WASHINGTON DC
OCTOBER 15-18, 2012

Spring + Akka - A Journey of Discovery

Josh Suereth
@jsuereth

Nilanjan Raychaudhuri
@nraychaudhuri

Story line

We have a successful business - Finding cheap hotels

But..

- We have scaling issues
- We cannot handle the load
- Customers are unhappy

We need...

- Solution that integrates with Spring
- Easy to implement and change
- Developer friendly

Usual suspects

?

We decided to use...

What is Akka?

Akka is a event-driven middleware framework, for building scalable and reliable distributed applications for JVM

Akka Philosophy?

Make it easy for developers to build concurrent and scalable applications

Why Akka?

- Scale up (Concurrency)
- Scale out (Remoting)
- Fault tolerance

Why Akka? (Contd...)

- Use it with Java today
- Option of leveraging Scala later
- Integrate with, deploy into your current infrastructure

We will use Scala

- It reads like pseudo code
- Pattern matching
- Integrates with Java

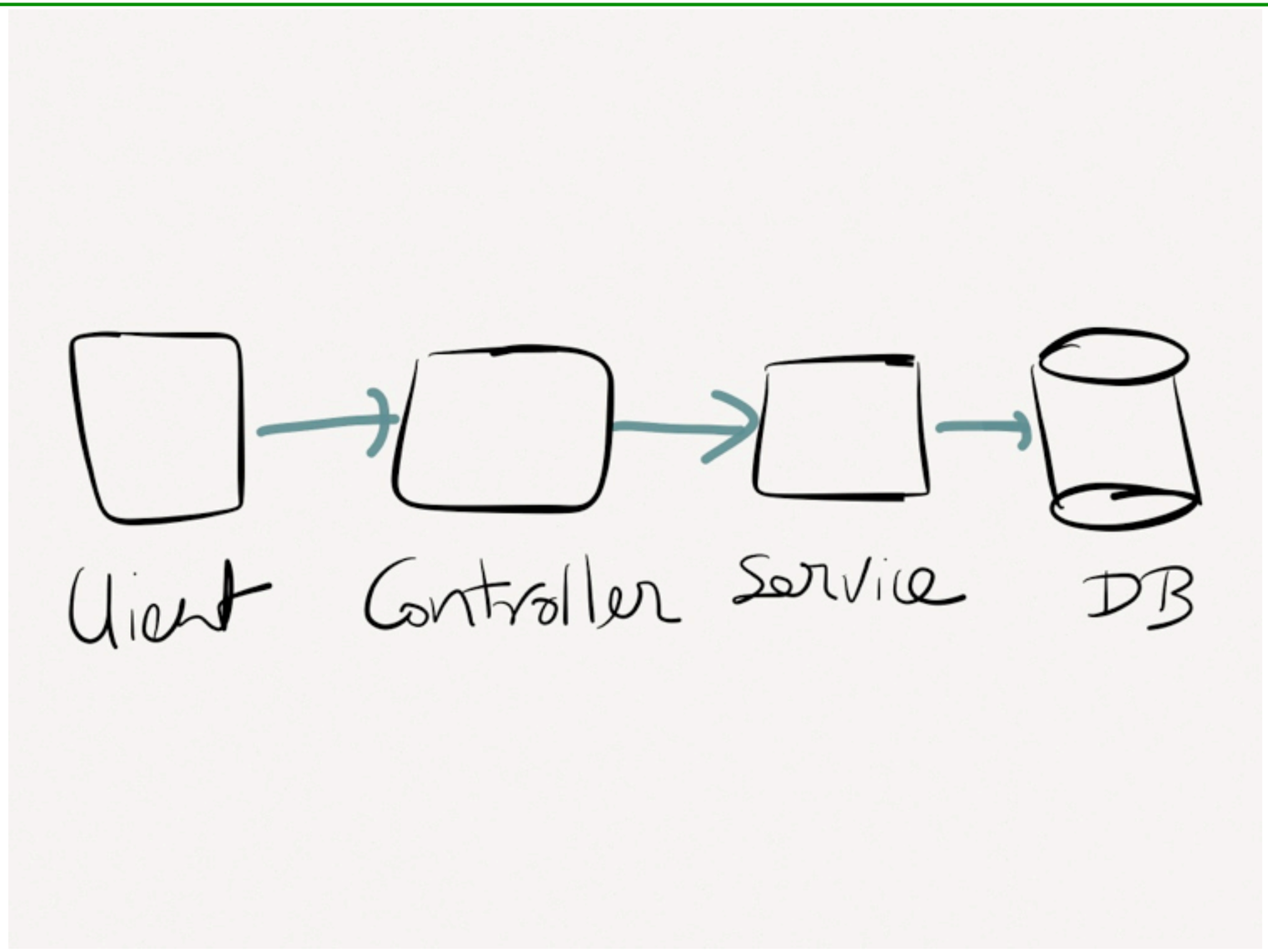
What problem we are solving again?

Searching for hotel

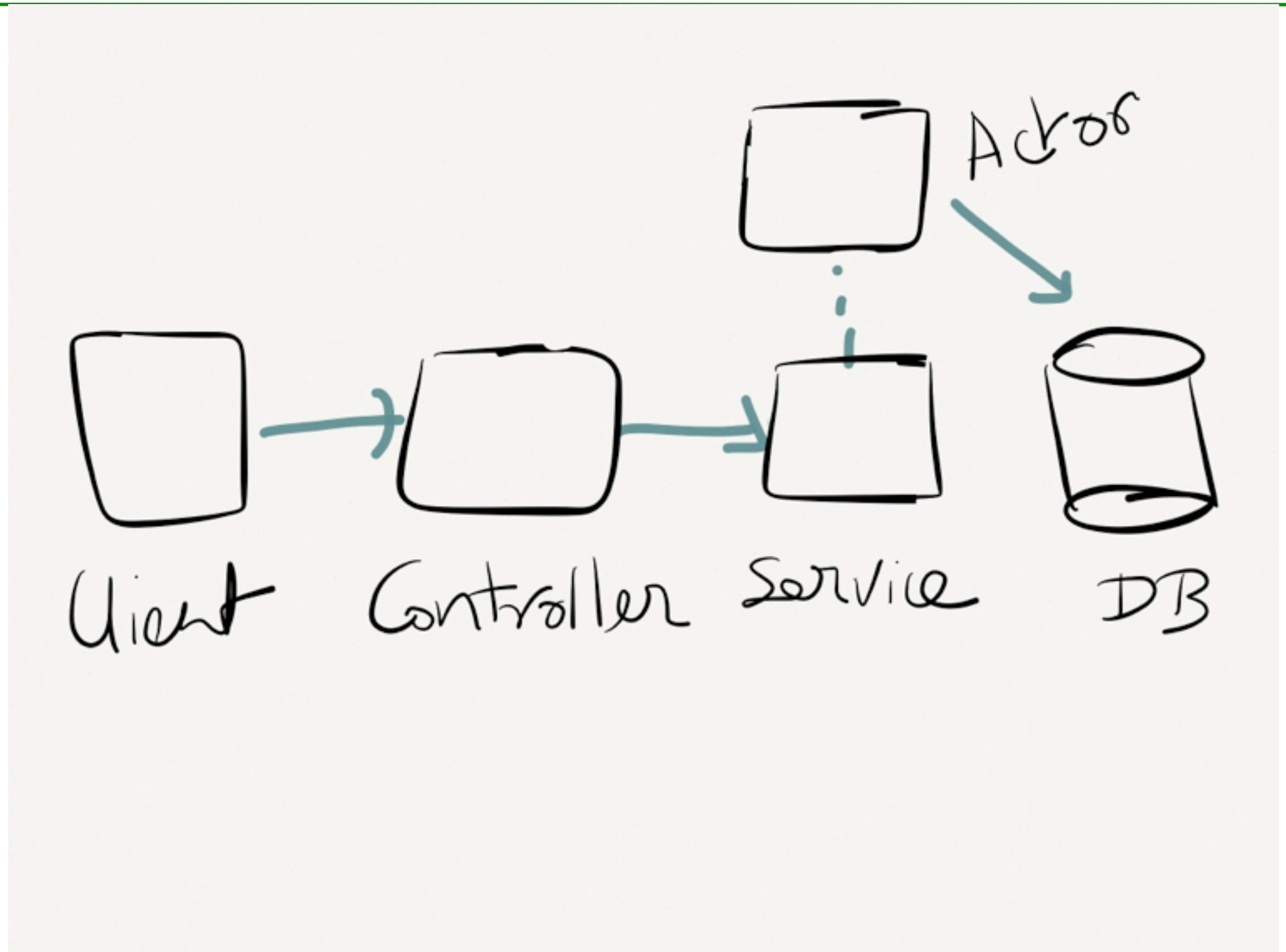
Codebase

https://github.com/jsuereth/spring-akka-sample

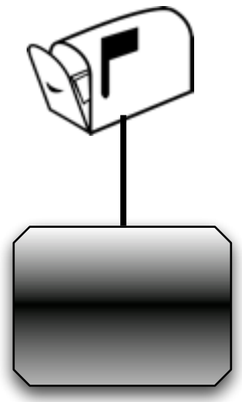
Current flow



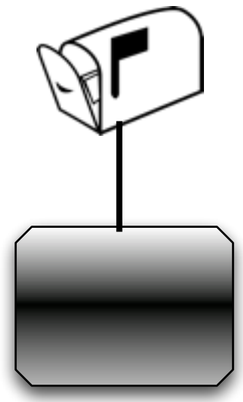
Lets cache data using Actor



What is an Actor?



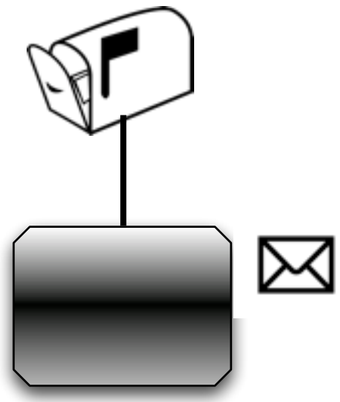
Actor



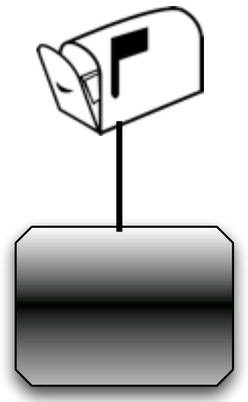
Actor

Actor Model

What is an Actor?



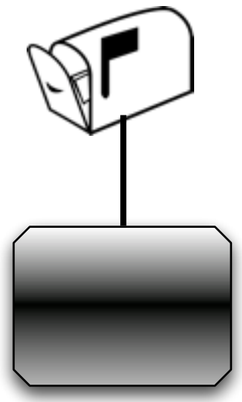
Actor



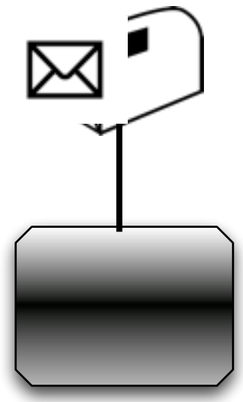
Actor

Actor Model

What is an Actor?



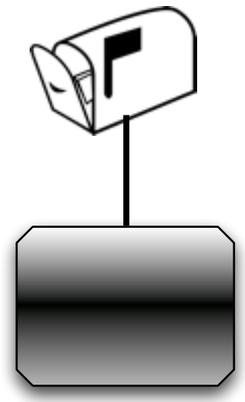
Actor



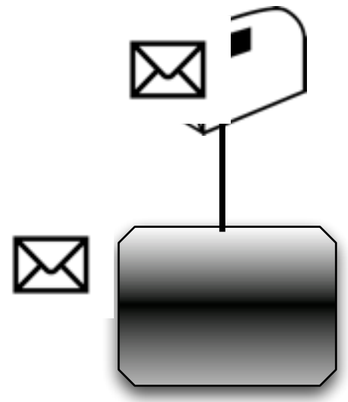
Actor

Actor Model

What is an Actor?



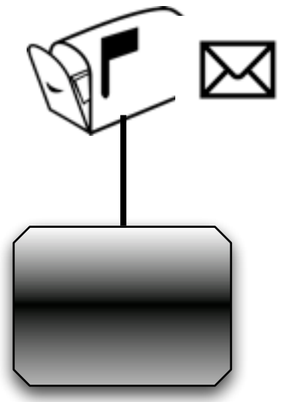
Actor



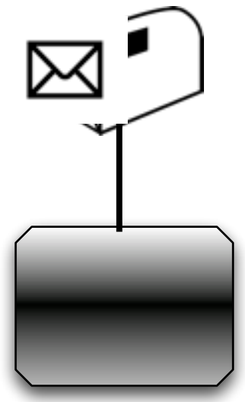
Actor

Actor Model

What is an Actor?



Actor



Actor

Actor Model

What is an Actor?

- Unit of code organization
- Keeps policy decisions separate
- Unit of computation

Code

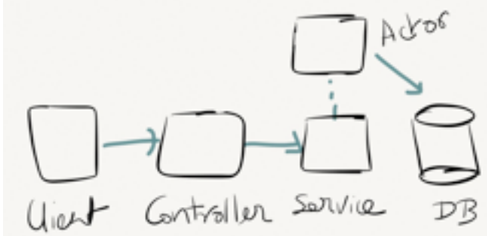
```
package org.springframework.samples.travel.actors

import akka.actor.Actor
import akka.actor.Props
import akka.event.Logging

class SampleActor extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case "test" => log.info("received test")
    case _      => log.info("received unknown message")
  }
}
```

Pattern matching...

```
def receive = {
  case "test" => log.info("received test")
  case _      => log.info("received unknown message")
}
```

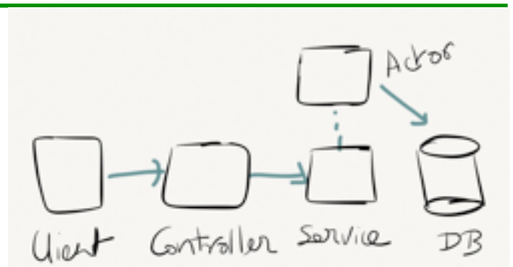
Solution using Actor

```
class SingleActorSearch(hotels: Seq[Hotel]) extends Actor {

  val index: Map[String, Hotel] = ...

  def receive: Receive = {
    case HotelQuery(search)  => sender ! findHotels(search)
  }
...
...
  private def findHotels(search: SearchCriteria): HotelResponse = {
    val matched = for {
      (s, hotel) <- index
      if s contains search.getSearchString.toLowerCase
    } yield hotel
    HotelResponse(matched.toSeq)
  }
}
```

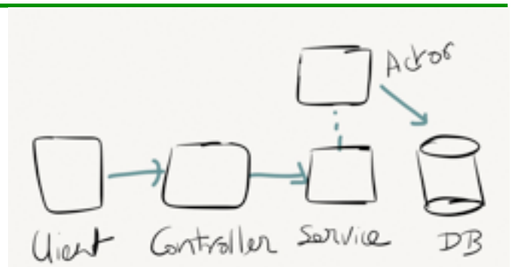


Initializing Actor

```
@Service
@Singleton
class AkkaSearchBean extends SearchService  {
  @(PersistenceContext @annotation.target.setter)
  var em: EntityManager = null

  val system =  ActorSystem("search-service")

  @PostConstruct
  def makeSearchActor = {
    // Startup....
    def getHotels = {
      val hotels = em.createQuery("select h from Hotel h") ...
      hotels foreach em.detach
      hotels
    }
    // Now feed data into Akka Search service.
    val searchProps = Props(new SingleActorSearch(getHotels))
    // Create the search actor
    system.actorOf(searchProps, "search-service-frontend")
  }
}
```



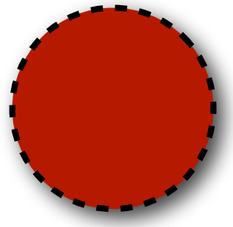
Using the Actor




```
/** Returns the current search service front-end actor. */
  def searchActor: ActorRef = system actorFor (system / "search-service-frontend")

  override def findHotels(criteria: SearchCriteria): java.util.List[Hotel] = {
    implicit val timeout = Timeout(5 seconds)
    val response = (searchActor ? HotelQuery(criteria)).mapTo[HotelResponse]
    Await.result(response, akka.util.Duration.Inf).hotels.asJava
  }
}
```

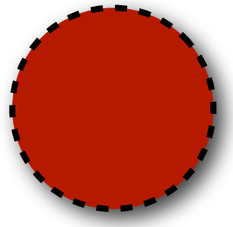
Topology

Guardian System Actor



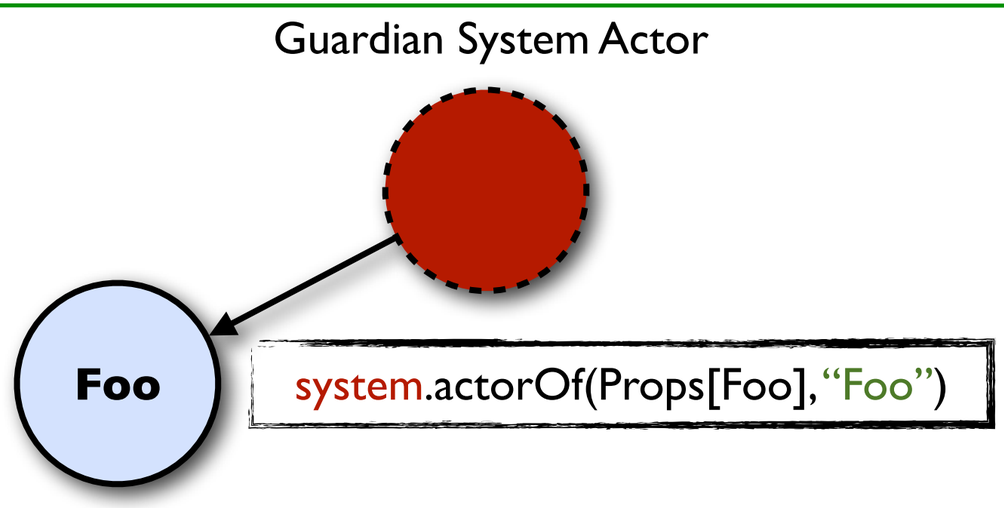
Topology

Guardian System Actor

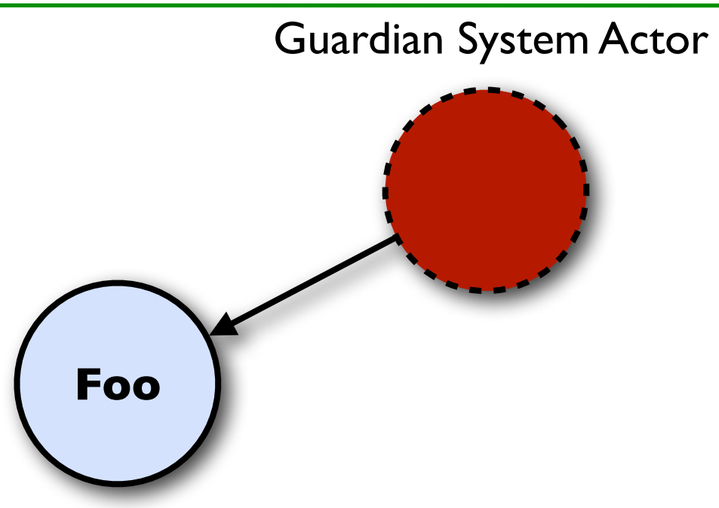


system.actorOf(Props[Foo], “Foo”)

Topology

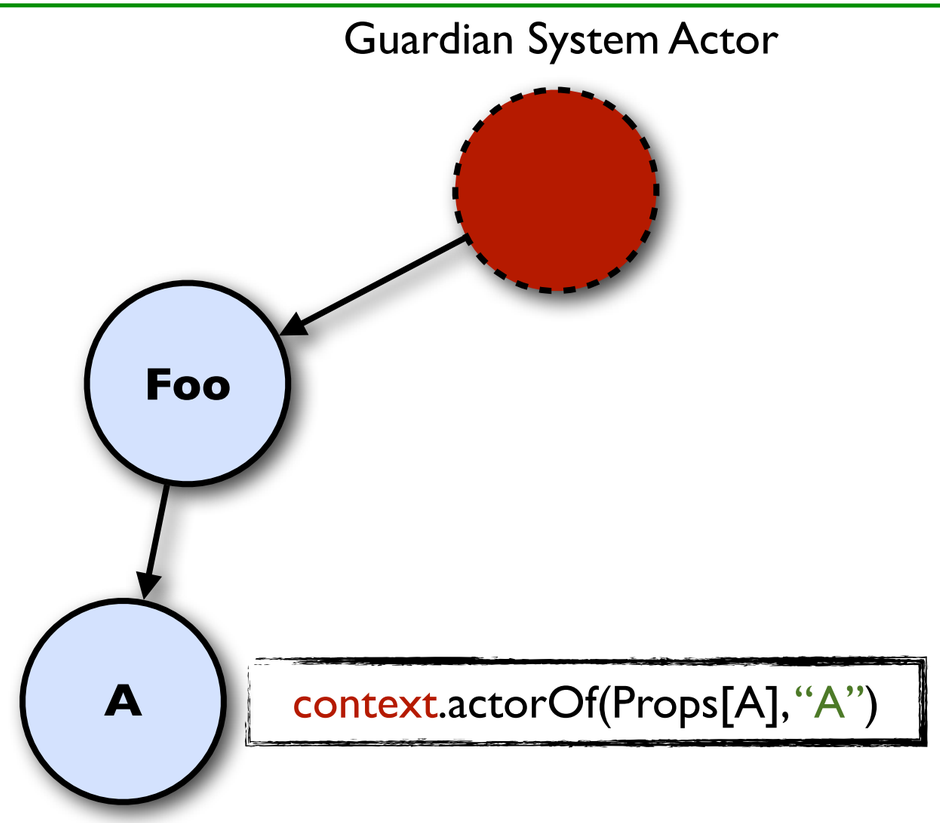


Topology

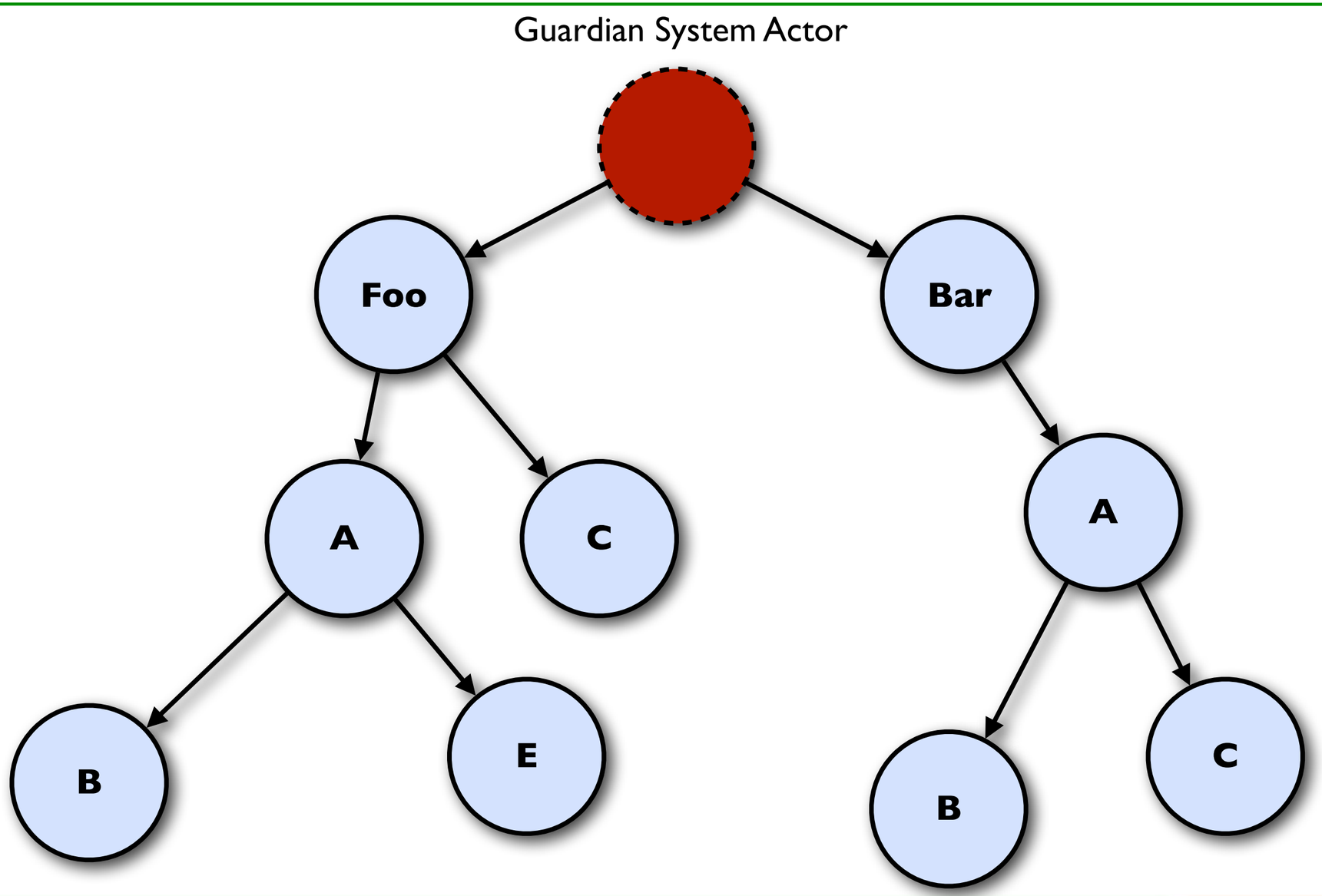



```
context.actorOf(Props[A], "A")
```

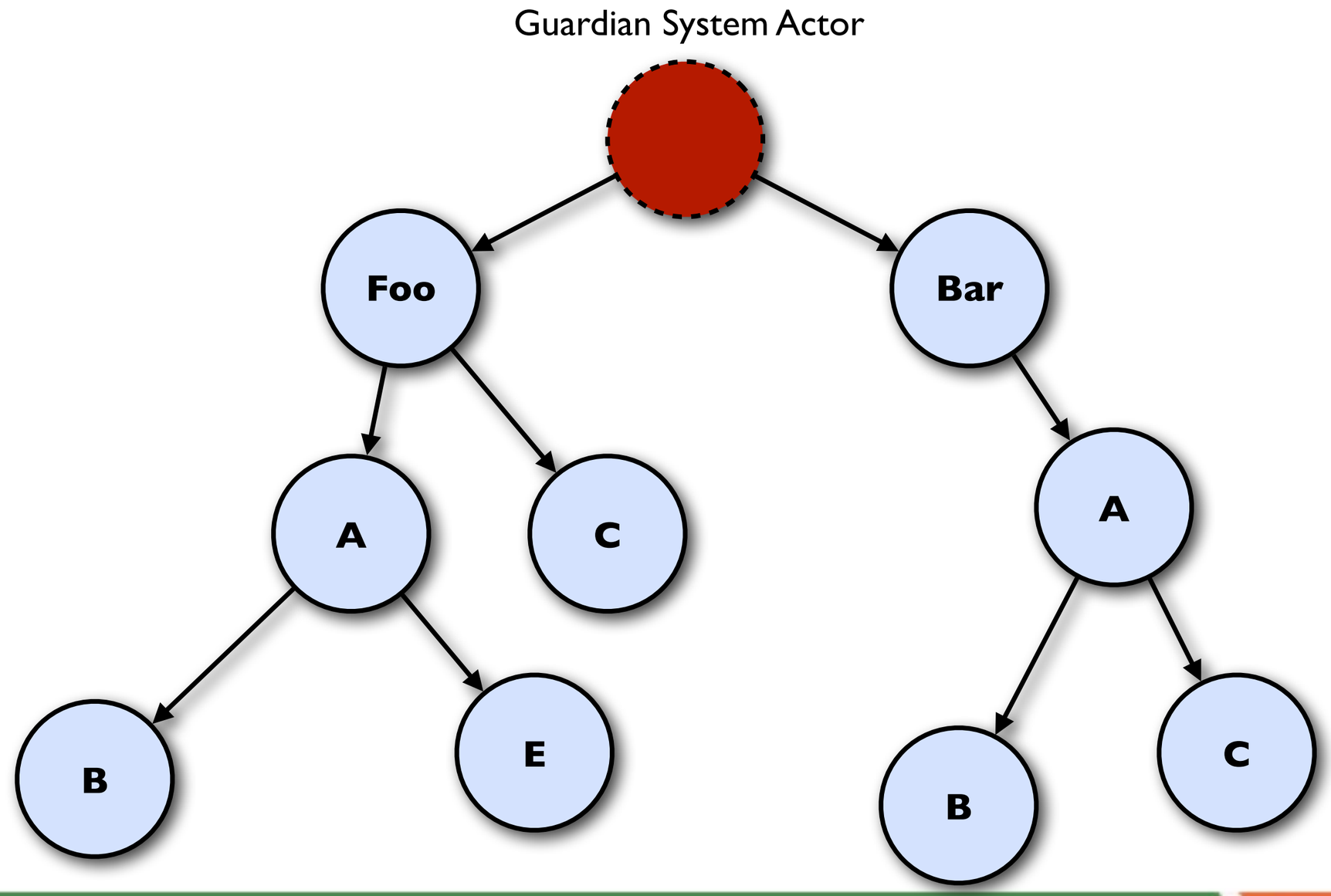
Topology



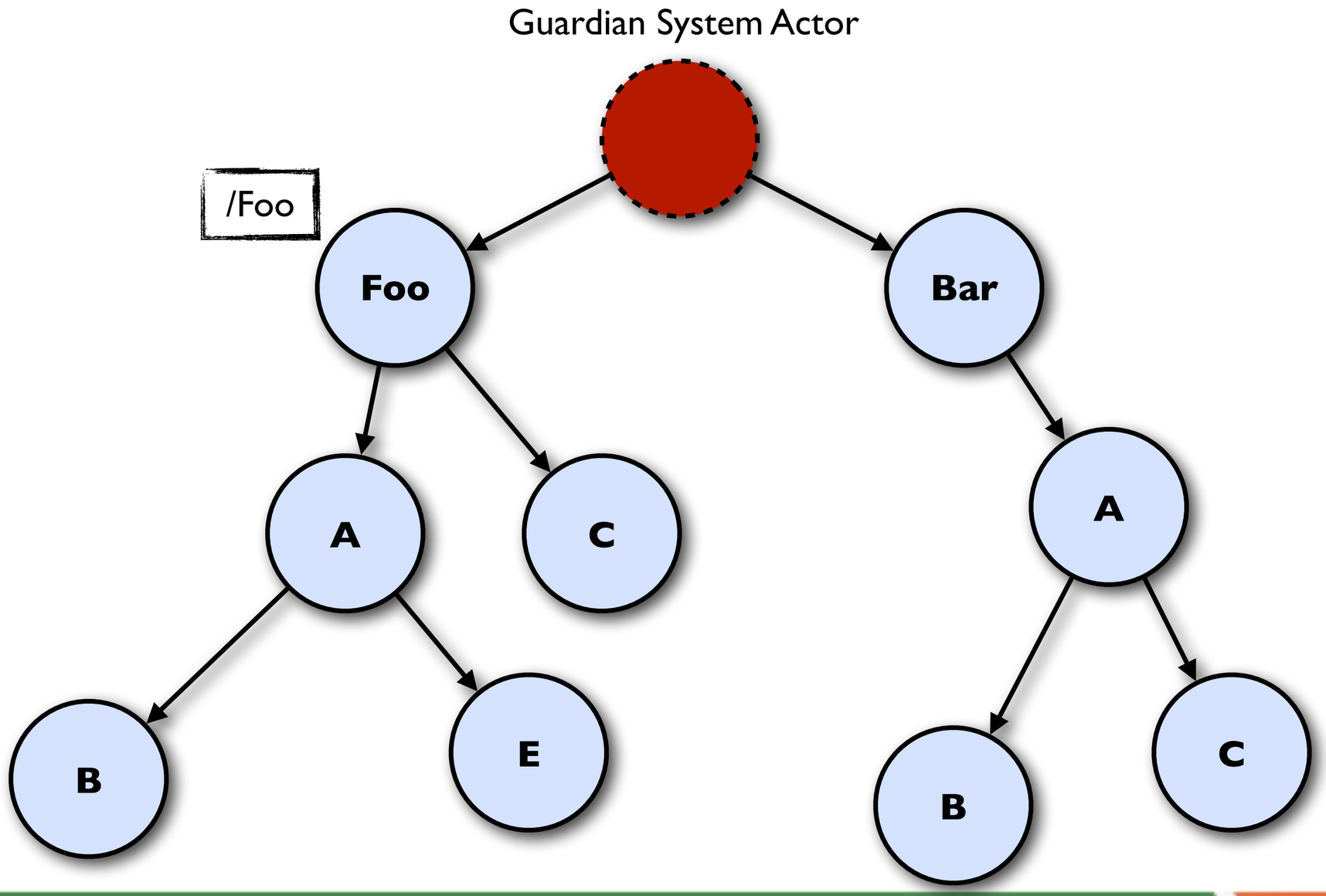
Topology



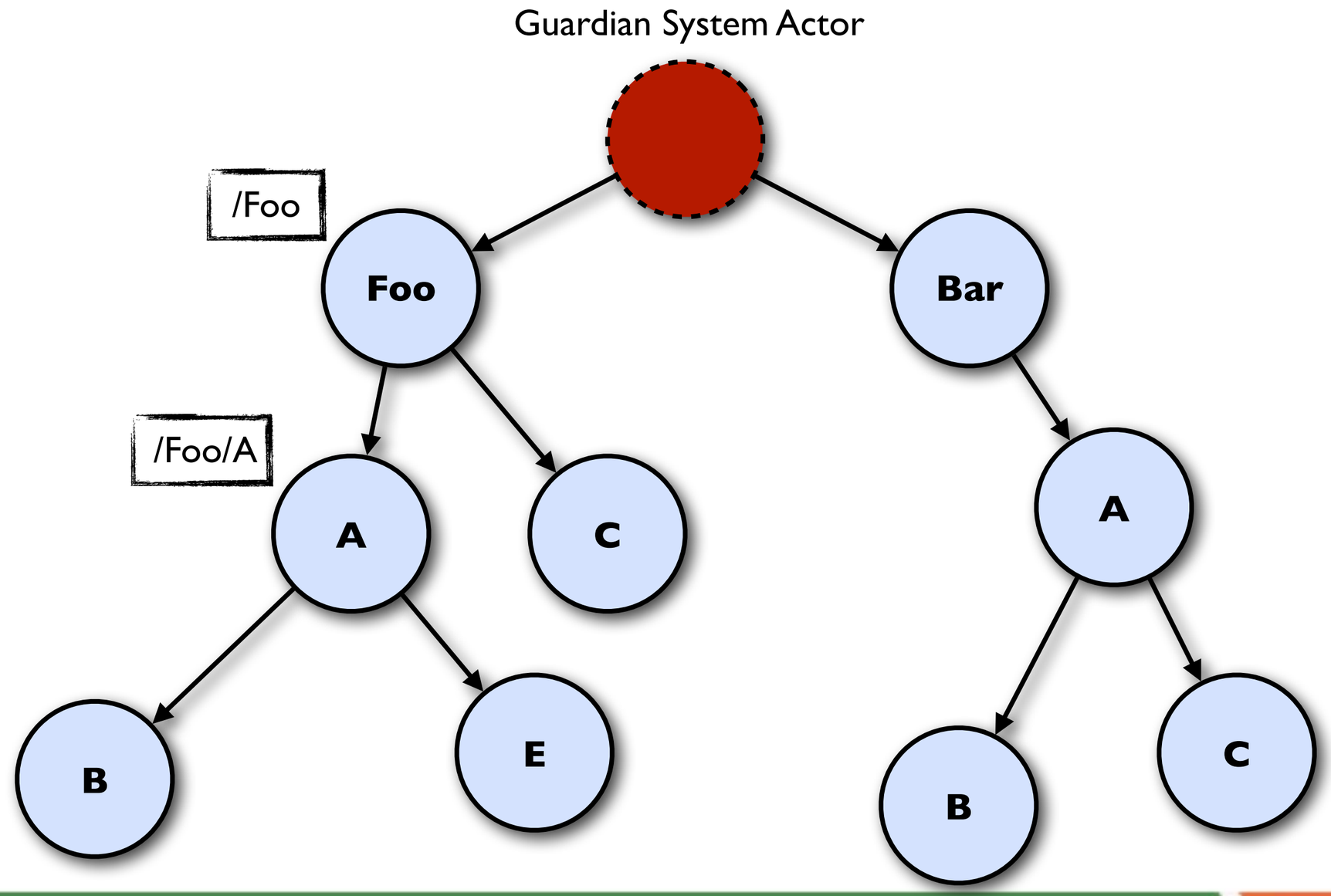
Name Resolution - like filesystem



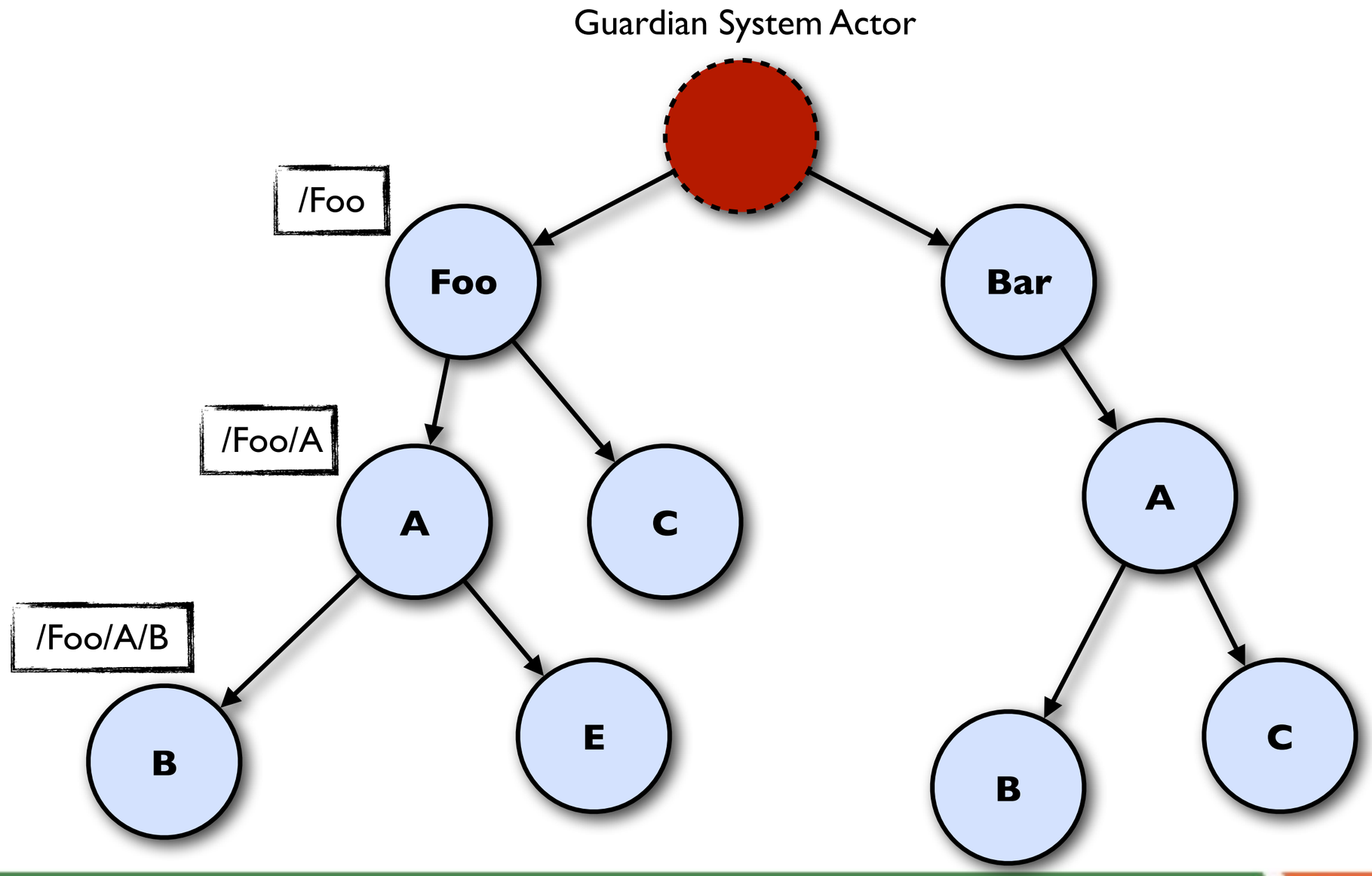
Name Resolution - like filesystem



Name Resolution - like filesystem



Name Resolution - like filesystem

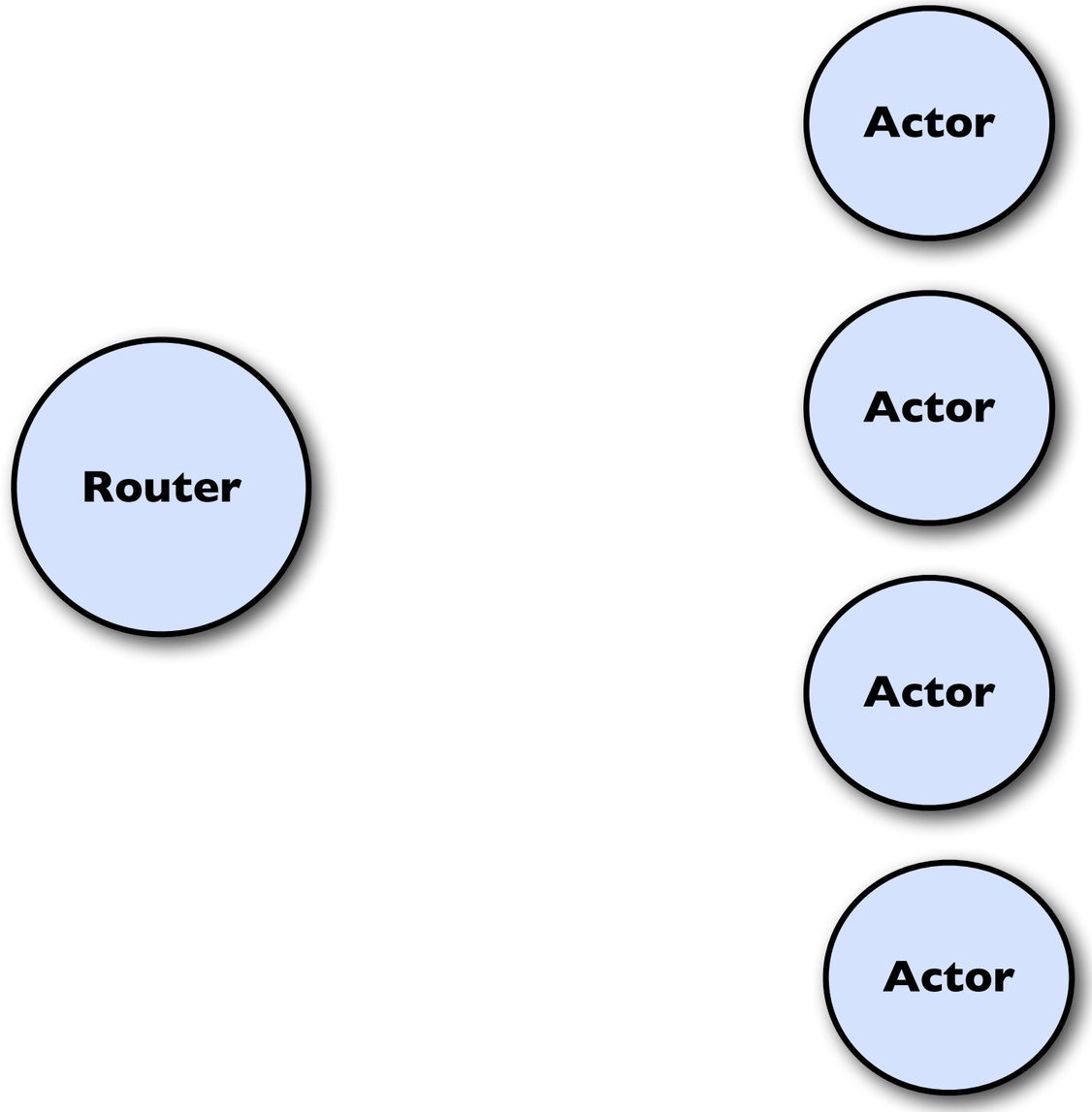


Recap

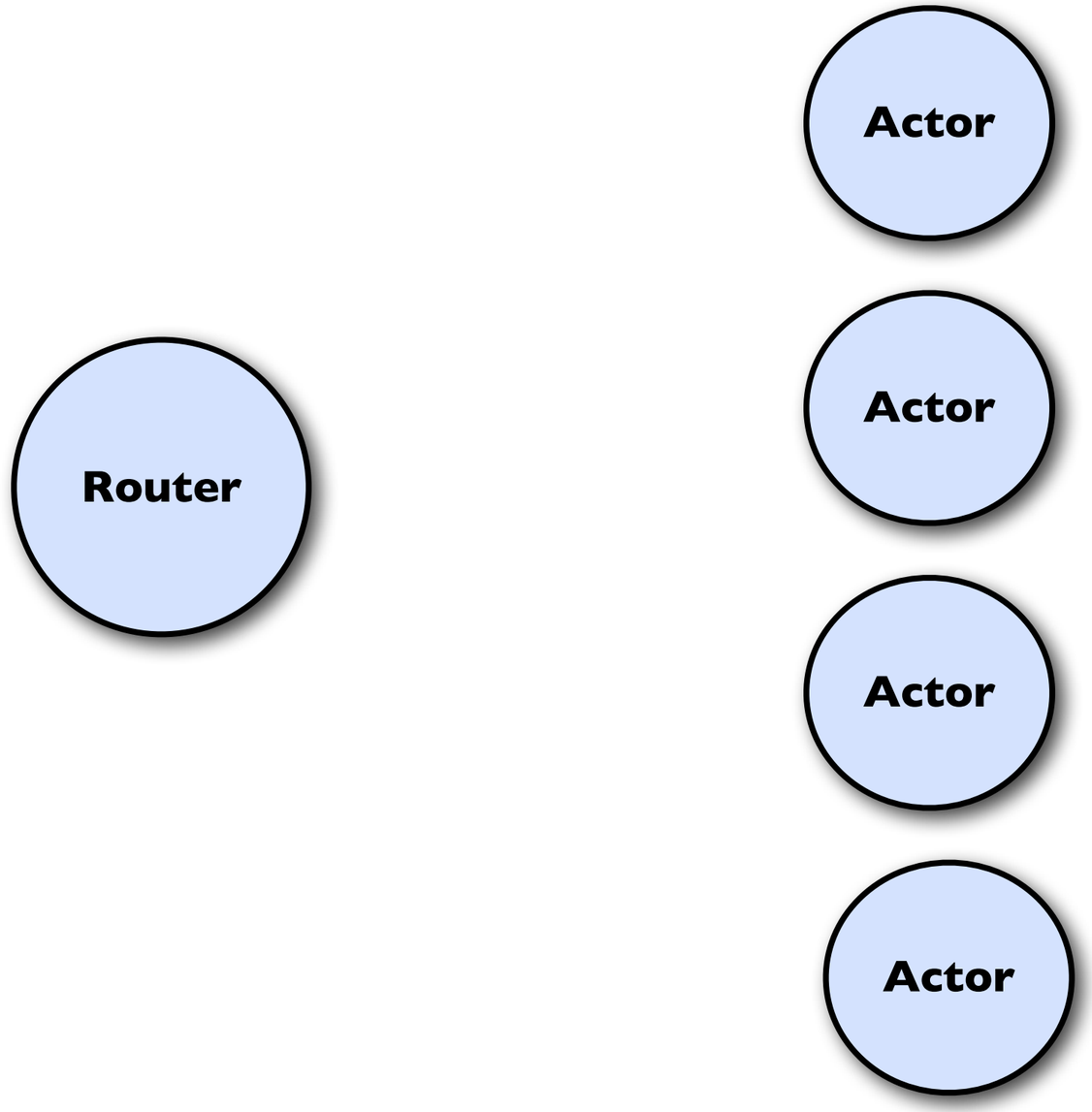
- Like organization, actors easily form hierarchy
- Actors are stateful
- Its a higher abstraction over threads

Next step - Router

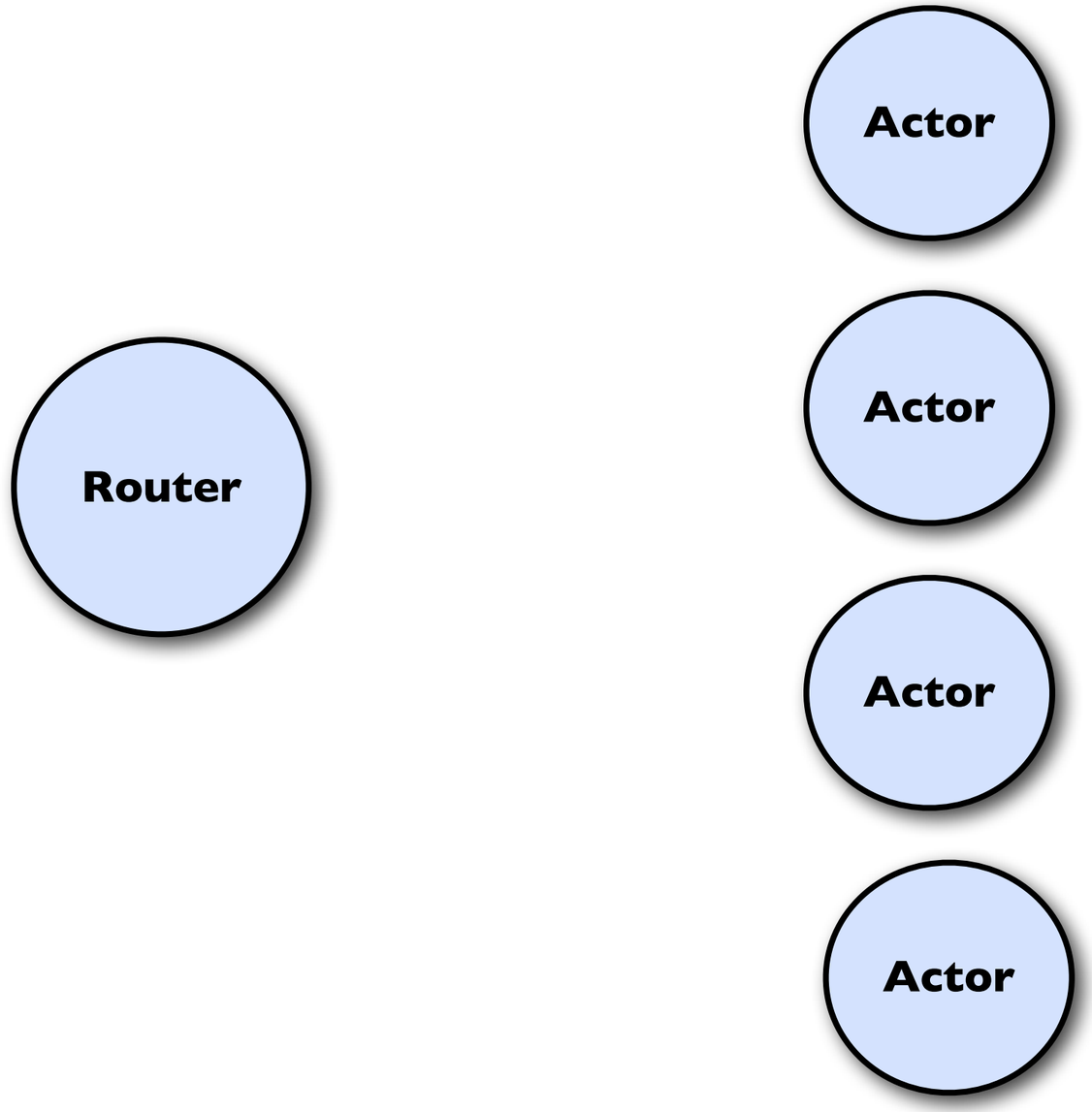
Next step - Router



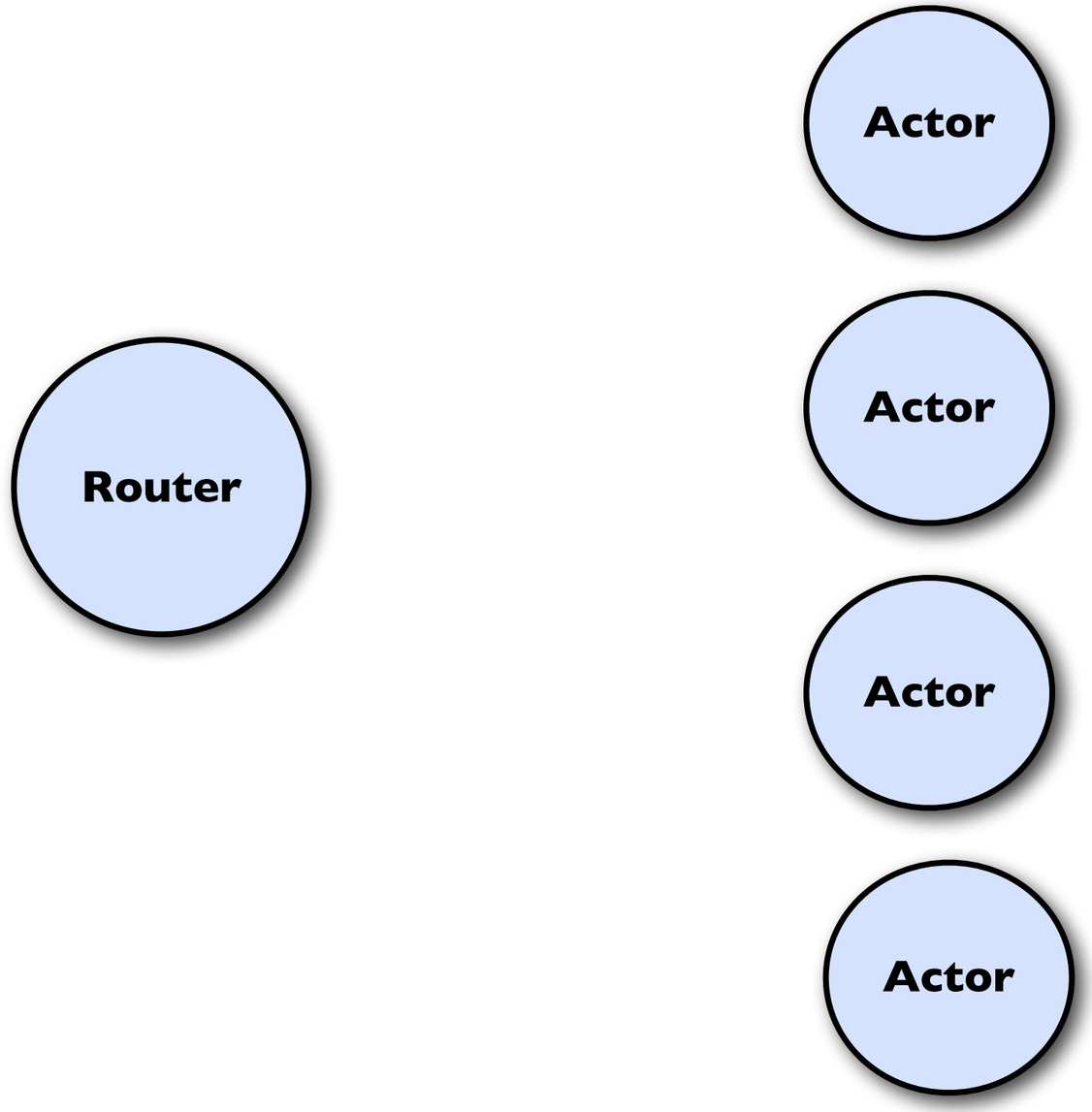
Next step - Router



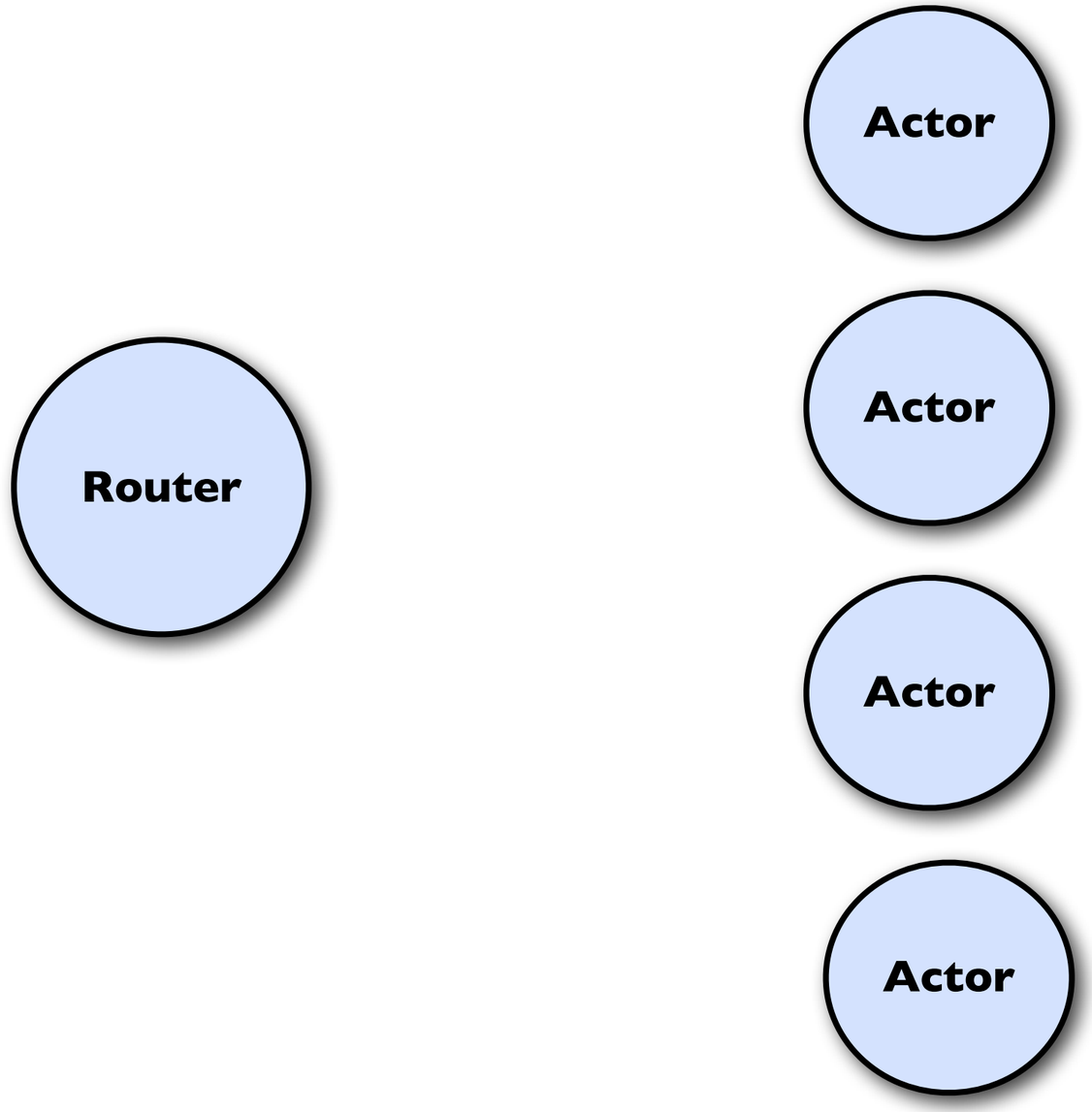
Next step - Router



Next step - Router

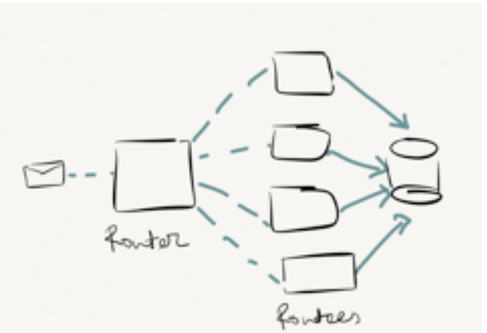


Next step - Router



Solution using Router

```
@PostConstruct
def makeSearchActor = {
  // Startup....
  def getHotels = {
    val hotels = em.createQuery("select h from Hotel h") ...
    hotels foreach em.detach
    hotels
  }
  // Now feed data into Akka Search service.
  val searchProps = Props(new SingleActorSearch(getHotels))
  val router = RoundRobinRouter(nrOfInstances=5)

  // Create the search actor
  system.actorOf(searchProps withRouter router , "search-service-frontend")
}
```



Scale out

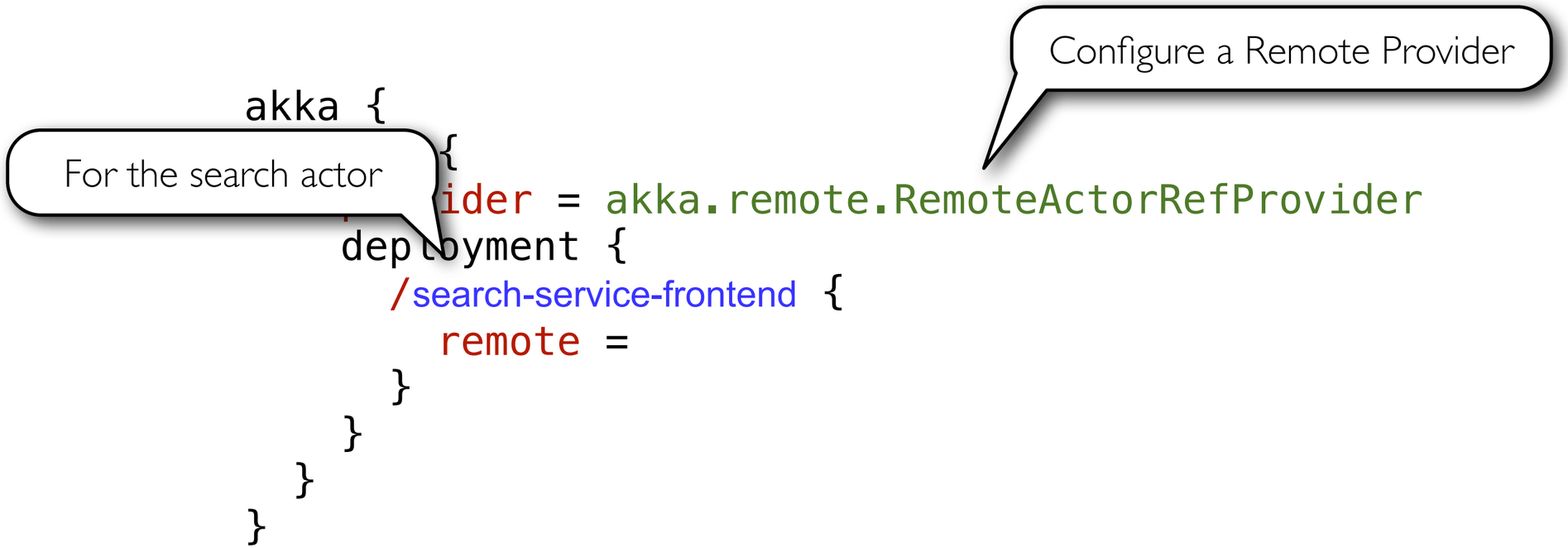
```
akka {
  actor {
    provider = akka.remote.RemoteActorRefProvider
    deployment {
      /search-service-frontend {
        remote =
      }
    }
  }
}
```

Scale out

```
akka {
  actor {
    provider = akka.remote.RemoteActorRefProvider
    deployment {
      /search-service-frontend {
        remote =
      }
    }
  }
}
```

Configure a Remote Provider

Scale out


Configure a Remote Provider
For the search actor
akka {
ider = akka.remote.RemoteActorRefProvider
deployment {
/search-service-frontend {
remote =
}
}
}
}

Scale out

```
akka {
  actor {
    provider = akka.remote.RemoteActorRefProvider
    deployment {
      /search-service-frontend {
        remote =
      }
    }
  }
}
```

Configure a Remote Provider

For the search actor

Define Remote Path

Scale out

```
akka {
  actor {
    provider = akka.remote.RemoteActorRefProvider
    deployment {
      /search-service-frontend {
        remote = akka://
      }
    }
  }
}
```

Configure a Remote Provider

For the search actor

Define Remote Path

Protocol

Scale out

```
akka {
  actor {
    provider = akka.remote.RemoteActorRefProvider
    deployment {
      /search-service-frontend {
        remote = akka://MySystem
      }
    }
  }
}
```

Configure a Remote Provider

For the search actor

Define Remote Path

Protocol

Actor System

Scale out

```
akka {
  actor {
    provider = akka.remote.RemoteActorRefProvider
    deployment {
      /search-service-frontend {
        remote = akka://MySystem@machine1
      }
    }
  }
}
```

Configure a Remote Provider

For the search actor

Define Remote Path

Protocol

Actor System

Hostname

Scale out

```
akka {
  actor {
    provider = akka.remote.RemoteActorRefProvider
    deployment {
      /search-service-frontend {
        remote = akka://MySystem@machine1:2552
      }
    }
  }
}
```

Configure a Remote Provider

For the search actor

Define Remote Path

Protocol

Actor System

Hostname

Port

Scale out

```
akka {
  actor {
    provider = akka.remote.RemoteActorRefProvider
    deployment {
      /search-service-frontend {
        remote = akka://MySystem@machine1:2552
      }
    }
  }
}
```

Configure a Remote Provider

For the search actor

Define Remote Path

Protocol

Actor System

Hostname

Port

Zero code changes

Recap

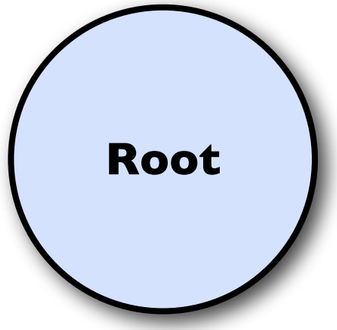
- Compile vs deployment decision
- Location Transparency
- Clustering support(Coming)

Scaling out

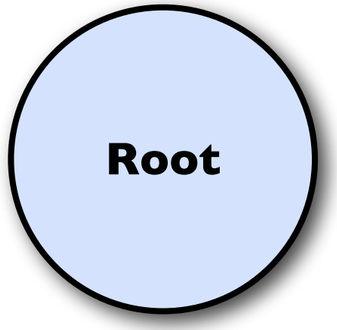
- We use split and join pattern called scatter and gather

Scatter Phase

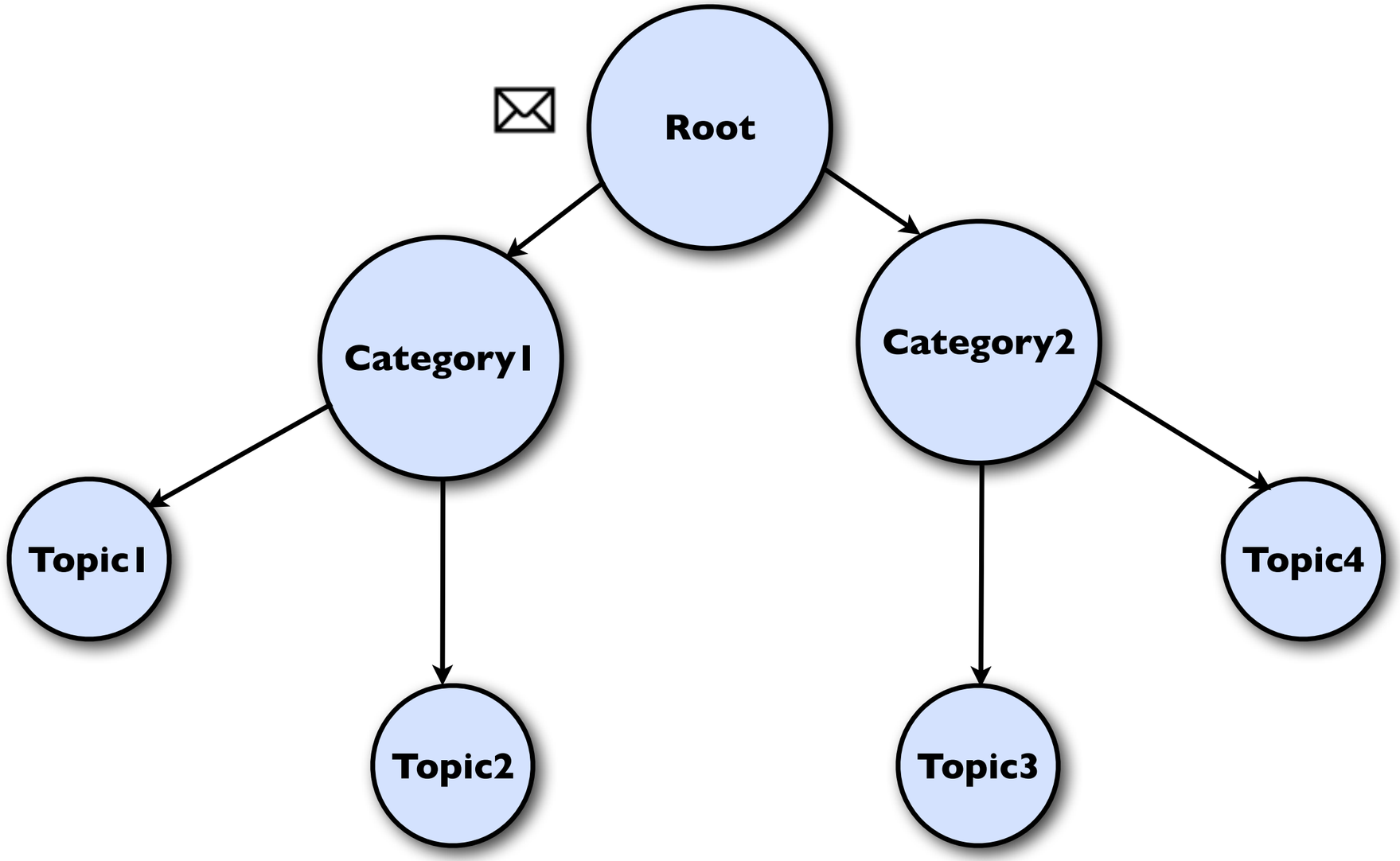
Scatter Phase



Scatter Phase

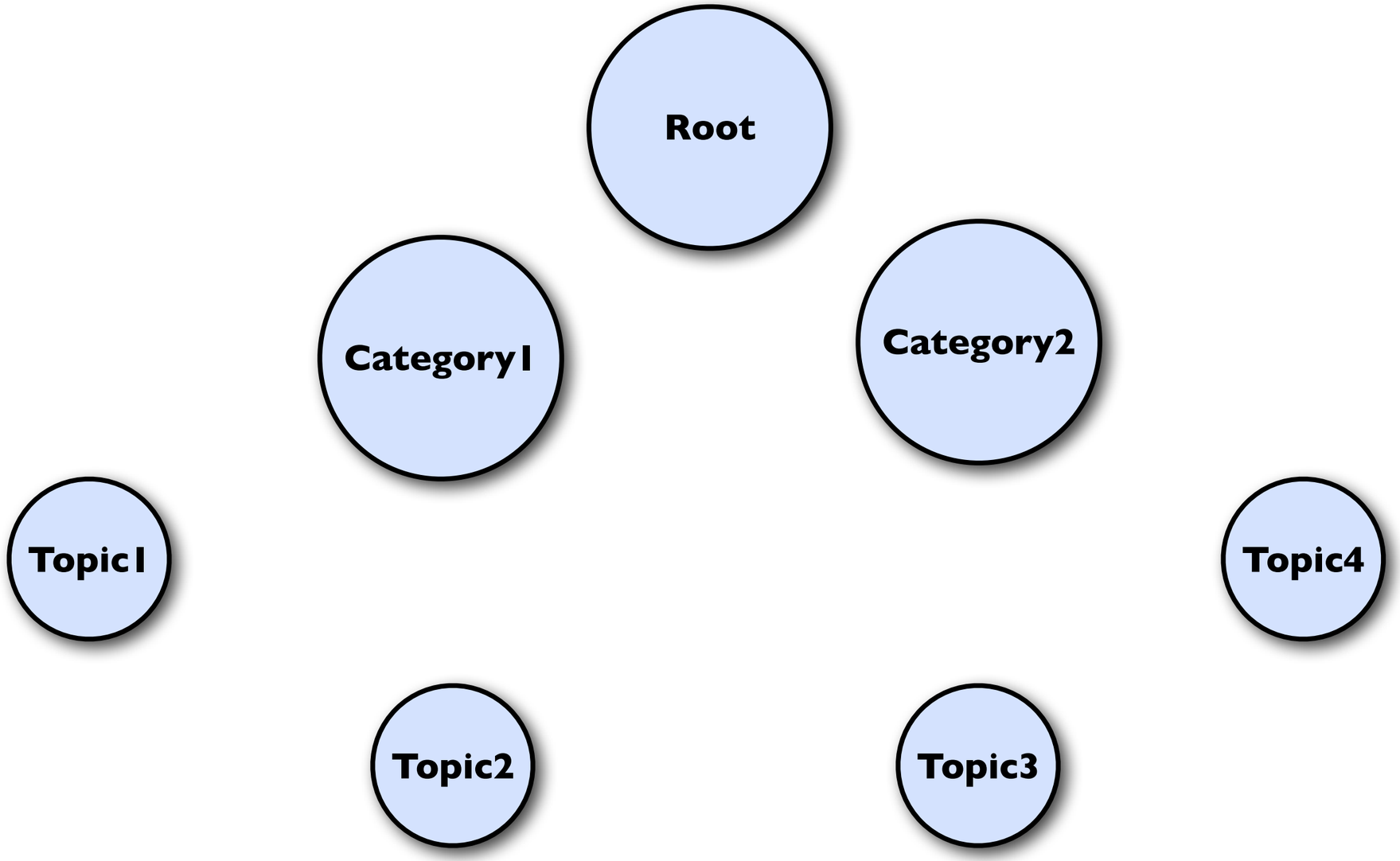


Scatter Phase

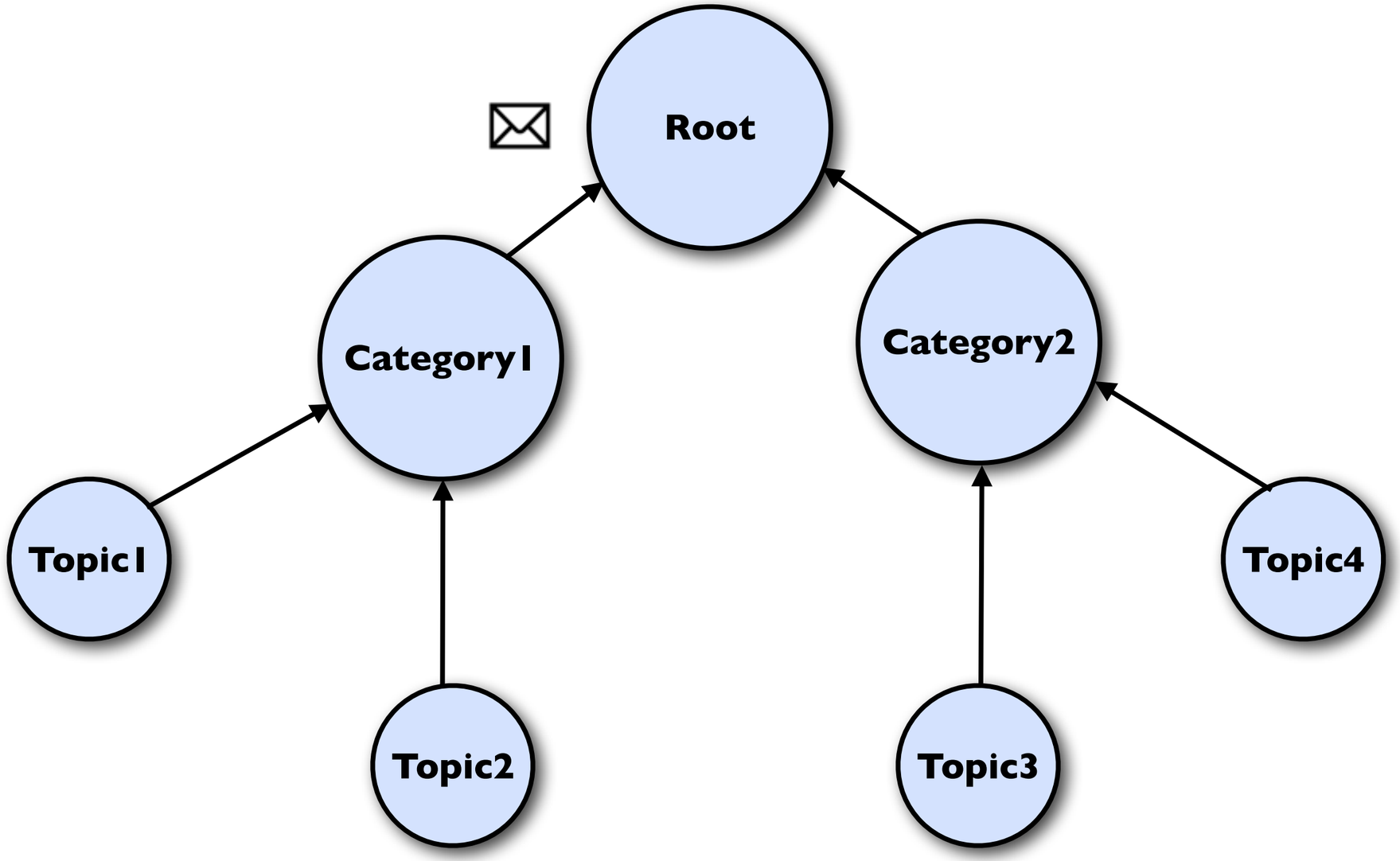


Gather Phase

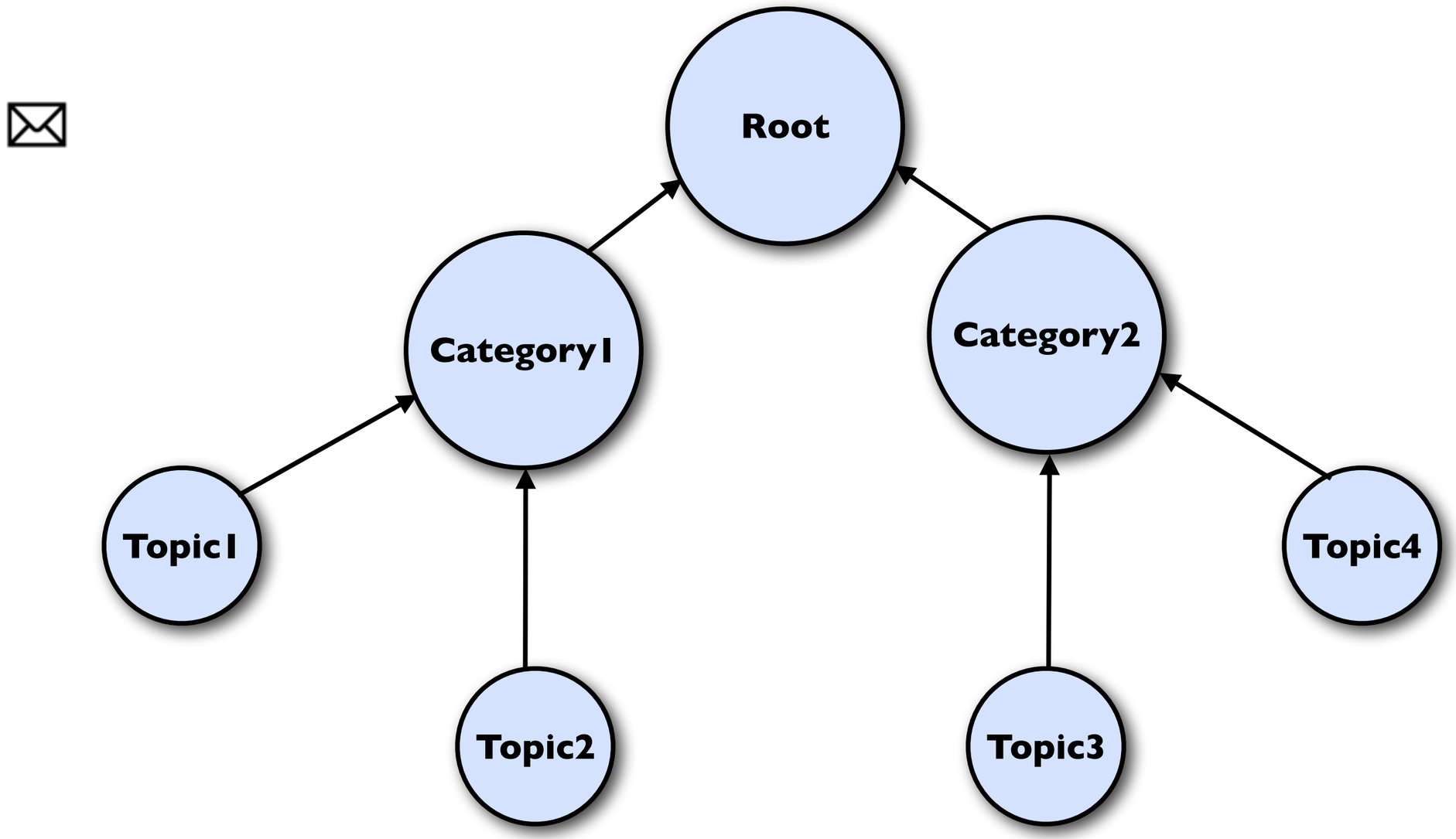
Gather Phase



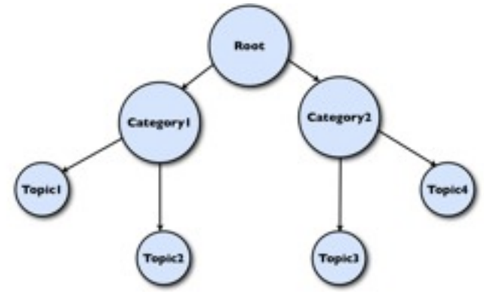
Gather Phase



Gather Phase



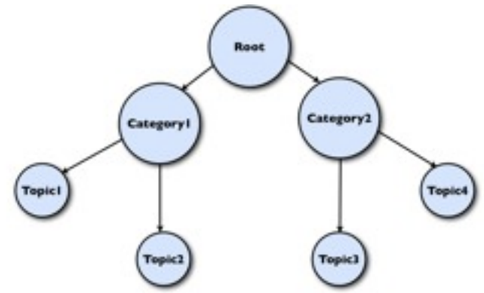
Solution using Scatter-Gather




```
@PostConstruct
def makeSearchActor = {
  // Startup....
  def getHotels = {
    val hotels = em.createQuery("select h from Hotel h") ...
    hotels foreach em.detach
    hotels
  }
  //val router = RoundRobinRouter(nrOfInstances=5)
  //val rawService = system.actorOf(searchProps withRouter router, "search-service-raw")

  val searchTreeProps = Props(new CountryCategoryActor(getHotels))
  val rawService = system.actorOf(searchTreeProps, "search-service-frontend")
  ()
}
```

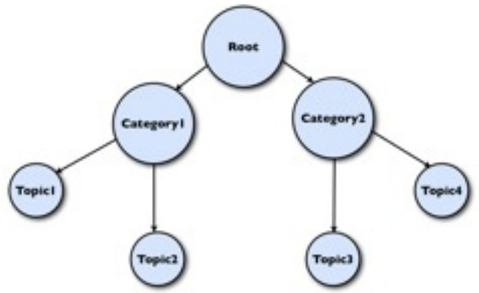
Scatter Actor

```
abstract class CategoryActor(hotels: Seq[Hotel]) extends Actor {
  val children = for {
    (cat, hotels) <- hotels groupBy category
    actorProps = Props(newChild(hotels))
  } yield context.actorOf(actorProps, scrubActorName(cat))

  def receive: Receive = {
    case query: HotelQuery =>
      val listener = sender
      val gathererProps = Props(new Gatherer(listener, children.size))
      val gatherer = context.actorOf(gathererProps)
      for(child <- children) {
        child.tell(query, gatherer)
      }
  }

  private def scrubActorName(name: String) = ...
  protected def category(hotel: Hotel): String
  protected def newChild(hotels: Seq[Hotel]): Actor
}
```



Gather Actor

```
class Gatherer(listener: ActorRef, numNodes: Long)
        extends Actor {
  var responses: Seq[HotelResponse] = Seq.empty

  def receive: Receive = {
    case response: HotelResponse =>
      responses = responses :+ response
      if(responses.size >= numNodes) joinResponses()
  }

  def joinResponses():  Unit = {
    listener ! HotelResponse(responses flatMap (_.hotels))
    context stop self
  }
}
```



Recap

- Splitting is better
- Small actors are better
- Actors are very cheap to create

Quiz

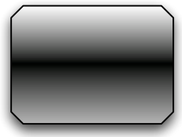
How you will know
that thread
crashed?

Stacktrace in log file

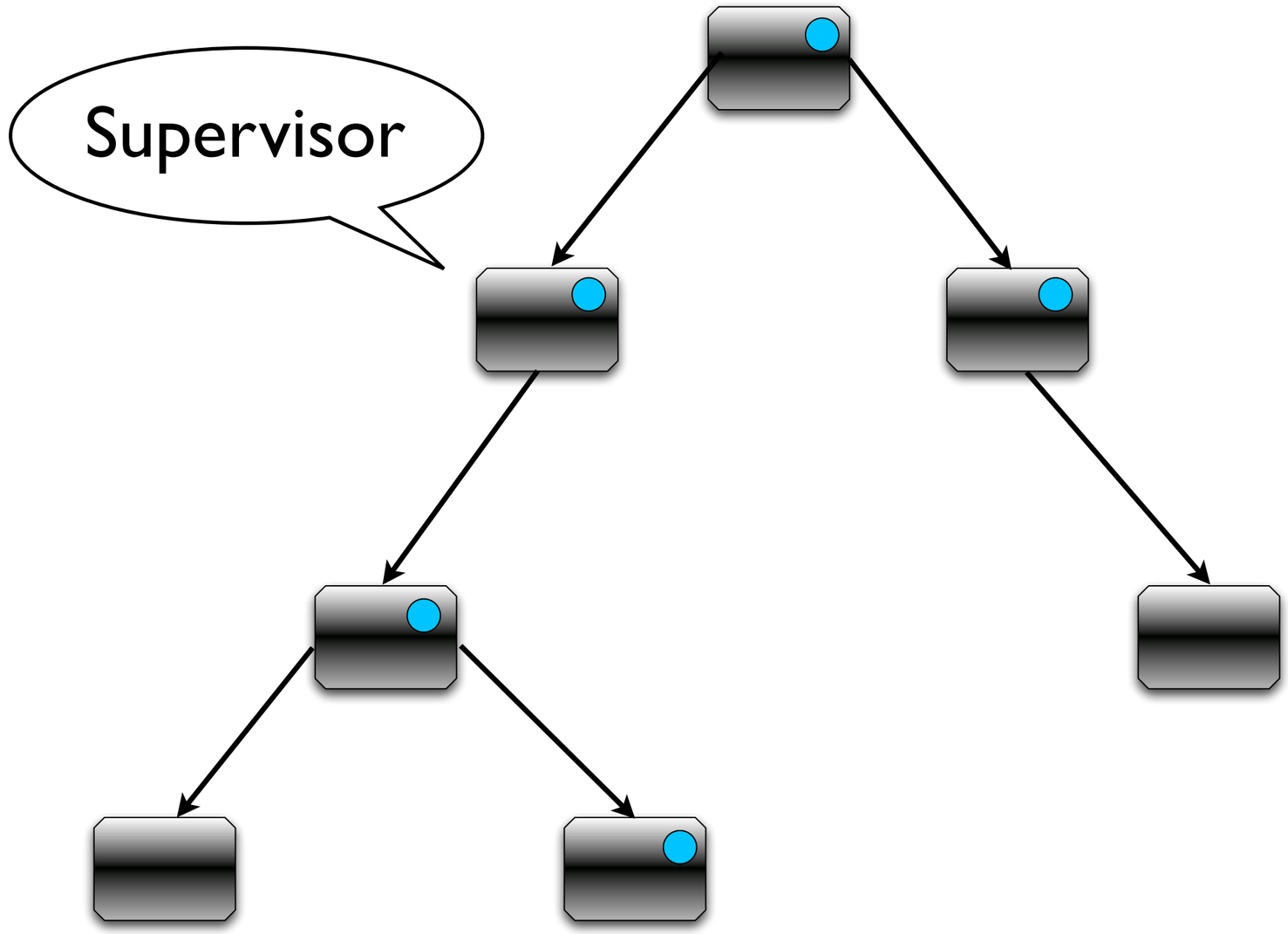
Akka model

Let it crash

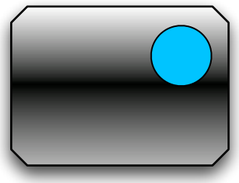
Actor Topology

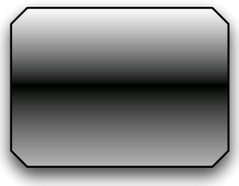


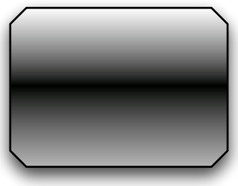
Actor Topology

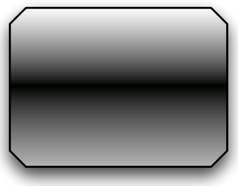


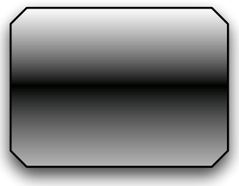
One for One strategy



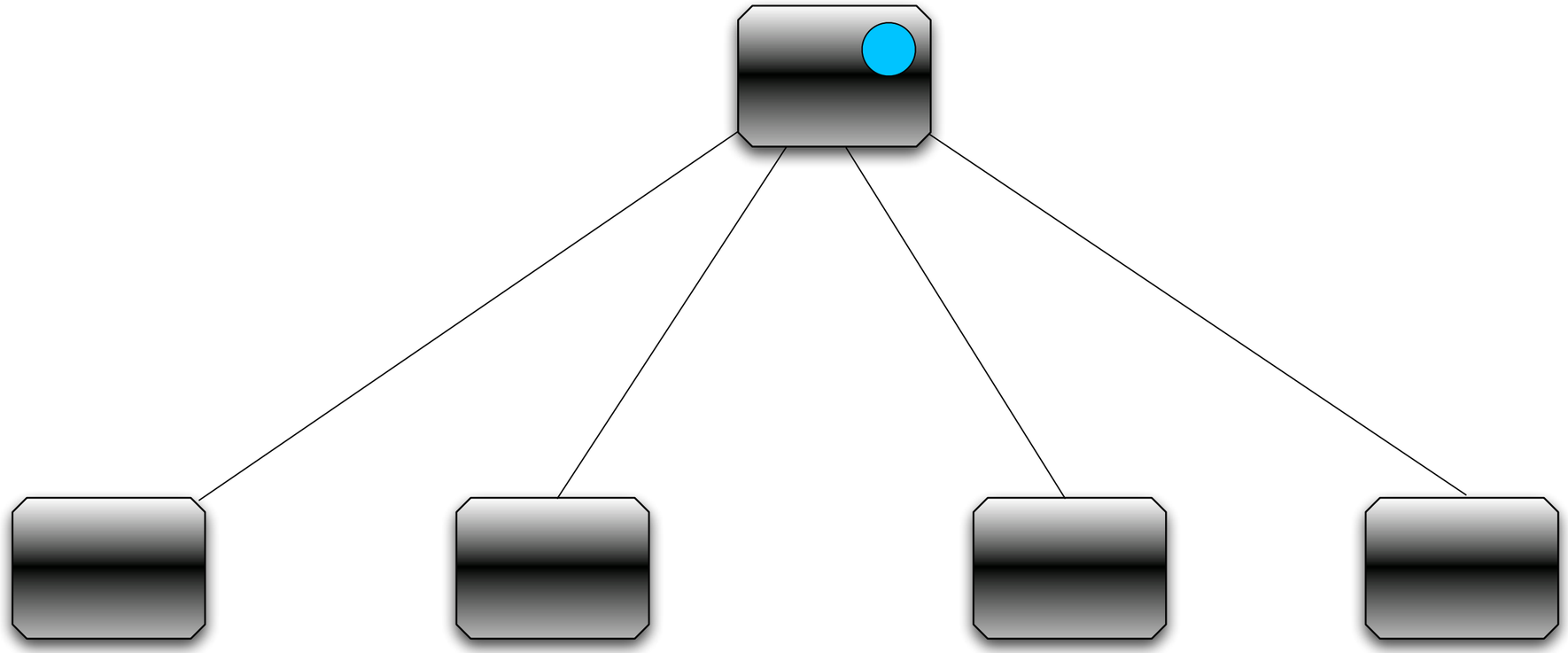




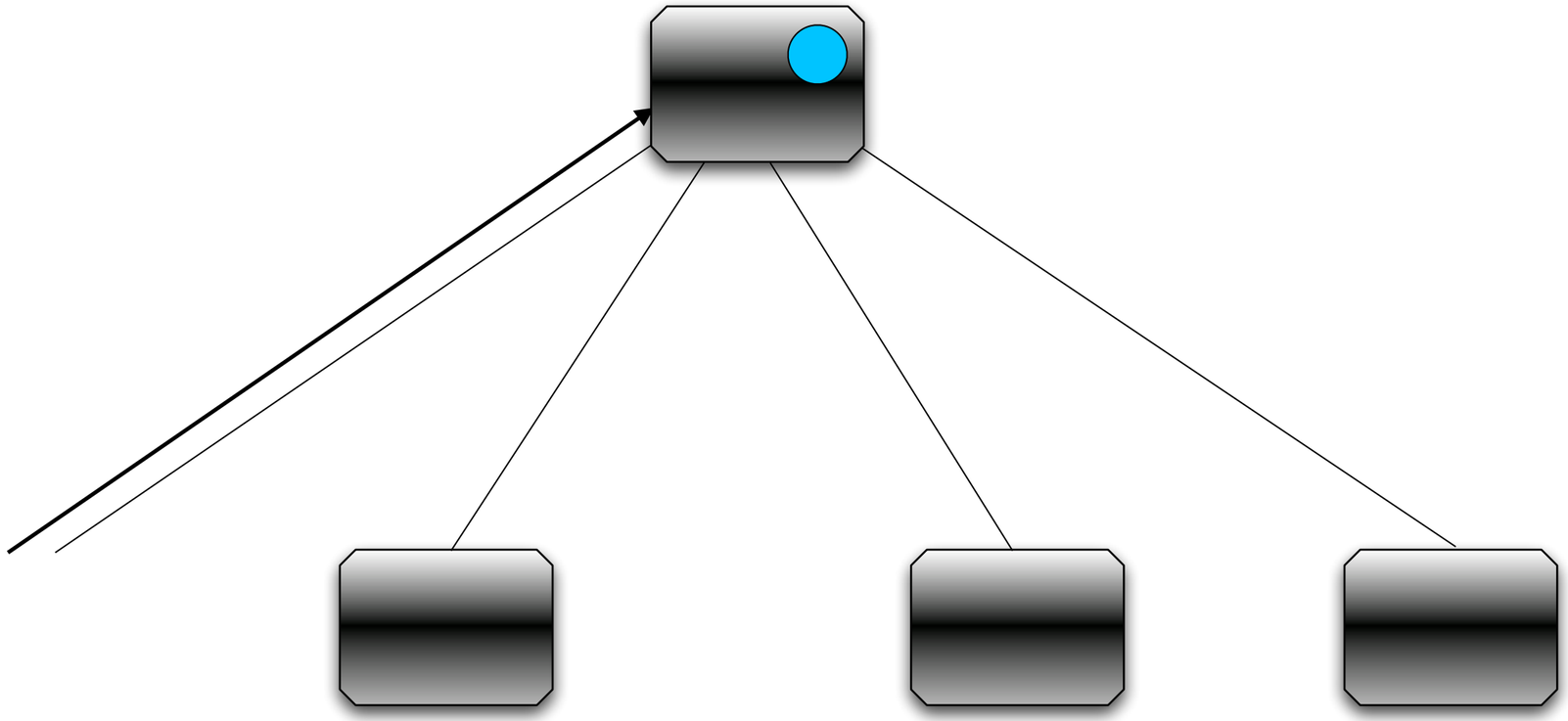




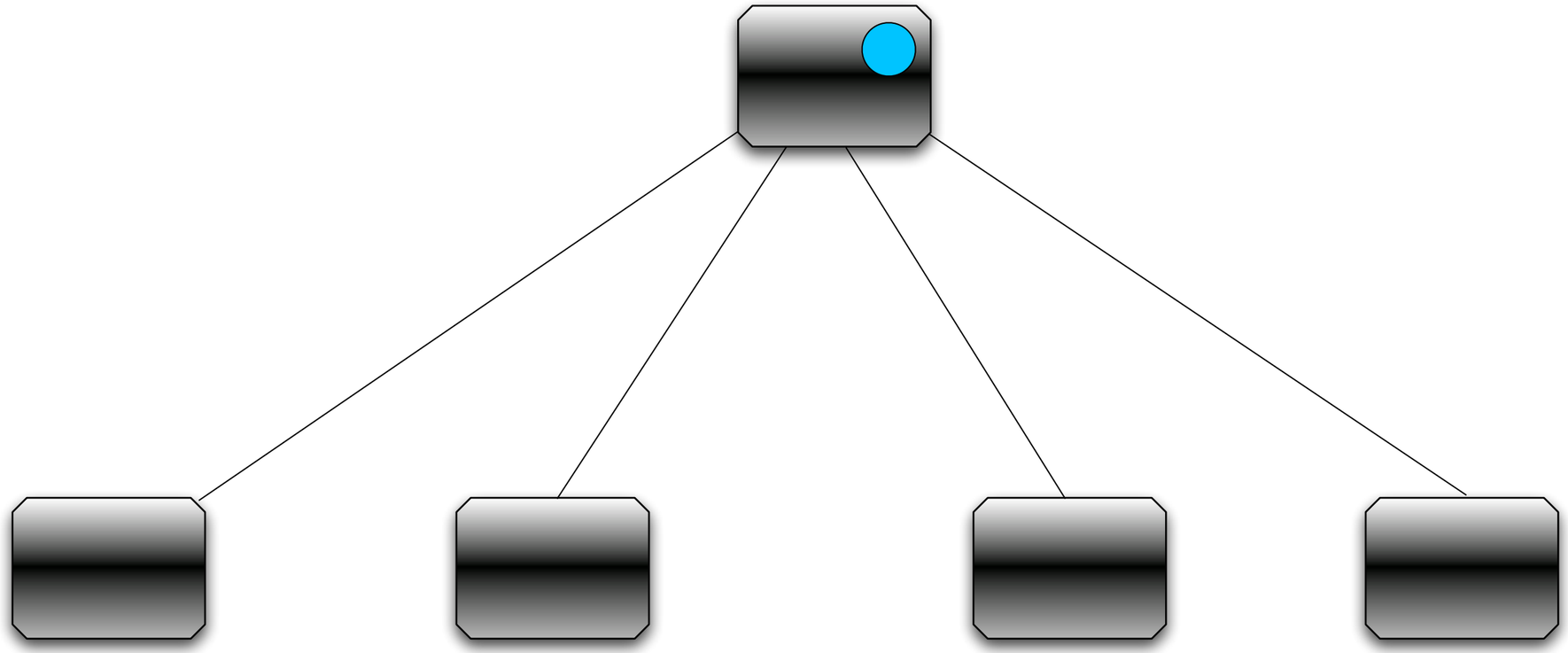
One for One strategy



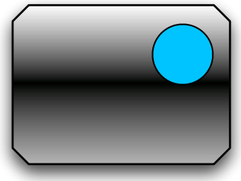
One for One strategy

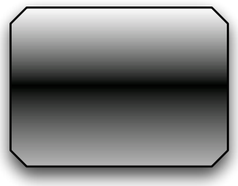


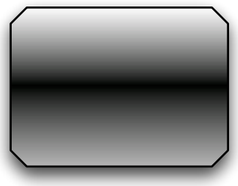
One for One strategy

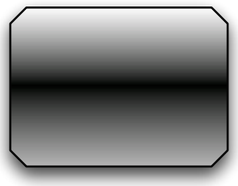


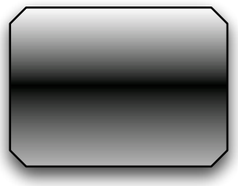
All for one strategy



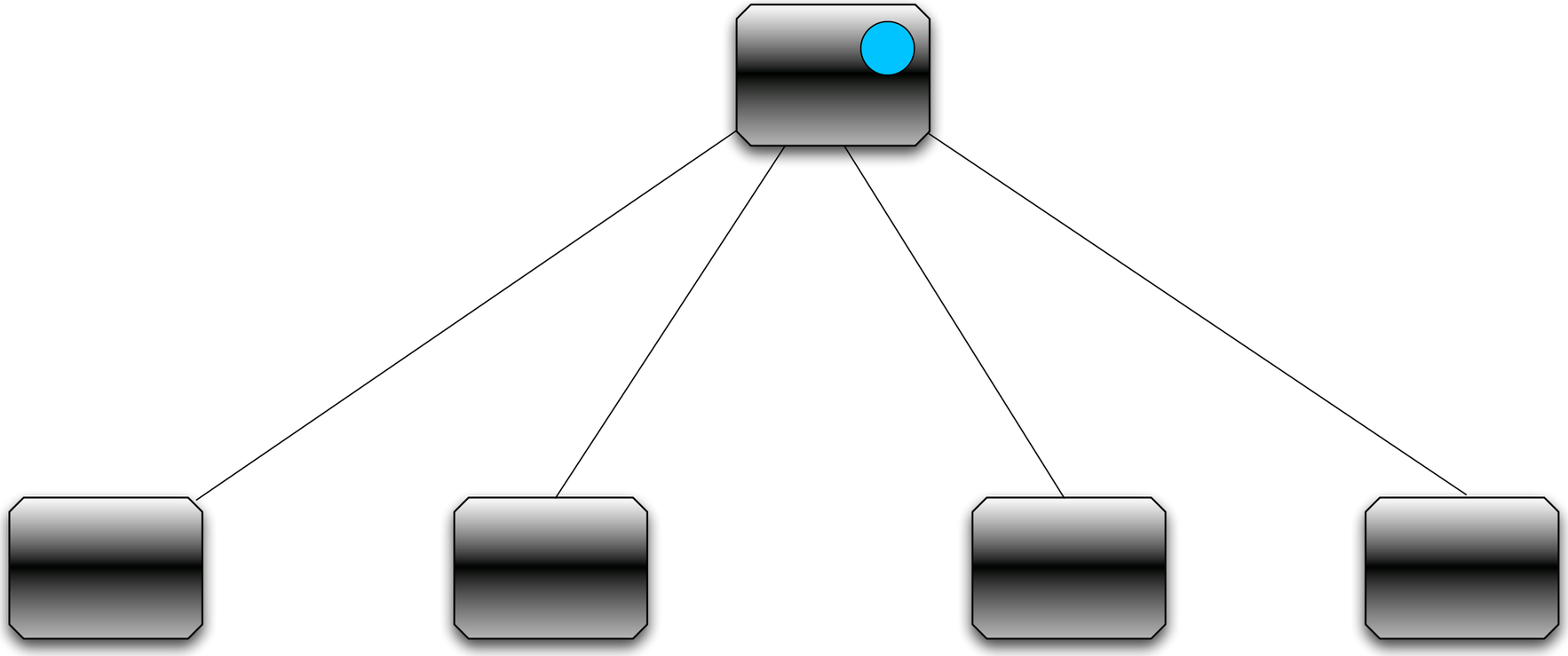




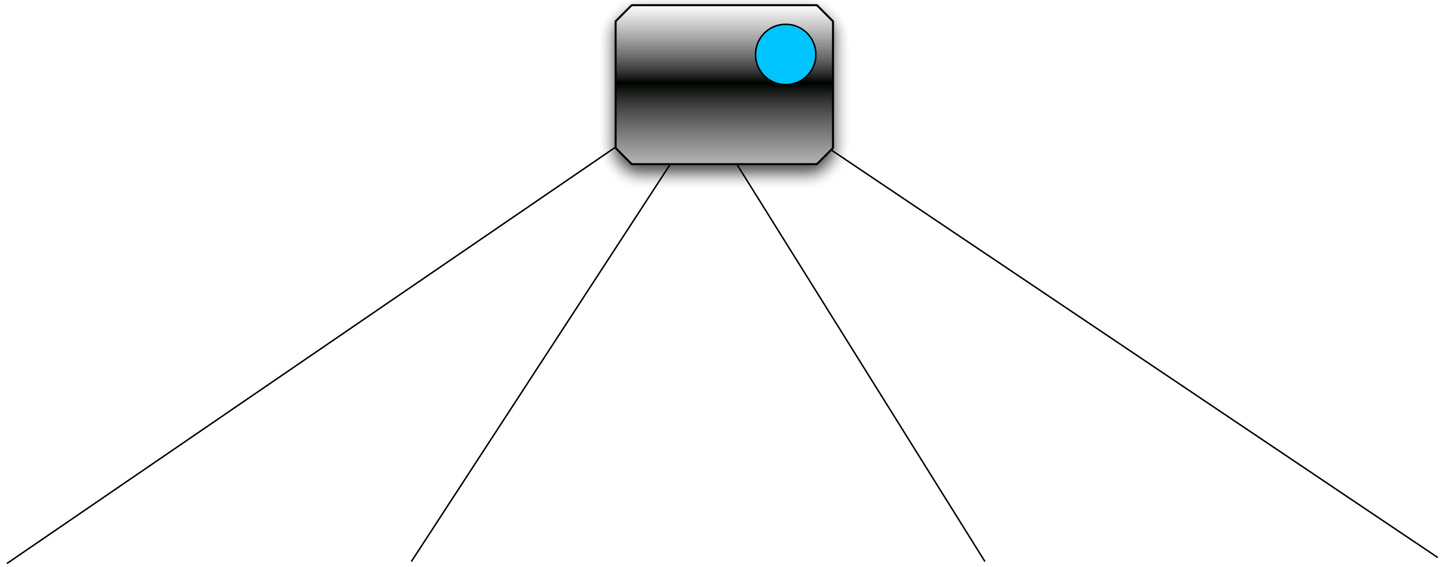




All for one strategy



All for one strategy



Recap

- No more defensive programming
- Let it crash
- Errors flow up the topology not the stack

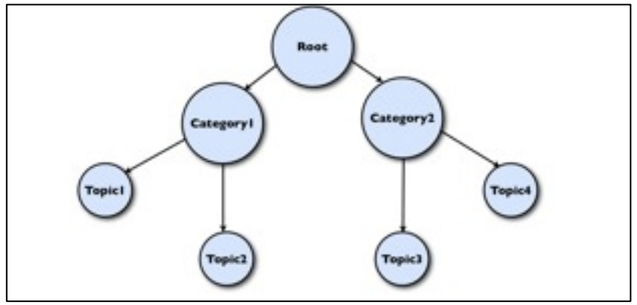
Become

- BECOME - dynamically redefines Actor's behavior
- Triggered reactively by receive of message
- Behaviors are stacked & can be pushed or popped

Why would I want to do that?

- Implement graceful degradation
- Adaptively transform to ActorPool or Router
- Use your imagination

Throttling

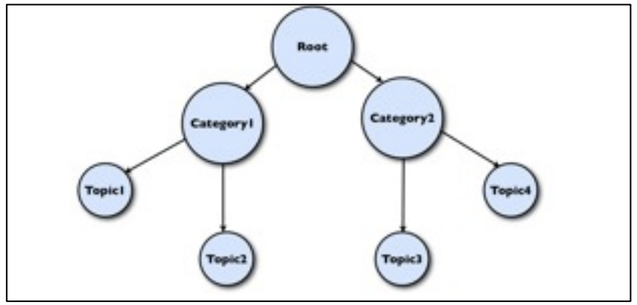


Throttling

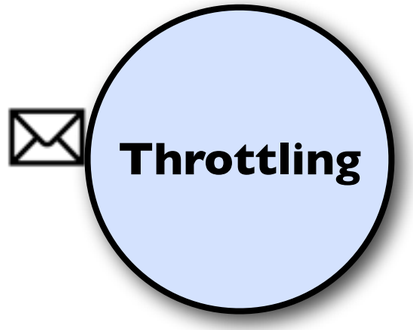
Throttling

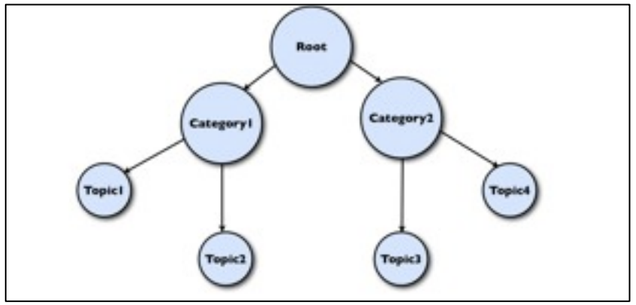
Interceptor

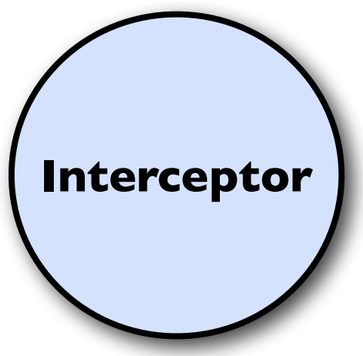
Detector

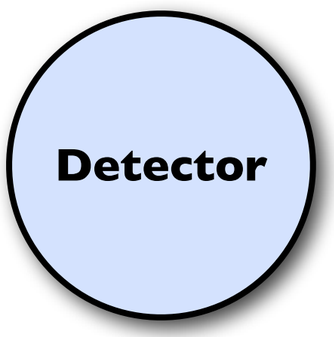


Throttling


Throttling


Root
Category1
Category2
Topic1
Topic2
Topic3
Topic4


Interceptor

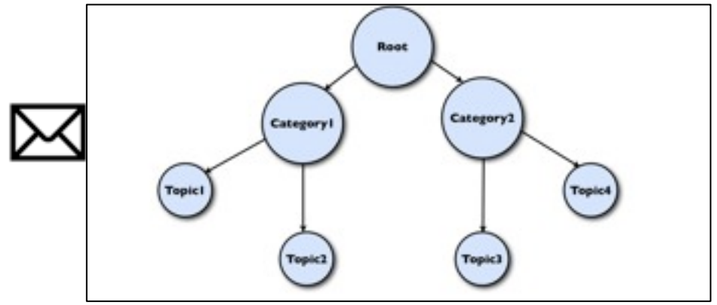

Detector

Throttling

Throttling

Interceptor

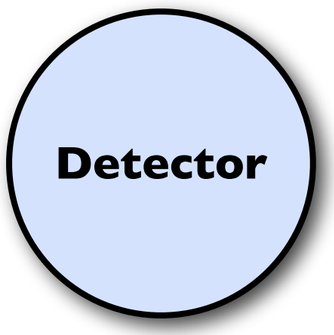
Detector

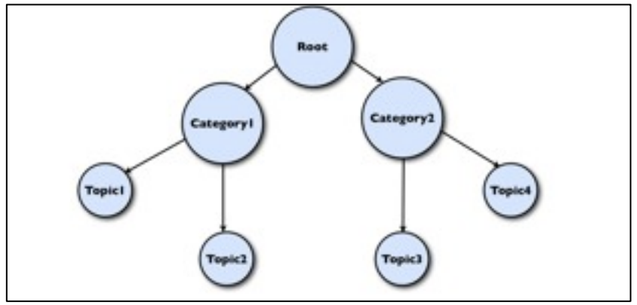


Throttling


Throttling

Interceptor


Detector

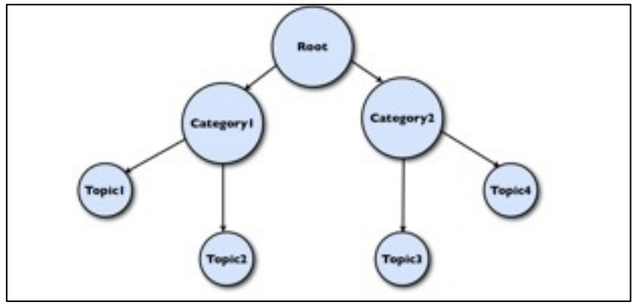

Root
Category1
Category2
Topic1
Topic2
Topic3
Topic4

Throttling

Throttling

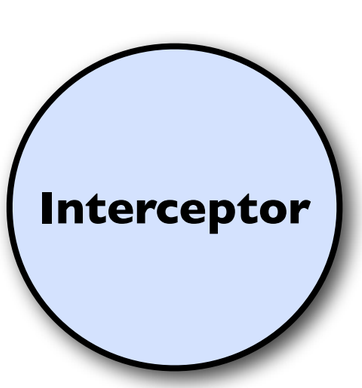
Interceptor

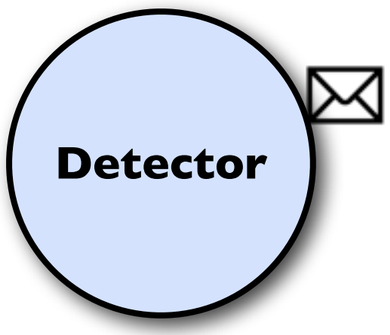
Detector

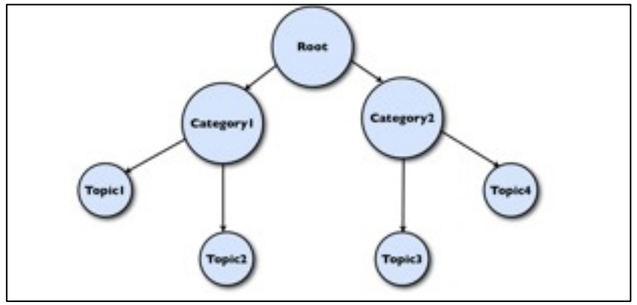


Throttling

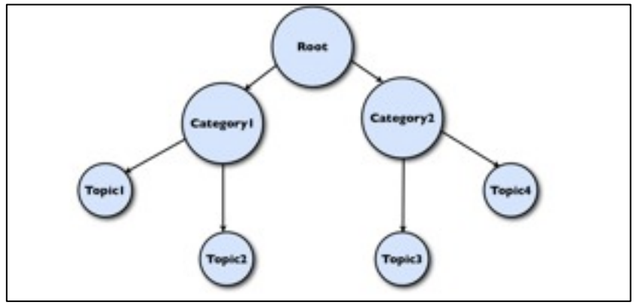

Throttling


Interceptor

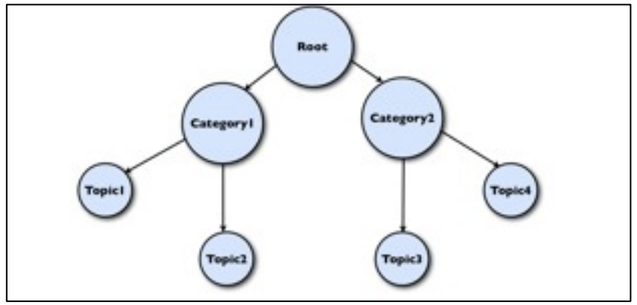

Detector

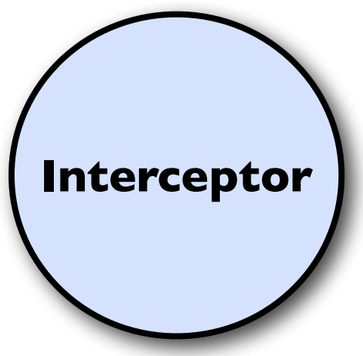

Root
Category1
Category2
Topic1
Topic2
Topic3
Topic4

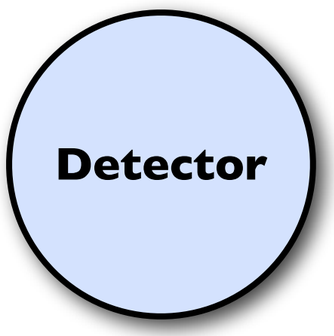
Throttling(timeout)



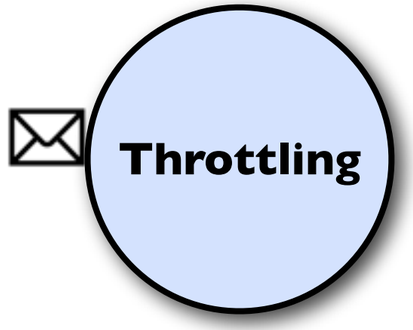
Throttling(timeout)

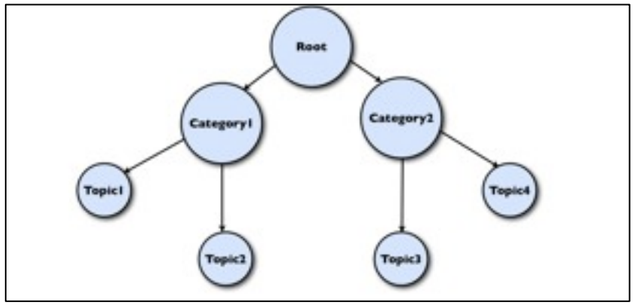


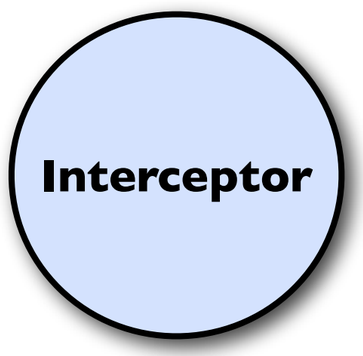


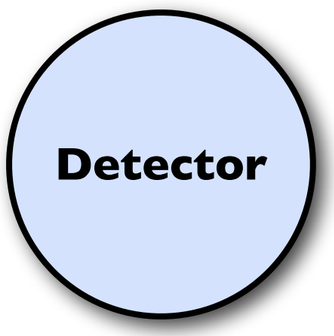


Throttling(timeout)









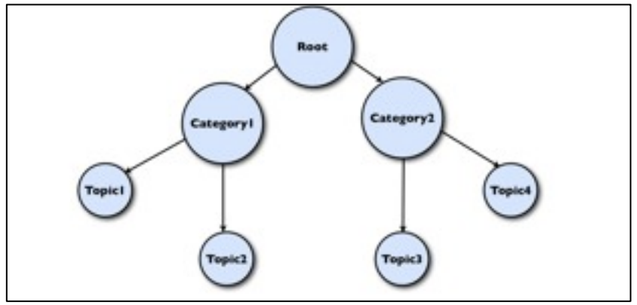
Throttling(timeout)

Throttling

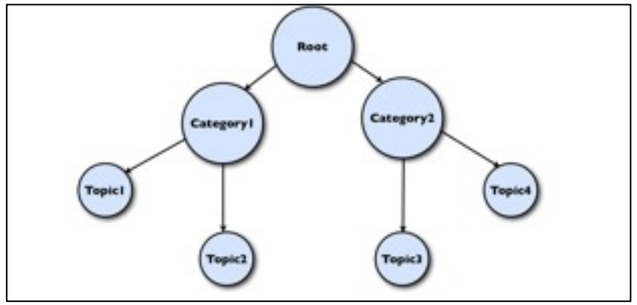
Interceptor

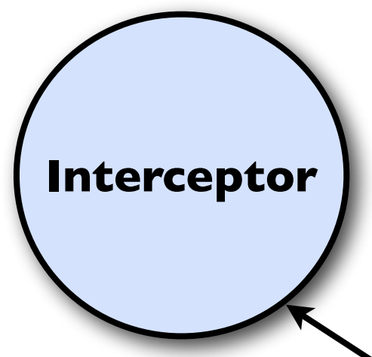
Detector

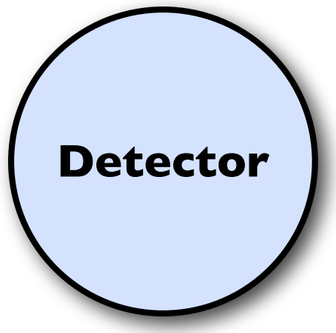
Akka



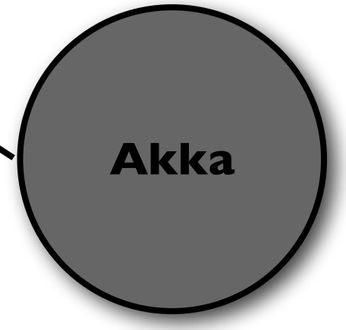
Throttling(timeout)



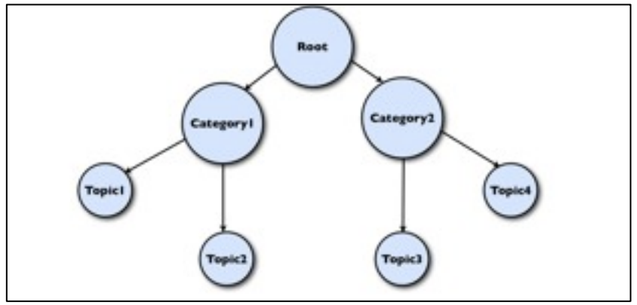


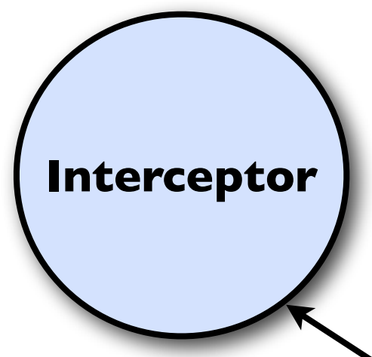


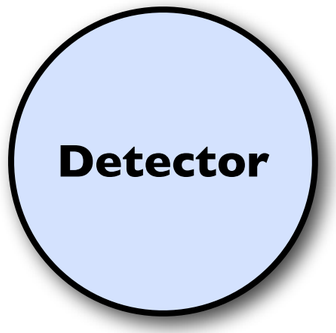
Timeout



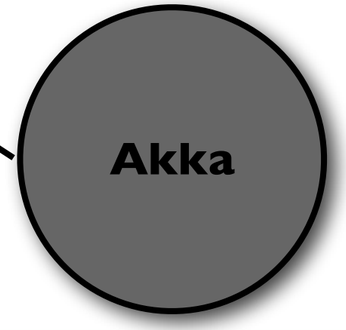
Throttling(timeout)



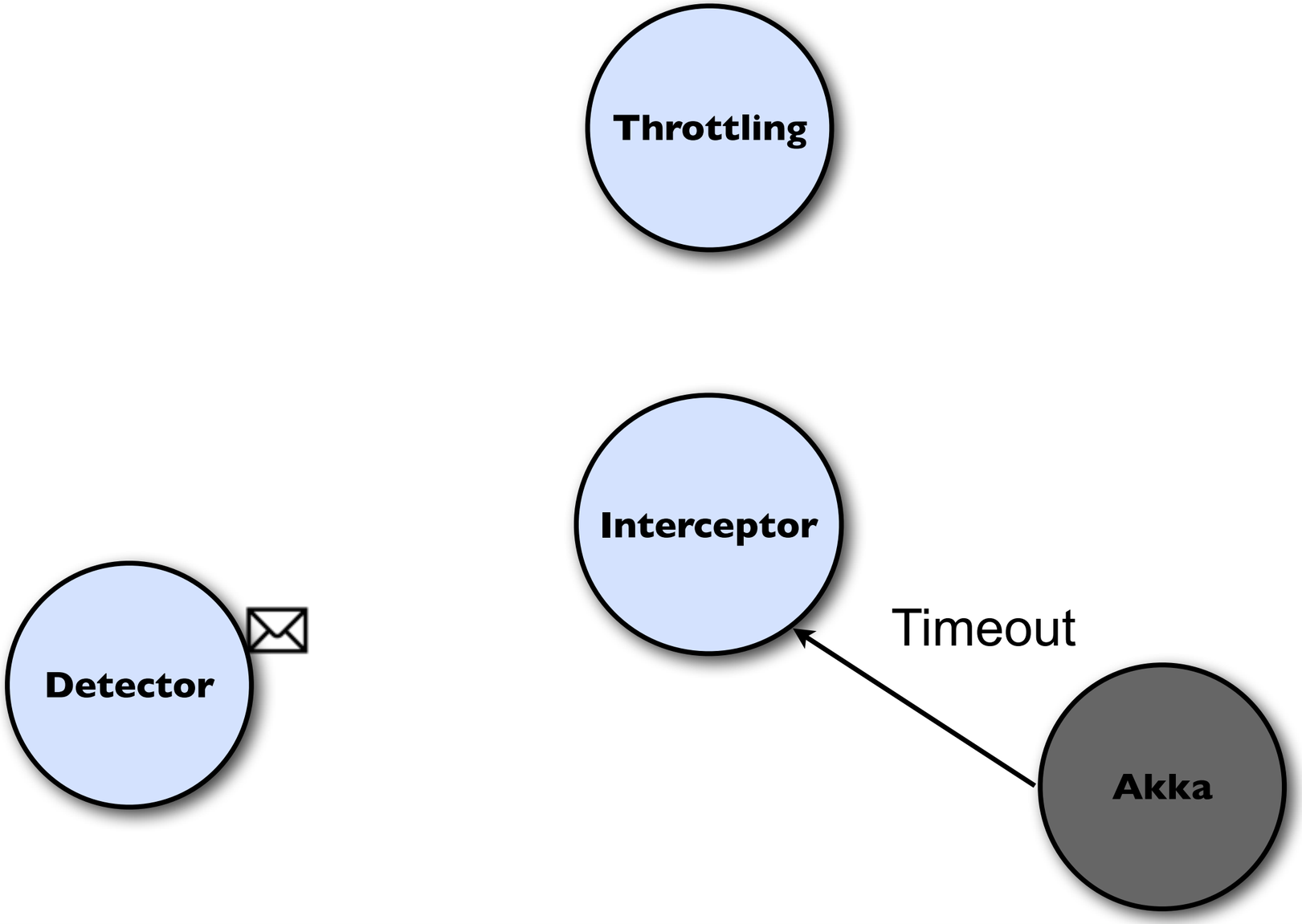


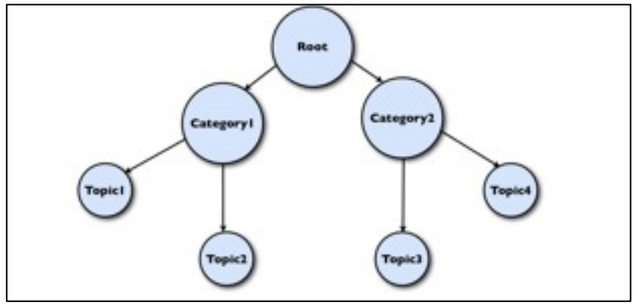


Timeout



Throttling(timeout)

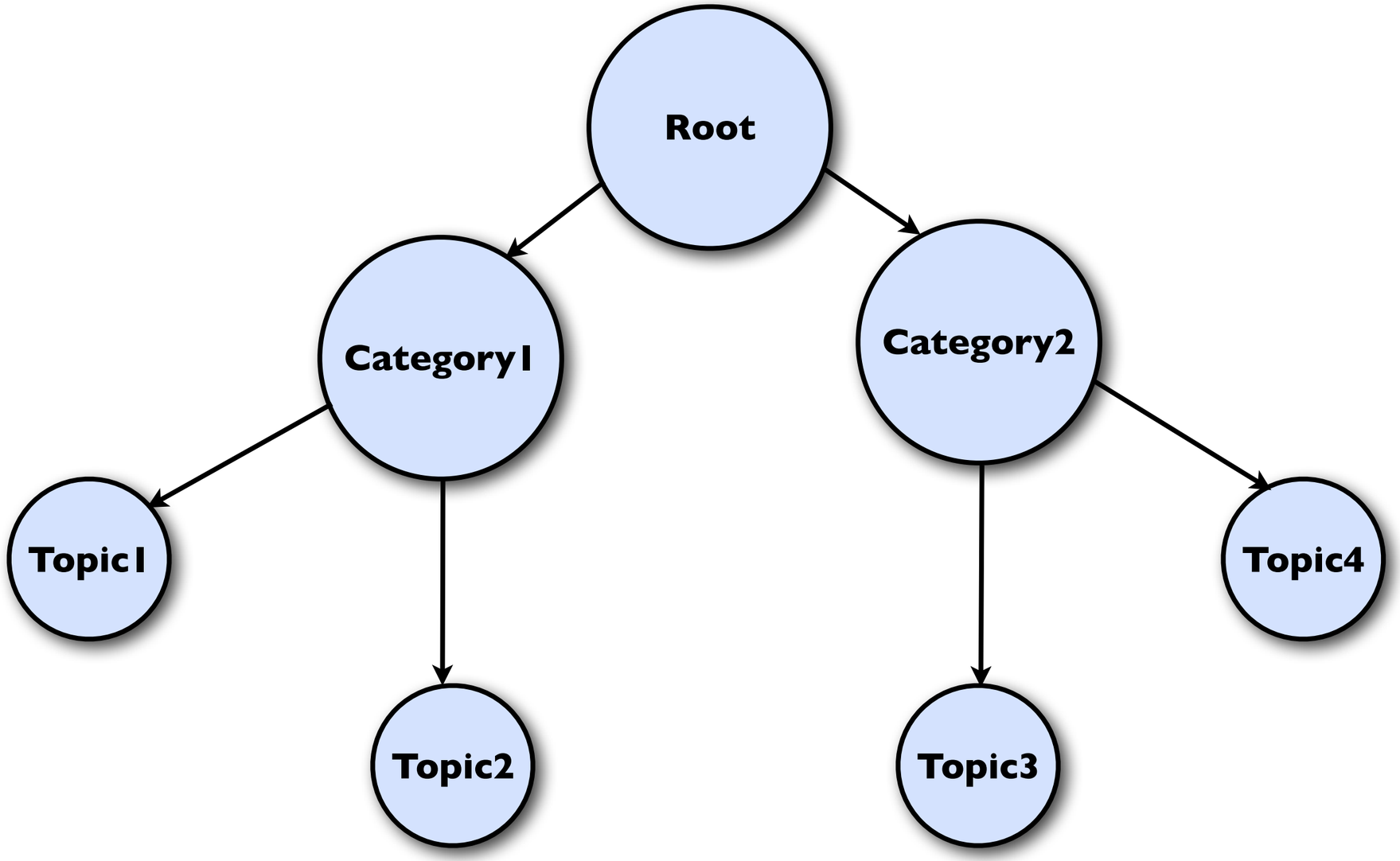




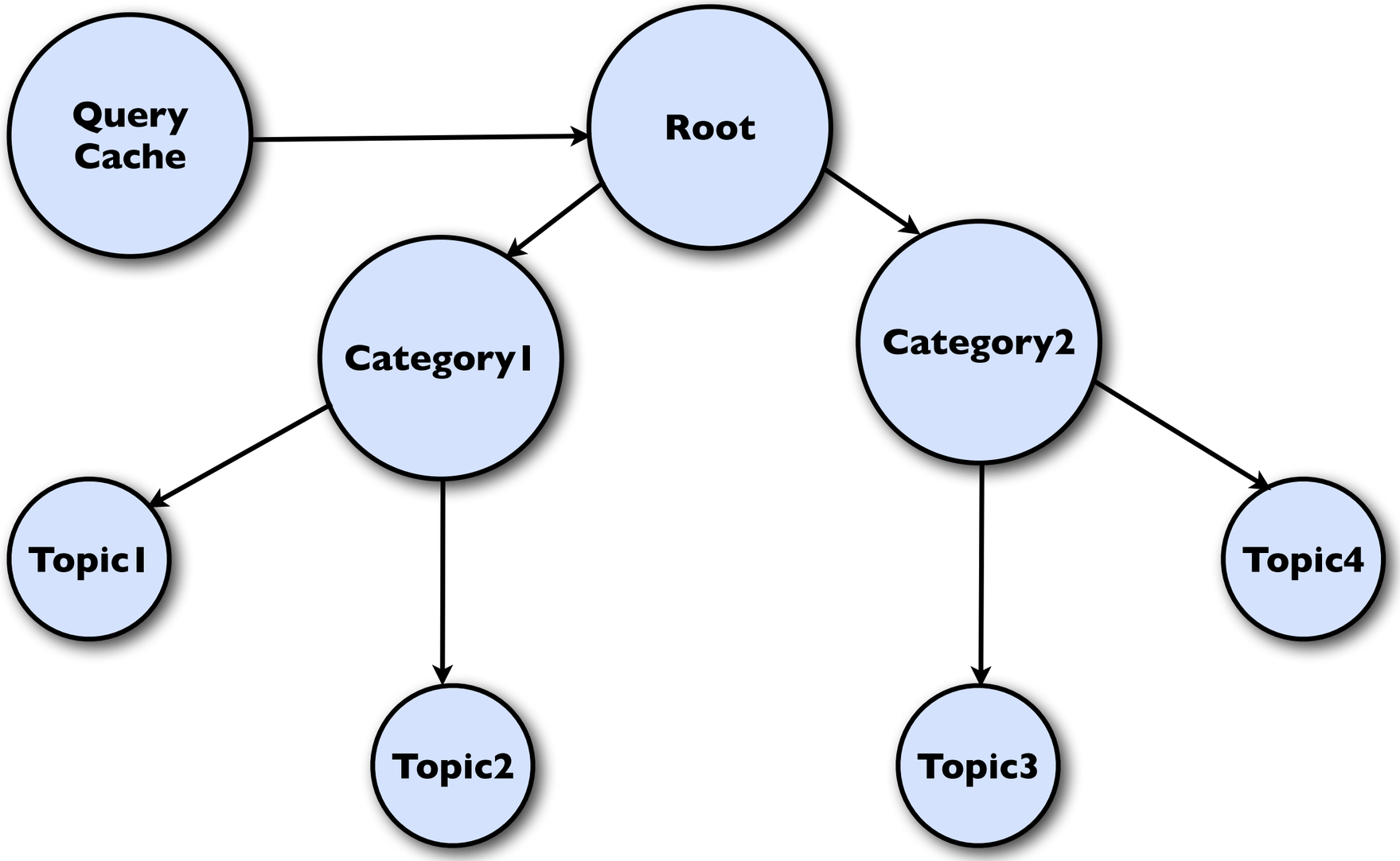
Self healing

Demo

Adding caching layer



Adding caching layer



Caching Actor

```
class QueryCacheActor(service: ActorRef, cachedQueries: Int) extends Actor {
  val cache = new LruCache[String, HotelResponse](cachedQueries)
  def receive: Receive = {
    case HotelQuery(query) =>
      val queryString = query.getSearchString.toLowerCase
      Option(cache get queryString) match {
        // query is not cached.
        case None =>
          val listener = sender
          val interceptor = context.actorOf(Props(
                  new QueryCacheInterceptor(queryString, listener, context.self)))
          service.tell(HotelQuery(query), interceptor)
        case Some(cached) =>
          sender ! cached
      }
    case UpdateQueryCache(query, response) =>
      cache.put(query, response)
  }
}
```

Cache Interceptor

```
class QueryCacheInterceptor(query: String,
    listener: ActorRef, cache: ActorRef) extends Actor {
  def receive: Receive = {
    case response: HotelResponse =>
      listener ! response
      cache ! UpdateQueryCache(query, response)
      context stop self
  }
}
```

Next step

Non-blocking all the way

How is using it?

How is using it?

Questions?

EØF

https://github.com/jsuereth/spring-akka-sample