Reactive Streaming with Kafka and Akka

{codemotion}

25.10.2016, Sebastian Alfers

Reactive Streaming with Kafka and Akka

plista

Akka

concurrent, distributed, fault tolerant apps on the JVM (*Actor Model*)

Reactive Streaming with Kafka and Akka

Akka

concurrent, distributed, fault tolerant apps on the JVM (*Actor Model*)

Reactive Streams

build data pipelines with focus on back pressure and concurrency (Akka module)

Reactive Streaming with Kafka and Akka

plista

- Akka
  - concurrent, distributed, fault tolerant apps on the JVM (*Actor Model*)
- Reactive Streams
  - build data pipelines with focus on back pressure and concurrency (Akka module)
- **Kafka**
  - distributed message log for streaming data


Platzierung unter Ihren Artikeln und Beiträgen
Das könnte Sie auch interessieren
Die Kino-Highlights des Monats
Schenken mit Wow-Effekt
Finde das passende Auto
ANZEIGE
Wie oft trainierst du wöchentlich?
SPONSORED
Der neue Detox-Trend
Besser als der Ruf
powered by plista
Auch interessant
Schenken mit Wow-Effekt
Finde Sie das passende Auto
ANZEIGE
Ad
Ad

Plista

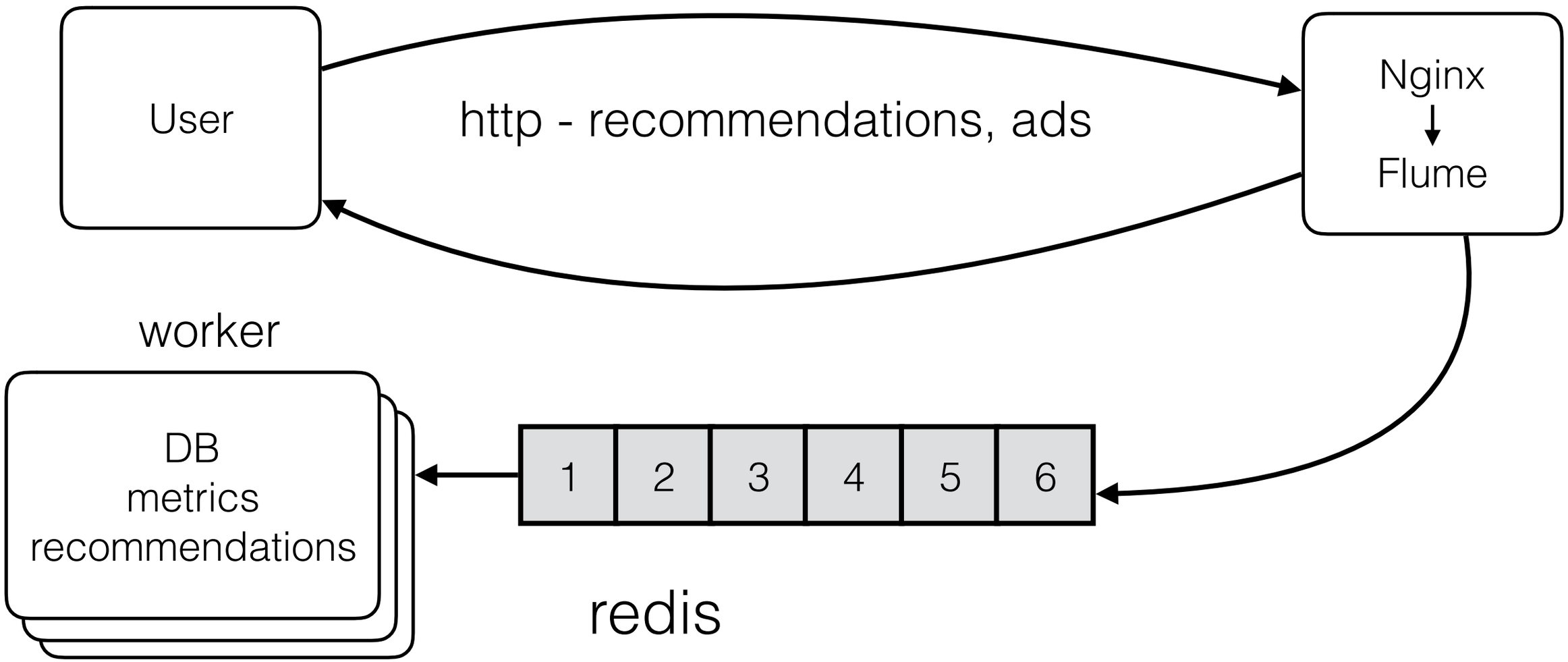
- content recommendation + ads
  - 10k events / second
  - ~150GB data / day

Plista - no async

- content recommendation + ads
  - 10k events / second
  - ~150GB data / day

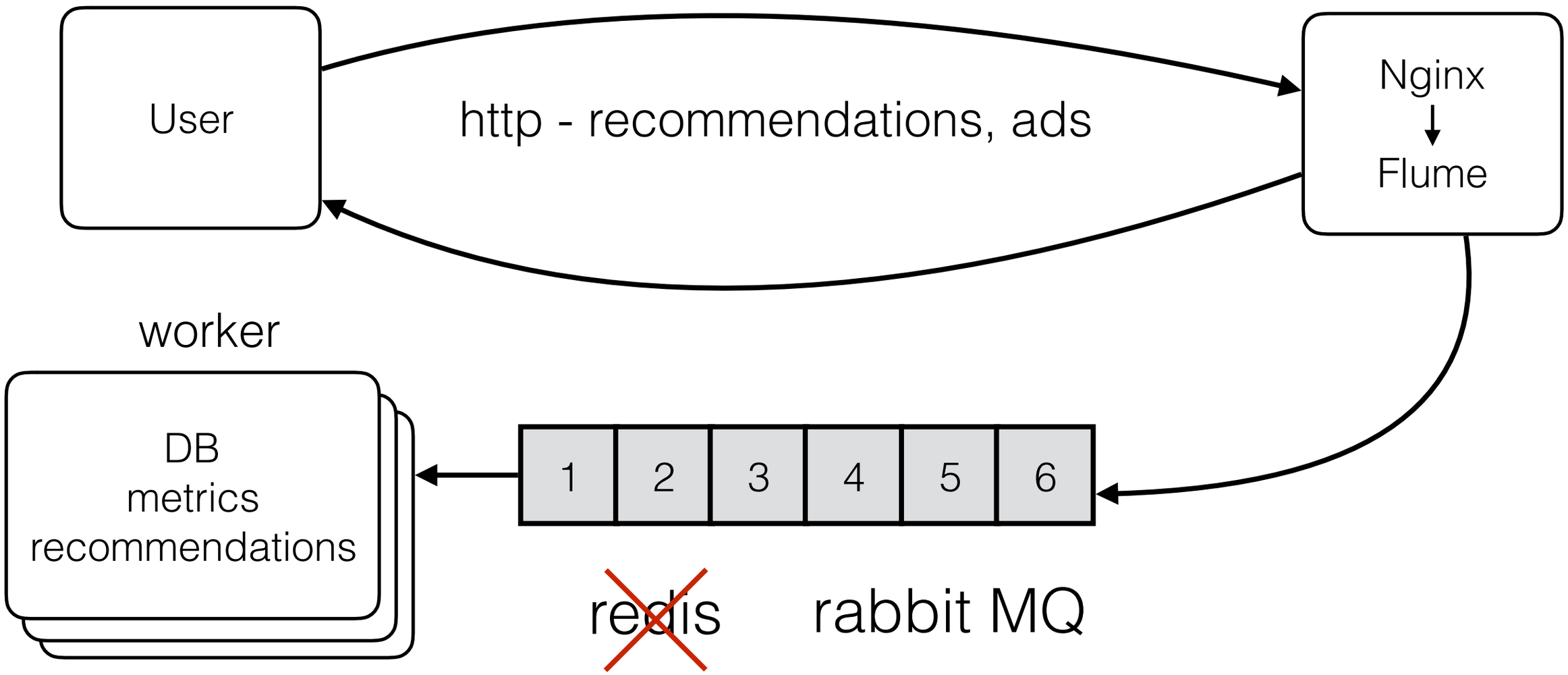
Plista - redis

- content recommendation + ads
  - 10k events / second
  - ~150GB data / day



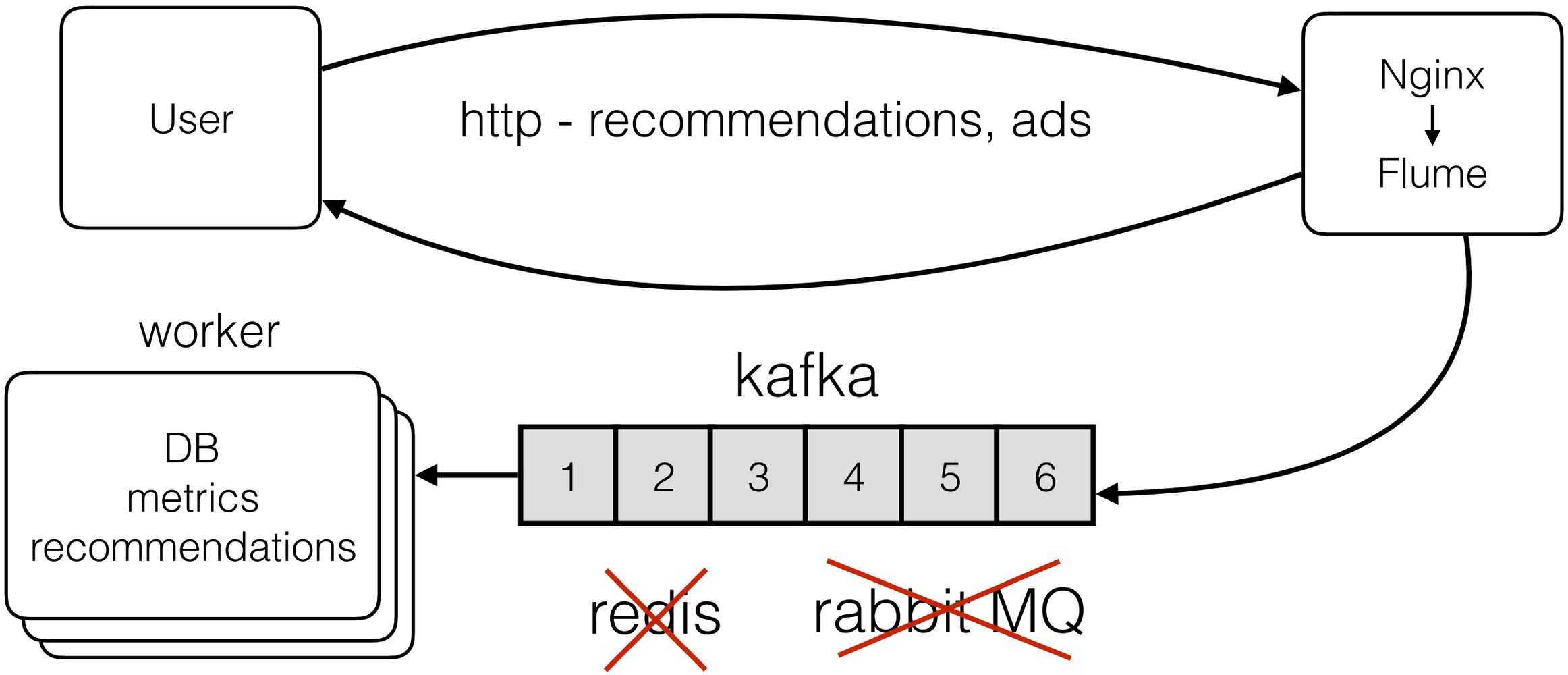
Plista - rabbit

- content recommendation + ads
  - 10k events / second
  - ~150GB data / day



Plista - kafka

- content recommendation + ads
  - 10k events / second
  - ~150GB data / day



Kafka

- „distributed streaming platform“ - message log

Kafka

- „distributed streaming platform“ - message log
- topics - spread **partitions** over **cluster-nodes**

Kafka

- „distributed streaming platform“ - message log
- topics - spread **partitions** over **cluster-nodes**
- automatic leader selection + replication management using **zookeeper**

Kafka

- „distributed streaming platform“ - message log
- topics - spread **partitions** over **cluster-nodes**
- automatic leader selection + replication management using **zookeeper**
- replay!

Kafka

- „distributed streaming platform“ - message log
- topics - spread **partitions** over **cluster-nodes**
- automatic leader selection + replication management using **zookeeper**
- replay!
- consumer have to **commit** (ack)

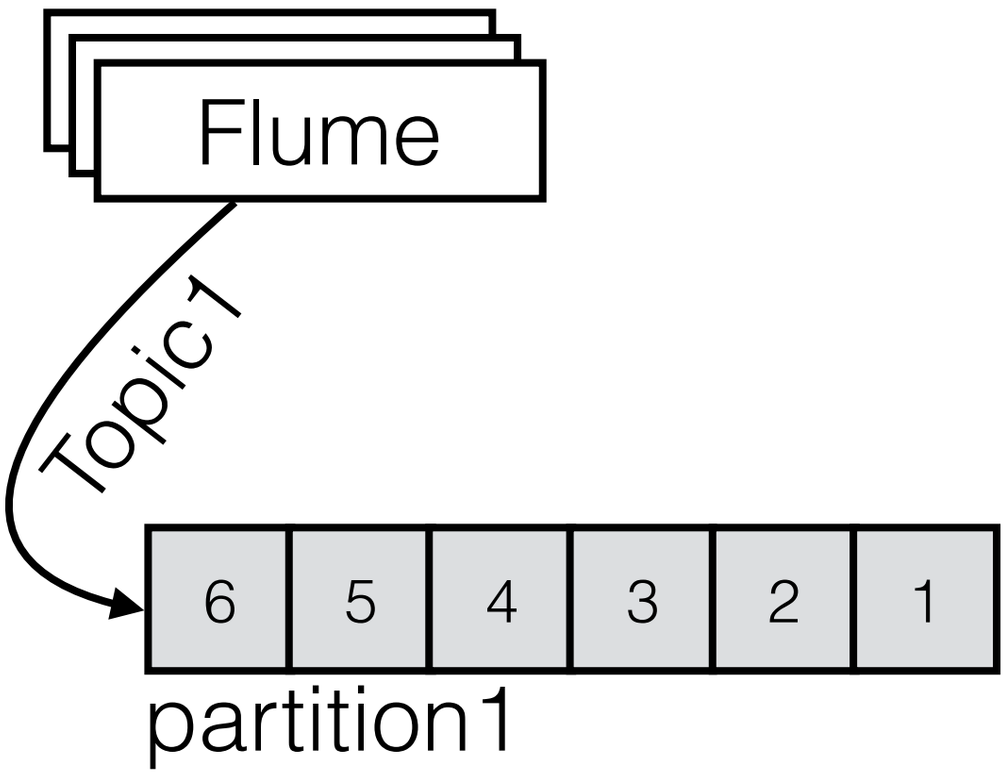
Kafka

- „distributed streaming platform“ - message log
- topics - spread **partitions** over **cluster-nodes**
- automatic leader selection + replication management using **zookeeper**
- replay!
- consumer have to **commit** (ack)
- combines pub-sub + queuing (consumer group)

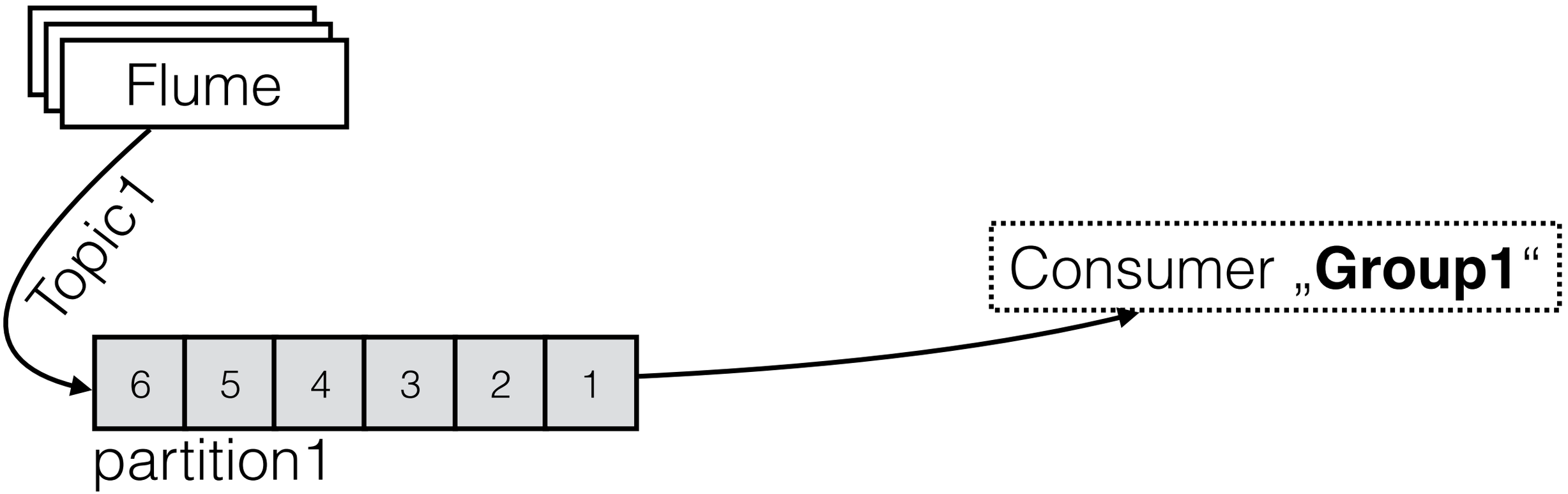
Kafka

- „distributed streaming platform“ - message log
- topics - spread **partitions** over **cluster-nodes**
- automatic leader selection + replication management using **zookeeper**
- replay!
- consumer have to **commit** (ack)
- combines pub-sub + queuing (consumer group)
- data retention (TTL)

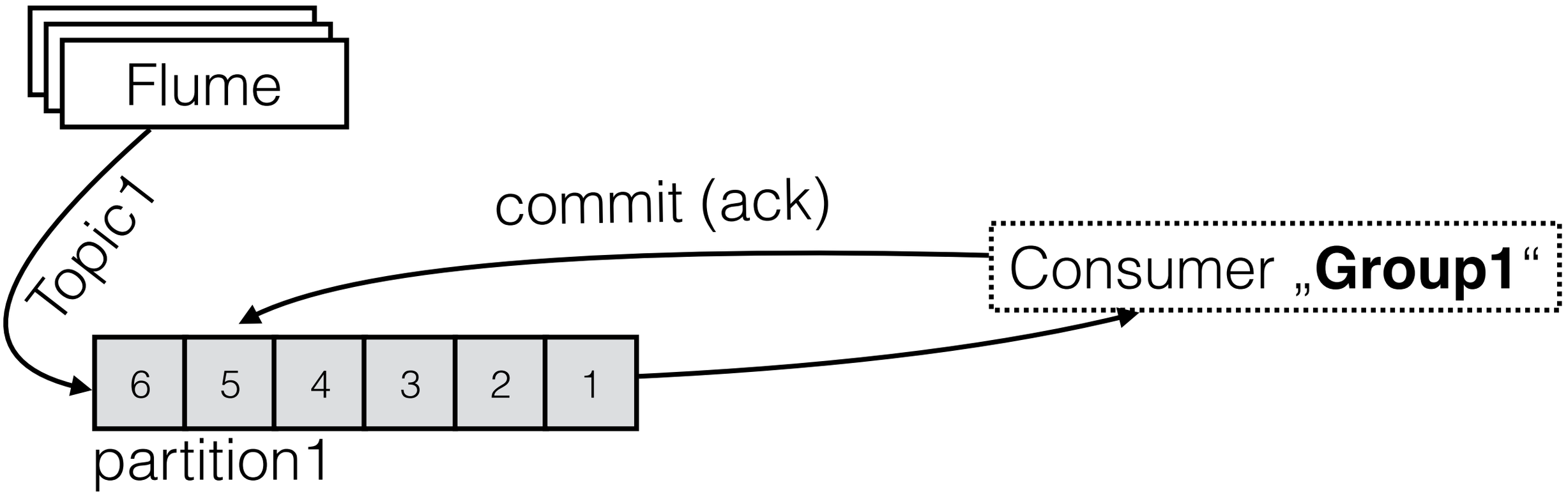
Kafka



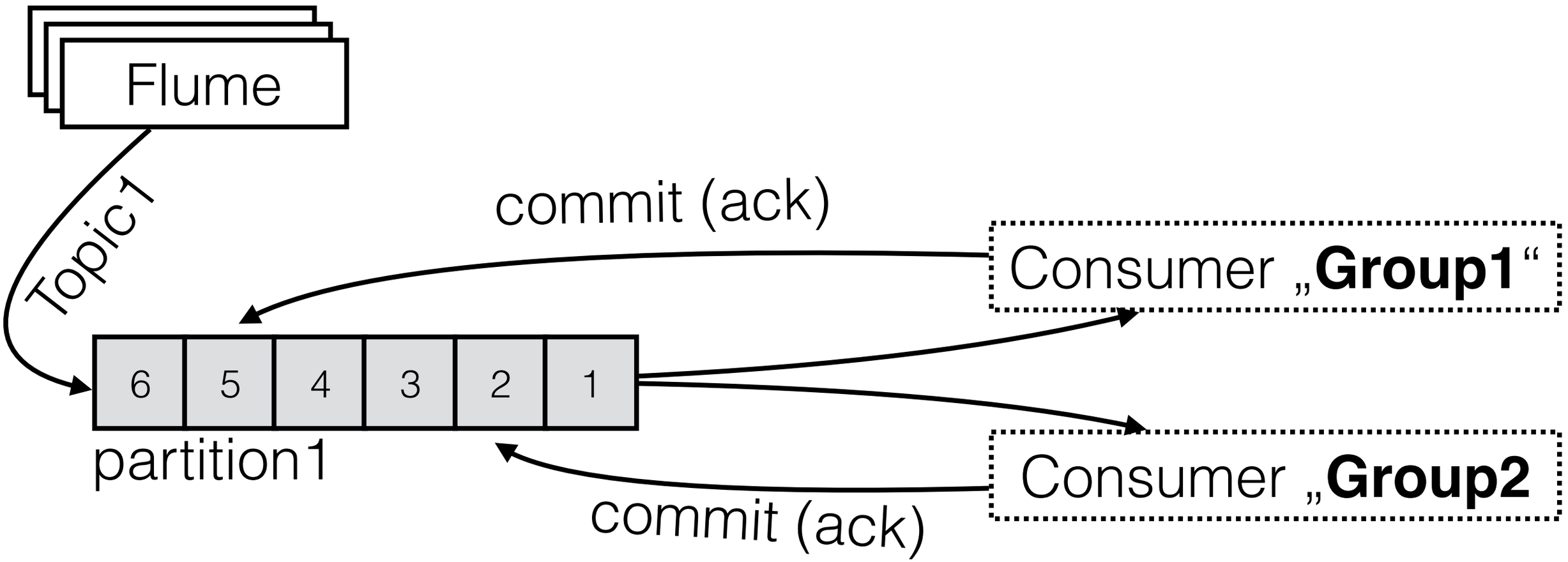
Kafka



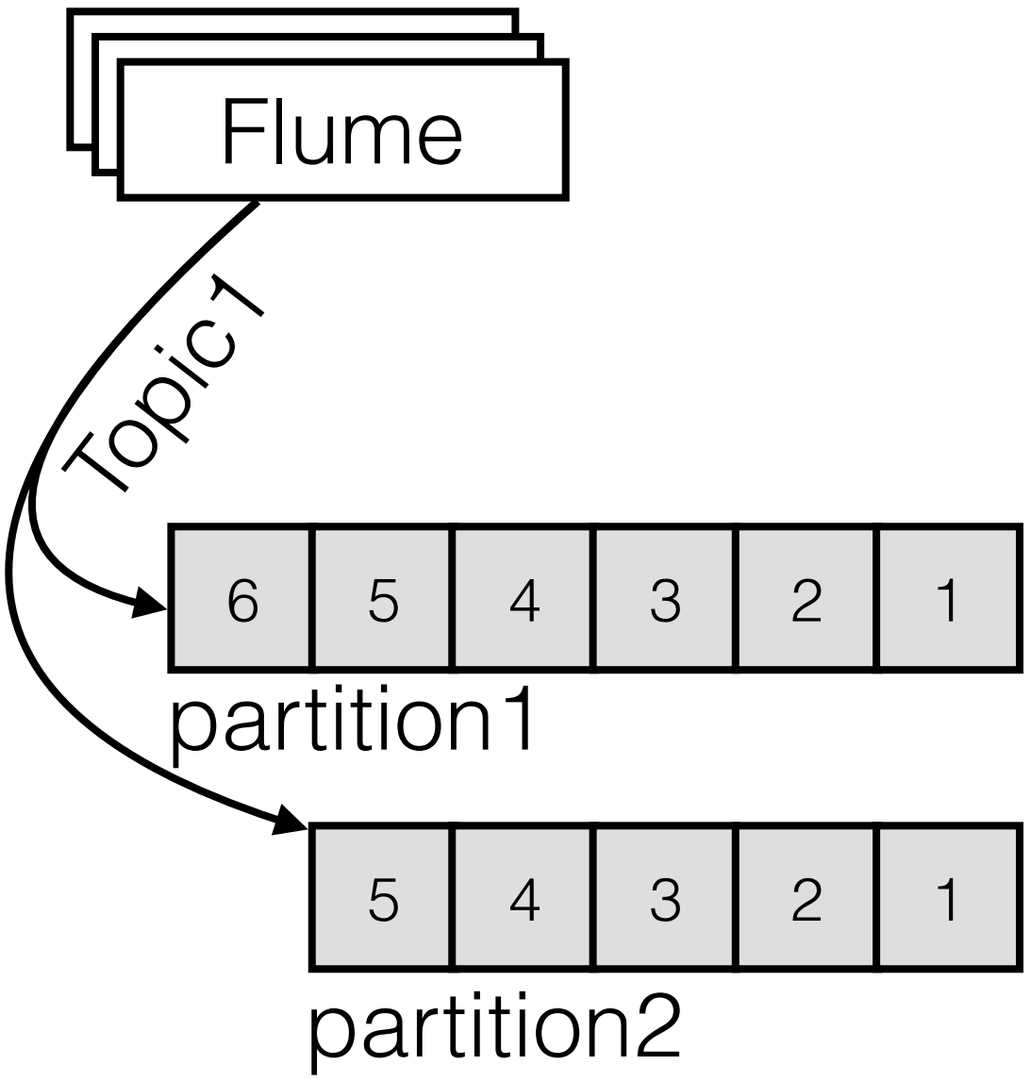
Kafka


Flume
Topic1
commit (ack)
Consumer „**Group1**“
6
5
4
3
2
1
partition1

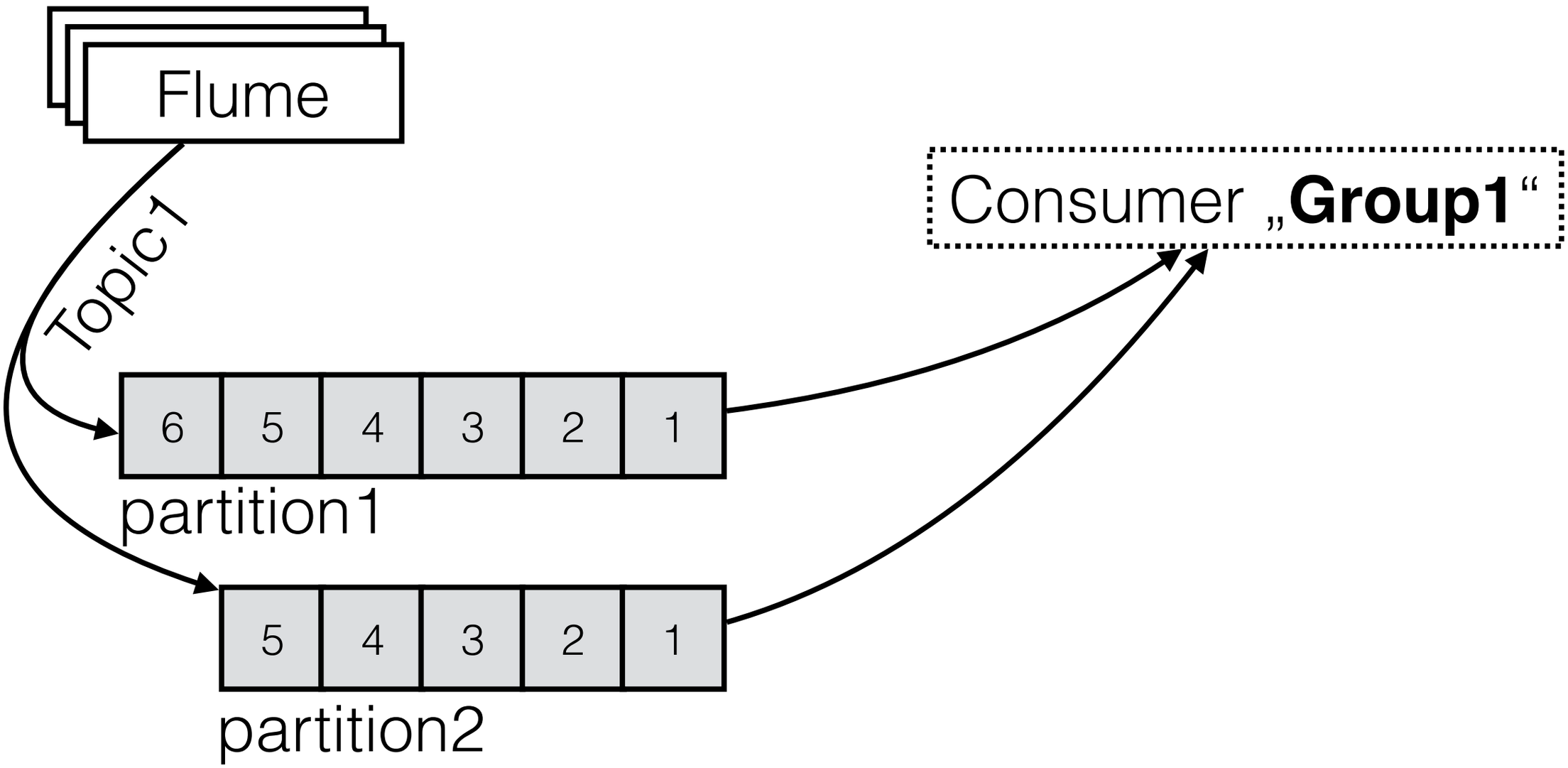
Kafka


Flume
Topic1
commit (ack)
Consumer „**Group1**“
6
5
4
3
2
1
partition1
Consumer „**Group2**
commit (ack)

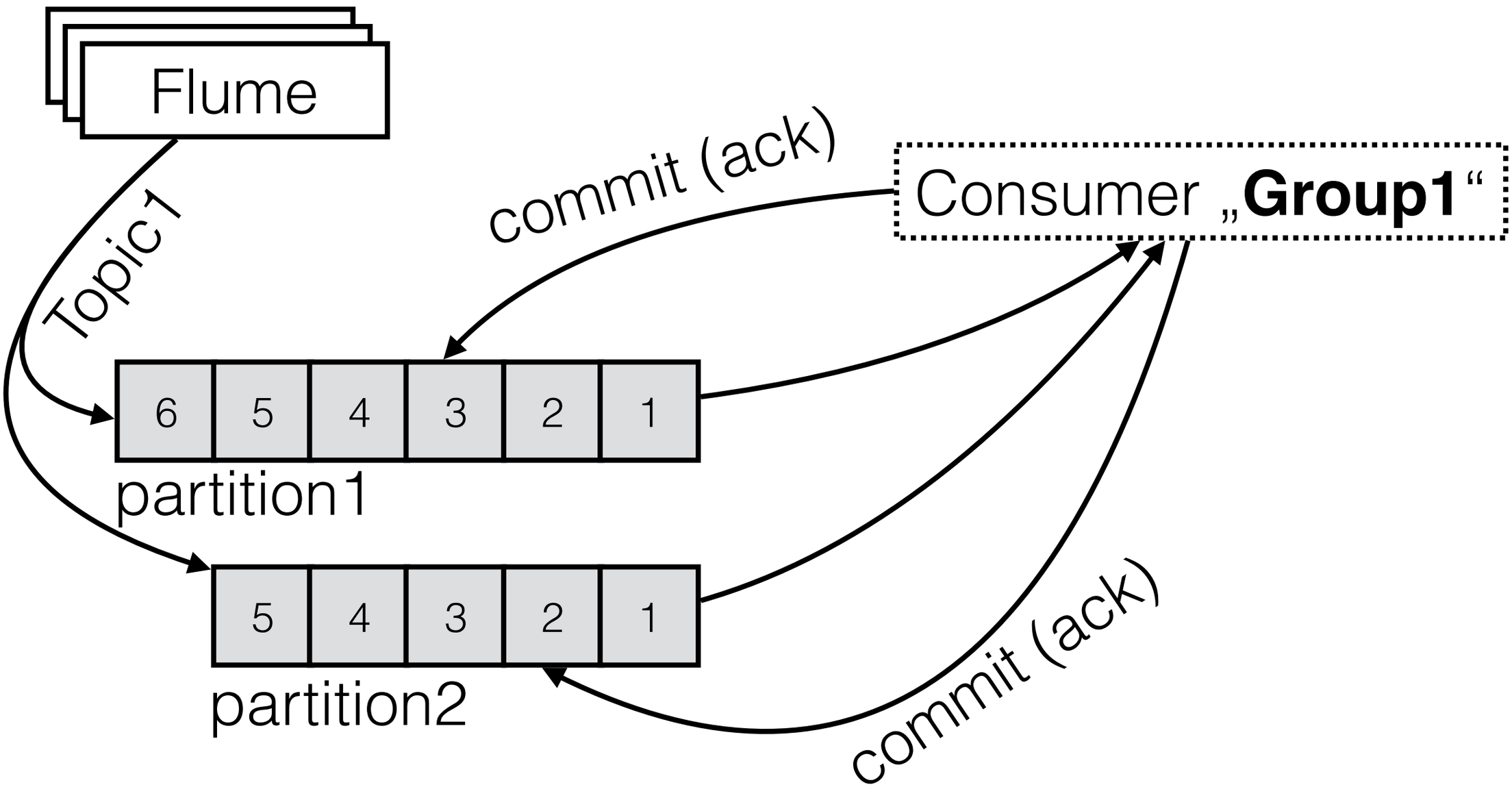
Kafka



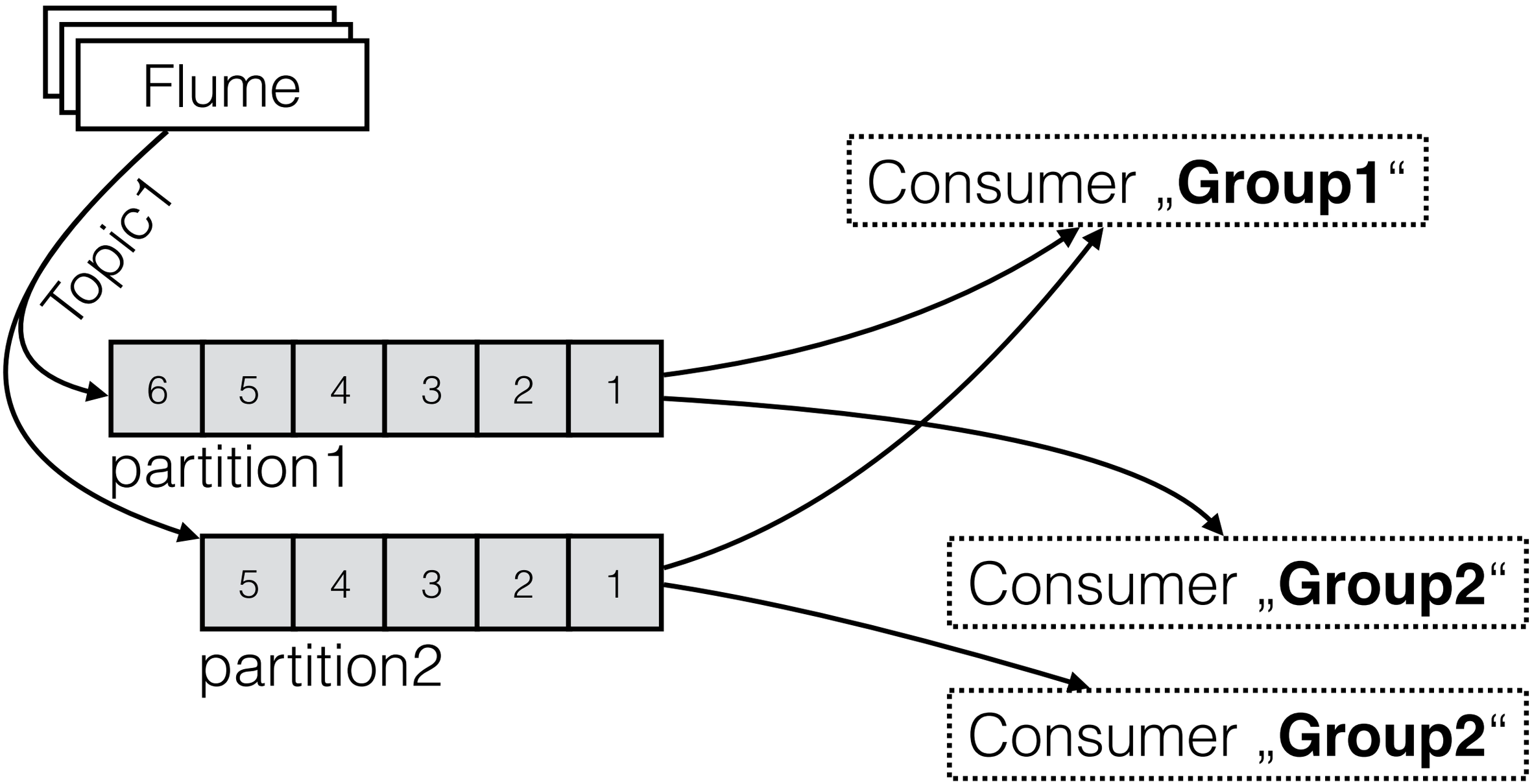
Kafka


Flume
Topic 1
6
5
4
3
2
1
partition1
5
4
3
2
1
partition2
Consumer „**Group1**“

Kafka


Flume
Topic1
commit (ack)
Consumer „**Group1**“
6
5
4
3
2
1
partition1
5
4
3
2
1
partition2
commit (ack)

Kafka

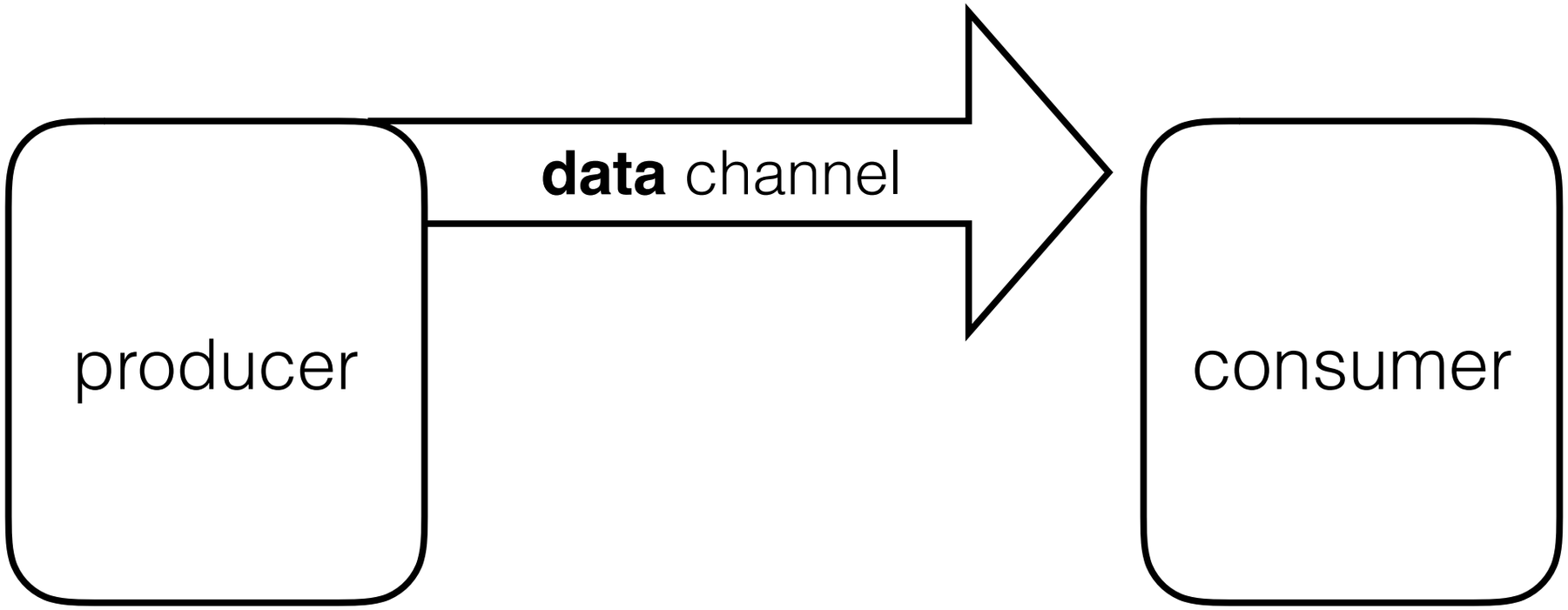

Flume
Topic 1
6
5
4
3
2
1
partition1
5
4
3
2
1
partition2
Consumer „**Group1**“
Consumer „**Group2**“
Consumer „**Group2**“

Reactive Streaming (protocol)

- Reactive Streams Protocol
  - implementation of http://www.reactive-streams.org/
  - pass data between **independent** units of execution
  - non-blocking
  - asynchronous boundaries (thread, tcp, http)

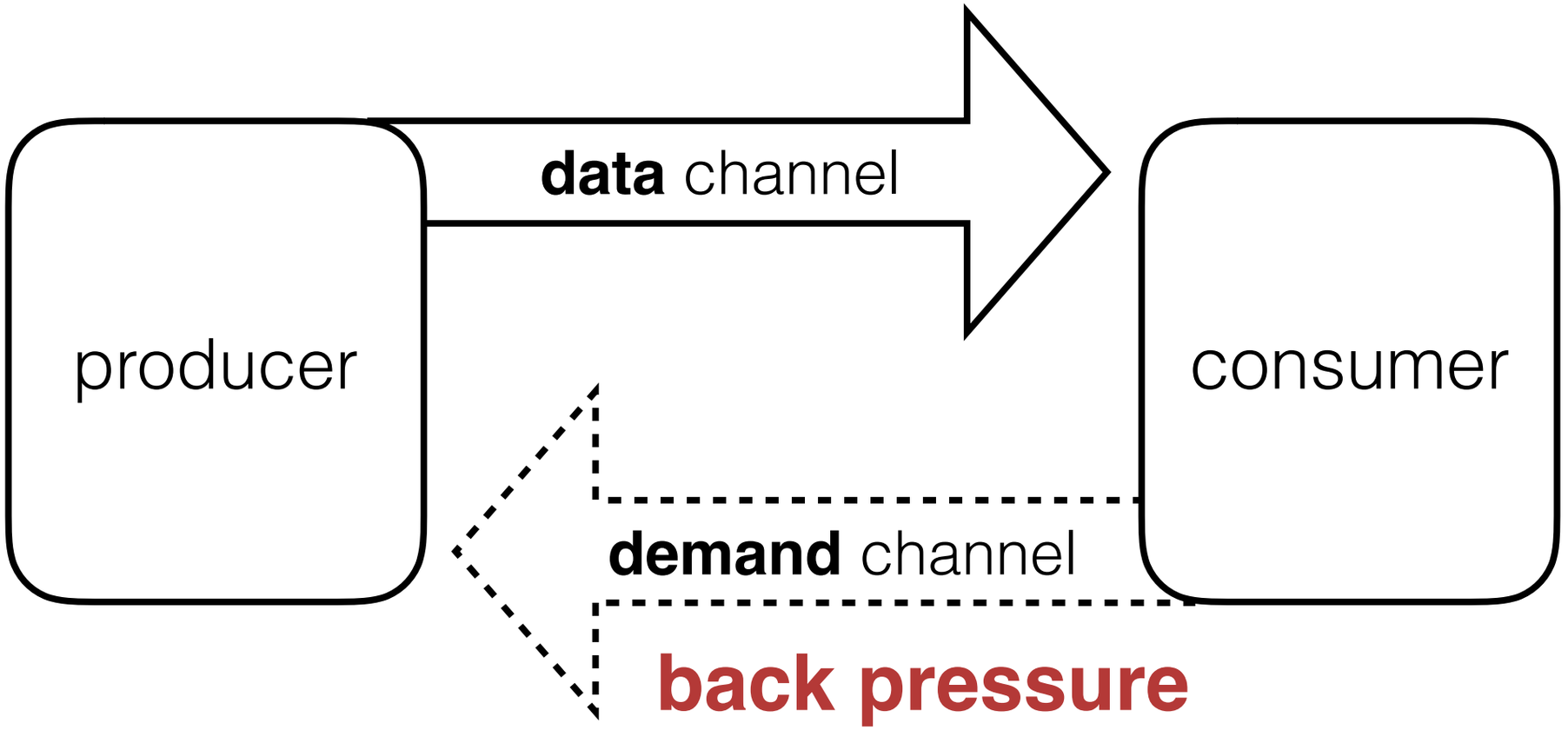
Reactive Streaming (protocol)

- Reactive Streams Protocol
  - implementation of http://www.reactive-streams.org/
  - pass data between **independent** units of execution
  - non-blocking
  - asynchronous boundaries (thread, tcp, http)



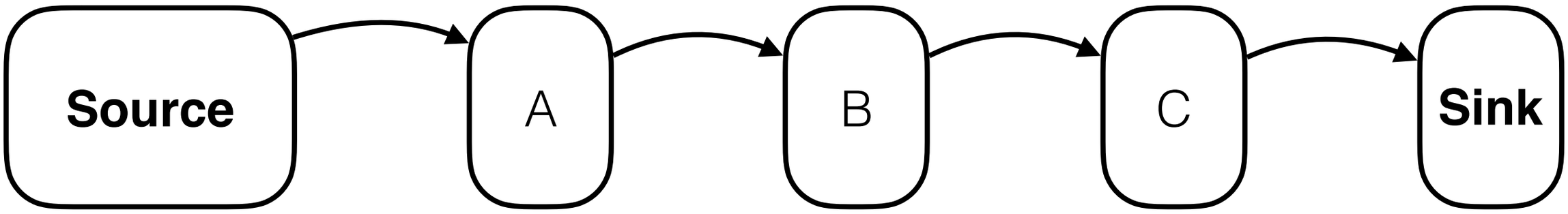
Reactive Streaming (protocol)

- Reactive Streams Protocol
  - implementation of http://www.reactive-streams.org/
  - pass data between **independent** units of execution
  - non-blocking
  - asynchronous boundaries (thread, tcp, http)



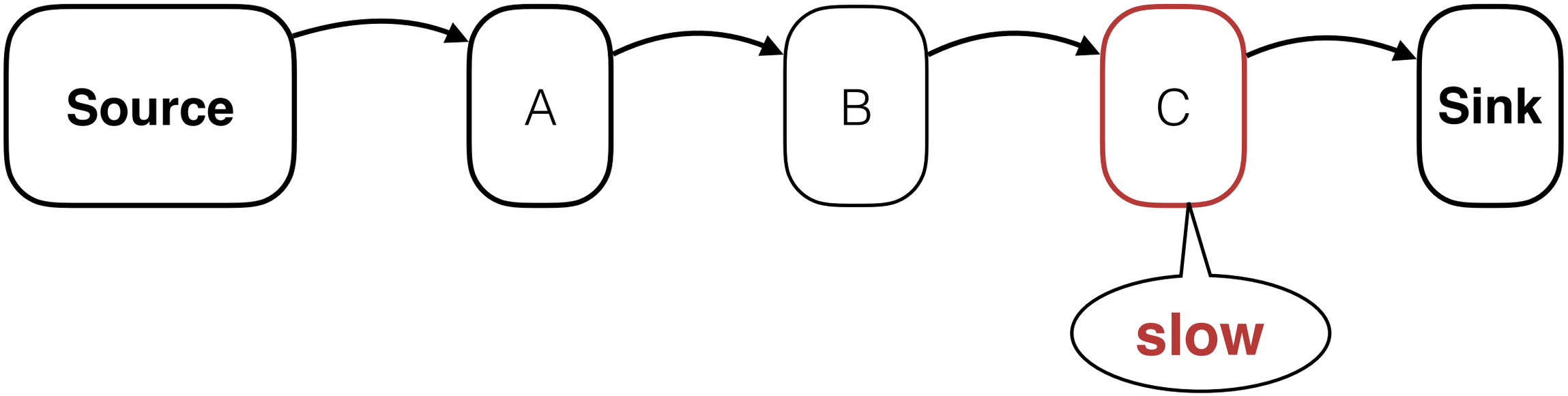
Reactive Streaming (Akka)

- traditional (bounded) data pipelines



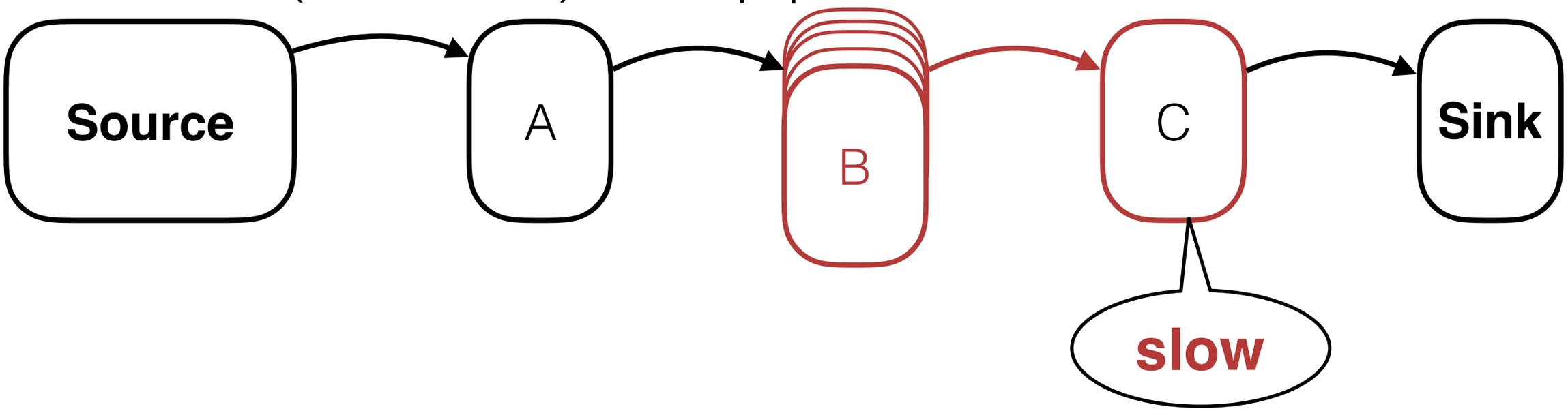
Reactive Streaming (Akka)

- traditional (bounded) data pipelines



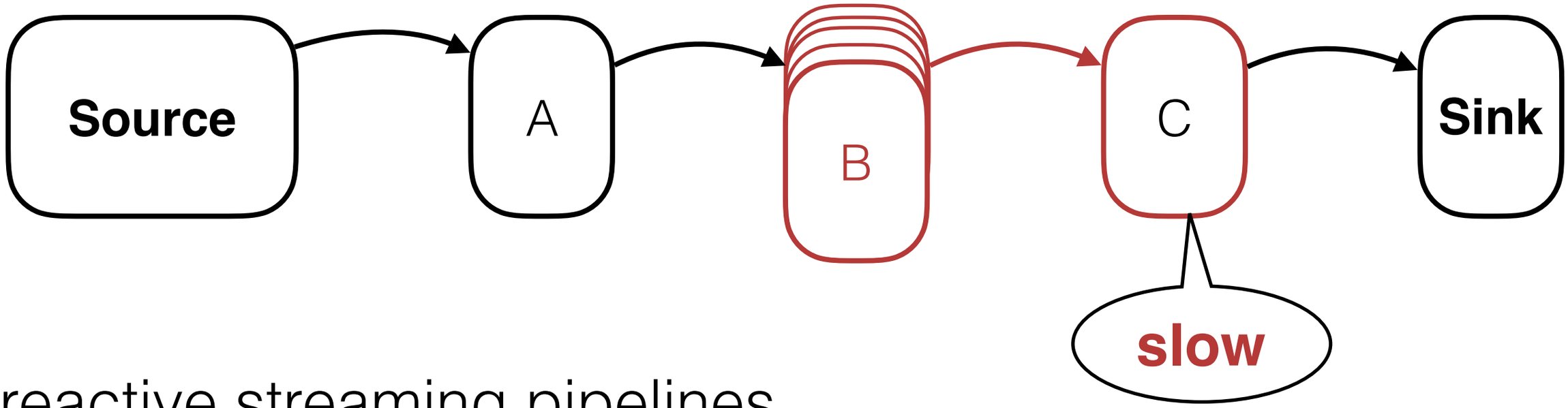
Reactive Streaming (Akka)

- traditional (bounded) data pipelines

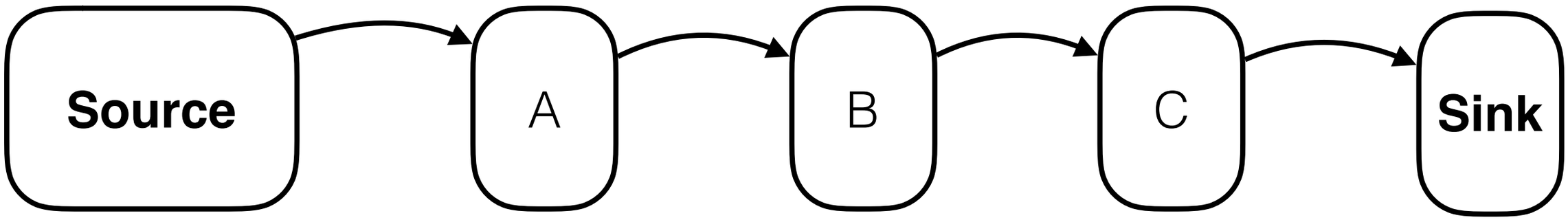


Reactive Streaming (Akka)

- traditional (bounded) data pipelines

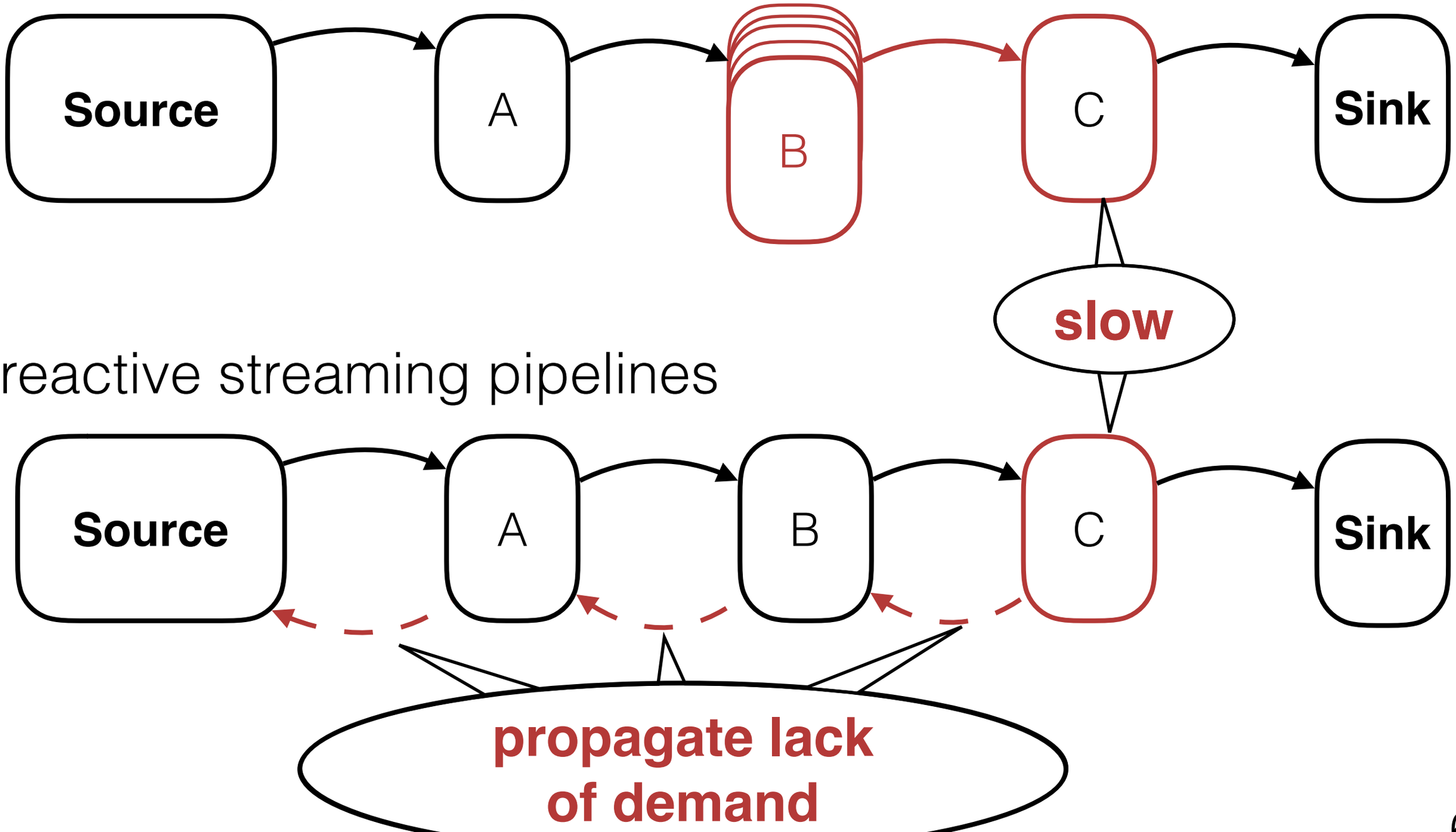


- reactive streaming pipelines



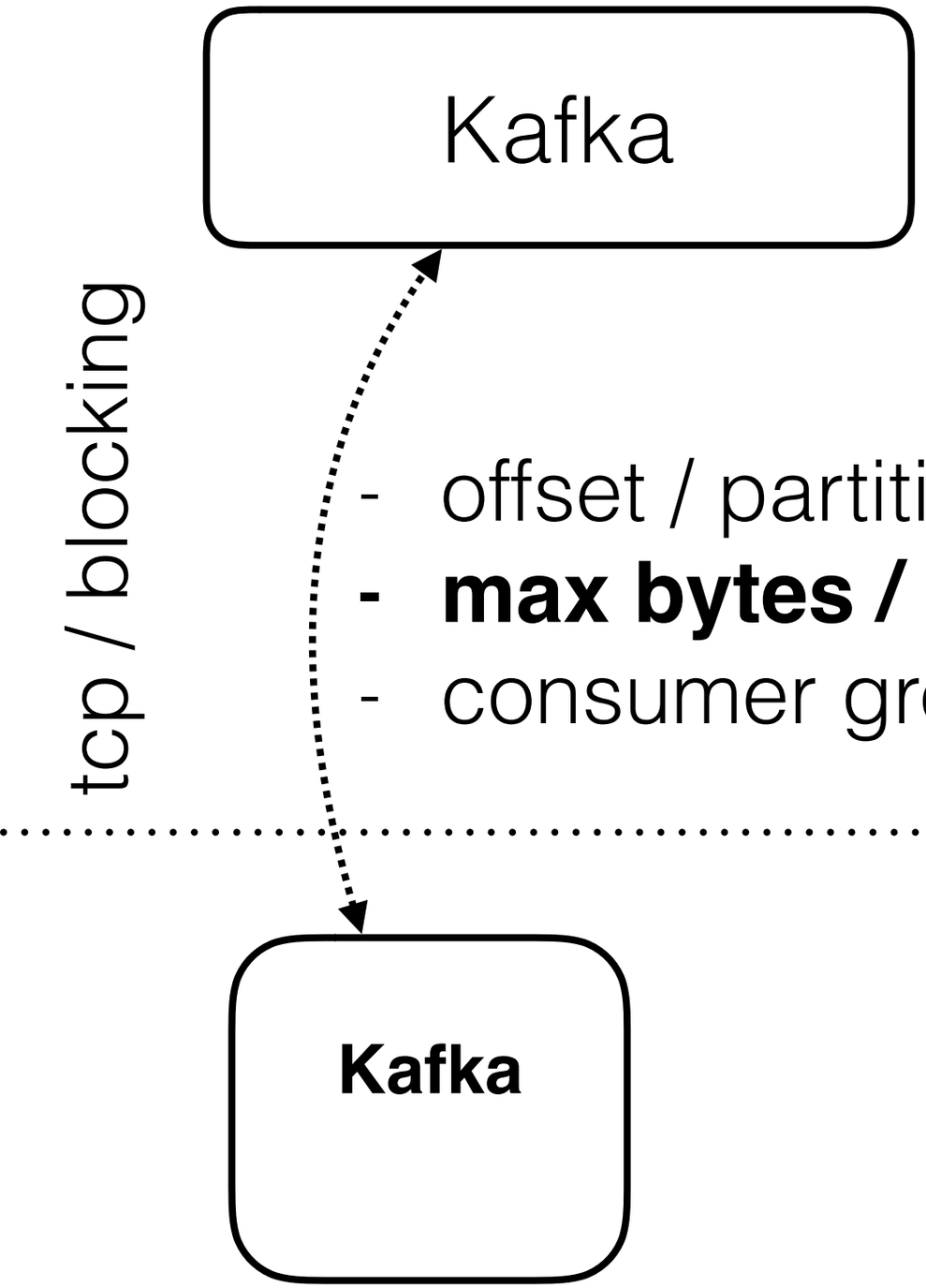
Reactive Streaming (Akka)

- traditional (bounded) data pipelines

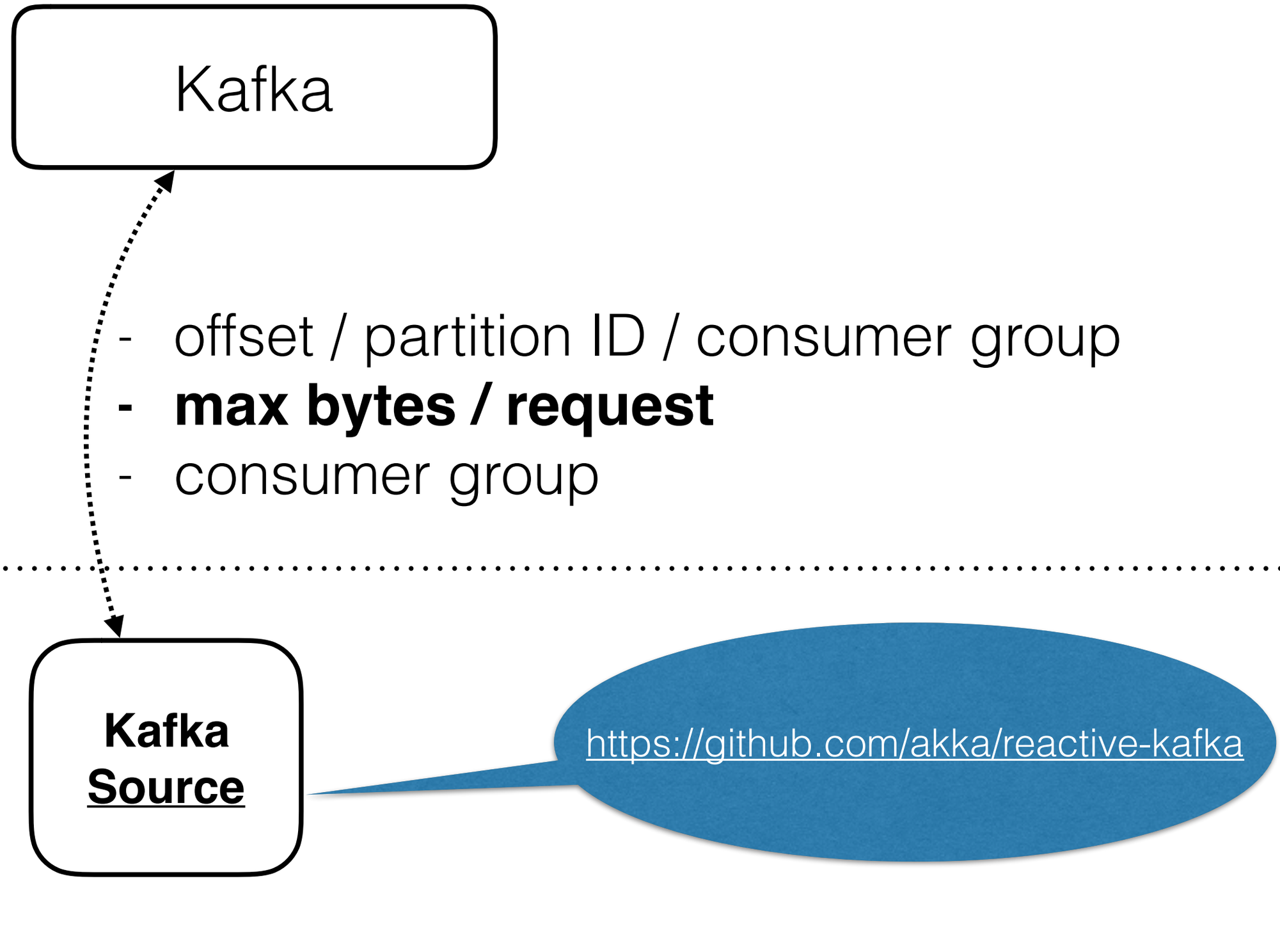


- reactive streaming pipelines

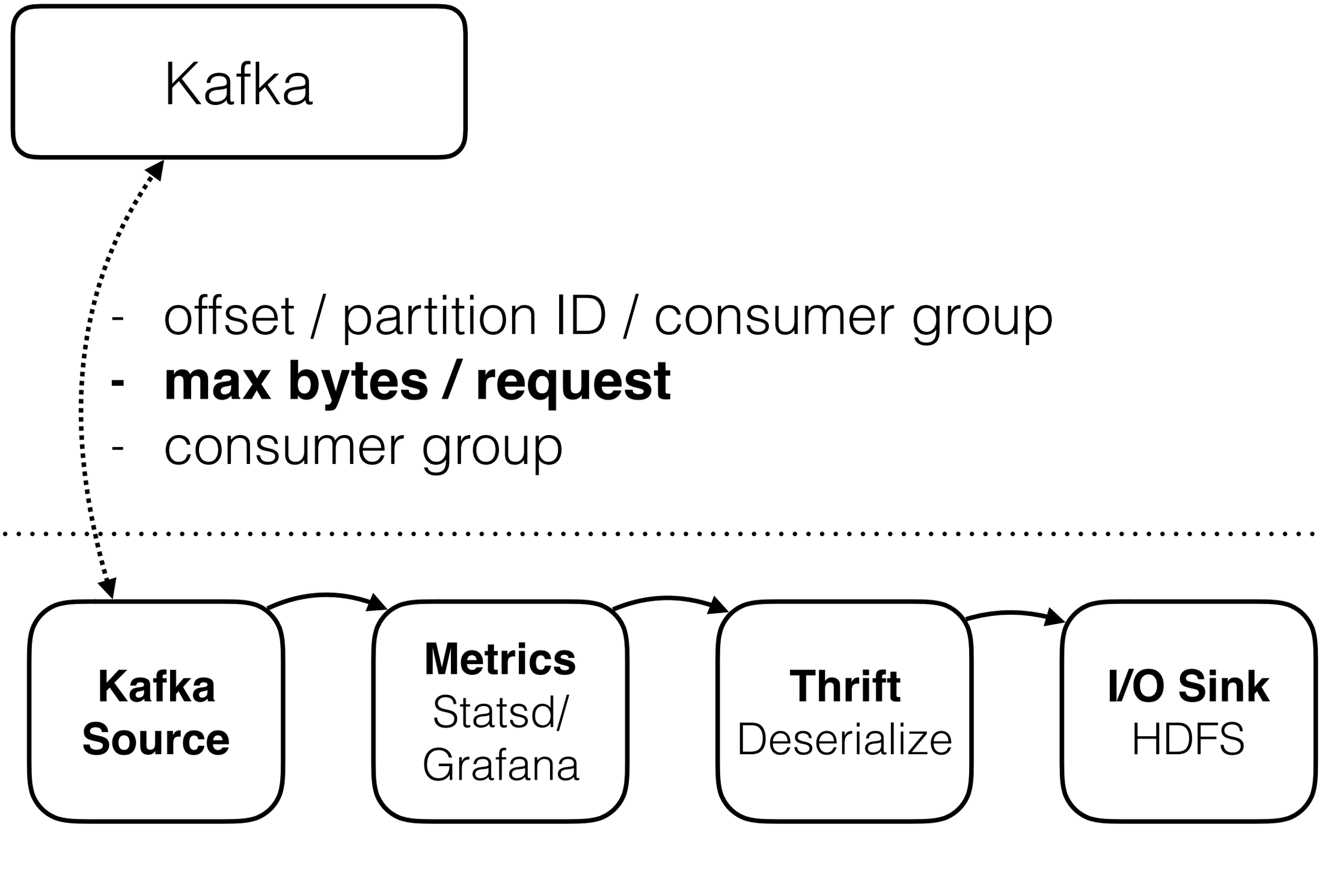
Kafka + Akka Reactive Streaming



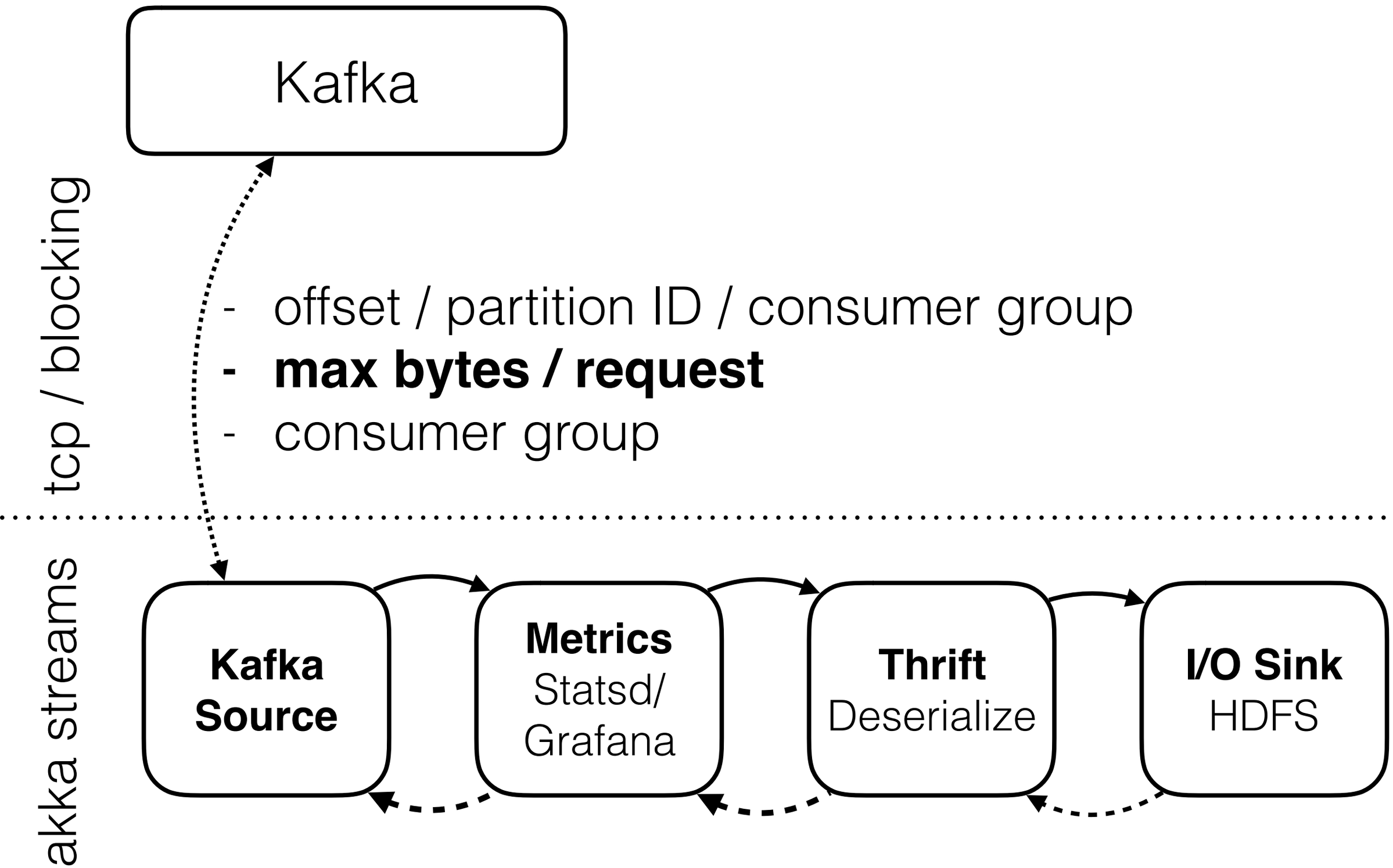
Kafka + Akka Reactive Streaming



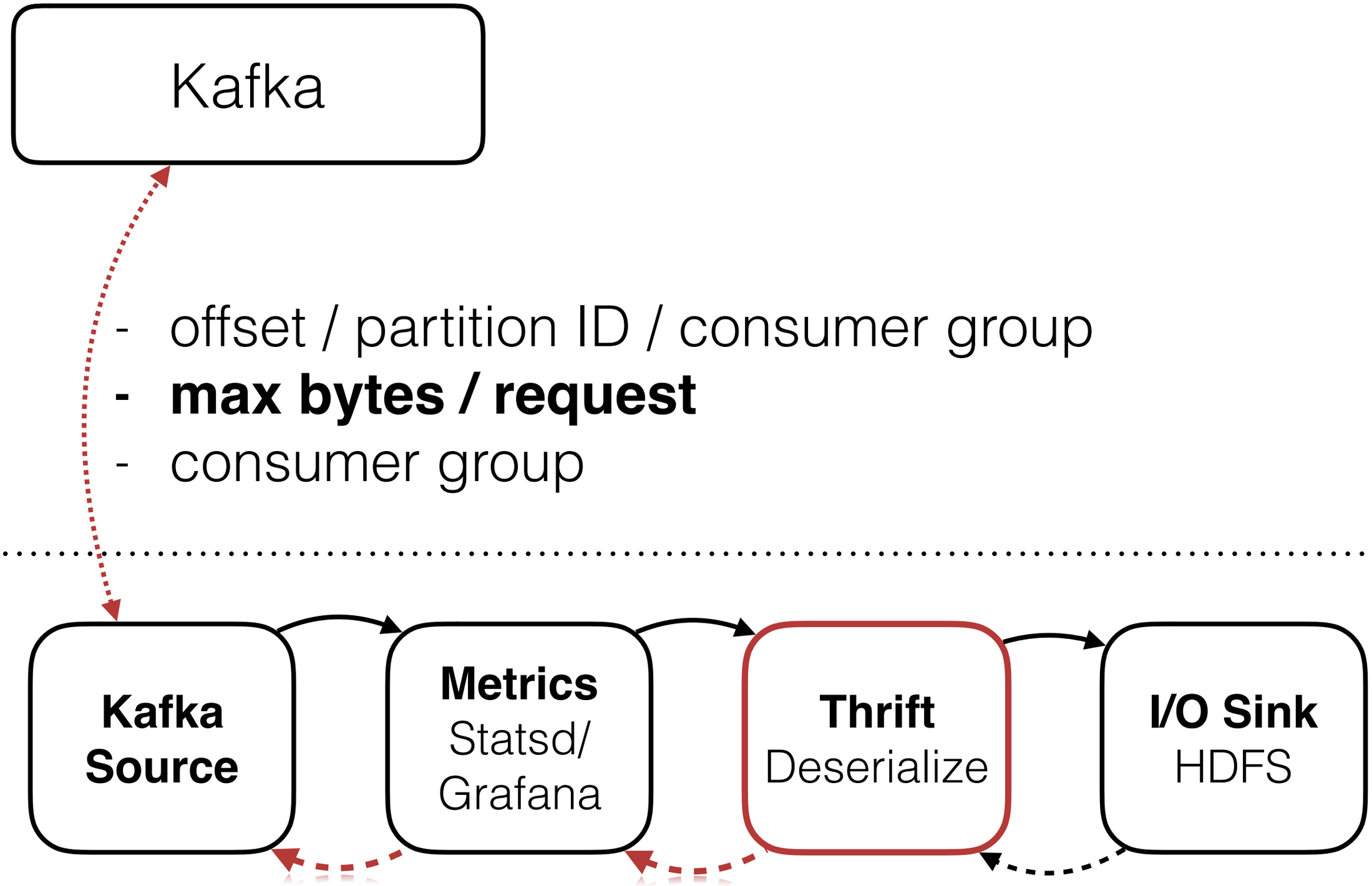
Kafka + Akka Reactive Streaming



Kafka + Akka Reactive Streaming



Kafka + Akka Reactive Streaming



Thank You!

Questions?

Links

http://flume.apache.org/
http://kafka.apache.org/
https://github.com/akka/reactive-kafka
http://doc.akka.io/docs/akka/2.4.7/scala/stream/index.html

Demo

https://github.com/sebastian-alfers/akka-streams-demo

```scala
/** ************************************************************
  * run the pipeline                                           *
  * ************************************************************/
val result =
  source
    .via(multiply)
    .via(printParallel)
    .runWith(Sink.ignore)
```

```scala
/** ************************************************************
  * multiply                                                   *
  * ************************************************************/
val multiply: Flow[Int, Int, Unit] =
  Flow[Int].map { item => item * item }
```

```scala
/** ************************************************************
  * run the pipeline                                           *
  * ************************************************************/
val result =
  source
    .via(multiply)
    .via(printParallel)
    .runWith(Sink.ignore)
```

```scala
/** ************************************************************
  * multiply                                                   *
  * ************************************************************/
val multiply: Flow[Int, Int, Unit] =
  Flow[Int].map { item => item * item }
```

```scala
/** ************************************************************
  * run the pipeline                                           *
  * ************************************************************/
val result =
  source
    .via(multiply)
    .via(printParallel)
    .runWith(Sink.ignore)
```

```scala
/** ************************************************************
  * print parallel                                             *
  * ************************************************************/
val parallelismFactor = 5
val printParallel: Flow[Int, Int, Unit] =
  Flow[Int].mapAsync(parallelismFactor) { item =>
    Future {
      if (item >= 40 && item <= 60) Thread.sleep(1000)
      print(s"$item")
      item
    }
  }
```

```scala
/** ***********************************************
  * actor that produces input                     *
  * ***********************************************/
class ActorSource extends ActorPublisher[Int] {

  var buffer: List[Int] = (0 until 1000).toList

  def onRequest(requestedAmount: Long) = {
    print(s"requested amount: $requestedAmount, total demant: $totalDemand")
    print(s"buffer before demand: ${buffer.size}")

    val (take, keep) = buffer.splitAt(requestedAmount.toInt)
    buffer = keep
    take.foreach { i =>
      onNext(i) //send out the data
    }

    print(s"buffer after demand: ${buffer.size}")

    if (buffer.isEmpty) {
      print("Stream is finished")
      onCompleteThenStop()
    }
  }

  //the api
  override def receive: Receive = {
    case Request(requestedAmount: Long) => onRequest(requestedAmount)
    case a: Any => print(s"got msg: ${a.getClass}")
  }
}
```

```scala
/** ************************************************
  * actor that produces input                      *
  * ************************************************/
class ActorSource extends ActorPublisher[Int] {

  var buffer: List[Int] = (0 until 1000).toList

  def onRequest(requestedAmount: Long) = {
    print(s"requested amount: $requestedAmount, total demant: $totalDemand")
    print(s"buffer before demand: ${buffer.size}")

    val (take, keep) = buffer.splitAt(requestedAmount.toInt)
    buffer = keep
    take.foreach { i =>
      onNext(i) //send out the data
    }

    print(s"buffer after demand: ${buffer.size}")

    if (buffer.isEmpty) {
      print("Stream is finished")
      onCompleteThenStop()
    }
  }

  //the api
  override def receive: Receive = {
    case Request(requestedAmount: Long) => onRequest(requestedAmount)
    case a: Any => print(s"got msg: ${a.getClass}")
  }
}
```

```scala
/** ************************************************
  * actor that produces input                      *
  * ************************************************/
class ActorSource extends ActorPublisher[Int] {

  var buffer: List[Int] = (0 until 1000).toList

  def onRequest(requestedAmount: Long) = {
    print(s"requested amount: $requestedAmount, total demant: $totalDemand")
    print(s"buffer before demand: ${buffer.size}")

    val (take, keep) = buffer.splitAt(requestedAmount.toInt)
    buffer = keep
    take.foreach { i =>
      onNext(i) //send out the data
    }

    print(s"buffer after demand: ${buffer.size}")

    if (buffer.isEmpty) {
      print("Stream is finished")
      onCompleteThenStop()
    }
  }

  //the api
  override def receive: Receive = {
    case Request(requestedAmount: Long) => onRequest(requestedAmount)
    case a: Any => print(s"got msg: ${a.getClass}")
  }
}
```

```scala
/** ********************************************************
  * actor that produces input                              *
  * ********************************************************/
class ActorSource extends ActorPublisher[Int] {

  var buffer: List[Int] = (0 until 1000).toList

  def onRequest(requestedAmount: Long) = {
    print(s"requested amount: $requestedAmount, total demant: $totalDemand")
    print(s"buffer before demand: ${buffer.size}")

    val (take, keep) = buffer.splitAt(requestedAmount.toInt)
    buffer = keep
    take.foreach { i =>
      onNext(i) //send out the data
    }

    print(s"buffer after demand: ${buffer.size}")

    if (buffer.isEmpty) {
      print("Stream is finished")
      onCompleteThenStop()
    }
  }

  //the api
  override def receive: Receive = {
    case Request(requestedAmount: Long) => onRequest(requestedAmount)
    case a: Any => print(s"got msg: ${a.getClass}")
  }
}
```