Contemporary Concurrency Comparison

Akka Actors, Scala 2.10 Futures and Java EE 6 EJBs

Dr Gerald Loeffler
Senior Integration Architect
Guidewire Software

Contents

- "Active Objects": Akka Actors, Async SBs, JMS MDBs
- Local One-Way Async Request
- Remote One-Way Async Request
- Request - Async Out-of-Context Response
- Request - Async In-Context Response
- Commonalities and Differences

References and Bibliography

- Akka Documentation
  http://doc.akka.io/docs/akka/2.1.4/
- Akka Concurrency, Derek Wyatt, Artima
  http://www.artima.com/shop/akka_concurrency
- JSR 318: Enterprise JavaBeans,Version 3.1, EJB 3.1 Expert Group
  http://jcp.org/aboutJava/communityprocess/mrel/jsr318/index.html

Common Abbreviations

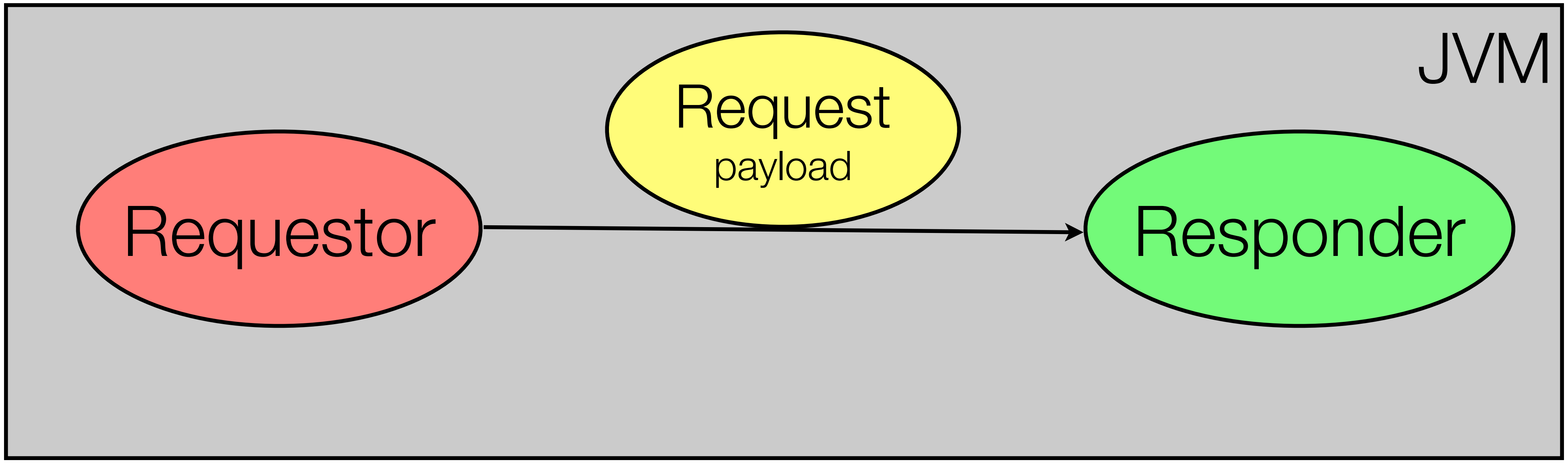
- **EJB**: Enterprise JavaBean
- **SB**: Session Bean
- **MDB**: Message-Driven Bean
- **JMS**: Java Message Service
- **JNDI**: Java Naming and Directory Interface
- **msg**: message
- **sync**, **async**: synchronous, asynchronous
- **DI**: Dependency Injection
- **TX**: transaction

Enterprise JavaBeans and Akka

- Enterprise JavaBeans (EJB) 3.1
  - bean types: SBs (stateful, stateless, singleton), MDBs
  - part of Java Platform, Enterprise Edition 6 (Java EE 6) approved in 2009 (Java EE 7: April 2013)
  - standard (specification, TCK, reference implementation)
  - certified implementations (JBoss AS/WildFly, GlassFish, WebSphere AS, WebLogic, ...)
- Akka
  - 2.1.4 (current in May 2013), 1.0 released in 2011)
  - toolkit/framework/platform (code, documentation)

Simple Mental Models of "Active Objects"

- All our "active objects" execute asynchronously on threads typically from a pool, with serialized access to the same instance (but: Singleton SBs)
- **Actors**
  - receive messages as arbitrary objects (*Any*), enqueued in a mailbox
  - addressed individually, via common or own (**Typed Actors**) interface
- `**Async (Stateless and Singleton) Session Beans**
  - expose async methods to be invoked by clients
  - are addressed by type, not by instance, via their own interface
- **JMS MDBs**
  - receive messages of pre-defined types from a JMS queue/topic
  - are invisible to clients, only reachable 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*
- Message ordering guarantee for any pair of actors

```
object Responder {
  case class Request(payload: String)
}
class Responder extends Actor {
  val log = Logging(context.system, Responder.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 {
  implicit val sys = ActorSystem("TheActorSystem")
  val req = sys actorOf Props[Requestor]
  Thread.sleep(1000)
  Await.result(gracefulStop(req, 1 minute), 1 minute)
  sys shutdown
}
```

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

```scala
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
}
```

Async Session Bean

- Start/stop external, 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
- 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

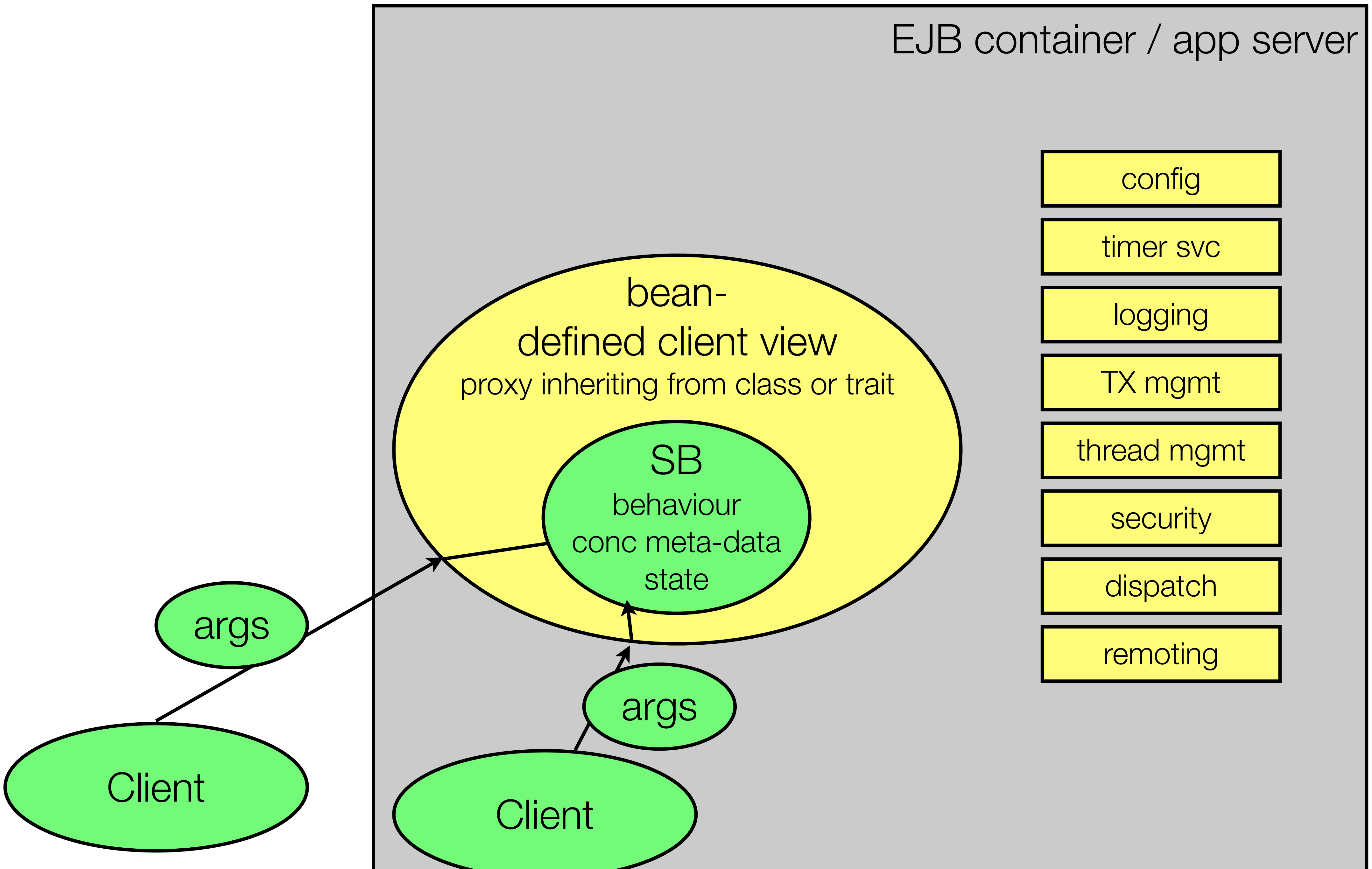
- Generic message type
- *Requestor* and *Responder* decoupled by *requestQueue*
- XA-capable transactional send/receive to/from queue
- Security context handling
- DI via JNDI names of "administered objects"
- Message ordering guarantee for *Session-Destination* pairs
- JMS 2.0 (Java EE 7) greatly simplifies message sending

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

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

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

@Startup @Singleton
class Requestor {
  @Resource(lookup = "java:/JmsXA")
  var cf: ConnectionFactory = _

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

  @PostConstruct
  def sendRequests() {
    val conn = cf.createConnection()
    val sess = conn.createSession(true, 0)
    val prod = sess.createProducer(reqQ)
    try {
      prod.send(sess createTextMessage "first")
      prod.send(sess createTextMessage "second")
    } finally {
      prod.close()
      sess.close()
      conn.close()
    }
  }
}
```

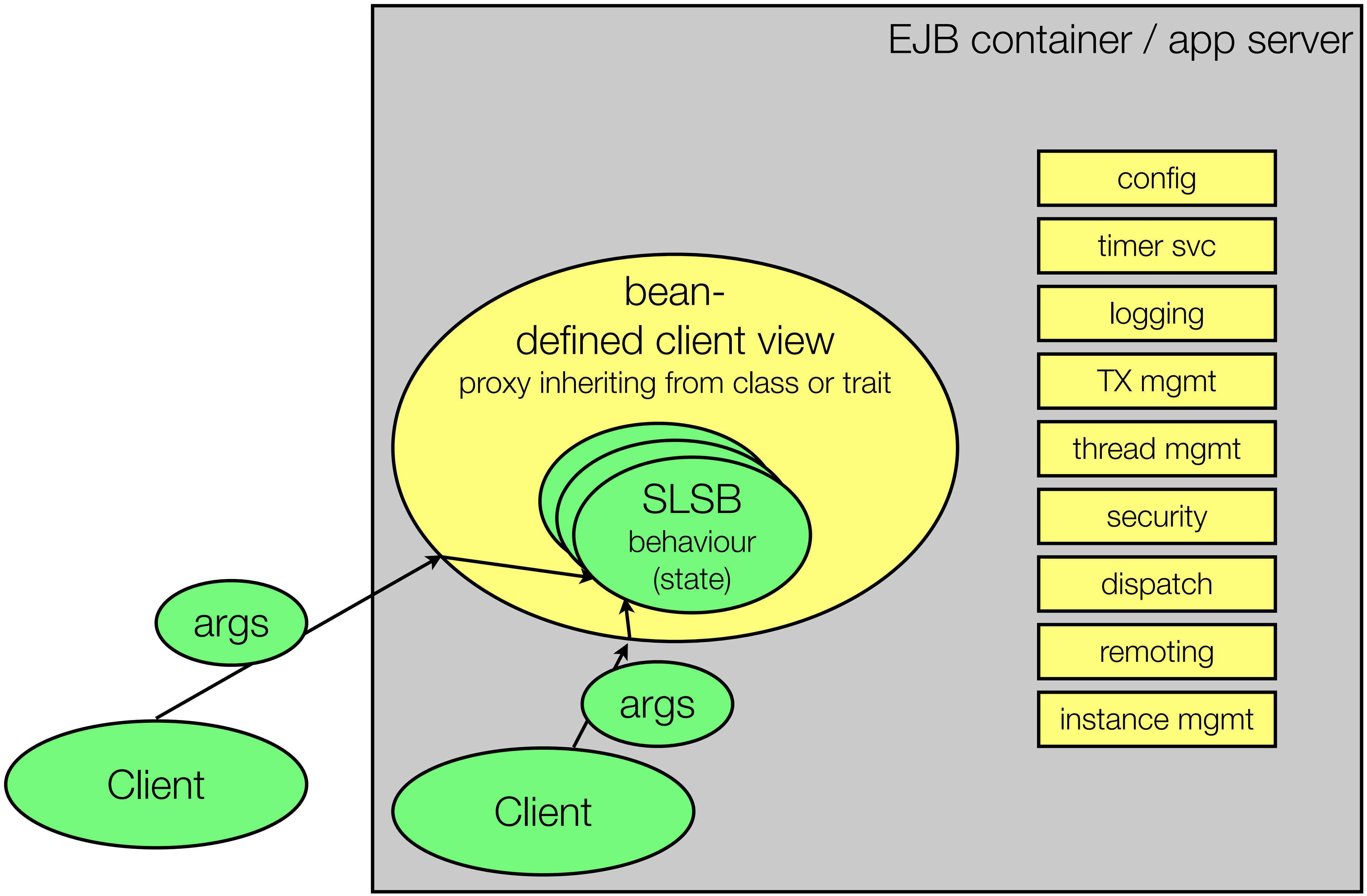
Slightly More Detailed Models of "Active Objects"

Async Singleton SBs, Async SLSBs, JMS MDBs, Actors

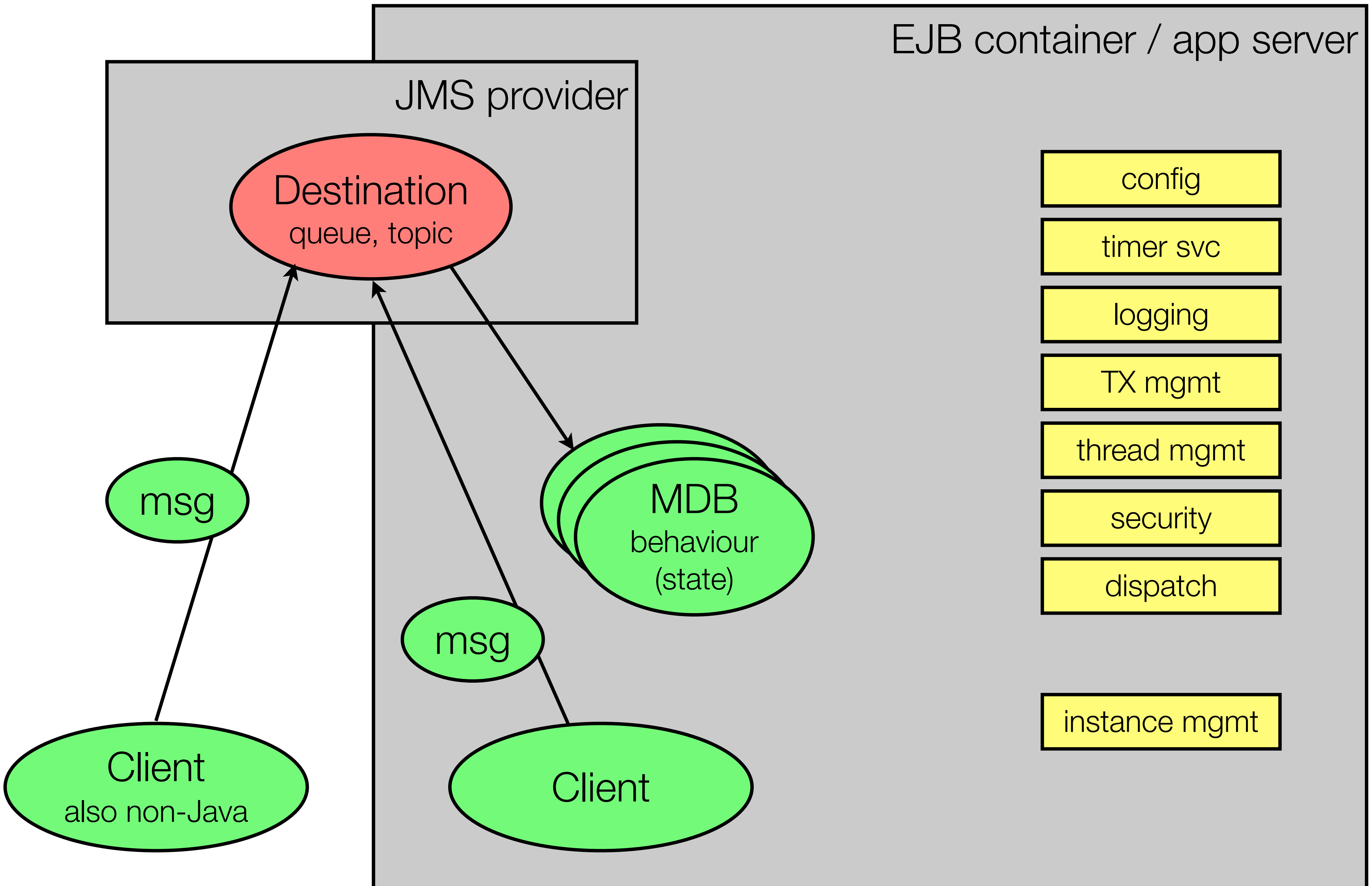
Visual Model of Async Singleton SBs



Visual Model of Async SLSB



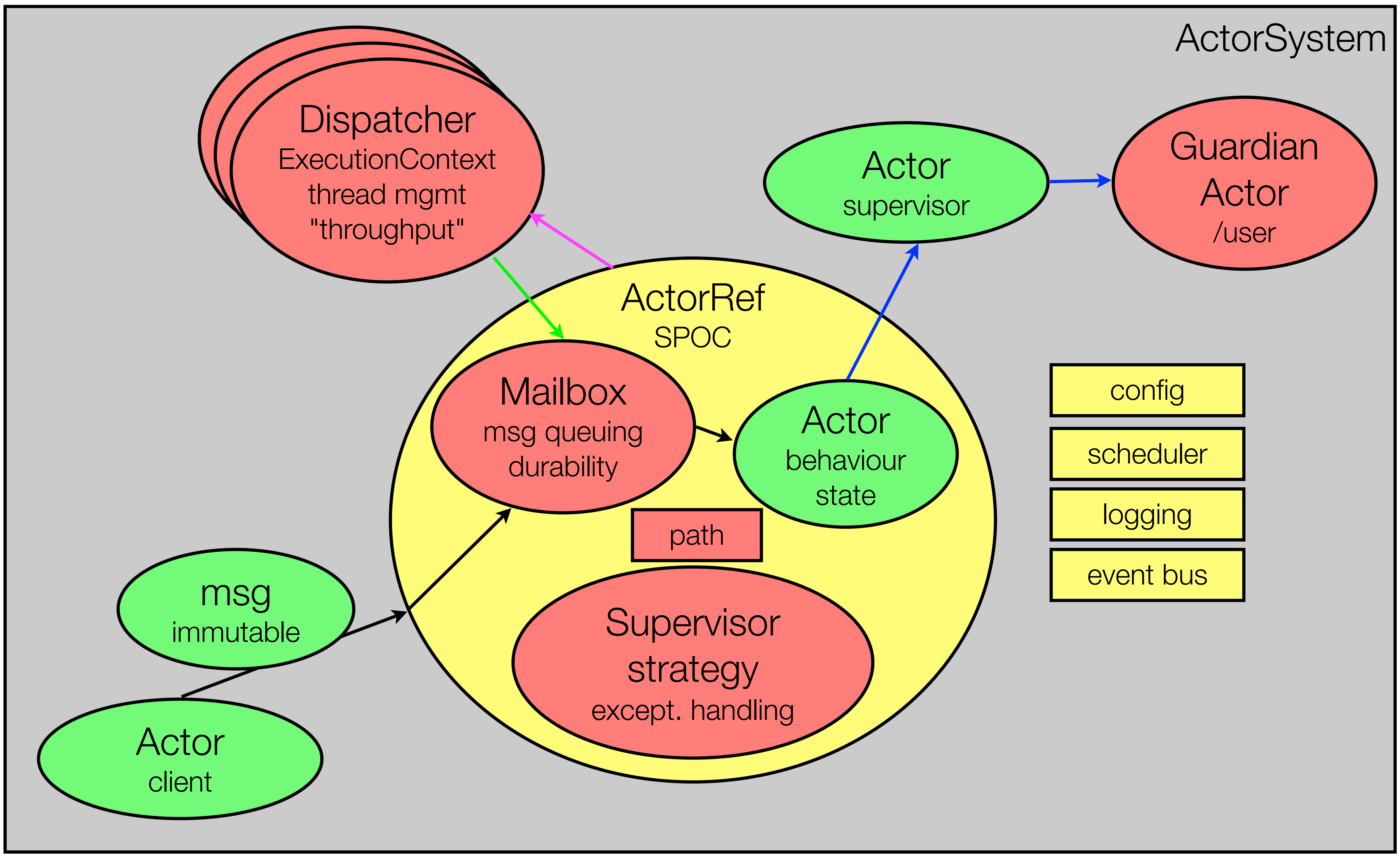
Visual Model of JMS MDBs



EJB Exception Handling

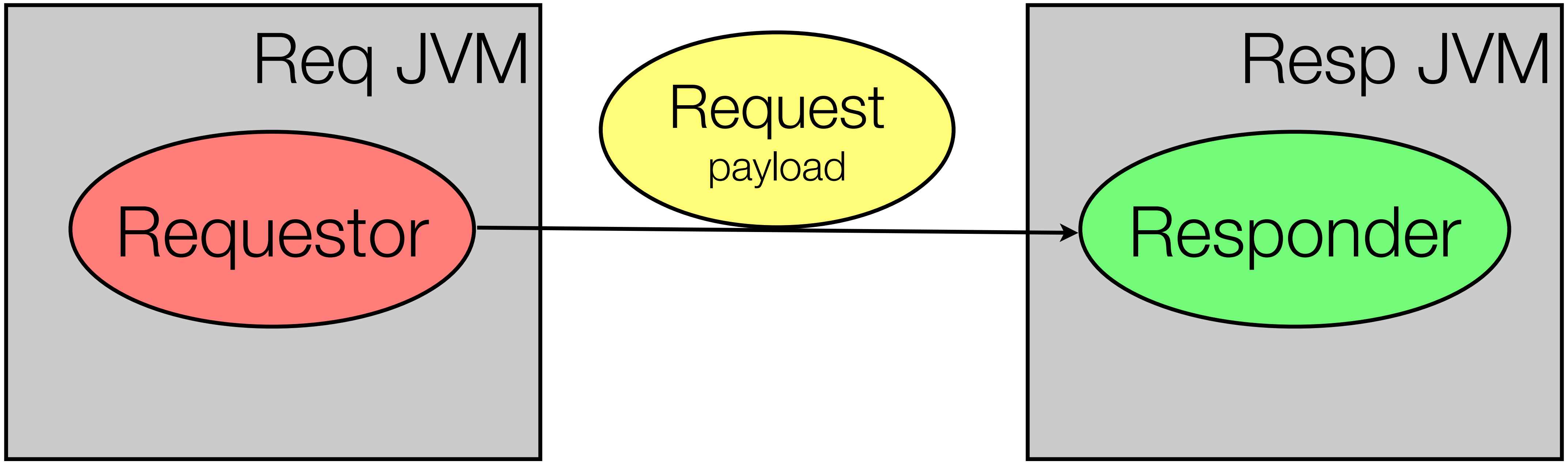
- Defined by the EJB spec
- Distinction by exception type between application exceptions (and whether they cause TX rollback or not) and system exceptions
- Exception thrown from EJB methods are caught by the container, reacted upon, and possibly re-thrown (possibly wrapped in *EJBException* or similar)
- Possible actions by container upon catching an exception:
  - Mark current TX for rollback, or rollback current TX, or commit current TX
  - Discard EJB instance
  - Log exception

Visual Model of Actors



Actor Supervision

- An inherent part of the actor programming model, with no direct EJB parallel
- 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 its hierarchy of children (overridable via *preStart()*, *preRestart()*, *postRestart()* and *postStop()* callbacks)
- Actor instances may thus change behind an *ActorRef*
- Currently processed msg is normally lost upon exception, but mailbox (and messages therein) survives restart



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" (*ActorRef*s 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
}
```

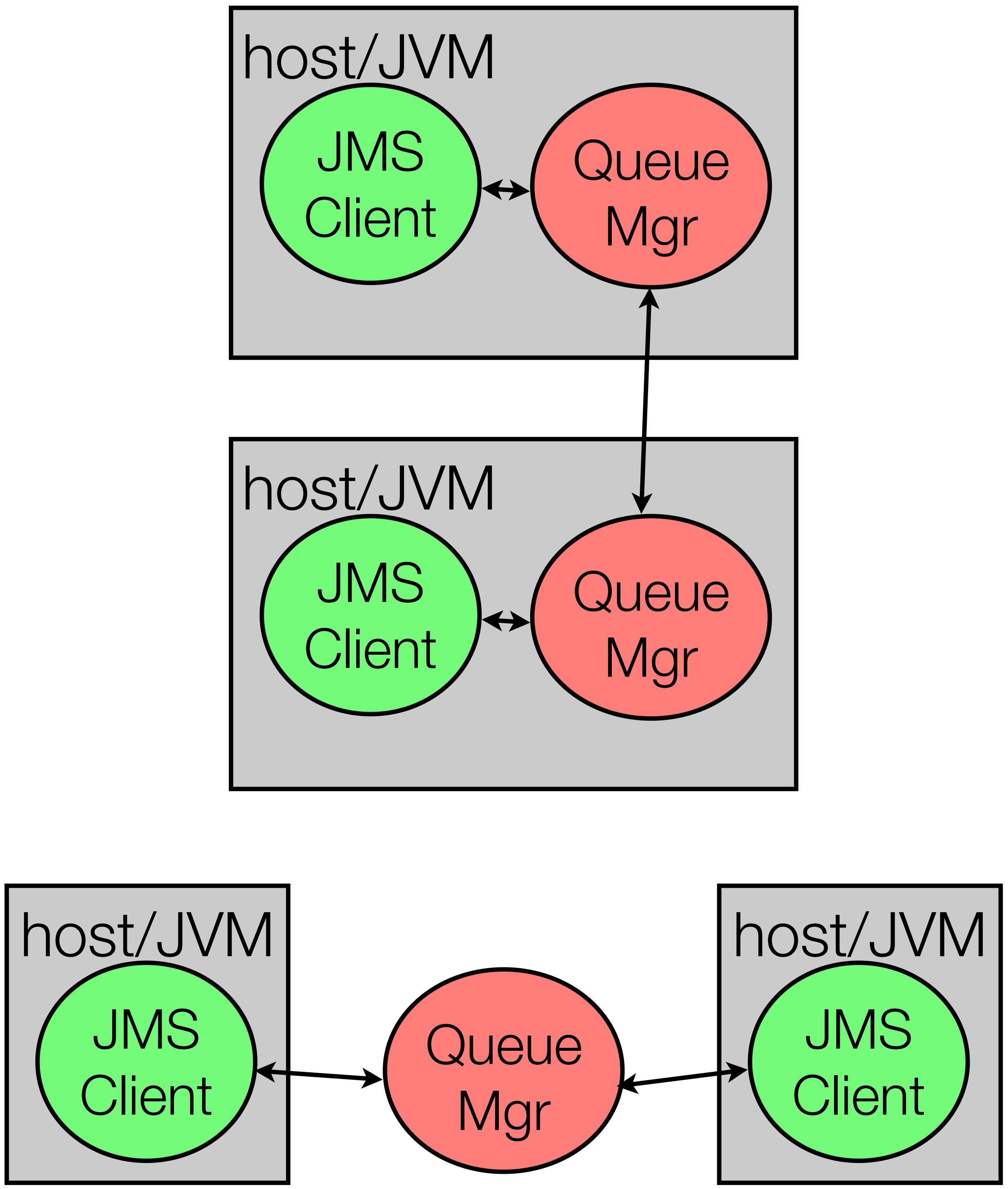
Async Session Bean

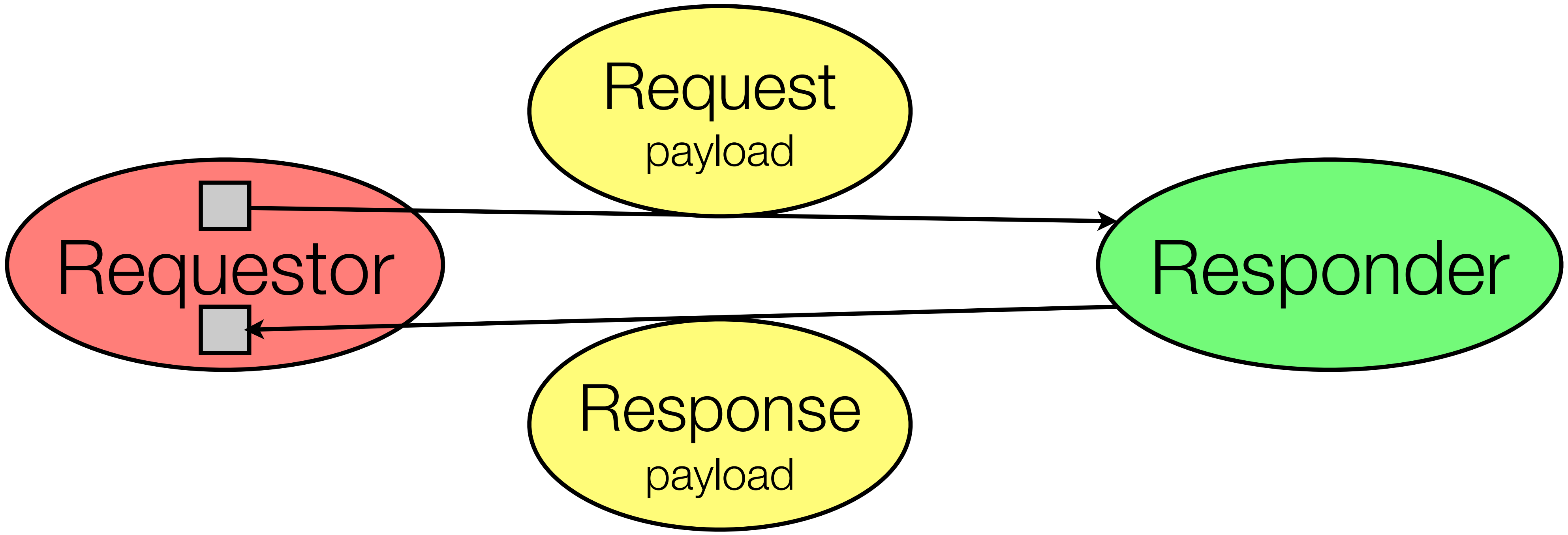
- Remote business interface/trait, DI thereof
- *Requestor*'s deployment descriptor maps ejb-ref for injection point (*my.package.Requestor/resp* under *java:comp/env*) to *Responder*'s (remote) JNDI name (app server specific)
- Works also locally, ensuring pass-by-value semantics
- *Responder* arguments 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





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 Session Bean

- 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(payload: String) {
    req.respond(payload, payload.reverse)
  }
}

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

  @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 (1/2)

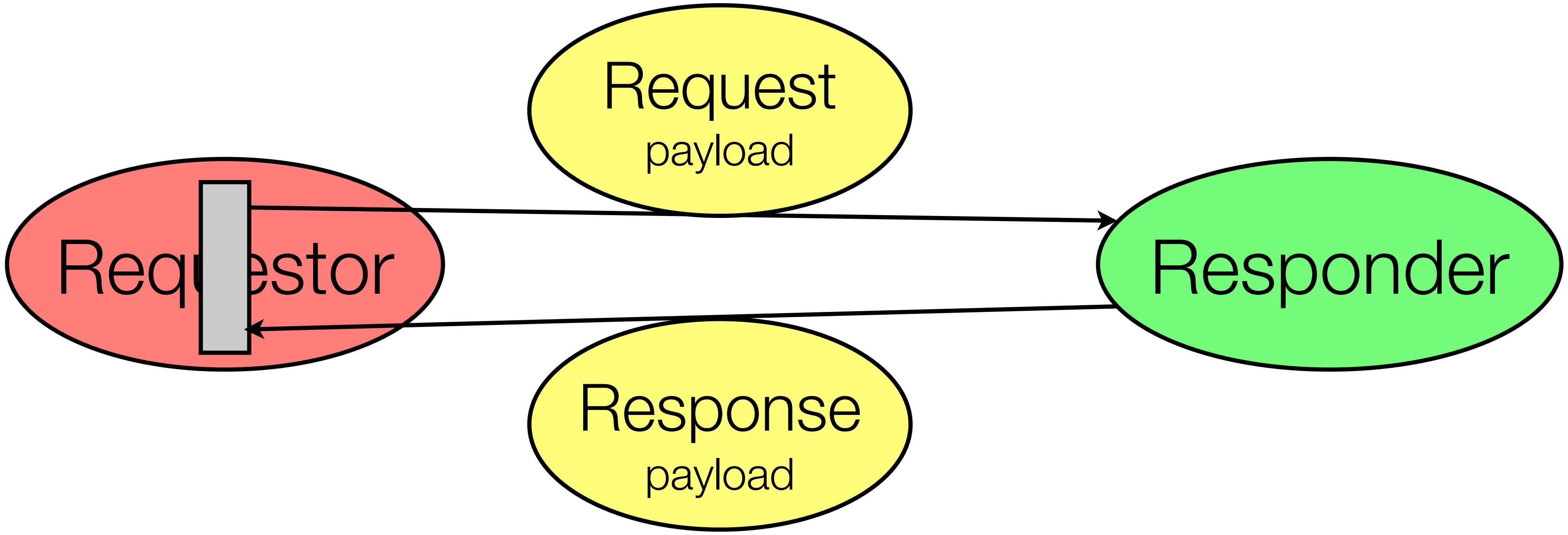
```scala
import javax.ejb.{ ActivationConfigProperty => ACP }

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

  @Resource(lookup = "java:/JmsXA")
  var cf: ConnectionFactory = _

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

  @Schedule(second = "0,20,40", minute = "*", hour = "*")
  def sendRequests() {
    // ...
  }
  override def onMessage(response: Message) {
    val in = response getJMSCorrelationID
    val out = response.asInstanceOf[TextMessage].getText
    log info s"received $out for $in"
  }
}
```

```scala
def sendRequests() {
  val conn = cf.createConnection()
  val sess = conn.createSession(true, 0)
  val prod = sess.createProducer(reqQ)

  def createAndSendMsg(text: String) {
    val msg = sess createTextMessage text
    msg setJMSCorrelationID text
    msg setJMSReplyTo resQ
    prod send msg
  }
  try {
    createAndSendMsg("first")
    createAndSendMsg("second")
  } finally {
    prod.close()
    sess.close()
    conn.close()
  }
}
```

JMS MDB (2/2)

- 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[ACP](
  new ACP(propertyName = "destination",
    propertyValue = "java:/jms/queue/requestQueue")))
class Responder extends MessageListener {
  @Resource(lookup = "java:/JmsXA")
  var cf: ConnectionFactory = _

  override def onMessage(request: Message) {
    val resQ = request getJMSReplyTo
    val p = request.asInstanceOf[TextMessage].getText

    val conn = cf.createConnection()
    val sess = conn.createSession(true, 0)
    val prod = sess.createProducer(resQ)
    try {
      val msg = sess createTextMessage p.reverse
      msg setJMSCorrelationID request.getJMSCorrelationID
      prod send msg
    } finally {
      prod.close()
      sess.close()
      conn.close()
    }
  }
}
```



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 that triggered that *Request*.

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*

```scala
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
}
```

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
}
```

Async Session Bean

- Method *request* returns result as Java *Future*, either successfully completed with *AsyncResult* or failed by throwing exception (exception delivered to client)
- *Requestor* receives *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!)

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

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

  @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
  }
}
```

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

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

JMS MDB

- In-context receipt of JMS response messages is not possible solely with MDBs, as they always receive messages asynchronously, out-of-context, via the *onMessage* callback, which is invoked by the resource adapter ("app server").

- Using the JMS API directly from any EJB (including an MDB), it would theoretically be possible to create a *MessageConsumer* and use it to do a blocking response *Message receive* in-context after having sent the request message. But this is highly discouraged, as it blocks an application server thread (the one allocated to run the EJB method from which the *receive* is performed). It also requires manual (bean-managed) TX demarcation, as the request message will only be sent after TX commit, and hence a receive waiting for a response message to that request message within the same TX will always be unsuccessful.

Summary and Comparison

Actors, Async Session Beans, 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

- Akka cluster and specific Java EE app server cluster features
- Details of Akka paths
- Akka configuration
- EJB deployment descriptors, environment entries and JNDI
- Distinction between application and system exceptions for EJBs
- Actor death watch
- Akka scheduler and EJB timer service details
- Design patterns