Streaming Data Microservices

With Akka Streams and Kafka Streams

Dean Wampler, Ph.D.
dean@lightbend.com
@deanwampler

Lightbend

Who am I?

I lead the Lightbend Fast Data Platform project; streaming data and microservices

lightbend.com/fast-data-platform

Free as in 🍺

New second edition!

lbnd.io/fast-data-book

Streaming architectures (from the report)

Kubernetes, Mesos, YARN, ...
Cloud or on-premise
10
Reactive Platform
Go
Node.js
...
Microservices
3
ZK
ZooKeeper Cluster
4
REST
2
Sockets
1
Files
Broker
Kafka Cluster
5
6
7
8
9
Spark
Flink
Beam
Akka Streams
Kafka Streams
Low Latency
Spark
Beam
Mini-batch
Spark
...
Batch
Disk
HDFS
S3, ...
Search
SQL/
NoSQL
Persistence
Events
Streams
Storage

This talk's focus:

- Kafka - the data backplane
- Akka Streams and Kafka Streams - the engine for streaming data microservices

Why Kafka?

Reactive Platform
Go
Node.js
...
Microservices
3
ZK
ZooKeeper Cluster
4
1
REST
2
6
Sockets
1
Broker
Kafka Cluster
7
5
8
Files
Disk
S3, ...
HDFS
Events
Streams

Organized into topics

Topics are partitioned, replicated, and distributed

Logs, *not* queues!

M Producers

N Consumers, who start reading where they want

Unlike queues, consumers don't delete entries; Kafka manages their lifecycles

Architectural Benefits of Kafka

Before:

Producers N * M links Consumers

Kafka for Connectivity

Kafka for Connectivity

- Simplify dependencies
- Resilient against data loss
- M producers, N consumers
- Simplicity of one “API” for communication

Streaming Engines

Spark, Flink - services to which you submit work. Large scale, automatic data partitioning.

Beam - similar. Google's project that has been instrumental in defining streaming semantics.

They do a *lot* (Spark example)

They do a *lot* (Spark example)

Streaming Engines

Akka Streams, Kafka Streams - libraries for “data-centric microservices”. Smaller scale, but great flexibility

Microservice All the Things!!

A Spectrum of Microservices

Event-driven μ-services

"Record-centric" μ-services

Events

← Data Spectrum →

Records

A Spectrum of Microservices

- Each datum has an identity
- Process each one uniquely
- Think *sessions* and *state machines*

A Spectrum of Microservices

Event-driven μ-services

"Record-centric" μ-services

Events

← Data Spectrum →

Records

A Spectrum of Microservices

- “Anonymous” records
- Process *en masse*
- Think SQL queries for analytics

A Spectrum of Microservices

Event-driven μ-services

"Record-centric" μ-services

Events

← Data Spectrum →

Records

A Spectrum of Microservices

akka Streams

Event-driven μ-services

Akka emerged from the left-hand side of the spectrum, the world of highly *Reactive* microservices.

Akka Streams pushes to the right, more data-centric.

https://www.reactivemanifesto.org/

Events

← Data Spectrum →

Records

A Spectrum of Microservices

Streams

Emerged from the right-hand side.

Kafka Streams pushes to the left, supporting many event-processing scenarios.

"Record-centric" μ-services

Events

← Data Spectrum →

Records

Kafka Streams

Kafka Streams

- Important stream-processing semantics, e.g.,
  - Windowing support (e.g., group by within a window)

See my O'Reilly report for details.

Kafka Streams

- Important stream-processing semantics, e.g.,
  - Distinguish between *event time* and *processing time*

Kafka Streams

- *Exactly once*
- vs.
  - *At most once* - “fire and forget”
  - *At least once* - keep trying until acknowledged (but you have to handle de-duplication)
- Note: it’s really *effectively once*

Kafka Streams

- KStream
  - Per record transformations, one to one mapping
- KTable
  - Last value per key
  - Useful for holding *state* values

Kafka Streams

- Read from and write to Kafka topics, memory
  - Kafka Connect used for other sources and sinks
- Load balance and scale using topic partitioning

Kafka Streams

- Java API
- Scala API: written by Lightbend
- SQL!!

Kafka Streams Example

Kafka Streams Example

We'll use the new Scala Kafka Streams API:

- Developed by Debasish Ghosh, Boris Lublinsky, Sean Glover, et al. from Lightbend
- See also the following, if you know about *queryable state*:
  - https://github.com/lightbend/kafka-streams-query

See github.com/lightbend/
kafka-with-akka-streams-kafka-streams-tutorial

```scala
val builder = new StreamsBuilderS // New Scala Wrapper API.

val data  = builder.stream[Array[Byte], Array[Byte]](rawDataTopic)
val model = builder.stream[Array[Byte], Array[Byte]](modelTopic)
val modelProcessor = new ModelProcessor
val scorer = new Scorer(modelProcessor) // scorer.score(record) used

model.mapValues(bytes => Model.parseBytes(bytes)) // array => record
     .filter((key, model) => model.valid) // Successful?
     .mapValues(model => ModelImpl.findModel(model))
     .process(() => modelProcessor, …) // Set up actual model
data.mapValues(bytes => DataRecord.parseBytes(bytes))
     .filter((key, record) => record.valid)
     .mapValues(record => new ScoredRecord(scorer.score(record),record))
     .to(scoredRecordsTopic)

val streams = new KafkaStreams(
  builder.build, streamsConfiguration)
streams.start()
sys.addShutdownHook(streams.close())
```

```scala
val builder = new StreamsBuilderS // New Scala Wrapper API.

val data  = builder.stream[Array[Byte], Array[Byte]](rawDataTopic)
val model = builder.stream[Array[Byte], Array[Byte]](modelTopic)
val modelProcessor = new ModelProcessor
val scorer = new Scorer(modelProcessor) // scorer.score(record) used

model.mapValues(bytes => Model.parseBytes(bytes)) // array => record
    .filter((key, model) => model.valid) // Successful?
    .mapValues(model => ModelImpl.findModel(model))
    .process(() => modelProcessor, …) // Set up actual model
data.mapValues(bytes => DataRecord.parseBytes(bytes))
    .filter((key, record) => record.valid)
    .mapValues(record => new ScoredRecord(scorer.score(record),record))
    .to(scoredRecordsTopic)

val streams = new KafkaStreams(
  builder.build, streamsConfiguration)
streams.start()
sys.addShutdownHook(streams.close())
```

```scala
val builder = new StreamsBuilderS // New Scala Wrapper API.

val data  = builder.stream[Array[Byte], Array[Byte]](rawDataTopic)
val model = builder.stream[Array[Byte], Array[Byte]](modelTopic)
val modelProcessor = new ModelProcessor
val scorer = new Scorer(modelProcessor) // scorer.score(record) used

model.mapValues(bytes => Model.parseBytes(bytes)) // array => record
    .filter((key, model) => model.valid) // Successful?
    .mapValues(model => ModelImpl.findModel(model))
    .process(() => modelProcessor, …) // Set up actual model
data.mapValues(bytes => DataRecord.parseBytes(bytes))
    .filter((key, record) => record.valid)
    .mapValues(record => new ScoredRecord(scorer.score(record),record))
    .to(scoredRecordsTopic)

val streams = new KafkaStreams(
  builder.build, streamsConfiguration)
streams.start()
sys.addShutdownHook(streams.close())
```

```scala
val builder = new StreamsBuilderS // New Scala Wrapper API.

val data  = builder.stream[Array[Byte], Array[Byte]](rawDataTopic)
val model = builder.stream[Array[Byte], Array[Byte]](modelTopic)
val modelProcessor = new ModelProcessor
val scorer = new Scorer(modelProcessor) // scorer.score(record) used

model.mapValues(bytes => Model.parseBytes(bytes)) // array => record
     .filter((key, model) => model.valid) // Successful?
     .mapValues(model => ModelImpl.findModel(model))
     .process(() => modelProcessor, …) // Set up actual model
data.mapValues(bytes => DataRecord.parseBytes(bytes))
     .filter((key, record) => record.valid)
     .mapValues(record => new ScoredRecord(scorer.score(record),record))
     .to(scoredRecordsTopic)

val streams = new KafkaStreams(
  builder.build, streamsConfiguration)
streams.start()
sys.addShutdownHook(streams.close())
```

```scala
val builder = new StreamsBuilderS // New Scala Wrapper API.

val data  = builder.stream[Array[Byte], Array[Byte]](rawDataTopic)
val model = builder.stream[Array[Byte], Array[Byte]](modelTopic)
val modelProcessor = new ModelProcessor
val scorer = new Scorer(modelProcessor) // scorer.score(record) used

model.mapValues(bytes => Model.parseBytes(bytes)) // array => record
    .filter((key, model) => model.valid) // Successful?
    .mapValues(model => ModelImpl.findModel(model))
    .process(() => modelProcessor, …) // Set up actual model
data.mapValues(bytes => DataRecord.parseBytes(bytes))
    .filter((key, record) => record.valid)
    .mapValues(record => new ScoredRecord(scorer.score(record),record))
    .to(scoredRecordsTopic)

val streams = new KafkaStreams(
  builder.build, streamsConfiguration)
streams.start()
sys.addShutdownHook(streams.close())
```

```scala
val builder = new StreamsBuilderS // New Scala Wrapper API.

val data  = builder.stream[Array[Byte], Array[Byte]](rawDataTopic)
val model = builder.stream[Array[Byte], Array[Byte]](modelTopic)
val modelProcessor = new ModelProcessor
val scorer = new Scorer(modelProcessor) // scorer.score(record) used

model.mapValues(bytes => Model.parseBytes(bytes)) // array => record
    .filter((key, model) => model.valid) // Successful?
    .mapValues(model => ModelImpl.findModel(model))
    .process(() => modelProcessor, …) // Set up actual model
data.mapValues(bytes => DataRecord.parseBytes(bytes))
    .filter((key, record) => record.valid)
    .mapValues(record => new ScoredRecord(scorer.score(record),record))
    .to(scoredRecordsTopic)

val streams = new KafkaStreams(
  builder.build, streamsConfiguration)
streams.start()
sys.addShutdownHook(streams.close())
```

```scala
val builder = new StreamsBuilderS // New Scala Wrapper API.

val data  = builder.stream[Array[Byte], Array[Byte]](rawDataTopic)
val model = builder.stream[Array[Byte], Array[Byte]](modelTopic)
val modelProcessor = new ModelProcessor
val scorer = new Scorer(modelProcessor) // scorer.score(record) used

model.mapValues(bytes => Model.parseBytes(bytes)) // array => record
    .filter((key, model) => model.valid) // Successful?
    .mapValues(model => ModelImpl.findModel(model))
    .process(() => modelProcessor, …) // Set up actual model
data.mapValues(bytes => DataRecord.parseBytes(bytes))
    .filter((key, record) => record.valid)
    .mapValues(record => new ScoredRecord(scorer.score(record),record))
    .to(scoredRecordsTopic)

val streams = new KafkaStreams(
  builder.build, streamsConfiguration)
streams.start()
sys.addShutdownHook(streams.close())
```

```scala
val builder = new StreamsBuilderS // New Scala Wrapper API.

val data  = builder.stream[Array[Byte], Array[Byte]](rawDataTopic)
val model = builder.stream[Array[Byte], Array[Byte]](modelTopic)
val modelProcessor = new ModelProcessor
val scorer = new Scorer(modelProcessor) // scorer.score(record) used

model.mapValues(bytes => Model.parseBytes(bytes)) // array => record
    .filter((key, model) => model.valid) // Successful?
    .mapValues(model => ModelImpl.findModel(model))
    .process(() => modelProcessor, …) // Set up actual model
data.mapValues(bytes => DataRecord.parseBytes(bytes))
    .filter((key, record) => record.valid)
    .mapValues(record => new ScoredRecord(scorer.score(record),record))
    .to(scoredRecordsTopic)

val streams = new KafkaStreams(
  builder.build, streamsConfiguration)
streams.start()
sys.addShutdownHook(streams.close())
```

What’s Missing?

The rest of the microservice tools you need…

Embed your Kafka Streams code in microservices written with Akka or other JVM tools.

Akka Streams

Akka Streams

- Implements *Reactive Streams*
  - http://www.reactive-streams.org/
- *Back pressure* for flow control

Akka Streams

Akka Streams

Back pressure composes!

Akka Streams

- Part of the Akka ecosystem
  - Akka Actors, Akka Cluster, Akka HTTP, Akka Persistence, …
  - Alpakka - rich connection library
- Optimized for low overhead and latency

Akka Streams

- The “gist” - calculate factorials:

```
val source: Source[Int, NotUsed] = Source(1 to 10)
val factorials = source.scan(BigInt(1)) (
                    (total, next) => total * next )
factorials.runWith(Sink.foreach(println))
```

```
1
2
6
24
120
720
5040
40320
362880
3628800
```

Akka Streams

- The “gist” - calculate factorials:

```
val source: Source[Int, NotUsed] = Source(1 to 10)
val factorials = source.scan(BigInt(1)) (
                    (total, next) => total * next )
factorials.runWith(Sink.foreach(println))
```

```
1
2
6
24
120
720
5040
40320
362880
3628800
```

Akka Streams Example

Akka Streams Example - “Direct” Variation

```scala
implicit val system = ActorSystem("ModelServing")
implicit val materializer = ActorMaterializer()
implicit val executionContext = system.dispatcher

val modelProcessor = new ModelProcessor // Same as KS example
val scorer = new Scorer(modelProcessor) // Same as KS example
val modelScoringStage = new ModelScoringStage(scorer)// AS custom “stage”

val dataStream: Source[Record, Consumer.Control] =
  Consumer.atMostOnceSource(dataConsumerSettings,
      Subscriptions.topics(rawDataTopic))
    .map(input => DataRecord.parseBytes(input.value()))
    .collect{ case Success(data) => data }

val modelStream: Source[ModelImpl, Consumer.Control] =
  Consumer.atMostOnceSource(modelConsumerSettings,
      Subscriptions.topics(modelTopic))
    .map(input => Model.parseBytes(input.value()))
    .collect{ case Success(mod) => mod }
    .map(model => ModelImpl.findModel(model))
    .collect{ case Success(modImpl) => modImpl }
```

```scala
implicit val system = ActorSystem("ModelServing")
implicit val materializer = ActorMaterializer()
implicit val executionContext = system.dispatcher

val modelProcessor = new ModelProcessor // Same as KS example
val scorer = new Scorer(modelProcessor) // Same as KS example
val modelScoringStage = new ModelScoringStage(scorer)// AS custom “stage”

val dataStream: Source[Record, Consumer.Control] =
  Consumer.atMostOnceSource(dataConsumerSettings,
      Subscriptions.topics(rawDataTopic))
    .map(input => DataRecord.parseBytes(input.value()))
    .collect{ case Success(data) => data }

val modelStream: Source[ModelImpl, Consumer.Control] =
  Consumer.atMostOnceSource(modelConsumerSettings,
      Subscriptions.topics(modelTopic))
    .map(input => Model.parseBytes(input.value()))
    .collect{ case Success(mod) => mod }
    .map(model => ModelImpl.findModel(model))
    .collect{ case Success(modImpl) => modImpl }
```

```scala
implicit val system = ActorSystem("ModelServing")
implicit val materializer = ActorMaterializer()
implicit val executionContext = system.dispatcher

val modelProcessor = new ModelProcessor // Same as KS example
val scorer = new Scorer(modelProcessor) // Same as KS example
val modelScoringStage = new ModelScoringStage(scorer)// AS custom “stage”

val dataStream: Source[Record, Consumer.Control] =
  Consumer.atMostOnceSource(dataConsumerSettings,
      Subscriptions.topics(rawDataTopic))
    .map(input => DataRecord.parseBytes(input.value()))
    .collect{ case Success(data) => data }

val modelStream: Source[ModelImpl, Consumer.Control] =
  Consumer.atMostOnceSource(modelConsumerSettings,
      Subscriptions.topics(modelTopic))
    .map(input => Model.parseBytes(input.value()))
    .collect{ case Success(mod) => mod }
    .map(model => ModelImpl.findModel(model))
    .collect{ case Success(modImpl) => modImpl }
```

```scala
implicit val system = ActorSystem("ModelServing")
implicit val materializer = ActorMaterializer()
implicit val executionContext = system.dispatcher

val modelProcessor = new ModelProcessor // Same as KS example
val scorer = new Scorer(modelProcessor) // Same as KS example
val modelScoringStage = new ModelScoringStage(scorer)// AS custom “stage”

val dataStream: Source[Record, Consumer.Control] =
  Consumer.atMostOnceSource(dataConsumerSettings,
      Subscriptions.topics(rawDataTopic))
    .map(input => DataRecord.parseBytes(input.value()))
    .collect{ case Success(data) => data }

val modelStream: Source[ModelImpl, Consumer.Control] =
  Consumer.atMostOnceSource(modelConsumerSettings,
      Subscriptions.topics(modelTopic))
    .map(input => Model.parseBytes(input.value()))
    .collect{ case Success(mod) => mod }
    .map(model => ModelImpl.findModel(model))
    .collect{ case Success(modImpl) => modImpl }
```

```
implicit val system = ActorSystem("ModelServing")
implicit val materializer = ActorMaterializer()
```

```
case class ModelScoringStage(scorer: …) extends
    GraphStageWithMaterializedValue[…, …] {

  val dataRecordIn = Inlet[Record]("dataRecordIn")
  val modelRecordIn = Inlet[ModelImpl]("modelRecordIn")
  val scoringResultOut = Outlet[ScoredRecord]("scoringOut")
  …
  setHandler(dataRecordIn, new InHandler {
    override def onPush(): Unit = {
      val record = grab(dataRecordIn)
      val newRecord = new ScoredRecord(scorer.score(record),record))
     push(scoringResultOut, Some(newRecord))
     pull(dataRecordIn)
  })
 …
}
```

```
    .map(model => ModelImpl.findModel(model))
    .collect{ case Success(modImpl) => modImpl }
```

```scala
implicit val system = ActorSystem("ModelServing")
implicit val materializer = ActorMaterializer()
implicit val executionContext = system.dispatcher

val modelProcessor = new ModelProcessor // Same as KS example
val scorer = new Scorer(modelProcessor) // Same as KS example
val modelScoringStage = new ModelScoringStage(scorer)// AS custom “stage”

val dataStream: Source[Record, Consumer.Control] =
  Consumer.atMostOnceSource(dataConsumerSettings,
      Subscriptions.topics(rawDataTopic))
    .map(input => DataRecord.parseBytes(input.value()))
    .collect{ case Success(data) => data }

val modelStream: Source[ModelImpl, Consumer.Control] =
  Consumer.atMostOnceSource(modelConsumerSettings,
      Subscriptions.topics(modelTopic))
    .map(input => Model.parseBytes(input.value()))
    .collect{ case Success(mod) => mod }
    .map(model => ModelImpl.findModel(model))
    .collect{ case Success(modImpl) => modImpl }
```

```
    .collect{ case Success(data) => data }
```

```
val modelStream: Source[ModelImpl, Consumer.Control] =
  Consumer.atMostOnceSource(modelConsumerSettings,
      Subscriptions.topics(modelTopic))
    .map(input => Model.parseBytes(input.value()))
    .collect{ case Success(mod) => mod }
    .map(model => ModelImpl.findModel(model))
    .collect{ case Success(modImpl) => modImpl }
    .foreach(modImpl => modelProcessor.setModel(modImpl))
modelStream.to(Sink.ignore).run() // No “sinking” required; just run
```

```
dataStream
  .viaMat(modelScoringStage)(Keep.right)
  .map(result => new ProducerRecord[Array[Byte], ScoredRecord](
    scoredRecordsTopic, result))
  .runWith(Producer.plainSink(producerSettings))
```

```
    .collect{ case Success(data) => data }

val modelStream: Source[ModelImpl, Consumer.Control] =
  Consumer.atMostOnceSource(modelConsumerSettings,
      Subscriptions.topics(modelTopic))
    .map(input => Model.parseBytes(input.value()))
    .collect{ case Success(mod) => mod }
    .map(model => ModelImpl.findModel(model))
    .collect{ case Success(modImpl) => modImpl }
    .foreach(modImpl => modelProcessor.setModel(modImpl))
modelStream.to(Sink.ignore).run() // No “sinking” required; just run

dataStream
  .viaMat(modelScoringStage)(Keep.right)
  .map(result => new ProducerRecord[Array[Byte], ScoredRecord](
    scoredRecordsTopic, result))
  .runWith(Producer.plainSink(producerSettings))
```

Other Concerns

Go Direct or Through Kafka?

vs.

?

- Extremely low latency
- Minimal I/O and memory overhead. No marshaling overhead
- Hard to scale, evolve independently

- Higher latency (network, queue depth)
- Higher I/O and processing (marshaling) overhead
- Easy independent scalability, evolution

Go Direct or Through Kafka?

vs.

?

- *Reactive Streams* back pressure
- "Direct" coupling between sender and receiver, but actually uses a URL abstraction

- Very deep buffer (partition limited by disk size)
- Strong decoupling - M producers, N consumers, completely disconnected

Wrapping Up…

A Spectrum of Microservices

akka Streams

Streams

Event-driven μ-services

“Record-centric” μ-services

Events

← Data Spectrum →

Records

Free as in 🍺

New second edition!
lbnd.io/fast-data-book

lightbend.com/fast-data-platform

Questions?

@deanwampler
lightbend.com/fast-data-platform
polyglotprogramming.com/talks

Lightbend