Akka Streams, Kafka, Kinesis

Peter Vandenabeele
Mechelen, June 25, 2015

#StreamProcessingBe

whoami : Peter Vandenabeele

@peter_v

@All_Things_Data (my consultancy)

current client:

Real Impact Analytics @RIAnalytics

Telecom Analytics (emerging markets)

Agenda

5’ Intro (Peter)

40’ Akka Streams, Kafka, Kinesis (Peter)

45’ Spark Streaming and Kafka Demo (Gerard)

15’ Open discussion (all)

30’ beers (doors close at 21:30)

Many thanks to

@All_Things_Data (beer)

@maasg (Gerard Maas)

you !

=> Note: always looking for locations

Agenda

Why ?

Akka + Akka Streams

Demo

Kafka + Kinesis

Demo

Why !

Why ?

distributed state

distributed failure

slow consumers

Akka

Why (for iHeartRadio)

Why we picked Akka cluster as the core architecture for our microservice

- Out-of-the-box clustering infrastructure
- Loose coupling without the cost of JSON parsing.
- Transparent programming model within and across microservices.
- Strong community and commercial support
- High resiliency, performance and scalability.

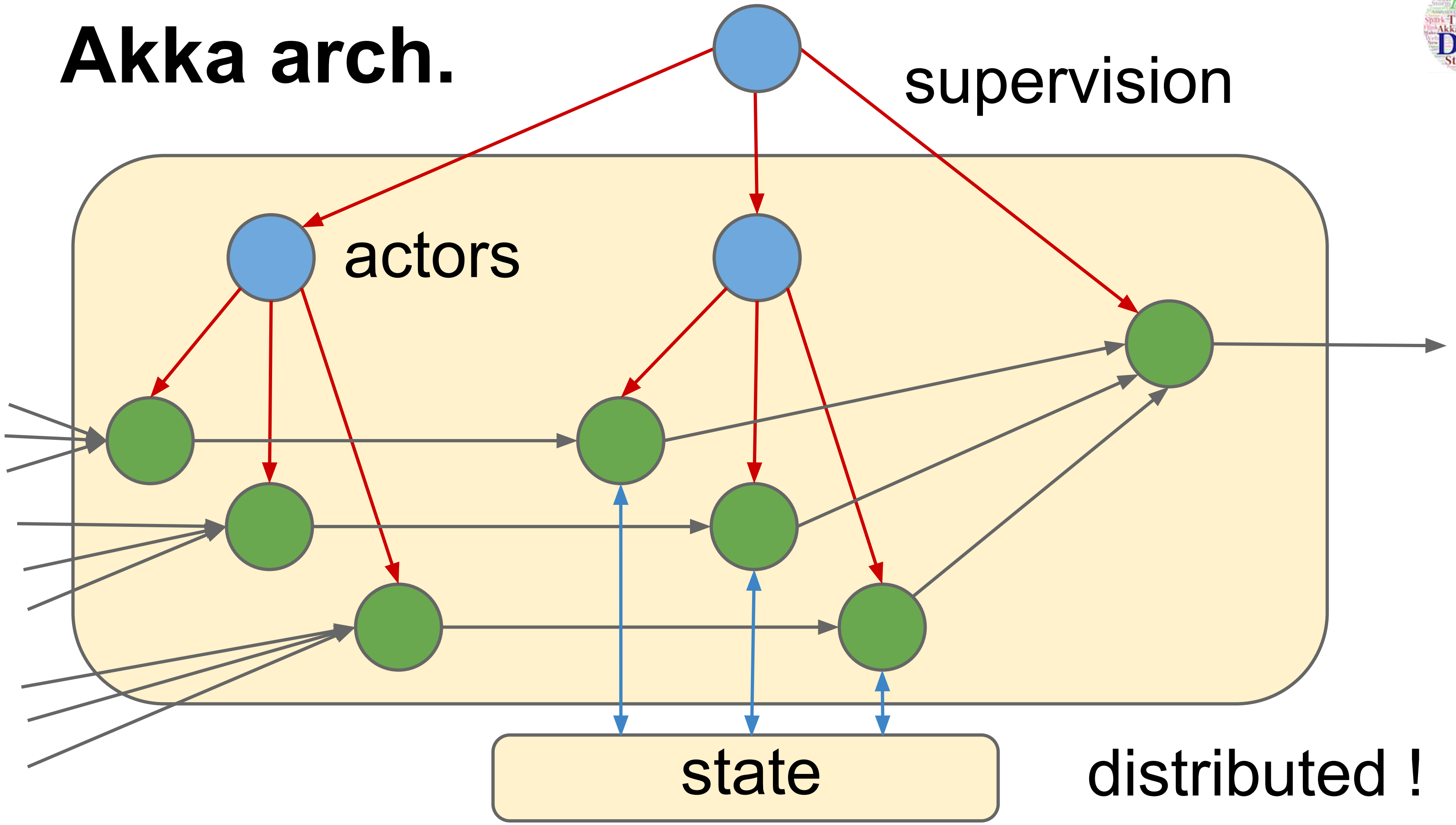
source: tech.iheart.com/post/121599571574/why-we-picked-akka-cluster-as-our-microservice

Akka design

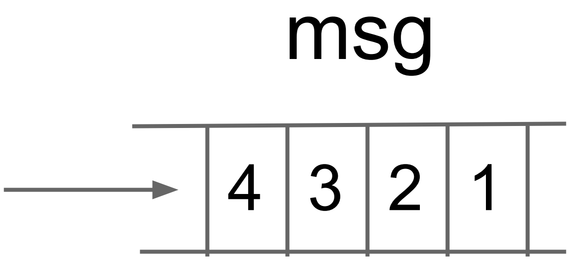
Building

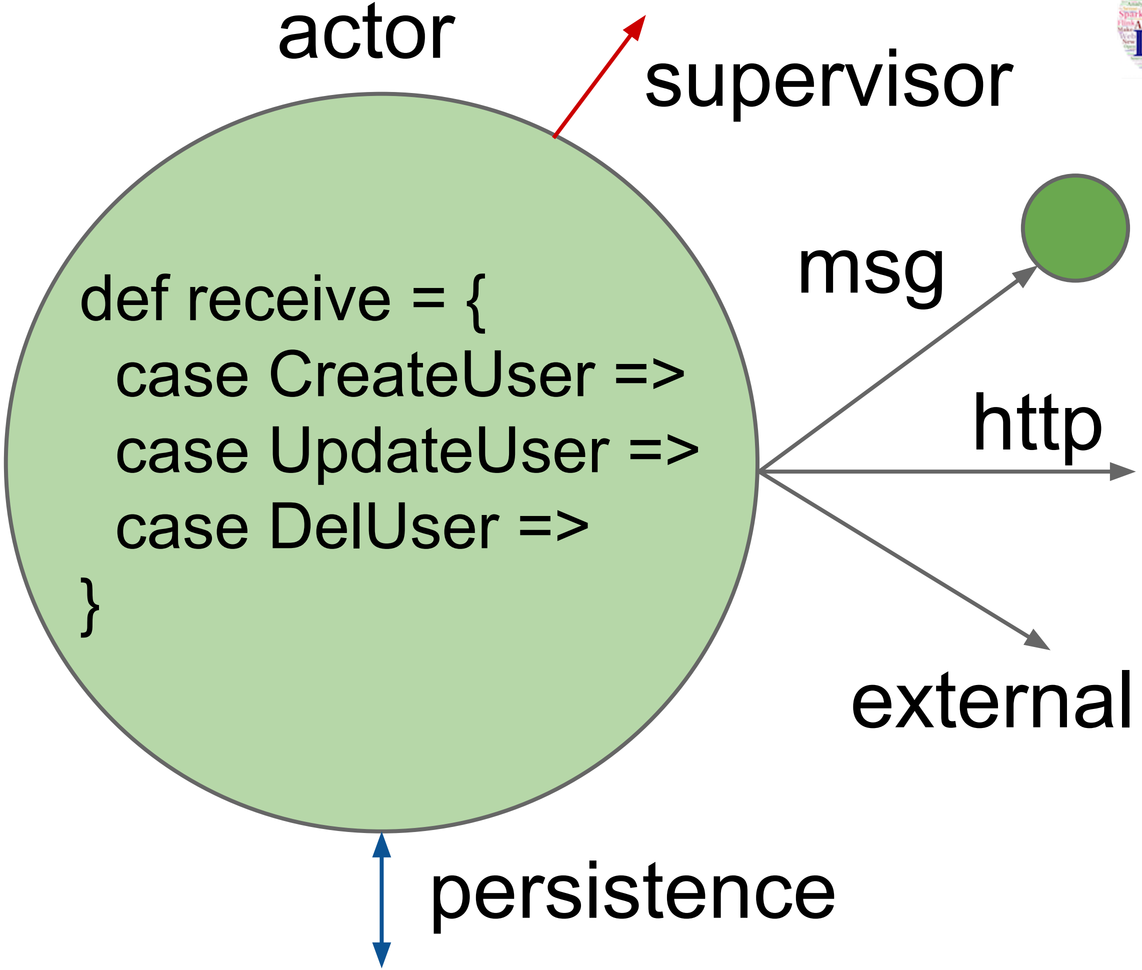
- concurrent <= many (slow) CPU’s
- distributed <= distributed state
- resilient <= distributed failure
- applications <= platform
- on JVM <= Erlang OTP

Akka arch.



Akka actor





- msgs are *sent*
- recvd in order
- single thread
- stateful !
- errors go “up”

Akka usage + courses

- concurrent programming *not* easy …
- but *without* Akka … would be much harder
- Spark (see log extract next slide)
- Flink (version 0.9 of 24 June)
- local projects (e.g.”Wegen en verkeer”)
- BeScala Meetup now runs Akka intro course
- commercial courses (Cronos, Scala World...)

Spark heavily based on Akka

log extract from Spark:

java -cp ...

"org.apache.spark.executor.CoarseGrainedExecutorBackend"

"--driver-url" "akka.tcp://sparkDriver@docker-master:51810/user/CoarseGrainedScheduler"

"--executor-id" "23" "--hostname" "docker-slave2" "--cores" "8"

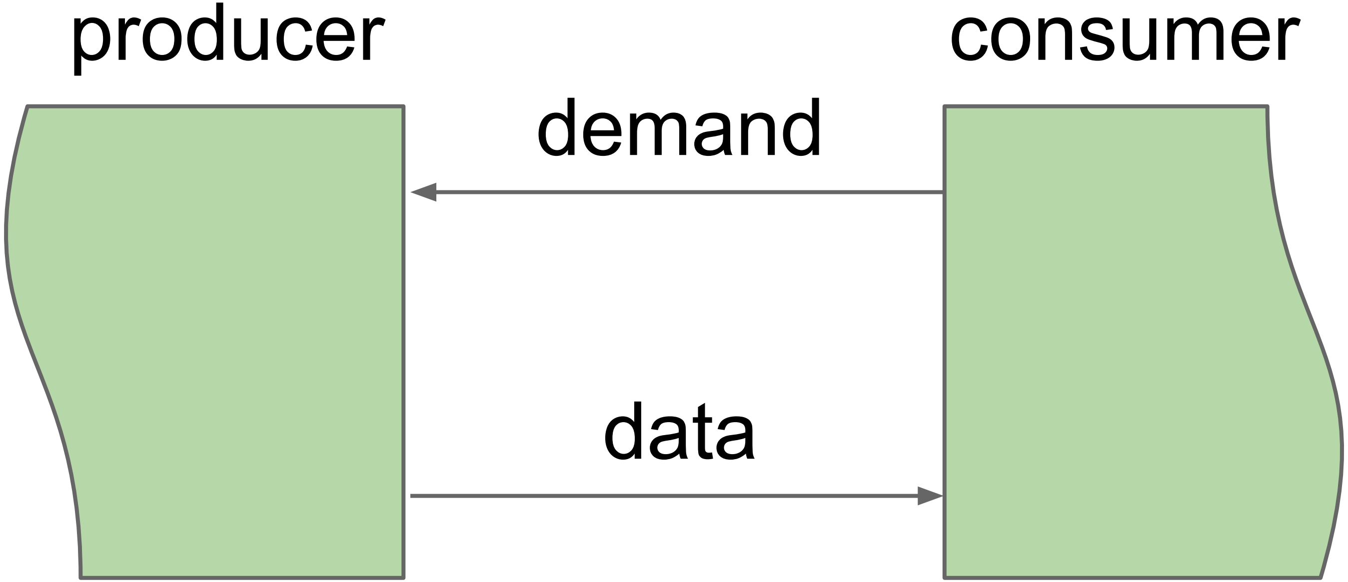
"--worker-url" "akka.tcp://sparkWorker@docker-slave2:50268/user/Worker"

Akka Streams

Reactive Streams

- http://reactive-streams.org
- exchange of *stream data* across *asynchronous* boundary in *bounded fashion*
- building and industry standard (open IP)

Demand based



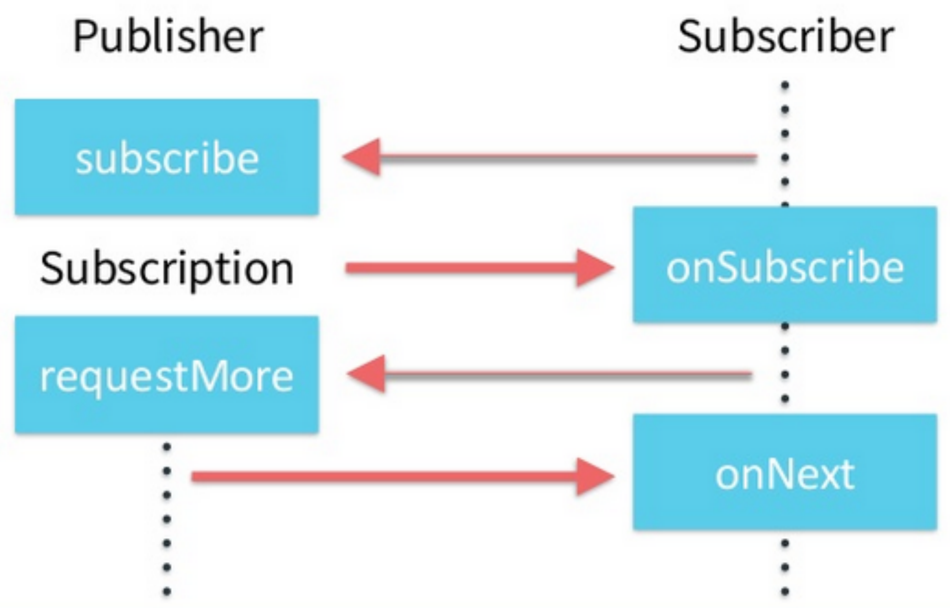
“give me max 20”

“sending 2, 5, 10, ...”

“give me max 10 more”

How does it work?

Don’t try this at home

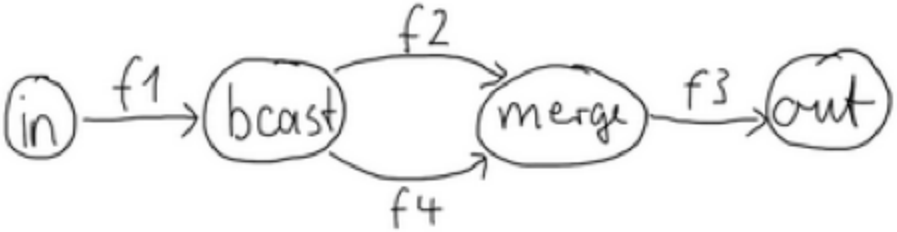



```scala
trait Publisher[T] {
  def subscribe(sub: Subscriber[T])
}
trait Subscription {
  def requestMore(n: Int): Unit
  def cancel(): Unit
}
trait Subscriber[T] {
  def onSubscribe(s: Subscription):
  def onNext(elem: T): Unit
  def onError(thr: Throwable): Unit
  def onComplete(): Unit
}
```

source: http://www.slideshare.net/rolandkuhn/reactive-streams Roland Kuhn (TypeSafe) @rolandkuhn

Akka Streams

- Source ~> Flow ~> Flow ~> Sink
- MaterializedFlow



source: http://www.slideshare.net/rolandkuhn/reactive-streams Roland Kuhn (TypeSafe) @rolandkuhn

Akka Streams : advantages

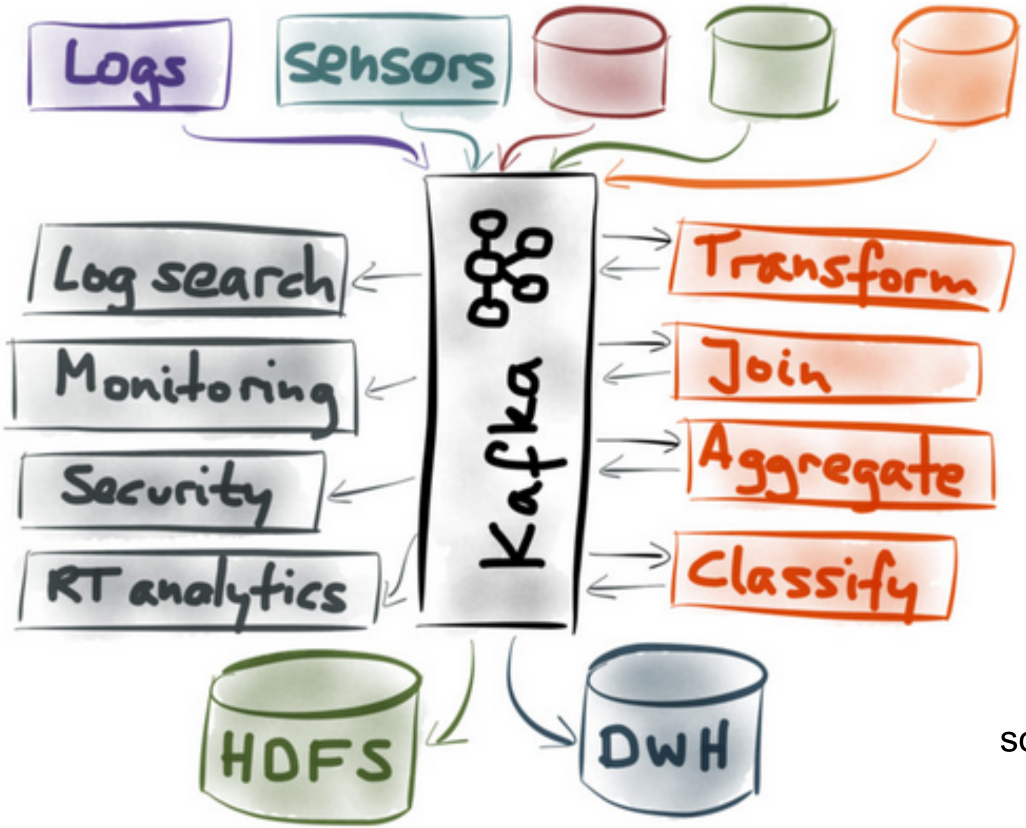
- Types (stream of T)
- makes it trivially simple :-)
- Many examples online (fast and simple)
- demo of simplistic case

simplistic Akka Streams demo

```scala
import akka.actor.ActorSystem
import akka.stream.ActorFlowMaterializer
import akka.stream.scaladsl._

object Words {

  implicit val system = ActorSystem("System")
  implicit val materializer = ActorFlowMaterializer()
  import system.dispatcher

  def processWords(sinkAction: (String => Unit)): Unit = {

    val text =
      """|Lorem Ipsum is simply text
         |Lorem Ipsum has been the industry standard dummy text ever since the 1500s
         |when an unknown printer took a galley of type and scrambled it to make a type
         |specimen book""".stripMargin

    Source(() => text.split("\\s").iterator)
      .map(_.toUpperCase)
      .filter(line => line.length > 3)
      .runForeach(sinkAction)
      .onComplete(_ => system.shutdown())


//    in ~> f1 ~> bcast ~> f2 ~> merge ~> f3 ~> out
//                bcast ~> f4 ~> merge
  }
}
```

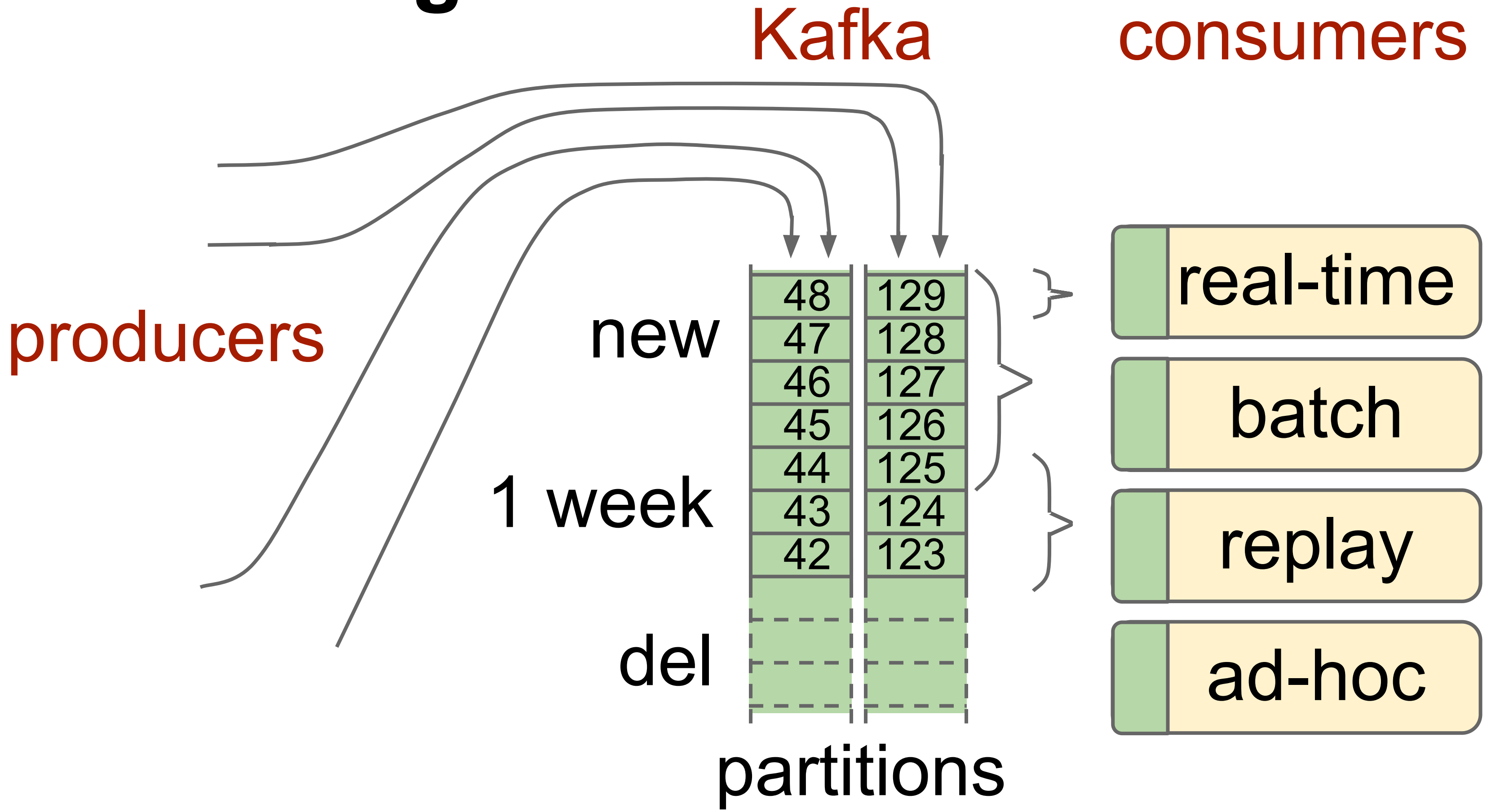
Kafka

Kafka (LinkedIn) : Martin Kleppmann

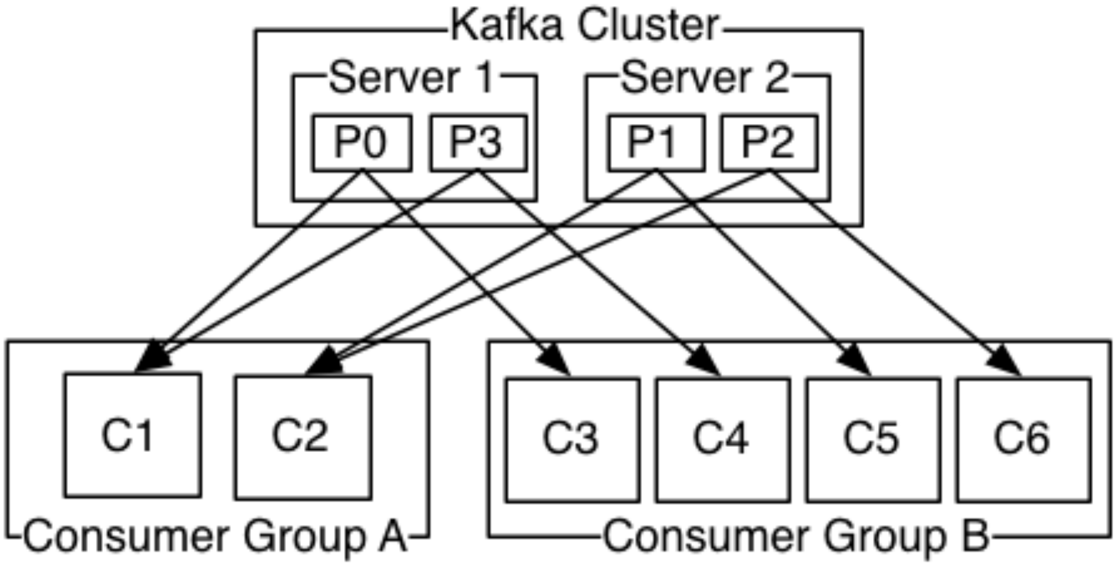


source : Martin Kleppmann
at strata Hadoop London

Kafka log based



Kafka partitions



A two server Kafka cluster hosting four partitions (P0-P3) with two consumer groups. Consumer group A has two consumer instances and group B has four.

source: http://kafka.apache.org/documentation.html

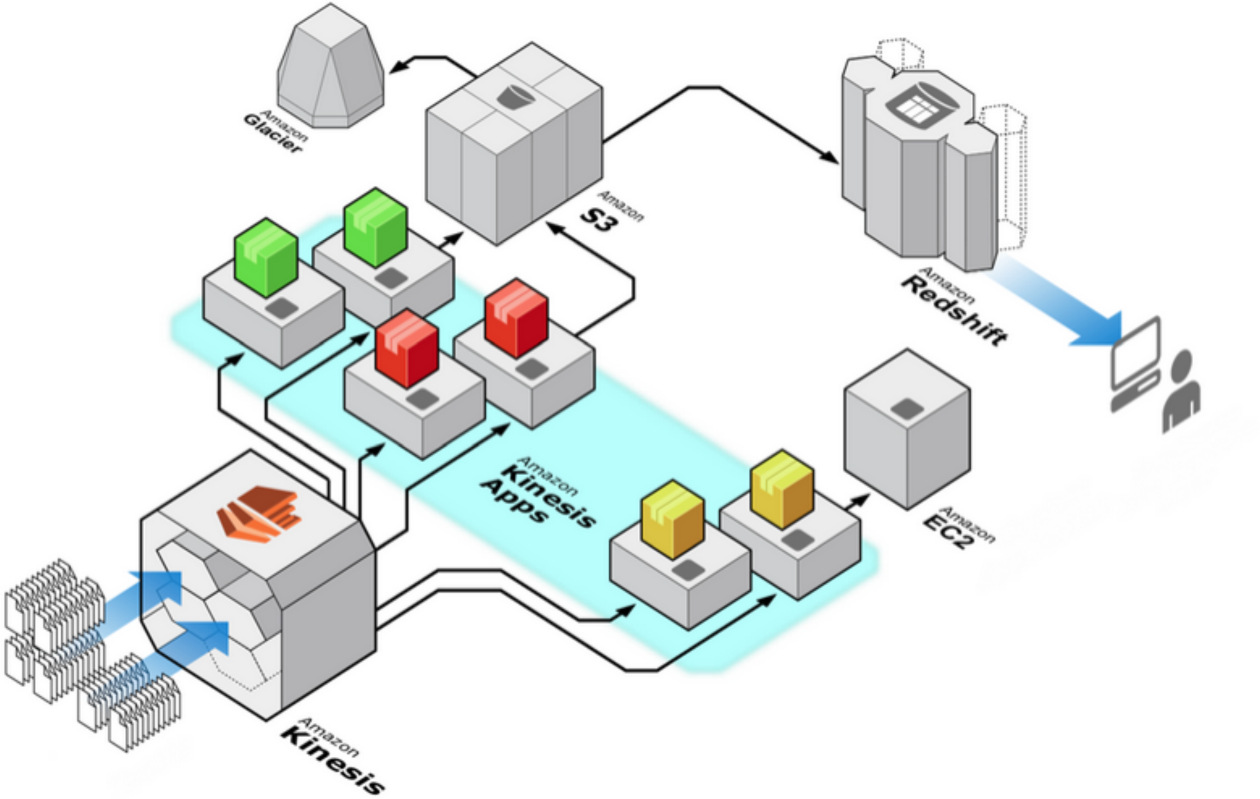
Kafka (LinkedIn) : Jay Kreps

- 175 TB of in-flight log data per colo
- Low-latency: ~1.5 ms
- Replicated to each datacenter
- Tens of thousands of data producers
- Thousands of consumers
- 7 million messages written/sec
- 35 million messages read/sec
- Hadoop integration

source: Jay Kreps
on slideshare
“I ♥ Log”
Real-time Data and Apache Kafka

Kinesis

Kinesis : Kafka as a Service



source: http://aws.amazon.com/kinesis/details/

Kinesis design

- Fully (auto-)managed
- Strong durability guarantees
- Stream (= topic)
- Shard (= partition)
- “fast” writers (but … round-trip 20 ms ?)
- “slow” readers (max 5/s per shard ??)
- Kinesis Client Library (java)

Kinesis limitations ...

- writing latency (20 ms per entry - replicated)
- 24 hours data retention
- 5 reads per second

https://brandur.org/kinesis-in-production

- “vanishing history” after shard split
- “if I’d understood the consequences ... earlier, I probably would have pushed harder for Kafka”

simplistic Kinesis demo

```scala
import com.amazonaws.services.kinesis.AmazonKinesisClient
import com.amazonaws.services.kinesis.model.{PutRecordResult, PutRecordRequest}

class KinesisProducer(streamName: String) {

  val kinesisClient = new AmazonKinesisClient()

  def sendMessage(msg: String): String = {
    val putRequest: PutRecordRequest = new PutRecordRequest()
    putRequest.setStreamName(streamName)
    putRequest.setData(ByteBuffer.wrap(jsonMessage(msg).getBytes("UTF-8")))
    putRequest.setPartitionKey("single_partition")

    val putRecordResult: PutRecordResult = kinesisClient.putRecord(putRequest)
    putRecordResult.getSequenceNumber
  }

  // The demo Kinesis reader needs this resource + referrer format
  private def jsonMessage(referrer: String):String = {...}
}
```

Kinesis consumer with Amazom DynamoDB :: reused from http://docs.aws.amazon.com/kinesis/latest/dev/kinesis-sample-application.html

Why !

(a *personal* view)

Note: “thanks for the feedback on this section. Indeed Kafka and Akka serve very different purposes, but they both offer solutions for distributed state, distributed failure and slow consumers”

Problem 1: Distributed state

Akka

=> state encapsulated in Actors

=> exchange self-contained messages

Kafka

=> immutable, ordered update queue (Kappa)

Problem 2: Distributed failure

Akka

=> explicit failure management (supervisor)

Kafka

=> partitions are replicated over brokers

=> consumers can replay from log

Problem 3: Slow consumers

Akka Streams

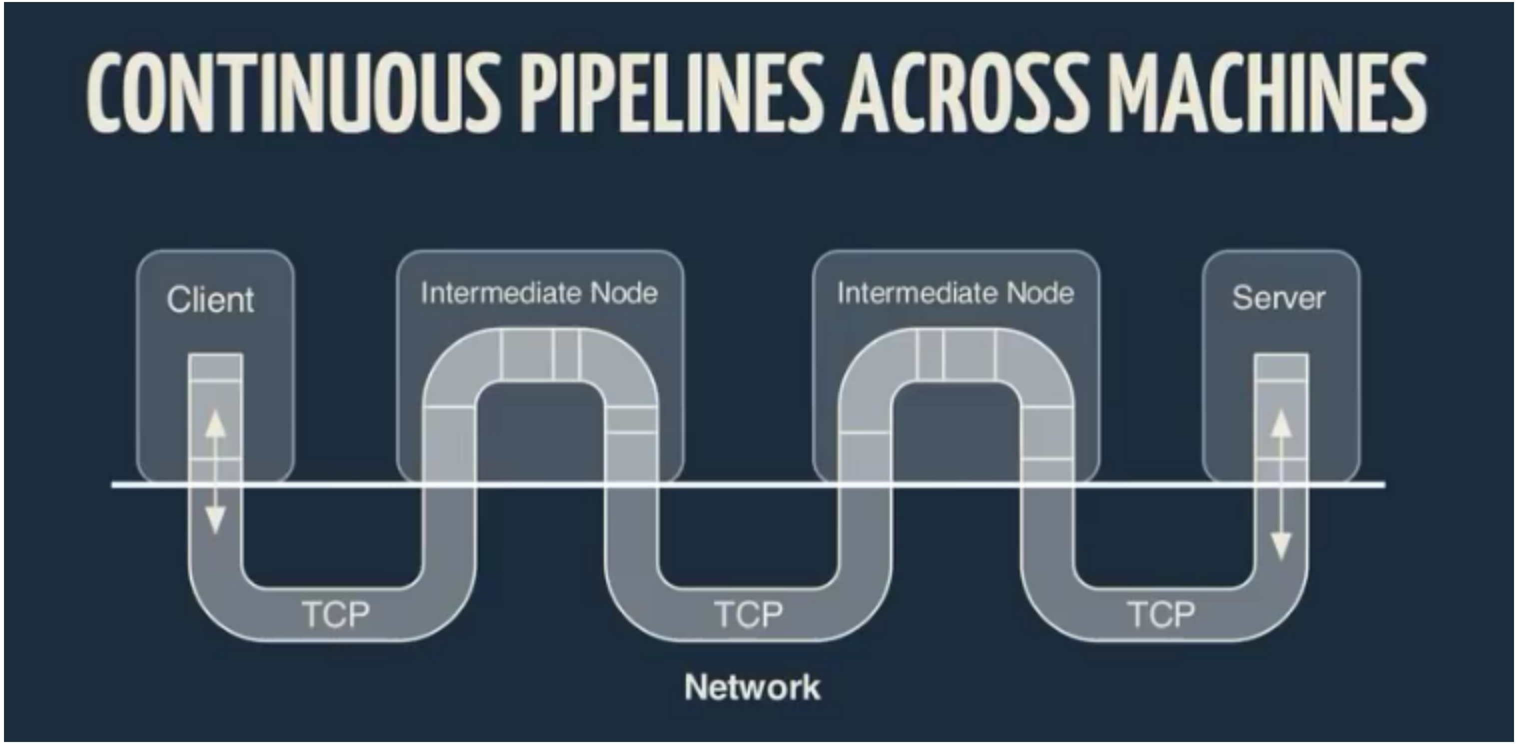
=> automatic back-pressure (avoid overflow)

Kafka

=> consumers fully decoupled

=> keeps data for 1 week ! (Kinesis: 1 day)

Avoid overflow in Akka: “tight seals”



source : https://www.youtube.com/watch?v=o5PUDI4qi10 by @sirthias

Avoid overflow in Kafka: “big lake”