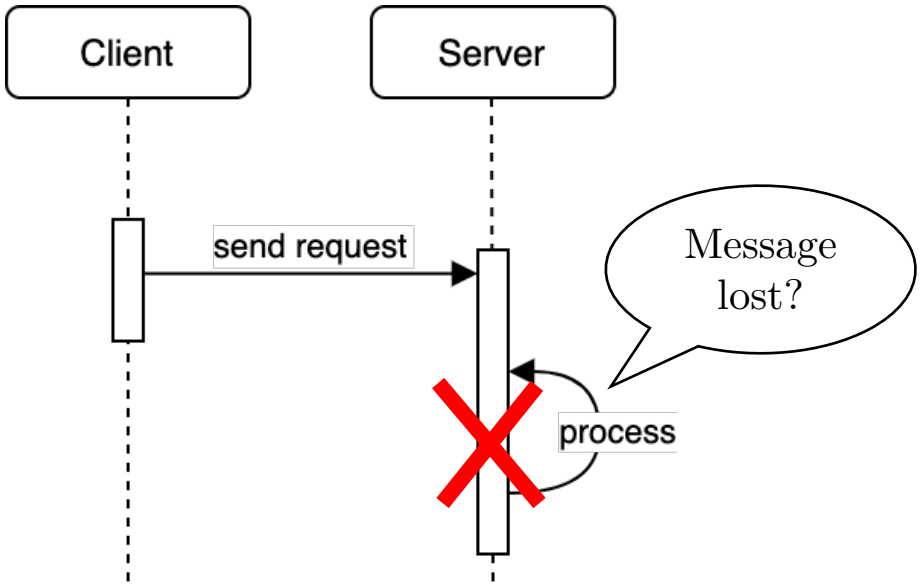
Distributed Messaging Systems

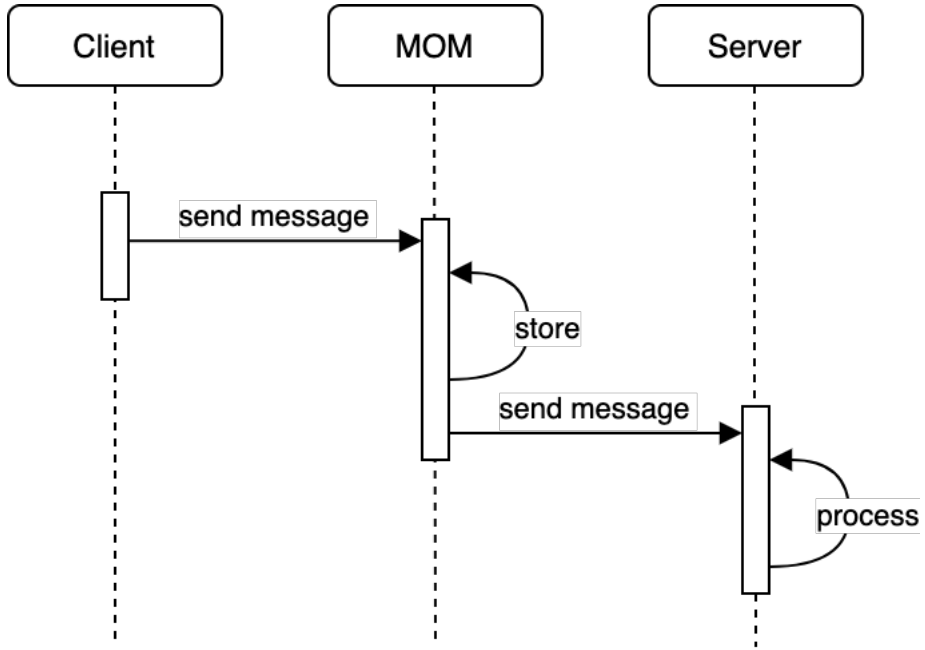
Agenda

- Asynchronous communication pattern and message-oriented middleware
  - Point-to-point delivery
  - Publish-subscribe delivery
- Short introduction to Apache Kafka
- Apache Kafka as a distributed system
  - Highly available Kafka cluster
  - Resilient message producer
  - Resilient message consumer
- Event-driven architecture

Message-Oriented Middleware

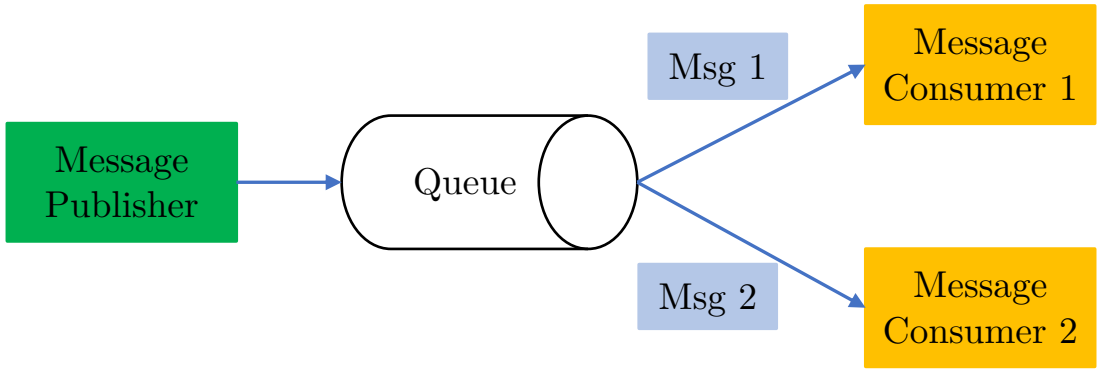
- Asynchronous communication pattern does not require caller to wait for service response
- Decoupled execution of two services
  - Client applications can continue to execute other tasks
  - Clients do not suffer from unavailability of the server
  - Server does not notice burst in client requests (MOM as a buffer)
- What happens if server instance terminates abnormally, and does not complete its task?



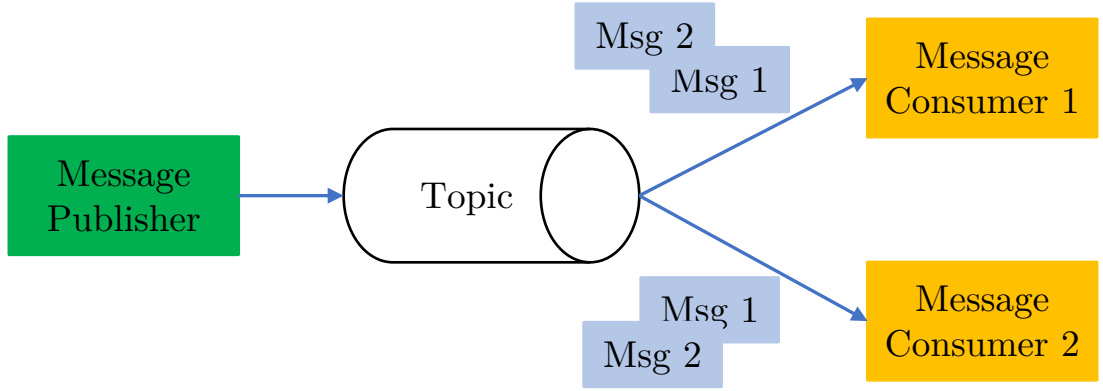


Message-Oriented Middleware

- Point-to-point – each message delivered to one subscriber
  - Load balances traffic across consumers
  - Publishers send messages to logical destination of “queue” type
  - Consumers receive messages from queue



- Publish-subscribe – each message delivered to all subscribers
  - Broadcast, one-to-many delivery
  - Publisher sends message to logical destination of ”topic” type
  - Consumers receive messages from topic



Message-Oriented Middleware

- Examples of messaging solutions:
  - IBM MQ – traditional messaging solution, used even on mainframes
  - TIBCO Enterprise Message Service
  - Apache ActiveMQ – popular open-source of JMS specification
  - Rabbit MQ
  - Apache Kafka – streaming platform
  - Apache Pulsar
  - Eclipse Mosquitto – perfect for Internet of Things, implements MQTT standard

Short Introduction to Apache Kafka

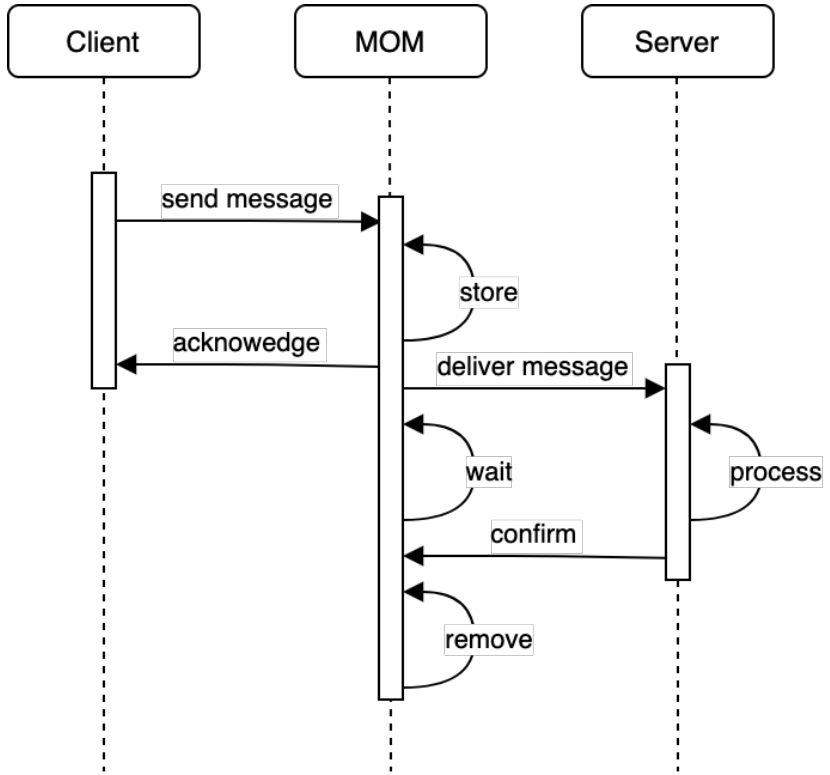
- Started as internal project at LinkedIn in 2009
- Solve the problem of real-time asynchronous ingestion of large amounts of event data from the company website to analytical big-data cluster
  - Traditional messaging queue solutions could not handle the load
- Apache Kafka is a distributed, persistent streaming platform
  - Do not think in terms of messaging system, but processing stream of events

"Kafka ecosystem at LinkedIn is sent over 800 billion messages per day which amounts to over 175 terabytes of data. [...] At the busiest times of day, we are receiving over 13 million messages per second, or 2.75 gigabytes of data per second."

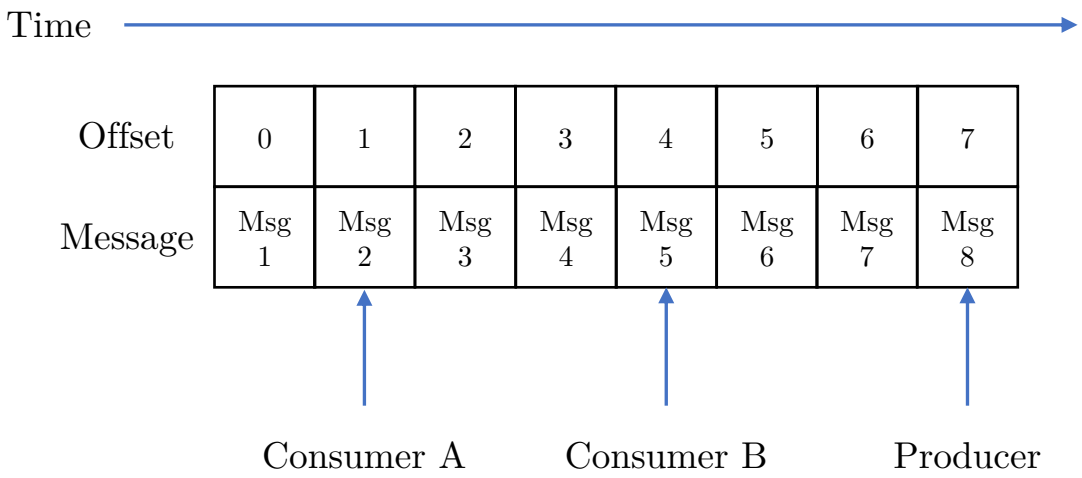
Source: https://engineering.linkedin.com/kafka/running-kafka-scale

Short Introduction to Apache Kafka

- Traditional messaging solutions experience a bottleneck of single server handling each message separately (delivery and acknowledgement)
- Transactions support requirements

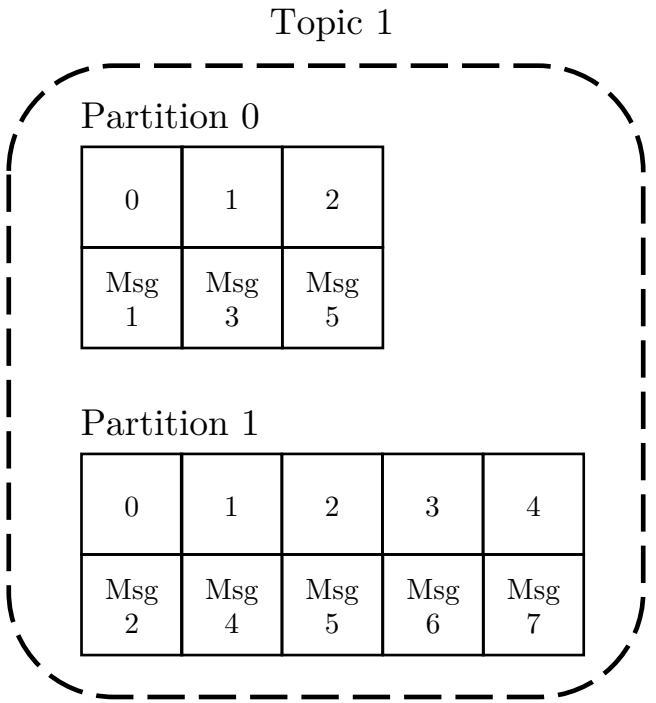


- Persist all messages in append-only commit log
- Maintain only a pointer to last acknowledged position of each consuming application
- Do not handle each message separately



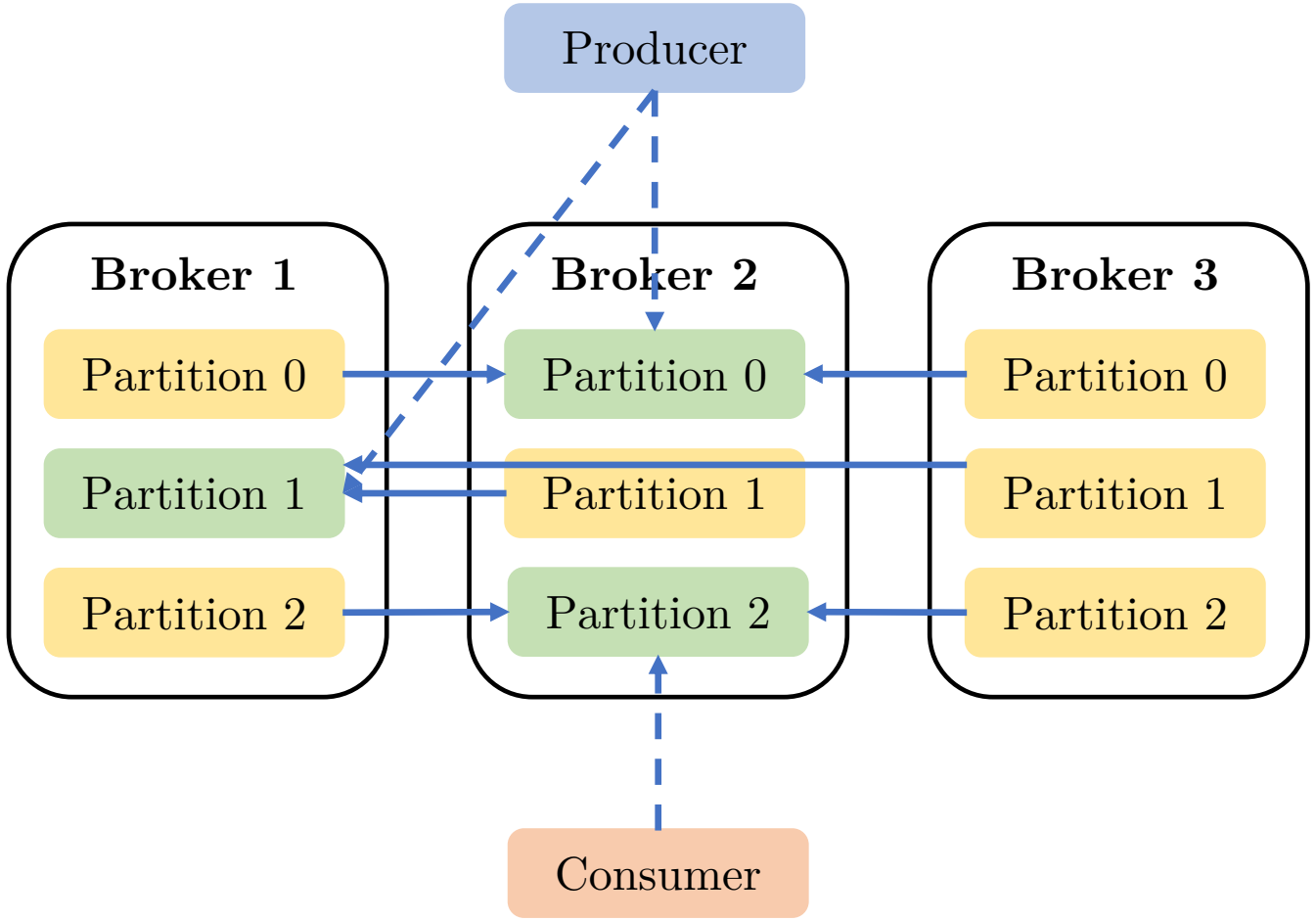
Apache Kafka as a Distributed System

- Message producers send data to logical destinations called topics
- Each topic can be divided into multiple partitions (data sharding)
  - Kafka message contains key, value and optional headers
  - Sharding performed on message key
  - Unique position of given message within partition is called an offset
- Each topic-partition is backed by independent commit log data structure



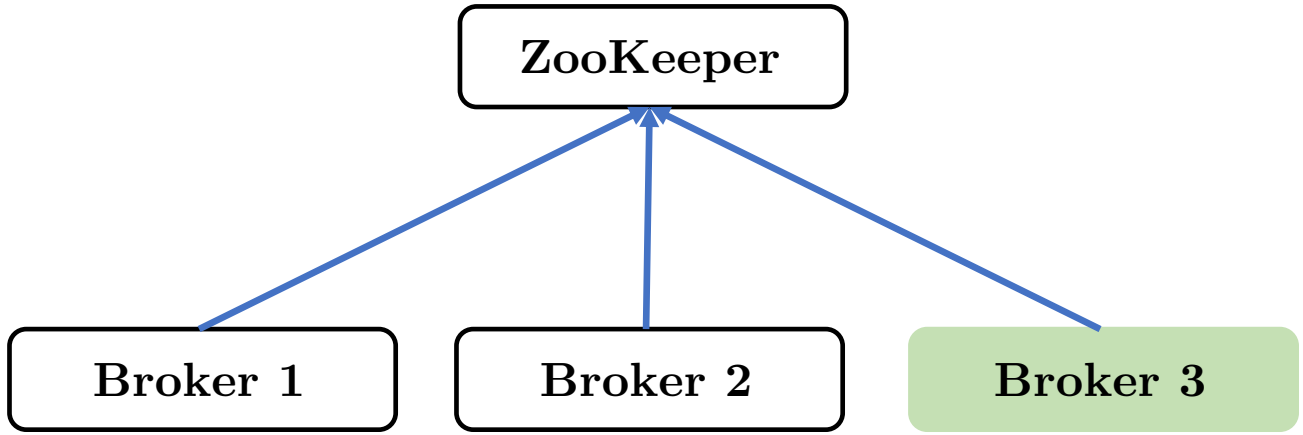
Apache Kafka as a Distributed System

- Topics can be replicated according to configured replication factor
- Partition leader handles producer and consumer requests
- Replicas continuously poll data from partition leader
- Kafka is a (long) poll-based system
  - Less frequent polling for large amount of data improves throughput
  - More frequent polling, fetching smaller amount of data with each request, improves latency
- Partition leader maintains list of in-sync replicas (ISR), who can become new partition leader when it terminates



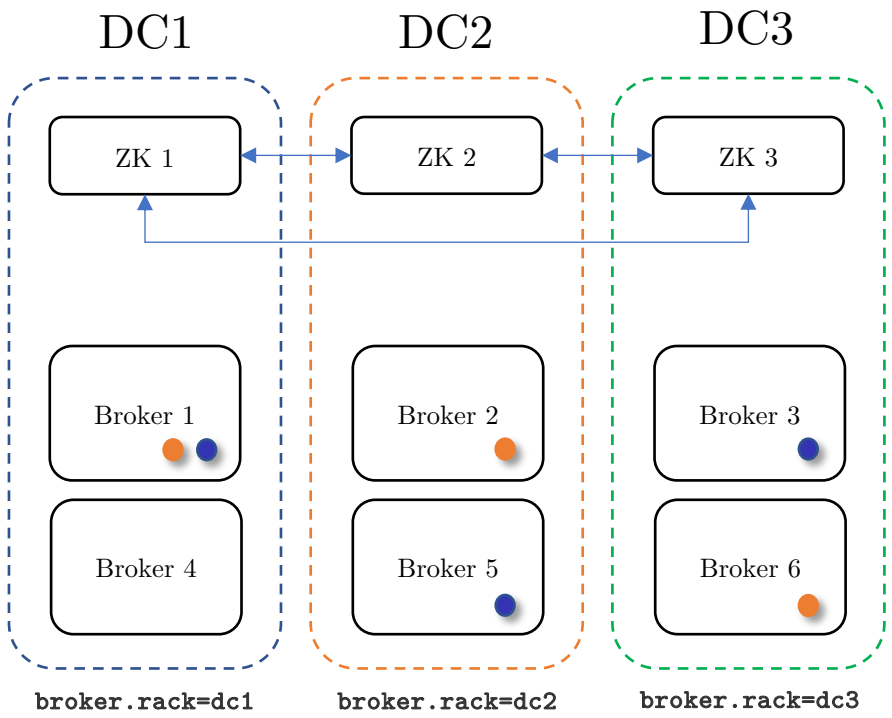
Apache Kafka as a Distributed System

- One node in Kafka cluster acts as controller
  - Controller manages the state of partitions and replicas
  - Performs cluster-wide administrative tasks
- Kafka (until version 3.0) requires ZooKeeper for cluster coordination
  - Discovery of brokers joining and leaving the cluster
  - Configuration store for topic metadata, quotas, security
  - Persists current cluster state, e.g. ISR replicas
- Kafka 3.0 implements internal RAFT protocol



Highly Available Kafka Cluster

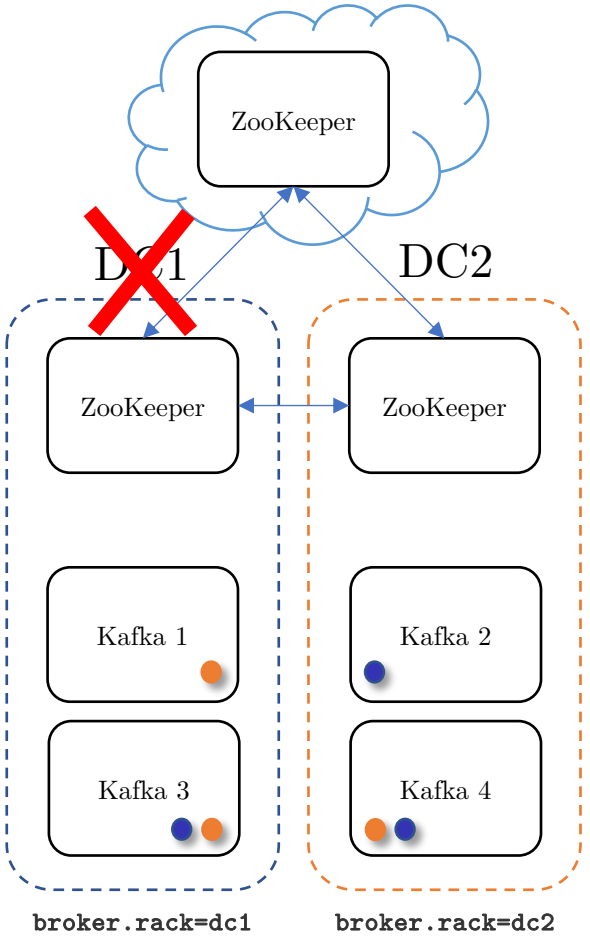
- Assume availability of 3 remote locations, where network round-trip is less than 30 ms.
- Deploy 3, 5, or 7 ZooKeeper instances depending on the size of your environment
- Deploy at least 3 Kafka brokers in different availability zones
  - `default.replication.factor >= 3`
  - `min.insync.replicas = 2`
  - `broker.rack = us-west` (according to availability zone name)
- At least single copy of each record present in each data centre



Highly Available Kafka Cluster

- Assume availability of only 2 remote data centres
- Deploy third ZooKeeper instance in the public cloud (minimal hardware requirements)
  - Often referenced as “2,5” DC deployment
- Issue with Kafka replication when one DC becomes unavailable

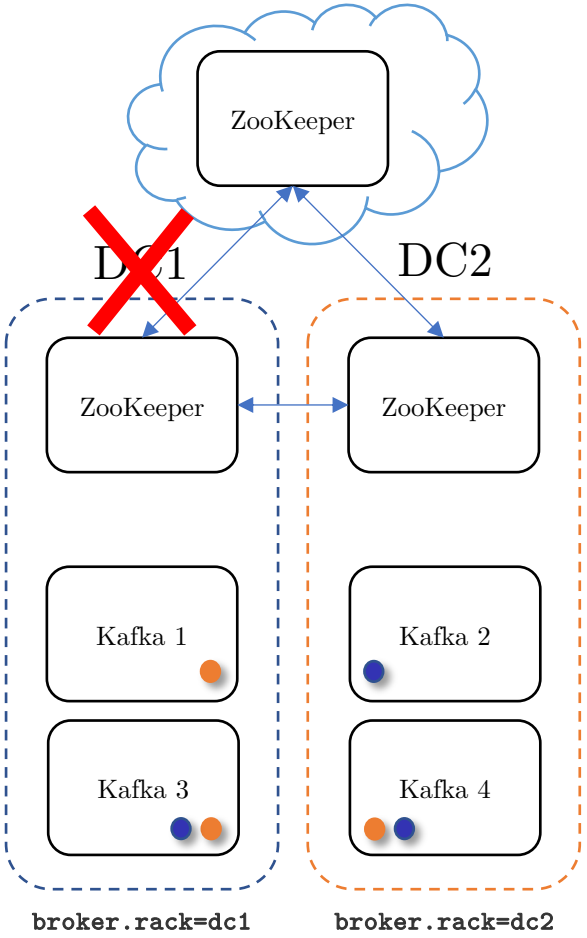
Replication factor	min ISR	Outcome
3	2	Replication factor of 3 implies that one of the data centers will hold two replicas, whereas the other one only one. When data center managing two replicas experiences outage, given partition will not accept new write requests – 2 acknowledgements required but only 1 replica online.
4	2	Both data centers persist two replicas of each partition. Producer waits for only 2 acknowledgements, so we do not guarantee that latest data has been successfully replicated to remote data center before sending acknowledgement to message producer.



Highly Available Kafka Cluster

- Choice between availability and consistency

Replication factor	min ISR	Unclean leader election	Outcome
4	2	true	**Availability over consistency** Possibility of data loss. Data not guaranteed to replicate to both DC as described on previous slide.
3	2	false	**Consistency over availability** Latest copy of acknowledged data guaranteed in both data centers. Administrator needs to reconfigure Kafka topics and decreasing min ISR to 1, when one of DCs becomes unavailable.

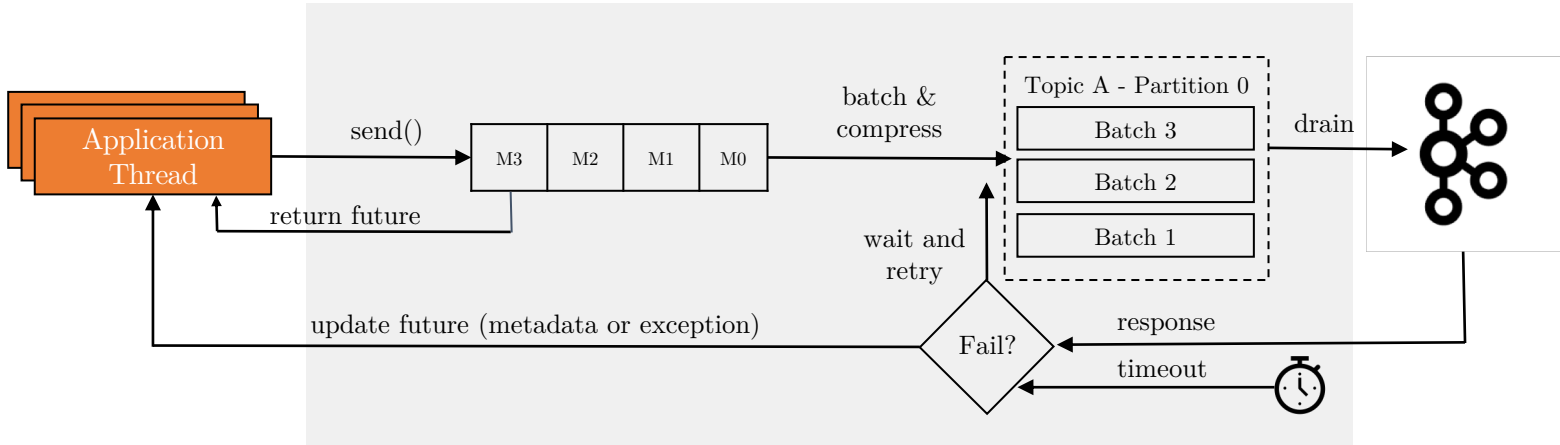


Source: https://www.slideshare.net/ConfluentInc/the-foundations-of-multidc-kafka-jakub-korab-solutions-architect-confluent-kafka-summit-london-2019

Resilient Message Producer

- Kafka producer is thread-safe and shall be shared between all application threads
- Send method of Kafka producer is asynchronous, remember to wait for final outcome
  - Fun-out pattern: trigger multiple requests and collect future objects in list, wait for all futures to complete
- Messages are accumulated in internal buffer (`buffer.memory` defines size)
  - Buffer is flushed when it is full, or configured period elapses (`linger.ms` parameter)
  - Batching requests together improves throughput (and compression ratio), but degrades latency
  - Automatic retry in case of failure

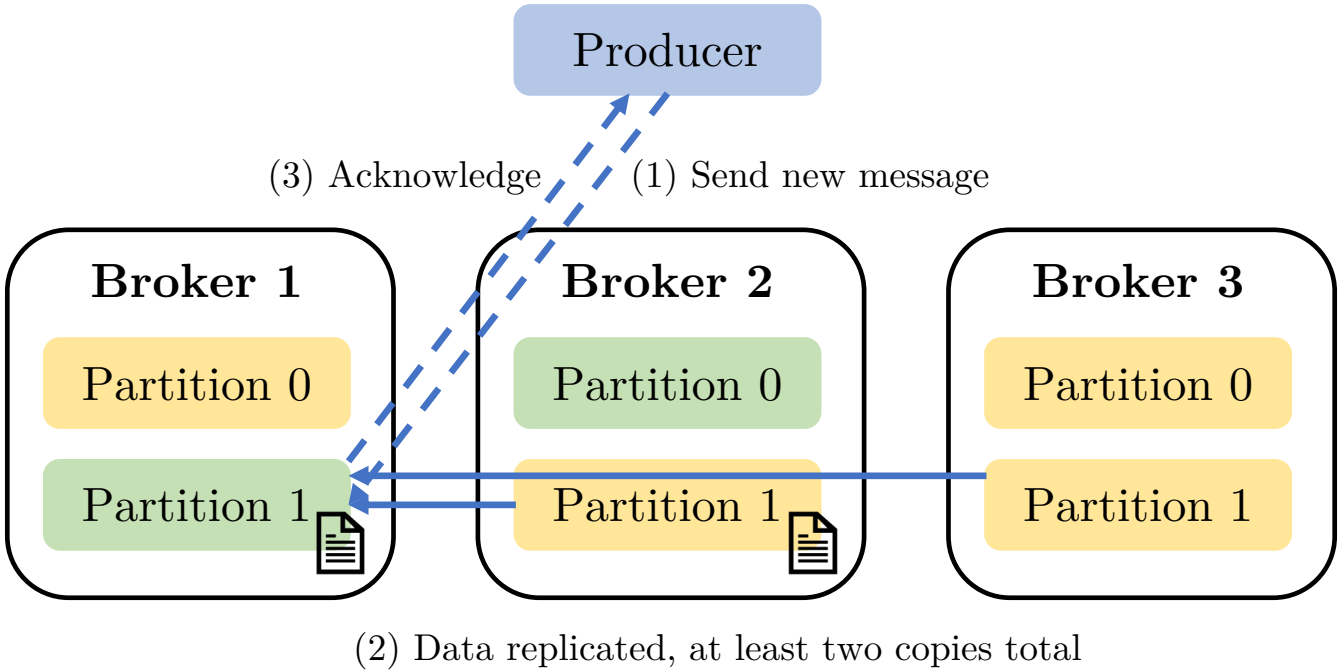
```
List<CompletableFuture> futures = new ArrayList<>();
for (MyEvent event : events) {
   futures.add(
      kafkaTemplate.send(
         myTopic, event.getKey(), event
      ).completable()
   );
}
CompletableFuture.allOf(
   futures.toArray(new CompletableFuture[0])
).join();
```



Source: https://www.slideshare.net/ConfluentInc/reliability-guarantees-for-apache-kafka

Resilient Message Producer

- Producer sends new messages always to partition leaders
- Producer may wait for different acknowledgement level
  - `acks = 0` – producer does not wait for any confirmation
  - `acks = 1` – producer waits until message is persisted on partition leader
  - `acks = ALL` – message is replicated and persisted to required number of in-sync replicas

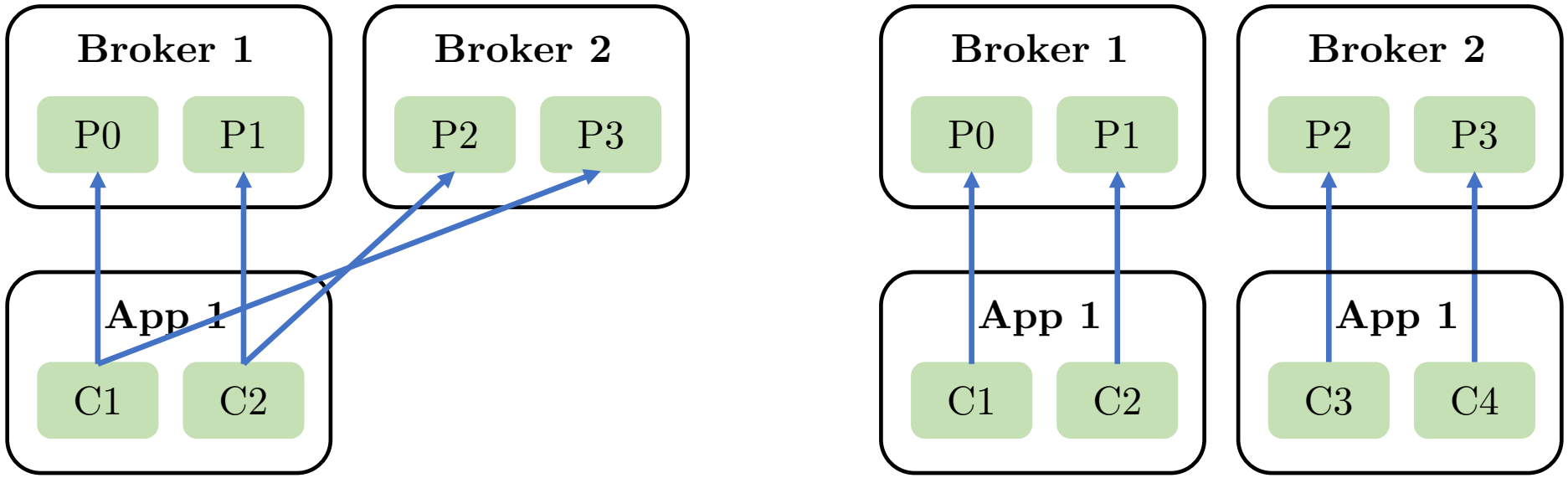


Resilient Message Producer

- Configuration of Kafka producer for durability
  - `enable.idempotence = true` – prevents generation of duplicated messages in some edge cases
  - `retries = Integer.MAX_VALUE`
  - `acks = ALL`
  - `max.block.ms = Long.MAX_VALUE` – wait infinitely for space in internal buffer
  - Remember to gracefully shutdown producer instance, otherwise internal buffer will not be drained before application terminates
  - Handle exceptions raised by send function
  - Use `delivery.timeout.ms` parameter to configure timeout of `send()` method, not to block thread infinitely

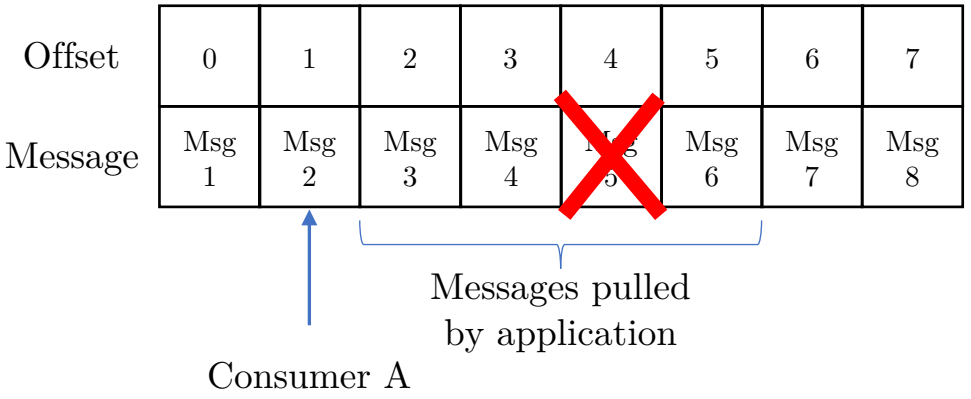
Resilient Message Consumer

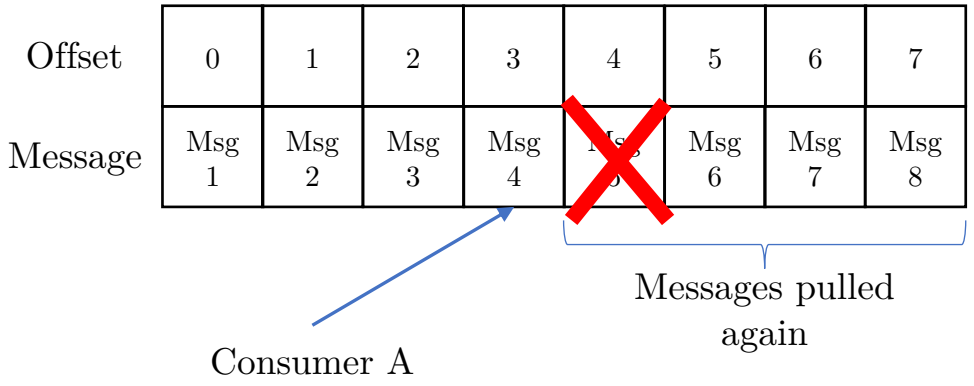
- Kafka consumer is not thread-safe
  - Each processing thread should use its own Kafka consumer instance
  - Each consumer polls data from one or more dedicated partitions
- Consumers with the same consumer group name, are treated as single application
- Kafka will distribute topic-partitions evenly across members of consumer group
- Topic-partition is an unit of parallelism in Kafka



Resilient Message Consumer

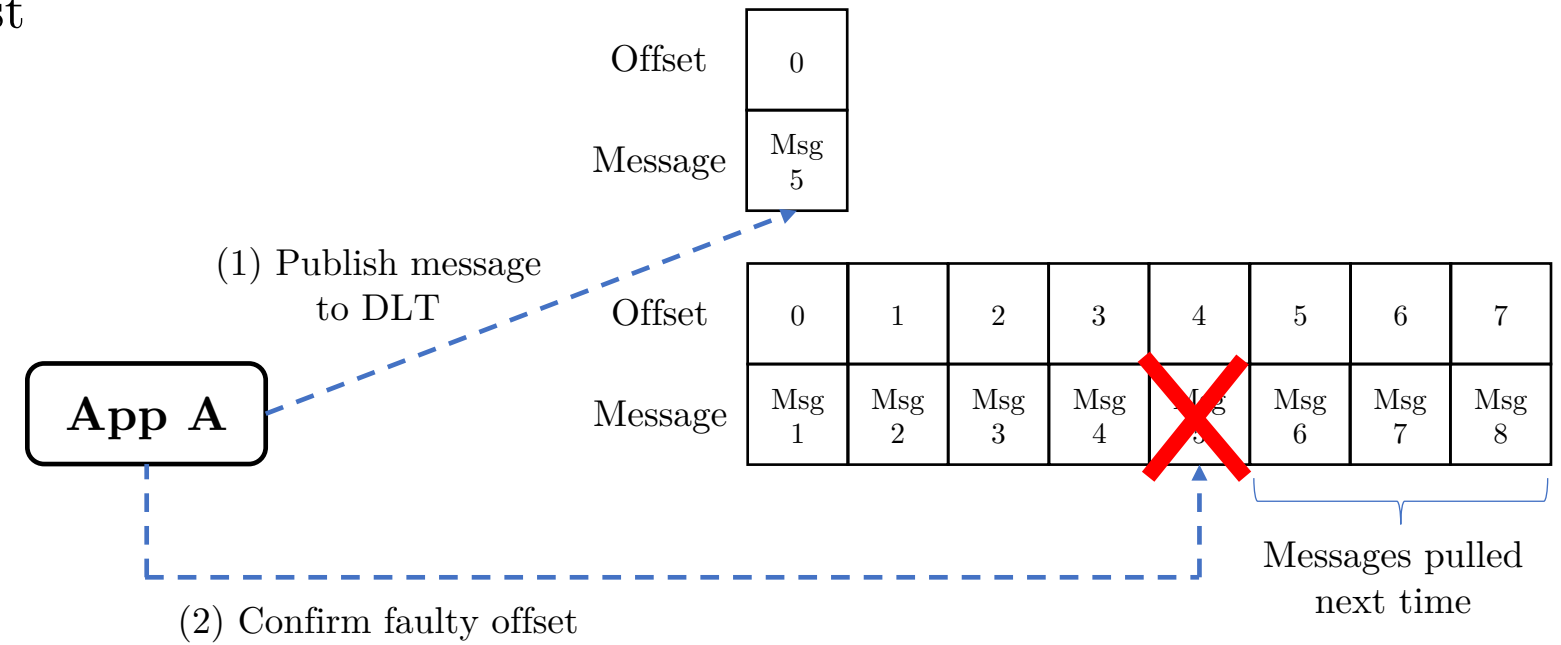
- Kafka consumer long-polls broker for new messages
  - Best practice to poll multiple records with each request, increased throughput
- Consuming application processes messages
- Kafka consumer acknowledges processed offsets
  - Automatic sends periodically acknowledgement every N seconds. Not recommended, no control over failures in processing records
  - Explicit client confirmation after processing offsets
  - Retry processing given offset few times, and if still failing, put corresponding record to DLT topic





Resilient Message Consumer

- In distributed systems, order of executed actions matters
  - First send the message to DLT
  - Only when message was successfully send to DLT, confirm faulty offset
  - Possible duplicates on DLT (at-least-once guarantee)
- At-most-once guarantee in case of reversed sequence of operations
  - Potential message lost



Resilient Message Consumer

- Key configuration parameters
  - `fetch.max.wait.ms` – controls maximum waiting time for new messages
  - `fetch.min.bytes`, `fetch.max.bytes` – control minimum and maximum size of returned payload for single poll request. Kafka will wait at most `fetch.max.wait.ms`, if the collected data is less than `fetch.min.bytes`.
  - `max.poll.records` – controls maximum number of records returned with each poll
  - `max.poll.interval.ms` – detects live-locked consumer
  - `session.timeout.ms` – if broker misses heartbeats for given time, it assumes that consumer process has terminated
  - `auto.offset.commit = false` – always use explicit client-acknowledgement

```
while (true) {
    ConsumerRecords<String, MyEvent> records = consumer.poll(100);
    for (ConsumerRecord<String, MyEvent> record : records) {
        // process
    }
    consumer.commitSync();
}
```

Resilient Message Consumer

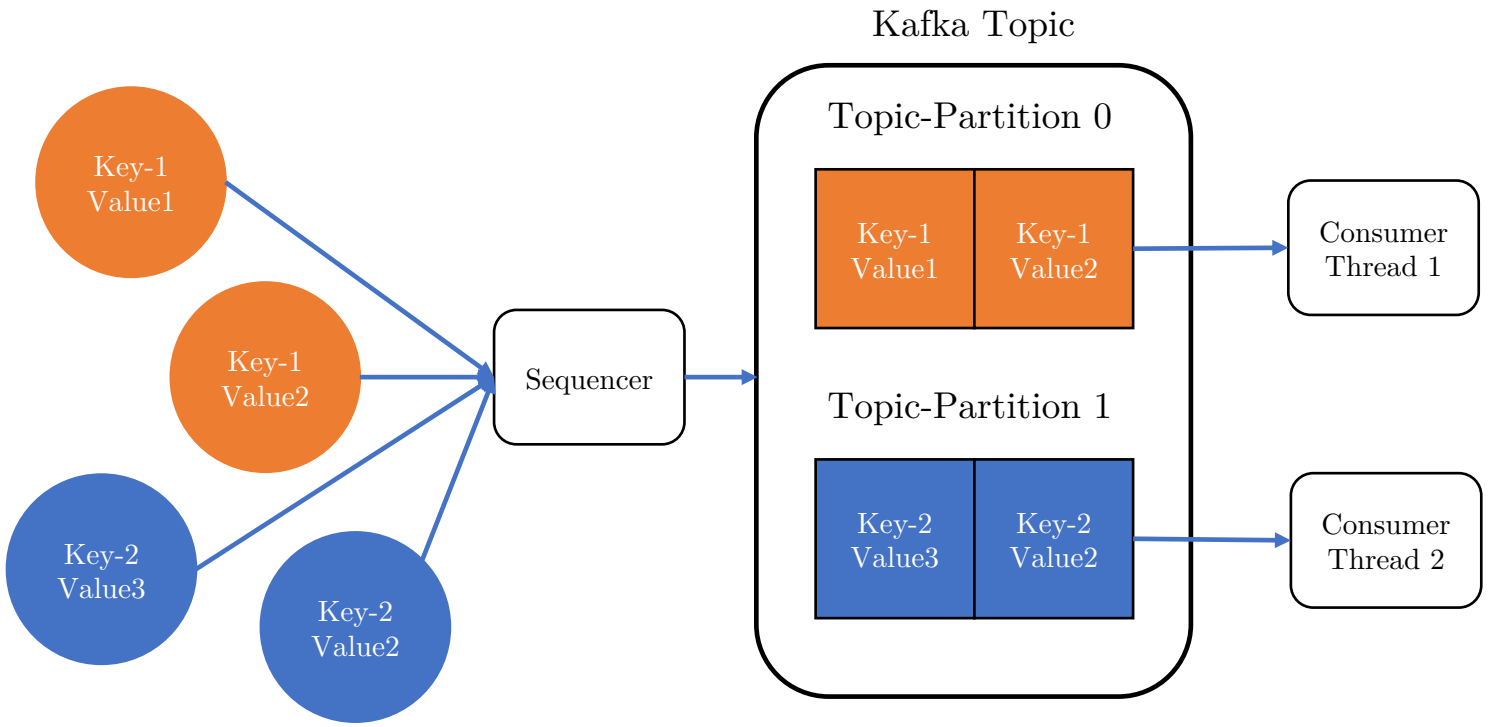
```
@Bean
public ConsumerFactory<Object, Object> consumerFactory() {
  final JsonDeserializer deserializer = new JsonDeserializer<>();
  deserializer.addTrustedPackages("*");
  final Map<String, Object> props = new HashMap<>(kafkaProperties.buildConsumerProperties());
  props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
  return new DefaultKafkaConsumerFactory(props, new StringDeserializer(), deserializer);
}


@Bean
public ConcurrentKafkaListenerContainerFactory<Object, Object> listenerContainerFactory(
    ConsumerFactory<Object, Object> consumerFactory, KafkaTemplate<Object, Object> template) {
  final ConcurrentKafkaListenerContainerFactory<Object, Object> factory =
    new ConcurrentKafkaListenerContainerFactory<>();
  factory.setConsumerFactory(consumerFactory);
  factory.setBatchListener(true);
  factory.setBatchErrorHandler(errorHandler(template));
  return factory;
}


@Bean
public BatchErrorHandler errorHandler(KafkaTemplate<Object, Object> template) {
  return new RecoveringBatchErrorHandler(
      new DeadLetterPublishingRecoverer(template), new FixedBackOff(100L, 2) // dead-letter after 3 tries
  );
}
```

Resilient Message Consumer

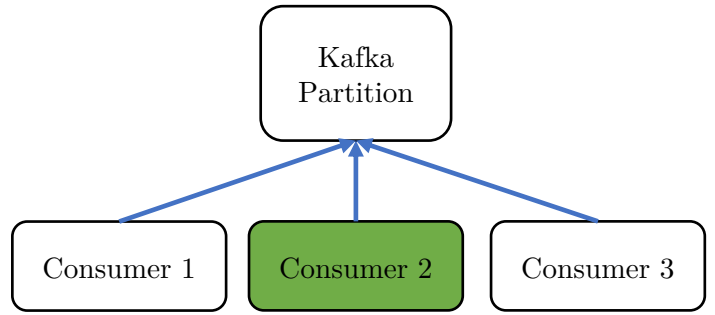
```
@KafkaListener(topics = "#{'${kafka.consumer.topic}'}", concurrency = "#{'${kafka.consumer.concurrency}’}",
   conatinerFactory = "listenerContainerFactory")
public void listen(List<MyEvent> request) {
  // throw BatchListenerFailedException with correct index of failed message from the argument list
  // when no exceptions thrown, Spring Kafka will confirm offset of last message from the list
}
```

Interesting Kafka Usage Examples

- Sharding of events received asynchronously by defined key
- Sequential processing of all events related to given key



- Leader election on top of Kafka
  - Create topic with one partition
  - All candidates should try to consume messages from given topic
  - Partition will be assigned to only one consumer thread (leader)
  - When leader thread terminates, Kafka will assign partition to another consumer



Interesting Kafka Usage Examples

- Log compaction retains the last known value for each record key per topic
- Compacted topic contains a full snapshot of “final” record values for every distinct key, not a complete change stream
- Message with a key and a null payload acts like a tombstone and allows to remove certain key permanently
- Log compaction runs as a background process and copies, modifies and swaps data files
- Example use-cases:
  - Publishing currency exchange rates
  - Publishing stock quotes

Before Compaction

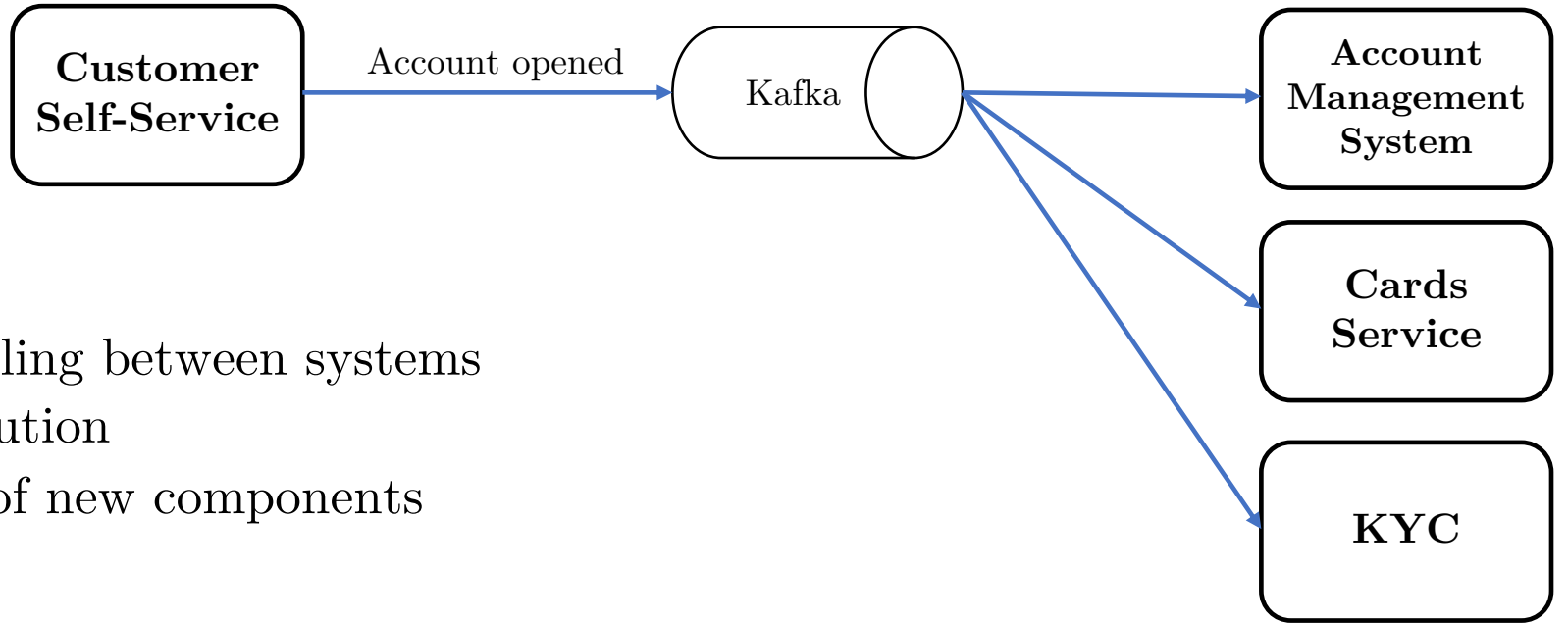
Offset	2	3	4	5	6	7	8	9
Key	K3	K5	K3	K3	K4	K4	K5	K6
Value	V1	V2	V3	V4	V5	Null	V7	V8

After Compaction

Offset	5	8	9
Key	K3	K5	K6
Value	V4	V7	V8

Definition

Event-driven architecture is a software architecture promoting the detection and action upon important business events in real time. More technically, an event can be defined as a significant change in state, that should be broadcasted to other components of the system.



- Advantages:
  - Removes tight coupling between systems
  - Easily auditable solution
  - Agile development of new components

Summary

- Asynchronous communication pattern and message-oriented middleware
  - Point-to-point delivery
  - Publish-subscribe delivery
- Short introduction to Apache Kafka
- Apache Kafka as a distributed system
  - Highly available Kafka cluster
  - Resilient message producer
  - Resilient message consumer
- Event-driven architecture