Kafka

Krzysztof Jankiewicz

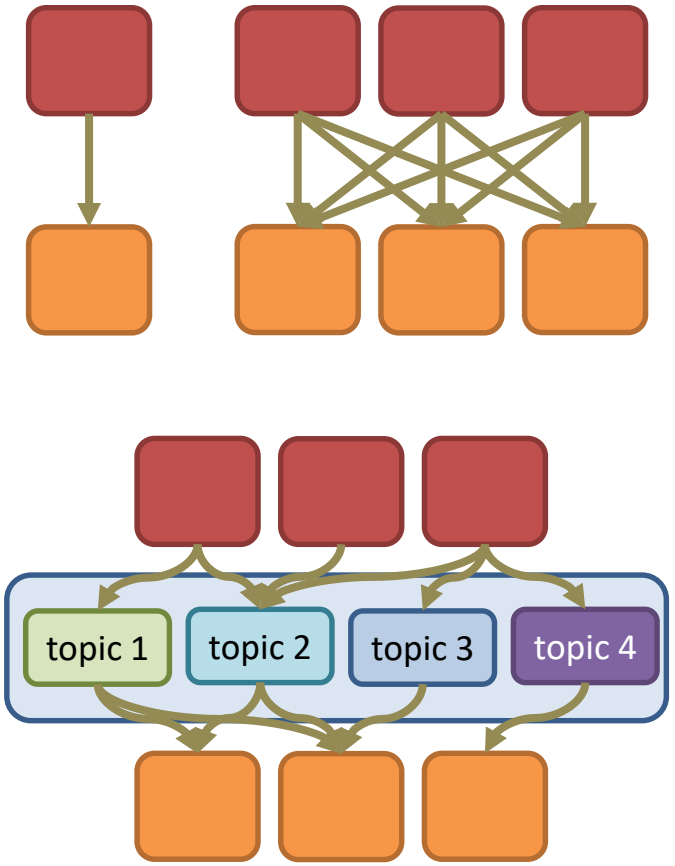
Plan

- Systemy wymiany wiadomości – wprowadzenie
- Wprowadzenie do Kafki
- Architektura
- Topik
- Konsumenci
- Architektura – uzupełnienie
- Administracja
- Producenci – API
- Konsumenci - API

Systemy wymiany wiadomości

- System wymiany wiadomości (*Messaging System*) są rozwiązaniami pozwalającymi na wymianę danych pomiędzy aplikacjami

- Dwa typy architektury
  - *Point to Point*
    - funkcjonują na zasadzie kolejki
    - każdy komunikat w kolejce przetwarzany jest przez jednego odbiorcę
  - *Publish-Subscribe*
    - funkcjonują w oparciu o tematy (*topiks*)
    - odbiorcy (*subscribers*) mogą zarejestrować się na otrzymywanie komunikatów z określonego tematu



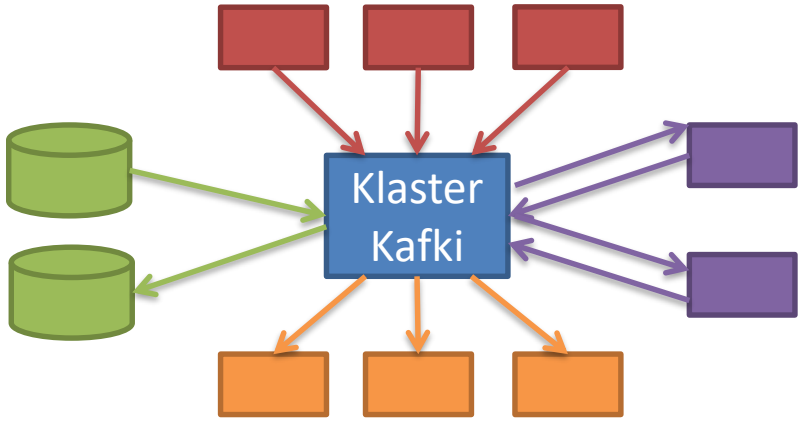
Wprowadzenie

- Kafka to oprogramowanie funkcjonujące jako broker wiadomości
- Napisana jest w Scali przez LinkedIn – otwarcie kodu w 2011
- Główne zastosowania
  - wiarygodna wymiana komunikatów pomiędzy aplikacjami
  - *aplikacje przetwarzania strumieniowego danych, które dokonują transformacji lub reagują aktywnie na dane pojawiające się w strumieniu*
- W roku 2014 w LinkedIn komunikacja za pośrednictwem Kafki przebiegała pomiędzy klastrami oraz pomiędzy centrami danych
  - 300 serwerów/brokerów w klastrach Kafki
  - 18 000 tematów
  - 140 000 partycji
  - wolumen przesyłanych komunikatów ok. 220 miliardów dziennie,
  - przepustowość zapisu 7 milionów komunikatów na sekundę (40 TB dziennie),
  - przepustowość odczytu 35 milionów na sekundę (160 TB dziennie),
  - chwilowe obciążenia do: 3,25mln komunikatów/s., 5,5GB/s IN, 18GB/s OUT
- Aby zdać sobie sprawę z popularności Kafki warto przytoczyć następujące statystyki dotyczące klientów warto zaglądnąć na jej stronę domową i zobaczyć, że korzysta z niej 80% firm z listy *Fortune 100*.

Patrz: `https://cwiki.apache.org/confluence/display/KAFKA/Powered+By`

Architektura

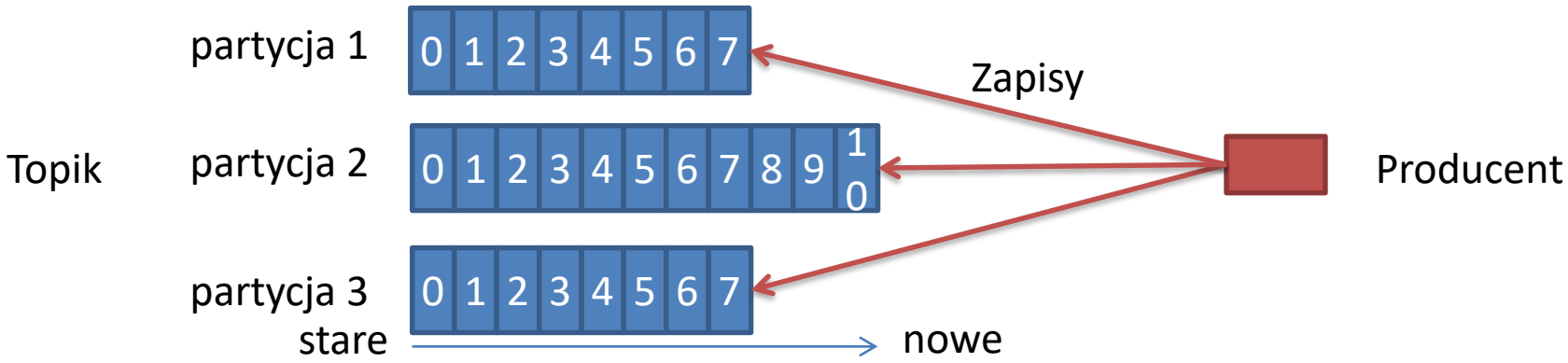
- Podstawy architektury
  - Kafka funkcjonuje w klastrze, na jednym lub wielu serwerach
  - Zapisuje strumienie rekordów w kategoriach zwanych tematami
  - Każdy rekord posiada klucz, wartość oraz etykietę czasową



- Podstawowe API
  - **Producenta** – pozwala aplikacjom dostarczać wiadomości do tematów
  - **Konsumenta** – pozwala zarejestrować się jako konsumenci wiadomości z tematów
  - **Strumieniowe** – dla aplikacji, które przetwarzają strumienie danych konsumując je z jednych topików i wytwarzając strumieniowe wyniki do innych topików.
  - **Połączeniowe** (*connector*) – pozwala na budowę mechanizmów wykorzystywanych do łączenia tematów z istniejącymi aplikacjami lub bazami danych.

Temat (*topic*) – cechy

- **Partycjonowane** – wiele serwerów, zrównoleglanie operacji, nieograniczona wielkość
- **Wielu konsumentów** – każdy sam zarządza offsetem, którym jest zainteresowany, offset jest krotką (offset, partycja, temat)
- Dane w tematach **usuwane** są po
  - konfigurowalnym okresie **czasu** (`retention.ms`) – domyślne ustawienie), lub
  - osiągnięciu przez partycję danego rozmiaru (`retention.bytes`)
- **Immutable** – postać wiadomości (rekordów) jest niezmienna
- To producenci decydują o tym, gdzie (do jakich partycji) będą zapisywać kolejne rekordy w temacie (np.: *round robin*, na podstawie klucza)

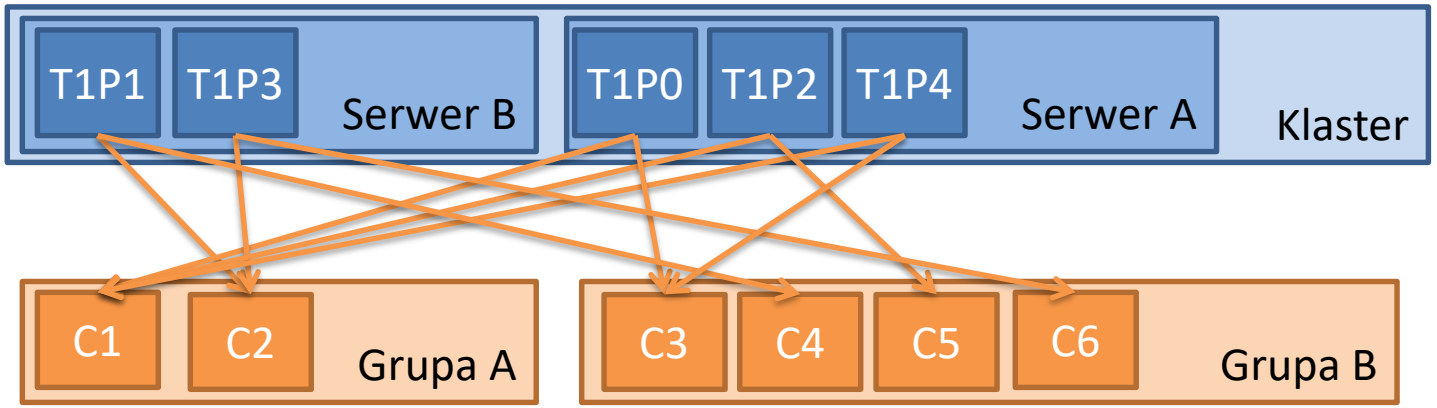


Temat (*topic*) – replikacja

- Każda partycja jest **replikowana** na wiele serwerów, jednego lidera (*leader*) oraz obserwatorów (*followers*)
- **Lider** odpowiedzialny jest za obsługę wszystkich operacji odczytu i zapisu, obserwatorzy pasywnie replikują lidera
- **Awaria** lidera powoduje, że jeden z obserwatorów przejmuje jego rolę
- Każdy węzeł w klastrze pełni rolę lidera dla jednych partycji oraz obserwatora dla innych

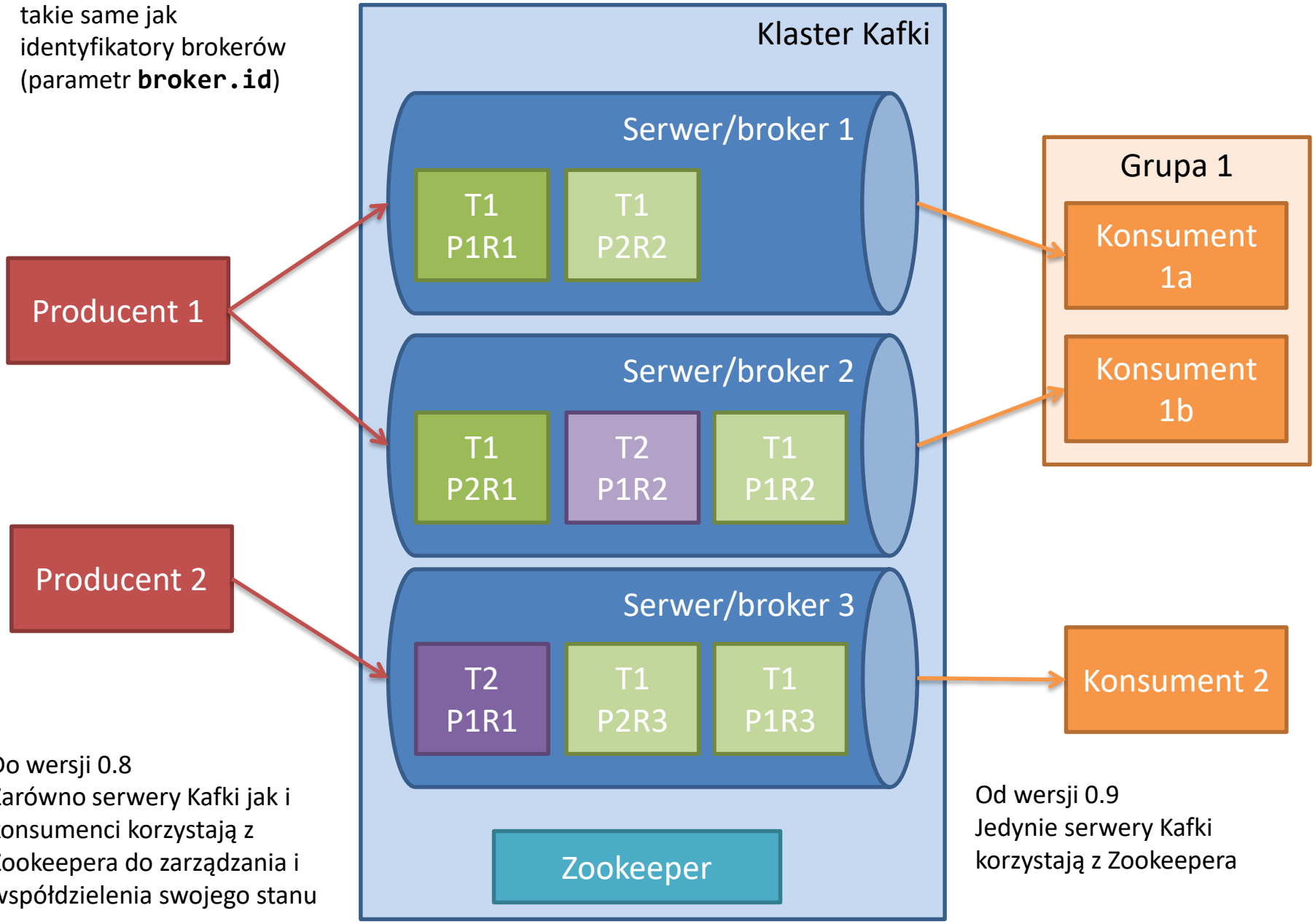
Konsumenci

- Konsumenci rejestrują się do odczytu wskazanych tematów **deklarując** swoją **grupę** konsumentów
- Konsumenci w ramach tej samej grupy mogą rezydować jako **oddzielne procesy** na tej samej maszynie, lub na oddzielnych maszynach
- Grupy konsumentów pozwalają **zrównoleglić** obsługę komunikatów dostarczanych przez serwer Kafki
- Każdy komunikat z danego tematu dostarczany jest do każdej zarejestrowanej grupy konsumentów
- Sposób dystrybucji komunikatów w Kafce przypisuje do każdego z konsumentów określone partycje. Przydział konsumenta do partycji odbywa się **dynamicznie**.
- Gwarancja kolejności dostarczania wiadomości spełniana jest na poziomie partycji. Jeżeli wymagana jest gwarancja kolejności na poziomie tematu, musimy ograniczyć się do jednego konsumenta w grupie
- Konsumenci przechowują wskaźniki odczytanych wiadomości w postaci offsetu



Architektura

Identyfikatory replik są takie same jak identyfikatory brokerów (parametr **`broker.id`**)



Do wersji 0.8
Zarówno serwery Kafki jak i konsumenci korzystają z Zookeepera do zarządzania i współdzielenia swojego stanu

Od wersji 0.9
Jedynie serwery Kafki korzystają z Zookeepera

Administracja

- Tworzenie tematu
  - CLI

```
bin/kafka-topics.sh --create --zookeeper localhost:2181 \
    --replication-factor 1 --partitions 1 --topic kafka-to-ss
```

  - Automatycznie
    parametr: `auto.create.topics.enable = true`
- Pobieranie listy tematów
  - CLI

```
bin/kafka-topics.sh --list --zookeeper localhost:2181
```

- Usuwanie tematu – UWAGA!
  - CLI

```
bin/kafka-topics.sh --delete --zookeeper localhost:2181 --topic kafka-to-ss
```

konieczny parametr: `delete.topic.enable = true`

Administracja

- Uzyskanie szczegółów dot. tematu
  - CLI

```
bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic test
Topic:test      PartitionCount:1        ReplicationFactor:1     Configs:
        Topic: test     Partition: 0    Leader: 1001    Replicas: 1001  Isr: 1001
```

Leader: broker wybrany jako lider
Isr: "in-sync replica", repliki synchronizowane z liderem

- Inne ważne operacje:
  - Dodawanie i usuwanie brokerów do/z klastra (wymaga przenoszenia istniejących partycji)
  - Rekonfiguracja tematów (np. zmiana liczby partycji lub poziomu replikacji)

```
kafka-topics.sh --alter --zookeeper localhost:2181 --topic test --partitions 3
```

  - Zarządzanie aplikacjami (dodawanie producentów i konsumentów)
  - Strojenie brokera – kluczowe parametry:
    - `num.io.threads` – X >= liczba dostępnych dysków
    - `num.network.threads` – dostosowana do (1) liczby współbieżnie działających producentów, konsumentów oraz (2) poziomu replikacji

Administracja

- Uzyskanie listy serwerów funkcjonujących w ramach klastra Kafki

```
zookeeper-shell.sh localhost:2181 ls /brokers/ids
```

- Zapoznanie się ze szczegółami na temat określonego serwera Kafki

```
zookeeper-shell.sh localhost:2181 get /brokers/ids/{ID_SERWERA}
```

Producenci

- Różne wymagania dotyczące:
  - utraty wiadomości
  - duplikowania wiadomości
  - opóźnienia
  - przepustowości
- Przykłady:
  - transakcje kartami bankowymi w dużym międzynarodowym banku
  - średniej wielkości portal chcący rejestrować statystyki dotyczące odwiedzin, kliknięć itp.
- W zależności od wymagań sposób implementacji producentów może być różny

Producenci – API

- API dostępne dla języków: Scala, Java, C/C++, Python, Ruby, itd.
- Podstawowa klasa udostępniająca API producenta: `KafkaProducer`
- Patrz: `https://kafka.apache.org/ver/javadoc/index.html?org/apache/kafka/clients/producer/KafkaProducer.html`

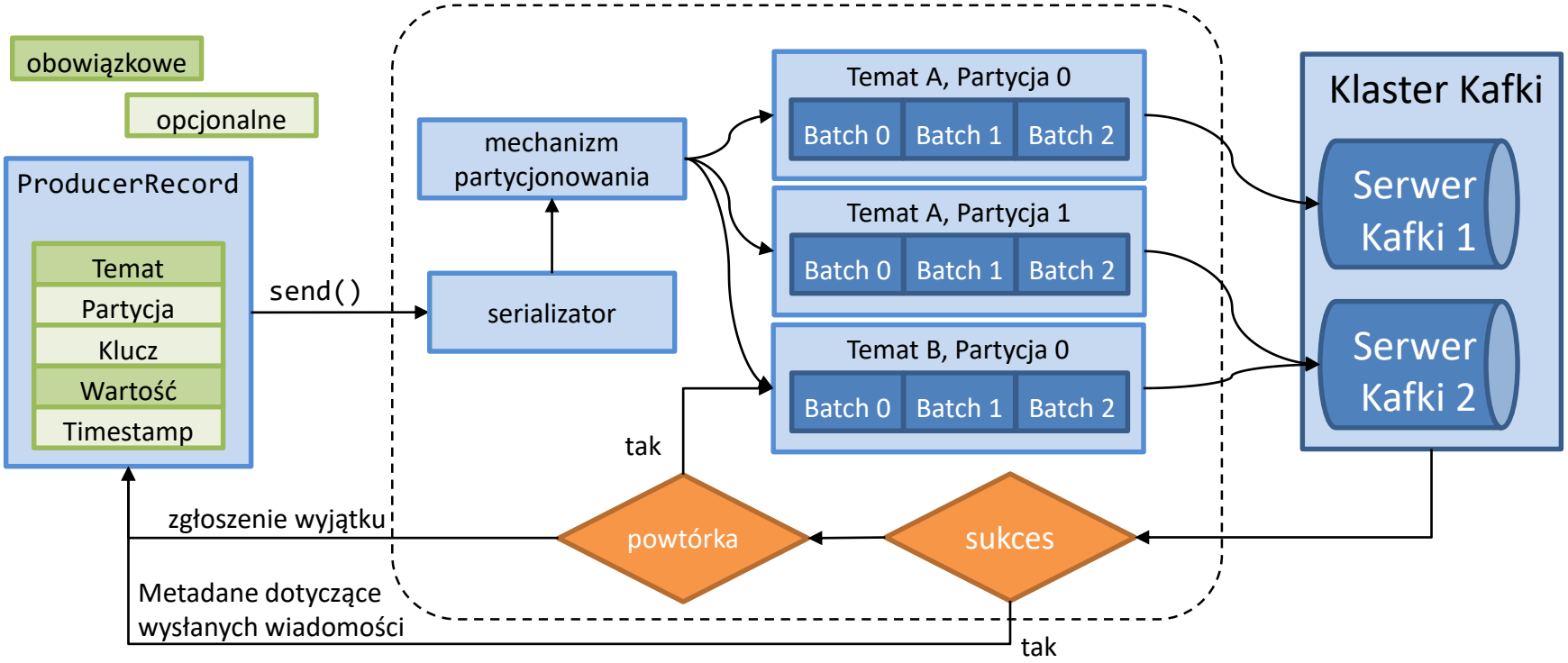
```
public class TestProducer {
     public static void main(String[] args) {
          Properties props = new Properties();
          props.put("bootstrap.servers", "localhost:6667");
          props.put("acks", "all");
          props.put("retries", 0);
          props.put("batch.size", 16384);
          props.put("linger.ms", 1);
          props.put("buffer.memory", 33554432);
          props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
          props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

          KafkaProducer<String, String> producer = new KafkaProducer<>(props);
          for (int i = 0; i < 100; i++)
             producer.send(new ProducerRecord<String, String>("my-topic",
                                                Integer.toString(i), Integer.toString(i)));
          producer.close();
     }
}
```

Producenci – API

- Jak widać na przykładzie, API dla producentów jest proste
- Pod spodem jednak dzieje się znacznie więcej

- Istotnymi elementami są:
  - mechanizmy serializacji
  - metody partycjonowania
  - wysyłka oparta o przetwarzanie wsadowe
  - wysyłka synchroniczna i asynchroniczna
  - ponawianie nieudanych wysyłek wiadomości



Producenci – API – parametry i wysyłka

- Tworzenie producenta

```
KafkaProducer<String, String> producer = new KafkaProducer<>(props);
```

- Obowiązkowe parametry
  - `bootstrap.servers` – serwery z którymi producent będzie ustanawiał połączenie
  - `key.serializer` – nazwa klasy wykorzystywanej do serializacji kluczy
  - `value.serializer` – nazwa klasy wykorzystywanej do serializacji wartości
- Trzy sposoby wysyłki wiadomości
  - wyślij i zapomnij *(fire and forget)* – brak informacji o utraconych wiadomościach, brak obsługi błędów

```
try {
     producer.send(new ProducerRecord<String, String>("my-topic",
                   Integer.toString(i), Integer.toString(i)));
} catch (Exception e) {
     e.printStackTrace();
}
```

  - wysyłka synchroniczna – metoda `send` zwraca obiekt `Future`, który możemy wykorzystać (metoda `get`) do uzyskania rezultatu wysyłki
  - wysyłka asynchroniczna – wysyłka z funkcją callback, która otrzymuje wynik wysyłki

Patrz: `http://tutorials.jenkov.com/java-util-concurrent/java-future.html`

Producenci – API – wysyłka

- Wysyłka synchroniczna
  - metoda `Future.get()` pobiera obiekt `RecordMetadata`, który zawiera szczegóły dotyczące wysyłki komunikatu (np. offset, numer partycji, etykieta czasowa wysyłki) lub zwraca wyjątek

```
ProducerRecord<String, String> record =
  new ProducerRecord<String, String>("my-topic", "klucz", "wartość");
try {
     producer.send(record).get();
} catch (Exception e) {
     e.printStackTrace();
}
```

- Wysyłka asynchroniczna
  - odbiór informacji zwrotnych, przy zachowaniu dużej wydajności wysyłki

```
class ProducerCallback implements Callback {
    @Override
    public void onCompletion(RecordMetadata recordMetadata, Exception e) {
        if (e != null) {
            e.printStackTrace();
        }
    }
}

producer.send(new ProducerRecord<String, String>(params[2],
                String.valueOf(line.hashCode()), line),new ProducerCallback());
```

Producenci – API – dodatkowe parametry

- `client.id` – identyfikator klienta
- `acks` – liczba partycji w temacie, która musi odebrać wiadomość aby móc uznać wysyłkę za poprawną
  - 0 – producent nie czeka na nic i zakłada, że wysyłka powiodła się
  - 1 – leader
  - all – wszystkie repliki
- `linger.ms` – czas oczekiwania na kolejne komunikaty celem wysłania wsadu (jeśli wsad będzie pełny dla określonej partycji – parametr `batch.size`, wówczas wysyłka może nastąpić wcześniej)
- `buffer.memory` – ilość pamięci producenta przeznaczona na buforowanie wszelkich komunikatów

```
props.put("bootstrap.servers", "localhost:6667");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("acks", "all");
props.put("retries", 0);
props.put("batch.size", 16384);
props.put("linger.ms", 1);
props.put("buffer.memory", 33554432);
```

Producenci API – serializacja

- Obiekty na poziomie aplikacji producenta muszą przed wysyłką zostać zserializowane.
- Podstawowe mechanizmy dla liczb, ciągów znaków, czy tablic bajtów mogą być niewystarczające
  - `StringSerializer`
  - `ShortSerializer`
  - `IntegerSerializer`
  - `LongSerializer`
  - `DoubleSerializer`
  - `BytesSerializer`
- Alternatywy:
  - własne serializatory
  - JSON,
  - Apache Avro,
  - Thrift,
  - Protobuf
- Apache Avro
  - popularny w świecie Big Data
  - stworzony przez Douga Cuttinga
  - niezależny od języka programowania
  - zewnętrzna definicja schematu
  - obsługuje zmiany schematu

Patrz: `https://www.baeldung.com/kafka-custom-serializer`

Producenci API – partycjonowanie

- Wiadomości Kafki są parami klucz-wartość
- Klucz nie jest obowiązkowy

```
ProducerRecord<String, String> record =
  new ProducerRecord<String, String>("my-topic", "wartość");
```

- Klucze mają dwa cele
  - dodatkowa informacja związana z wartością
  - rozpraszają komunikaty pomiędzy partycjami – wszystkie wiadomości z takim samym kluczem trafiają do tej samej partycji tematu
- Domyślna metoda partycjonowania
  - round-robin w przypadku pustego klucza
  - hash klucza w przeciwnym przypadku
- Partycja jest stała dla danego klucza do czasu zmiany liczby partycji w temacie
- Apache Kafka dostarcza dodatkowe mechanizmy partycjonowania
  - RoundRobinPartitioner
  - UniformStickyPartitioner
- Domyślne rozwiązania, oraz dodatkowe mechanizmy sprawdzają się w przypadku w miarę równomiernej dystrybucji wartości klucza.

Producenci API – partycjonowanie

- W przypadku gdy rozkład wartości klucza jest znacząco zaburzona, warto sięgnąć po możliwość definiowania własnego mechanizmu partycjonowania
- Interfejs partycjonera zawiera metody: `configure`, `partition`, `close`.
- Warto wykorzystywać metodę `configure` do odpowiedniej parametryzacji mechanizmu partycjonowania

```
public class SkewPartitioner implements Partitioner {
    public void configure(Map<String, ?> configs) {}
    public int partition(String topic, Object key, byte[] keyBytes,
                         Object value, byte[] valueBytes,
                         Cluster cluster) {
        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();
        if ((keyBytes == null) || (!(key instanceof String)))
        throw new InvalidRecordException("We expect all messages to have a key");
        if (((String) key).equals("BigKey"))
            return numPartitions - 1; // BigKey has its own partition
        return (Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1));
    }
    public void close() {}
}
```

```
props.put("partitioner.class","com.example.bigdata.SkewPartitioner");
```

Producenci

- Kafka dostarcza szeregu skryptów pozwalających przetestowanie działania mechanizmów Kafki np.:
  - producent z konsoli

```
bin/kafka-console-producer.sh \
     --broker-list sandbox.hortonworks.com:6667 --topic kafka-to-ss
```

  - connector dostarczający zmieniającą się zawartość pliku

```
bin/connect-standalone.sh \
  /home/maria_dev/labs/spark/connect-standalone_1.properties \
  /home/maria_dev/labs/spark/connect-file-source_1.properties
```

connect-standalone_1.properties

```
bootstrap.servers=sandbox.hortonworks.com:6667
key.converter.schemas.enable=true
value.converter.schemas.enable=true
key.converter=org.apache.kafka.connect.storage.StringConverter
value.converter=org.apache.kafka.connect.storage.StringConverter
internal.key.converter=org.apache.kafka.connect.storage.StringConverter
internal.value.converter=org.apache.kafka.connect.storage.StringConverter
internal.key.converter.schemas.enable=false
internal.value.converter.schemas.enable=false
offset.storage.file.filename=/tmp/connect.offsets
offset.flush.interval.ms=10000
```

connect-file-source_1.properties

```
name=local-file-source
connector.class=FileStreamSource
tasks.max=1
file=/tmp/apachelogdir/access_log_Jul95.log
topic=kafka-to-ss
```

Konsumenci

- API dostępne dla języków: Scala, Java, C/C++, Python, Ruby, itd.
- Podstawowa klasa udostępniająca API konsumenta: `KafkaConsumer`
- Patrz: https://kafka.apache.org/ver/javadoc/index.html?org/apache/kafka/clients/consumer/KafkaConsumer.html

```
offset = 300, key = 0, value = 0
offset = 301, key = 1, value = 1
offset = 302, key = 2, value = 2
offset = 303, key = 3, value = 3
```

```
public class TestConsumer {
     public static void main(String[] args) {
          Properties props = new Properties();
          props.put("bootstrap.servers", "localhost:6667");
          props.put("group.id", "test");
          props.put("enable.auto.commit", "true");
          props.put("auto.commit.interval.ms", "1000");
          props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
          props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
          KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
          consumer.subscribe(Arrays.asList("my-topic"));
          while (true) {
               ConsumerRecords<String, String> records =
                                                  consumer.poll(Duration.ofMillis(100));
               for (ConsumerRecord<String, String> record : records)
                    System.out.printf("offset = %d, key = %s, value = %s%n",
                                      record.offset(), record.key(), record.value());
          }
     }
}
```

Klient Scala (Spark)

```
$SPARK_HOME/bin/spark-shell --master=local[2] --packages org.apache.kafka:kafka-clients:ver

import java.util
import java.util.Properties

import org.apache.kafka.clients.consumer.{ConsumerConfig, KafkaConsumer}
import scala.collection.JavaConversions._
import org.apache.kafka.common.requests.MetadataResponse.TopicMetadata
import org.apache.kafka.common.requests.MetadataRequest._
import org.codehaus.jackson.map.deser.std.StringDeserializer

val properties = new Properties()
properties.put("bootstrap.servers", "localhost:6667")
properties.put("group.id", "ZeppelinKafkaConsumer")
properties.put("enable.auto.commit", "true")
properties.put("auto.commit.interval.ms", "1000");
properties.put("key.deserializer","org.apache.kafka.common.serialization.StringDeserializer")
properties.put("value.deserializer","org.apache.kafka.common.serialization.StringDeserializer")

val consumer = new KafkaConsumer[String, String](properties)
consumer.subscribe(util.Arrays.asList("kafka-tt"))

val recordsFromConsumer = consumer.poll(100)
val recordsFromConsumerList = recordsFromConsumer.records("kafka-tt").toList
val lastOffset = recordsFromConsumerList.last.offset()
val partitionsAssigned = consumer.assignment()
val endOffsetsPartitionMap = consumer.endOffsets(partitionsAssigned)
val topicPartition = consumer.assignment().toList
val endOffsets = consumer.endOffsets(partitionsAssigned)
val consumerLag = endOffsets.get(topicPartition.head) - lastOffset
```

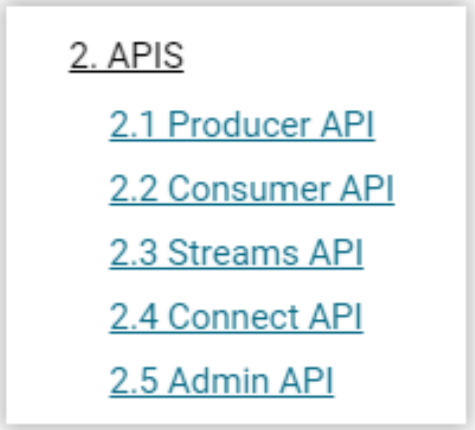
Konsumenci

- Podobnie jak w przypadku producentów Kafka dostarcza szeregu skryptów np.:
  - konsument "na konsolę"

```
bin/kafka-console-consumer.sh \
  --bootstrap-server localhost:6667 \
  --topic my-topic --from-beginning
```

Pozostałe API

- `https://kafka.apache.org/documentation/`



Podsumowanie

- Systemy wymiany wiadomości – wprowadzenie
- Wprowadzenie do Kafki
- Architektura
- Topik
- Konsumenci
- Architektura – uzupełnienie
- Administracja
- Producenci – API
- Konsumenci - API