Scala School

Лекция 11: Akka

План лекции

- Message Delivery Reliability

- Cluster
- Cluster Sharding
- Distributed Publish Subscribe
- Event Sourcing

- Persistence
- Streams
- Кейсы

Message Delivery Reliability

The General Rules

These are the rules for message sends (i.e. the tell or ! method, which also underlies the ask pattern):

- at-most-once delivery, i.e. no guaranteed delivery
- message ordering per sender–receiver pair

The first rule is typically found also in other actor implementations while the second is specific to Akka.

Message Delivery Reliability

Discussion: What does “at-most-once” mean?

When it comes to describing the semantics of a delivery mechanism, there are three basic categories:

- at-most-once delivery means that for each message handed to the mechanism, that message is delivered zero or one times; in more casual terms it means that messages may be lost.
- at-least-once delivery means that for each message handed to the mechanism potentially multiple attempts are made at delivering it, such that at least one succeeds; again, in more casual terms this means that messages may be duplicated but not lost.
- exactly-once delivery means that for each message handed to the mechanism exactly one delivery is made to the recipient; the message can neither be lost nor duplicated.

Message Delivery Reliability

Reliability of Local Message Sends

The Akka test suite relies on not losing messages in the local context (and for non-error condition tests also for remote deployment), meaning that we actually do apply the best effort to keep our tests stable. A local tell operation can however fail for the same reasons as a normal method call can on the JVM:

- StackOverflowError
- OutOfMemoryError
- other VirtualMachineError

Message Delivery Reliability

In addition, local sends can fail in Akka-specific ways:

- if the mailbox does not accept the message (e.g. full BoundedMailbox)
- if the receiving actor fails while processing the message or is already terminated

While the first is clearly a matter of configuration the second deserves some thought: the sender of a message does not get feedback if there was an exception while processing, that notification goes to the supervisor instead. This is in general not distinguishable from a lost message for an outside observer.

Message Delivery Reliability: Higher-level abstractions

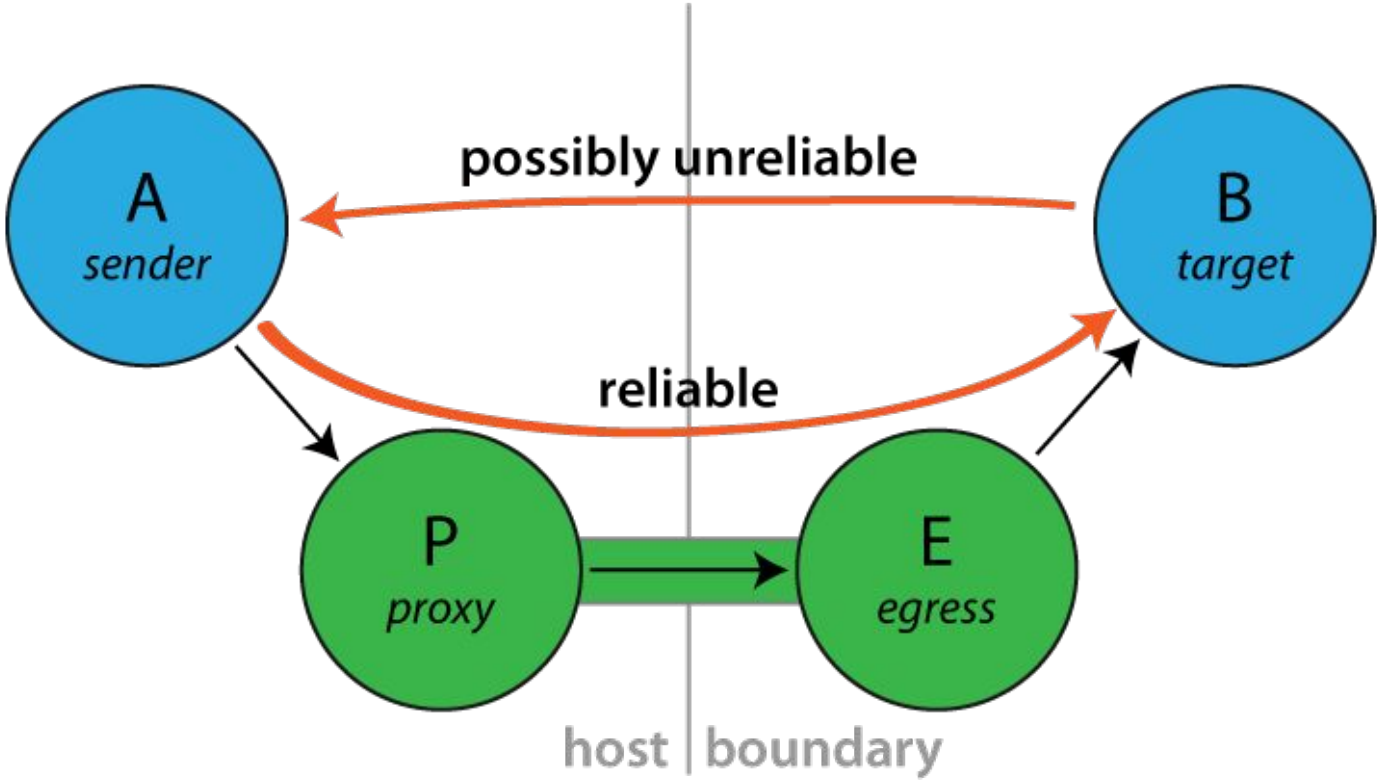
As discussed above a straight-forward answer to the requirement of reliable delivery is an explicit ACK–RETRY protocol. In its simplest form this requires

- a way to identify individual messages to correlate message with acknowledgement
- a retry mechanism which will resend messages if not acknowledged in time
- a way for the receiver to detect and discard duplicates

The third becomes necessary by virtue of the acknowledgements not being guaranteed to arrive either. An ACK-RETRY protocol with business-level acknowledgements is supported by At-Least-Once Delivery of the Akka Persistence module. Duplicates can be detected by tracking the identifiers of messages sent via At-Least-Once Delivery. Another way of implementing the third part would be to make processing the messages idempotent on the level of the business logic.

Another example of implementing all three requirements is shown at Reliable Proxy Pattern (which is now superseded by At-Least-Once Delivery).

Message Delivery Reliability: Reliable Proxy Pattern



Message Delivery Reliability

Since 2.5.0: Akka Persistence + trait AtLeastOnceDelivery

At-least-once delivery implies that original message sending order is not always preserved, and the destination may receive duplicate messages. Semantics do not match those of a normal ActorRef send operation:

- it is not at-most-once delivery
- message order for the same sender–receiver pair is not preserved due to possible resends
- after a crash and restart of the destination messages are still delivered to the new actor incarnation

Mailbox with Explicit Acknowledgement

```
class MyActor extends Actor {
  def receive = {
    case msg ⇒
      println(msg)
      doStuff(msg) // may fail
      PeekMailboxExtension.ack()
  }

  // business logic elided ...
}
```

Mailbox with Explicit Acknowledgement

```
object MyApp extends App {
  val system = ActorSystem("MySystem", ConfigFactory.parseString("""
    peek-dispatcher {
      mailbox-type = "akka.contrib.mailbox.PeekMailboxType"
      max-retries = 2
    }
    """))

  val myActor = system.actorOf(
    Props[MyActor].withDispatcher("peek-dispatcher"),
    name = "myActor")
  myActor ! "Hello"
  myActor ! "World"
  myActor ! PoisonPill
}
```

Cluster

Akka Cluster provides a fault-tolerant decentralized peer-to-peer based cluster membership service with no single point of failure or single point of bottleneck. It does this using gossip protocols and an automatic failure detector.

node - a logical member of a cluster. There could be multiple nodes on a physical machine. Defined by a hostname:port:uid tuple.

cluster - a set of nodes joined together through the membership service.

leader - a single node in the cluster that acts as the leader. Managing cluster convergence and membership state transitions.

Cluster

build.sbt:

```
"com.typesafe.akka" %% "akka-cluster" % "2.4.17"
```

application.conf:

```
akka {
  actor {
    provider = cluster
  }
  remote {
    netty.tcp {
      hostname = "127.0.0.1"
      port = 0
    }
  }
  cluster {
    seed-nodes = [
      "akka.tcp://ClusterSystem@127.0.0.1:2551",
      "akka.tcp://ClusterSystem@127.0.0.1:2552"] }}
```

Cluster

- Cluster Singleton

For some use cases it is convenient and sometimes also mandatory to ensure that you have exactly one actor of a certain type running somewhere in the cluster.

- Cluster Sharding

Distributes actors across several nodes in the cluster and supports interaction with the actors using their logical identifier, but without having to care about their physical location in the cluster.

Cluster

- Distributed Publish Subscribe

Publish-subscribe messaging between actors in the cluster, and point-to-point messaging using the logical path of the actors, i.e. the sender does not have to know on which node the destination actor is running.

- Cluster Client

Communication from an actor system that is not part of the cluster to actors running somewhere in the cluster. The client does not have to know on which node the destination actor is running.

Cluster

- Distributed Data

Akka Distributed Data is useful when you need to share data between nodes in an Akka Cluster. The data is accessed with an actor providing a key-value store like API.

Cluster Sharding

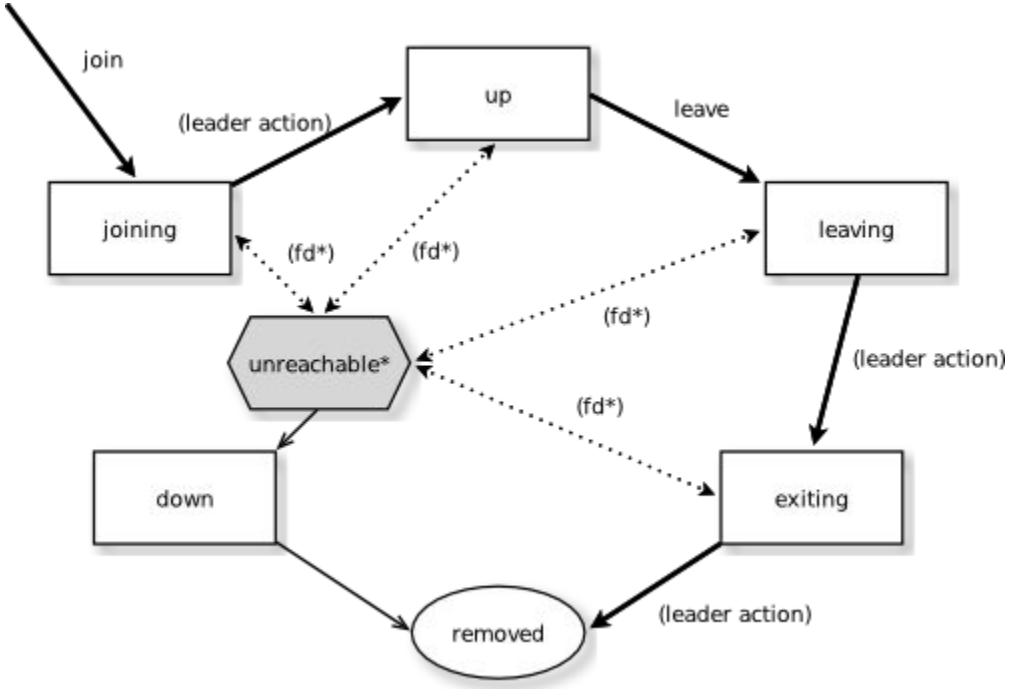
Cluster sharding is useful when you need to distribute actors across several nodes in the cluster and want to be able to interact with them using their logical identifier, but without having to care about their physical location in the cluster, which might also change over time.

Cluster sharding is typically used when you have many stateful actors that together consume more resources (e.g. memory) than fit on one machine. If you only have a few stateful actors it might be easier to run them on a Cluster Singleton node.

Cluster Sharding

In this context sharding means that actors with an identifier, so called entities, can be automatically distributed across multiple nodes in the cluster. Each entity actor runs only at one place, and messages can be sent to the entity without requiring the sender to know the location of the destination actor. This is achieved by sending the messages via a ShardRegion actor provided by this extension, which knows how to route the message with the entity id to the final destination.

Cluster Sharding



Cluster Sharding

build.sbt:

```
"com.typesafe.akka" %% "akka-cluster-sharding" % "2.4.17"
```

Cluster Sharding

```
case object Increment
case object Decrement
final case class Get(counterId: Long)
final case class EntityEnvelope(id: Long, payload: Any)

case object Stop
final case class CounterChanged(delta: Int)

class Counter extends PersistentActor {
  import ShardRegion.Passivate
  context.setReceiveTimeout(120.seconds)
  // self.path.name is the entity identifier (utf-8 URL-encoded)
  override def persistenceId: String = "Counter-" + self.path.name
  var count = 0
```

Cluster Sharding

```
def updateState(event: CounterChanged): Unit =
    count += event.delta

  override def receiveRecover: Receive = {
    case evt: CounterChanged ⇒ updateState(evt)
  }

  override def receiveCommand: Receive = {
    case Increment      ⇒ persist(CounterChanged(+1))(updateState)
    case Decrement      ⇒ persist(CounterChanged(-1))(updateState)
    case Get(_)         ⇒ sender() ! count
    case ReceiveTimeout ⇒ context.parent ! Passivate(stopMessage = Stop)
    case Stop           ⇒ context.stop(self)
  }
}
```

Cluster Sharding

```
val counterRegion: ActorRef = ClusterSharding(system).start(
  typeName = "Counter",
  entityProps = Props[Counter],
  settings = ClusterShardingSettings(system),
  extractEntityId = extractEntityId,
  extractShardId = extractShardId)
counterRegion ! Get(123)
counterRegion ! Get(123)
val extractEntityId: ShardRegion.ExtractEntityId = {
  case EntityEnvelope(id, payload) ⇒ (id.toString, payload)
  case msg @ Get(id)               ⇒ (id.toString, msg)
}
val numberOfShards = 100
val extractShardId: ShardRegion.ExtractShardId = {
  case EntityEnvelope(id, _) ⇒ (id % numberOfShards).toString
  case Get(id)               ⇒ (id % numberOfShards).toString
}
```

Distributed Publish Subscribe

- How do I send a message to an actor without knowing which node it is running on?
- How do I send messages to all actors in the cluster that have registered interest in a named topic?
- There a two different modes of message delivery: Publish and Send.

Distributed Publish Subscribe

build.sbt:

```
"com.typesafe.akka" %% "akka-cluster-tools" % "2.4.17"
```

application.conf:

```
akka.extensions = ["akka.cluster.pubsub.DistributedPubSub"]
```

Distributed Publish Subscribe: Publish

This is the true pub/sub mode. A typical usage of this mode is a chat room in an instant messaging application.

Actors are registered to a named topic. This enables many subscribers on each node. The message will be delivered to all subscribers of the topic.

Distributed Publish Subscribe: Publish

```
class Subscriber extends Actor with ActorLogging {
 import DistributedPubSubMediator.{ Subscribe, SubscribeAck }
 val mediator = DistributedPubSub(context.system).mediator
  // subscribe to the topic named "content"
  mediator ! Subscribe("content", self)

  def receive = {
    case s: String ⇒
      log.info("Got {}", s)
    case SubscribeAck(Subscribe("content", None, `self`)) ⇒
      log.info("subscribing");
  }
}
```

Distributed Publish Subscribe: Publish

```
system.actorOf(Props[Subscriber], "subscriber1")

class Publisher extends Actor {
 import DistributedPubSubMediator.Publish
 // activate the extension
  val mediator = DistributedPubSub(context.system).mediator

  def receive = {
    case in: String ⇒
      val out = in.toUpperCase
      mediator ! Publish("content", out)
  }
}
```

Distributed Publish Subscribe: Send

This is a point-to-point mode where each message is delivered to one destination, but you still does not have to know where the destination is located. A typical usage of this mode is private chat to one other user in an instant messaging application. It can also be used for distributing tasks to registered workers, like a cluster aware router where the routees dynamically can register themselves.

Distributed Publish Subscribe: Send

```
 system.actorOf(Props[Destination], "destination")
 system.actorOf(Props[Destination], "destination")

class Sender extends Actor {
 import DistributedPubSubMediator.Send
 // activate the extension
  val mediator = DistributedPubSub(context.system).mediator

  def receive = {
    case in: String ⇒
      val out = in.toUpperCase
      mediator ! Send(path = "/user/destination", msg = out, localAffinity = true)
  }
}
```

Distributed Publish Subscribe: Delivery Guarantee

As in Message Delivery Reliability of Akka, message delivery guarantee in distributed pub sub modes is at-most-once delivery. In other words, messages can be lost over the wire.

If you are looking for at-least-once delivery guarantee, we recommend Kafka Akka Streams integration.

Event Sourcing

Akka persistence enables stateful actors to persist their internal state so that it can be recovered when an actor is started, restarted after a JVM crash or by a supervisor, or migrated in a cluster. The key concept behind Akka persistence is that only changes to an actor's internal state are persisted but never its current state directly (except for optional snapshots). These changes are only ever appended to storage, nothing is ever mutated, which allows for very high transaction rates and efficient replication. Stateful actors are recovered by replaying stored changes to these actors from which they can rebuild internal state. This can be either the full history of changes or starting from a snapshot which can dramatically reduce recovery times. Akka persistence also provides point-to-point communication with at-least-once message delivery semantics.

Event Sourcing

Event sourcing (and sharding) is what makes large websites scale to billions of users, and the idea is quite simple: when a component (think actor) processes a command it will generate a list of events representing the effect of the command. These events are stored in addition to being applied to the component's state. The nice thing about this scheme is that events only ever are appended to the storage, nothing is ever mutated; this enables perfect replication and scaling of consumers of this event stream (i.e. other components may consume the event stream as a means to replicate the component's state on a different continent or to react to changes). If the component's state is lost—due to a machine failure or by being pushed out of a cache—it can easily be reconstructed by replaying the event stream (usually employing snapshots to speed up the process). Event sourcing is supported by Akka Persistence.

Event Sourcing

The basic idea behind Event Sourcing is quite simple. A persistent actor receives a (non-persistent) command which is first validated if it can be applied to the current state. Here validation can mean anything, from simple inspection of a command message's fields up to a conversation with several external services, for example. If validation succeeds, events are generated from the command, representing the effect of the command. These events are then persisted and, after successful persistence, used to change the actor's state. When the persistent actor needs to be recovered, only the persisted events are replayed of which we know that they can be successfully applied. In other words, events cannot fail when being replayed to a persistent actor, in contrast to commands. Event sourced actors may of course also process commands that do not change application state such as query commands for example.

Akka Persistence

Akka persistence is a separate jar file. Make sure that you have the following dependency in your project:

```
"com.typesafe.akka" %% "akka-persistence" % "2.4.17"
```

The Akka persistence extension comes with few built-in persistence plugins, including in-memory heap based journal, local file-system based snapshot-store and LevelDB based journal.

LevelDB based plugins will require the following additional dependency declaration:

```
"org.iq80.leveldb"            % "leveldb"          % "0.7"

"org.fusesource.leveldbjni"   % "leveldbjni-all"   % "1.8"
```

Akka Persistence: Architecture

- PersistentActor: Is a persistent, stateful actor. It is able to persist events to a journal and can react to them in a thread-safe manner. It can be used to implement both command as well as event sourced actors. When a persistent actor is started or restarted, journaled messages are replayed to that actor so that it can recover internal state from these messages.

- PersistentView: A view is a persistent, stateful actor that receives journaled messages that have been written by another persistent actor. A view itself does not journal new messages, instead, it updates internal state only from a persistent actor's replicated message stream.

- AtLeastOnceDelivery: To send messages with at-least-once delivery semantics to destinations, also in case of sender and receiver JVM crashes.

Akka Persistence: Architecture

- AsyncWriteJournal: A journal stores the sequence of messages sent to a persistent actor. An application can control which messages are journaled and which are received by the persistent actor without being journaled. Journal maintains highestSequenceNr that is increased on each message. The storage backend of a journal is pluggable. The persistence extension comes with a "leveldb" journal plugin, which writes to the local filesystem. Replicated journals are available as Community plugins.

- Snapshot store: A snapshot store persists snapshots of a persistent actor's or a view's internal state. Snapshots are used for optimizing recovery times. The storage backend of a snapshot store is pluggable. The persistence extension comes with a "local" snapshot storage plugin, which writes to the local filesystem. Replicated snapshot stores are available as Community plugins.

Akka Persistence: Architecture

```
import akka.actor._
import akka.persistence._
case class Cmd(data: String)
case class Evt(data: String)

case class ExampleState(events: List[String] = Nil) {
  def updated(evt: Evt): ExampleState = copy(evt.data :: events)
  def size: Int = events.length
  override def toString: String = events.reverse.toString
}

class ExamplePersistentActor extends PersistentActor {
  override def persistenceId = "sample-id-1"

  var state = ExampleState()
```

Akka Persistence: Architecture

```
...
val receiveRecover: Receive = {
  case evt: Evt                                 => updateState(evt)
  case SnapshotOffer(_, snapshot: ExampleState) => state = snapshot
}

val receiveCommand: Receive = {
  case Cmd(data) =>
    persist(Evt(s"${data}-${numEvents}"))(updateState)
    persist(Evt(s"${data}-${numEvents + 1}")) { event =>
      updateState(event)
      context.system.eventStream.publish(event)
    }
  case "snap"  => saveSnapshot(state)
  case "print" => println(state)}}
```

Akka Streams

Actors can be seen as dealing with streams as well: they send and receive series of messages in order to transfer knowledge (or data) from one place to another. We have found it tedious and error-prone to implement all the proper measures in order to achieve stable streaming between actors, since in addition to sending and receiving we also need to take care to not overflow any buffers or mailboxes in the process. Another pitfall is that Actor messages can be lost and must be retransmitted in that case lest the stream have holes on the receiving side. When dealing with streams of elements of a fixed given type, Actors also do not currently offer good static guarantees that no wiring errors are made: type-safety could be improved in this case.

Akka Streams

For these reasons we decided to bundle up a solution to these problems as an Akka Streams API. The purpose is to offer an intuitive and safe way to formulate stream processing setups such that we can then execute them efficiently and with bounded resource usage—no more OutOfMemoryErrors. In order to achieve this our streams need to be able to limit the buffering that they employ, they need to be able to slow down producers if the consumers cannot keep up. This feature is called back-pressure and is at the core of the Reactive Streams initiative of which Akka is a founding member. For you this means that the hard problem of propagating and reacting to back-pressure has been incorporated in the design of Akka Streams already, so you have one less thing to worry about; it also means that Akka Streams interoperate seamlessly with all other Reactive Streams implementations (where Reactive Streams interfaces define the interoperability SPI while implementations like Akka Streams offer a nice user API).

Akka Streams

```
object SimpleStreamExample extends App {
  val source: Source[Int, NotUsed] = Source(1 to 100)
  implicit val system = ActorSystem("QuickStart")
  implicit val materializer = ActorMaterializer()
  implicit val ec = ExecutionContext.global
  source.runForeach(i => println(i))(materializer).onComplete(_ =>
Await.result(system.terminate(), Duration.Inf))
}
```

Akka Streams

```
object IOStreamExample extends App {
  implicit val system = ActorSystem()
  implicit val materializer = ActorMaterializer()
  implicit val ec = ExecutionContext.global
  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")))
  result onComplete {
    res =>
    println(res)
    Await.result(system.terminate(), Duration.Inf)
  }
}
```

Akka Streams

```
object ReuseIOStreamExample extends App {
  implicit val system = ActorSystem()
  implicit val materializer = ActorMaterializer()
  implicit val ec = ExecutionContext.global
  val source: Source[Int, NotUsed] = Source(1 to 100)
  val factorials = source.scan(BigInt(1))((acc, next) => acc * next)
  def lineSink(filename: String): Sink[String, Future[IOResult]] =
    Flow[String]
      .map(s => ByteString(s + "\n"))
      .toMat(FileIO.toPath(Paths.get(filename)))(Keep.right)
  val result: Future[IOResult] =
    factorials.map(_.toString).runWith(lineSink("factorial2.txt"))
  result onComplete (_ => Await.result(system.terminate(), Duration.Inf))
}
```

Akka Streams

```
object StreamZipExample extends App {
  implicit val system = ActorSystem()
  implicit val materializer = ActorMaterializer()
  implicit val ec = ExecutionContext.global
  val source: Source[Int, NotUsed] = Source(1 to 100)
  val factorials = source.scan(BigInt(1))((acc, next) => acc * next)
  val done: Future[Done] =
    factorials
      .zipWith(Source(0 to 100))((num, idx) => s"$idx! = $num")
      .throttle(1, 1.second, 1, ThrottleMode.shaping)
      .runForeach(println)
  done.onComplete(_ => Await.result(system.terminate(), Duration.Inf))
}
```

Akka HTTP

The Akka HTTP modules implement a full server- and client-side HTTP stack on top of *akka-actor* and *akka-stream*. It's not a web-framework but rather a more general toolkit for providing and consuming HTTP-based services. While interaction with a browser is of course also in scope it is not the primary focus of Akka HTTP.

Akka HTTP

```
object WebServer {
  def main(args: Array[String]) {

    implicit val system = ActorSystem("my-system")
    implicit val materializer = ActorMaterializer()
    // needed for the future flatMap/onComplete in the end
    implicit val executionContext = system.dispatcher

    val route =
      path("hello") {
        get {
          complete(HttpEntity(ContentTypes.`text/html(UTF-8)`, "<h1>Say hello to akka-http</h1>"))
        }
      }
```

Akka HTTP

```
    val bindingFuture = Http().bindAndHandle(route, "localhost", 8080)

    println(s"Server online at http://localhost:8080/\nPress RETURN to stop...")
    StdIn.readLine() // let it run until user presses return
    bindingFuture
      .flatMap(_.unbind()) // trigger unbinding from the port
      .onComplete(_ => system.terminate()) // and shutdown when done
  }
}
```

Akka HTTP

```
object WebServer {

  def main(args: Array[String]) {

    implicit val system = ActorSystem()
    implicit val materializer = ActorMaterializer()
    // needed for the future flatMap/onComplete in the end
    implicit val executionContext = system.dispatcher

    // streams are re-usable so we can define it here
    // and use it for every request
    val numbers = Source.fromIterator(() =>
      Iterator.continually(Random.nextInt()))
```

Akka HTTP

```
val route =
    path("random") {
      get {
        complete(
          HttpEntity(
            ContentTypes.`text/plain(UTF-8)`,
            // transform each number to a chunk of bytes
            numbers.map(n => ByteString(s"$n\n"))
          )
        )
      }
    }
```

Akka HTTP

```
object WebServer {

  case class Bid(userId: String, offer: Int)
  case object GetBids
  case class Bids(bids: List[Bid])

  class Auction extends Actor with ActorLogging {
    var bids = List.empty[Bid]
    def receive = {
      case bid @ Bid(userId, offer) =>
        bids = bids :+ bid
        log.info(s"Bid complete: $userId, $offer")
      case GetBids => sender() ! Bids(bids)
      case _ => log.info("Invalid message")
    }
  }
```

Akka HTTP

```
// these are from spray-json
  implicit val bidFormat = jsonFormat2(Bid)
  implicit val bidsFormat = jsonFormat1(Bids)

  def main(args: Array[String]) {
    implicit val system = ActorSystem()
    implicit val materializer = ActorMaterializer()
    // needed for the future flatMap/onComplete in the end
    implicit val executionContext = system.dispatcher

    val auction = system.actorOf(Props[Auction], "auction")
```

Akka HTTP

```
val route =
      path("auction") {
        put {
          parameter("bid".as[Int], "user") { (bid, user) =>
            // place a bid, fire-and-forget
            auction ! Bid(user, bid)
            complete((StatusCodes.Accepted, "bid placed"))
          }
        } ~
        get {
          implicit val timeout: Timeout = 5.seconds
          // query the actor for the current auction state
          val bids: Future[Bids] = (auction ? GetBids).mapTo[Bids]
          complete(bids)}}
```

Akka HTTP

```
    val bindingFuture = Http().bindAndHandle(route, "localhost", 8080)
     println(s"Server online at http://localhost:8080/\nPress RETURN to stop...")
     StdIn.readLine() // let it run until user presses return
     bindingFuture
       .flatMap(_.unbind()) // trigger unbinding from the port
       .onComplete(_ => system.terminate()) // and shutdown when done
  }
}
```

Материалы

https://efekahraman.github.io/2017/02/a-simple-akka-cluster-application

http://eax.me/akka-cluster-basics/

http://blog.colinbreck.com/akka-streams-a-motivating-example/

http://blog.colinbreck.com/patterns-for-streaming-measurement-data-with-akka-streams/

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

Спасибо