

High-Level Concurrency

Akka Actors and Java EE 7 EJBs

Dr Gerald Loeffler
Senior Integration Architect
Guidewire Software

Contents

- "Active Objects" Intro: Akka Actors, Async Stateless SBs, JMS MDBs
- Pattern 1: Local One-Way Async Request
- More Details on "Active Objects"
- Pattern 2: Request - Async Out-of-Context Response
- Pattern 3: Request - Async In-Context Response
- Commonalities and Differences

References and Bibliography

- Akka Documentation
<http://doc.akka.io/docs/akka/2.2.3/>
- Akka Concurrency, Derek Wyatt, Artima
http://www.artima.com/shop/akka_concurrency
- JSR 345: Enterprise JavaBeans, Version 3.2, EJB 3.2 Expert Group
<https://jcp.org/aboutJava/communityprocess/final/jsr345/index.html>
- JSR 343: Java Message Service (JMS) 2.0, Nigel Deakin et al.
<https://jcp.org/aboutJava/communityprocess/final/jsr343/index.html>

Common Abbreviations

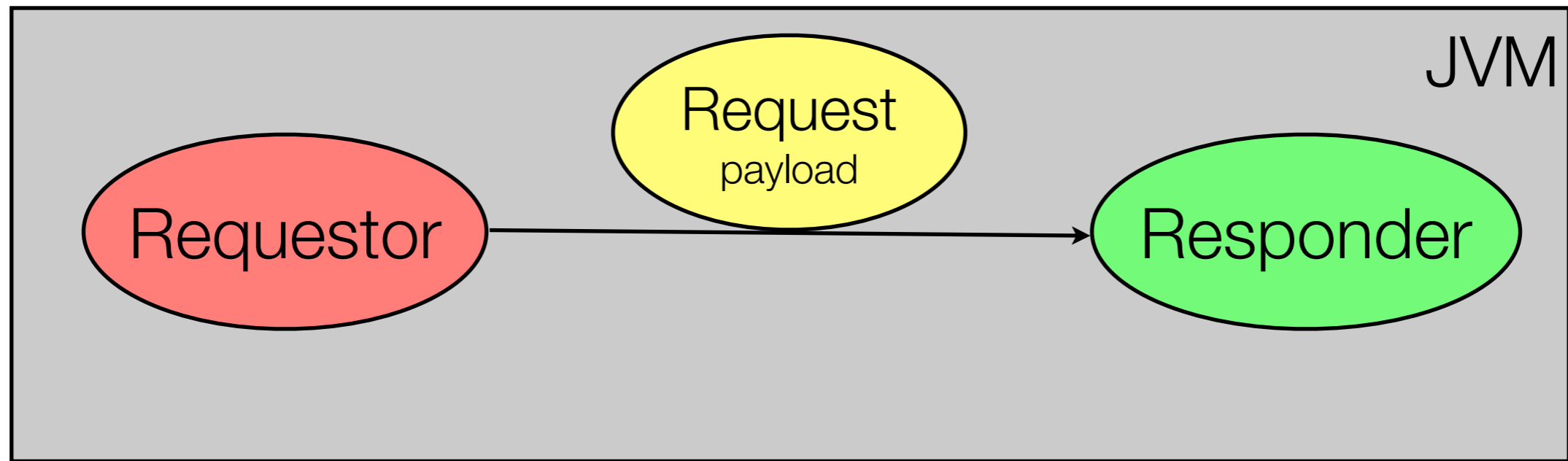
- **EJB**: Enterprise JavaBean
- **SB**: Session Bean
- **MDB**: Message-Driven Bean
- **JMS**: Java Message Service
- **JNDI**: Java Naming and Directory Interface
- **msg**: message
- **sync, async**: synchronous, asynchronous
- **DI**: Dependency Injection
- **TX**: transaction

Enterprise JavaBeans and Akka

- Enterprise JavaBeans (EJB) 3.2
 - bean types: SBs (stateful, stateless, singleton), MDBs
 - part of Java Platform, Enterprise Edition 7 (Java EE 7) approved in 2013
 - standard (specification, TCK, reference implementation)
 - certified implementations (GlassFish 4, TMAX JEUS 8; soon: JBoss/WildFly 8, WebSphere AS, WebLogic, ...)
- Akka
 - 2.2.3 (current in Feb 2014), 1.0 released in 2011)
 - toolkit/framework/platform (code, documentation)

Simple Mental Models of "Active Objects"

- All these "active objects" execute asynchronously on threads typically from a pool, with serialized access to the same instance (exception: Singleton SBs)
- **Actors**
 - receive messages as arbitrary objects (*Any*), enqueued in a mailbox
 - managed explicitly, addressed individually via common interface
- **Async Stateless SBs**
 - expose async methods to be invoked by clients
 - container-managed, pooled, addressed by type via their own interface
- **JMS MDBs**
 - receive messages of pre-defined types from a JMS queue/topic
 - container-managed, pooled, invisible to clients, addressed via queue/topic



Local One-Way Async Request

JVM-local, async send of *Request* from *Requestor* to *Responder* (who doesn't actually respond at this stage)

Actor

- Start/stop actor system
- Start/stop each actor instance explicitly, plus lifecycle hooks
- Supervision hierarchy
- Custom message type
- Send (*tell, !*) and *receive* operate on *Any*
- actor state accessed single-threadedly
- Message ordering guarantee for any pair of actors

```
object Responder {
  case class Request(payload: String)
}
class Responder extends Actor {
  val log = Logging(context.system, this)

  override def receive = {
    case Request(payload) => log info s"received $payload"
  }
}

class Requestor extends Actor {
  val resp = context.actorOf Props[Responder]

  override def preStart() {
    resp ! Request("first")
    resp ! Request("second")
  }
  override def receive = Actor.emptyBehavior
}

object Bootstrap extends App {
  val sys = ActorSystem("TheActorSystem")
  val req = sys.actorOf Props[Requestor]
  Thread.sleep(1000)
  Await.result(gracefulStop(req, 1 minute), 1 minute)
  sys.shutdown
}
```


Async Stateless SB

- Start/stop external (container-managed), communicated by lifecycle event callbacks
- Undeclared number of *Responder* instances, one *Requestor* instance per JVM
- Async and sync methods could be mixed in same SB
- DI of (sub-class of) *Responder* into *Requestor*
- No exception delivery to client
- TX demarcation, security context propagation

```
@Stateless
class Responder {
    val log = Logger getLogger classOf[Responder].getName

    @Asynchronous
    def request(payload: String) {
        log info s"received $payload"
    }
}

@Startup @Singleton
class Requestor {
    @EJB var resp: Responder = _

    @PostConstruct
    def sendRequests() {
        resp request "first"
        resp request "second"
    }
}
```

JMS MDB

- Generic message types
- *Requestor* and *Responder* decoupled by *requestQueue*
- XA-capable transactional send/receive to/from queue
- DI of queue as "administered object" via JNDI name
- DI of container-managed *JMSContext*
- Message ordering guarantee for *Session-Destination* pairs (Session hidden in *JMSContext*, one Session per Connection)

```
import javax.ejb.{ ActivationConfigProperty => ACP }
import javax.jms.{ JMSDestinationDefinition => JMSDD }

@MessageDriven(activationConfig = Array(
  new ACP(propertyName = "destinationLookup",
    propertyValue = "java:global/requestQueue")))
class Responder extends MessageListener {
  val log = Logger getLogger classOf[Responder].getName

  override def onMessage(req: Message) {
    val pl = req.asInstanceOf[TextMessage].getText
    log info s"received $pl"
  }
}

@JMSDD(name = "java:global/requestQueue",
  interfaceName = "javax.jms.Queue",
  destinationName = "requestQueue")
@Startup @Singleton
class Requestor {
  val log = Logger getLogger classOf[Requestor].getName

  @Inject var jmsc: JMSContext = _

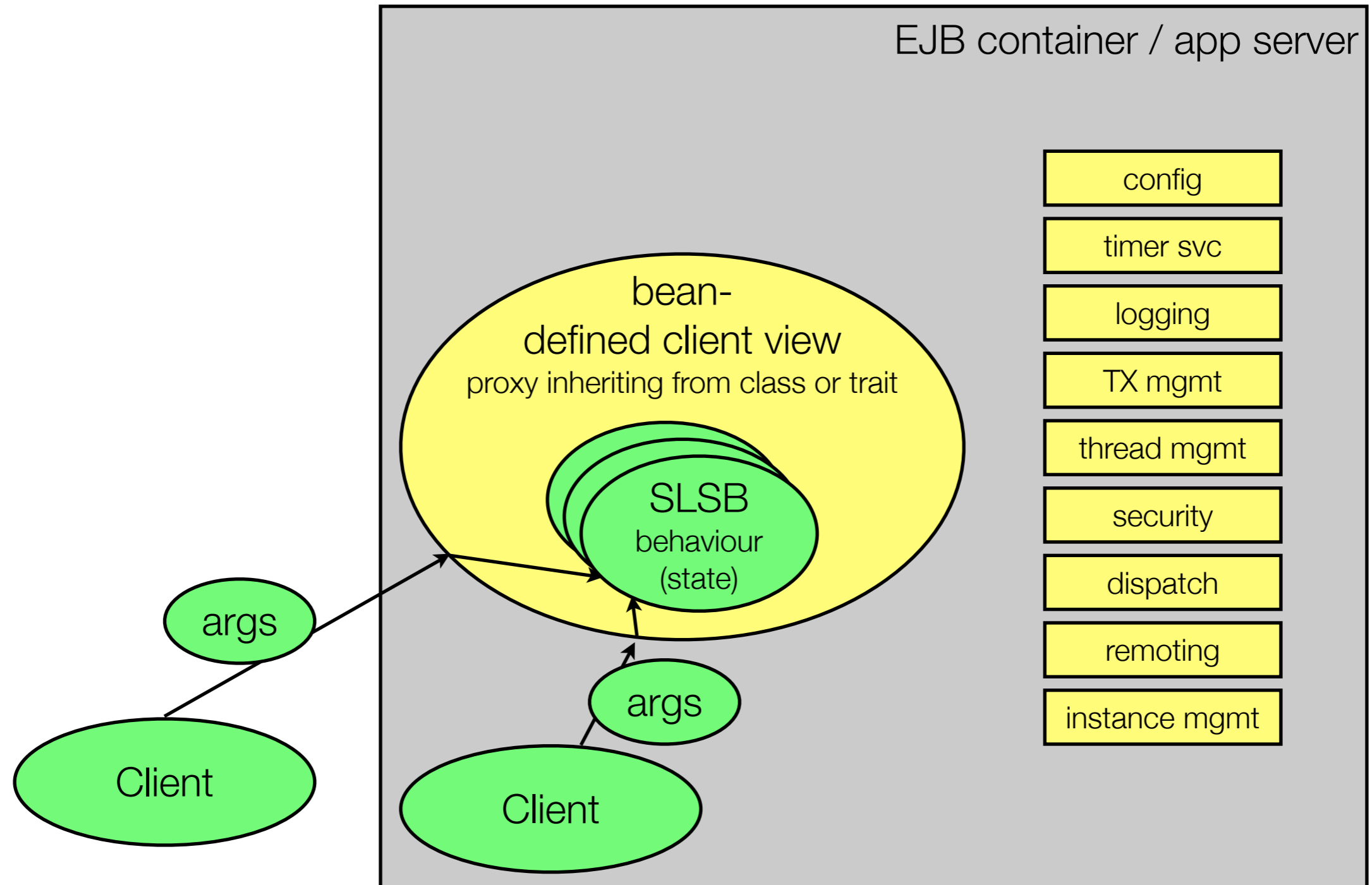
  @Resource(lookup = "java:global/requestQueue")
  var reqQ: Queue = _

  @PostConstruct
  def sendRequests() {
    jmsc.createProducer()
      .send(reqQ, "first")
      .send(reqQ, "second")
  }
}
```

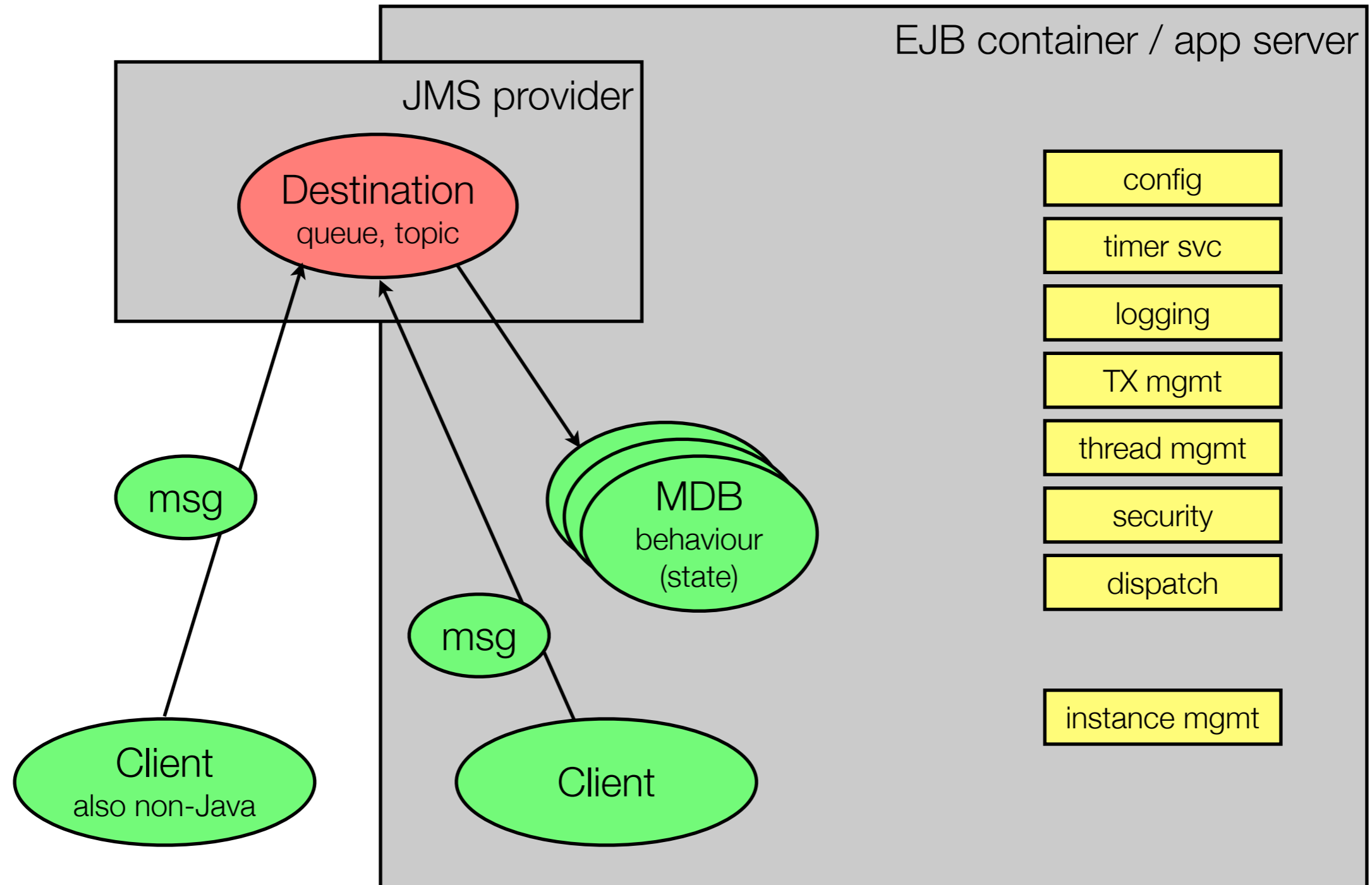
Slightly More Detailed Models of "Active Objects"

Async Stateless Session Beans, JMS Message Driven Beans, Actors

Visual Model of Async Stateless SBs



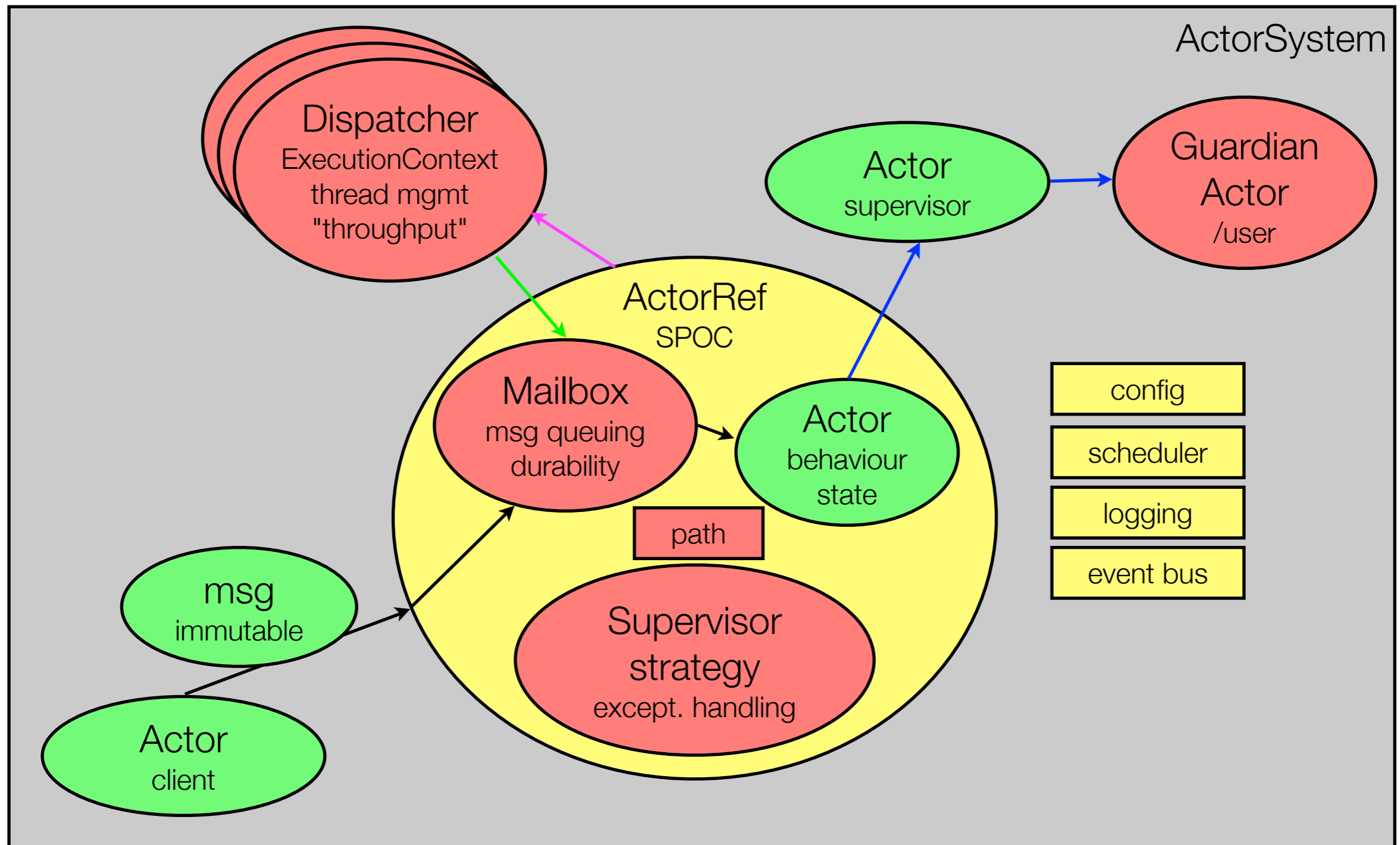
Visual Model of JMS MDBs



EJB Exception Handling

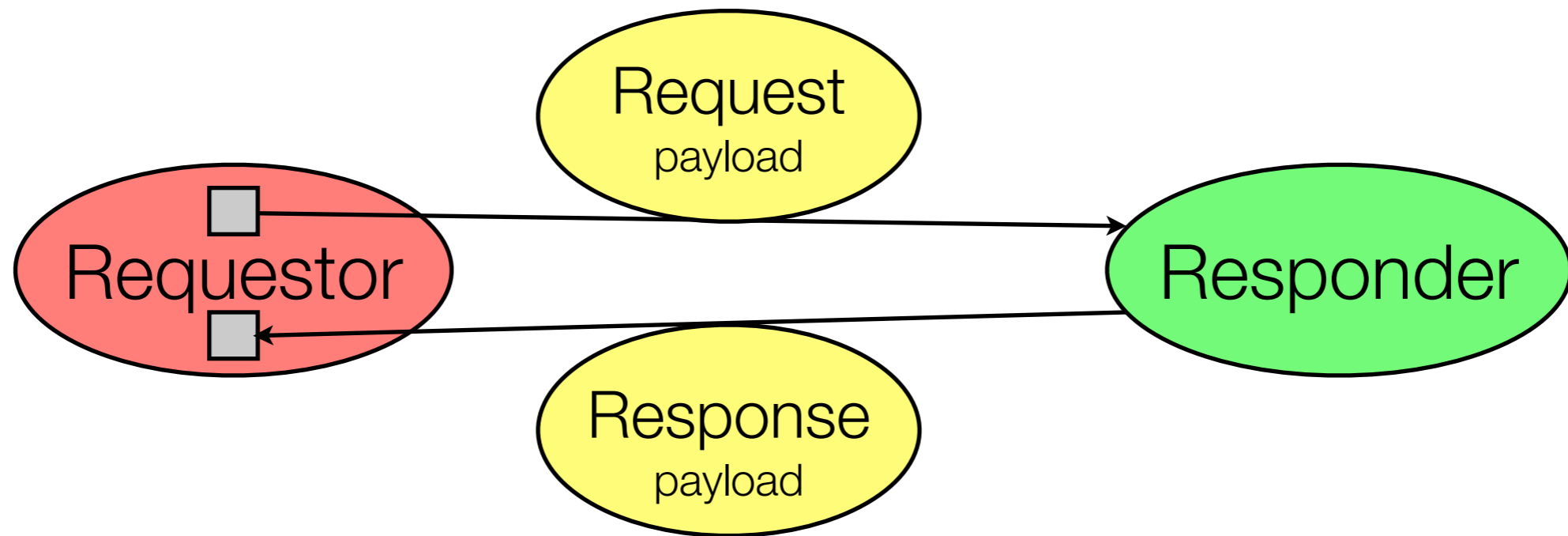
- Defined by the EJB spec: uniform across apps
- Exception thrown from EJB methods are caught by the container, reacted upon, and possibly re-thrown (possibly wrapped in *EJBException* or similar)
- Distinction by exception type between
 - **application exceptions** (checked or *@ApplicationException*): TX will optionally be rolled back, keep EJB instance, re-throw exception to client
 - **system exceptions** (non-application unchecked or *RemoteException*): log, TX will be rolled back, discard EJB instance, throw wrapped exception to client

Visual Model of Actors



Actor Supervision

- A form of exception handling - inherent part of the actor programming model
- Every Actor has a parent; each parent is the supervisor of all its children, receiving notifications of their exceptions
- Through its *SupervisorStrategy*, a parent has these options, decided upon based on exception type: **resume/restart/stop** child or **escalate** exception
- Actor resume/restart/stop propagate to hierarchy of children (overridable via *preStart()*, *preRestart()*, *postRestart()* and *postStop()* callbacks)
- Actor instances may thus change transparently behind an *ActorRef* (for restart), or *ActorRef* may become obsolete (for stop)
- Currently processed msg is normally lost upon exception, but mailbox (and messages therein) survives restart



Request - Async Out-of-Context Response

In *Requestor*, sending of *Request* and async receipt of the corresponding *Response* occur in different contexts: upon receipt of *Response*, the *Requestor* must typically correlate that *Response* back to the original *Request* to re-establish (parts of) that context.

Actor

- *sender* holds *ActorRef* of sender of currently received msg
- Otherwise like all other actor-actor message passing

```
object Responder {
  case class Request(payload: String)
}
class Responder extends Actor {
  override def receive = {
    case Request(p) => sender ! Response(p, p.reverse)
  }
}

object Requestor {
  case class Response(in: String, out: String)
}
class Requestor extends Actor {
  val log = Logging(context.system, this)
  val resp = context.actorOf Props[Responder]

  override def preStart() {
    resp ! Request("first")
    resp ! Request("second")
  }
  override def receive = {
    case Response(in, out) => log info s"received $out for $in"
  }
}
```

Async Stateless SB

- Mutual DI of collaborators
- No concept of "caller EJB"
- Otherwise like all other async SB method calls
- Note `@Asynchronous` on bean classes: applies to all public methods and hence requires hiding of Scala-generated accessor methods with *private*

```
@Asynchronous @Stateless
class Responder {
  @EJB private var req: Requestor = _

  def request(p: String) {
    req.respond(p, p.reverse)
  }
}

@Asynchronous @Startup @Singleton
class Requestor {
  private val log = Logger.getLogger("Requestor")

  @EJB private var resp: Responder = _

  @PostConstruct
  private def sendRequests() {
    resp.request("first")
    resp.request("second")
  }

  def respond(in: String, out: String) {
    log.info(s"received $out for $in")
  }
}
```

JMS MDB

- Request-reply directly supported by *JMSReplyTo* and *JMSCorrelationID* msg headers
- Hence *Requestor* may determine response destination (queue) (or *Responder* may use DI to get reference to queue)
- Async msg listener such as *Requestor* (listening for responses) must be MDB
- Otherwise like normal JMS MDB messaging

```
import javax.ejb.{ ActivationConfigProperty => ACP }

@MessageDriven(activationConfig = Array(
  new ACP(propertyName = "destinationLookup",
    propertyValue = "java:global/requestQueue")))
class Responder extends MessageListener {
  @Inject var jmsc: JMSContext = _

  override def onMessage(req: Message) {
    val pl = req.asInstanceOf[TextMessage].getText
    val corID = req.getJMSCorrelationID
    val resQ = req.getJMSReplyTo

    jmsc.createProducer().setJMSCorrelationID(corID).send(resQ, pl.reverse)
  }
}

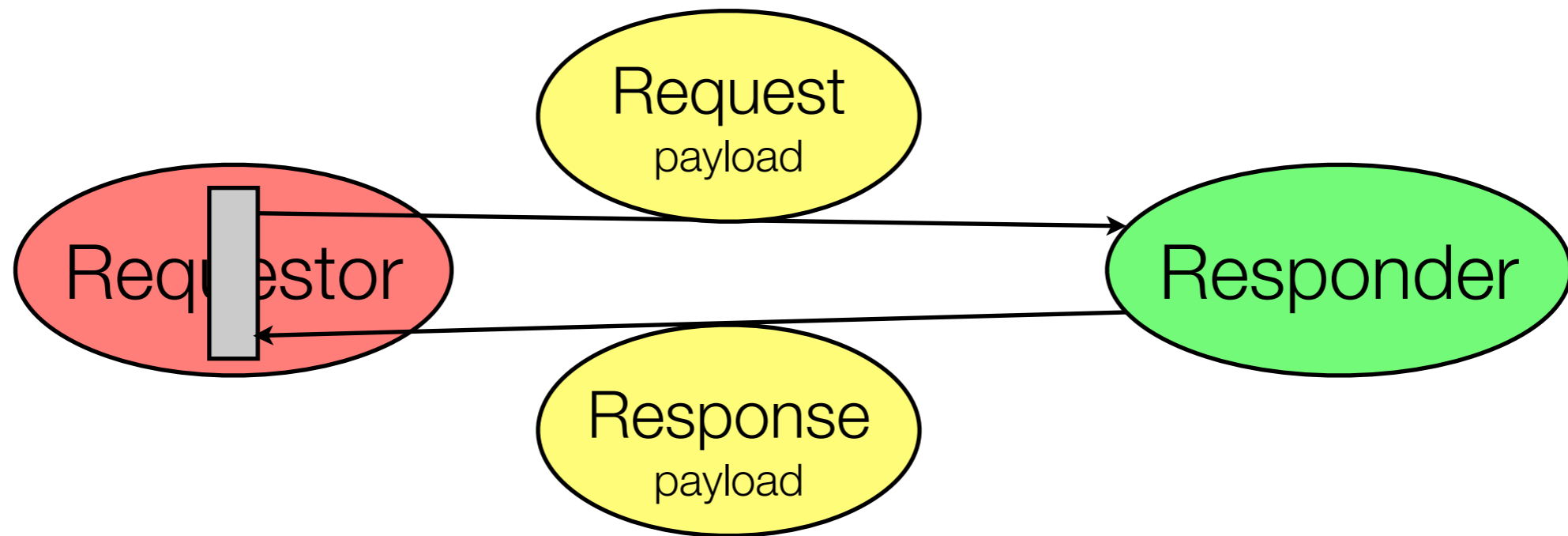
@MessageDriven(activationConfig = Array(
  new ACP(propertyName = "destinationLookup",
    propertyValue = "java:global/responseQueue")))
class Requestor extends MessageListener {
  val log = Logger.getLogger(classOf[Requestor].getName)

  @Inject var jmsc: JMSContext = _

  @Resource(lookup = "java:global/requestQueue")
  var reqQ: Queue = _
  @Resource(lookup = "java:global/responseQueue")
  var resQ: Queue = _

  @PostConstruct
  def sendRequests() {
    jmsc.createProducer().setJMSReplyTo(resQ)
      .setJMSCorrelationID("first").send(reqQ, "first")
      .setJMSCorrelationID("second").send(reqQ, "second")
  }

  override def onMessage(resp: Message) {
    val in = resp.getJMSCorrelationID
    val out = resp.asInstanceOf[TextMessage].getText
    log.info(s"received $out for $in")
  }
}
```



Request - Async In-Context Response

In *Requestor*, sending of *Request* and asynchronous receipt of the corresponding *Response* occur in the same context: upon receipt of *Response*, the *Requestor*, if desired, has immediate access to all aspects of the *Request* and the state of the system when that *Request* was sent.

Actor

- *Responder* as before
- *Requestor* sending *Request* with *?* (*ask*) to retrieve *Future* to *Response* (as *Future[Any]*), which may time out
- Allows in-context installation of *onSuccess* callback, which needs an *ExecutionContext*
- Context for *Response* delivery and processing is non-persistent, may be lost, leaving orphaned *Responses*

```
object Responder {
  case class Request(payload: String)
}
class Responder extends Actor {
  override def receive = {
    case Request(p) => sender ! Response(p, p.reverse)
  }
}

object Requestor {
  case class Response(in: String, out: String)
}
class Requestor extends Actor {
  val log = Logging(context.system, this)
  val resp = context.actorOf Props[Responder]

  override def preStart() {
    import context.dispatcher // EC for onSuccess
    implicit val to = Timeout(10 seconds) // for ?
    resp ? Request("first") onSuccess {
      case Response(in, out) => log info s"1st: $out"
    }
    resp ? Request("second") onSuccess {
      case Response(in, out) => log info s"2nd: $out"
    }
  }
  override def receive = Actor.emptyBehavior
}
```

Async Stateless SB

- Method *request* returns result as Java *Future*, either successfully completed with *AsyncResult* or failed by throwing exception (exception delivered to client)
- *Requestor* receives Java *Future*, but can only enquire completion and block on getting result (possibly timing out)
- Scala *Future* is entirely distinct, but can be used to async retrieve results from Java *Future* (increasing blocked thread count)

```
@PostConstruct
private def sendRequests() {
    val f1 = resp request "first"
    val f2 = resp request "second"

    import scala.concurrent.ExecutionContext.Implicits.global
    future { log info s"1st: " + f1.get }
    future { log info s"2nd: " + f2.get }
}
```

```
@Asynchronous @Stateless
class Responder {
    def request(p: String): Future[String] = {
        new AsyncResult(p.reverse)
    }
}
```

```
@Startup @Singleton
class Requestor {
    val log = Logger getLogger "Requestor"

    @EJB var resp: Responder = _

    @PostConstruct
    private def sendRequests() {
        val f1 = resp request "first"
        val f2 = resp request "second"
        log info s"1st: " + f1.get
        log info s"2nd: " + f2.get
    }
}
```

JMS MDB

- *Responder* MDB as before
- To receive responses in-context, *Requestor* cannot use MDB's *onMessage*
- Sync msg receive blocks EJB thread (with 10s timeout): not recommended
- Msg send happens only at TX commit: receive must be from separate later TX (REQUIRES_NEW)
- each *JMSContext* instance is TX-scoped

```
@Startup @Singleton
class Requestor {
    val log = Logger getLogger classOf[Requestor].getName

    @EJB var helper: RequestorHelper = _

    @PostConstruct
    @TransactionAttribute(NOT_SUPPORTED)
    def sendRequests() {
        helper send "first"
        helper send "second"

        var out1 = helper recv; log info s"1st: $out1"
        var out2 = helper recv; log info s"2nd: $out2"
    }
}
@Singleton
@TransactionAttribute(REQUIRES_NEW)
class RequestorHelper {
    @Inject var jmsc: JMSContext = _

    @Resource(lookup = "java:global/requestQueue")
    var reqQ: Queue = _
    @Resource(lookup = "java:global/responseQueue")
    var resQ: Queue = _

    def send(p: String) {
        jmsc.createProducer().setJMSReplyTo(resQ)
            .setJMSCorrelationID(p).send(reqQ, p)
    }
    def recv() = {
        jmsc.createConsumer(resQ).receiveBody(classOf[String], 10000)
    }
}
```


Summary and Comparison

Actors, Async Stateless Session Bean, JMS Message Driven Beans

Commonalities and Differences (1/2)

- **Application and container lifecycle:** app server, app, deployment
- **"Active object" instance lifecycle:** start/stop, number, identity, callbacks
- **Client contract:** type (*ActorRef*, class/trait, *Destination*), semantics, DI
- **Message-passing interface:** method signature, custom/generic msg type
- **Message dispatch:** method call, message handler
- **Message queuing:** mailbox, queue/topic, internal
- **Thread management:** thread pool, instance-thread allocation, dispatcher
- **"Active Object" configuration:** instantiation/implementation, code/config

Commonalities and Differences (2/2)

- **Services:** load-balancing, failover, clustering, security, transactions
- **Failure handling:** instance removal, message retry, transaction rollback
- **Adaptability:** configurability, extensibility, standard/proprietary
- **Interoperability:** network protocols, plugins/extensions

Out of Scope

- Remoting
- Akka cluster and specific Java EE app server cluster features
- Details of Akka paths
- Akka and app server-specific configuration
- EJB deployment descriptors, environment entries and JNDI
- Actor death watch
- Akka scheduler and EJB timer service details
- Design patterns

Thank You

High-Level Concurrency: Akka Actors and Java EE 7 EJBs

Dr Gerald Loeffler
Senior Integration Architect
Guidewire Software

Additional Material

Typed Actors, Remoting, Singleton SBs

Typed Actor

- JDK Proxy for trait *Responder* around hidden actor delegating to *ResponderImpl*
- All specific messages sent through method calls
- Generic Actor interaction (arbitrary messages, lifecycle hooks) through implementing pre-defined traits

```
trait Responder {
  def request(payload: String)
}
class ResponderImpl extends Responder {
  val log = Logger getLogger classOf[Responder].getName

  def request(payload: String) {
    log info s"received $payload"
  }
}

class Requestor extends Actor {
  val resp = TypedActor(context).typedActorOf(
    TypedProps(classOf[Responder], new ResponderImpl))

  override def preStart() {
    resp request "first"
    resp request "second"
  }
  override def receive = Actor.emptyBehavior
}

object Bootstrap extends App {
  // as before
}
```

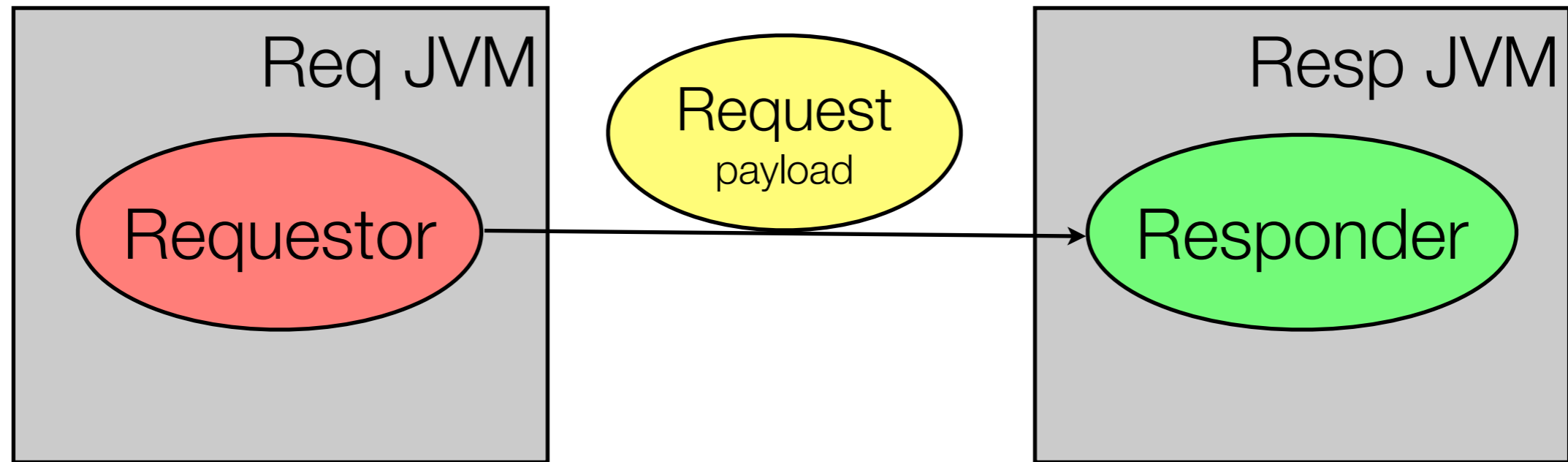
Typed Actor

- Returning a *Future* from a method in the typed actor trait corresponds to using *?/ask* on an untyped *Actor*
- But this *Future* can have a specific type parameter, whereas *?/ask* returns *Future[Any]*
- Method *request* must return *Future* of a successful completed or failed *Promise*.

```
trait Responder {
  def request(payload: String): Future[String]
}
class ResponderImpl extends Responder {
  def request(p: String) = Future.successful(p.reverse)
}

class Requestor extends Actor {
  val log = Logger getLogger classOf[Requestor].getName
  val resp = TypedActor(context).typedActorOf(
    TypedProps(classOf[Responder], new ResponderImpl))

  override def preStart() {
    import context.dispatcher // EC for onSuccess
    resp request "first" onSuccess {
      case out => log info s"1st: $out"
    }
    resp request "second" onSuccess {
      case out => log info s"2nd: $out"
    }
  }
  override def receive = Actor.emptyBehavior
}
```

Remote One-Way Async Request

Inter-JVM, typically also across the network, async send of *Request* from *Requestor* to *Responder* (who doesn't actually respond at this stage)

Actor

- Actor systems (each confined to its JVM and identified by name@host:port) can be federated. Fine-grained events inform about state-changes.
- Configurable: transport (Netty, optionally over SSL), mutual authentication, "serializers" (Google protocol buffers, ...), ...
- Instantiation of actor in remote actor system using *actorOf* by mapping local actor path (*/resp*) in *application.conf* to remote actor path (*akka://Resp@host:2552/user/resp*)
- Messages must be "serializable" (*ActorRefs* are "serializable")
- Or (preferred) look-up remote actor (see next slide)

```
class Requestor extends Actor {  
  val resp = context.actorOf(Props[Responder], "resp")  
  
  // as before  
}
```

Typed Actor

- Remotely create actor (see previous slide), or (preferred):
- In local actor system wrap typed actor proxy around the hidden actor instantiated in the remote actor system
- Look-up of remote actor with *actorFor* and remote path
- *Responder* arguments and return types must be "serializable"
- A remotely looked-up actor may also act as a factory for actors in that (remote) actor system by accepting messages containing *Props/TypedProps*

```
class Requestor extends Actor {  
  val resp = TypedActor(context).typedActorOf(  
    TypedProps[Responder],  
    context actorFor "akka://Resp@host:2552/user/resp")  
  
  // as before  
}
```

Async Stateless SB

- Remote business interface/trait, DI thereof
- *Requestor's* deployment descriptor maps ejb-ref for injection point (*my.package.Requestor/resp* under *java:comp/env*) to *Responder's* (remote) JNDI name (app server specific)
- Works also locally, ensuring pass-by-value semantics
- *Responder* arguments and return types must be serializable
- Remote dispatch done asynchronously
- No longer guaranteed interoperability (CORBA/IIOP) across app server vendors

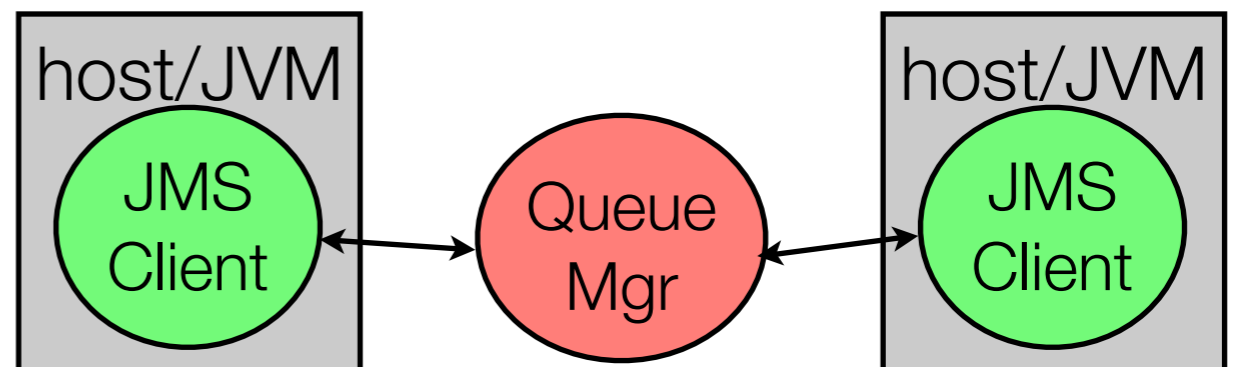
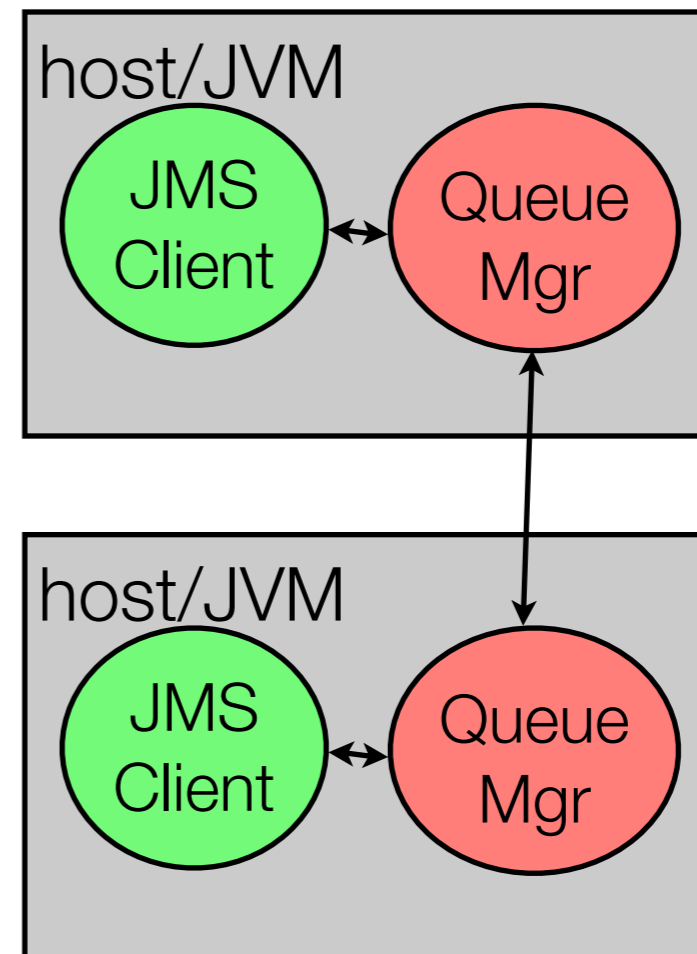
```
@Remote
trait Responder {
  def request(payload: String)
}
@Stateless
class ResponderBean extends Responder {
  // as before
}

@Startup @Singleton
class Requestor {
  @EJB var resp: Responder = _

  // as before
}
```

JMS MDB

- JMS inherently remotable
- No difference in code, but in configuration, performance and failure modes
- JMS provider ("queue manager", "broker") can be co-located (even JVM-local) or remote to JMS "client" (sender/receiver of messages)
 - It participates in TXs!
- "Network of brokers" vs. "hub-and-spoke" architecture



Visual Model of Async Singleton SBs

