enable (výchozí hodnota: true)
    - Není best practice aby aplikace automaticky vytvářela Topic při jeho prvním použití zejména z toho důvodu, že každý Topic může mít odlišnou konfiguraci:
    - log.retention.hours (default: 7 dní): Retence dat v Kafce
    - min.insync.replicas (default: 1): Když má Producer acks = all, min.insync.replicas specifikuje minimální počet replik, které vrátí informaci, že úspěšně zapsaly záznam. Správná hodnota je (replication-factor – 1)
    - replication.factor (default: 1): Počet replik topiku. Hodnota záleží na tom, kolik chceme mít replik Topiců.
    - num.partitions (default: 1): Počet partitions. Hodnota určuje paralelismus.
  - offsets.retention.minutes (default: 24 hodin): Po jak dlouhé době se smažou nepoužívané offsety Consumerů v Kafce.
- https://www.javierholguera.com/2018/06/13/kafka-defaults-that-you-should-re-consider-i/

Partitions: Throughput

- Topic Partition je jednotka paralelizace v Kafce. Jak Producer, tak Consumer mohou pracovat s rozdílnými partitions plně paralelním způsobem. Na straně Consumera Kafka dá data z jedné partition vždy jednomu Consumerovi. Tudíž stupeň paralelizace u Consumera (v rámci jedné Consumer Group) je omezen počtem partitions. Čili obecně čím větší množství partitions, tím větší throughput.
- https://www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster/
- Maximální počet partitions v Kafce:
  - Broker: 4 000 partitions
  - Cluster: 200 000 partitions
    - https://blogs.apache.org/kafka/entry/apache-kafka-supports-more-partitions

Number of Partitions

- Kolik mít partitions?
  - Na to jsou různé názory. Nejvíc se mi líbí tento: 10 partitions, nebo podle toho, jaký je požadovaný throughput se dá použít tento vzoreček:



    - https://dattell.com/data-architecture-blog/kafka-optimization-how-many-partitions-are-needed/
  - Zde jsou pěkně vysvětlené důsledky špatného nastavení:
    - https://docs.cloudera.com/runtime/7.2.10/kafka-performance-tuning/topics/kafka-tune-sizing-partition-number.html

Kafka REST Proxy

- Kafka REST Proxy nabízí REST rozhraní pro ovládání Kafky, které dokáže:
  - CRUD operace nad Topicy
  - Jednoduchý producer & consumer
- https://github.com/confluentinc/kafka-rest
- List topiců:
  - http://localhost:8082/topics

Kafka UI

https://towardsdatascience.com/overview-of-ui-tools-for-monitoring-and-management-of-apache-kafka-clusters-8c383f897e80

- kafdrop:
  - https://hub.docker.com/r/obsidiandynamics/kafdrop
- kowl
  - https://github.com/cloudhut/kowl
- akhq
  - https://github.com/tchiotludo/akhq
- kafka-ui
  - https://github.com/provectus/kafka-ui

Programový přístup k topicu

- Programově je možné vytvářet / mazat apod. topic pomocí třídy AdminClient:
  - https://stackoverflow.com/a/45122955/894643
- Jak získat list topiců:
  - KafkaConsumer#listTopics()
  - NEBO:
  - AdminClient#listTopics()

kafka-console-producer

- Jednoduché přidávání zpráv do topicu z konzole:

```
kafka-console-producer --broker-list localhost:9092 --topic first_topic
```

- Poznámky:
  - 1 řádek = 1 zpráva
  - ukončení: CTRL + C
  - zprávy jdou do náhodné partition (protože nespecifikujeme klíč)
  - když přidáme zprávu do neexistujícího topicu, tak se automaticky vytvoří nový topic
  - specifikování klíče:

```
kafka-console-producer --broker-list localhost:9092 --topic first_topic \
--property "parse.key=true" --property "key.separator=:"
```

kafka-console-consumer

- Jednoduché čtení zpráv z topicu (výpis je do konzole):

```
kafka-console-consumer --bootstrap-server localhost:9092 \
--topic first_topic

kafka-console-consumer --bootstrap-server localhost:9092 \
--topic first_topic --from-beginning

kafka-console-consumer --bootstrap-server localhost:9092 \
--topic first_topic --from-beginning --partition 0
```

- Další zajímavé přepínače:

  `--from-beginning` (vypíše všechny zprávy od začátku)

  `--partition X` (vypíše zprávy v partition X)

  `--offset Y` (vypíše zprávy od offset Y)

  `--consumer-property group.id=mygroup1` (uloží poslední navštívený offset a příště čte zprávy od něj dál)

Poznámka: Offset skupiny se dá zresetovat pomocí `kafka-consumer-groups`:
https://gist.github.com/marwei/cd40657c481f94ebe273ecc16601674b

Custom Deserializer

- Kafka-console-consumer (a další nástroje) dokáží pracovat pouze s nějakými formáty dat (kafka-console-consumer prakticky jenom se Stringy). V případě použití jiných formátů může být nutné přidat custom deserializer. Například když hodnoty (value) jsou v Double:

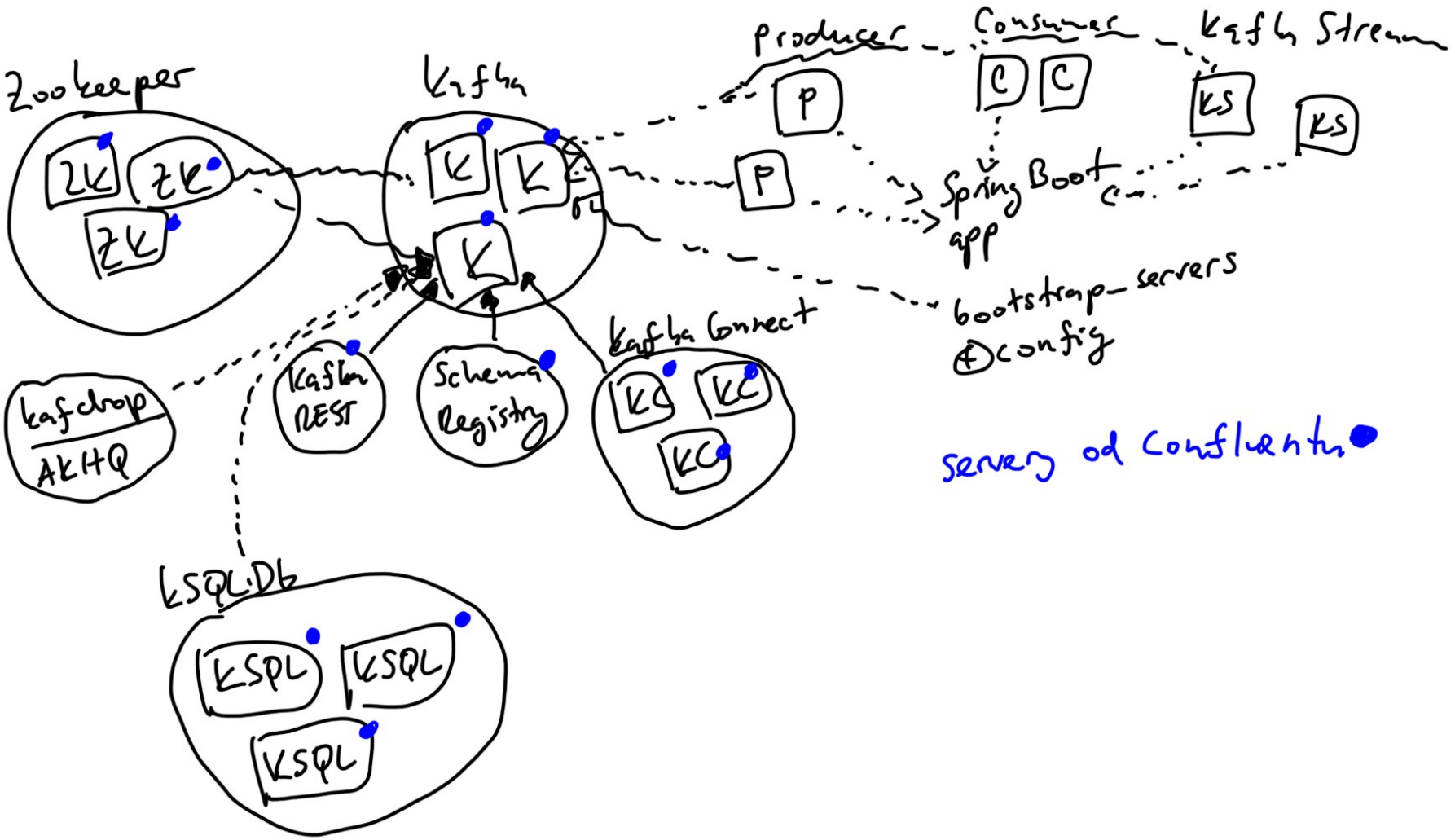
```
kafka-console-consumer --bootstrap-server kafka:9092 \
  --group ConsoleConsumer --from-beginning --topic TODO_TOPIC_NAME \
  --property \
    value.deserializer=org.apache.kafka.common.serialization.DoubleDeserializer
```

- https://stackoverflow.com/questions/44530773/kafka-stream-giving-weird-output

kafkacat

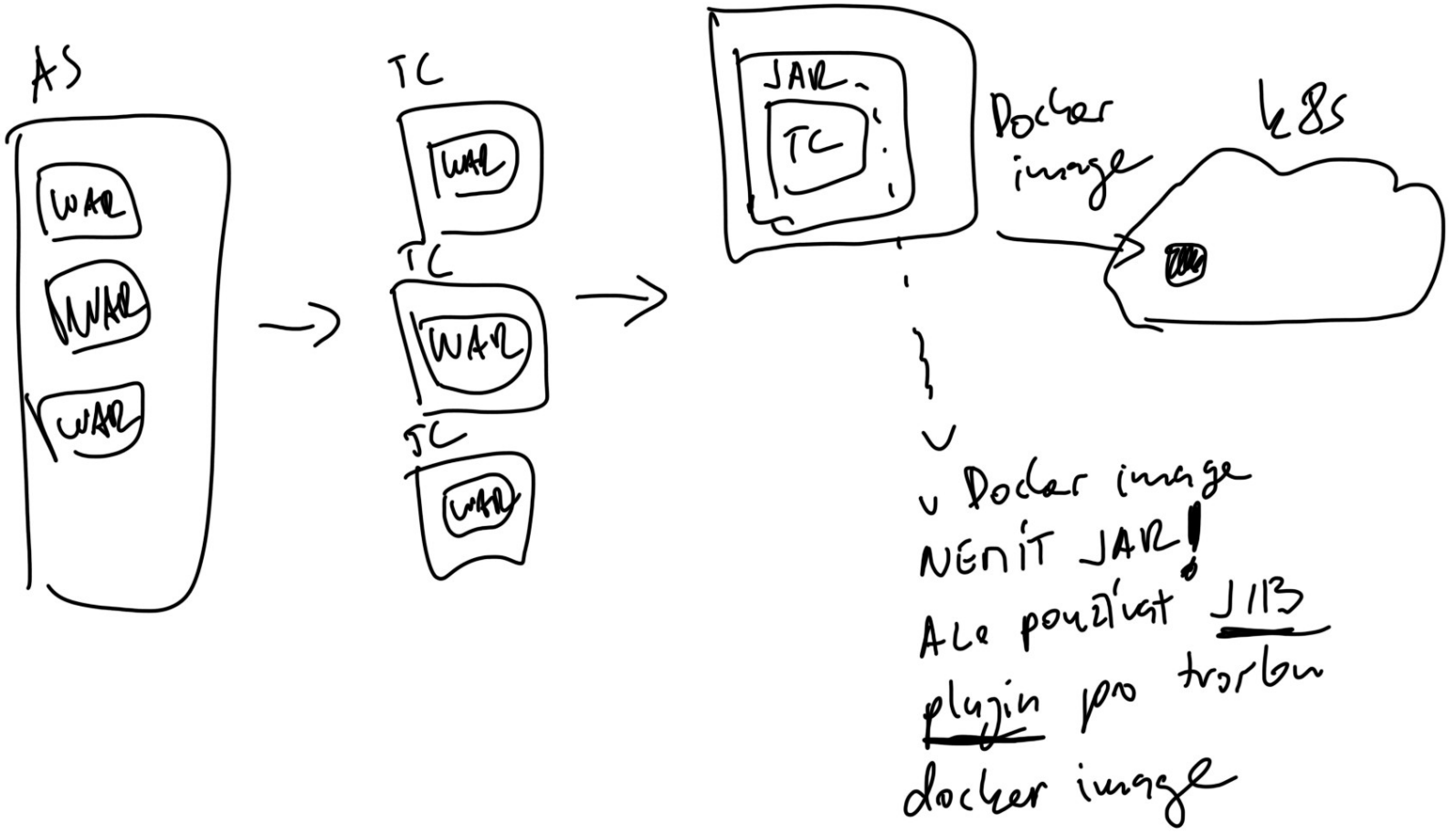
- Kafkacat je alternativa kafka-console-producer/consumer nástrojů
  - https://docs.confluent.io/platform/current/app-development/kafkacat-usage.html
- Příklady:
  - List topiců:
    - `docker run --rm --tty --network kafka_net confluentinc/cp-kafkacat kafkacat -b kafka:9092 -L`
  - Kafkacat jako consumer topicu „first_topic“:
    - `docker run --rm --tty --network kafka_net confluentinc/cp-kafkacat kafkacat -b kafka:9092 -t first_topic`
  - Kafkacat jako producer topicu „first_topic“:
    - `docker run --rm --tty --network kafka_net confluentinc/cp-kafkacat kafkacat -b kafka:9092 -t first_topic -P`
      - Zadávání by se mělo potvrdit CTRL+D, ale z nějakého důvodu mi to nefunguje :-(

Kafka Ecosystem



Kafka: producer, consumer, stream aplikace

- Kafku nezajímá, kde Vaše producer, consumer, nebo Kafka Stream aplikace bude běžet. V dnešní době typicky běží v Kubernetes clusteru a ideální je, když to je Spring Boot aplikace. Dřív tyto aplikace také běžely na aplikačních serverech:



pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>3.1.0</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
</build>
```

MyProducer

acks hodnoty: 0, 1, -1 (all)

```java
public class MyProducer {

    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        properties.setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        properties.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        properties.setProperty(ProducerConfig.ACKS_CONFIG, "1");
        properties.setProperty(ProducerConfig.RETRIES_CONFIG, "3");

        try(Producer<String, String> producer = new KafkaProducer<>(properties)) {
            ProducerRecord<String, String> producerRecord
               = new ProducerRecord<>("first_topic", "mykey", "myvalue");
            producer.send(producerRecord);
            producer.flush(); // can be replaced with properties.setProperty("linger.ms", "1");
        }
    }

}
```

Jak funguje Producer

1) Producer naváže připojení s jedním z bootstrap serverů (Kafka Broker), best practice je do konfigurace Producera nastavit minimálně dva bootstrap servery.
2) Bootstrap server vrátí list všech brokerů v clusteru a metadata jako topics, partitions, replication factors atd.
3) Na základě tohoto listu Producer identifikuje leader brokera, který má leader partition a zapíše záznam do ní.

- https://dzone.com/articles/kafka-producer-overview
- Producer všechny tyto operace provádí ve vlákně kafka-producer-network-thread (Sender), které je typu daemon:
  - https://jaceklaskowski.gitbooks.io/apache-kafka/content/kafka-producer-internals-Sender.html
- Producer JE thread safe!!!
  - https://stackoverflow.com/questions/36191933/using-kafka-producer-by-different-threads

Producer Configuration I.

- acks
  - The number of acknowledgments the producer requires the leader to have received before considering a request complete. This controls the durability of records that are sent.
  - Výchozí nastavení je „all“. Toto nastavení se vztahuje k nastavení min.insync.replicas (na Topicu), které nastavuje počet brokerů, kteří musí zalogovat message předtím, než je klientovi poslán „acknowledge“. Výchozí hodnota min.insync.replicas je 1. Správné nastavení se vztahuje k „replication factor“. Pokud je replication factor 3, pak správná hodnota min.insync.replicas je 2.
  - https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html#producerconfigs_acks
- https://strimzi.io/blog/2020/10/15/producer-tuning/
- https://www.javierholguera.com/2018/06/13/kafka-defaults-that-you-should-re-consider-i/

Producer Configuration II.

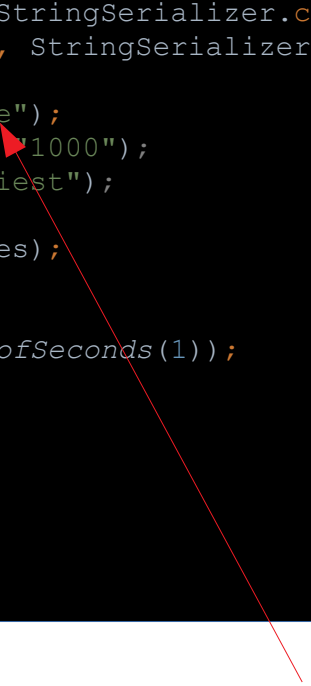
- retries
  - Kolikrát se bude Producer snažit doručit zprávu Kafka brokerovi. Výchozí hodnota je 0. Když se nastaví retries na hodnotu vyšší než 0, pak by se mělo nastavit max.in.flight.requests.per.connection = 1, jinak by mohlo dojít k situaci, že zpráva, která se odesílá v retry módu by mohla být doručena ve špatném pořadí.
    - https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html#producerconfigs_retries
- compression.type
  - Komprese je užitečná pro zvýšení throughput a snížení storage, ale nemusí být vhodná pro low-latency aplikace, komprese a dekomprese zvýší latenci.
  - Podporované typy: gzip, snappy, lz4, zstd

Producer Configuration III.

- Batching zpráv zvýší throughput. Důležitá nastavení:
  - batch.size
    - Maximum batch size
  - linger.ms
    - Maximální doba do odeslání batche
  - Zprávy se pošlou když jeden z těchto dvou parametrů je dosažen
- buffer.memory:
  - Když Kafka Producer nemůže poslat data Brokerovi (například kvůli výpadku Brokera), tak bude ukládat zprávy do buffer.memory. Jakmile se buffer.memory zaplní (default velikost: 32 MB), pak počká max.block.ms (default: 60s) jestli se buffer podaří vyprázdnit. Pokud tento čas uplyne a buffer není vyprázdněn, pak se vyhodí výjimka.

MyConsumer

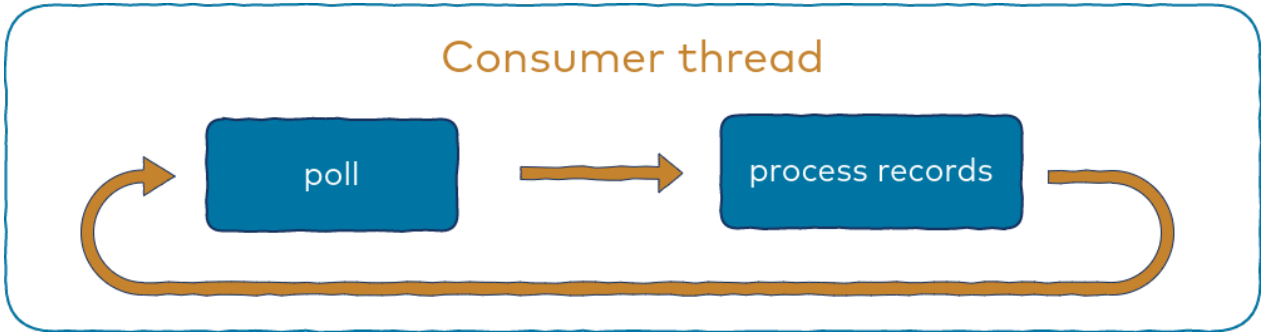
```java
public class MyConsumer {

    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.setProperty(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        properties.setProperty(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        properties.setProperty(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        properties.setProperty(ConsumerConfig.GROUP_ID_CONFIG, "test");
        properties.setProperty(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true");
        properties.setProperty(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "1000");
        properties.setProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(properties);
        consumer.subscribe(Arrays.asList("first_topic"));
        while(true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofSeconds(1));
            records.forEach(record -> {
                System.out.println(record.key() + ":" + record.value());
            });
        }
    }

}
```



Alternativa k:
`consumer.commitSync();`

Jak funguje Consumer I.

- Consumer provádí polling:



- Ve výchozí konfiguraci Consumer ukládá offsety do Kafky. Pomocí `auto.commit.interval.ms` se dá ovlivnit frekvence commitů. Offsety nejsou commitované v samostatném vlákně, ale jsou commitované ve stejném vlákně ve kterém se dělá polling. Commitované jsou pouze offsety, které byly zprocesované v předcházejícím volání poll metody. Protože processing záznamů probíhá mezi voláními poll metody, offsety dosud nezprocesovaných záznamů nebudou nikdy zprocesované. Tímto je garantována at-least-once delivery.
- Protože jsou záznamy získány a zprocesovány ve stejném vlákně, jsou zprocesovány ve stejném pořadí, ve kterém byly zapsány do partition. Tímto je garantované stejné pořadí zprocesování záznamů.

https://chrzaszcz.dev/2019/06/16/kafka-consumer-poll/

- Consumer NENÍ thread safe!!!
  - https://www.confluent.io/blog/kafka-consumer-multi-threaded-messaging/

Jak funguje Consumer II.

- Zajímavá konfigurace Consumera je `max.poll.records` (default: 500 záznamů), která nastavuje maximální počet záznamů vrácených v jenom volání poll() metody. Pokud byste tuto hodnotu zvyšovali nebo Vaše message v Kafce měly větší velikost, pak je ještě důležitá hodnota `max.partition.fetch.bytes` (default: 1 MB), která specifikuje maximální počet bytů, které server vrátí per partition.
  - Proč tyto omezení? Aby byl Consumer ochráněn proti situaci, kdy by bylo v Kafce velké množství záznamů, nebo ty záznamy měly velkou velikost.
- Také existuje konfigurace `fetch.min.bytes`, která pro změnu nastavuje minimální počet bytů, které má mít Kafka v Topicu aby vrátila klientovi nějaká data (default: 1 byte). Toto nastavení zvýší throughput, ale zvýší také latenci. Při jeho použití může mít ještě význam nastavit parametr `fetch.max.wait.ms`, díky kterému se nečeká pouze až bude min.bytes k poslání, ale po uplynutí wait.ms času broker pošle klientovi data (default: 500 ms).

Jak funguje Consumer III.

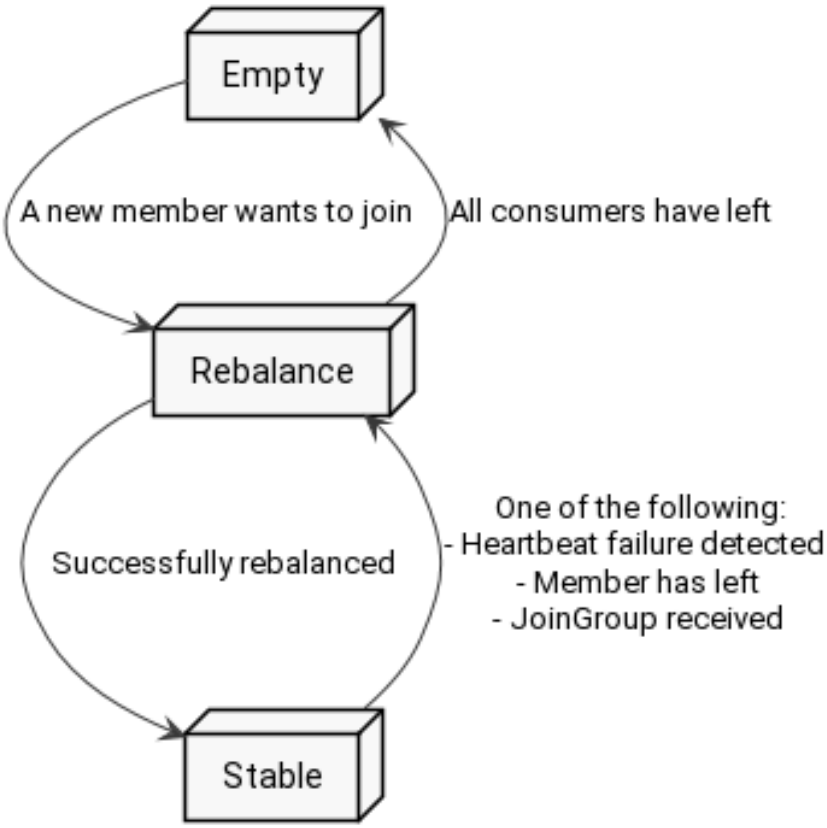
- Consumerovi se posílají pouze ty záznamy, které byly zapsané do všech in-sync replik. Je to kvůli konzistenci dat a díky tomu nemůže nastat situace, kdy by leader obdržel zprávu, klient by ji přečetl, leader by crashnul předtím než stačil tu zprávu zreplikovat do replik, nějaká replika je povýšena na leadera a najednou v Topicu zpráva není, ale byla zkonzumována klientem.
  - Toto nastavení také znamená, že když z nějakého důvodu bude pomalá replikace dat mezi brokery, tak bude trvat delší dobu než se zprávy dostanou ke klientům (protože se nejprve čeká až jsou zprávy zreplikovány).

Consumer: Group Leader vs. Followers

- Group Leader je jeden z Consumerů (v praxi první Consumer, který se připojil do skupiny), který funguje jako normální Consumer a normálně konzumuje data z Kafky, ale navíc při rebalancingu rozhoduje, který Consumer bude konzumovat které partitions.
- Rozdělení partitions se dá změnit pomocí property partition.assignment.strategy:
  - https://medium.com/streamthoughts/understanding-kafka-partition-assignment-strategies-and-how-to-write-your-own-custom-assignor-ebeda1fc06f3
- Kafka má tři build-in strategie:
  - Range (default)
  - RoundRobin
  - StickyAssignor
- Pokud budete měnit výchozí strategii, pak je důležité, že všichni Consumeři v Consumer Group musí mít stejnou strategii!

Jak funguje Consumer: Group Rebalancing I.

Zjednodušený graf stavů
a přechodů mezi nimi:

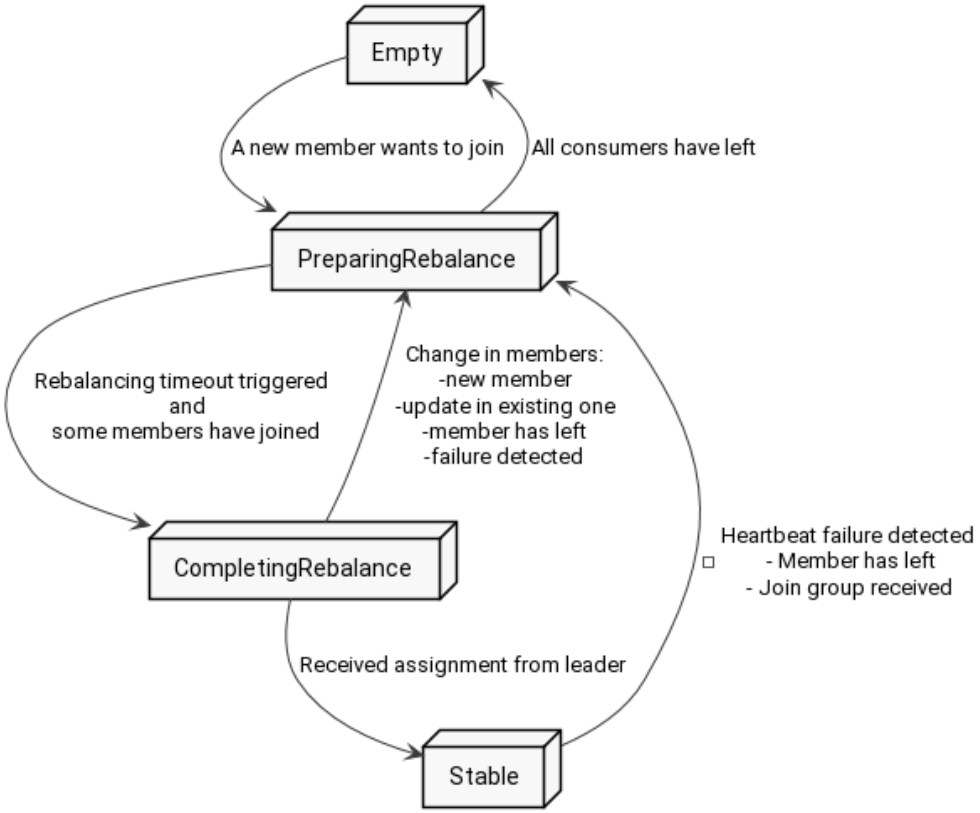


- Jakmile se nový Consumer zapojí do Consumer Group, stávající Consumer opustí skupinu, nebo do jednoho z odebíraných topiců se přidá partition, spustí se „Group Rebalancing“.
- Consumer Group může být v jednom z následujících stavů:
  - Empty: Consumer Group existuje, ale nikdo v ní není
  - Stable: Rebalancing proběhl a Consumeři konzumují zprávy
  - Preparing Rebalance & Completing Rebalance
  - Dead
- https://chrzaszcz.dev/2019/06/kafka-rebalancing/

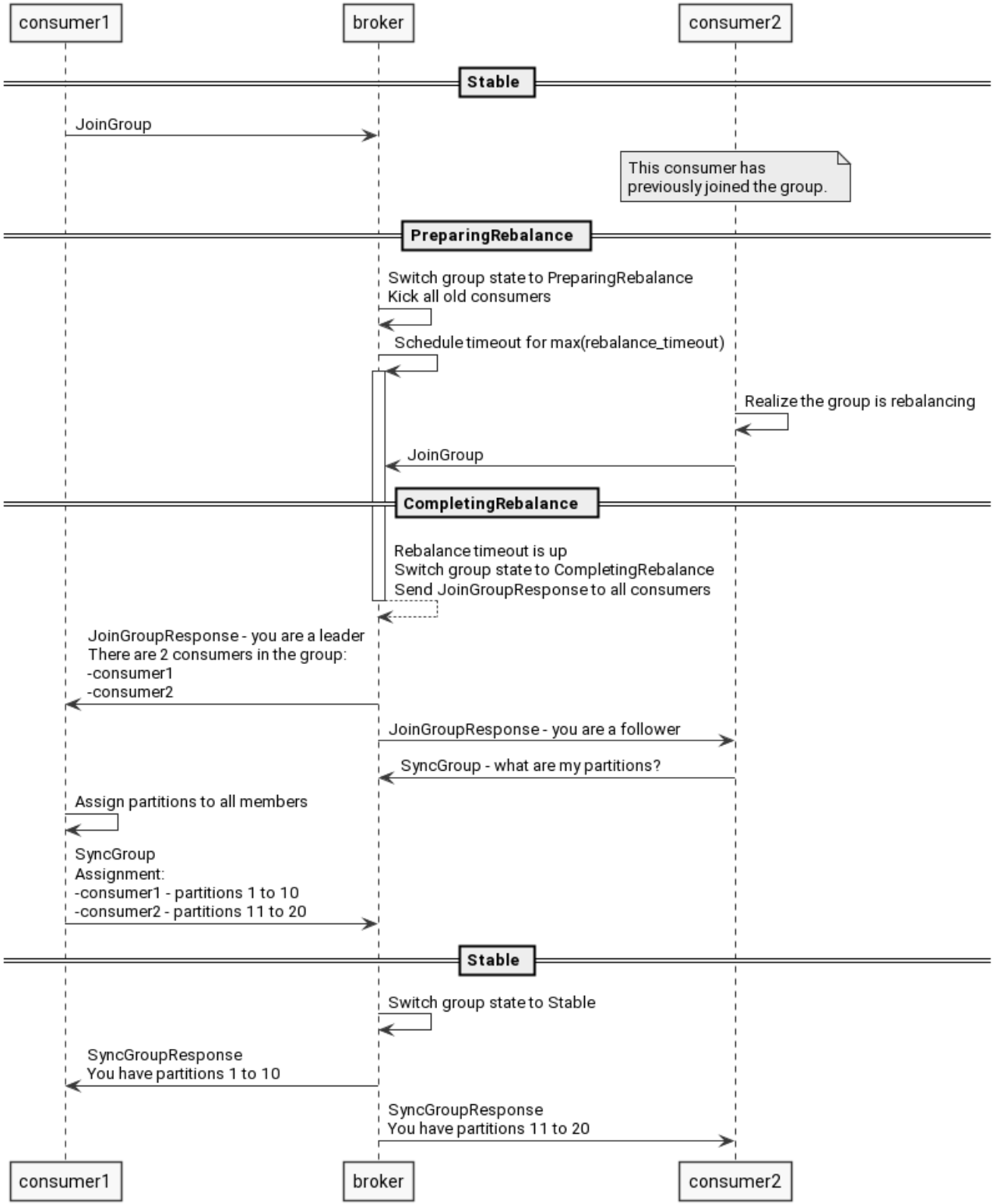
https://www.confluent.io/blog/kafka-consumer-multi-threaded-messaging/#group-rebalancing

Jak funguje Consumer: Group Rebalancing II.

- Jak vypadá rebalancing (z high level perspektivy):
  - Nový Consumer pošle požadavek jednomu z bootstrap serverů že se chce připojit do Consumer Group.
  - Kafka změní stav skupiny na PreparingRebalancing, odpojí všechny stávající členy a čeká až se znovu připojí.
  - Kafka:
    - Vybere jednoho z consumerů jako group leader.
    - Pošle všem consumerům info o tom, že se úspěšně připojili do skupiny
    - Leaderovi navíc pošle list followerů.
  - Followeři pošlou požadavek na partitions, které budou zpracovávat.
  - Leader rozhodne jaké partitions se budou komu posílat a pošle to Kafce.
  - Kafka dostane seznam partitions od leadera a pošle leaderovi a followerům jaké mají partitions.

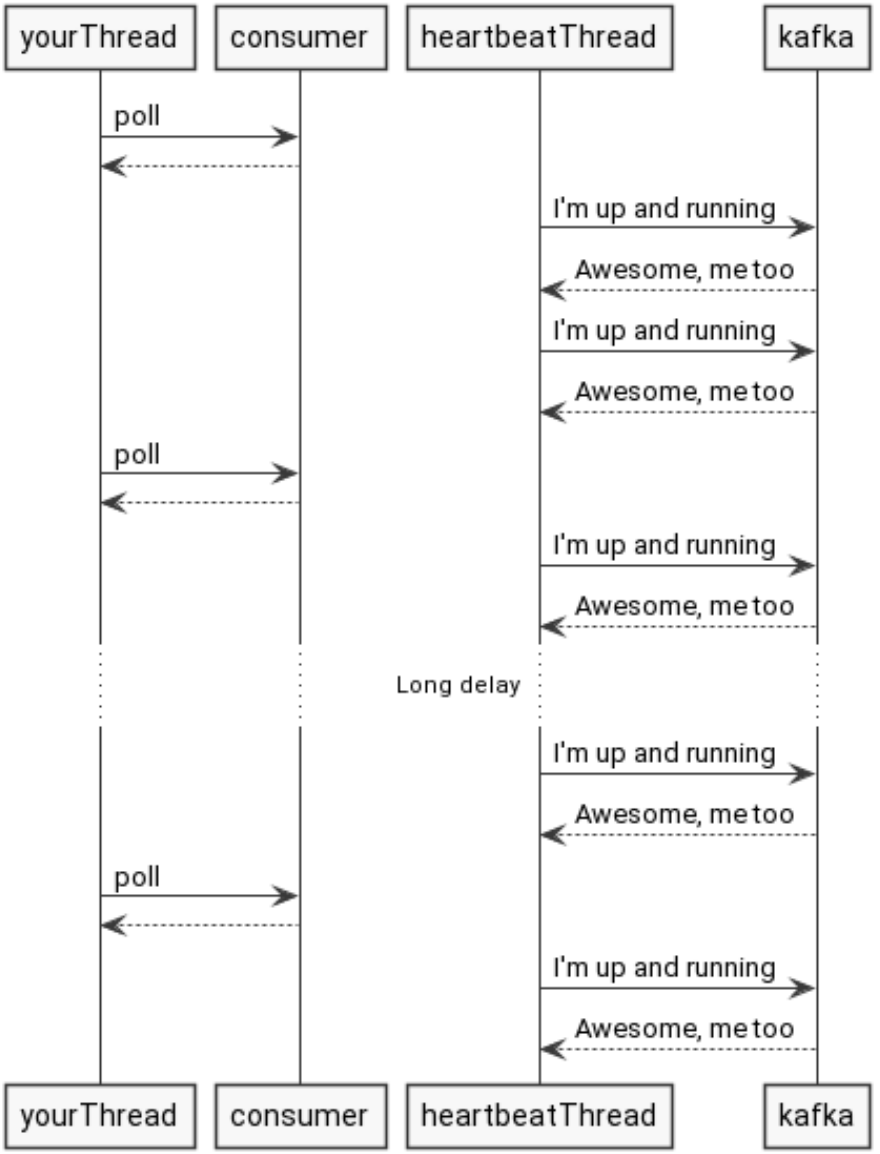


https://chrzaszcz.dev/2019/06/kafka-rebalancing/


consumer1
broker
consumer2
Stable
JoinGroup
This consumer has previously joined the group.
PreparingRebalance
Switch group state to PreparingRebalance
Kick all old consumers
Schedule timeout for max(rebalance_timeout)
Realize the group is rebalancing
JoinGroup
CompletingRebalance
Rebalance timeout is up
Switch group state to CompletingRebalance
Send JoinGroupResponse to all consumers
JoinGroupResponse - you are a leader
There are 2 consumers in the group:
-consumer1
-consumer2
JoinGroupResponse - you are a follower
SyncGroup - what are my partitions?
Assign partitions to all members
SyncGroup
Assignment:
-consumer1 - partitions 1 to 10
-consumer2 - partitions 11 to 20
Stable
Switch group state to Stable
SyncGroupResponse
You have partitions 1 to 10
SyncGroupResponse
You have partitions 11 to 20
consumer1
broker
consumer2

Jak funguje Consumer: Heartbeat

- Consumer má vlákno kafka-coordinator-heartbeat-thread, které zjišťuje, jestli připojení ke Kafce funguje.
  - https://chrzaszcz.dev/2019/06/kafka-heartbeat-thread/
- Heartbeat konfigurace je pomocí parametrů:
  - `heartbeat.interval.ms`: Frekvence posílání heartbeat požadavků, default: 3s
  - `session.timeout.ms`: Časový interval, ve kterém broker musí získat alespoň jeden požadavek od Consumera, jinak je pro něj Consumer dead, default: 10s
- https://stackoverflow.com/questions/43881877/difference-between-heartbeat-interval-ms-and-session-timeout-ms-in-kafka-consume



Jak funguje Consumer: Heartbeat

- Další konfigurace, která je významná je `max.poll.interval.ms` (default: 5 minut). V případě, že processing dat trvá delší dobu než je toto nastavení, pak Broker považuje Consumera za dead, vykopne ho z Consumer Group a provede rebalancing.

Spring + Kafka

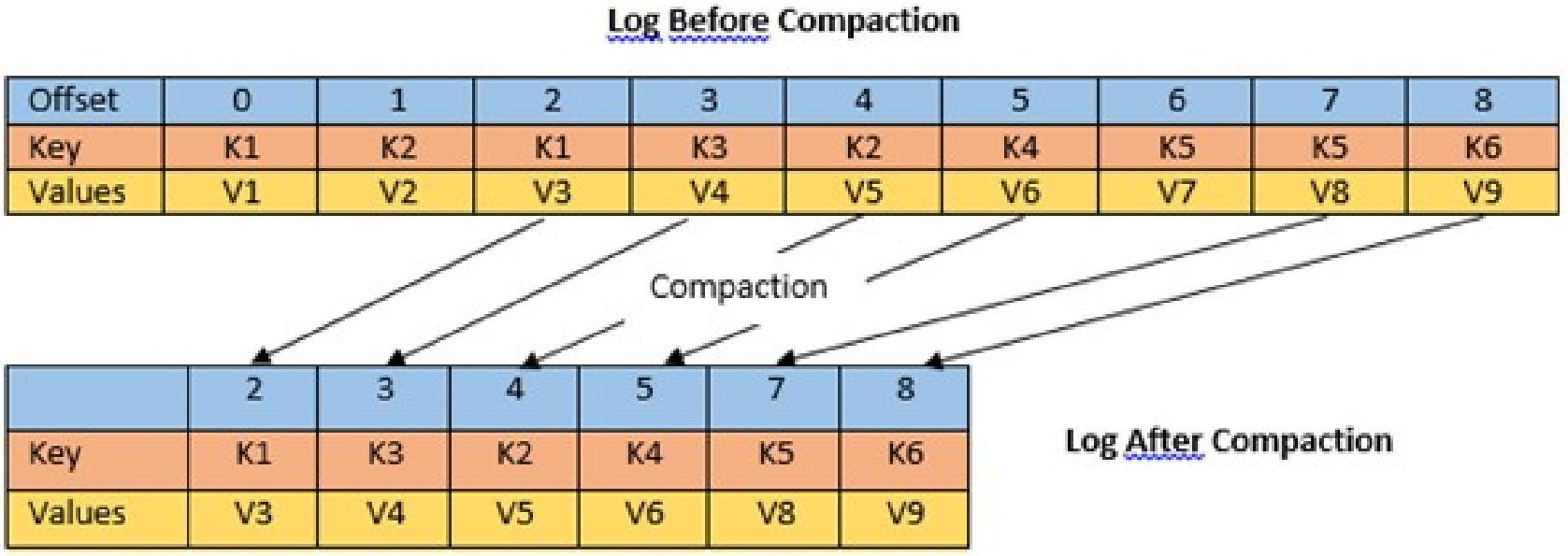
- Dokumentace:
  - https://spring.io/projects/spring-kafka
- KafkaTemplate je thread-safe!
- Funkční projekt:
  - https://memorynotfound.com/spring-kafka-json-serializer-deserializer-example/
- Spring + Avro:
  - https://www.codenotfound.com/spring-kafka-avro-bijection-example.html

Spring Cloud Streams

- Ještě větší abstrakce než Spring Kafka
- https://piotrminkowski.com/2021/11/11/kafka-streams-with-spring-cloud-stream/
- https://medium.com/geekculture/spring-cloud-streams-with-functional-programming-model-93d49696584c
- https://www.baeldung.com/spring-cloud-stream-kafka-avro-confluent

Log Compaction

- V některých situacích může mít význam použít Topic Log Compaction:
  - https://medium.com/swlh/introduction-to-topic-log-compaction-in-apache-kafka-3e4d4afd2262
- Jedná se o mechanismus, který selektivně maže staré záznamy, pro které existuje novější záznam se stejným klíčem. Pozor! Pokud nepoužíváme klíč (je NULL), pak Log Compaction použít nemůžeme!



Změna konfigurace topicu

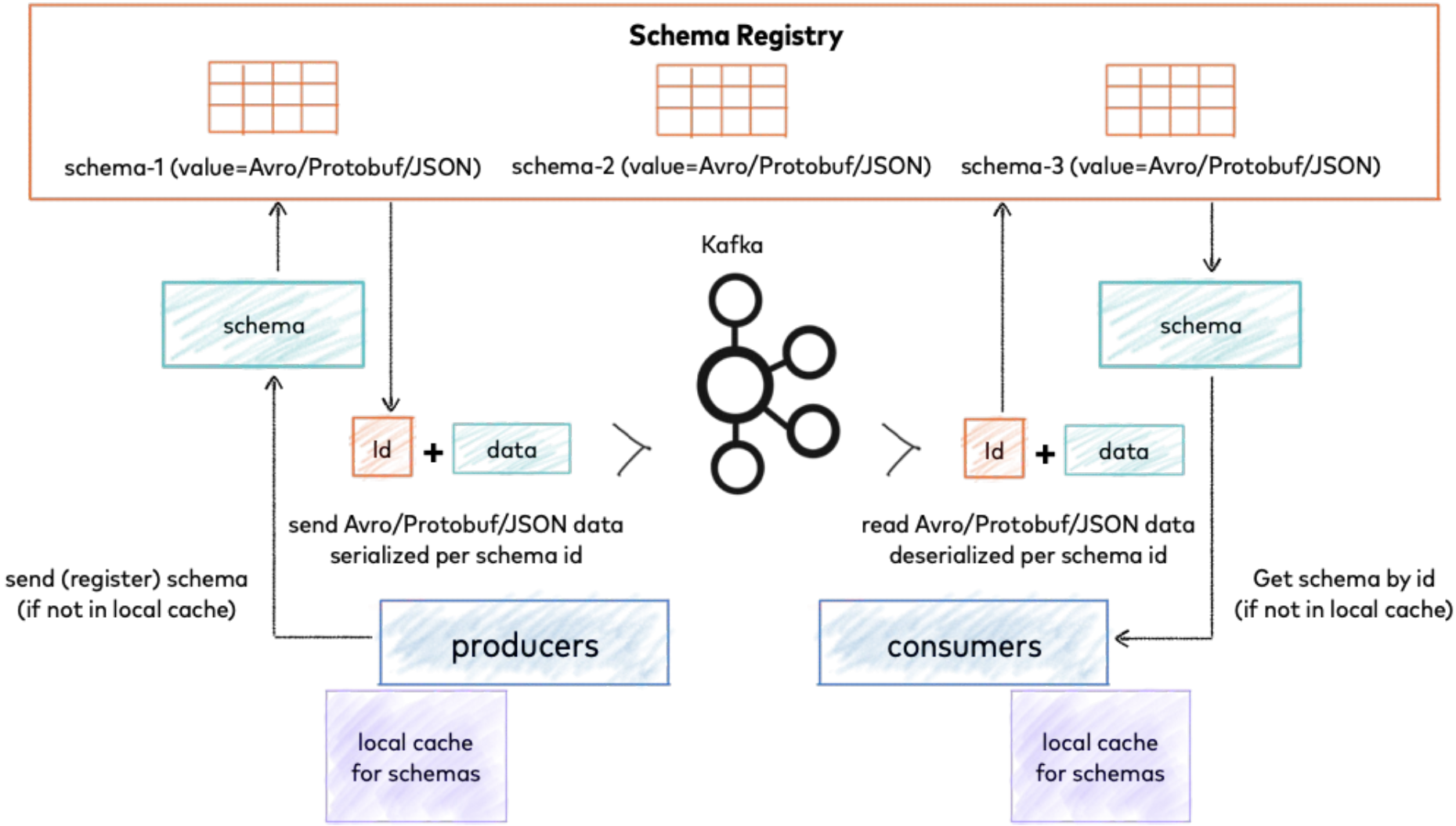
- Při vytvoření (`--create`) nebo změně (`--alter`) topicu je možné předat parametr `--config` a tím změnit výchozí konfiguraci:

```
--config cleanup.policy=compact
```

  - https://kafka.apache.org/documentation/#topicconfigs

Schema Registry + Avro

Schema Registry



Avro + Schema Registry I.

- Message v Kafce může mít libovolný formát (String, JSON, XML, binární). Kafka je agnostická vůči formátu serializace, pro ni je klíč i hodnota pouze pole bytů. To má ale nevýhodu, že se musí dávat velký pozor na to, aby si Producer a Consumer navzájem rozuměli.
- Řešení tohoto problému je schéma. Výchozí schéma v Kafce je Avro. Další podporované jsou například JSON Schema nebo Protobuf. Další výhodou schématu je to, že data mohou být v Kafce uložena efektivněji. Konkrétně při použití Avro schématu v datech interně nejsou uloženy názvy atributů, má binární formát a díky tomu je kompaktnější než například JSON.
  - https://www.baeldung.com/java-apache-avro
  - https://www.confluent.io/blog/avro-kafka-data/
  - http://avro.apache.org/docs/current/
  - https://medium.com/slalom-technology/introduction-to-schema-registry-in-kafka-915ccf06b902
    - Tady je také Maven plugin, který z .avsc souborů generuje Java třídy.

Avro + Schema Registry II.

- V případě, že chceme použít jako formát dat Avro, pak musíme specifikovat schéma. Schémata se nachází ve Schema Registry:
  - Landoop: http://localhost:3030/api/schema-registry/
  - Confluent: http://localhost:8081
- Konfigurace:
  - https://docs.confluent.io/platform/current/schema-registry/index.html
    - Schémata jsou uložena v Kafce v topicu `_schemas`
    - Názvy schémat mají následující formát: `<topic>-key, <topic>-value`
- Schema Registry obsahuje REST API, pomocí kterého se s ním dá pracovat:
  - https://github.com/confluentinc/schema-registry
- Nebo přes Maven:
  - https://docs.confluent.io/platform/current/schema-registry/develop/maven-plugin.html
- Seznam všech schémat:
  - http://localhost:8081/schemas

Poznámka: Schéma v registry se nazývá „subject“, tohle vrátí jejich seznam: http://localhost:8081/subjects

Schema Compatibility

- Schema může mít nastaven jeden z typů kompatibility:
  - NONE
  - FORWARD
  - BACKWARDS (default)
  - FULL
- U výchozí BACKWARDS kompatibility tyto změny způsobí breaking change:
  - Přidání nebo odstranění fieldu bez default hodnoty
  - Změna názvu nebo typu fieldu
  - Změna enum hodnot pokud není default enum symbol
- Poznámky:
  - Kompatibilita se dá nastavit globálně nebo per schéma
  - Změna nastavení se provádí přes REST API

kafka-avro-console-producer/consumer

```
kafka-avro-console-producer --broker-list localhost:9092 --topic first_topic_avro \
  --property parse.key=true \
  --property key.schema='{"type":"string"}' \
  --property value.schema='{"type":"record","name":"myrecord","fields":[{"name" : "name",
"type" : "string"}, {"name" : "age", "type" : "int"}]}'
```

Zprávy:

```
"jirka"<TAB>{ "name" : "Jirka Pinkas", "age" : 36 }
```

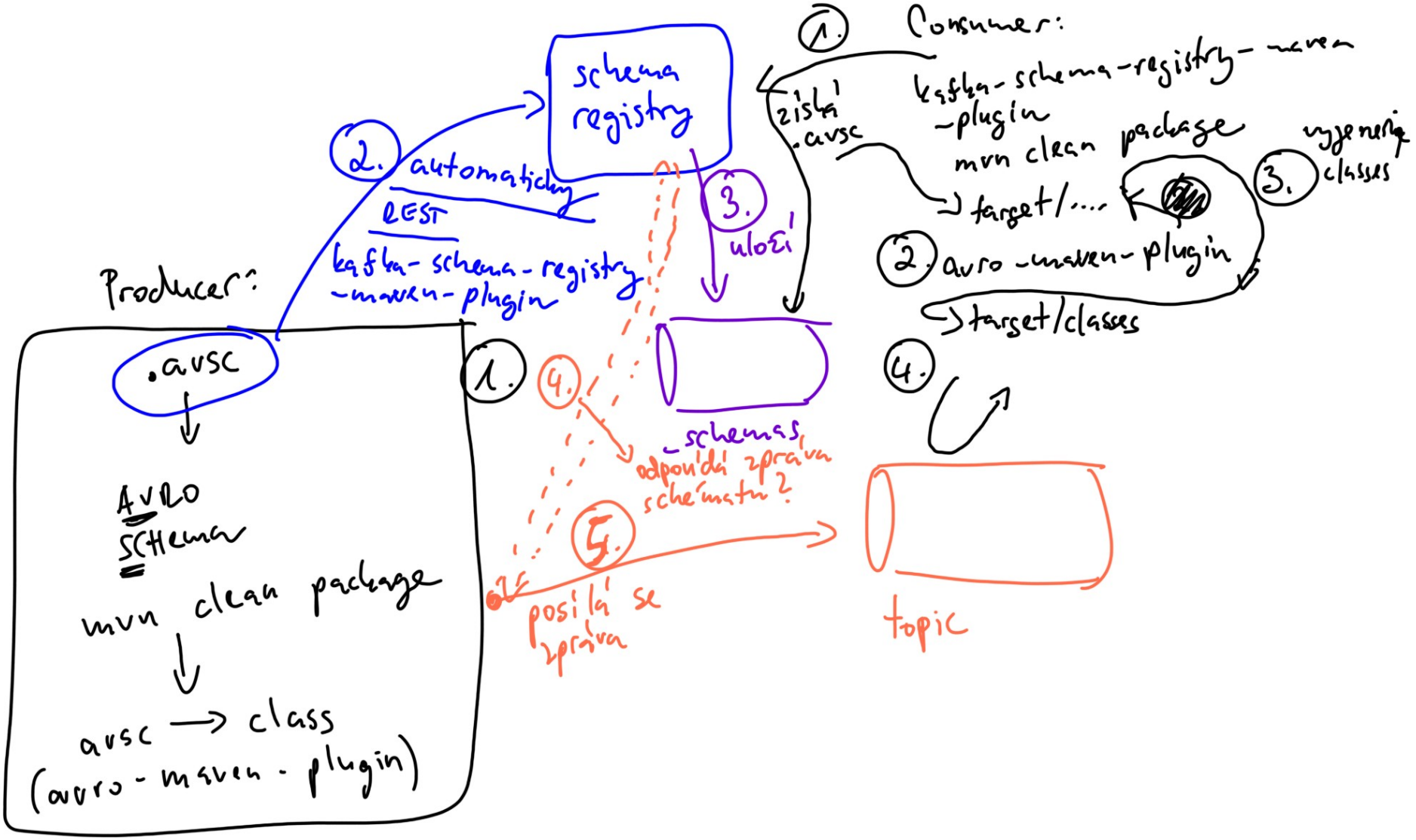
Zmáčknout klávesu TAB, protože klíč je oddělen od hodnoty tabulátorem

```
kafka-avro-console-consumer --topic first_topic_avro --bootstrap-server localhost:9092
```

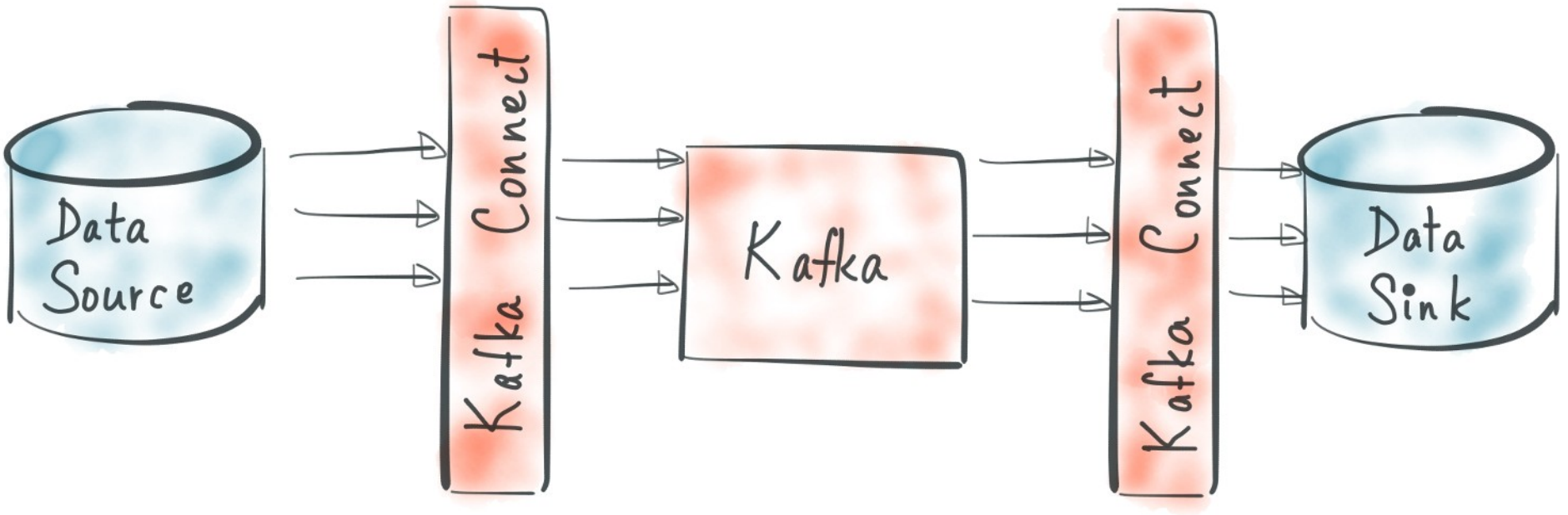
Poznámka: kafka-avro-console-producer vytvoří dvě schémata:
- first_topic_avro**-key**
- first_topic_avro**-value**

http://localhost:3030/schema-registry-ui/

Flow producera a consumera se schema registry



Kafka Connect


Data Source
Kafka Connect
Kafka
Kafka Connect
Data Sink

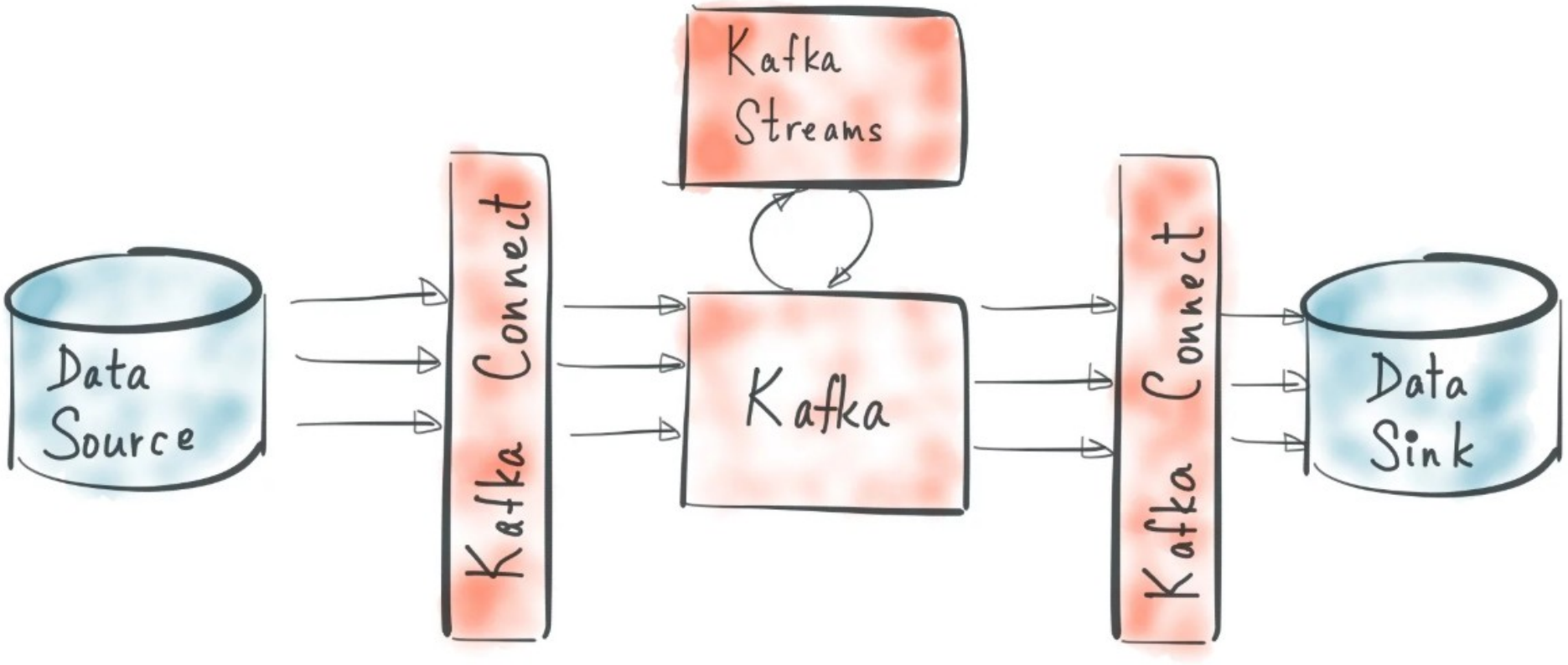
Základní pojmy

- Source => Kafka = Kafka Connect Source
- Kafka => Sink = Kafka Connect Sink
- Kafka Connect obsahuje Connectors
- Connectors + Konfigurace => Tasks
- Tasks jsou vykonávány procesy, které mají název „Worker“ (server):
  - Worker = Java proces
  - Worker může být:
    - Standalone - jeden (vhodný max. pro testování, stav workeru je uložen lokálně)
    - Distributed - celý cluster (ideálně všude, jak na produkci, tak na testování, protože stav workeru je uložen v Kafka topicu a konfigurace Standalone se od Distrubuted konfigurace liší)

Debezium

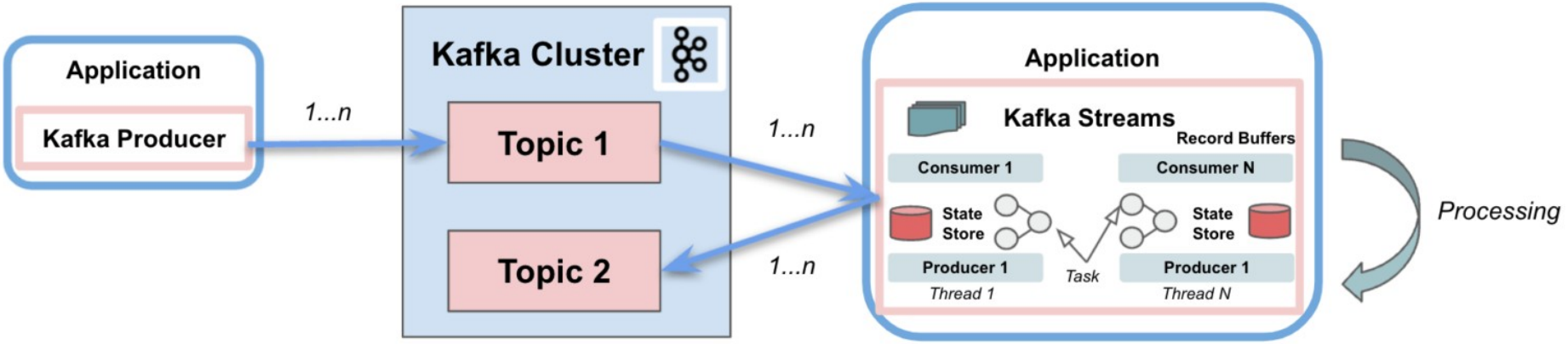
- Debezium je projekt postavený na Kafka Connect a implementuje CDC (Change Data Capture)

Kafka Streams


Kafka
Streams
Data
Source
Kafka Connect
Kafka
Kafka Connect
Data
Sink

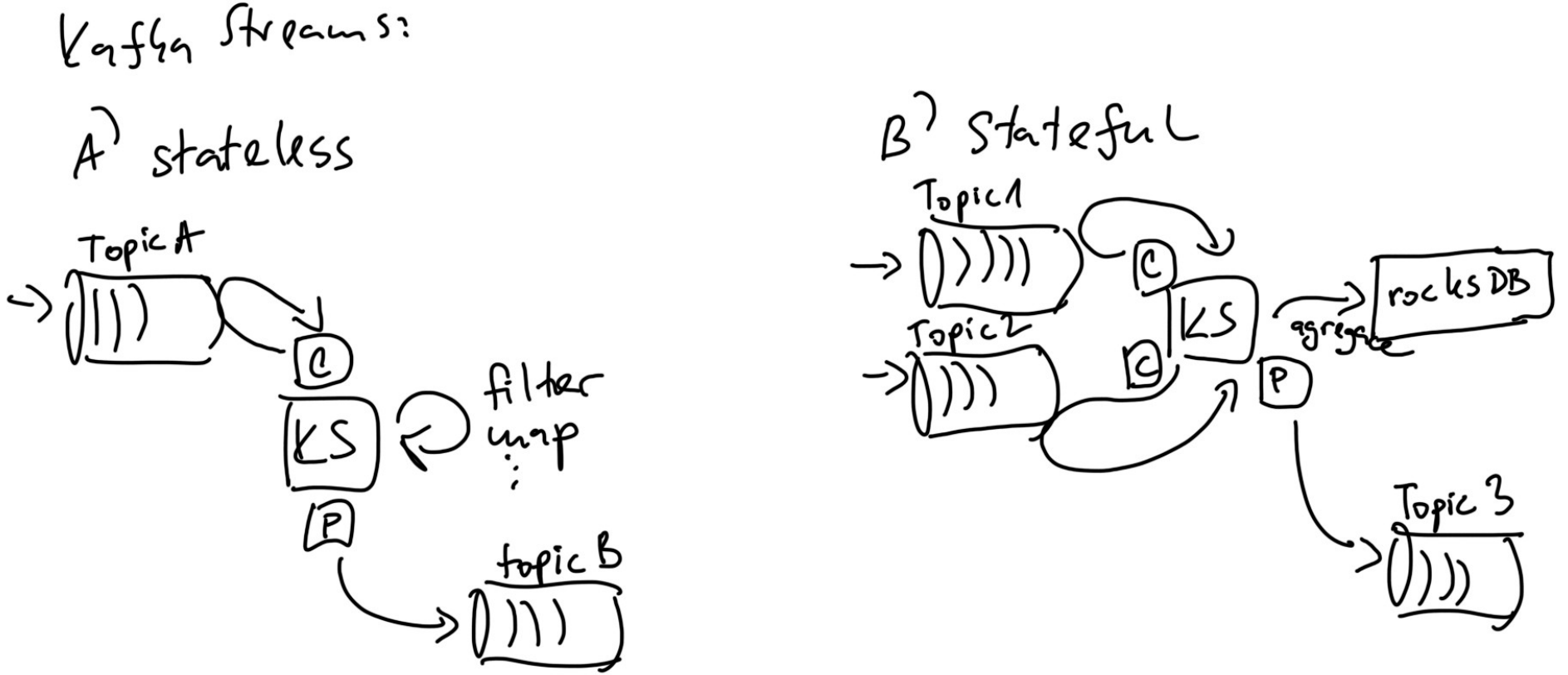
Kafka Streams

- Kafka Streams je knihovna, která slouží pro získání dat z Kafka Topicu, jejich následnou transformaci a uložení do jiného Kafka Topicu.
- Kafka Streams jsou postavené na Kafka klientských knihovnách (Kafka Consumer & Producer) a v dnešní době to je typicky Spring Boot konzolová aplikace, která obvykle běží v našem Kubernetes clusteru:



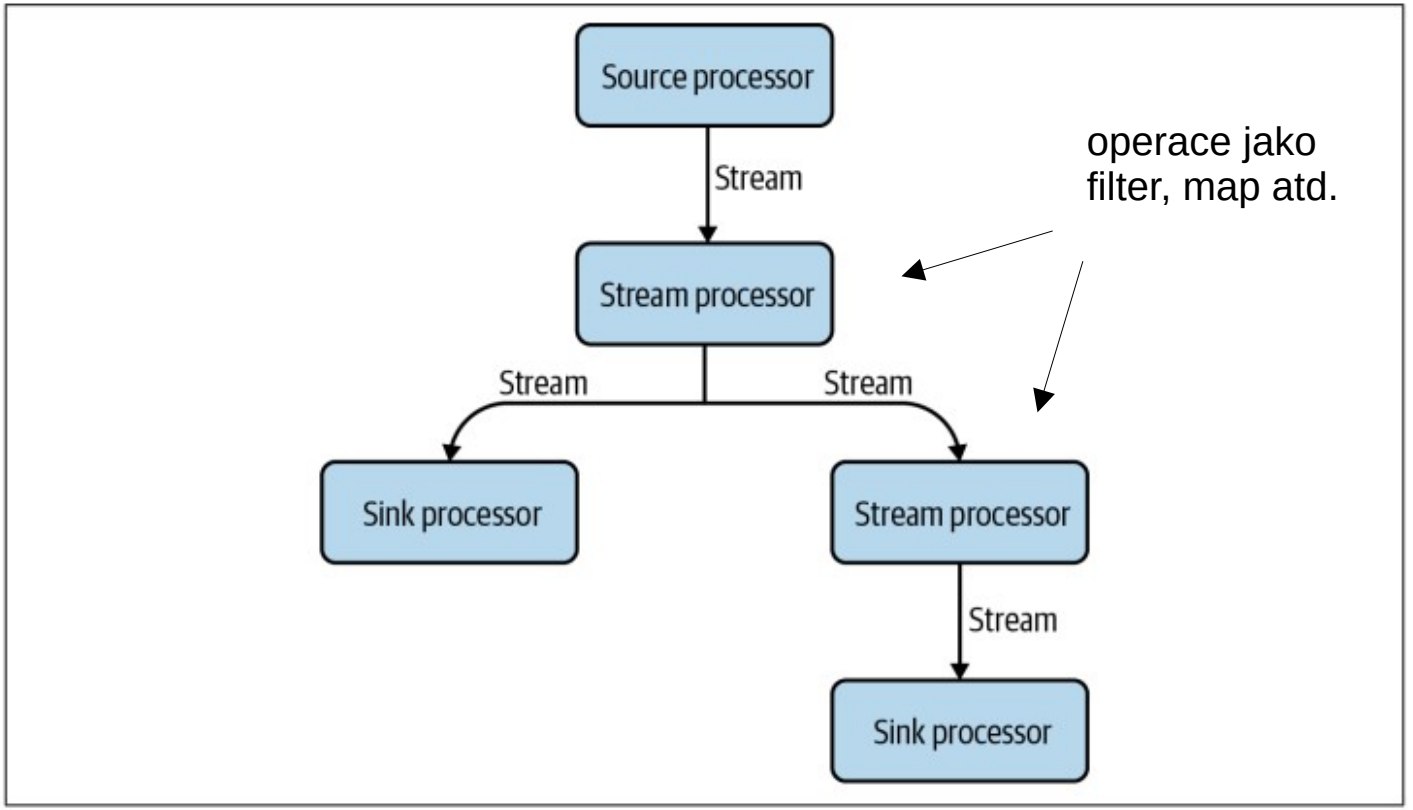
- https://www.baeldung.com/java-kafka-streams-vs-kafka-consumer
- Kafka Consumer API vs. Streams API:
  - https://stackoverflow.com/questions/44014975/kafka-consumer-api-vs-streams-api

Stateless vs. Stateful Kafka Streams



Processor Topology

- Kafka Stream aplikace je strukturovaná jako directed acyclic graph (DAG). Uzel je „processing step“ a koncové uzly jsou buď vstup z topicu (source), nebo výstupy (sink):



pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-streams</artifactId>
        <version>2.0.0</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Hello World

Tohle je jako group.id
u consumera

Záznamy se prochází
od posledního offsetu
(nebo 0)

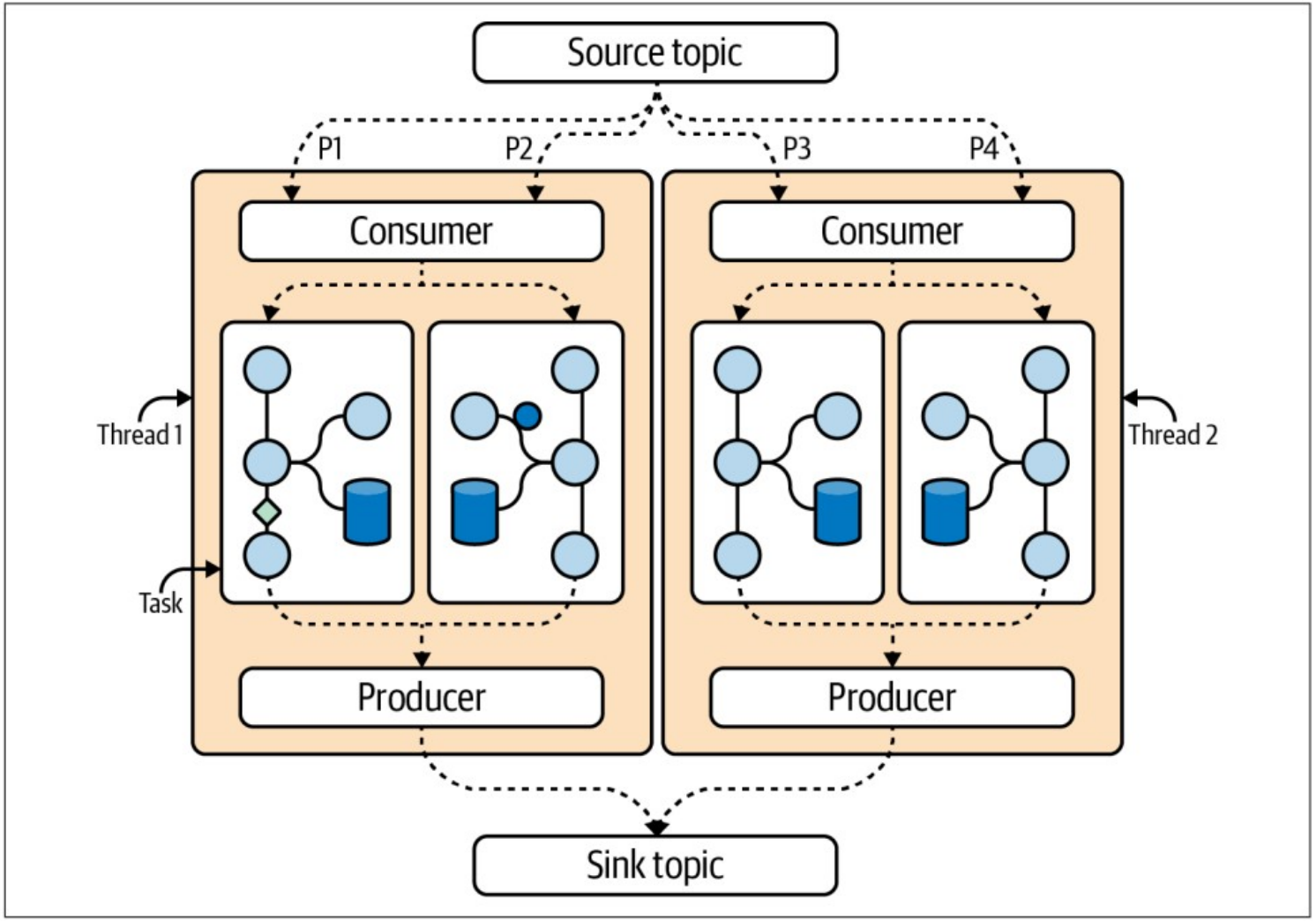
```java
public class Main {

    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.put(StreamsConfig.APPLICATION_ID_CONFIG, "kafka-streams-app");
        properties.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        properties.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
        properties.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());

        StreamsBuilder builder = new StreamsBuilder();
        KStream<String, String> stream = builder.stream("first_topic");
        stream
                .mapValues(s -> s.toLowerCase())
                .to("first_topic_lowercase");

        KafkaStreams streams = new KafkaStreams(builder.build(), properties);
        streams.start();
        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
    }
}
```

Graceful shutdown

Poznámka: Pokud chcete něco jako --from-beginning, tak musíte buď:
- změnit application id
- anebo před streams.start() zavolat streams.cleanUp()

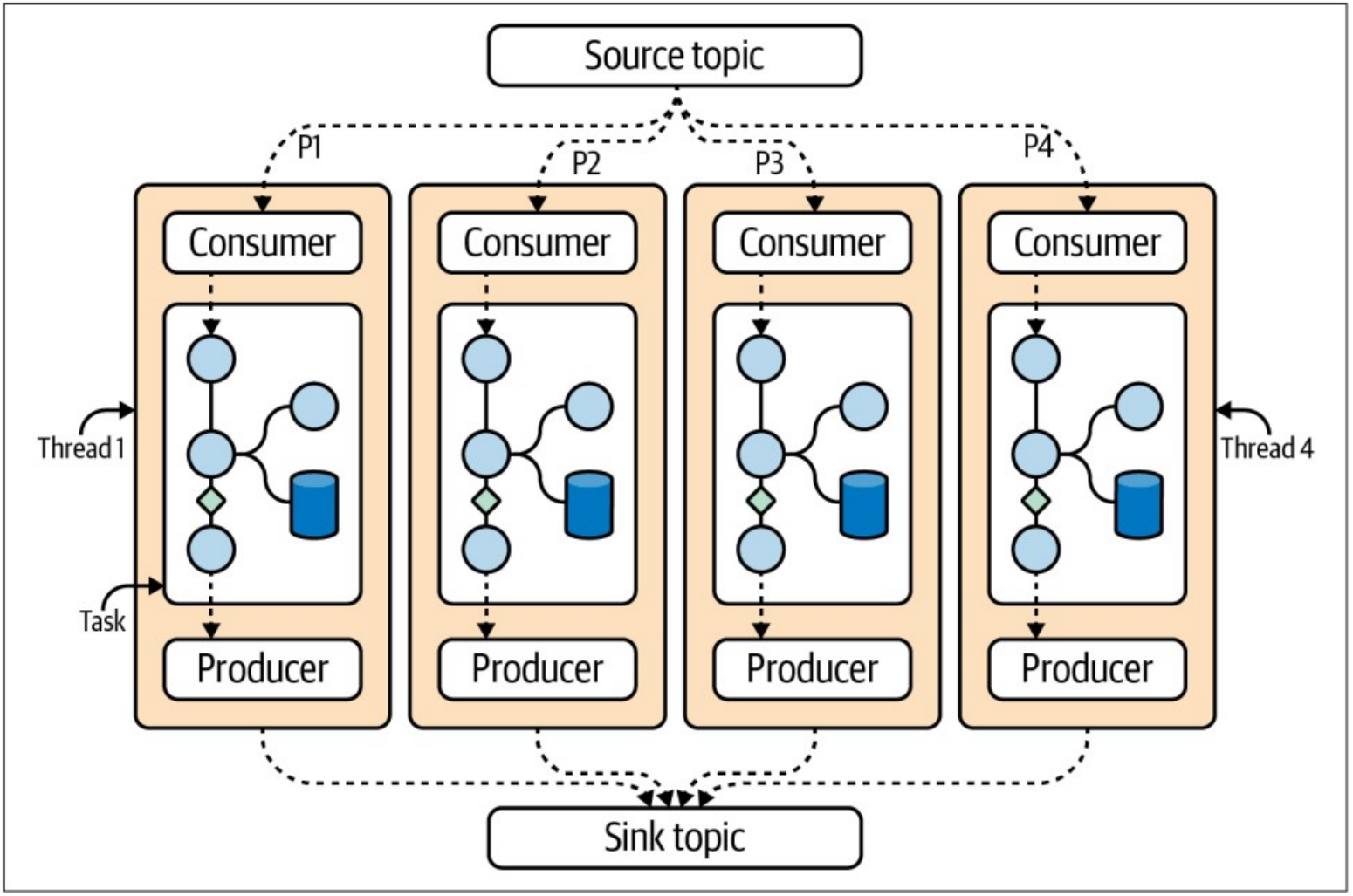
Kafka Streams Internals

- 4 Kafka Streams tasky běžící ve dvou threadech:



Kafka Streams Internals

- 4 Kafka Streams tasky běžící ve čtyřech threadech:



Tasks

- Task = nejmenší „unit of work“ Kafka Streams aplikace. Počet tasků je odvozen od počtu input partitions. Pokud máme Kafka Streams aplikaci, která používá pouze jeden input topic, pak je počet tasků roven počtu partitions toho topicu. Pokud používá více input topiců, pak je počet tasků roven maximálnímu počtu partitions, které má nějaký z topiců.
- Tasky se přiřazují StreamThreadu, který je vykonává. Ve výchozím nastavení je v aplikaci jeden StreamThread. Když bude 5 tasků, pak se nejprve vykoná logika jednoho, až skončí tak druhého atd. Když bude 5 threadů (nebo 5 instancí aplikace se stejným application.id), pak se budou tasky vykonávat paralelně.
- https://developer.confluent.io/learn-kafka/kafka-streams/internals/

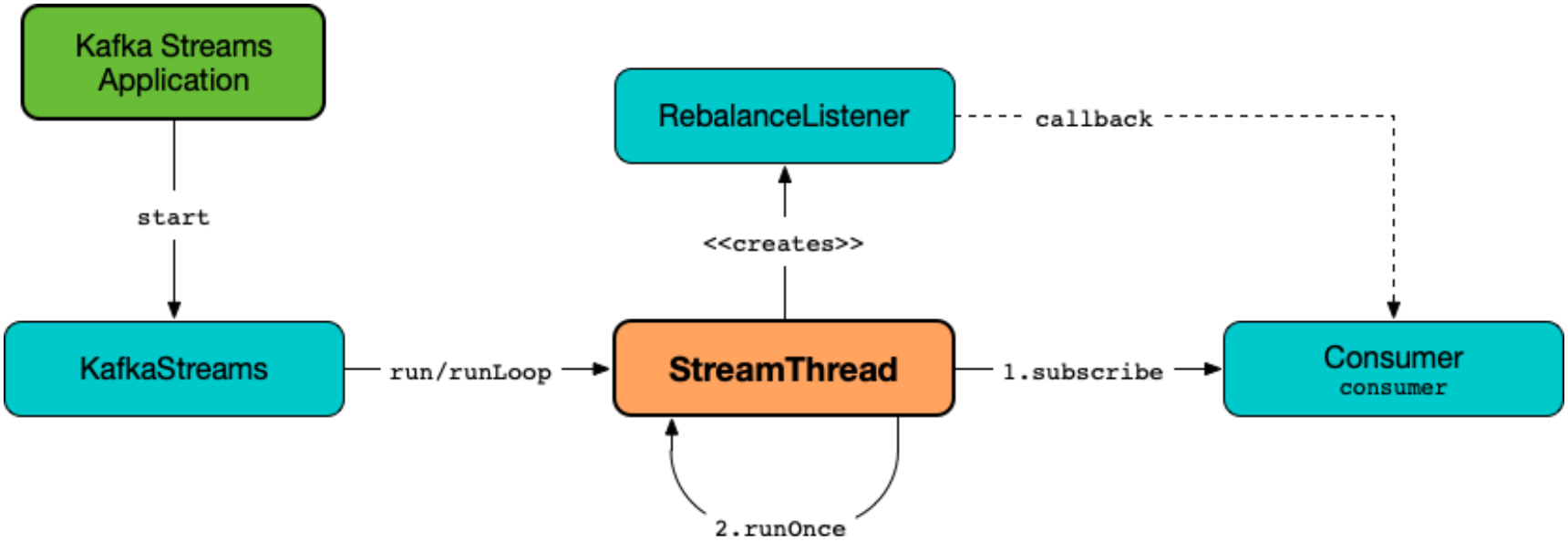
Stream & Threading

- Metoda KafkaStreams#start() spustí Kafka Stream, který se ve výchozím nastavení skládá z následujících vláken:
  - Počet těchto threadů se dá ovlivnit property num.stream.threads (default 1 vlákno):
    - ClientId-StreamThread
      - Hlavní thread, který provede subscribe na Consumera. Pokud chceme, aby právě X vláken získávalo data z input topicu, pak musí mít topic právě X partitions!
    - kafka-coordinator-heartbeat-thread
      - Consumer heartbeat thread
    - kafka-producer-network-thread
      - Kafka Producer thread
  - ClientId-CleanupThread
    - Používá se pro čištění StateDirectory
  - kafka-admin-client-thread
    - Tento thread používá Kafka AdminClient a vytváří StreamThread

StreamThread

- StreamThread
  - https://medium.com/@andy.bryant/kafka-streams-work-allocation-4f31c24753cc
- Jak zjistit počet běžících Consumerů:

```
kafka-consumer-groups --bootstrap-server kafka:9092 --describe --group STREAM_APPLICATION_ID
```



Stream Threads

- Mezi vlákny není žádný sdílený stav. Z pohledu Kafka Streams je jedno, jestli paralelismu u streamů docílíme zvýšením num.stream.threads, nebo spuštěním většího množství instancí Kafka Streams aplikace se stejným application.id.
- Osobně bych ale preferoval škálování pomocí většího množství instancí z následujících důvodů:
  - Větší throughput
  - Lepší vytěžování zdrojů
  - Větší odolnost vůči výpadkům jednotlivých instancí

Depth first processing

- Když Stream začne ze vstupního topicu procesovat nějaký záznam, tak ten záznam musí projít celou topologii, aby se mohl zpracovávat další záznam. Výhoda je, že chování streamu je jednoduché na pochopení. Nevýhoda je, že když je nějaká processing operace pomalá, tak bude blokovat zprocesování dalších záznamů.
  - **Pozor!!!** U sub-topologie toto taky platí, ale pro každou sub-topologii zvlášť a každá se musí vykonávat v dedikovaném vlákně. Když jsou 2 sub-topologie, tak musí být minimálně buď 1 instance streams aplikace s num.stream.threads=2, nebo 2 instance s num.stream.threads=1
    - Sub-topology je, když v rámci topologie se uloží data do sink processoru a pak se načtou dalším source processorem.
    - Jak získat Topology Description:

```
Topology topology = streamsBuilder.build();
log.info("Topology description {}", topology.describe());
```

    - Topology vizualizer:
      - https://zz85.github.io/kafka-streams-viz/
  - https://medium.com/@andy.bryant/kafka-streams-work-allocation-4f31c24753cc

Stateless operations

- Stateless Kafka Streams aplikace jsou nejjednodušší streamy. Nevyžadují žádnou znalost předcházejících eventů, neukládají si žádný stav, jenom vezmou vstupní data, udělají jejich processing a uloží je do výstupních topiců.

mapValues(), map()

- mapValues()
  - Provede transformaci každé hodnoty ve streamu.
- map()
  - Provádí transformaci klíčů i hodnot
- Pokud není nutné provést transformaci klíče, pak bychom měli vždy preferovat operaci mapValues(), protože nezpůsobí re-partitioning.
- Podobné operace jsou transformValues() a transform(), které ale používají low-level Processor API:
  - https://docs.confluent.io/platform/current/streams/developer-guide/processor-api.html

filter(), filterNot()

- filter()
  - Filtrování záznamů ze streamu
  - Neprovádí změny v klíčích / hodnotách
- filterNot()
  - Negace filter()

foreach(), peek(), print()

- foreach() je klasický foreach cyklus :-), jedná se o terminální operaci.
- peek() dělá prakticky to samé jako foreach, ale jedná se o intermediate operaci. Peek MUSÍ být idempotentní (nesmí mít side-efekty)!
- print() je pouze pro debugování a pro vypsání každého záznamu ve streamu, jedná se o terminální operaci.

branch(), merge()

- branch() rozdělí stream na základě 1 .. N predikátů do pole streamů.
- Od Kafka 2.8.0 se používá ve spojení s operací split()
- Podobného výsledku bychom docílili pomocí aplikování většího množství filter() operací.
- Opačnou operací je operace merge(), která kombinuje eventy dvou streamů dohromady.

selectKey()

- selectKey() provede transformaci klíče na novou hodnotu.

flatMapValues(), flatMap()

- Vstupem je jeden záznam, výstupem je 0, 1 nebo N záznamů.
- flatMapValues()
  - Neprovádí změny v klíčích
  - Pouze pro KStream
- flatMap()
  - Umožňuje změny v klíčích
  - Pouze pro KStream

to(), through(), repartition()

- KStream nebo KTable výstup je možné uložit do jakéhokoli topicu metodami:
  - to(): terminální operace – zapíše data do topicu a ukončí stream
  - through(): zapíše data do topicu a vrátí ho ve formátu nového KStream nebo Ktable. Od Kafka 2.6 deprecated.
    - repartition(): novější náhrada through()
    - Jak through(), tak repartition() je zkrácená varianta KStream#to() následované StreamsBuilder#stream()
      - Proč používat through() / repartition()? Vzhledem k tomu, že se data zapíšou do topicu, tak dojde k rozdělení jedné topologie do dvou sub-topologií a tím se zvýší paralelismus Kafka Streams aplikace.
      - U through() se musel předem ručně vytvořit topic, do kterého se budou ukládat data, u repartition() se topic vytvoří automaticky, má název KSTREAM-REPARTITION-XXXXX-repartition a dá se lehce v Kafka Streams aplikaci nastavit počet partitions tohoto topicu.

Stateful operations

- Use-cases:
  - Joining data
  - Aggregating data
  - Windowing Data
- Stav je uchováván ve „state store“. Kafka Streams aplikace může obsahovat větší množství state stores. State store je na úrovni Kafka Streams aplikace. Výchozí implementace je RocksDB.
- Pro zajištění failover je stav také v Kafce v topicu s názvem: <application.id>-<storeName>-changelog

RocksDB

- RocksDB je key-value store na straně Kafka Streams aplikace.
- https://www.confluent.io/blog/how-to-tune-rocksdb-kafka-streams-state-stores-performance/

KStream vs. KTable vs. GlobalKTable

- KStream je jako log, jede záznam po záznamu.
  - Příklad: máme 2 záznamy ve streamu:
    - (KEY, VALUE): (alice, 1), (alice, 3).
    - Když se provede SUM operace, bude výsledek: (alice, 4)
  - KStream je stateless
- KTable bere v úvahu poslední hodnotu záznamu s určitým klíčem
  - U předcházejícího příkladu by SUM operace vrátila (alice, 3)
  - KTable je stateful, interně je KTable implementována pomocí RocksDB a topicem v Kafkou. RocksDB obsahuje aktuální data tabulky a je uložena na disku Kafka Streams aplikace. RocksDB není fault-tolerant, proto jsou data také uložena v Kafka topicu.
    - https://stackoverflow.com/questions/52488070/difference-between-ktable-and-local-store
  - Pozor! KTable oproti KStream nezískává data real-time z topicu, ale načte poslední stav (key-value) do cache a ve výchozím nastavení po 30 vteřinách se provede flush cache a načte se aktuální stav (property commit.interval.ms). Nebo když je plná cache (ve výchozím nastavení 10 mb: property cache.max.bytes.buffering).
- GlobalKTable je jako KTable, ale data čte ze všech partition.

join, left join, outer join

Operator	Description
`join`	Inner join. The join is triggered when the input records on both sides of the join share the same key.
`leftJoin`	Left join. The join semantics are different depending on the type of join: • For stream-table joins: a join is triggered when a record on the *left side* of the join is received. If there is no record with the same key on the right side of the join, then the right value is set to null. • For stream-stream and table-table joins: same semantics as a stream-stream left join, except an input on the right side of the join can also trigger a lookup. If the right side triggers the join and there is no matching key on the left side, then the join will not produce a result.
`outerJoin`	Outer join. The join is triggered when a record on *either side* of the join is received. If there is no matching record with the same key on the opposite side of the join, then the corresponding value is set to null.

Type	Windowed	Operators	Co-partitioning required
`KStream-KStream`	Yes[a]	• `join` • `leftJoin` • `outerJoin`	Yes
`KTable-KTable`	No	• `join` • `leftJoin` • `outerJoin`	Yes
`KStream-KTable`	No	• `join` • `leftJoin`	Yes
`KStream-GlobalKTable`	No	• `join` • `leftJoin`	No

co-partitioning

- Při join operacích musí vstupní data (topicy) splňovat co-partitioning podmínku.
- Co je co-partitioning?
  - Input topic vlevo i vpravo musí mít stejný počet partitions.
  - Input topicy musí používat stejnou partitioning strategii (out-of-the-box tomu tak naštěstí je).
- Proč je co-partitioning vyžadován? Protože joiny se provádí na základě klíčů (musí se „spárovat“ klíč vlevo i vpravo). A na základě klíče se rozhoduje, v jaké partition daný záznam bude (z klíče se vypočítává hash).
- GlobalKTable nevyžaduje co-partitioning.

groupByKey, groupBy

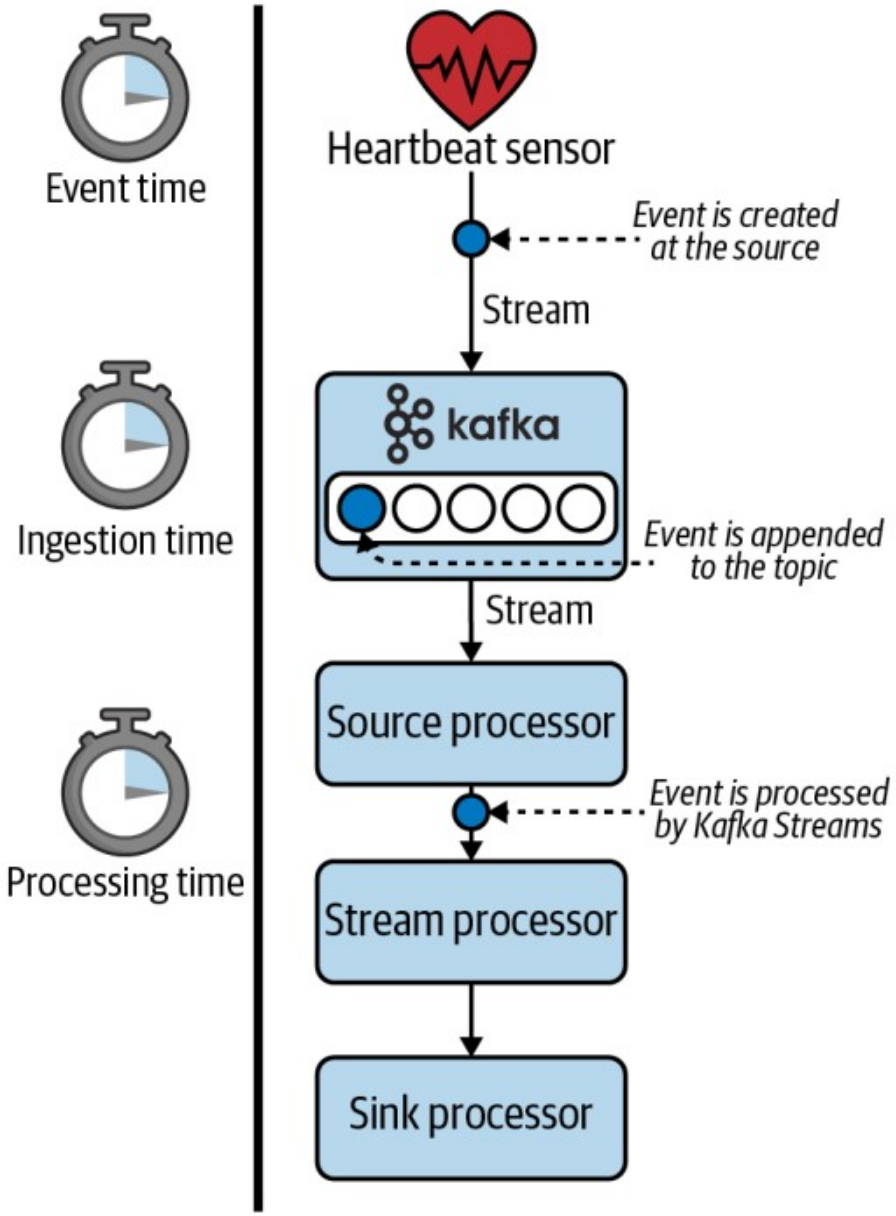
- Předtím, než je možné provádět agregace je nutné nejprve KTable nebo KStream, který se bude agregovat, sgrupovat:
  - groupByKey provede sgrupování bez transformace klíče
    - Efektivnější, neprovádí repartitioning
    - Je možné provést pouze na KStream
  - groupBy provede přemapování klíče a teprve poté sgrupování
    - Provádí repartitioning
    - Je možné provést na KStream i KTable

Aggregations

- Operace:
  - count
    - Klasický count.
  - reduce
    - Reduce algoritmus, oproti aggregate se liší v tom, že výsledný typ musí být stejný jako vstupní typ.
    - Dá se s ní implementovat min, max, sum.
  - aggregate
    - Dá se s ní implementovat to samé co s reduce operací, ale navíc pokročilejším způsobem. Plus například average.
- https://mail-narayank.medium.com/stream-aggregation-in-kafka-e57aff20d8ad

Timestamp

- Ve výchozím nastavení (od Kafka 0.10.0) když Producer pošle záznam do Kafky, tak je mu přiřazen timestamp (ten čas nastavuje Producer, jedná se o „Event time“).
- Když se vytváří ProducerRecord, tak se ten čas dá specifikovat (je datového typu long):
  - `new ProducerRecord<>(TOPIC, null, TIMESTAMP, KEY, VALUE);`
- Dá se také nastavit aby tento čas byl časem zařazení záznamu do topicu (pak by mu byl přiřazen čas „Ingestion time“):
  - https://kafka.apache.org/documentation/#log.message.timestamp.type



TimestampExtractor

- Out-of-the-box se při práci s časem používá výchozí timestamp, který byl probrán na předcházejícím snímku. V případě že je zapotřebí použít jiný (například definovaný v záznamu), pak je nutné implementovat TimestampExtractor, který je pak možné použít:
  - Globálně v Kafka Streams konfiguraci:
    - `streamsConfiguration.put(StreamsConfig.DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG, MyTimestampExtractor.class);`
  - Nebo ad-hoc:
    - `Consumed.with(Serdes.String(), temperatureValueSerde)`
      `.withTimestampExtractor(new MyTimestampExtractor());`

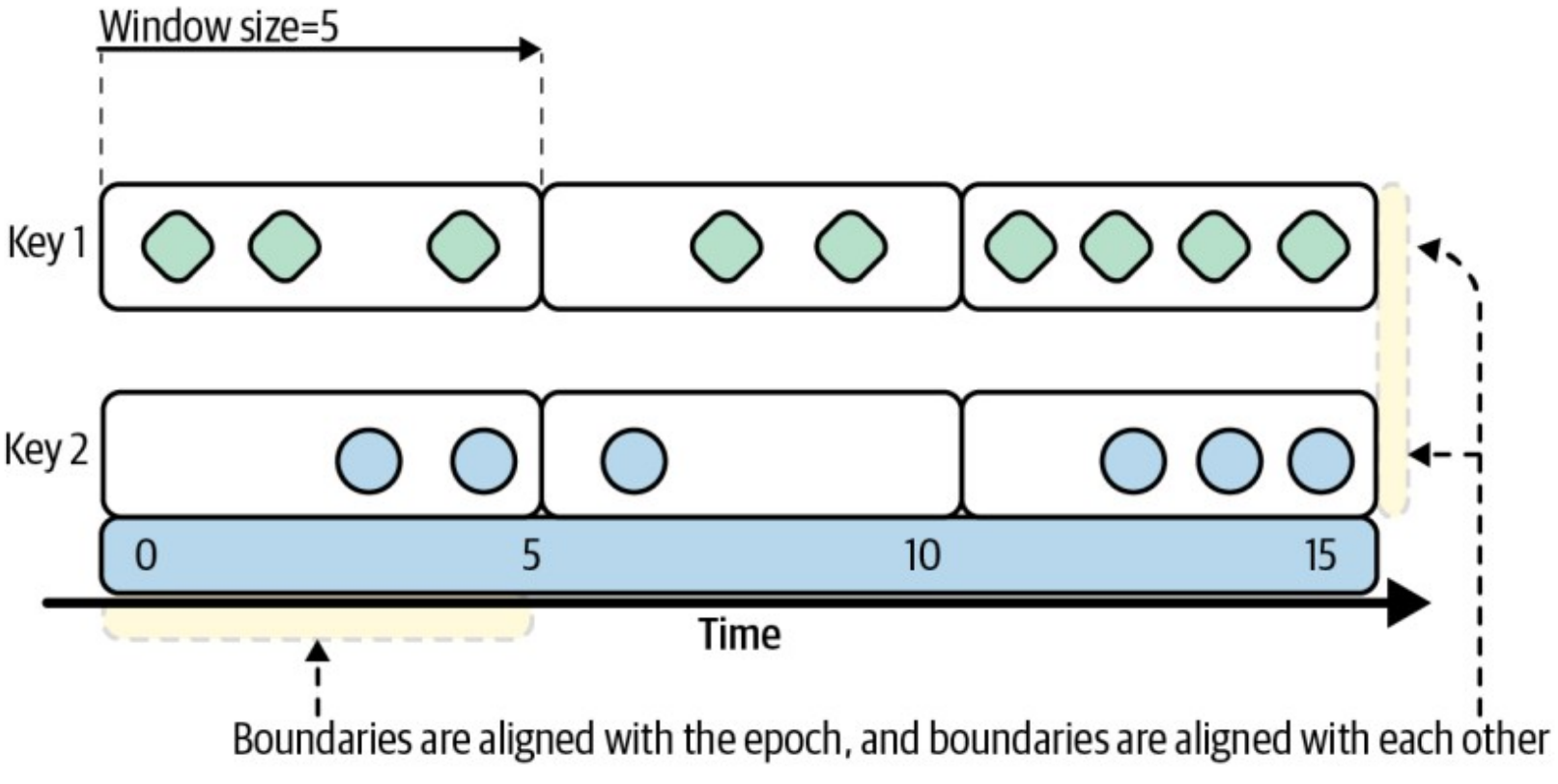
Windowing Streams

- Agregace nebo join operace pracují nad všemi záznamy. Často ale chceme výsledky za nějaké časové období. K tomu se používá koncept Windows. TimeWindow objekt je srovnaný s epoch time (tzn. okno o velikosti 60 000 ms bude mít hranice [0, 60 000), [60 000, 120 000) …
  - Poznámka: [= inclusive,) = exclusive
- Grace period
  - Velice důležitý koncept jak dlouho se bude čekat na záznamy pro nějaké okno, i když už to okno není platné. Pokud záznam přijde po ukončení grace periody, pak bude zahozen a nebude v dané okně již zprocesován.
- Supress
  - Další velice důležitý koncept. Ve výchozím nastavení se jednou za 30s provádí commit KTable. Abychom neměli v KTable mezivýsledky, je možné použít operaci supress(). Pozor! Tato operace nepoužívá state store, ale bufferuje záznamy v operační paměti.

Tumbling Window

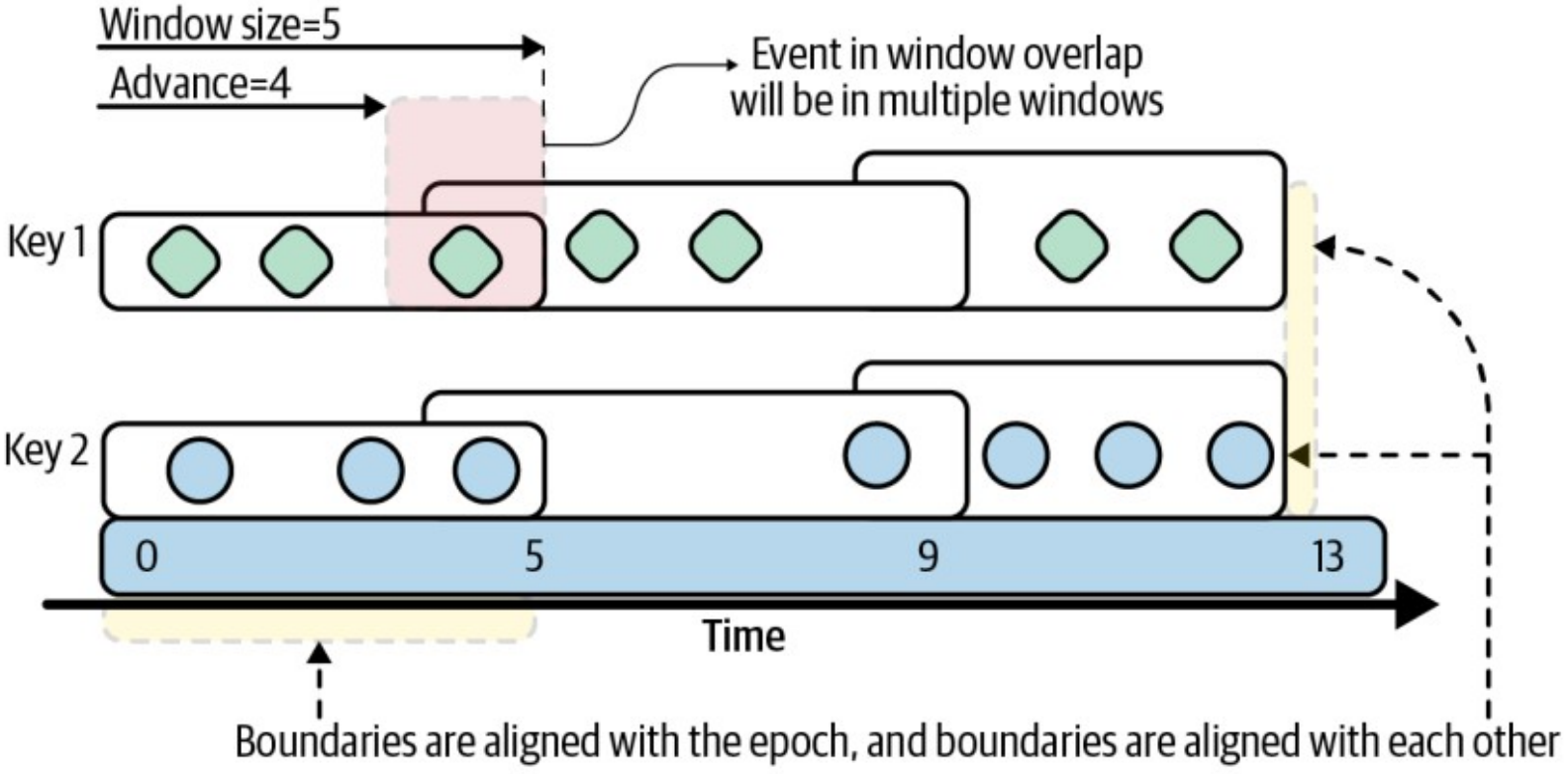
- Okna (Windows), která se nikdy nepřekrývají a jsou definovaná pomocí jedné property: window size.
  - `TimeWindows tumblingWindow =`

    `TimeWindows.ofSizeWithNoGrace(Duration.ofSeconds(5));`



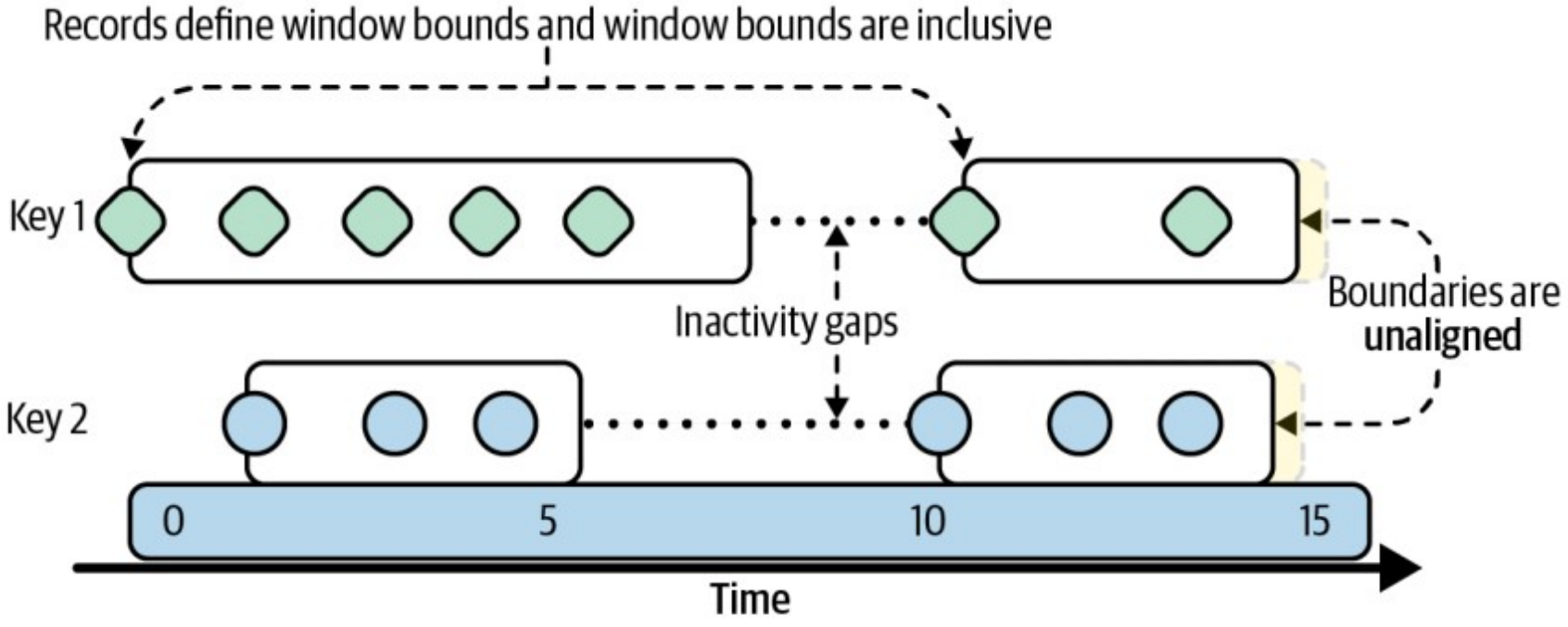
Hopping Window

- Okna, která jsou fixed-size a mohou se překrývat.
  - `TimeWindows hoppingWindow =`
    `TimeWindows.ofSizeWithNoGrace(Duration.ofSeconds(5))`
    `.advanceBy(Duration.ofSeconds(4));`



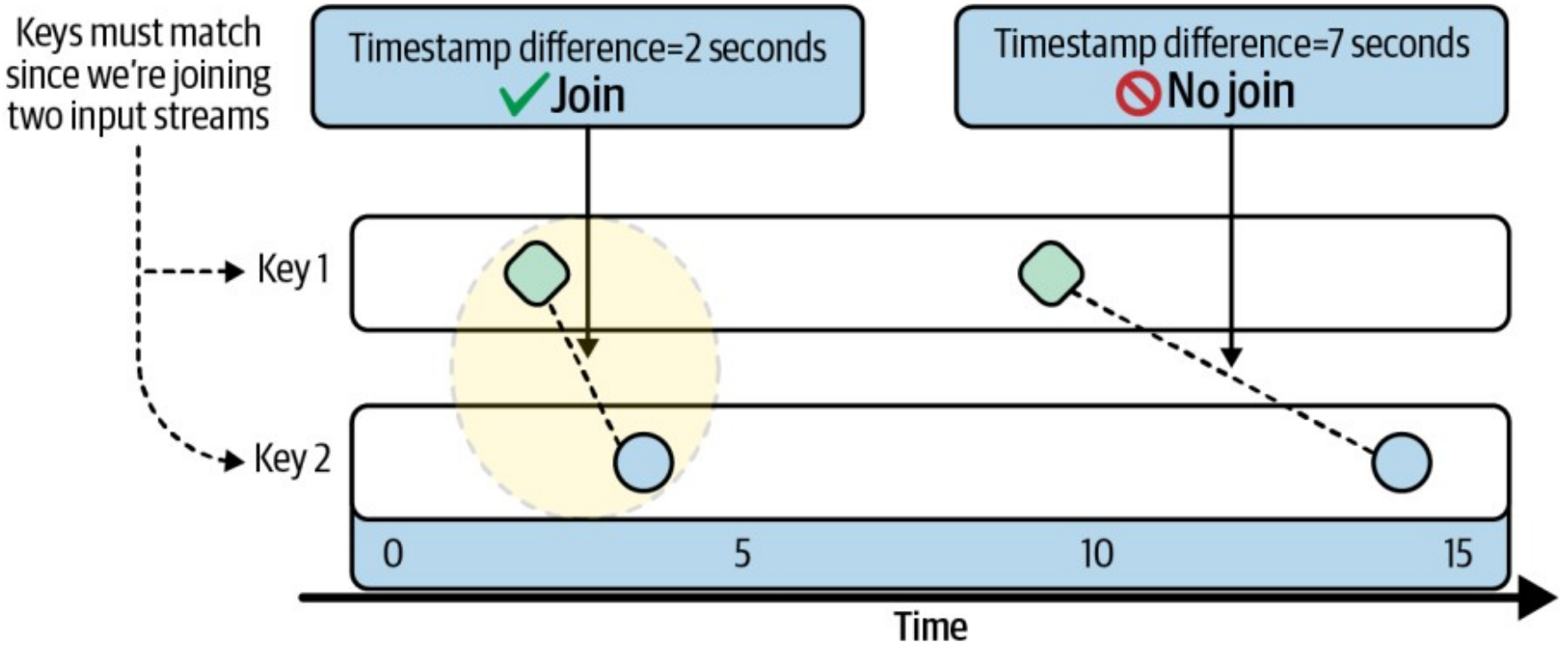
Session Window

- U Session Window se specifikuje inactivity gap. Jakmile dojde k této situaci (po tuto dobu nepřichází žádné zprávy), pak se stávající session window ukončí a při příchodu nové zprávy se započne nové session window.
  - `SessionWindows.ofInactivityGapWithNoGrace(Duration.ofSeconds(5))`



Join Window

- Dva záznamy se spojí, když rozdíl mezi jejich timestamp hodnotami je menší nebo roven window size.
  - `JoinWindows.of(Duration.ofSeconds(5));`
- U agregací je možné použít SlidingWindow, které není navázané na epoch, ale na timestamp záznamu obdobně jako JoinWindow:
  - `SlidingWindows.ofTimeDifferenceWithNoGrace(Duration.ofSeconds(5));`



Guarantees

- Kafka streamy out-of-the box garantují at_least_once Stream processing. Lze to změnit na „exactly once“, což garantuje atomicitu celého streamu:

```
properties.put(StreamsConfig.PROCESSING_GUARANTEE_CONFIG, "exactly_once_v2");
```

  - https://docs.confluent.io/platform/current/streams/developer-guide/config-streams.html#processing-guarantee
  - Poznámka: Exactly once vyžaduje cluster s minimálně třemi brokery!

Stream Exception Handling

- Kafka Streams v současnosti umožňují následující handlování výjimek:
  - Deserialization Exception Handler umožňuje odchytit výjimky, ke kterým může dojít při deserializaci záznamu.
  - Streams Uncaught Exception Handler odchytává výjimky, ke kterým může dojít při processingu streamu.
  - Production Exception Handler umožňuje odchytit výjimky, ke kterým může dojít při komunikaci s brokerem.

    - https://developer.confluent.io/learn-kafka/kafka-streams/error-handling/
    - https://developer.confluent.io/learn-kafka/kafka-streams/hands-on-error-handling/

Interactive Queries I.

- Doposud používaný State store se takto dá používat pouze když je jenom jedna instance Kafka Streams aplikace. V případě, že máme větší množství instancí v clusteru, tak musíme buď získat data z lokálního state store, nebo přes REST zavolat jinou Kafka Streams aplikaci, která vrátí data ze svého lokálního state store.
- Kafka Streams nám k tomu zjednodušuje nalezení instance, která obsahuje State Store s daty (discovery). K tomu je nutné nastavit proměnnou application.server. Jinak ale vystavení REST endpointu a komunikaci s ním musíme implementovat sami.
- https://kafka.apache.org/20/documentation/streams/developer-guide/interactive-queries.html
- https://github.com/confluentinc/kafka-streams-examples/tree/7.0.0-post/src/main/java/io/confluent/examples/streams/interactivequeries

Interactive Queries II.

State Store & Performance I.

- Z pohledu výkonu je problematický rebalancing, zejména když dojde k výpadku nějaké instance Kafka Streams aplikace. Následně musí dojít ke znovu-vytvoření state store z topiku v Kafce a zejména když je state store velký, tak může dojít k delšímu downtime. K řešení tohoto problému se používá:
  - Standby Replicas
    - Stav state store bude replikován na více instancí Kafka Streams aplikace. To se dá ovlivnit pomocí property num.standby.replicas (výchozí hodnota je 0, při hodnotě 1 bude jedna replika)
      - https://medium.com/transferwise-engineering/achieving-high-availability-with-stateful-kafka-streams-applications-cba429ca7238

State Store & Performance II.

- K rebalancingu dřív nebo později stejně dojde. Jak omezit jeho dopady? Zejména tak, aby ve state store bylo co nejmenší množství dat pomocí:
  - Tombstones: Když už není zapotřebí mít hodnotu ve state store, tak se nastaví na null.
  - Window retention: U windowed stores je možné nastavit retenci dat
  - Agressive topic compaction: KTable má zapnuté compaction, ale data na disku mohou zabírají více místa. Proč? Data jsou fyzicky na disku v segmentech. Vždy existuje aktivní segment, do kterého se zapisují data a až postupem času se z aktivních segmentů stanou neaktivní (když překročí velikost nebo uplyne nějaký čas). Pouze neaktivní segment je možné promazat. Tyto parametry ovlivňují rychlost čištění segmentů:
    - segment.bytes (default: 1GB): Maximální velikost segmentu
    - segment.ms (default: 7 dnů): Maximální doba kdy dojde k vytvoření nového segmentu i když nebylo naplněno segment.bytes
    - min.cleanable.dirty.ratio (default: 0.5): Jak často se bude čistit log. Ve výchozím nastavení se nebude čistit log, když je víc než 50% logu compacted.

Testing

- https://chrzaszcz.dev/2020/08/kafka-testing/
- https://www.baeldung.com/spring-boot-kafka-testing
- https://docs.confluent.io/platform/current/streams/developer-guide/test-streams.html
- https://blog.jdriven.com/2019/12/kafka-streams-topologytestdriver-with-avro/

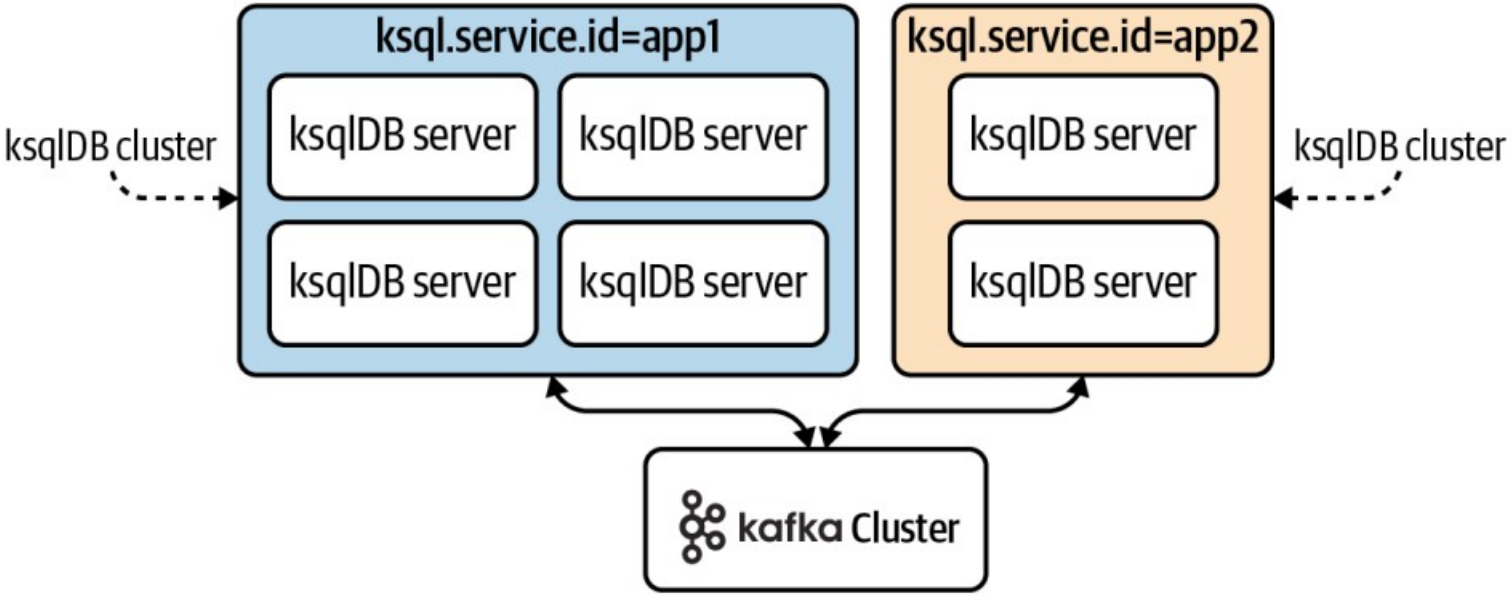
Kafka Streams Examples

- Další příklady od společnosti Confluent:
  - https://github.com/confluentinc/kafka-streams-examples/tree/7.0.0-post/src/main/java/io/confluent/examples/streams

ksqlDB

ksqlDB

- ksqlDB je nadstavba nad Kafka Streams a Kafka Connect. Každá instance ksqlDB funguje jako jedna microservice Kafka Streams aplikace a když se jim nastaví stejné ksql.service.id, pak to je to samé jako když více Kafka Streams instancí má stejné application.id.



Základní konfigurace

- Interactive vs. headless mode:
  - ksqlDB ve výchozím nastavení běží v „interactive mode“, kdy klienti mohou ovládat ksqlDB pomocí REST API. V tomto módu se všechny query ukládají do Kafka topicu _confluent-ksql-default__command_topic.
  - Alternativně se dá spustit v headless módu, ve kterém je vypnuté REST API a všechny query jsou definované při startu v souboru, jehož umístění se nastavuje pomocí property queries.file.
- Kafka Connect integrace má také dva módy:
  - External mode: Kafka Connect cluster může být mimo ksqlDB cluster, poté se jeho umístění nastavuje pomocí property ksql.connect.url.
  - Embedded mode: V tomto módu každá instance ksqlDB serveru plní zároveň roli Kafka Connect workeru. V takovém případě se nastavuje property ksql.connect.worker.config, která odkazuje na properties soubor, ve kterém je konfigurace workeru.

ksqlDB CLI & REST API

- ksqlDB CLI je konzolová aplikace, která slouží k ovládání ksqlDB:
  - https://docs.ksqldb.io/en/latest/operate-and-deploy/installation/cli-config/
- Pro ovládání ksqlDB instance je je také možné použít REST API (pokud neběží v headless módu):
  - https://docs.ksqldb.io/en/latest/developer-guide/api/index.html

use-cases

- Co je možné s ksqlDB dělat?
  - Vytvářet Kafka Connect source & sink connectory
  - Tvořit transient (ad-hoc) query
    - Tyto query nepřežijí restart serveru
  - Tvořit persistent query (query, které získají vstupní data, provedou transformaci a výsledek zapíší do topicu).
    - Tyto query přežijí restart serveru
  - Query mohou být buď ve formátu streamu, nebo ve formátu tabulky.
  - Přidávat záznamy do streamu

Books

- Kafka The Definitive Guide 2nd Edition:
  - https://www.oreilly.com/library/view/kafka-the-definitive/9781492043072/
- Kafka streams & ksqlDB:
  - https://www.amazon.com/Mastering-Kafka-Streams-ksqlDB-Real-Time/dp/1492062499
- Free ebooks (from Confluent):
  - https://www.confluent.io/resources/?language=english&assetType=ebook
  - Včetně výše doporučovaných knih

Kafka & Multiple Datacenters

- https://mbukowicz.github.io/kafka/2020/08/31/kafka-in-multiple-datacenters.html
- https://docs.confluent.io/platform/current/tutorials/examples/multiregion/docs/multiregion.html
- https://cloud.redhat.com/blog/geographically-distributed-stateful-workloads-part-four-kafka