Streaming Design Patterns Using Alpakka Kafka Connector

Sean Glover, Lightbend
@seg1o

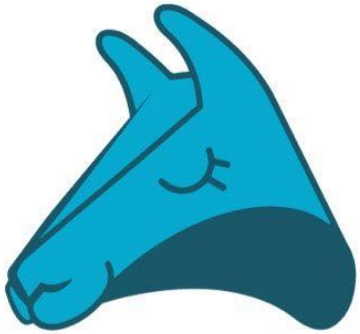
Lightbend

Who am I?

I'm Sean Glover

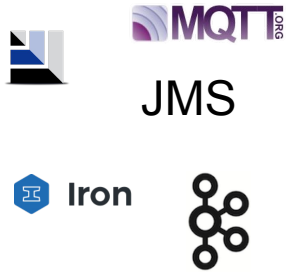
- Principal Engineer at Lightbend
- Member of the Lightbend Pipelines team
- Organizer of Scala Toronto (scalator)
- Author and contributor to various projects in the Kafka ecosystem including Kafka, Alpakka Kafka (reactive-kafka), Strimzi, Kafka Lag Exporter, DC/OS Commons SDK

/ seg1o



alpakka

> *The Alpakka project is an initiative to implement a library of integration modules to build stream-aware, reactive, pipelines for Java and Scala.*



Cloud Services

Data Stores

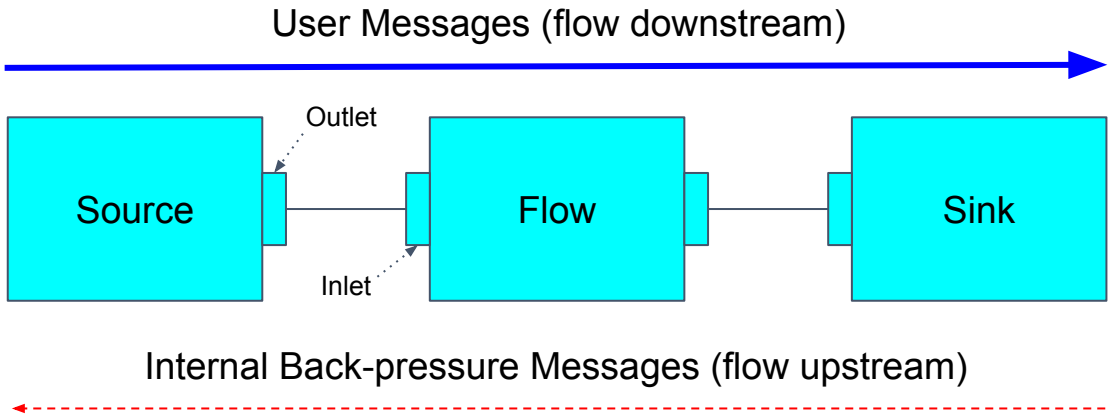
Messaging

This Alpakka Kafka connector lets you connect Apache Kafka to Akka Streams.

Top Alpakka Modules

Alpakka Module	Downloads in August 2018
Kafka	**61177**
Cassandra	15946
AWS S3	15075
MQTT	11403
File	10636
Simple Codecs	8285
CSV	7428
AWS SQS	5385
AMQP	4036

Akka Streams is a library toolkit to provide low latency complex event processing streaming semantics using the Reactive Streams specification implemented internally with an Akka actor system.


User Messages (flow downstream)
Outlet
Source
Flow
Sink
Inlet
Internal Back-pressure Messages (flow upstream)

Reactive Streams Specification

Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure.

http://www.reactive-streams.org/

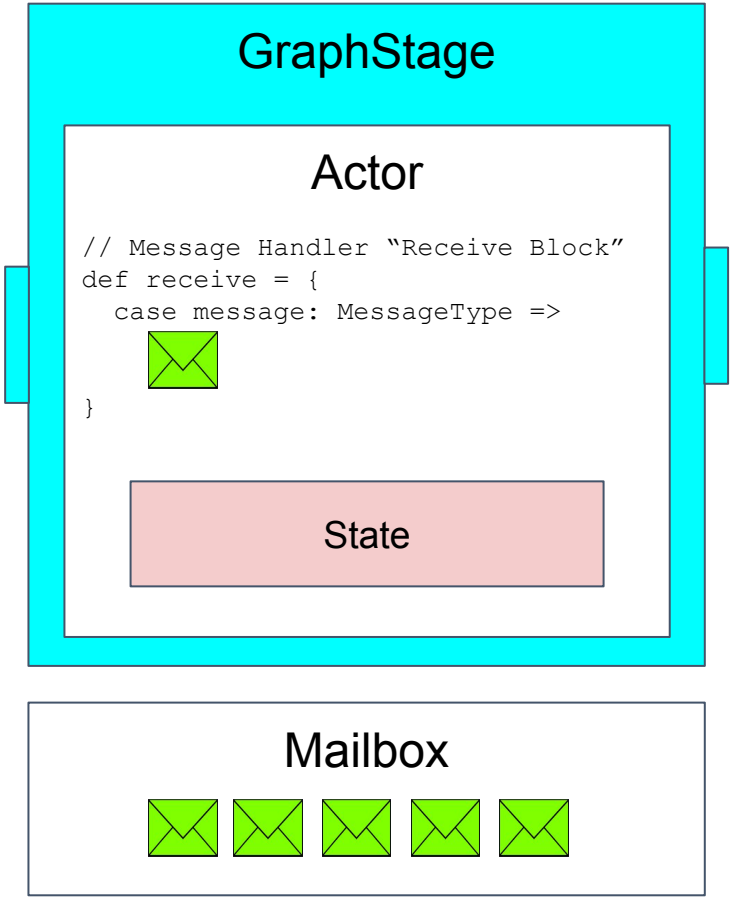
Reactive Streams Libraries

Spec now part of JDK 9

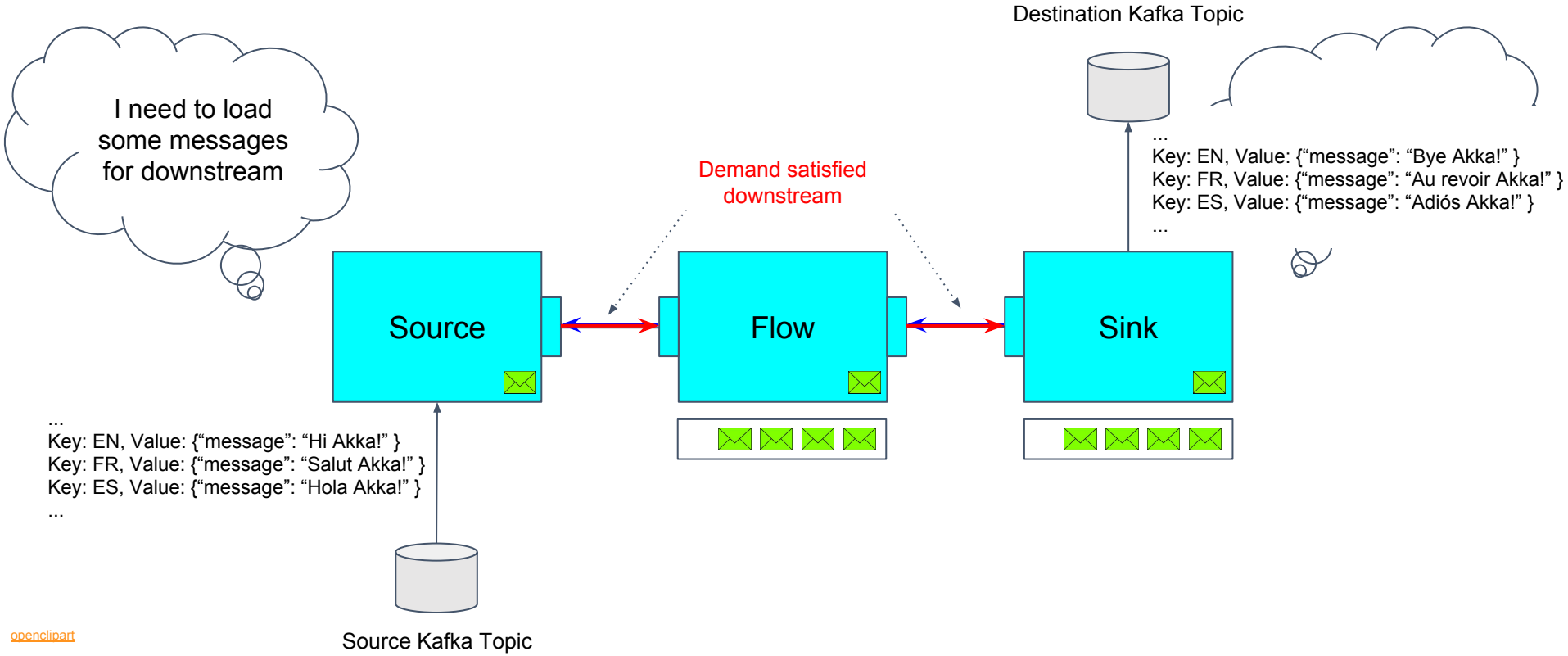
`java.util.concurrent.Flow`

Akka Actor Concepts

1. Constrained Actor Mailbox
2. One message at a time
   “Single Threaded Illusion”
3. May contain state

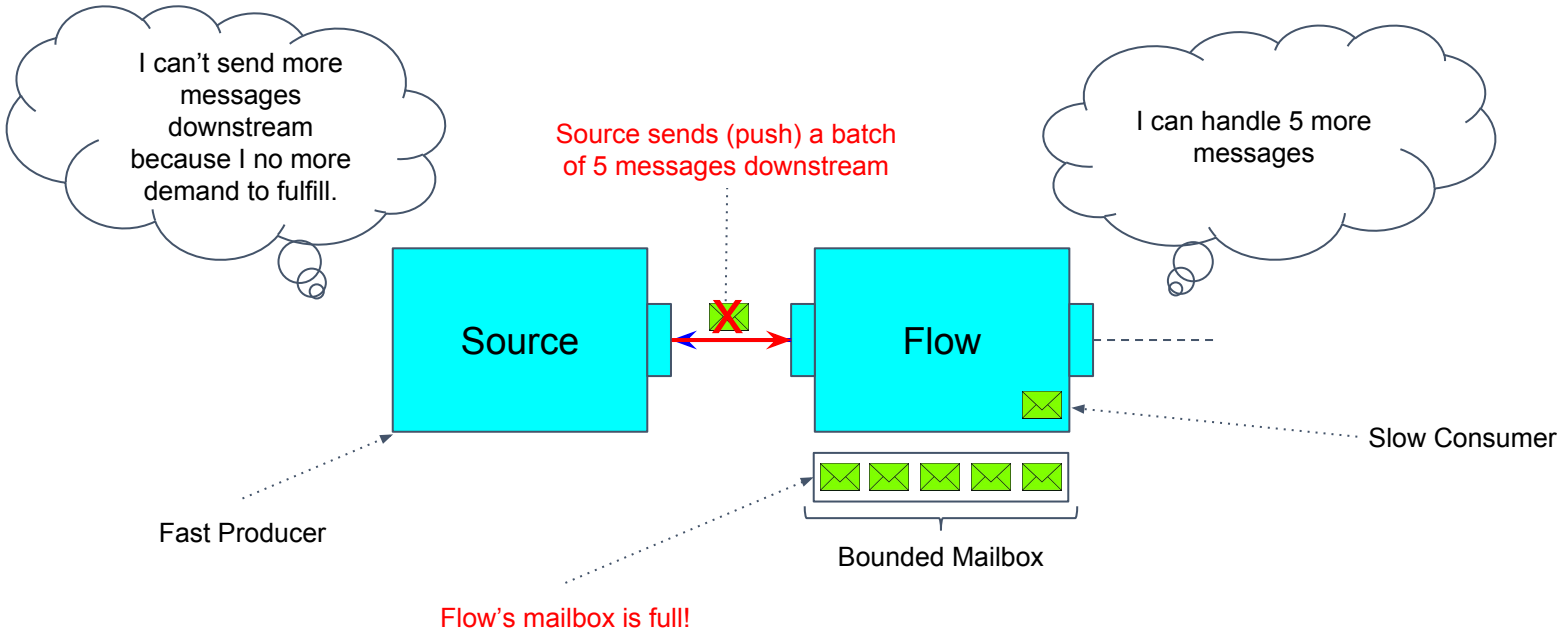


Back Pressure Demo



openclipart

Dynamic Push Pull



openclipart

Why Back Pressure?

- Prevent cascading failure
- Alternative to using a big buffer (i.e. Kafka)
- Back Pressure flow control can use several strategies
  - Slow down until there's demand (classic back pressure, "throttling")
  - Discard elements
  - Buffer in memory to some max, then discard elements
  - Shutdown

Why Back Pressure? A case study.

https://medium.com/@programmerohit/back-pressure-implementation-aws-sqs-polling-from-a-sharded-akka-cluster-running-on-kubernetes-56ee8c67efb

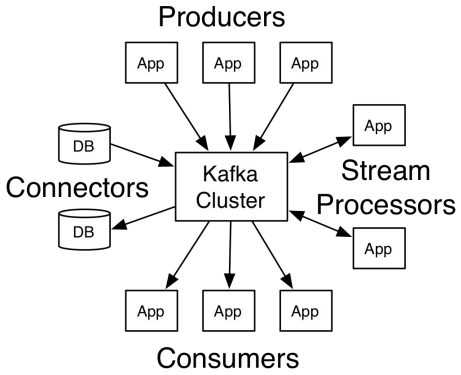
Akka Streams Factorial Example

```
import ...

object Main extends App {
 implicit val system = ActorSystem("QuickStart")
 implicit val materializer = ActorMaterializer()

 val source: Source[Int, NotUsed] = Source(1 to 100)
 val factorials = source.scan(BigInt(1))((acc, next) ⇒ acc * next)
 val result: Future[IOResult] =
   factorials
     .map(num => ByteString(s"$num\n"))
     .runWith(FileIO.toPath(Paths.get("factorials.txt")))
}
```

https://doc.akka.io/docs/akka/2.5/stream/stream-quickstart.html

Apache Kafka

*Apache Kafka is a distributed streaming system. It's best suited to support **fast**, **high volume**, and **fault tolerant**, data streaming platforms.*



Kafka Documentation

When to use Alpakka Kafka?

!=

Akka Streams and Kafka Streams solve different problems

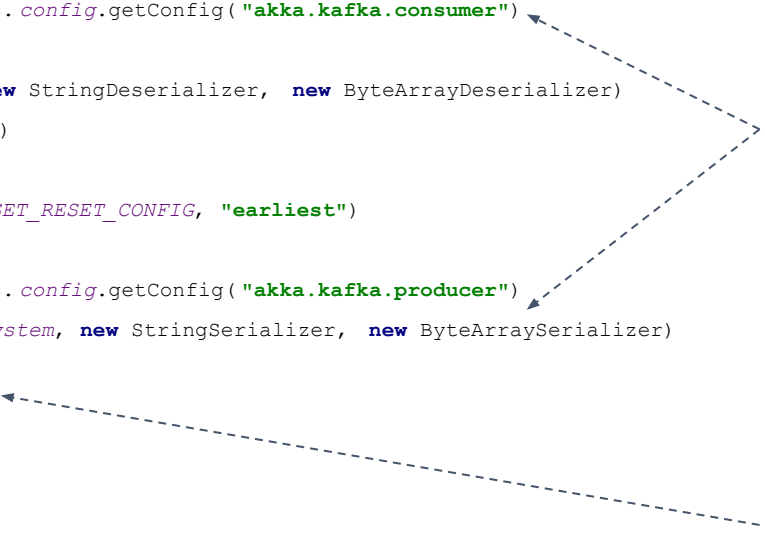
When to use Alpakka Kafka?

1. To build back pressure aware integrations
2. Complex Event Processing
3. A need to model the most complex of graphs

Anatomy of an Alpakka Kafka app

Alpakka Kafka Setup

```
val consumerClientConfig = system.settings.config.getConfig("akka.kafka.consumer")
val consumerSettings =
 ConsumerSettings(consumerClientConfig, new StringDeserializer,  new ByteArrayDeserializer)
   .withBootstrapServers( "localhost:9092")
   .withGroupId( "group1")
   .withProperty(ConsumerConfig. AUTO_OFFSET_RESET_CONFIG, "earliest")

val producerClientConfig = system.settings.config.getConfig("akka.kafka.producer")
val producerSettings = ProducerSettings(system, new StringSerializer,  new ByteArraySerializer)
 .withBootstrapServers( "localhost:9092")
```



Anatomy of an Alpakka Kafka App

A small Consume -> Transform -> Produce Akka Streams app using Alpakka Kafka

```
val control = Consumer
   .committableSource(consumerSettings, Subscriptions. topics(topic1))
   .map { msg =>
     ProducerMessage. single(
           new ProducerRecord(topic1, msg.record.key, msg.record.value),
           passThrough = msg.committableOffset)
   }
   .via(Producer. flexiFlow(producerSettings))
   .map(_.passThrough)
   .toMat(Committer. sink(committerSettings))(Keep. both)
   .mapMaterializedValue(DrainingControl.  apply)
   .run()
// Add shutdown hook to respond to SIGTERM and gracefully shutdown stream
sys.ShutdownHookThread {
 Await.result(control.shutdown(),  10.seconds)
}
```

Anatomy of an Alpakka Kafka App

```
val control = Consumer
   .committableSource(consumerSettings, Subscriptions. topics(topic1))
   .map { msg =>
     ProducerMessage. single(
           new ProducerRecord(topic1, msg.record.key, msg.record.value),
           passThrough = msg.committableOffset)
   }
   .via(Producer. flexiFlow(producerSettings))
   .map(_.passThrough)
   .toMat(Committer. sink(committerSettings))(Keep. both)
   .mapMaterializedValue(DrainingControl.  apply)
   .run()
// Add shutdown hook to respond to SIGTERM and gracefully shutdown stream
sys.ShutdownHookThread {
 Await.result(control.shutdown(), 10.seconds)
}
```

The Committable Source propagates Kafka offset information downstream with consumed messages

Anatomy of an Alpakka Kafka App

```
val control = Consumer
   .committableSource(consumerSettings, Subscriptions. topics(topic1))
   .map { msg =>
     ProducerMessage. single(
           new ProducerRecord(topic1, msg.record.key, msg.record.value),
           passThrough = msg.committableOffset)
   }
   .via(Producer. flexiFlow(producerSettings))
   .map(_.passThrough)
   .toMat(Committer. sink(committerSettings))(Keep. both)
   .mapMaterializedValue(DrainingControl. apply)
   .run()
// Add shutdown hook to respond to SIGTERM and gracefully shutdown stream
sys.ShutdownHookThread {
 Await.result(control.shutdown(), 10.seconds)
}
```

ProducerMessage used to map consumed offset to transformed results.

One to One (**1:1**)

```
ProducerMessage. single
```

One to Many (**1:M**)

```
ProducerMessage. multi(
 immutable.Seq(
   new ProducerRecord(topic1, msg.record.key, msg.record.value),
   new ProducerRecord(topic2, msg.record.key, msg.record.value)),
 passthrough = msg.committableOffset
)
```

One to None (**1:0**)

```
ProducerMessage. passThrough(msg.committableOffset)
```

Anatomy of an Alpakka Kafka App

```
val control = Consumer
   .committableSource(consumerSettings, Subscriptions. topics(topic1))
   .map { msg =>
     ProducerMessage. single(
           new ProducerRecord(topic1, msg.record.key, msg.record.value),
           passThrough = msg.committableOffset)
   }
   .via(Producer. flexiFlow(producerSettings))
   .map(_.passThrough)
   .toMat(Committer. sink(committerSettings))(Keep. both)
   .mapMaterializedValue(DrainingControl.  apply)
   .run()
// Add shutdown hook to respond to SIGTERM and gracefully shutdown stream
sys.ShutdownHookThread {
 Await.result(control.shutdown(), 10.seconds)
}
```

Produce messages to destination topic

`flexiFlow` accepts new `ProducerMessage` type and will replace deprecated `flow` in the future.

Anatomy of an Alpakka Kafka App

```
val control = Consumer
   .committableSource(consumerSettings, Subscriptions. topics(topic1))
   .map { msg =>
     ProducerMessage. single(
           new ProducerRecord(topic1, msg.record.key, msg.record.value),
           passThrough = msg.committableOffset)
   }
   .via(Producer. flexiFlow(producerSettings))
   .map(_.passThrough)
   .toMat(Committer.sink(committerSettings))(Keep.both)
   .mapMaterializedValue(DrainingControl. apply)
   .run()
// Add shutdown hook to respond to SIGTERM and gracefully shutdown stream
sys.ShutdownHookThread {
 Await.result(control.shutdown(), 10.seconds)
}
```

Batches consumed offset commits

Passthrough allows us to track what messages have been successfully processed for **At Least Once** message delivery guarantees.

Anatomy of an Alpakka Kafka App

```
val control = Consumer
   .committableSource(consumerSettings, Subscriptions. topics(topic1))
   .map { msg =>
     ProducerMessage. single(
           new ProducerRecord(topic1, msg.record.key, msg.record.value),
           passThrough = msg.committableOffset)
   }
   .via(Producer. flexiFlow(producerSettings))
   .map(_.passThrough)
   .toMat(Committer. sink(committerSettings))(Keep. both)
   .mapMaterializedValue(DrainingControl. apply)
   .run()
// Add shutdown hook to respond to SIGTERM and gracefully shutdown stream
sys.ShutdownHookThread {
 Await.result(control.shutdown(), 10.seconds)
}
```

Gacefully shutdown stream

1. Stop consuming (polling) new messages from Source
2. Wait for all messages to be successfully committed (when applicable)
3. Wait for all produced messages to ACK

Anatomy of an Alpakka Kafka App

```
val control = Consumer
    .committableSource(consumerSettings, Subscriptions. topics(topic1))
    .map { msg =>
      ProducerMessage. single(
            new ProducerRecord(topic1, msg.record.key, msg.record.value),
            passThrough = msg.committableOffset)
    }
    .via(Producer. flexiFlow(producerSettings))
    .map(_.passThrough)
    .toMat(Committer. sink(committerSettings))(Keep. both)
    .mapMaterializedValue(DrainingControl. apply)
    .run()
// Add shutdown hook to respond to SIGTERM and gracefully shutdown stream
sys.ShutdownHookThread {
 Await.result(control.shutdown(), 10.seconds)
}
```

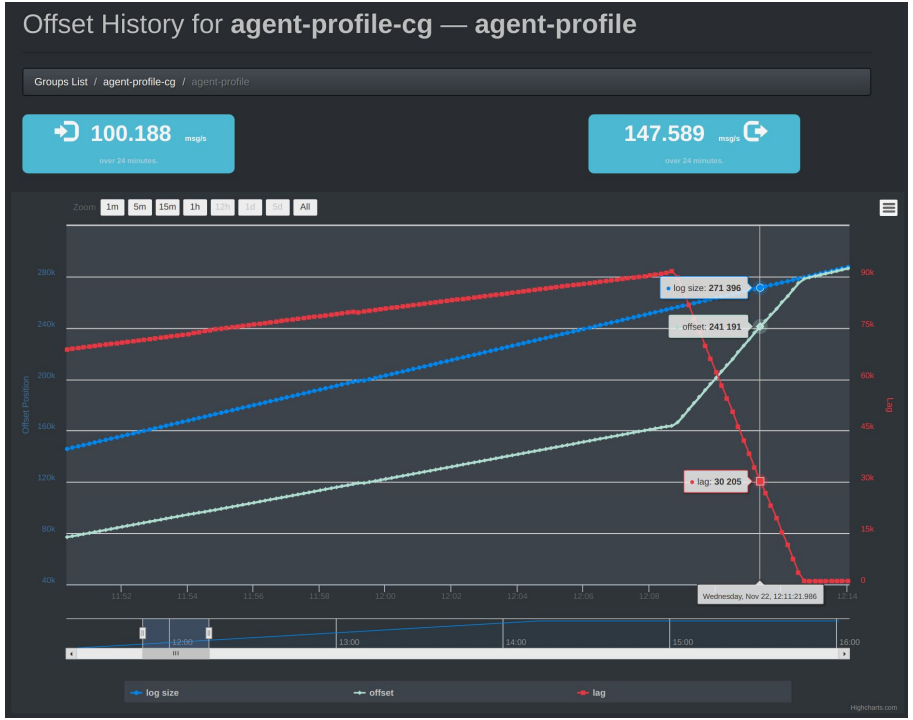
Graceful shutdown when SIGTERM sent to app (i.e. by docker daemon)

Force shutdown after grace interval

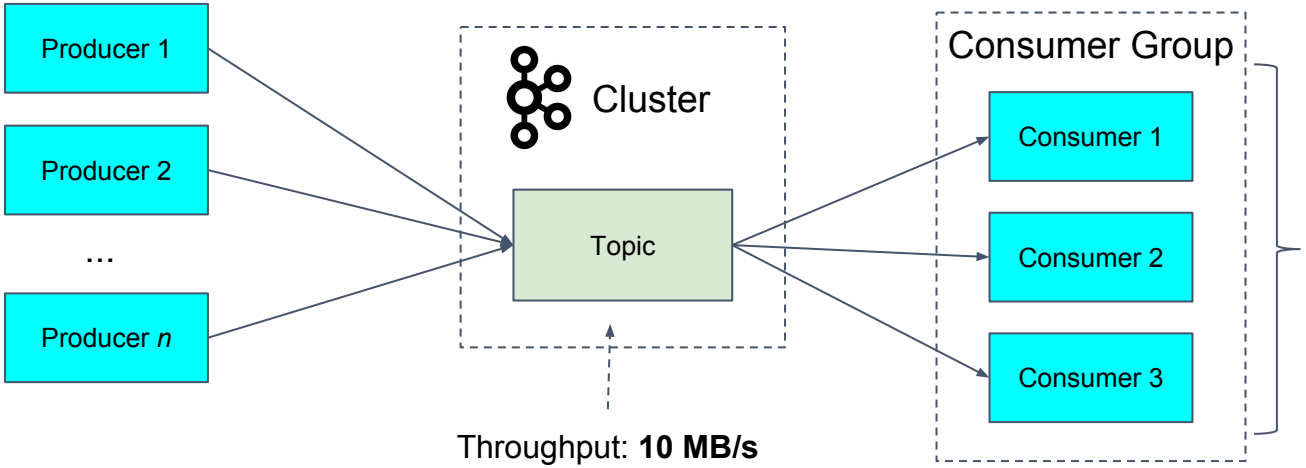
Consumer Group Rebalancing

Why use Consumer Groups?

1. Easy, robust, and performant scaling of consumers to reduce **consumer lag**



Latency and Offset Lag



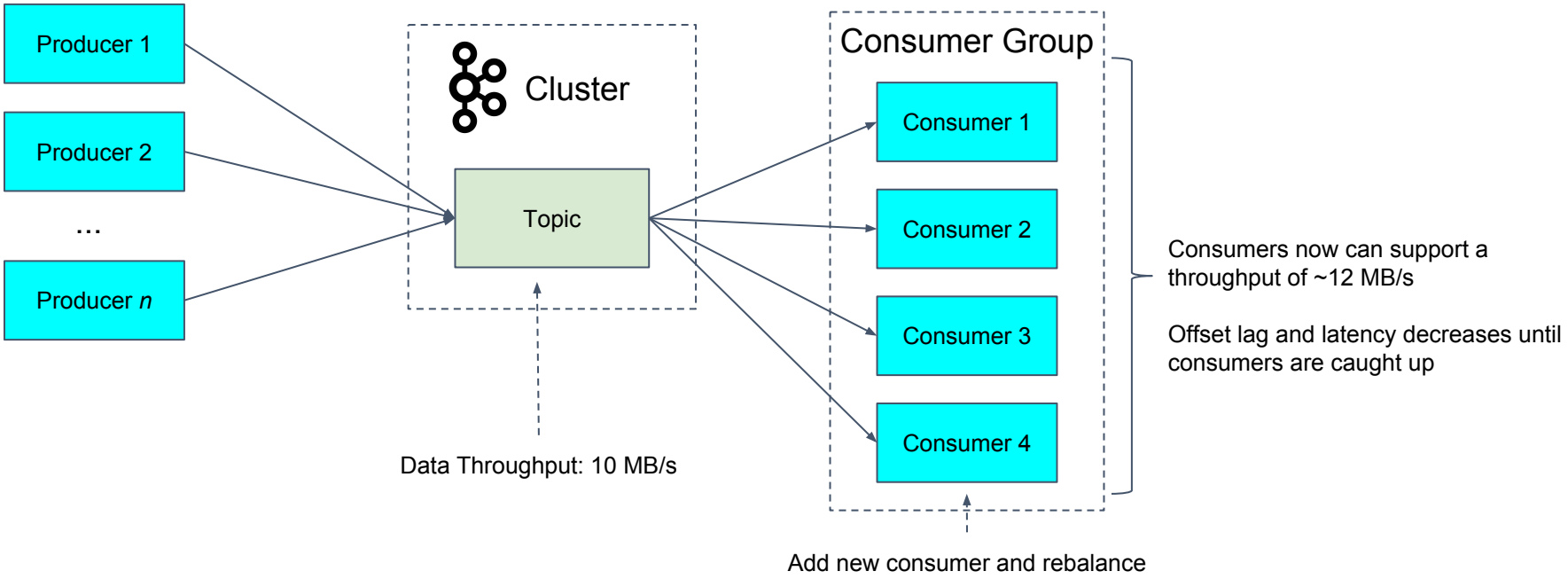
Consumer Throughput ~3 MB/s each

~9 MB/s

Total offset lag and latency is growing.

openclipart

Latency and Offset Lag



Committable Sink

```
val committerSettings = CommitterSettings(system)

val control: DrainingControl[Done] =
  Consumer
    .committableSource(consumerSettings, Subscriptions.topics(topic))
    .mapAsync(1) { msg =>
      business(msg.record.key, msg.record.value)
        .map(_ => msg.committableOffset)
    }
    .toMat(Committer.sink(committerSettings))(Keep.both)
    .mapMaterializedValue(DrainingControl.apply)
    .run()
```

Anatomy of a Consumer Group

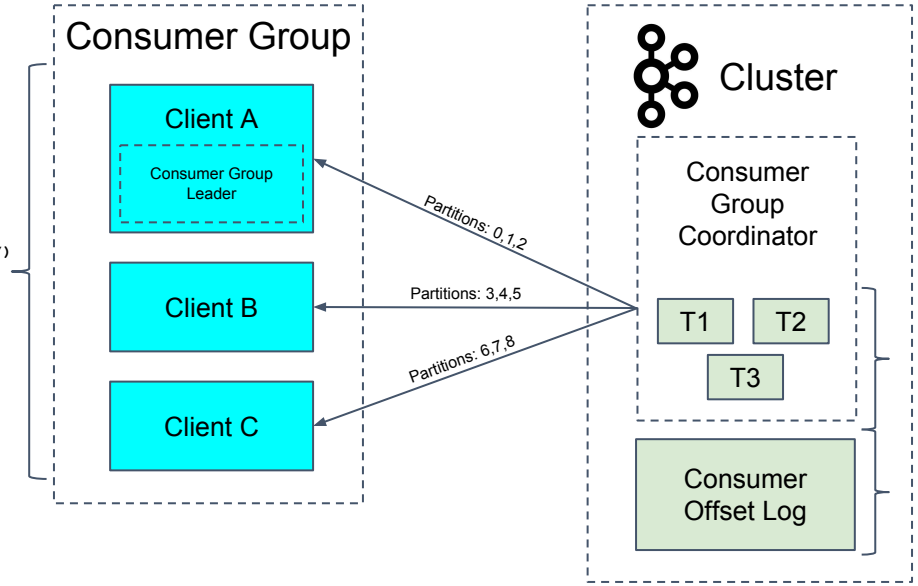
Important Consumer Group Client Config

Topic Subscription:

```
Subscription.topics("Topic1", "Topic2", "Topic3")
```

Kafka Consumer Properties:

```
group.id:                        ["my-group"]
session.timeout.ms:              [30000 ms]
partition.assignment.strategy:   [RangeAssignor]
heartbeat.interval.ms:           [3000 ms]
```

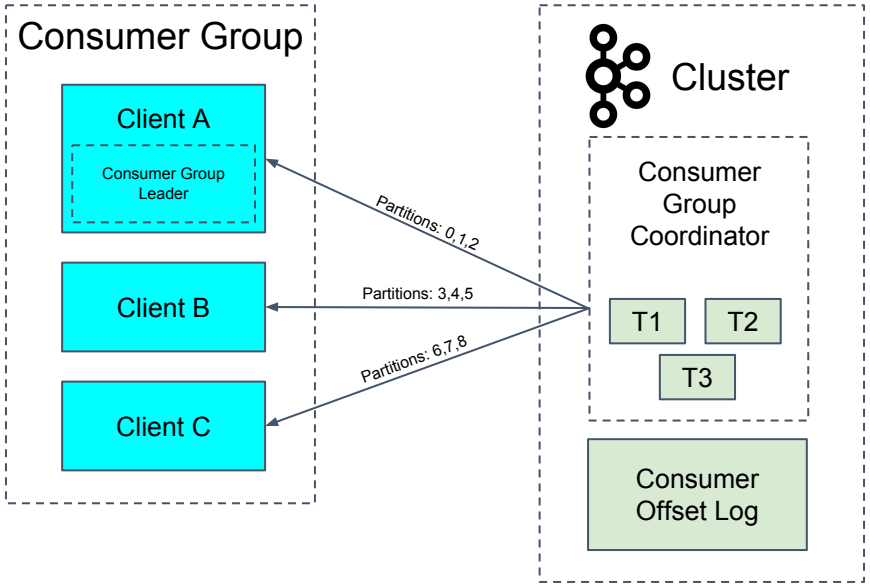


Consumer Group Topic Subscription

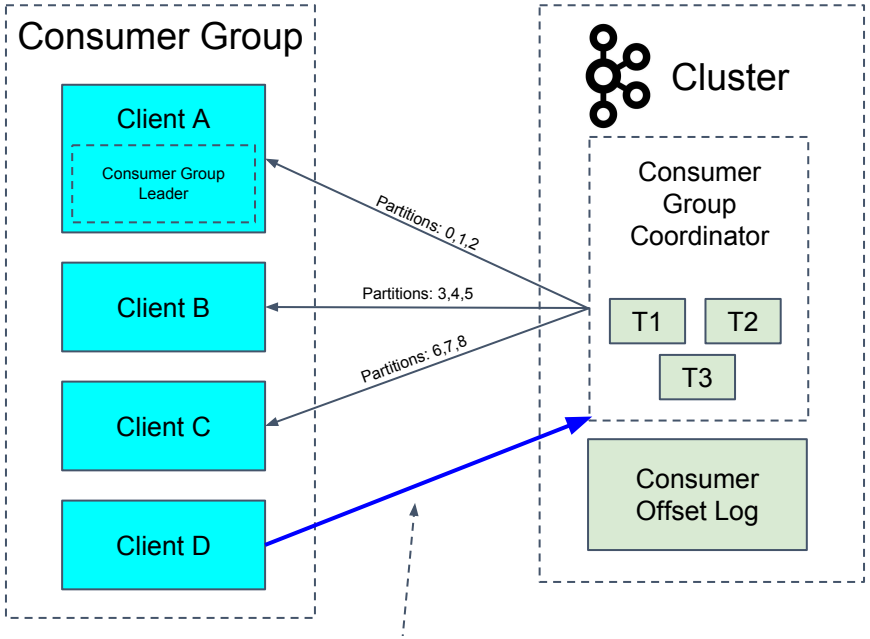
Consumer Group Offsets topic
Ex)

```
P0: 100489
P1: 128048
P2: 184082
P3: 596837
P4: 110847
P5: 99472
P6: 148270
P7: 3582785
P8: 182483
```

Consumer Group Rebalance (1/7)



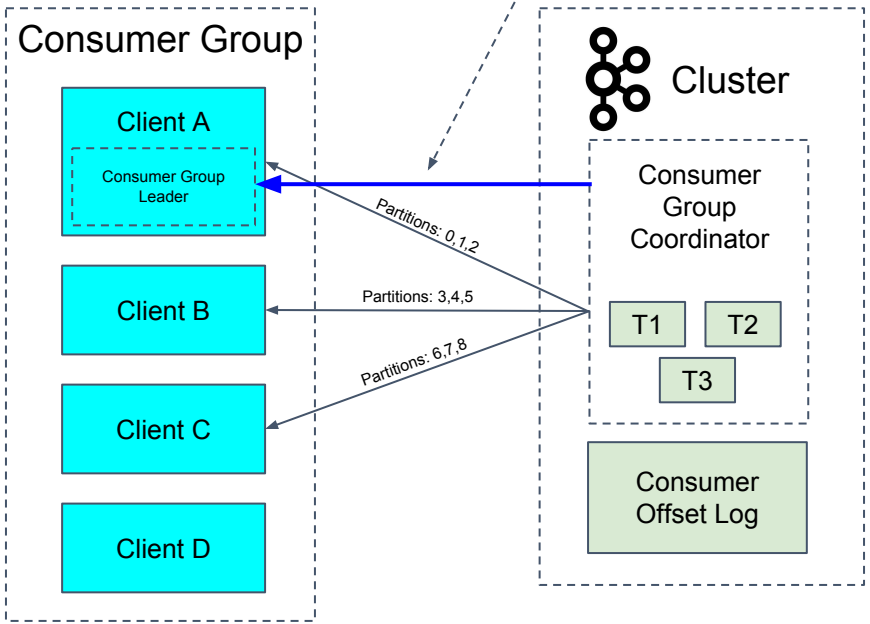
Consumer Group Rebalance (2/7)



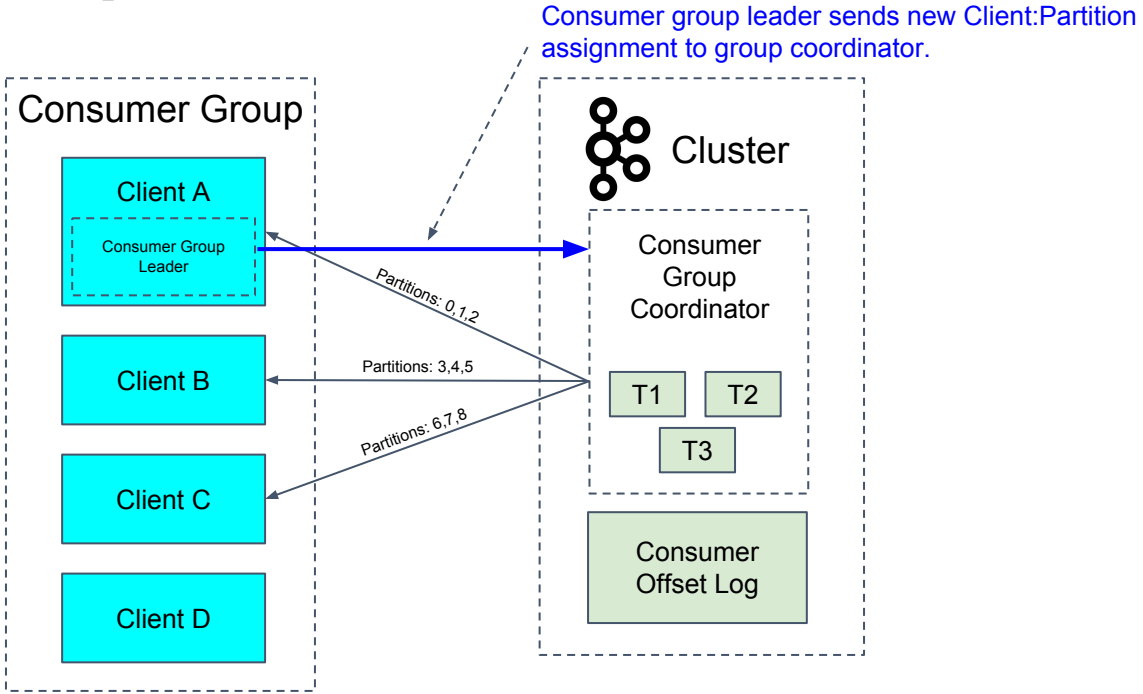
New Client D with same group.id sends a request to join the group to Coordinator

Consumer Group Rebalance (3/7)

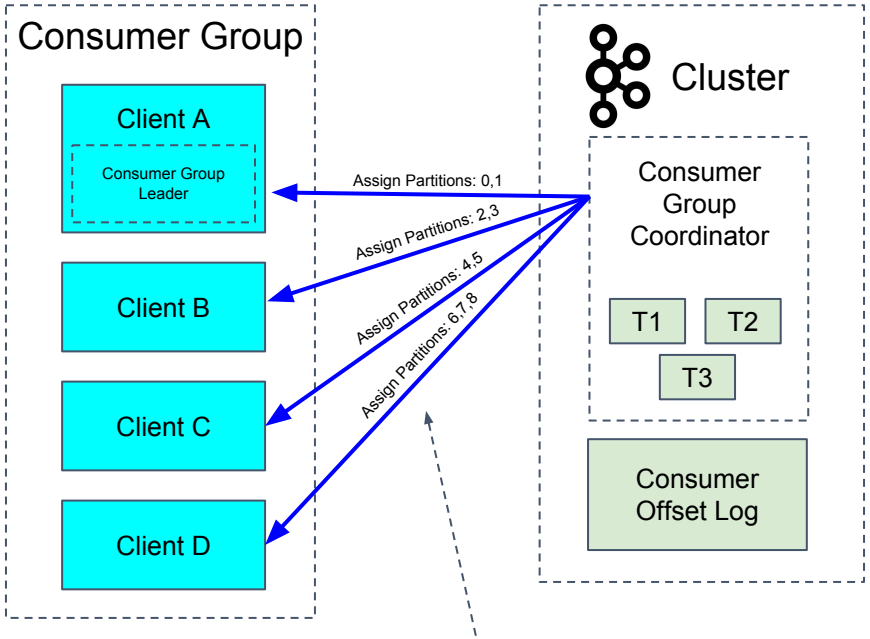
Consumer group coordinator requests group leader to calculate new Client:partition assignments.



Consumer Group Rebalance (4/7)

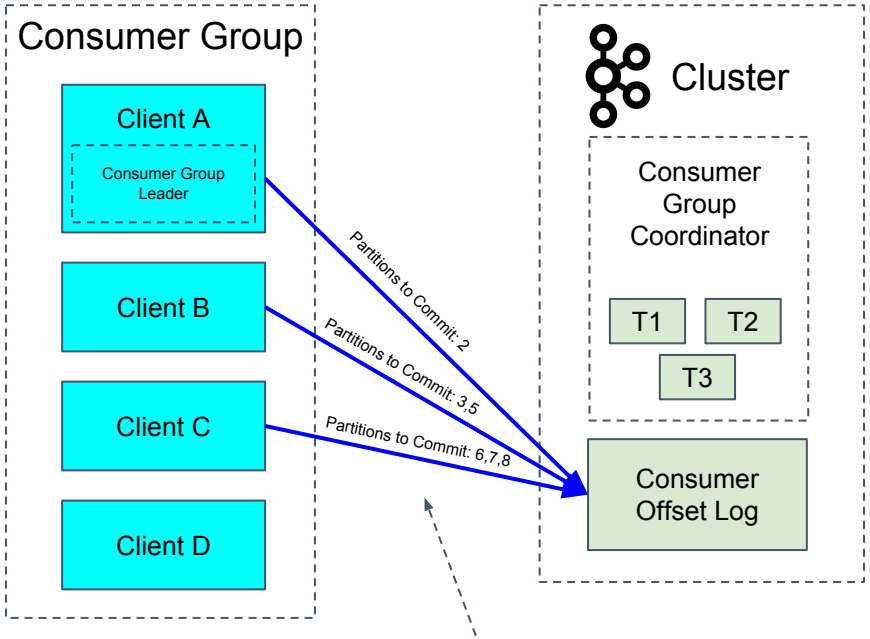


Consumer Group Rebalance (5/7)



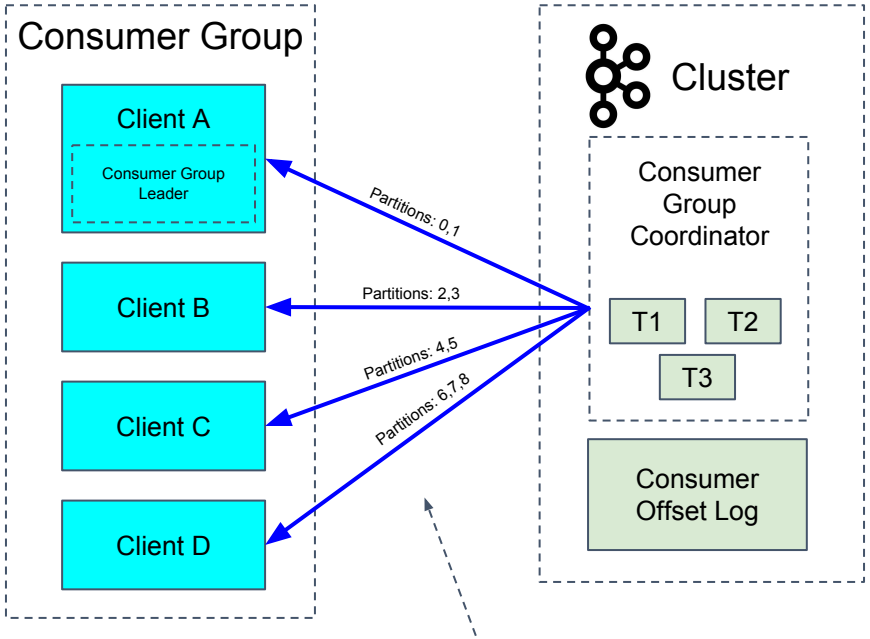
Consumer group coordinator informs all clients of their new Client:Partition assignments.

Consumer Group Rebalance (6/7)



Clients that had partitions revoked are given the chance to commit their latest processed offsets.

Consumer Group Rebalance (7/7)



Rebalance complete. Clients begin consuming partitions from their last committed offsets.

Consumer Group Rebalancing (asynchronous)

```
class RebalanceListener extends Actor with ActorLogging {
  def receive: Receive = {
    case TopicPartitionsAssigned(sub, assigned) =>
    case TopicPartitionsRevoked(sub, revoked) =>
      processRevokedPartitions(revoked)
  }
}

val subscription = Subscriptions.topics("topic1", "topic2")
  .withRebalanceListener(system.actorOf(Props[RebalanceListener]))

val control = Consumer.committableSource(consumerSettings, subscription)
   ...
```

Declare an Akka Actor to handle assigned and revoked partition messages asynchronously.

Useful to perform **asynchronous** actions during rebalance, but not for blocking operations you want to happen during rebalance.

Consumer Group Rebalancing (asynchronous)

```
class RebalanceListener extends Actor with ActorLogging {
  def receive: Receive = {
    case TopicPartitionsAssigned(sub, assigned) =>
    case TopicPartitionsRevoked(sub, revoked) =>
      processRevokedPartitions(revoked)
  }
}
```

Add Actor Reference to Topic subscription to use.

```
val subscription = Subscriptions.topics("topic1", "topic2")
  .withRebalanceListener(system.actorOf(Props[RebalanceListener]))

val control = Consumer.committableSource(consumerSettings, subscription)
   ...
```

Consumer Group Rebalancing (synchronous)

Synchronous partition assignment handler for next release by Enno Runne
https://github.com/akka/alpakka-kafka/pull/761

- Synchronous operations difficult to model in async library
- Will allow users block Consumer Actor thread (Consumer.poll thread)
- Provides limited consumer operations
  - Seek to offset
  - Synchronous commit

Transactional “Exactly-Once”

Kafka Transactions

Transactions enable atomic writes to multiple Kafka topics and partitions. All of the messages included in the transaction will be successfully written or none of them will be.

Message Delivery Semantics

- At most once
- At least once
- “Exactly once” 

openclipart

Exactly Once Delivery vs Exactly Once Processing

Exactly-once message delivery is impossible between two parties where failures of communication are possible.

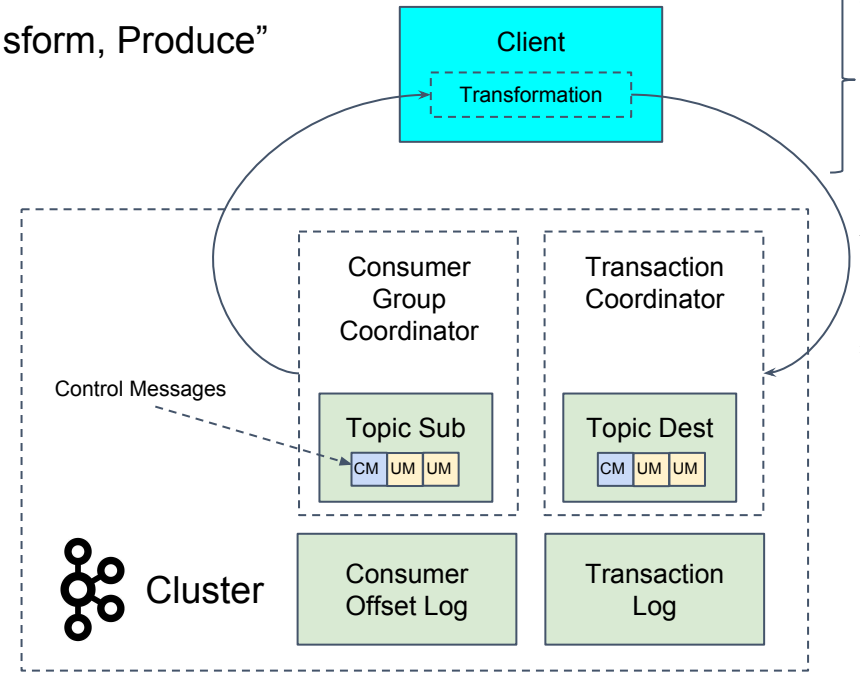
Two Generals/Byzantine Generals problem

Why use Transactions?

1. Zero tolerance for duplicate messages
2. Less boilerplate (deduping, client offset management)

Anatomy of Kafka Transactions

"Consume, Transform, Produce"



Important Client Config

Topic Subscription:

```
Subscription.topics("Topic1", "Topic2", "Topic3")
```

Destination topic partitions get included in the transaction based on messages that are produced.

Kafka Consumer Properties:

```
group.id:                                   "my-group"
isolation.level:                            "read_committed"
plus other relevant consumer group configuration
```

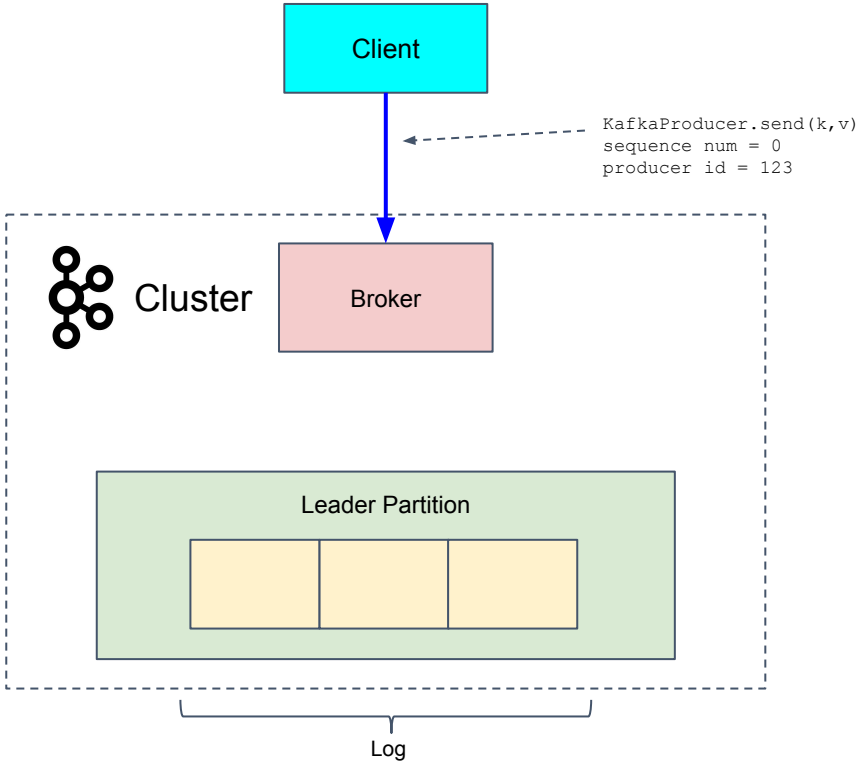
Kafka Producer Properties:

```
transactional.id:                           "my-transaction"
enable.idempotence:                         "true" (implicit)
max.in.flight.requests.per.connection:      "1" (implicit)
```

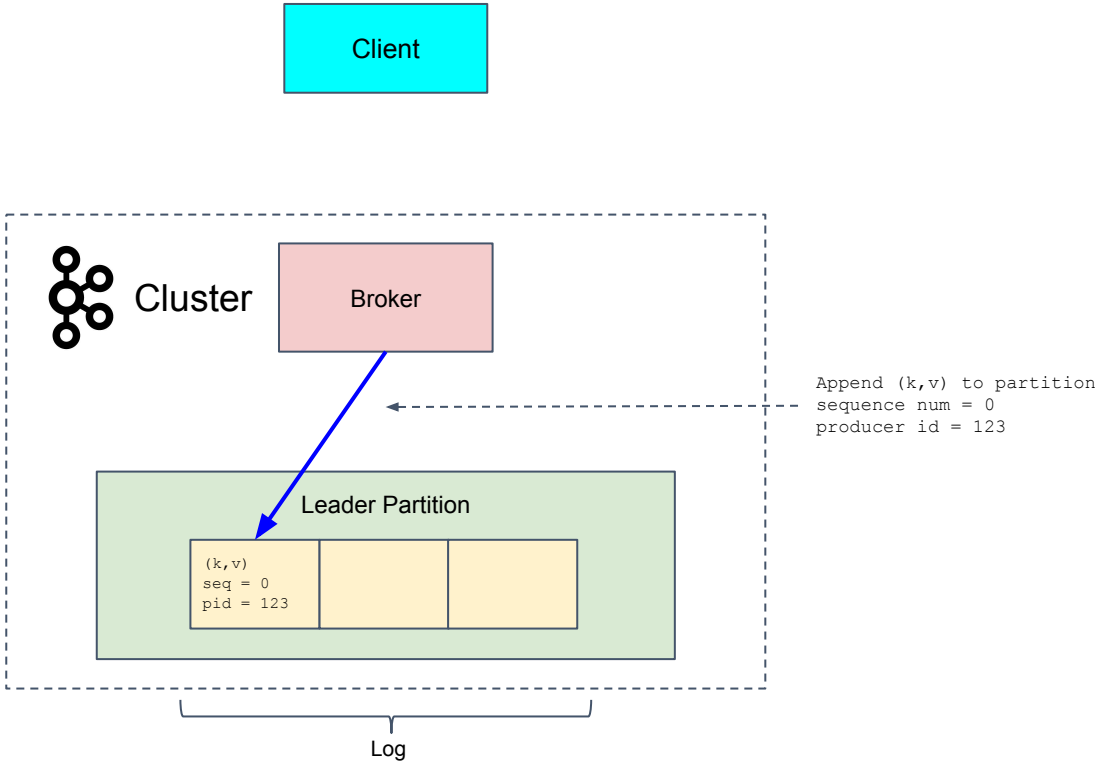
Kafka Features That Enable Transactions

1. Idempotent producer
2. Multiple partition atomic writes
3. Consumer read isolation level

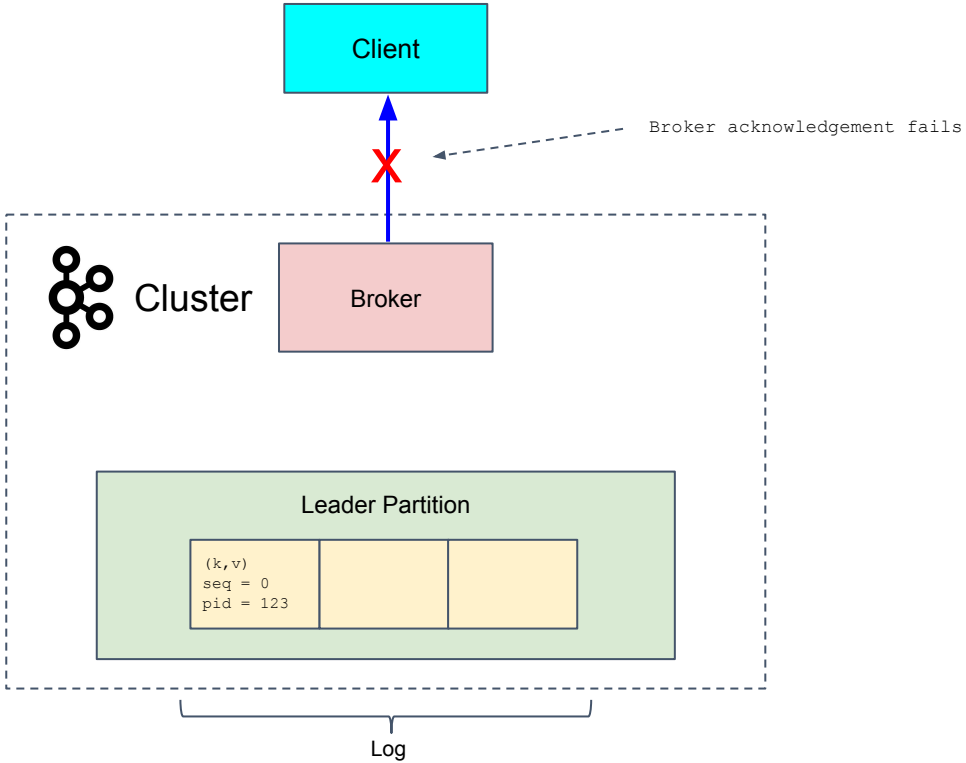
Idempotent Producer (1/5)



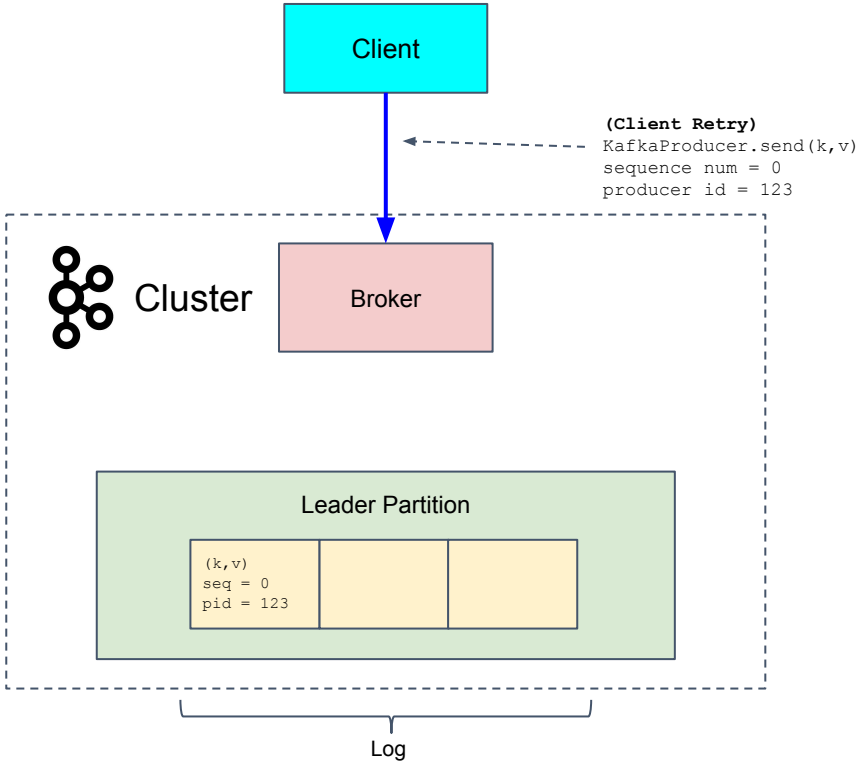
Idempotent Producer (2/5)



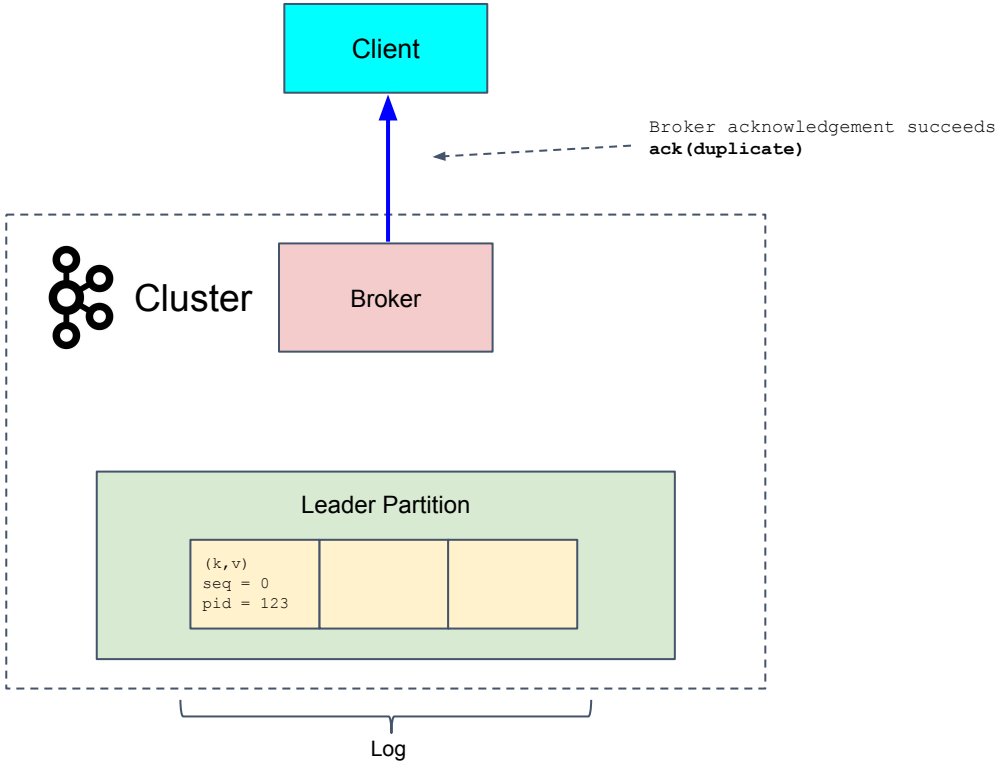
Idempotent Producer (3/5)



Idempotent Producer (4/5)

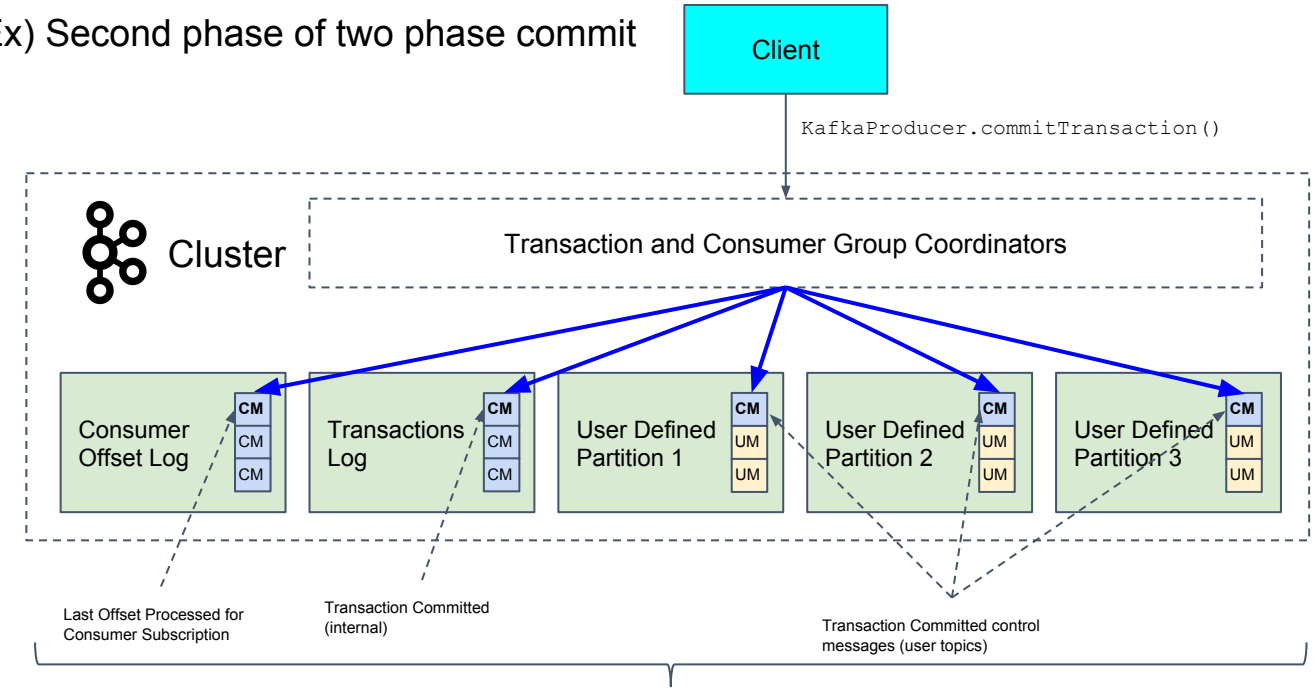


Idempotent Producer (5/5)



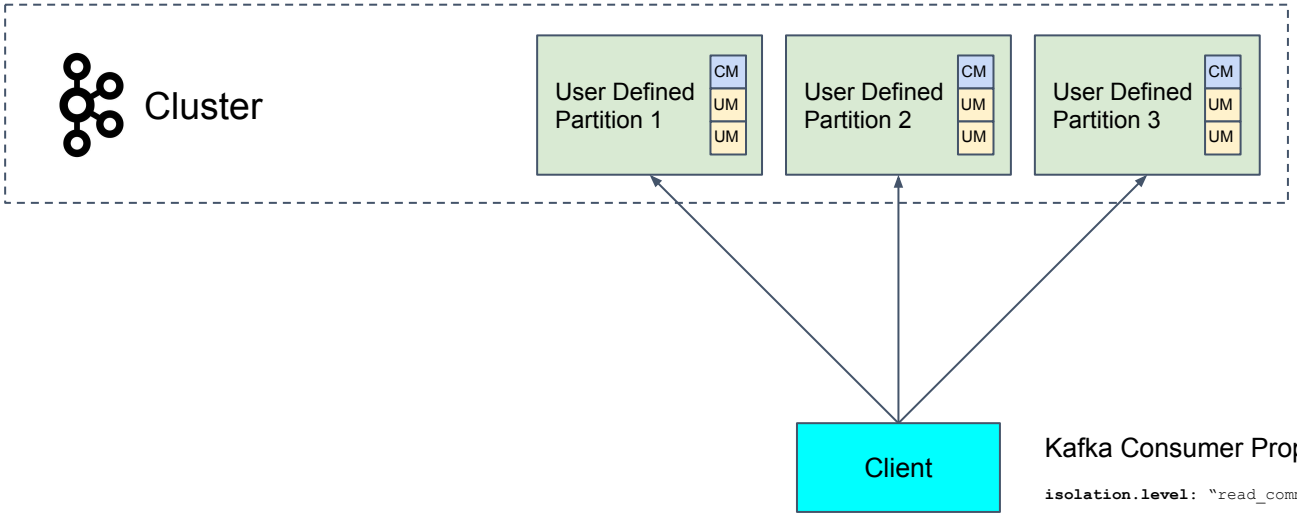
Multiple Partition Atomic Writes

Ex) Second phase of two phase commit



Multiple Partitions Committed Atomically, “All or nothing”

Consumer Read Isolation Level

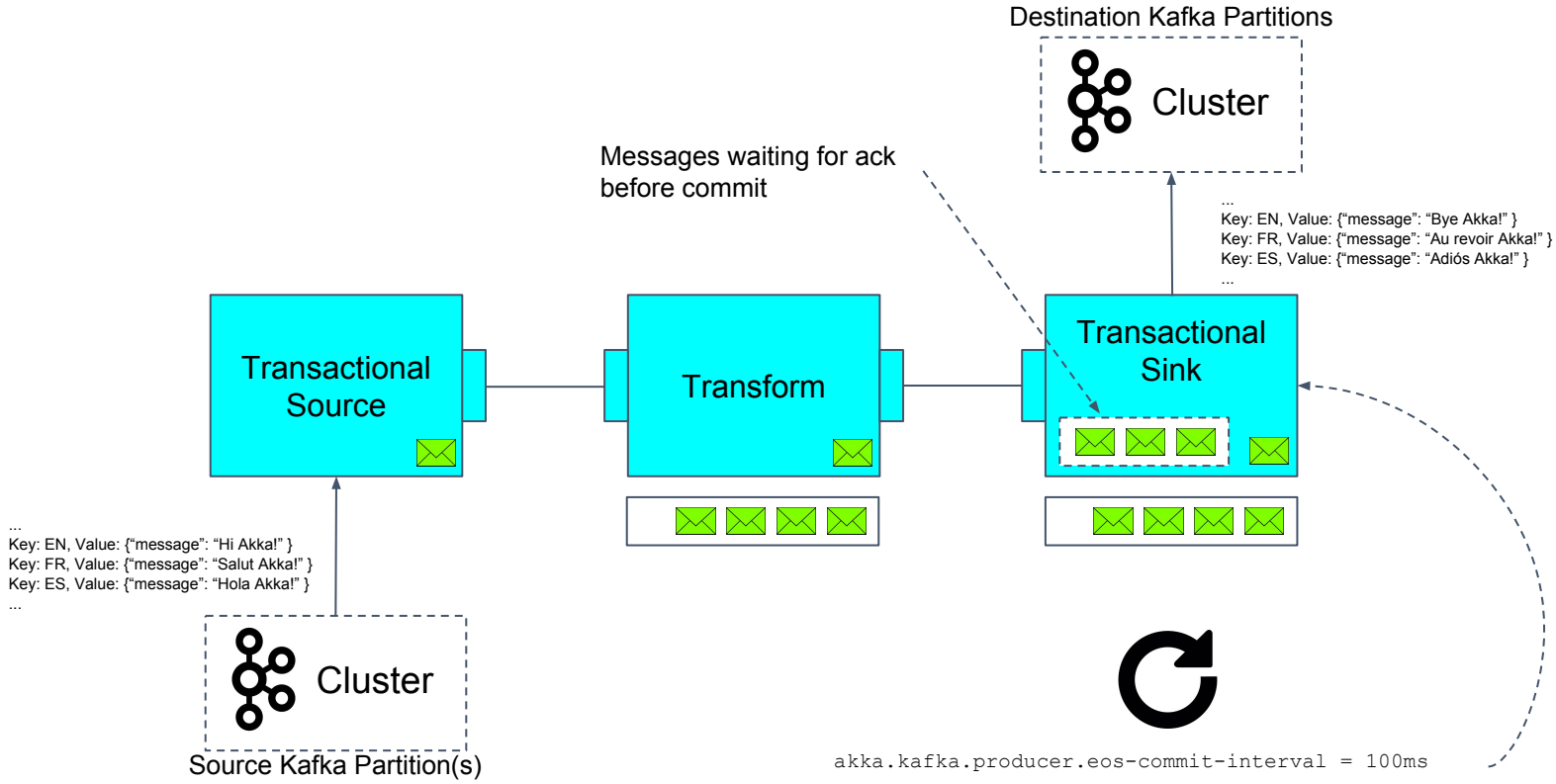


Transactional Pipeline Latency

Transaction Batches every 100ms

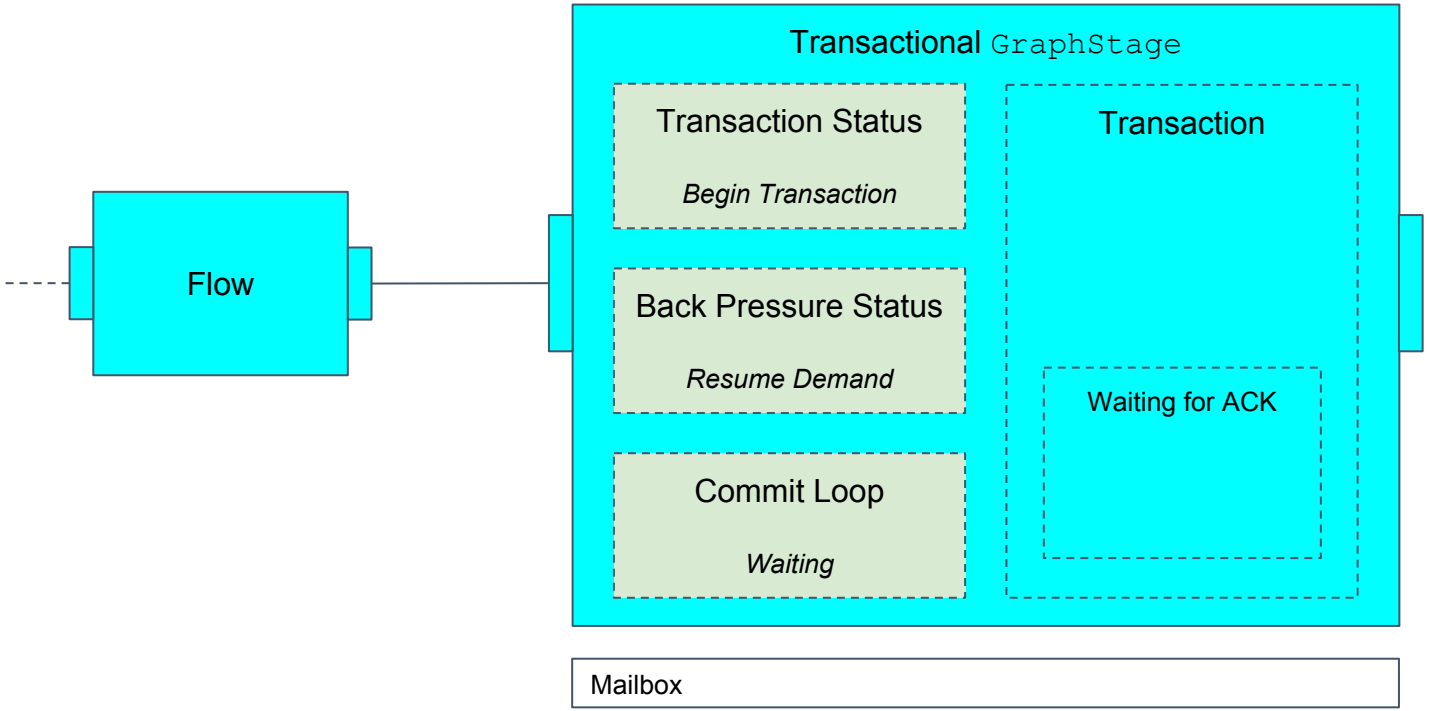
End-to-end Latency
~300ms

Alpakka Kafka Transactions

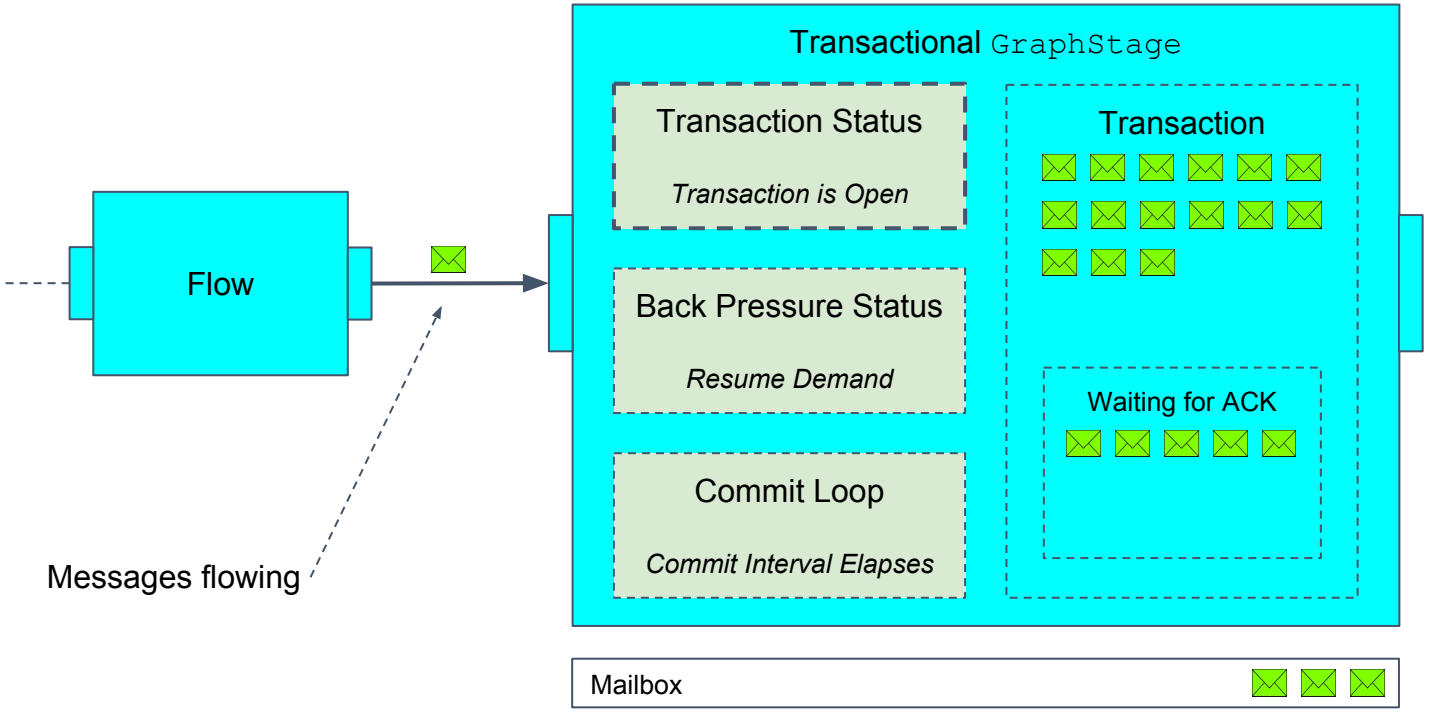


openclipart

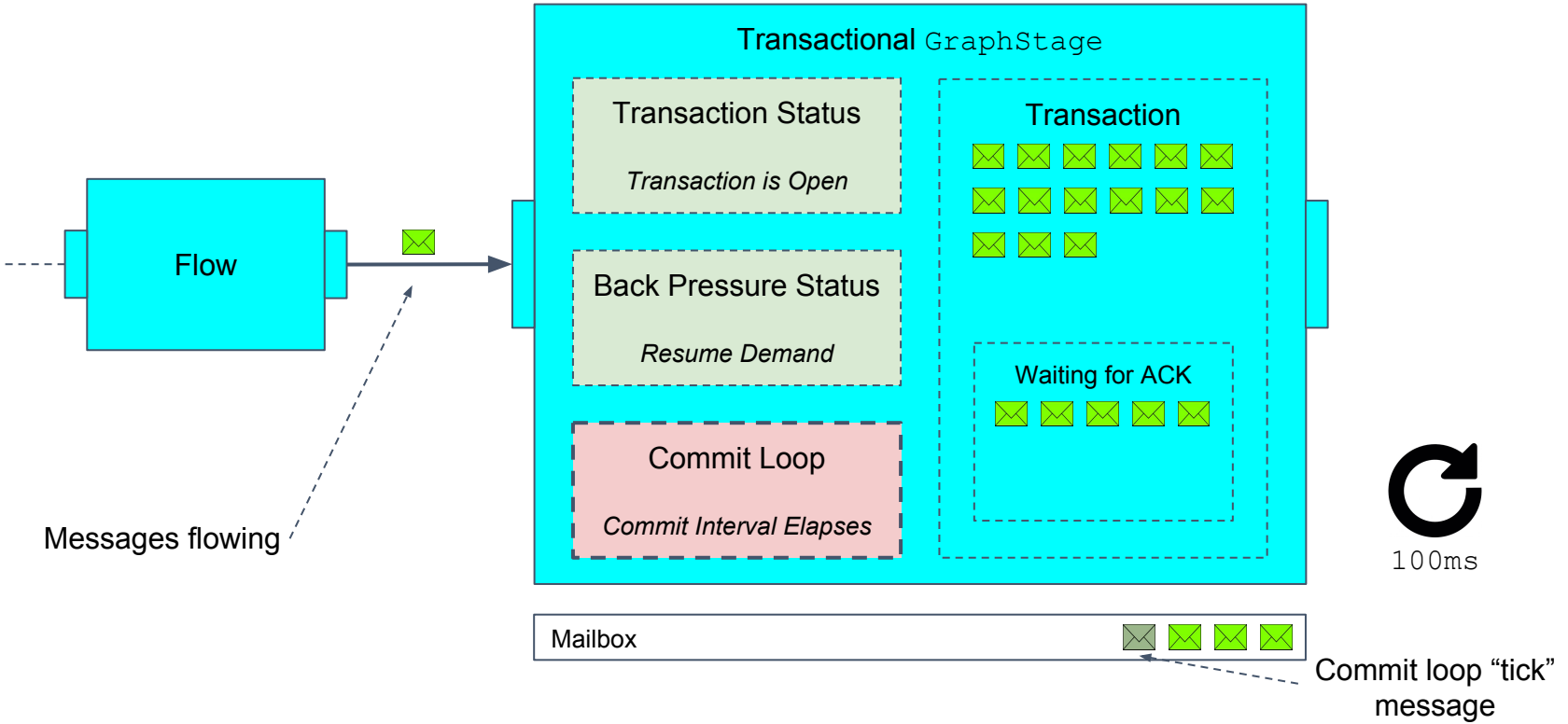
Transactional `GraphStage` (1/7)



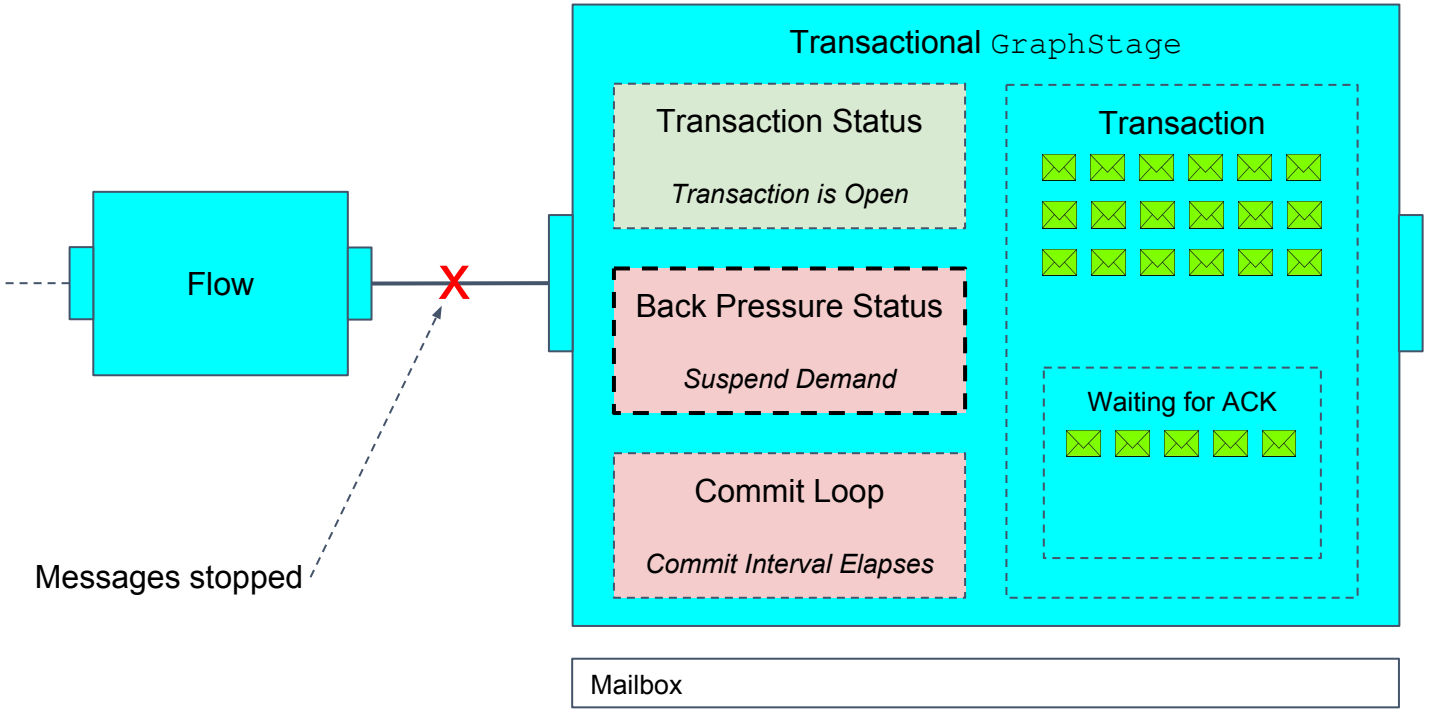
Transactional `GraphStage` (2/7)



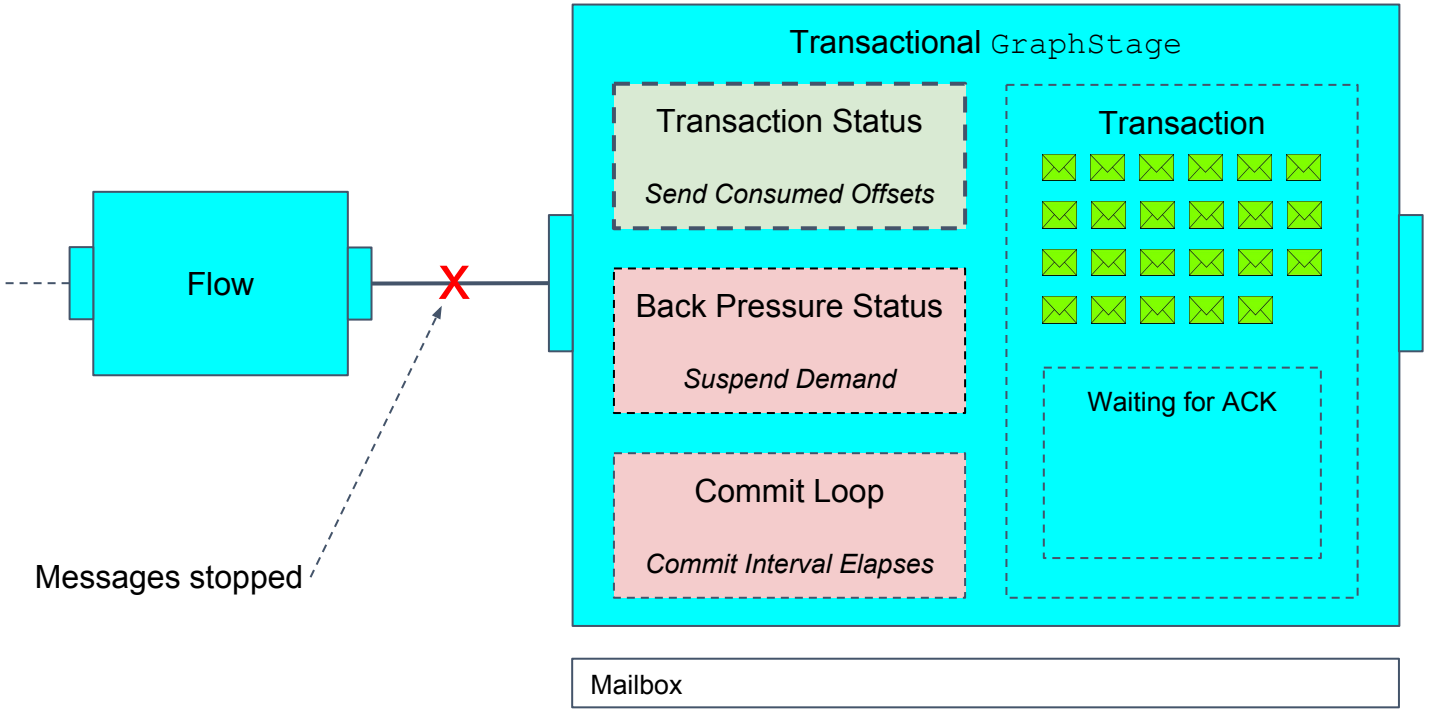
Transactional `GraphStage` (3/7)



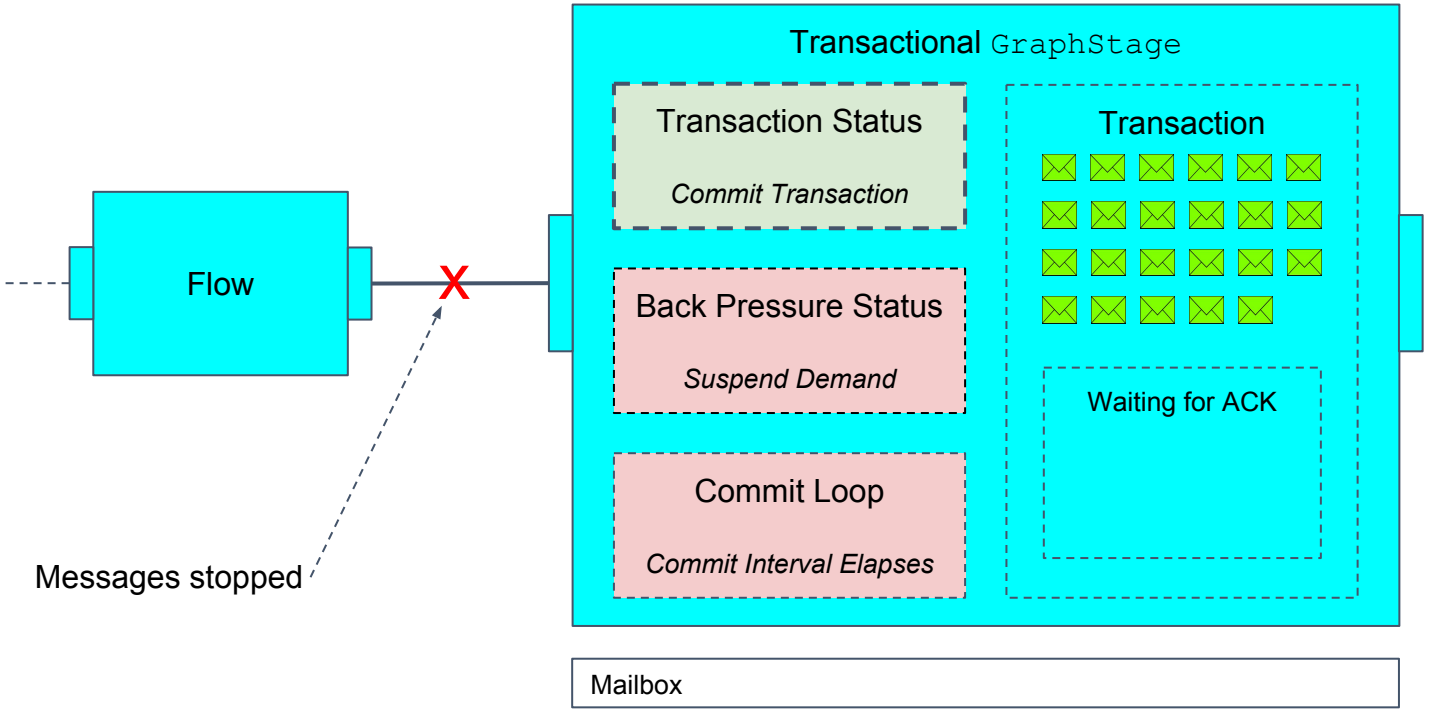
Transactional `GraphStage` (4/7)



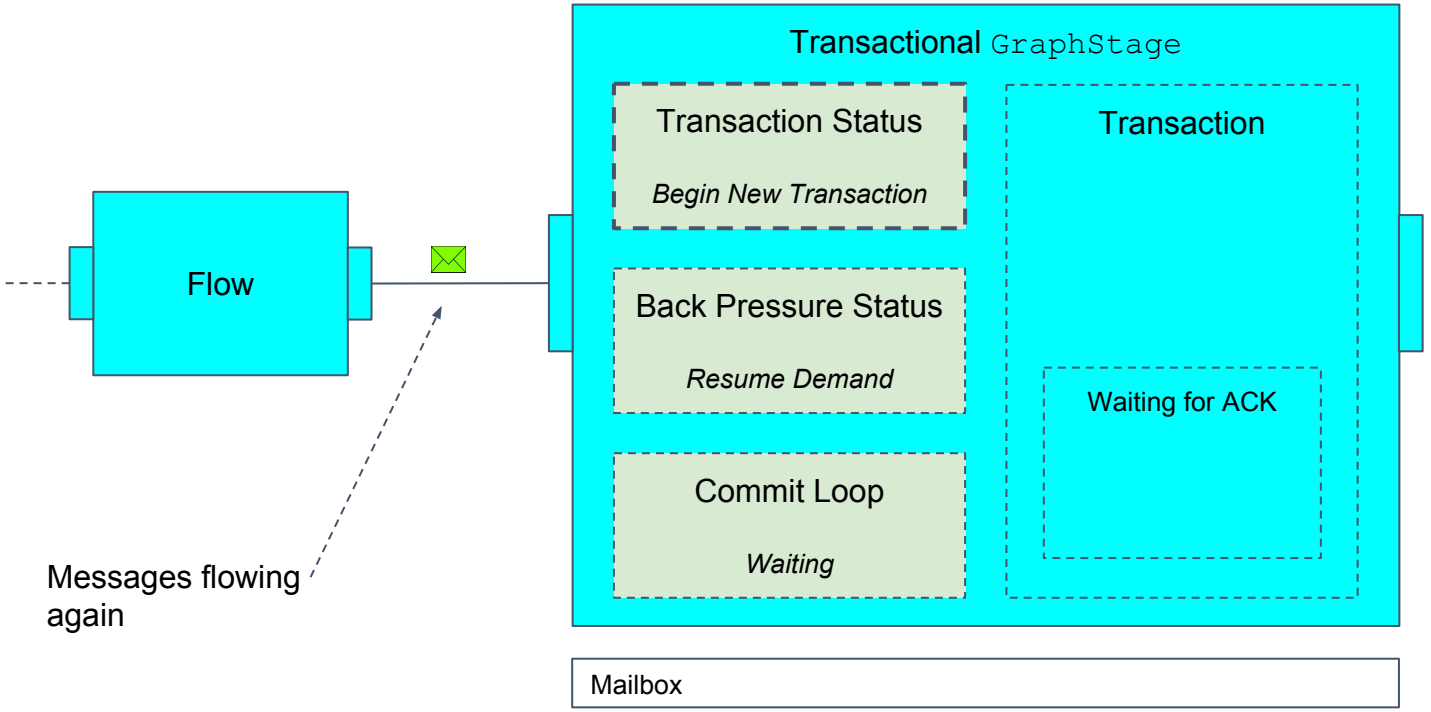
Transactional `GraphStage` (5/7)



Transactional `GraphStage` (6/7)



Transactional `GraphStage` (7/7)



Alpakka Kafka Transactions

```
val producerSettings = ProducerSettings(system, new StringSerializer, new ByteArraySerializer)
 .withBootstrapServers("localhost:9092")
 .withEosCommitInterval(100.millis)

val control =
 Transactional
   .source(consumerSettings, Subscriptions.topics("source-topic"))
   .via(transform)
   .map { msg =>
     ProducerMessage.Single(new ProducerRecord[String, Array[Byte]]("sink-topic", msg.record.value),
       msg.partitionOffset)
   }
   .to(Transactional.sink(producerSettings, "transactional-id"))
   .run()
```

Optionally provide a Transaction commit interval (default is 100ms)

Alpakka Kafka Transactions

```
val producerSettings = ProducerSettings(system, new StringSerializer, new ByteArraySerializer)
  .withBootstrapServers("localhost:9092")
  .withEosCommitInterval(100.millis)

val control =
  Transactional
    .source(consumerSettings, Subscriptions.topics("source-topic"))
    .via(transform)
    .map { msg =>
      ProducerMessage.Single(new ProducerRecord[String, Array[Byte]]("sink-topic", msg.record.value),
        msg.partitionOffset)
    }
    .to(Transactional.sink(producerSettings, "transactional-id"))
    .run()
```

Use `Transactional.source` to propagate necessary info to `Transactional.sink` (CG ID, Offsets)

Alpakka Kafka Transactions

```
val producerSettings = ProducerSettings(system, new StringSerializer, new ByteArraySerializer)
 .withBootstrapServers("localhost:9092")
 .withEosCommitInterval(100.millis)

val control =
 Transactional
   .source(consumerSettings, Subscriptions.topics("source-topic"))
   .via(transform)
   .map { msg =>
     ProducerMessage.Single(new ProducerRecord[String, Array[Byte]]("sink-topic", msg.record.value),
       msg.partitionOffset)
   }
   .to(Transactional.sink(producerSettings, "transactional-id"))
   .run()
```

Call `Transactional.sink` or `.flow` to produce and commit messages.

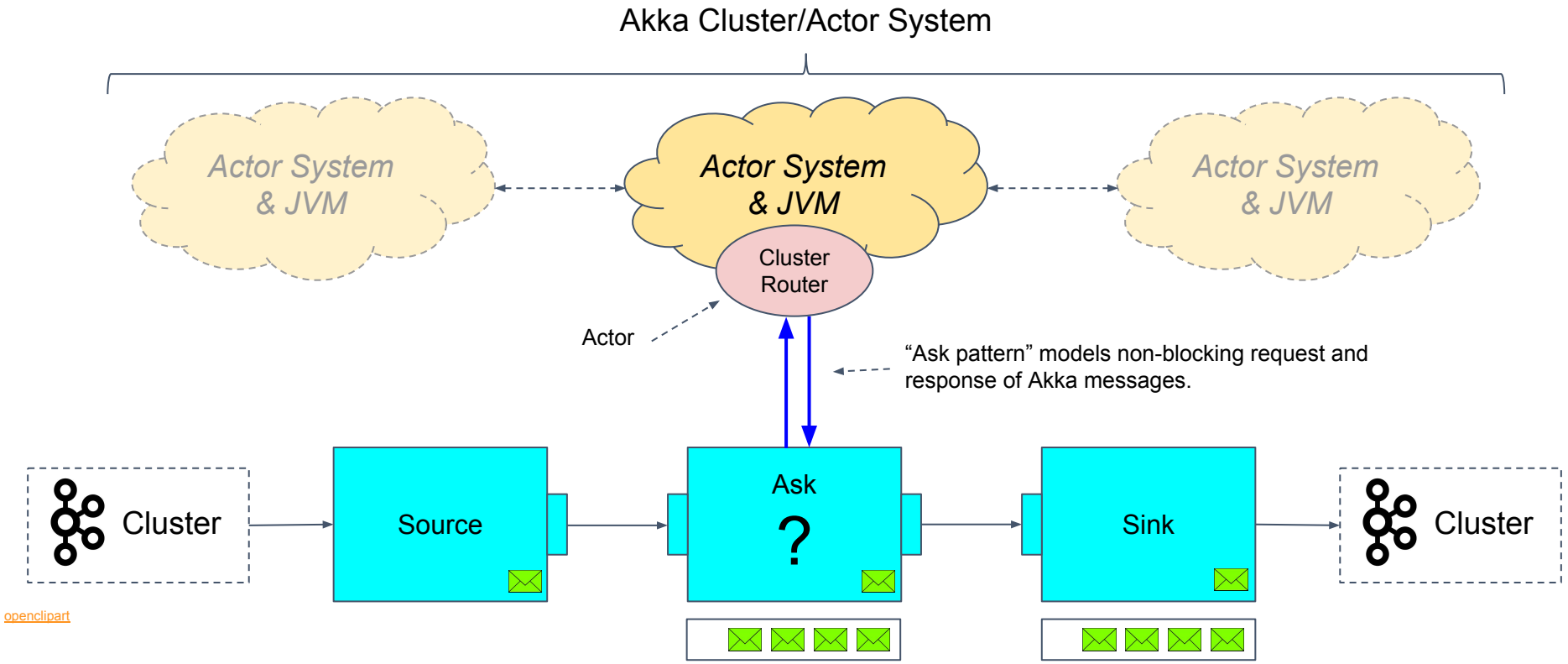
Complex Event Processing

What is Complex Event Processing (CEP)?

Complex event processing, or CEP, is event processing that combines data from multiple sources to infer events or patterns that suggest more complicated circumstances.

Foundations of Complex Event Processing, Cornell

Calling into an Akka Actor System



Actor System Integration

```
class ProblemSolverRouter extends Actor {
  def receive = {
    case problem: Problem =>
      val solution = businessLogic(problem)
      sender() ! solution // reply to the ask
  }
}
...
val control = Consumer
   .committableSource(consumerSettings, Subscriptions.topics("topic1", "topic2"))
   ...
   .mapAsync(parallelism = 5)(problem => (problemSolverRouter ? problem).mapTo[Solution])
   ...
   .run()
```

Transform your stream by processing messages in an Actor System. All you need is an `ActorRef`.

Actor System Integration

```
class ProblemSolverRouter extends Actor {
  def receive = {
    case problem: Problem =>
      val solution = businessLogic(problem)
      sender() ! solution // reply to the ask
  }
}
...
val control = Consumer
   .committableSource(consumerSettings, Subscriptions.topics("topic1", "topic2"))
   ...
   .mapAsync(parallelism = 5)(problem => (problem ? problem).mapTo[Solution])
   ...
   .run()
```

Use Ask pattern (? function) to call provided ActorRef to get an async response

Actor System Integration

```
class ProblemSolverRouter extends Actor {
  def receive = {
    case problem: Problem =>
      val solution = businessLogic(problem)
      sender() ! solution // reply to the ask
  }
}
...
val control = Consumer
   .committableSource(consumerSettings, Subscriptions.topics("topic1", "topic2"))
   ...
   .mapAsync(parallelism = 5)(problem => (problemSolverRouter ? problem).mapTo[Solution])
   ...
   .run()
```

Parallelism used to limit how many messages in flight so we don't overwhelm mailbox of destination Actor and maintain stream back-pressure.

Persistent Stateful Stages

Options for implementing Stateful Streams

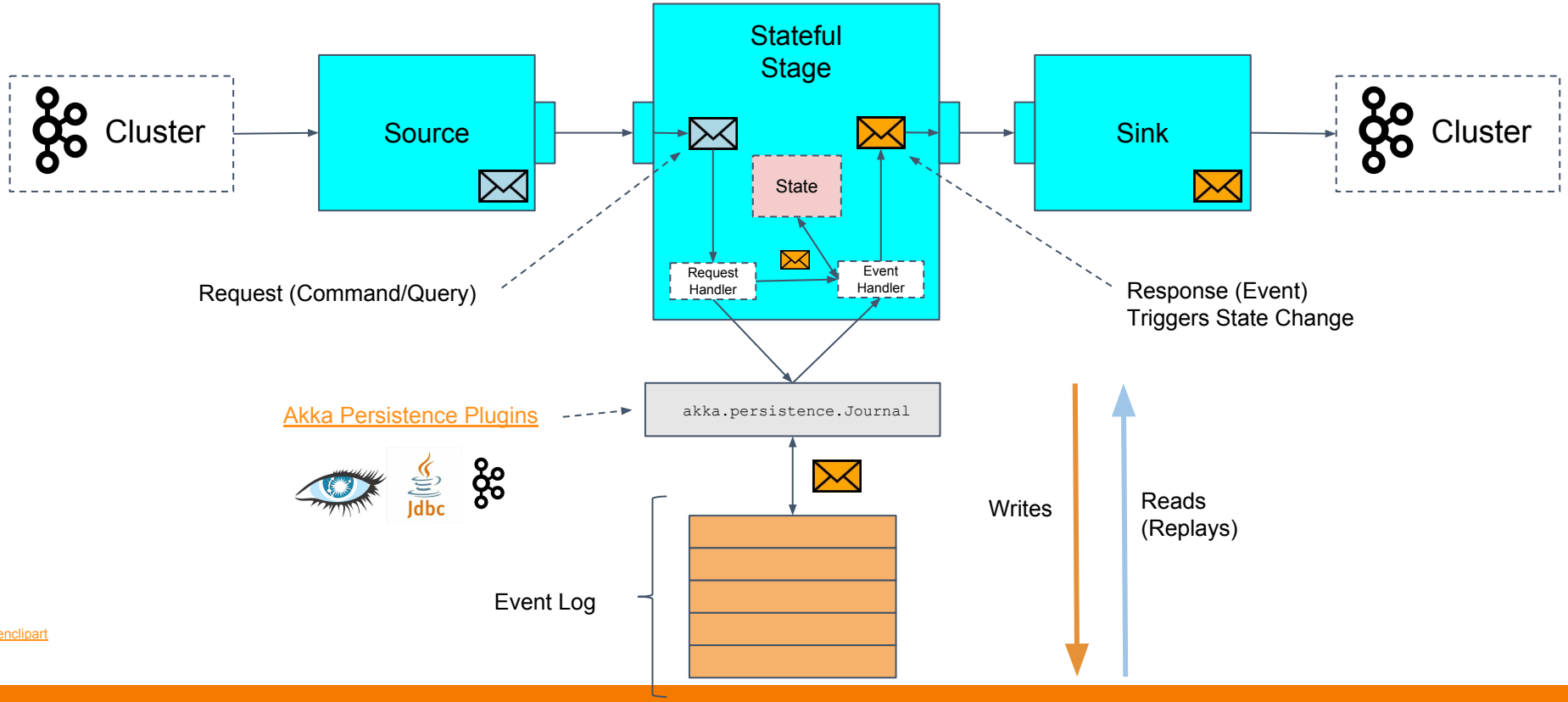
1. Provided Akka Streams stages: `fold`, `scan`, etc.
2. Custom GraphStage
3. Call into an Akka Actor System

Persistent Stateful Stages using Event Sourcing

1. Recover state after failure
2. Create an event log
3. Share state

Sound familiar? `KTable`*'s!*

Persistent `GraphStage` using Event Sourcing



openclipart

krasserm / **akka-stream-eventsourcing**

This project brings to Akka Streams what Akka Persistence brings to Akka Actors: persistence via event sourcing.

Public Domain Vectors

Experimental

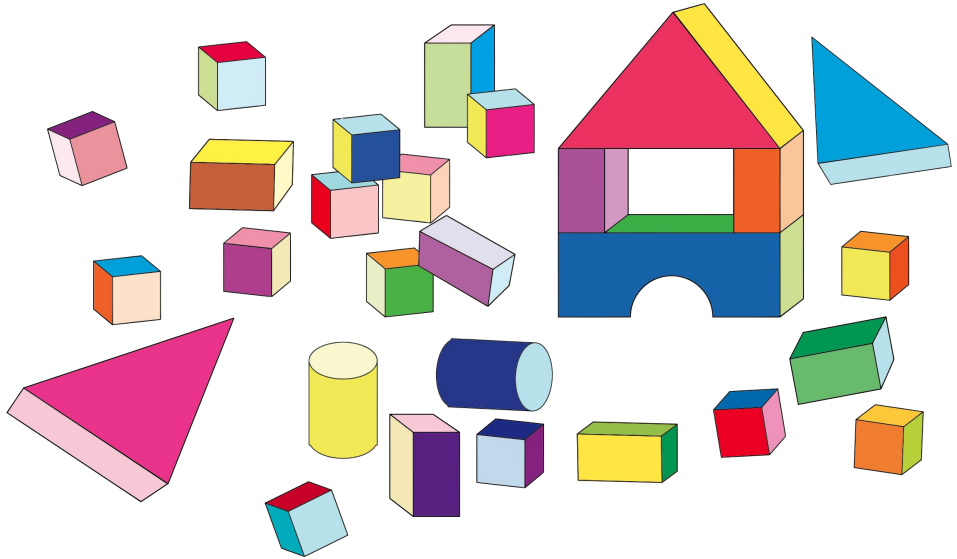
New in Alpakka Kafka 1.0

Alpakka Kafka 1.0 Release Notes

Released **Feb 28, 2019**. Highlights:

- Upgraded the Kafka client to version 2.0.0 #544 by @fr3akX
  - Support new API's from KIP-299: Fix Consumer indefinite blocking behaviour in #614 by @zaharidichev
- New Committer.sink for standardised committing #622 by @rtimush
- Commit with metadata #563 and #579 by @johnclara
- Factored out akka.kafka.testkit for internal and external use: see Testing
- Support for merging commit batches #584 by @rtimush
- Reduced risk of message loss for partitioned sources #589
- Expose Kafka errors to stream #617
- Java APIs for all settings classes #616
- Much more comprehensive tests

Conclusion



openclipart

Lightbend

Thank You!

Sean Glover

@seg1o

in/seanaglover

sean.glover@lightbend.com

Free eBook!

https://bit.ly/2J9xmZm