Akka: Arquitetura Orientada a Atores

Fabiano Guizellini Modos

Arquiteto de Software – HBSIS

@fmodos

ACTOR Model = Modelo de Atores

Agenda

- Conceito de Atores
- Non Blocking / Blocking
- Comunicação Assíncrona
- Supervisor
- Router
- Cluster

"If you can't explain it simply, you don't understand it well enough"

Albert Einstein

Quem Usa Akka?

“1. Scalable, both horizontally to hundreds of nodes and vertically to very many processors, handling billions of requests per day

2. Low latency, controllable at a very fine grain

3. Resilient to failure

4. Flexibility in adjusting the service boundaries

5. A programming model AND culture encouraging scalability and simplicity, including clean failure and error handling”

"How PayPal Scaled To Billions Of Transactions Daily Using Just 8VMs"
http://highscalability.com/blog/2016/8/15/how-paypal-scaled-to-billions-of-transactions-daily-using-ju.html

"A New, Reactive Way for PayPal to Build Applications"
https://www.paypal-engineering.com/2016/05/11/squbs-a-new-reactive-way-for-paypal-to-build-applications/

Quem Usa Akka?

"How does one display them in real time for roughly 500 million members on a social network like LinkedIn?"

"Now You See Me, Now You Don't: LinkedIn's Real-Time Presence Platform"
https://engineering.linkedin.com/blog/2018/01/now-you-see-me--now-you-dont--linkedins-real-time-presence-platf

Ano 1973 - A Universal Modular ACTOR Formalism for Artificial Intelligence

"Current developments in hardware technology are making it economically attractive to run many physical processors in parallel.... The actor formalism provides a coherent method for organizing and controlling all these processors." by Carl Hewitt

Ano 1986 - ERLANG

A Ericsson desenvolveu essa linguagem baseada no conceito de atores com o objetivo de melhorar o desenvolvimento de aplicações de telefonia.

"when you build a system in Erlang, you can see the system as a serious of very small programs"
Greg Young
https://vimeo.com/108441214

QUEM USA?

RabbitMQ

WhatsApp

Ano 2009 - Akka

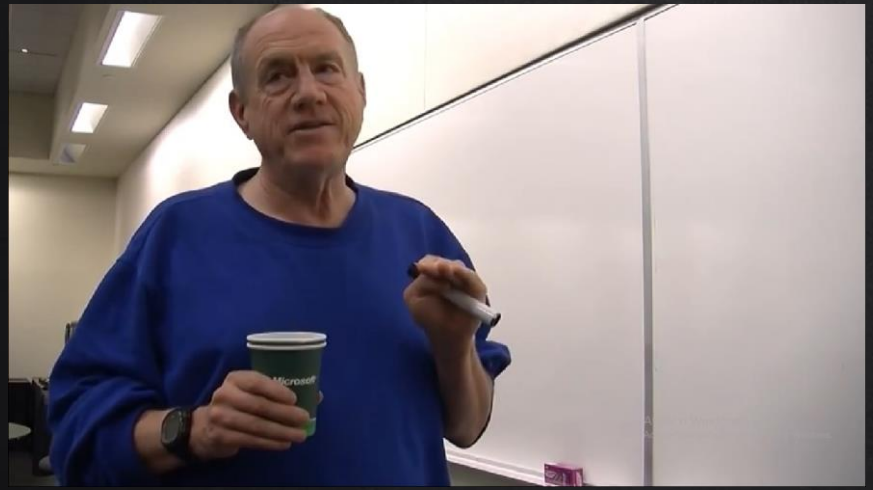
Jonas Bonér é um Java Champion. Ele criou o Akka em Java inspirado no conceito de ACTOR e da linguagem Erlang.

Ano 2015 - Quando eu conheci Akka 

Utilizado no produto Unidocs da HBSIS

- ✗ Responsável pela emissão de documentos de transporte
- ✗ ~30 milhões de documentos
- ✗ ~1000 atores divididos em 8 microservices
- ✗ Processa ~80k eventos por hora

ACTOR by Carl Hewitt



https://www.youtube.com/watch?v=7erJ1DV_Tlo

ACTOR é um agente com as seguinte responsabilidades:

- Processing
- Storage
- Communication

Hands On

Otimizar as filas do Subway

Projeto Subway

- ✗ FIFO – Fila
- ✗ Exception – Cliente Desmaiou
- ✗ Load – Garçom Sobrecarregado
- ✗ Pergunta do Milhão

Problema: Um Cliente Lento Afeta Toda a Fila

FIFO – First In First Out

```
Waiter waiter = n
int delayToChoose
waiter.receiveOrd                                    oChoose));
waiter.receiveOrder(new NewSandwich("Fast Client", 0));
```

BLOCKING

Fila Blocking

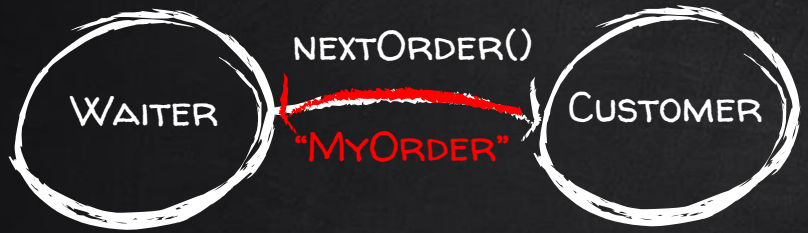
Varias Filas Blocking

~~FIFO - Blocking~~

FOFO - Faster Order Faster Out

Comunicação

Blocking



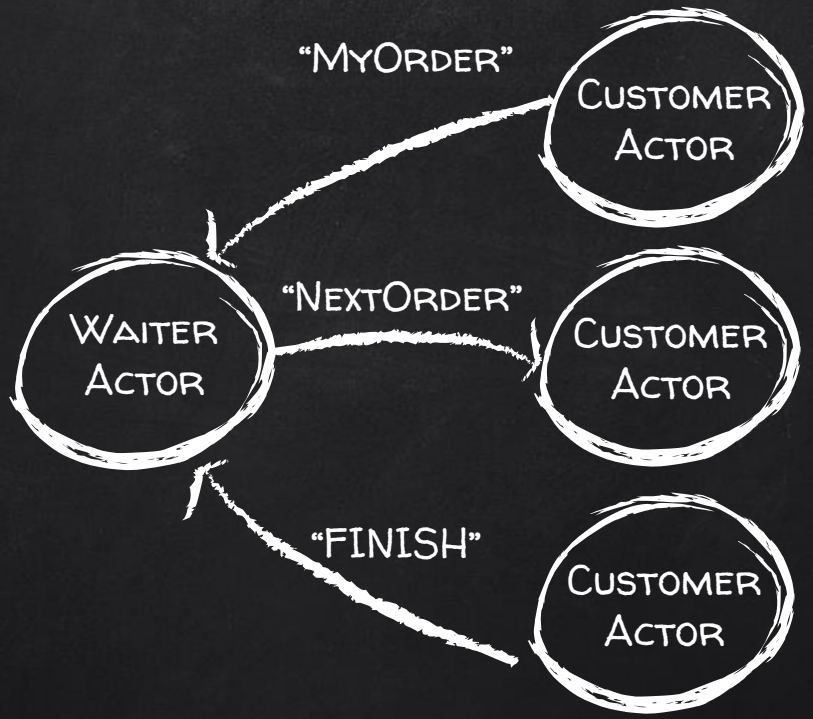
Non Blocking

Comunicação

Blocking

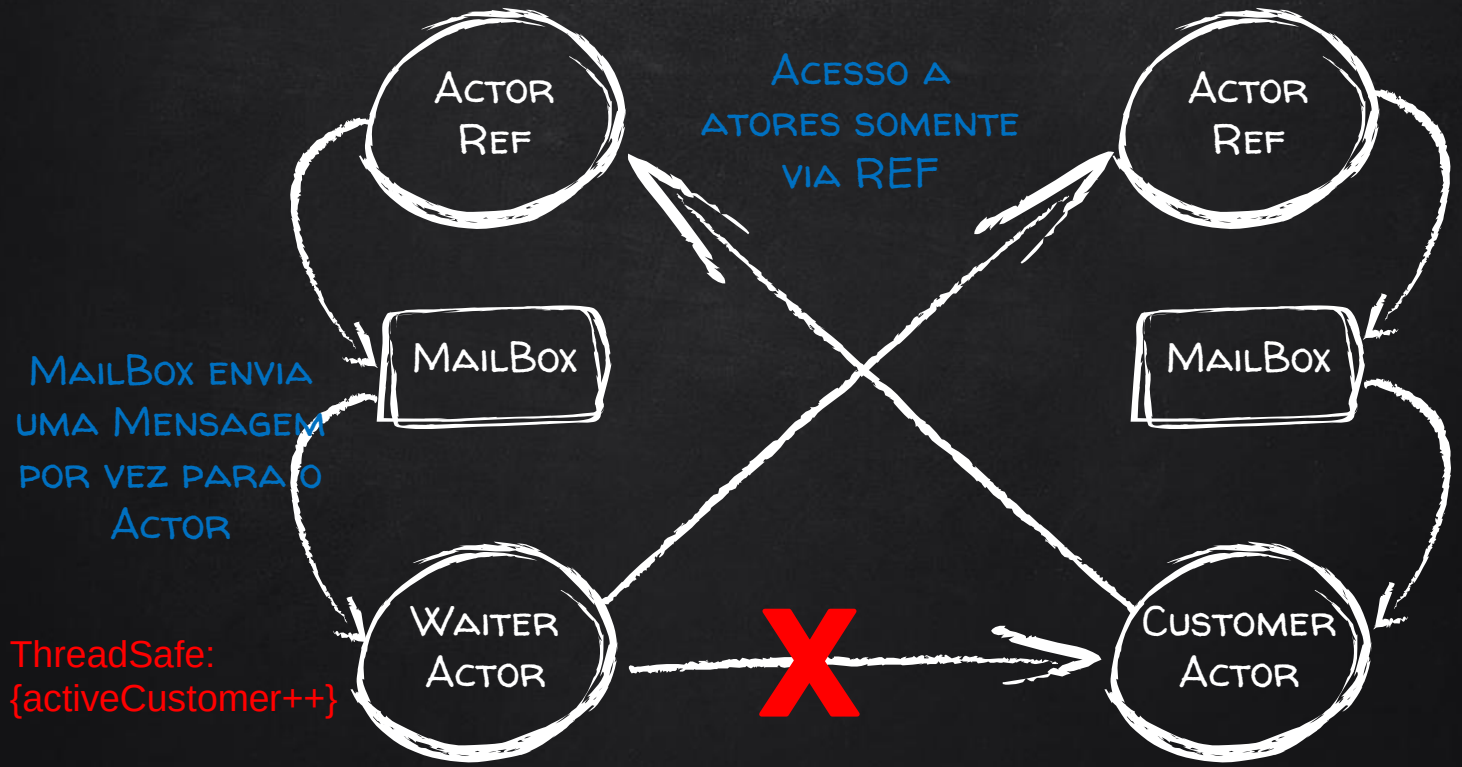
Non Blocking

FOFO – Faster Order Faster Out

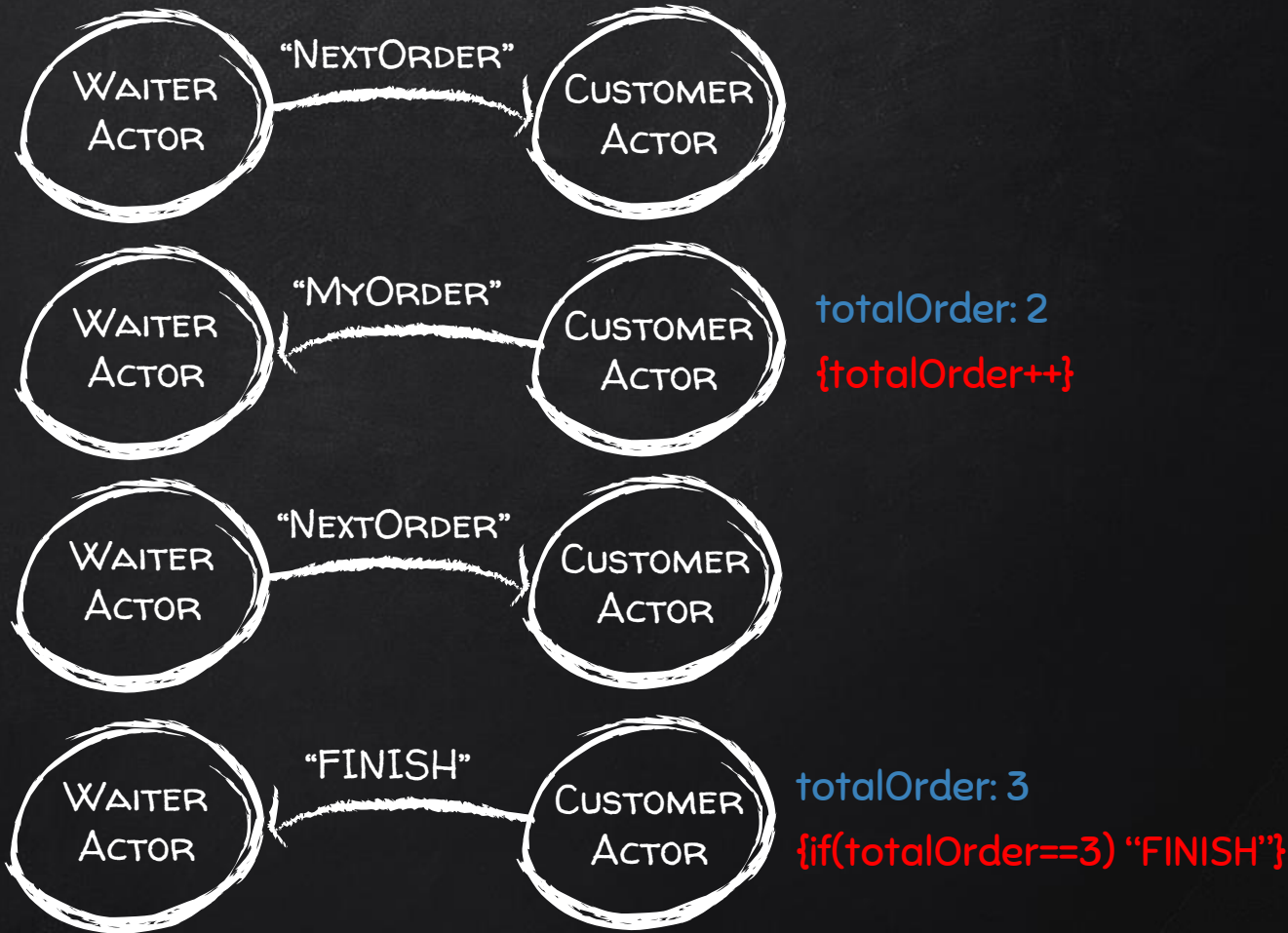


Multithreaded programming

Concorrência com ACTOR



Actor toma decisão a partir do "Estado"



HANDS ON – ACTORSYSTEM

```
ActorSystem system= ActorSystem.create("SubwaySystem");

ActorRef waiter =
system.actorOf(Props.create(WaiterActor.class), "waiter");

int delayToChoose = 10000;

waiter.tell(new NewSandwich("Slow Client", delayToChoose),
ActorRef.noSender());

waiter.tell(new NewSandwich("Fast Client", 0), ActorRef.noSender());
```

HANDS ON – Comunicação entre Atores

```
public class CustomerActor extends AbstractActor {

@Override
public Receive createReceive() {
     return receiveBuilder().//
          matchEquals(EnumOrder.NEXT_ORDER, order -> {
              doOrder();
      }).build();
}
```

```
public class WaiterActor extends AbstractActor {

@Override
public Receive createReceive() {
return receiveBuilder()//
     .match(Order.class, order -> {
          totalOrders++;
          log("Processando pedido "+order.getSalad());
          getSender().tell(EnumOrder.NEXT_ORDER, getSelf());
```

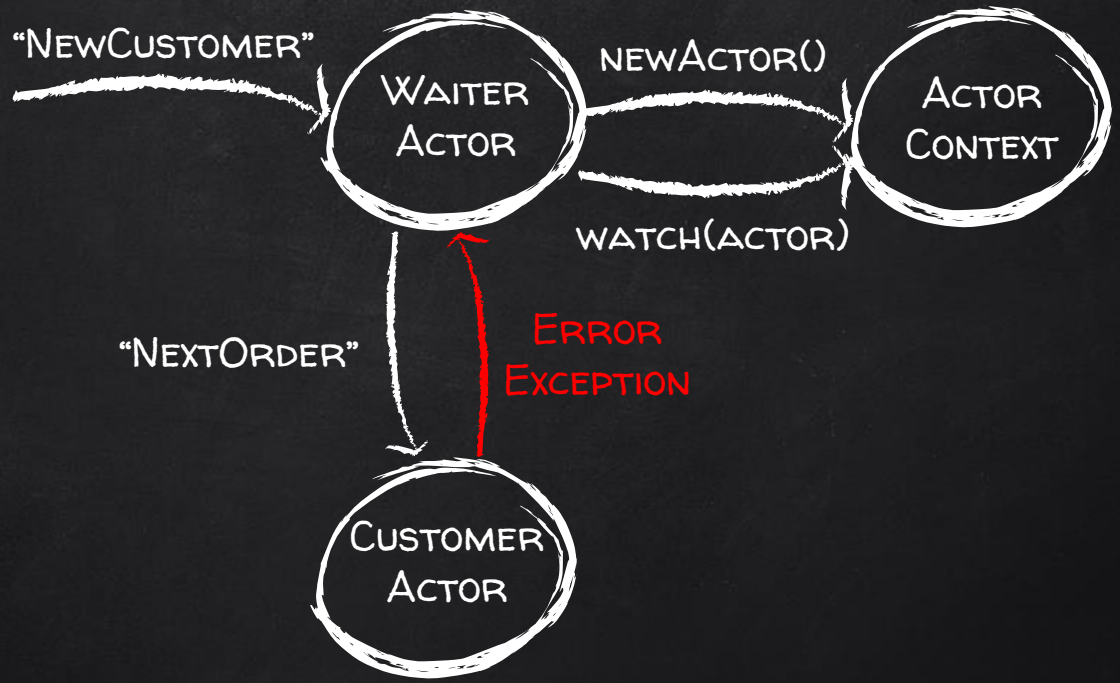
Projeto Subway

- ~~X FIFO – Fila Blocking~~
- X Exception – Cliente Desmaiou
- X Load – Garçom Sobrecarregado
- X Pergunta do Milhão

Catch Exception de outra Thread?

```
Try {
      New Thread(new Runnable(){
            Public void run(){
                  Throw new RuntimeException("HEY");
            }
       ).start();
}catch(Exception e){
            System.out.println("HEY EXCEPTION?")
}
```

Actor cria e supervisiona outros Actor



HANDS ON – SUPERVISORSTRATEGY

WaiterActor:

```
@Override
public SupervisorStrategy supervisorStrategy() {
return new OneForOneStrategy(10, Duration.create("1 minute"),
DeciderBuilder.match(RuntimeException.class, e -> restart())//
.matchAny(o -> escalate()).build());
}
```

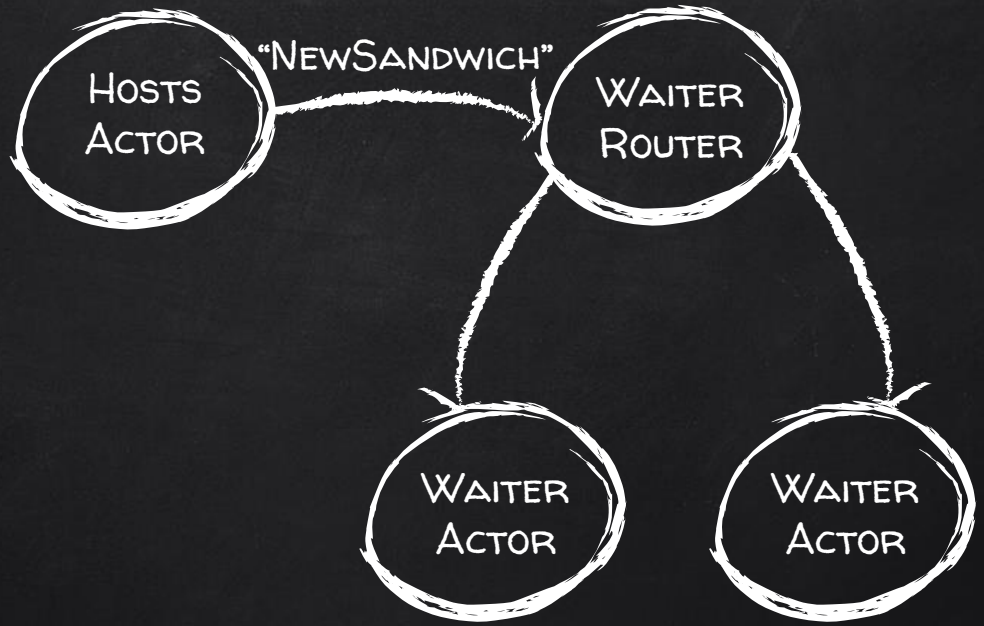
CustomerActor:

```
if (totalOrders == 1 && sandwich.isFaint()) {
   throw new RuntimeException("Problem here");
}
```

Projeto Subway

- ~~FIFO – Fila Blocking~~
- ~~Exception – Cliente Desmaiou~~
- Load – Garçom Sobrecarregado
- Pergunta do Milhão

Router


Hosts Actor
"NewSandwich"
Waiter Router
Waiter Actor
Waiter Actor

Implementações Router

- RoundRobinPool
- RandomPool
- SmallestMailBoxPool
- ConsistentHashingPool

HANDS ON – ROUTER

HostessActor:

```
private ActorRef waiters = getContext().actorOf(
          new RoundRobinPool(2).props(Props.create(WaiterActor.class))
          , "waiter");

@Override
public Receive createReceive() {
        return receiveBuilder()
                  .match(NewSandwich.class, s -> {
                        waiters.tell(s, getSelf());
        }).build();
}
```

Projeto Subway

- ~~FIFO - Fila Blocking~~
- ~~Exception - Cliente Desmaiou~~
- ~~Load - Garçom Sobrecarregado~~
- Pergunta do Milhão

Perguntas hipsters na TI

2004 – aplicação Web ou Desktop?

2008 – você usa Ajax?

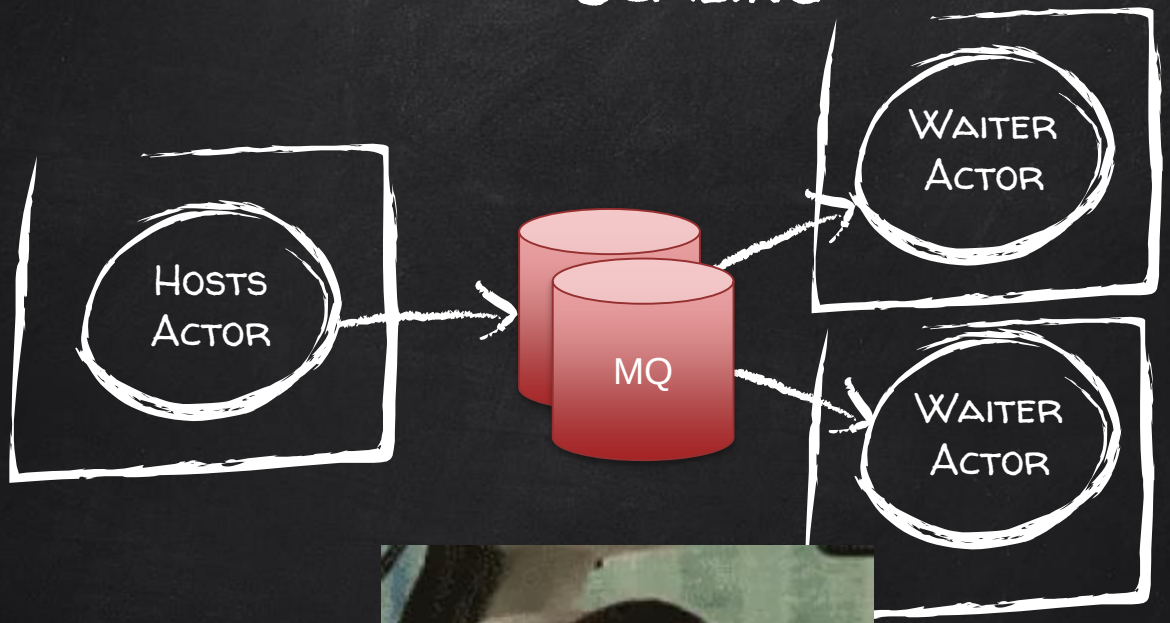
2012 – você desenvolve Mobile?

2016 – tem quantos Microservices?

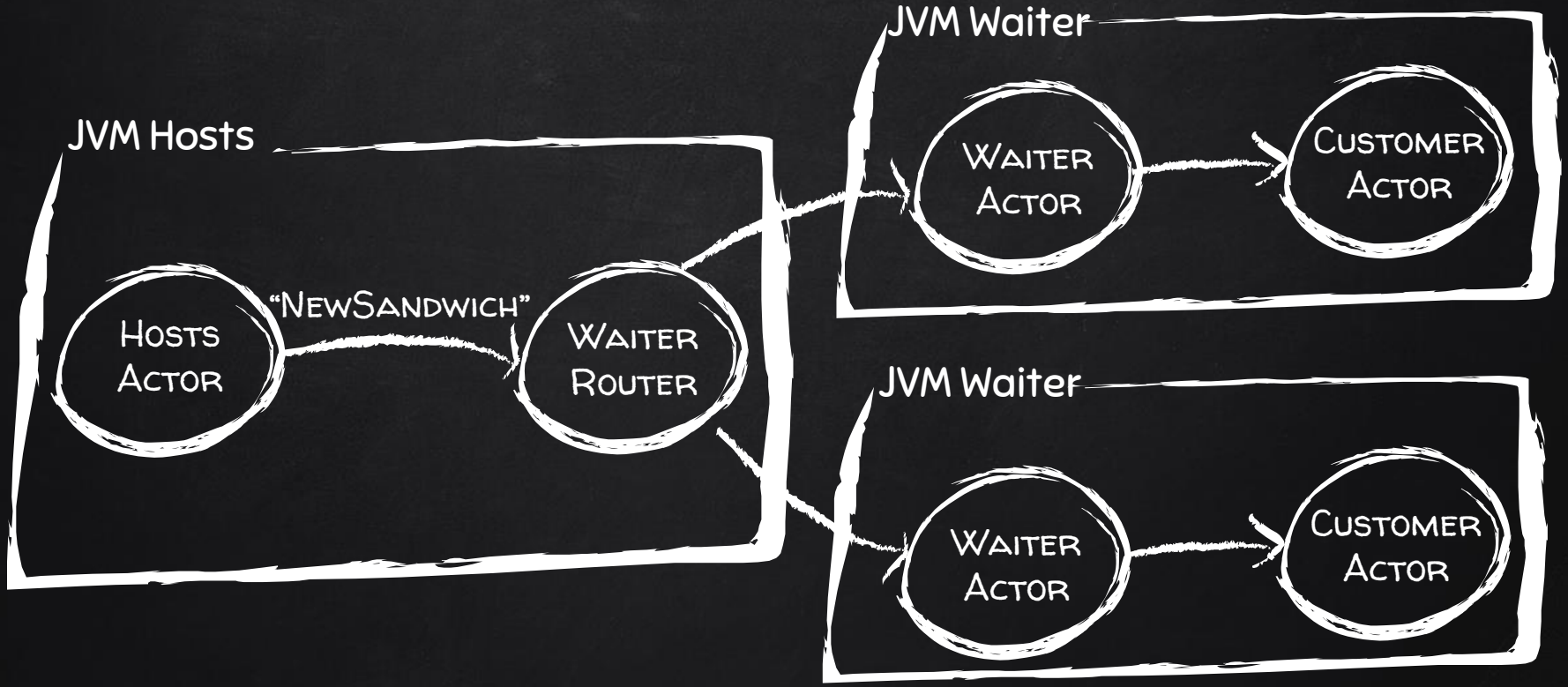
2017 – usa NoSQL?

2018 – como você escala a Aplicação?

Scaling



Clustering


JVM Hosts
Hosts Actor
"NewSandwich"
Waiter Router
JVM Waiter
Waiter Actor
Customer Actor
JVM Waiter
Waiter Actor
Customer Actor

HANDS ON – CLUSTER

```
int totalInstances = 2;
int maxInstancesPerNode = 1;
boolean allowLocalRoutees = false;
Set<String> useRoles = new HashSet<>(Arrays.asList("compute"));
ActorRef waiters = getContext()
.actorOf(
new ClusterRouterPool(new RoundRobinPool(2), new
ClusterRouterPoolSettings(totalInstances,
maxInstancesPerNode, allowLocalRoutees,
useRoles)).props(Props.create(WaiterActor.class)),
"waiterRouter");
```

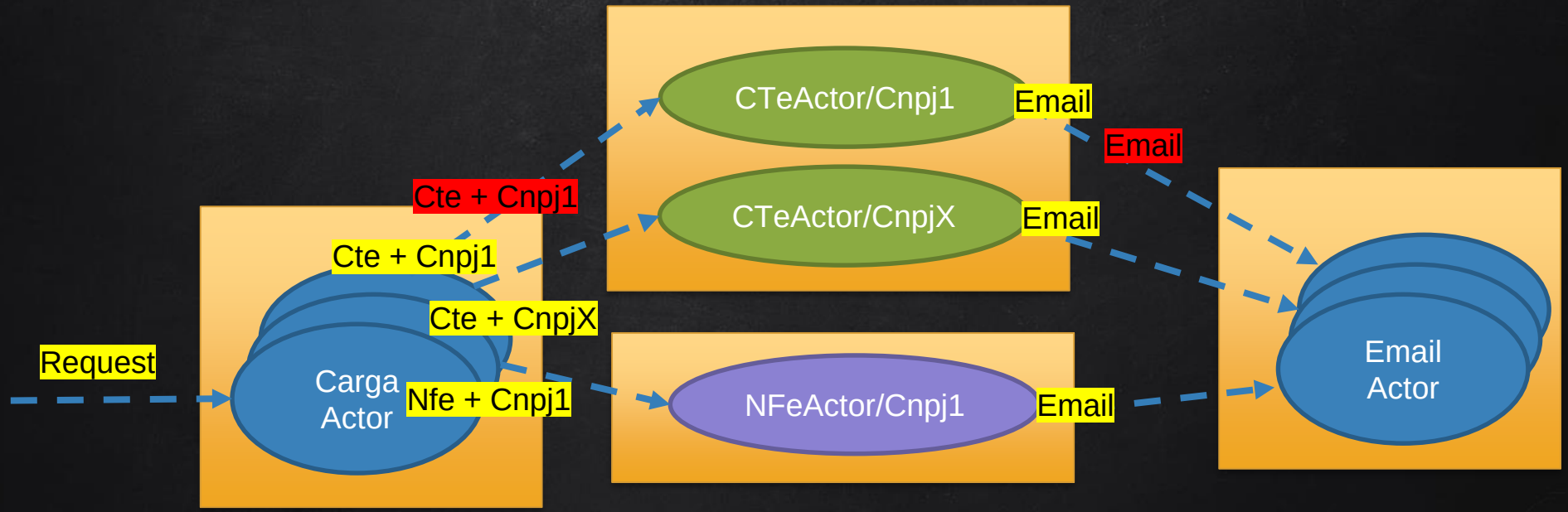
Projeto Subway

- ~~FIFO - Fila Blocking~~
- ~~Exception - Cliente Desmaiou~~
- ~~Load - Garçom Sobrecarregado~~
- ~~Pergunta do Milhão~~

Recapitulando

- ✗ Comunicação -> Non Blocking e Thread Safe
- ✗ Supervisor -> Tratativa de exception em threads diferentes
- ✗ Router -> Load Balancing do lado do cliente
- ✗ Cluster -> Todos os ítens acima de forma transparente em um ambiente distribuído

Actors são baseados em comportamento



"A good architecture will allow a system to be born as a monolith, deployed in a single file, but then to grow into a set of independently... microservices"

by Uncle Bob - Clean Architecture

Agradecimento

Obrigado Akka por me ensinar a arquitetar sistemas baseado em comportamento.

Muito Obrigado!

@fmodos