akka on Scala

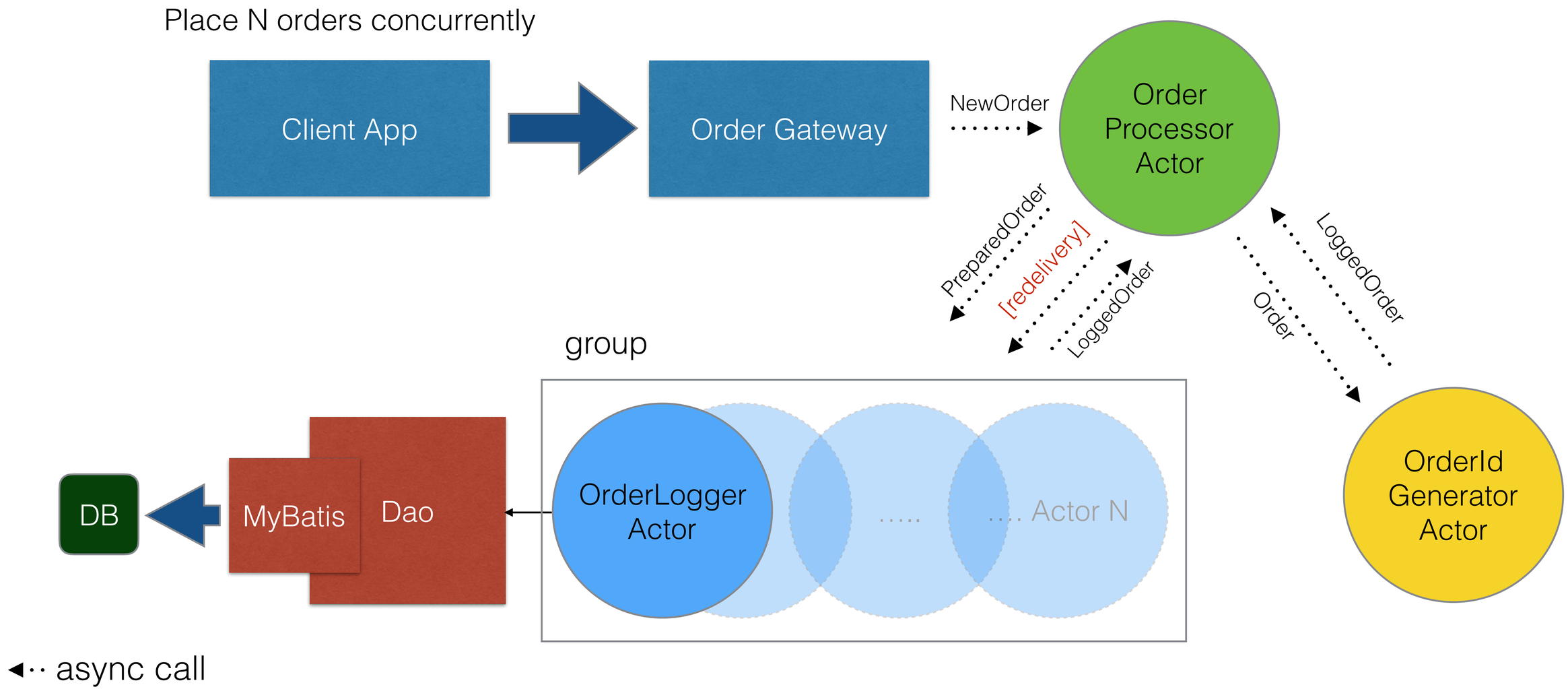
TestKit, Supervision, FSM

Alexey Novakov

Agenda

- Akka on Scala - main difference (key features)
- Supervision strategy & actor lifecycle
- Akka TestKit
- FSM actor

Recall „Order Service“ App



onReceive: java approach

```
public void onReceive(Object message) {

    if (msg instanceof NewOrder) {
     ……
     persist(msg, this::generateOrderId);

    } else if (msg instanceof PreparedOrder) {
      …
      persist(msg, this::updateState);

    } else if (msg instanceof LoggedOrder) {
      ……

      LoggedOrder loggedOrder = (LoggedOrder) msg;
      executor.tell(new ExecuteOrder(loggedOrder.order.orderId,
         loggedOrder.order.quantity, self())
    }
```

Pattern matching: scala approach

The act of checking a given sequence of **tokens** for the presence of the **constituents** of some pattern

```
override def receiveCommand: Receive = {
  case newOrder: NewOrder ⇒
    log.info("New order received. Going to generate an id: {}", newOrder)
    persist(newOrder)(generateOrderId)

  case preparedOrder@PreparedOrder(order, orderId) ⇒
    log.info("Prepared order received with id = {}, {}", orderId, order)
    persist(preparedOrder)(updateState)

  case loggedOrder: LoggedOrder ⇒
    updateState(loggedOrder)
    log.info("Delivery confirmed for order = {}", loggedOrder)
    executor ! ExecuteOrder(loggedOrder.order.orderId,
                            loggedOrder.order.quantity)

  ……
}
```

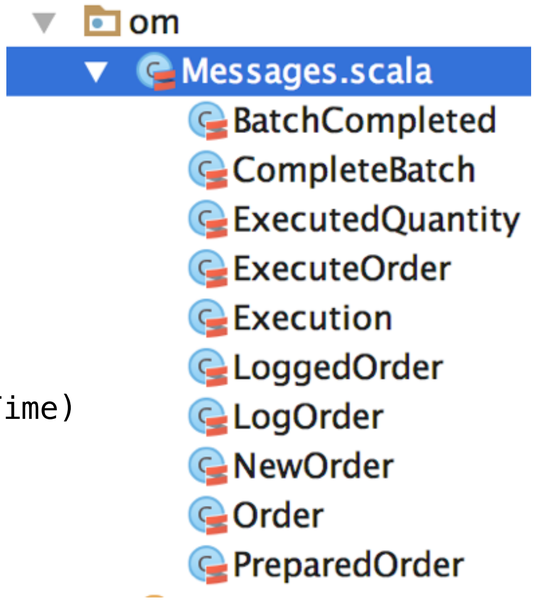
Concise API

- actor creation using “**apply**” method
  - `context.actorOf(Props[OrderIdGenerator], "orderIdGenerator")`
- new instance of BatchCompleted using “**apply**” method and concise method name for “tell”
  - `sender ! BatchCompleted(c.upToId)`
- **constructor** embedded in definition
  - `class OrderLogger(orderDao: IOrderDao, randomFail: Boolean) extends Actor`

3. Better encapsulation

All messages (commands, events) are in the single Scala file

```
case class Order(orderId: Long = -1, executionDate: LocalDateTime, orderType: OrderType,
          executionPrice: BigDecimal, symbol: String, userId: Int, quantity: Int)

case class Execution(orderId: Long, quantity: Int, executionDate: LocalDateTime)

case class NewOrder(order: Order)

case class PreparedOrder(order: Order, orderId: Long)

case class LoggedOrder(deliveryId: Long, order: Order)

case class LogOrder(deliveryId: Long, preparedOrder: PreparedOrder)

case class ExecuteOrder(orderId: Long, quantity: Int)

case class ExecutedQuantity(orderId: Long, quantity: Int, executionDate: LocalDateTime)

case class CompleteBatch(upToId: Int, withDate: LocalDateTime)

case class BatchCompleted(upToId: Int)
```



Supervision

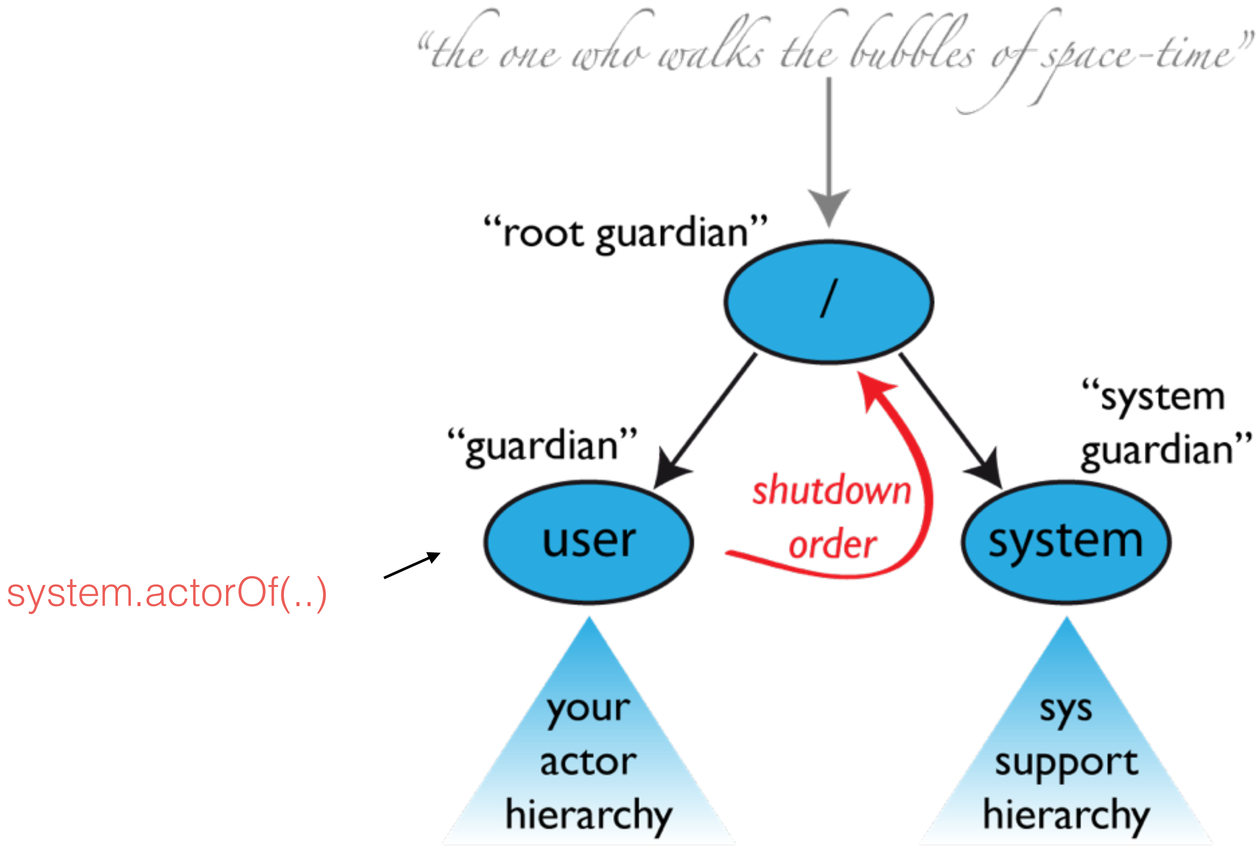
```
/**
 * Resumes message processing for the failed Actor
 */
case object Resume extends Directive

/**
 * Discards the old Actor instance and replaces it with a new,
 * then resumes message processing.
 */
case object Restart extends Directive

/**
 * Stops the Actor
 */
case object Stop extends Directive

/**
 * Escalates the failure to the supervisor of the supervisor,
 * by rethrowing the cause of the failure, i.e. the supervisor fails with
 * the same exception as the child.
 */
case object Escalate extends Directive
```

no mailbox clearing on restart

- it is a dependency **relationship** between actors: the supervisor delegates tasks to subordinates and therefore must respond to their failures.

4 Options

The Top-level Supervisors



* There are special **system messages** which maintain supervision and monitoring

example URI: [akka://**OrderGatewaySystem**/user/orderProcessor/orderLogger/$b]

Supervision Strategy

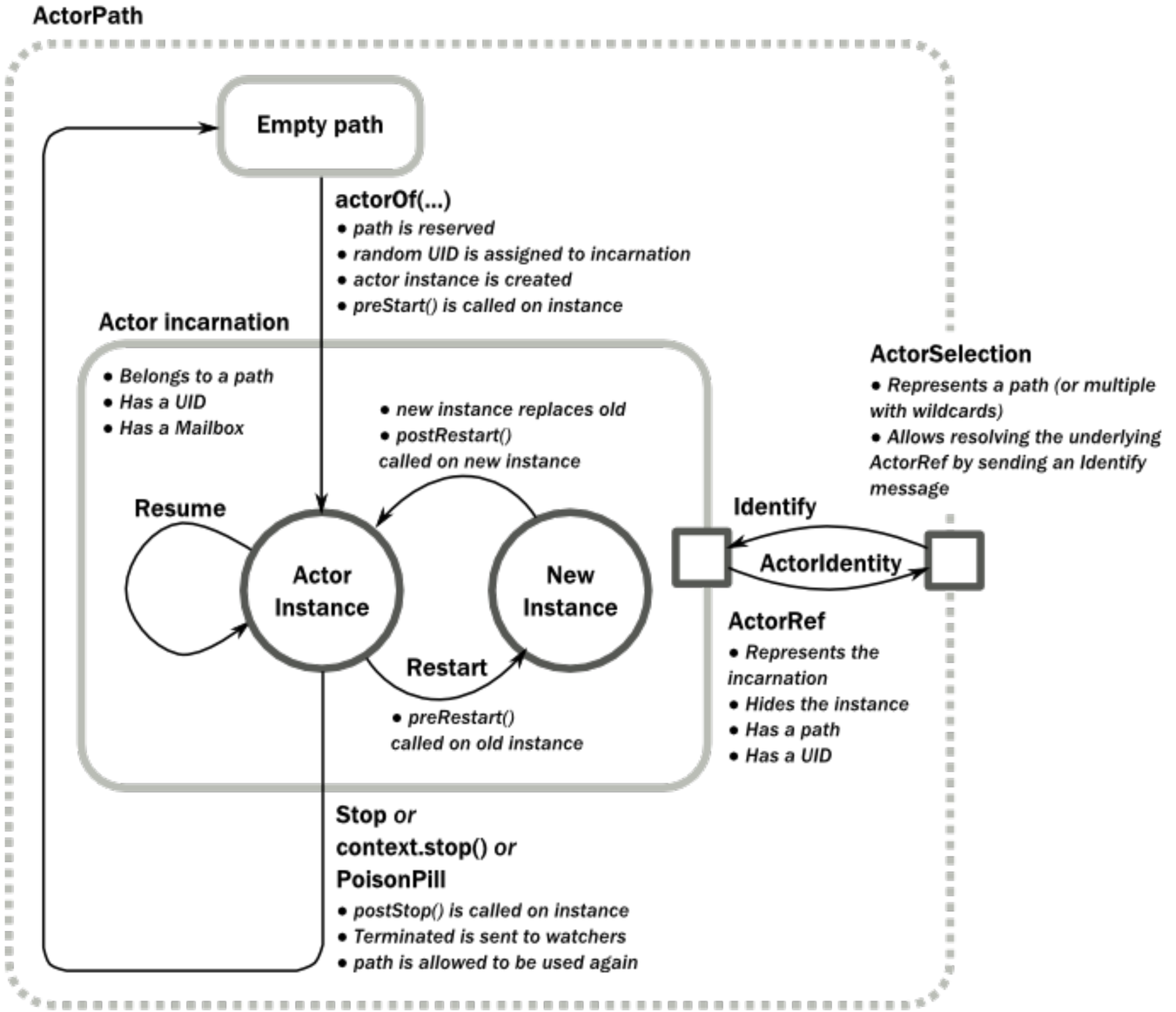
Two Strategies:

a) *OneForOneStrategy* - applies directive to the failed child

```
  override def supervisorStrategy = OneForOneStrategy(
   maxNrOfRetries = 10,
   withinTimeRange = 1 minute) {
     case _: UnsupportedOperationException => Resume
     case _: NullPointerException => Restart
     case _: IllegalArgumentException => Stop
     case _: Exception => Escalate
   }
```

b) *AllForOneStrategy* - applies action to all sibling child actors

Actor Lifecycle



TestKit

- **akka-testkit** module to dependencies

2 options:

- **Unit testing** - no actor model involved, no concurrency, deterministic behavior
- **Integration testing** - actor model, concurrency, non-deterministic behavior

Unit Testing

- Send message synchronously
- Verify actor state or interactions with other resources

Unit Test for OrderLogger actor:

```
class OrderLoggerTest extends TestKit(ActorSystem("testSystem"))


it should "save order into database" in {
  //given
  val orderDao = stub[IOrderDao]
  val orderLoggerRef = TestActorRef[OrderLogger]
                                  (Props(classOf[OrderLogger], orderDao, false))
  val generatedOrder = OrderUtil.generateRandomOrder

  //when
  orderLoggerRef ! LogOrder(1, PreparedOrder(generatedOrder, 2))

  //then
  orderDao.saveOrder _ verify new Order(2L, generatedOrder) once()
}
```

Integration Testing

- Send message asynchronously
- Verify actor works correctly within the environment

```
class ITOrderLogger extends TestKit(ActorSystem("OrderProcessing")) {

it should "save order" in {
  //given
  val orderDao = mock[IOrderDao]
  val orderLogger = actor(orderDao)
  val order = OrderUtil.generateRandomOrder
  val orderWithId = new Order(2, order)
  //when
  orderDao.saveOrder _ expects orderWithId
  orderLogger ! LogOrder(1L, PreparedOrder(order, 2))
  //then
  val savedOrder = expectMsgAnyClassOf(classOf[LoggedOrder])
  savedOrder.order should be(orderWithId)
}

def actor(orderDao: IOrderDao) = system.actorOf(Props(classOf[OrderLogger],
orderDao, false), "persist" + Random.nextInt())
```

Integration Testing: TestProbe

```
//given
val orderIdGenerator = TestProbe()
val orderLogger = TestProbe()
val orderExecutor = TestProbe()
val dao = mock[IOrderDao]
val orderProcessor = orderProcessorActor(dao, orderIdGenerator, orderLogger, orderExecutor)
val order = OrderUtil.generateRandomOrder

it should "generate id and persist incoming order" in {
  //when
  orderProcessor ! NewOrder(order)
  //then
  val receivedOrder = orderIdGenerator.expectMsgAnyClassOf(classOf[Order])
  receivedOrder should be(order)

  //when
  orderProcessor ! PreparedOrder(order, 1)
  //then
  val preparedOrderForAck = orderLogger.expectMsgAnyClassOf(classOf[LogOrder])
  preparedOrderForAck.preparedOrder.orderId should be(1)
}
```

Finite State Machine

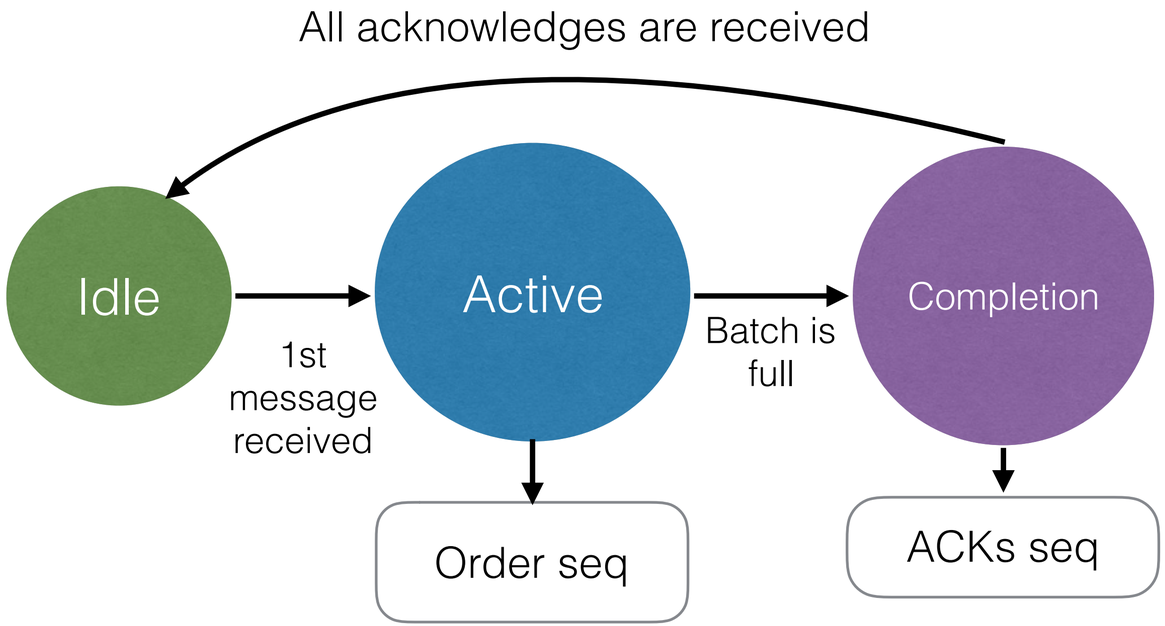
… or Finite State Automaton

is a mathematical model of computation used to design both computer programs and sequential logic circuits. It is conceived as an abstract machine that can be in one of a finite number of *states*

State(S) x Event(E) -> Actions (A), State(S')

If we are in state **S** and the event **E** occurs, we should perform the actions **A** and make a transition to the state **S'**.

“Batching” flow



State, Data types

```
sealed trait State

case object Idle extends State

case object Active extends State

case object Completion extends State

sealed trait Data

case object Uninitialized extends Data

final case class PendingBatch(queue: Seq[ExecuteOrder]) extends Data

final case class AckBatch(replies: Seq[Any]) extends Data
```

FSM Actor

- Describes **what to do** in particular **state** when **new messages** received. Change its State and Data, when needed
- Akka provides special DSL: when, onTransition, startWith, goto, using, stay, stop

```
when(Idle) {
  case Event(eo: ExecuteOrder, Uninitialized) =>
                                    goto(Active) using PendingBatch(Seq(eo))
}

when(Active) {
  case Event(eo: ExecuteOrder, b@PendingBatch(q)) if q.length < batchSize =>
    b.copy(queue = q :+ eo) match {
      case ub@PendingBatch(uq) if uq.length == batchSize =>
        flush(uq)
        goto(Idle) using Uninitialized
      case ub =>
        log.info("new message added = {}", eo)
        stay using ub
    }
}
```

The End. Questions?

Links for more details:

- http://akka.io/docs/
- https://www.lightbend.com/blog/akka
- http://blog.akka.io
- https://blog.codecentric.de/en/2015/07/a-map-of-akka/