Integration of the ACTOR MODEL into MAINSTREAM TECH

PHILIPP HALLER | Typesafe

What is Mainstream?

What is Mainstream?

ONE ANSWER:

What is Mainstream?

One Answer:

→ Platform/Execution Model:

- JIT compiled bytecode
- Threading based on OS processes or native POSIX threads

What is Mainstream?

One answer:

→ Platform/Execution Model:

- JIT compiled bytecode
- Threading based on OS processes or native POSIX threads

→ Static Typing

What is Mainstream?

One answer:

- **Platform/Execution model:**
  - JIT compiled bytecode
  - Threading based on OS processes or native POSIX threads
- **Static typing**

In our case

THE JVM + SCALA

SCALA

SCALA
What is it?

SCALA
What is it?

- OBJECT-ORIENTED
- FUNCTIONAL
- AGILE, LIGHTWEIGHT SYNTAX
- SAFE, PERFORMANT

with strong static typing

SCALA
Where does it come from?

SCALA
Where does it come from?

- **1996-2000**
  Pizza, GJ, Java generics, `javac`
- **2003-2006**
  The Scala "Experiment"

Who's Using Scala?

Sunday, October 21, 12

Scala Actors

- **LONGTIME CORE CONCURRENCY LIB**
  In the stdlib from early-on, (since Scala 2.1.7)
- **ERLANG-LIKE**
  Very close to Erlang's actor-like processes

```
val shop = actor {
  while (true) {
    receive {
      case Order(item) =>
        val order = handleOrder(item, sender)
        sender ! Ack(order)
      case Cancel(order) =>
        cancelOrder(order)
        sender ! Cancelled(order)
    }
  }
}
```

SCALA ACTORS

Early Goals

SCALA ACTORS
Early Goals

LIBRARY-BASED DESIGN

- *Unclear which concurrency paradigm will "win"*
- *Scalability: enable flexible concurrency libraries*

Scala Actors
Early Goals

Library-based design
- Unclear which concurrency paradigm will "win"
- Scalability: enable flexible concurrency libraries

Embrace the host language
- "Competitive" programming interface

SCALA ACTORS
Early Goals

LIBRARY-BASED DESIGN
- *Unclear which concurrency paradigm will "win"*
- *Scalability: enable flexible concurrency libraries*

EMBRACE THE HOST LANGUAGE
- *"Competitive" programming interface*

LIGHTWEIGHT EXECUTION ENVIRONMENT
- *Event-based actors much more lightweight*
- *Integration with JVM threads*

Event-Based ACTORS

IDEA: *Introduce an event-based* **react** *operation which takes a continuation closure:*

```
loop {
  react {
    case Order(item)   => ...
    case Cancel(order) => ...
  }
}
```

Actor detached from a thread while waiting to receive a message

⟹ Scales to much larger numbers of actors

⟹ Uses work-stealing thread pool for message processing

Integrating EVENTS & THREADS

EVENT-BASED & BLOCKING

Actors support both event-based `react` *and blocking operations*

MANAGED BLOCKING

Thread pool resizing

SEND/RECEIVE ANYWHERE

Message send and `receive` *also available on regular, non-actor threads of the JVM*

Philipp Haller, Martin Odersky: Scala Actors: *Unifying thread-based and event-based programming*. Theor. Comput. Sci, 2009 (citations: 110)

SCALA ACTORS: *Experience*

SCALA ACTORS: *Experience*

→ LIBRARY-BASED DESIGN WORKS WELL

Scala Actors: Experience

- **Library-based design works well**
- **Scalability**
  Through work-stealing thread pool

Proven in production!
For example, at Twitter during Obama inauguration

Scala Actors: *Experience*

- **Library-based design works well**
- **Scalability**
  Through work-stealing thread pool

Proven in production!
For example, at Twitter during Obama inauguration

- **Adoption**
  By many commercial users

Scala Actors: *Experience*

Library-based design works well

Scalability

Through work-stealing thread pool

Proven in production!

For example, at Twitter during Obama inauguration

Adoption

By many commercial users

Robust!

Only a handful of known issues even after years of low maintenance

SCALA ACTORS: *Challenges*

SCALA ACTORS: *Challenges*

ISOLATION

- *Actors are objects => direct access to its methods/state possible unless precautions are taken*
- *Exchange of mutable messages by reference*

SCALA ACTORS: *Challenges*

ISOLATION

- *Actors are objects => direct access to its methods/state possible unless precautions are taken*
- *Exchange of mutable messages by reference*

FAULT TOLERANCE

Restarting an actor is impractical, since it requires updating all references to that same logical actor in the entire system

SCALA ACTORS: *Challenges*

ISOLATION

- Actors are objects => direct access to its methods/state possible unless precautions are taken
- Exchange of mutable messages by reference

FAULT TOLERANCE

Restarting an actor is impractical, since it requires updating all references to that same logical actor in the entire system

REMOTING ONLY RUDIMENTARY

SCALA ACTORS: *Challenges*

ISOLATION

- Actors are objects => direct access to its methods/state possible unless precautions are taken
- Exchange of mutable messages by reference

FAULT TOLERANCE

Restarting an actor is impractical, since it requires updating all references to that same logical actor in the entire system

REMOTING ONLY RUDIMENTARY

MESSAGE PILE-UP

Erlang's queue model can lead to message pile-up, linear performance degradation

ISOLATION *through Uniqueness*

ISOLATION through Uniqueness

- Avoiding data races when exchanging mutable objects
- No need for full ownership types

Isolation through Uniqueness

- Avoiding data races when exchanging mutable objects
- No need for full ownership types

Foundations and soundness proof:

Philipp Haller, Martin Odersky. Capabilities for uniqueness and borrowing. ECOOP 2010

ISOLATION *through Uniqueness*

- Avoiding data races when exchanging mutable objects
- No need for full ownership types

FOUNDATIONS AND SOUNDNESS PROOF:

Philipp Haller, Martin Odersky. Capabilities for uniqueness and borrowing. ECOOP 2010

EXAMPLE: *using the prototype of a Scala compiler plug-in:*

```
actor {
  val buf: ArrayBuffer[Int] @unique =
    new ArrayBuffer[Int](3)
  buf ++= Array(0, 1, 2)
  someActor ! buf
}
```

ok!

```
actor {
  val buf: ArrayBuffer[Int] @unique =
    new ArrayBuffer[Int](3)
  buf ++= Array(0, 1, 2)
  someActor ! buf
  println(buf.remove(0))
}
```

illegal!

Requirements of Industry

Early goals not enough, need also:

- **High performance**
- **Extensive remoting capabilities**
  - *Support for third party remote transports*
  - *Flexible configuration*
- **Pragmatic solutions to challenges**
- **Short release cycles**
  - *Until 2.10.0 only infrequent releases of Scala distribution*

Enter:
AKKA

AKKA: Actors Reloaded

Main Differences:

- Distinction between actors and `ActorRefs` to avoid direct access to actor instances
- Actor-global event loop replaces blocking-style `react`
- Unhandled messages not kept in mailbox

AKKA: Actors Reloaded

Benefits

- Simpler implementation
- Higher performance
- Simplified fault-tolerance (actor restarts made easy)
- `ActorRefs` enable transparent remoting

AKKA's Actor API

- **SIMILAR TO** `scala.actors` **API**
- **EXAMPLE:**

```
class Shop extends Actor {
  def receive = {
    case Order(item) =>
      val order = handleOrder(item, sender)
      sender ! Ack(order)
    case Cancel(order) =>
      cancelOrder(order)
      sender ! Cancelled(order)
  }
}

val shop: ActorRef = system.actorOf(Props[Shop])
```

Partial Functions

BLOCK WITH PATTERN MATCHING CASES = PARTIAL FUNCTION

TYPE DEFINITIONS:

```
trait Function1[-A, +B] {
  def apply(x: A): B
}

trait PartialFunction[-A, +B]
  extends Function1[A, B] {
  def isDefinedAt(x: A): Boolean
  ...
}
```

USING *Partial Functions*

- `receive` *returns global message handler*
- *handler activated when message can be removed from mailbox*
- *will never leave a message in the mailbox*
- *if no pattern matches removed message, an event is published to the enclosing container ("actor system"), signaling an unhandled message*
- *works well with case class instances: matching on receiver's side*
- *use of partial functions as message handlers as well as case classes for message types introduced by Scala Actors*

Sending Messages

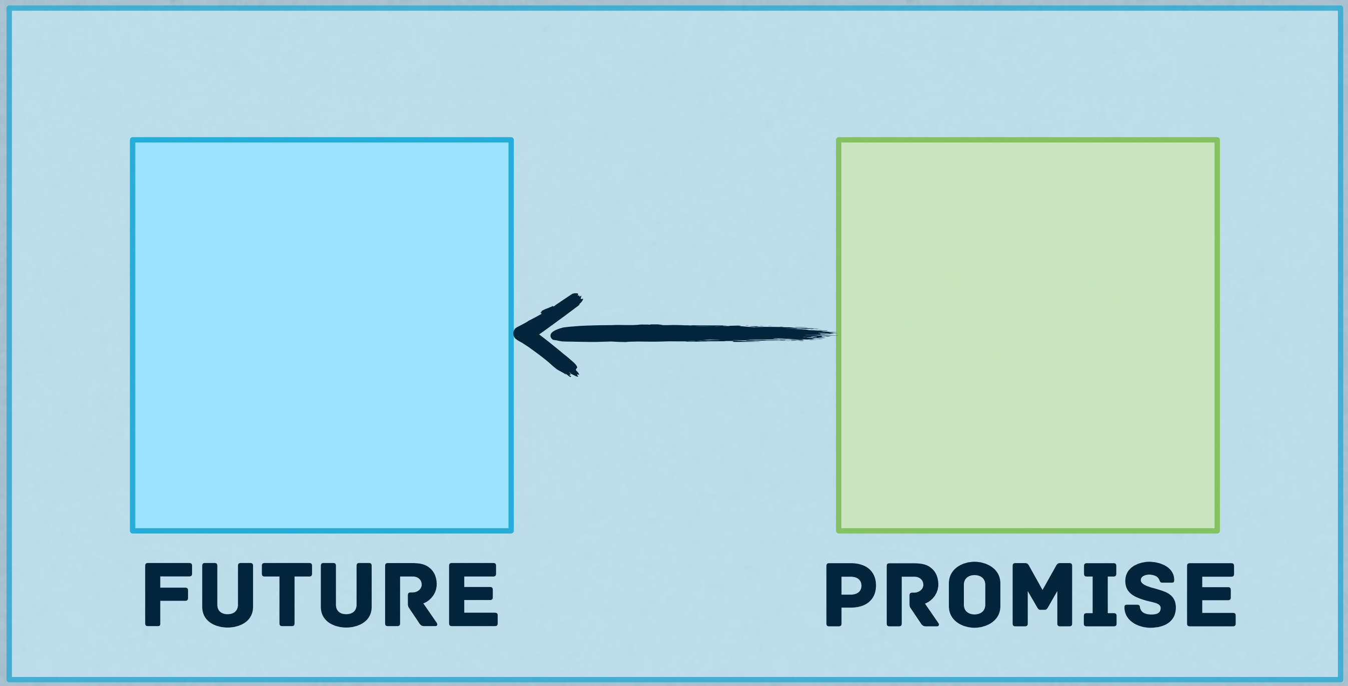
- *Like Scala Actors, Akka adopts the principal message send operator from Erlang:*
  - `a ! msg` *asynchronously sends* `msg` *to* `a`
- *Other constructs adopted from Scala Actors:*
  - `a forward msg, sender ! msg`
  - `a ? msg` *asynchronously sends* `msg` *to* `a` *and immediately returns a future (*`a !! msg` *in* `scala.actors`*)*
- *A future is a placeholder for a response that may eventually be received*

scala.concurrent.

FUTURE & PROMISE

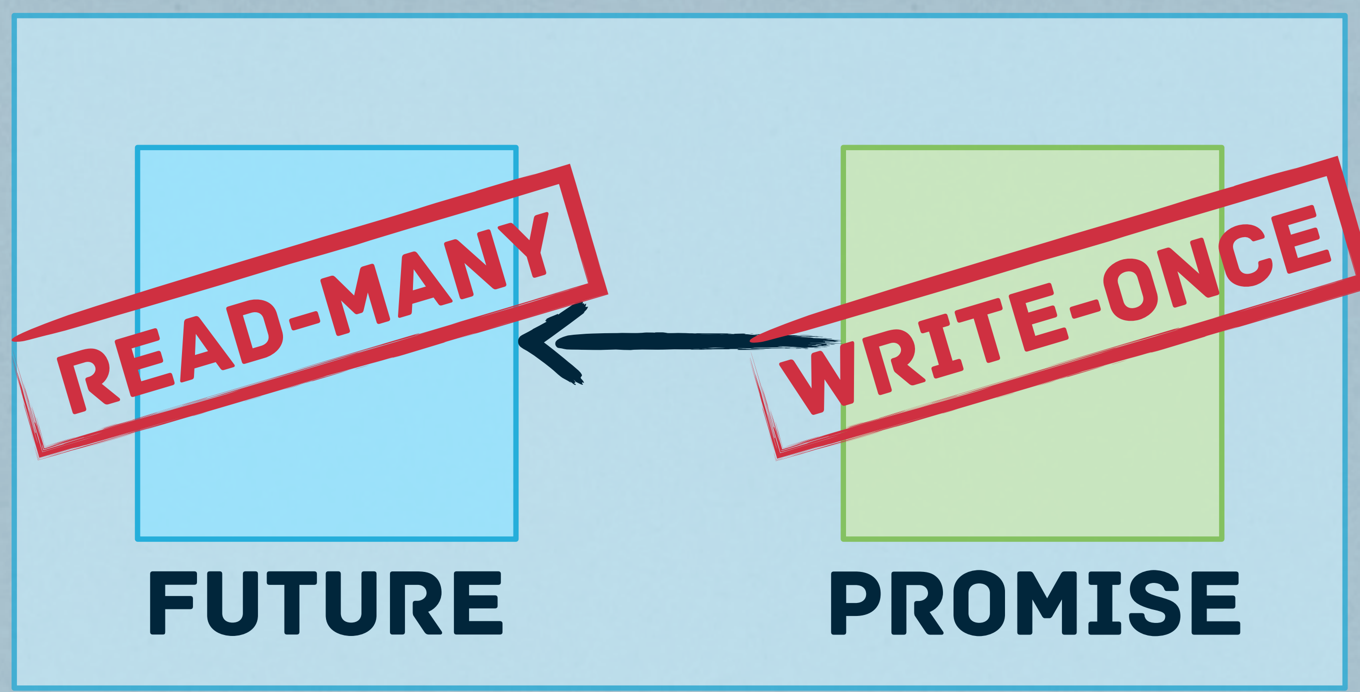
Futures&Promises

can be thought of as a single concurrency model



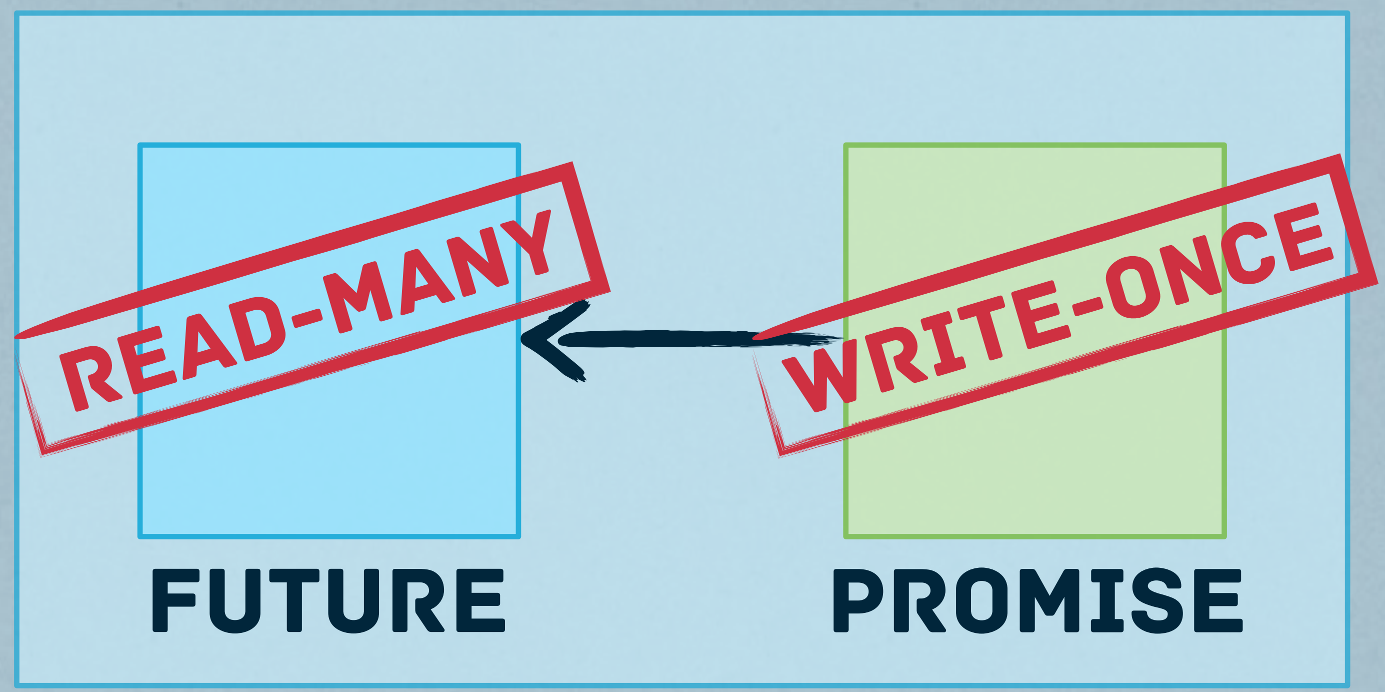
Futures&Promises

CAN BE THOUGHT OF AS A SINGLE CONCURRENCY MODEL



Futures&Promises

Can be thought of as a single concurrency model



Important ops

- ✔ Start async computation
- ✔ Wait for result
- ✔ Assign result value
- ✔ Obtain associated future object

Success & Failure

A PROMISE `p` OF TYPE `Promise[T]` CAN BE COMPLETED IN TWO WAYS...

Success

```
val result: T = ...
p.success(result)
```

Failure

```
val exc = new Exception("something went wrong")
p.failure(exc)
```

Async&NonBlocking

Async&NonBlocking

GOAL: *Do not block current thread while waiting for result of future*

Async&NonBlocking

GOAL: *Do not block current thread while waiting for result of future*

Callbacks

- **REGISTER CALLBACK** *which is invoked (asynchronously) when future is completed*
- **ASYNC COMPUTATIONS NEVER BLOCK** *(except for managed blocking)*

Async&NonBlocking

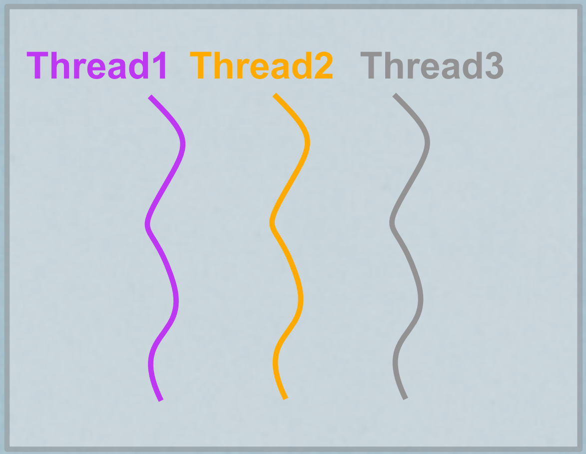
GOAL: *Do not block current thread while waiting for result of future*

Callbacks

- **REGISTER CALLBACK** *which is invoked (asynchronously) when future is completed*
- **ASYNC COMPUTATIONS NEVER BLOCK** *(except for managed blocking)*

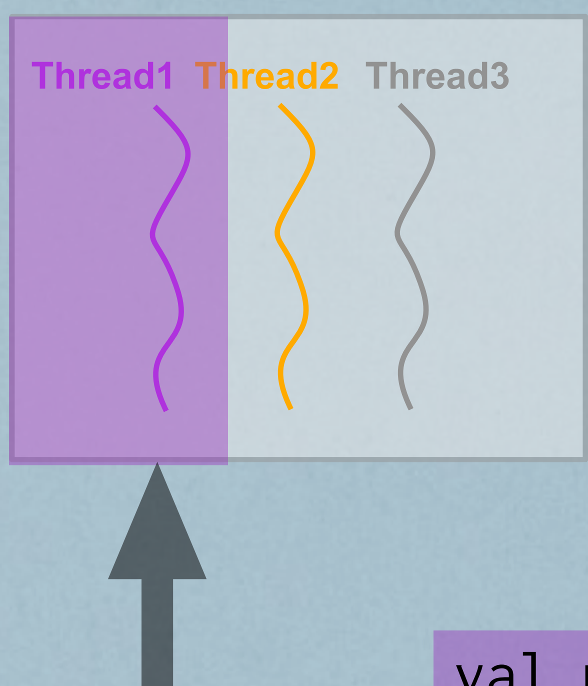
USER DOESN'T HAVE TO EXPLICITLY MANAGE CALLBACKS. HIGHER-ORDER FUNCTIONS INSTEAD!

Futures&Promises
EXAMPLE



Futures&Promises

EXAMPLE

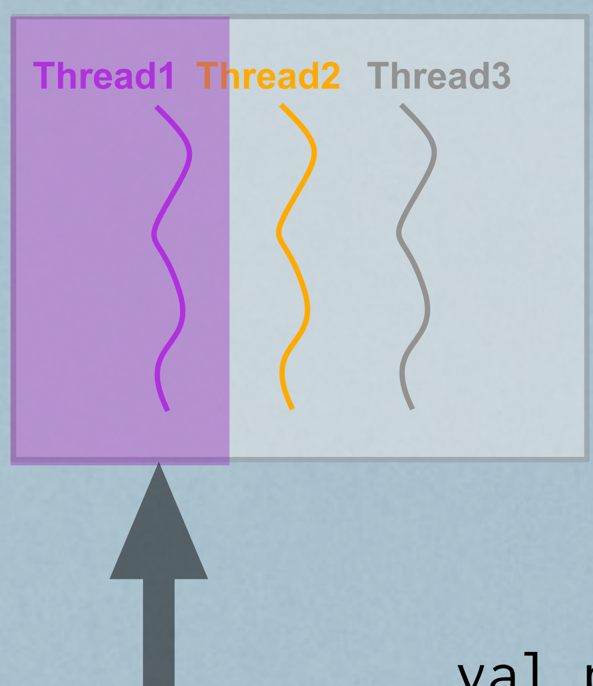


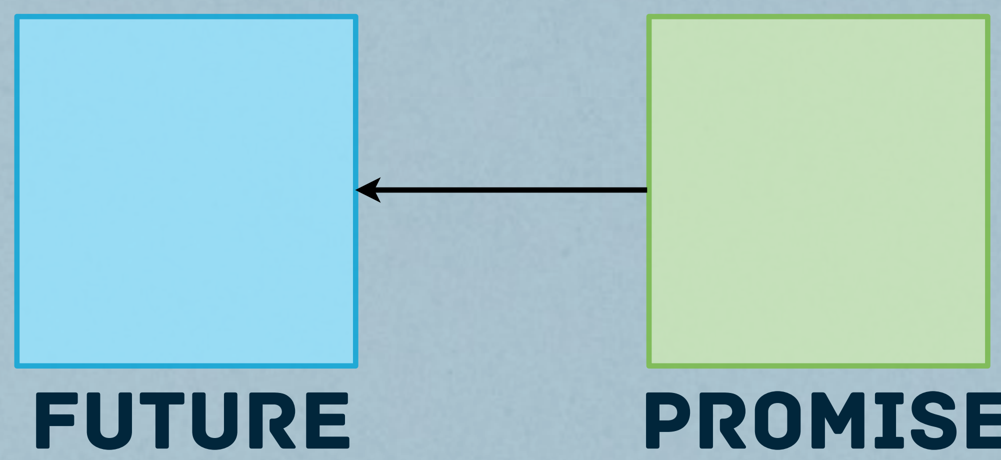
PROMISE

```
val p = Promise[Int]() // Thread 1
```

(CREATE PROMISE)

Futures&Promises
EXAMPLE

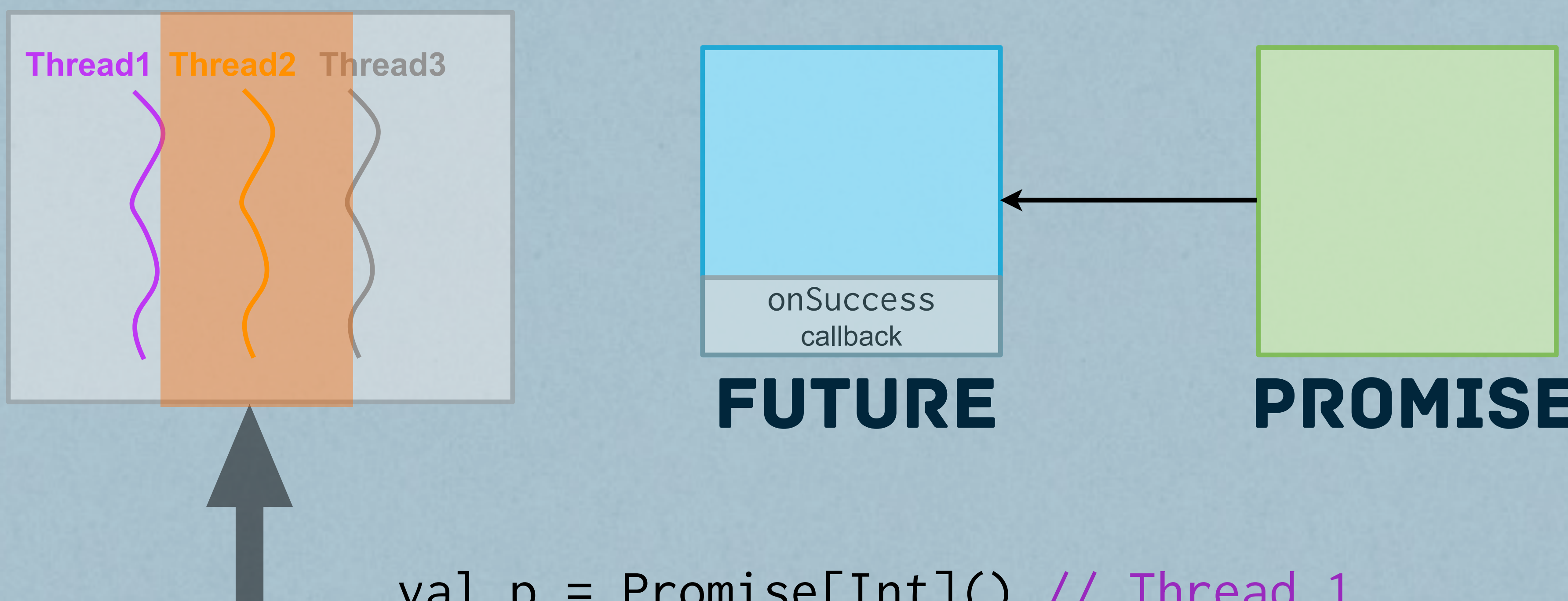





```
val p = Promise[Int]() // Thread 1
val f = p.future       // Thread 1
```

(CREATE PROMISE)
(GET REFERENCE TO FUTURE)

Futures&Promises
EXAMPLE

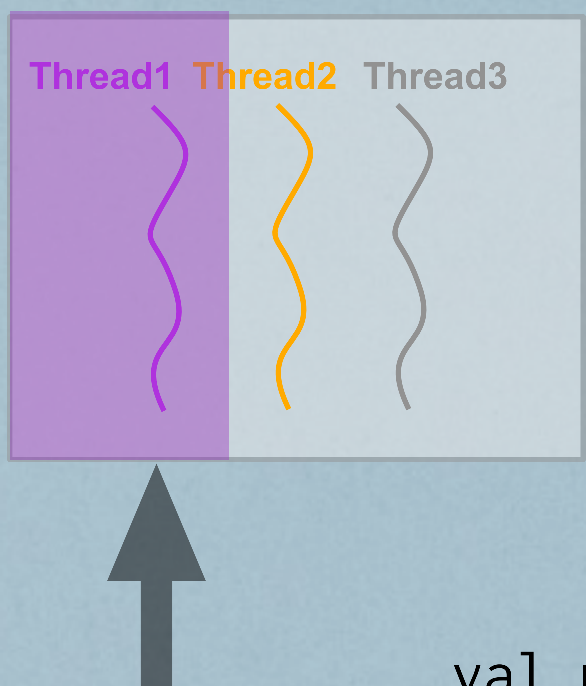


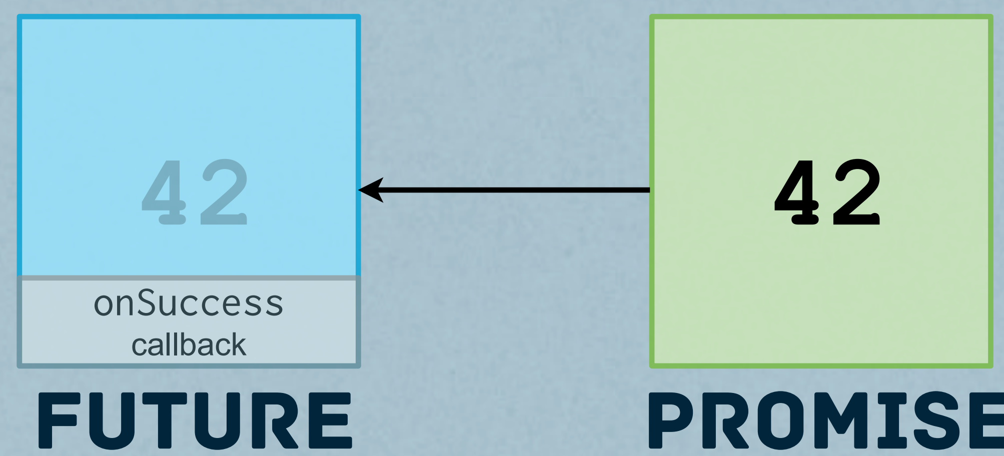

```
val p = Promise[Int]() // Thread 1
val f = p.future       // Thread 1
f onSuccess {            // Thread 2
  case x: Int => println("Successful!")
}
```

(CREATE PROMISE)
(GET REFERENCE TO FUTURE)
(REGISTER CALLBACK)

Futures&Promises

EXAMPLE

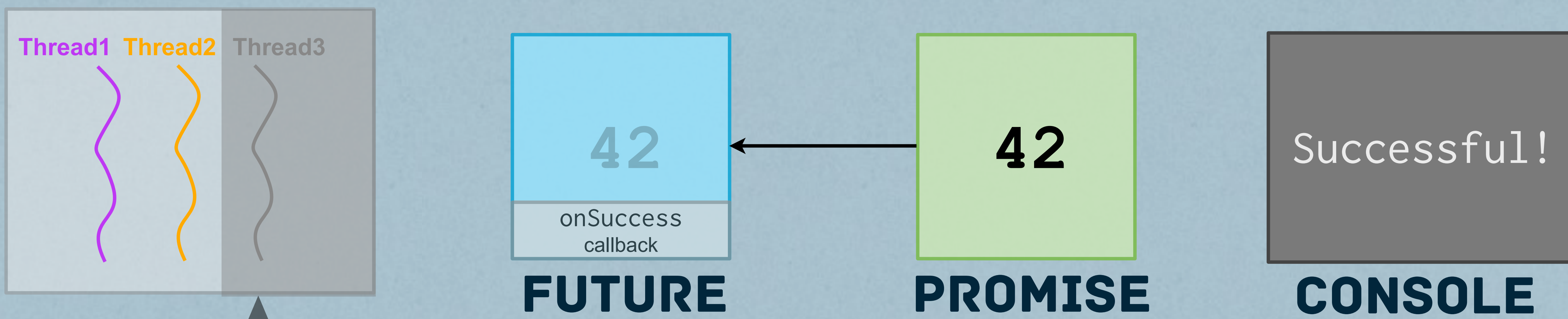





```
val p = Promise[Int]() // Thread 1      (CREATE PROMISE)
val f = p.future       // Thread 1      (GET REFERENCE TO FUTURE)
f onSuccess {           // Thread 2     (REGISTER CALLBACK)
  case x: Int => println("Successful!")
}
p.success(42)          // Thread 1      (WRITE TO PROMISE)
```

Futures&Promises

EXAMPLE




```
val p = Promise[Int]() // Thread 1
val f = p.future       // Thread 1
f onSuccess {           // Thread 2
  case x: Int => println("Successful!")
}                       // Thread
p.success(42)          // Thread 1
```

(CREATE PROMISE)

(GET REFERENCE TO FUTURE)

(REGISTER CALLBACK)

(EXECUTE CALLBACK)

(WRITE TO PROMISE)

NOTE: onSuccess **CALLBACK EXECUTED EVEN IF** f **HAS ALREADY BEEN COMPLETED AT TIME OF REGISTRATION**

Combinators

- **COMPOSABILITY THRU HIGHER-ORDER FUNCS**
- **STANDARD MONADIC COMBINATORS**

```
def map[S](f: T => S): Future[S]
```

```
val purchase: Future[Int] = rateQuote map {
  quote => connection.buy(amount, quote)
}
```

```
def filter(pred: T => Boolean): Future[T]
```

```
val postBySmith: Future[Post] =
  post.filter(_.author == “Smith”)
```

Combinators

- **COMPOSABILITY THRU HIGHER-ORDER FUNCS**
- **STANDARD MONADIC COMBINATORS**

```
def map[S](f: T => S): Future[S]
```

```
val purchase: Future[Int] = rateQuote map {
  quote => connection.buy(amount, quote)
}
```

IF MAP FAILS: purchase is completed with unhandled exception

```
def filter(pred: T => Boolean): Future[T]
```

```
val postBySmith: Future[Post] =
  post.filter(_.author == “Smith”)
```

IF FILTER FAILS: postBySmith completed with NoSuchElementException

Future
THE IMPLEMENTATION

Many operations implemented in terms of promises

SIMPLIFIED EXAMPLE

```
def map[S](f: T => S): Future[S] = {
  val p = Promise[S]()

  onComplete {
    case result =>
      try {
        result match {
          case Success(r) => p success f(r)
          case Failure(t) => p failure t
        }
      } catch {
        case t: Throwable => p failure t
      }
  }
  p.future
}
```

Future
THE REAL IMPLEMENTATION

The real implementation (a) adds an implicit `ExecutionContext`, (b) avoids extra object creations, and (c) catches only non-fatal exceptions:

```
def map[S](f: T => S)(implicit executor: ExecutionContext): Future[S] = {
  val p = Promise[S]()

  onComplete {
    case result =>
      try {
        result match {
          case Success(r) => p success f(r)
          case f: Failure[_] => p complete f.asInstanceOf[Failure[S]]
        }
      } catch {
        case NonFatal(t) => p failure t
      }
  }

  p.future
}
```

scala.concurrent.

EXECUTION CONTEXT

Threadpools...

ARE NEEDED BY:

- **FUTURES** *for executing callbacks and function arguments*
- **ACTORS** *for executing message handlers, scheduled tasks, etc.*
- **PARALLEL COLLECTIONS** *for executing data-parallel operations*

Scala 2.10 introduces

EXECUTION CONTEXTS

Scala 2.10 introduces

EXECUTION CONTEXTS

Goal

PROVIDE GLOBAL THREADPOOL AS PLATFORM SERVICE TO BE SHARED BY ALL PARALLEL FRAMEWORKS

Scala 2.10 introduces

EXECUTION CONTEXTS

Goal

PROVIDE GLOBAL THREADPOOL AS PLATFORM SERVICE TO BE SHARED BY ALL PARALLEL FRAMEWORKS

- `scala.concurrent` *package provides global* `ExecutionContext`
- Default `ExecutionContext` backed by the most recent fork join pool (collaboration with Doug Lea, SUNY Oswego)

Implicit Execution Ctxs

Asynchronous computations are executed on an `ExecutionContext` which is provided implicitly.

```
def map[S](f: T => S)(implicit executor: ExecutionContext): Future[S]

def onSuccess[U](pf: PartialFunction[T, U])
                (implicit executor: ExecutionContext): Unit
```

Implicit parameters enable fine-grained selection of the `ExecutionContext`:

```
implicit val context: ExecutionContext = customExecutionContext
val fut2 = fut1.filter(pred)
               .map(fun)
```

Implicit Execution Ctxs

IMPLICIT `ExecutionContexts` **ALLOW SHARING ECS BETWEEN FRAMEWORKS**

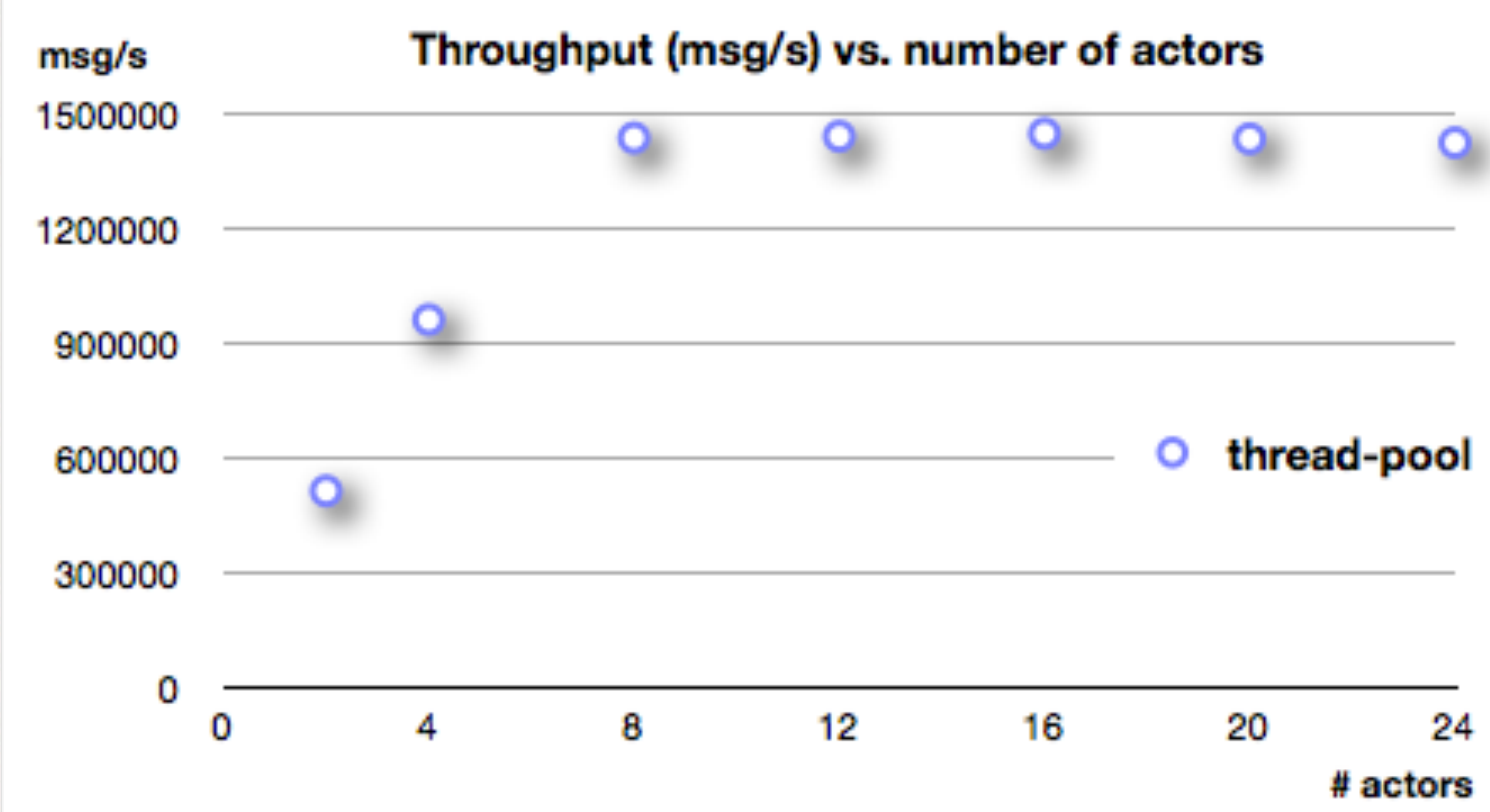
```
def map[S](f: T => S)(implicit executor: ExecutionContext): Future[S]

def onSuccess[U](pf: PartialFunction[T, U])
                (implicit executor: ExecutionContext): Unit
```

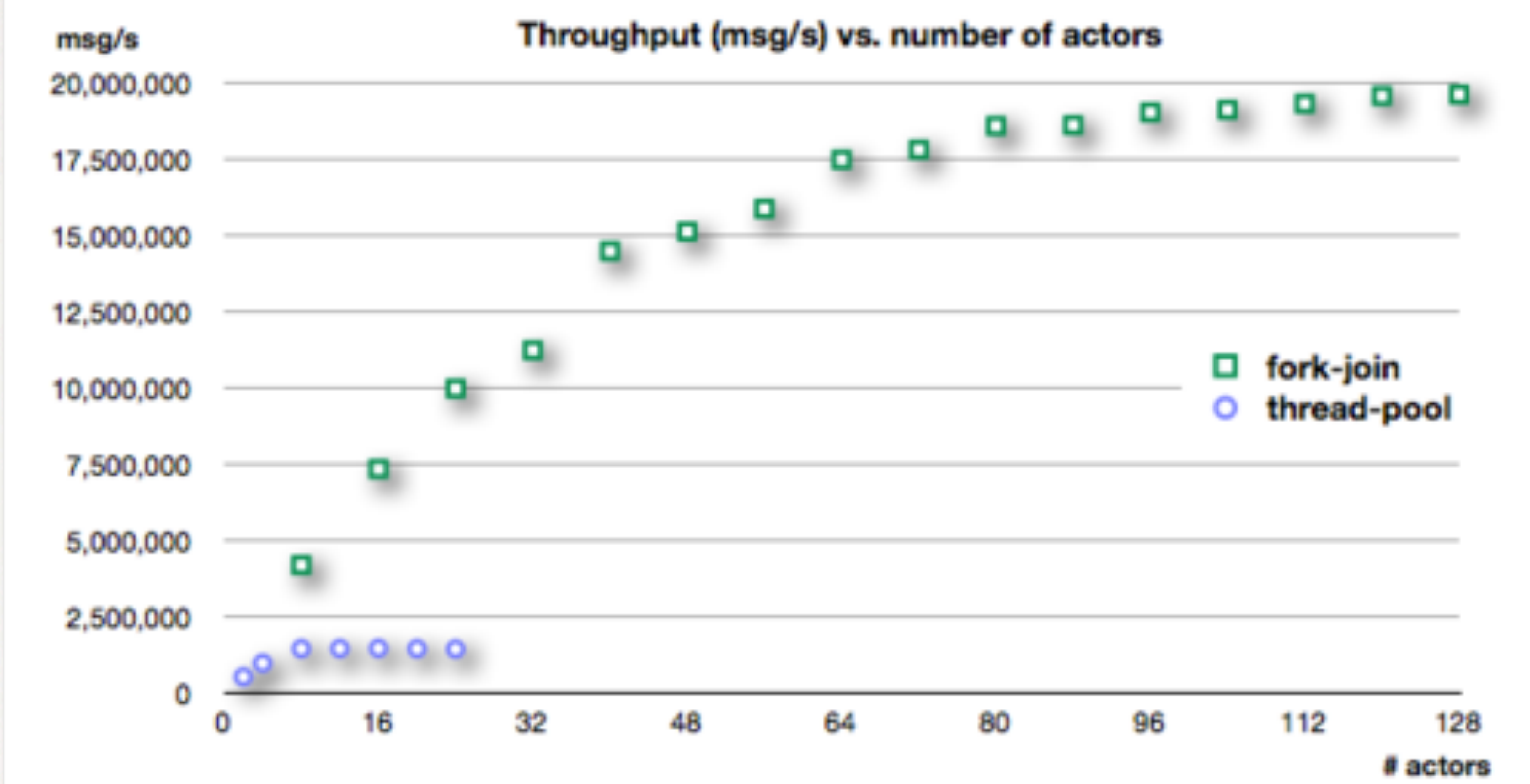
ENABLES FLEXIBLE SELECTION OF EXECUTION POLICY

```
implicit val context: ExecutionContext = customExecutionContext
val fut2 = fut1.filter(pred)
               .map(fun)
```

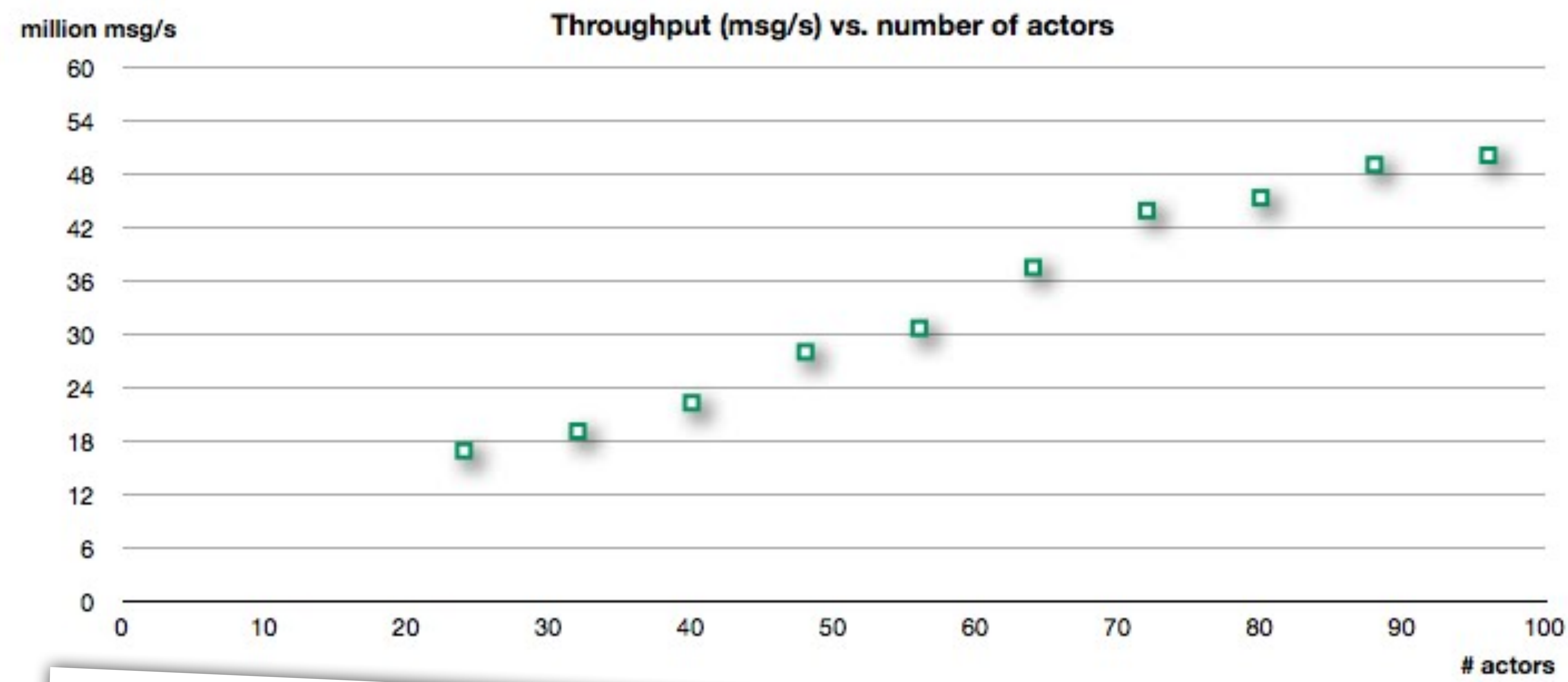
ThreadPoolExecutor



ForkJoinPool



After some tweaks...



+MILLIONS MESSAGES PER SECOND!

What about

FAULT TOLERANCE?

Akka embraces...

LET IT CRASH

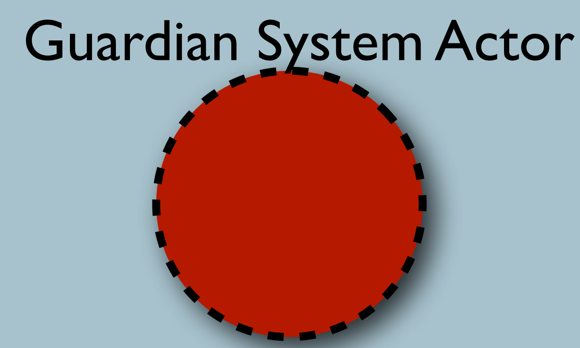
FAULT TOLERANCE

PARENTAL
Automatic Supervision

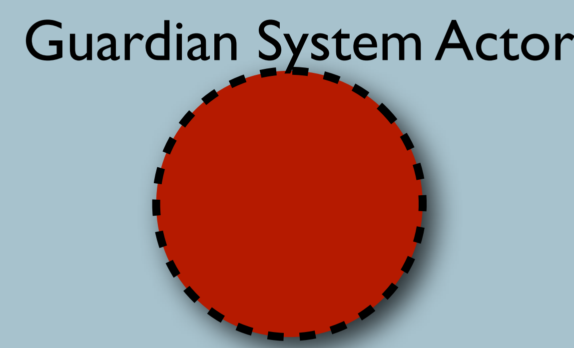
```
// from within an actor
val child = context.actorOf(Props[MyActor], “A”)
```

TRANSPARENT AND AUTOMATIC FAULT HANDLING BY DESIGN.

ACTORS *can form hierarchies...*

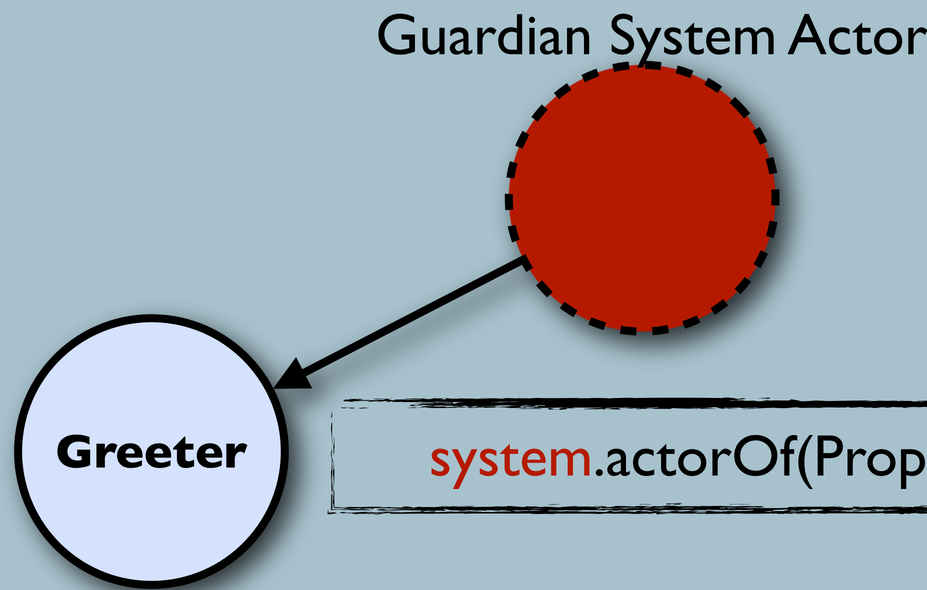


ACTORS *can form hierarchies...*



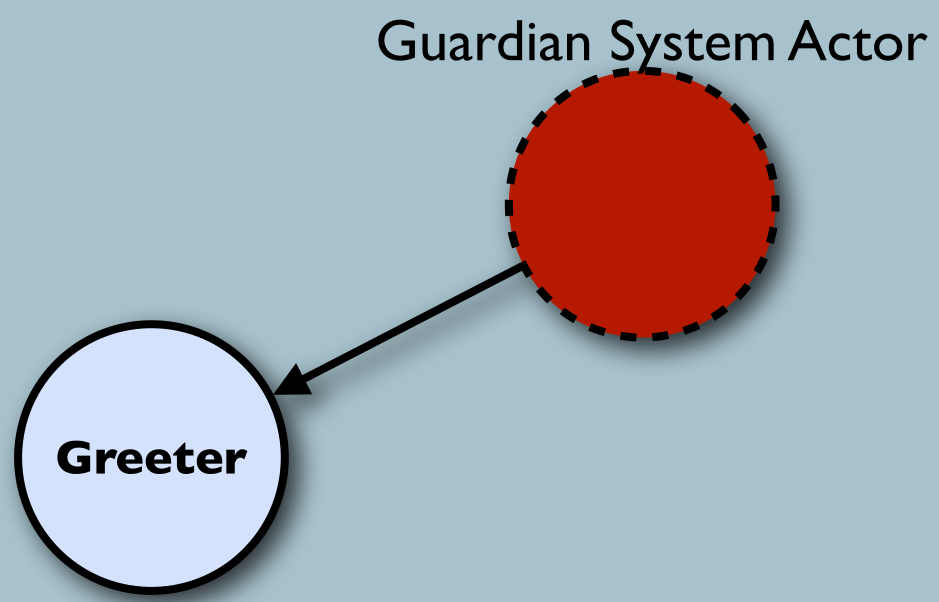

```
system.actorOf(Props[Greeter], "Greeter")
```

ACTORS *can form hierarchies...*



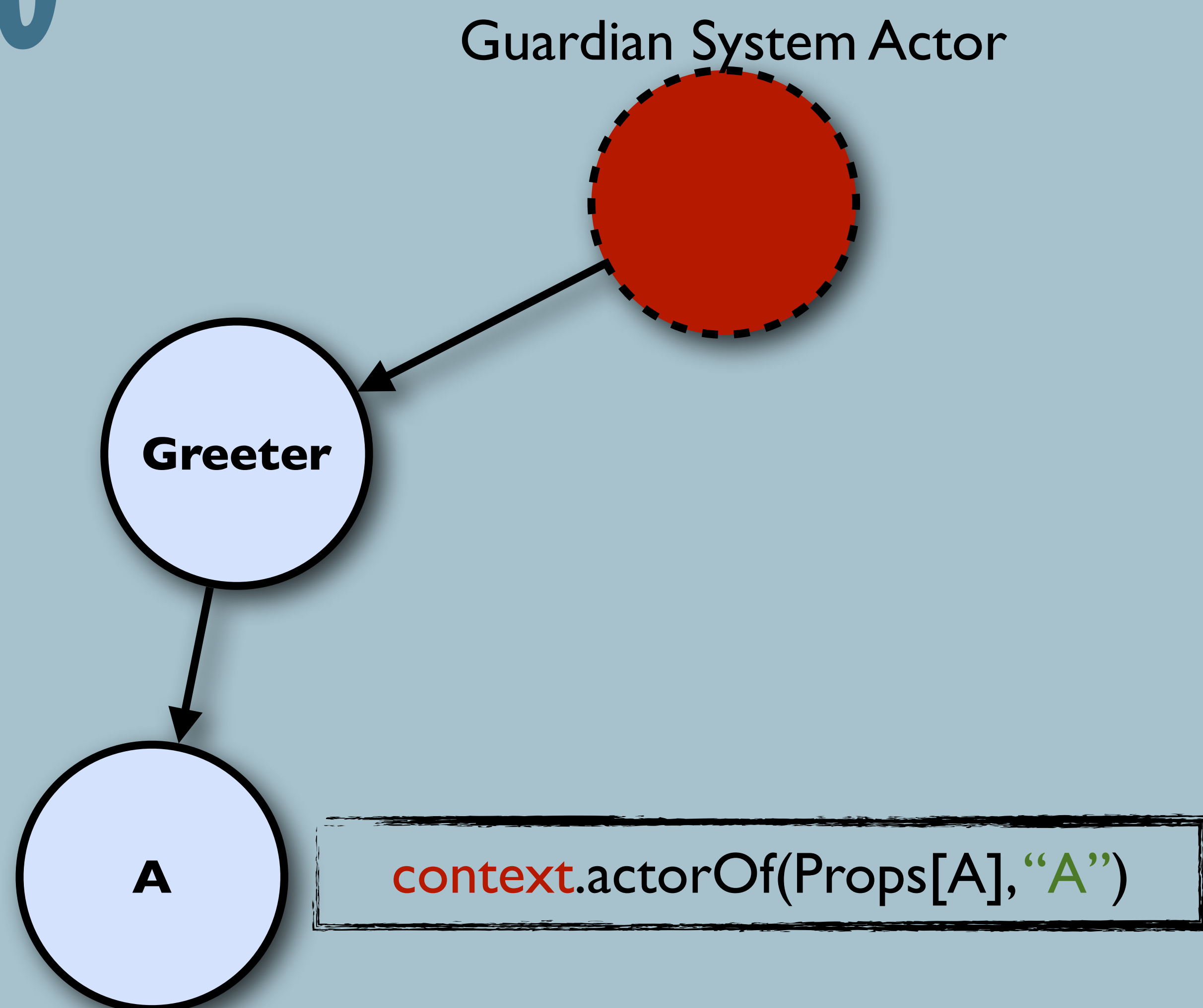

```
system.actorOf(Props[Greeter], "Greeter")
```

ACTORS *can form hierarchies...*

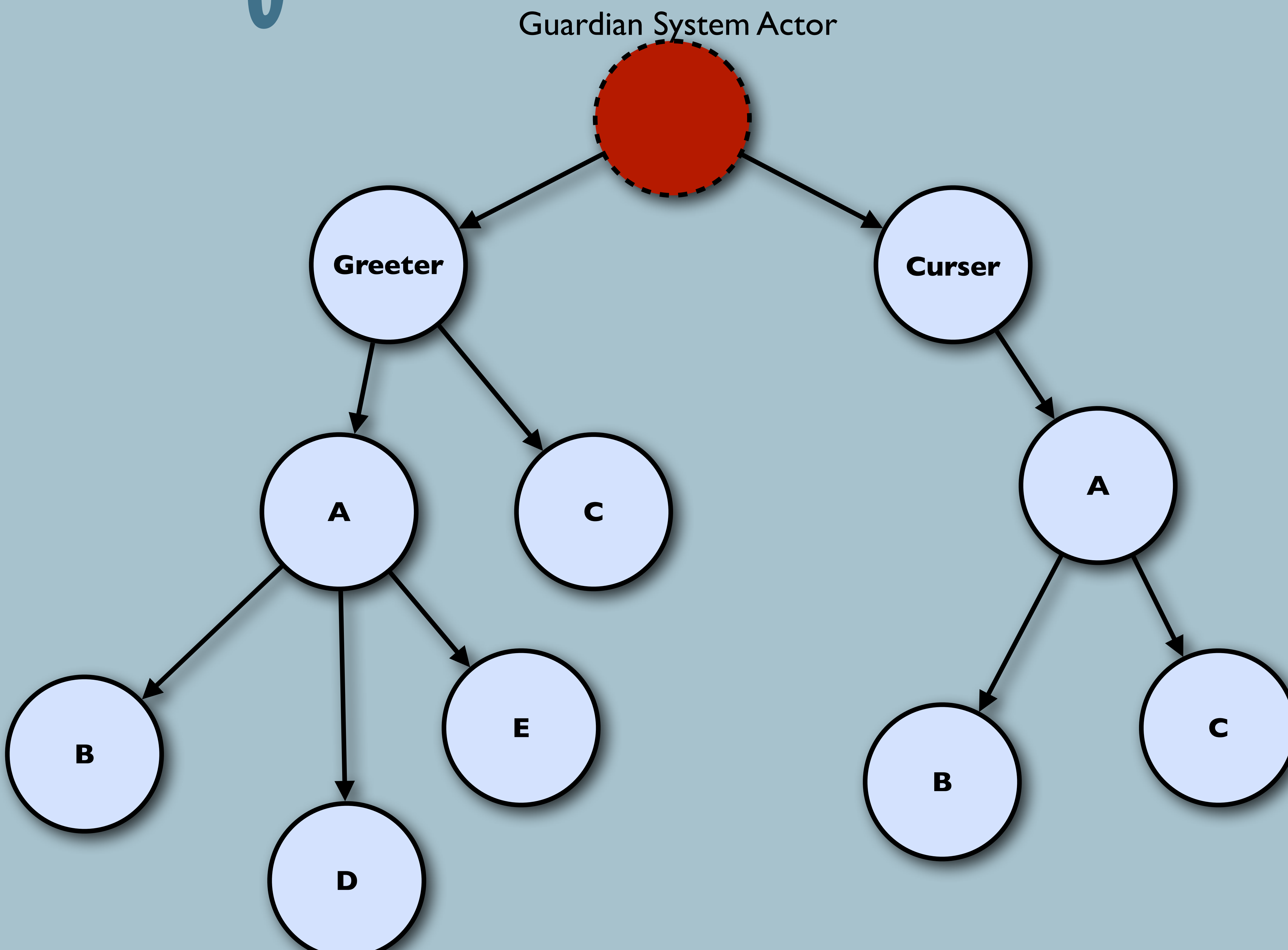


context.actorOf(Props[A], "A")

ACTORS *can form hierarchies...*

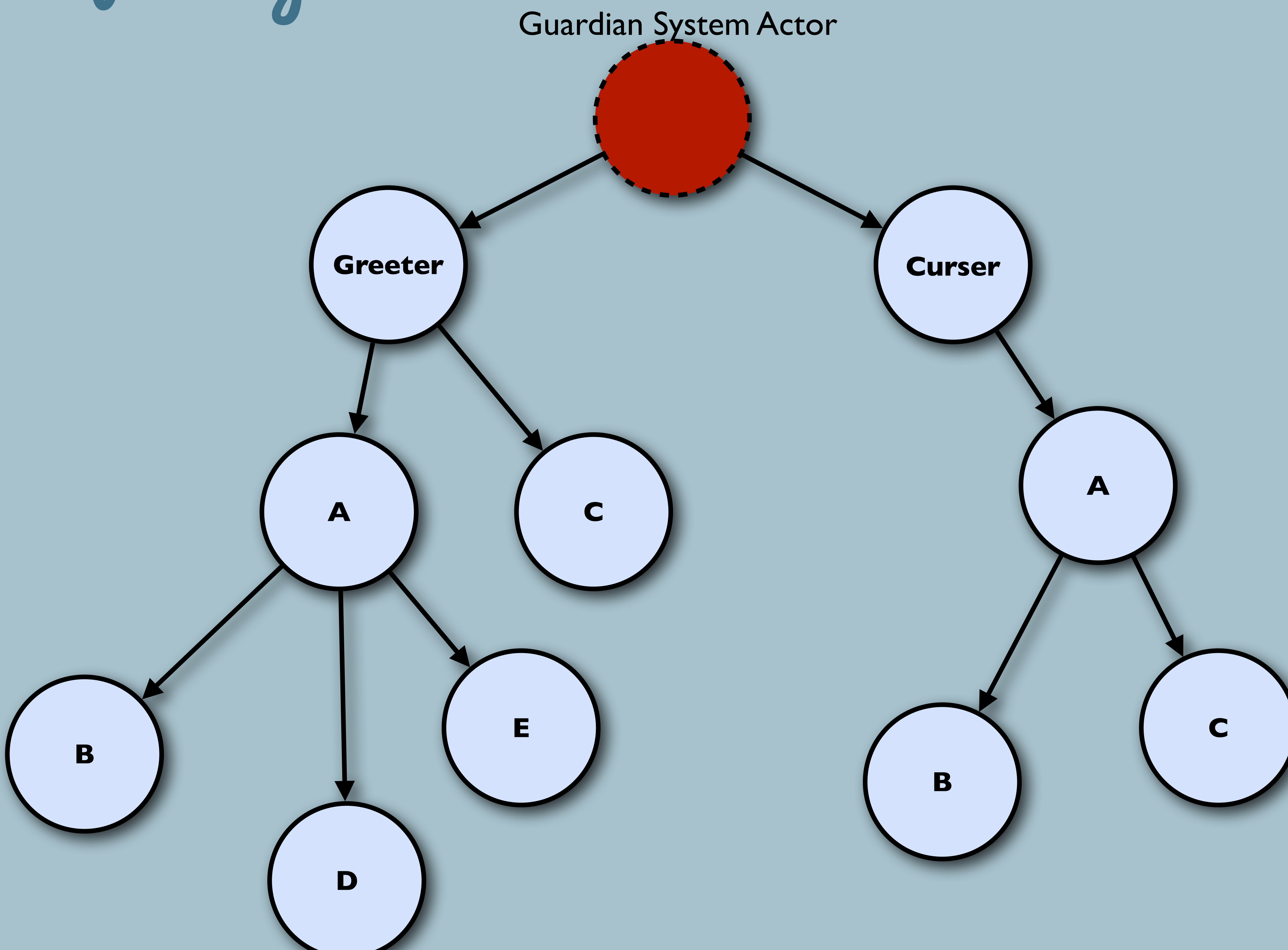


ACTORS *can form hierarchies...*



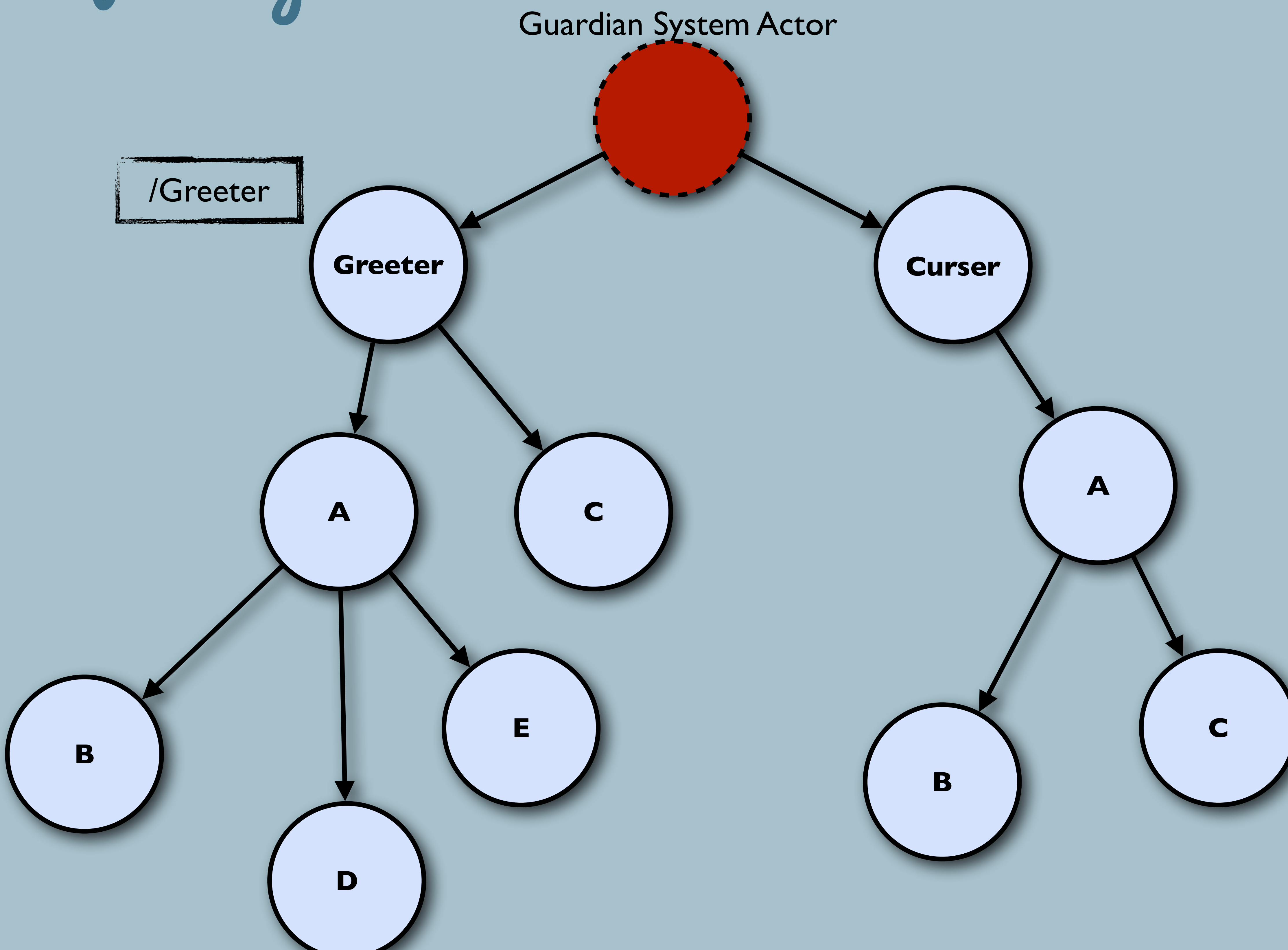
NAME RESOLUTION

like a file system...



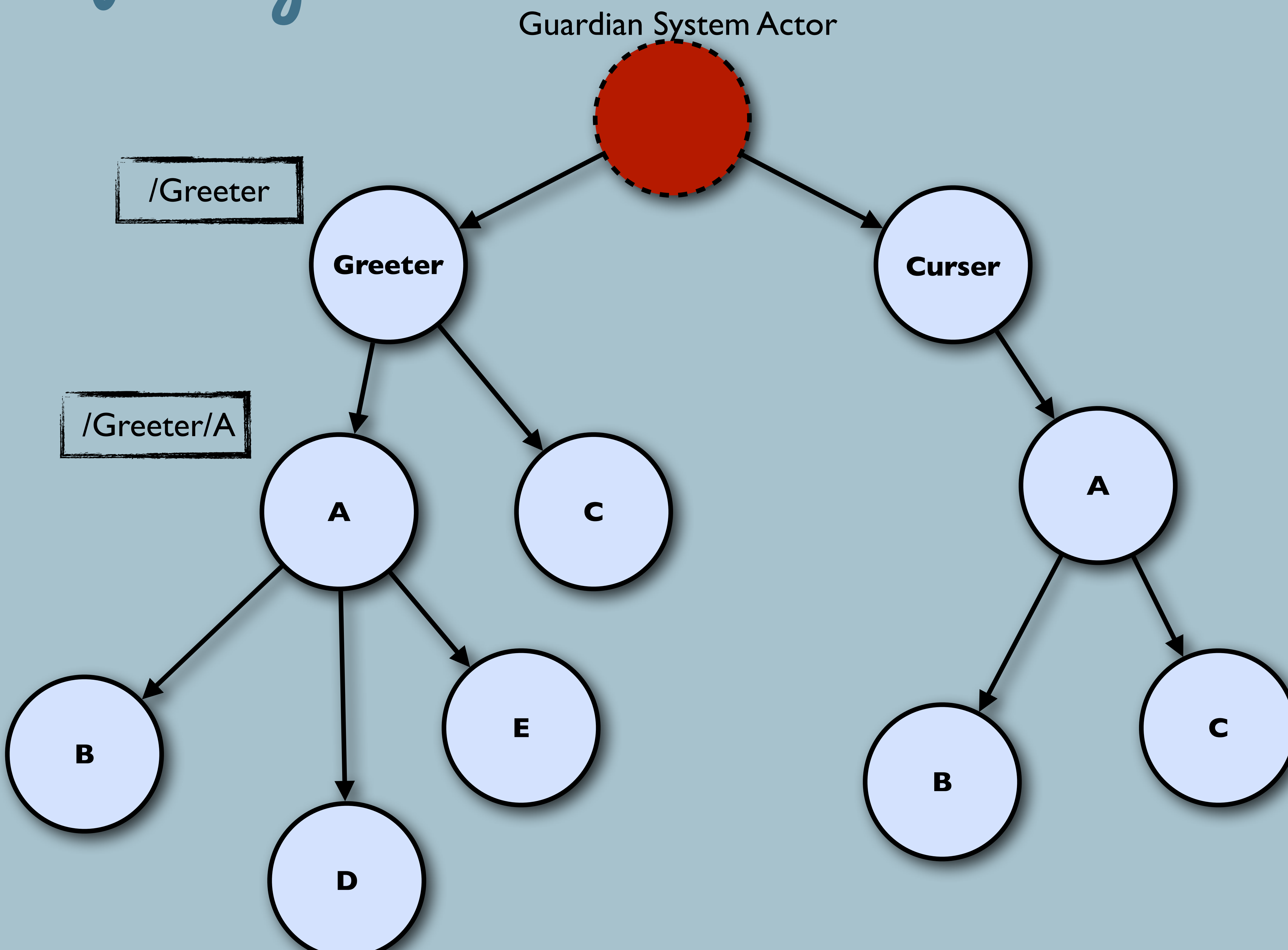
NAME RESOLUTION

like a file system...



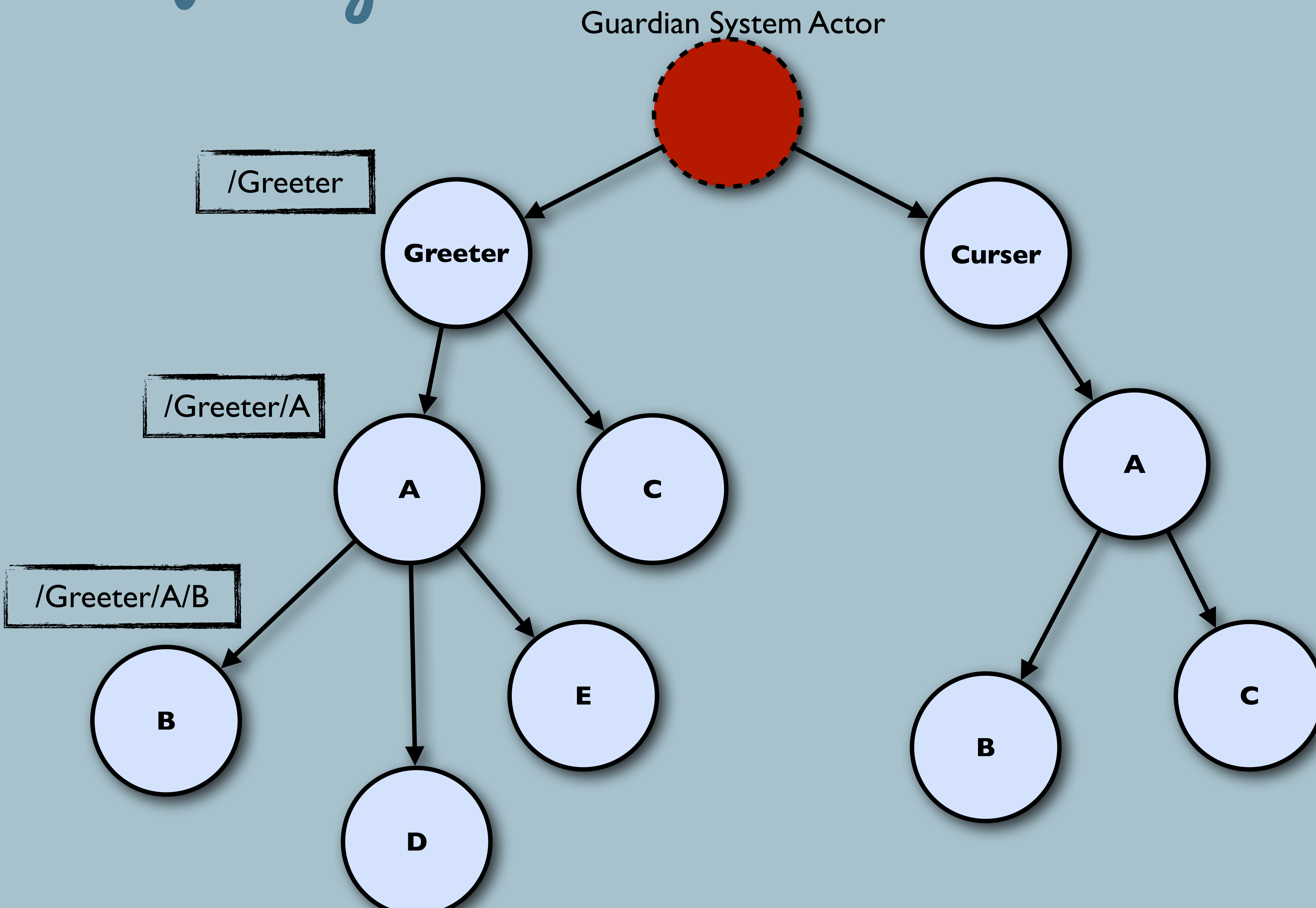
NAME RESOLUTION

like a file system...

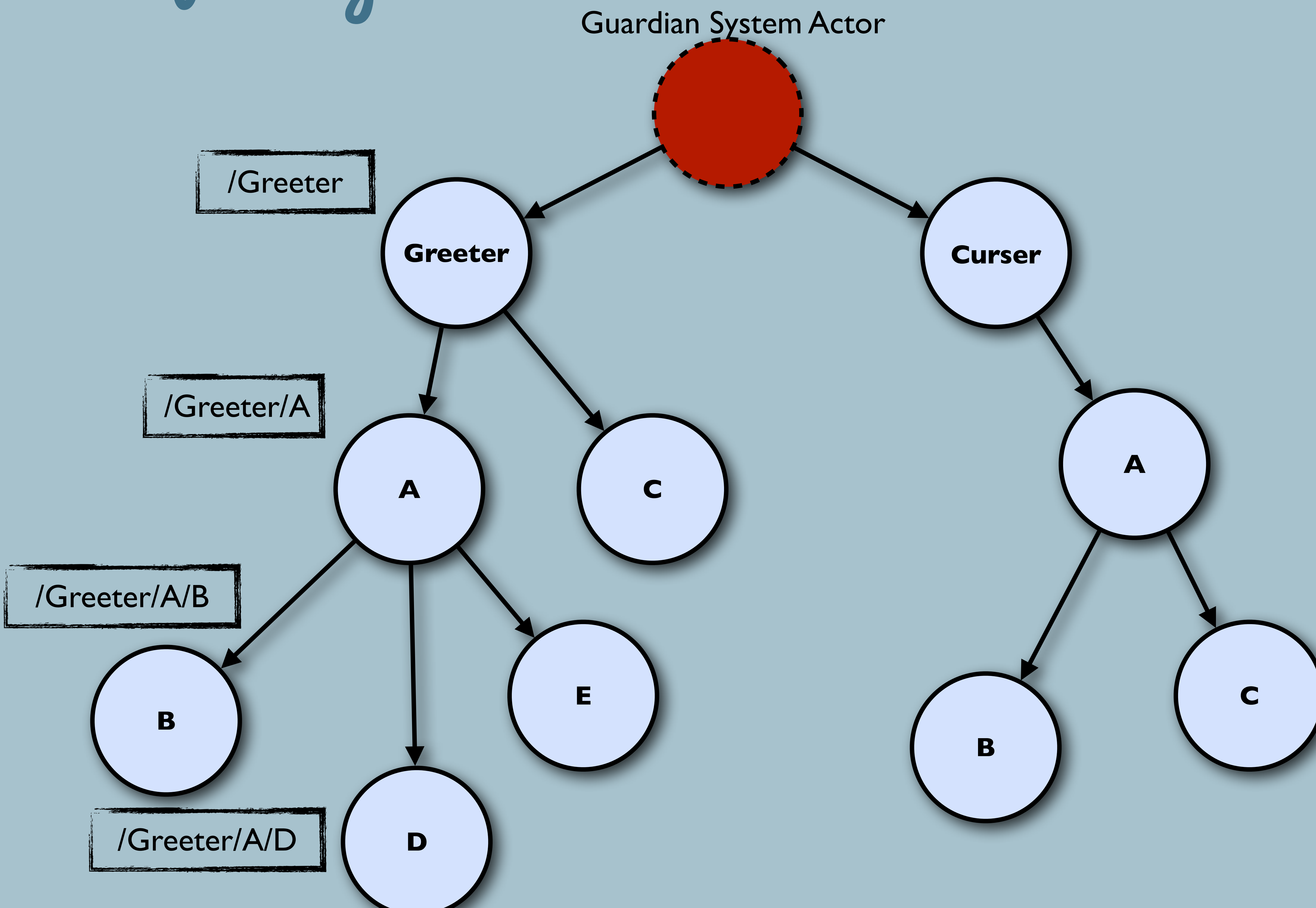


NAME RESOLUTION

like a file system...



NAME RESOLUTION
like a file system. . .



FIND ACTORS

```
val actorRef = system.actorFor("/user/Greeter/A")

val parent = context.actorFor("..")

val sibling = context.actorFor("../B")

val selection = system.actorSelection("/user/Greeter/*")
```

ERLANG-style Actors

In Scala Actors, Erlang-style `receive/react` is the default

Issues

- Implementation more expensive than Akka's global message handler
- Queue model can lead to message pile-up

But...

- Most real-world actor programs written in Erlang (probably)
- Erlang style can simplify complex messaging protocols

AKKA 2.0 INTRODUCES A `Stash` TRAIT FOR THIS

React vs. Global Event Loop

Erlang-style **react** *of Scala actors makes it easy to express certain messaging protocols through nested* **react***s:*

```
actor {
  react {
    case “open” =>
      var done = false
      loopWhile (!done) { react {
        case “read” => ...
        case “close” => done = true
      } }
  }
}
```

AKKA Become & Stash

*Using the "**stash**" to model the previous example using an Akka actor's global event loop:*

```
class ActorWithProtocol extends Actor with Stash {
  def receive = {
    case "open" =>
      unstashAll()
      context.become {
        case "read" => // do reading...
        case "close" => unstashAll(); context.unbecome()
        case msg => stash() }
    case msg => stash()
  }
}
```

Prepend all stashed messages to mailbox; leaves stash empty

Change actor's message handler

Restore previous message handler

Move message to stash to process later

CONCLUSION

SCALA IS A GROWABLE LANGUAGE

invaluable for establishing actors as one of its principle concurrency models

EMBRACING UNIQUE SCALA FEATURES

supports adoption in Scala community (but can provide Java API)

TIGHT INTEGRATION

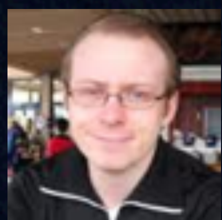
with execution environment ensures scalability and high performance

FIND OUT MORE

Akka: http://akka.io Futures in Scala 2.10: http://docs.scala-lang.org

The Typesafe Stack: http://www.typesafe.com/stack/

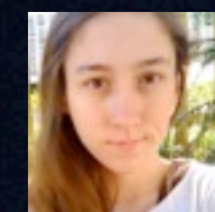
CREDITS



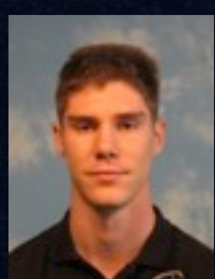
VIKTOR KLANG
TYPESAFE

JONAS BONÉR
TYPESAFE

PHILIPP HALLER
TYPESAFE

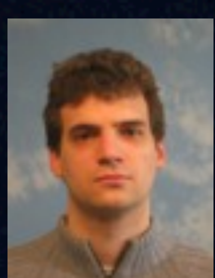


HEATHER MILLER
EPFL



ALEX PROKOPEC
EPFL

ROLAND KUHN
TYPESAFE



VOJIN JOVANOVIC
EPFL

DOUG LEA
SUNY

HAVOC PENNINGTON
TYPESAFE

QUESTIONS?