Effective Actors

Jamie Allen

Typesafe

Who Am I?

- Consultant at Typesafe
- Actor & Scala developer since 2009
- Author of Effective Akka from O'Reilly, coming at the end of August

jamie.allen@typesafe.com

Typesafe

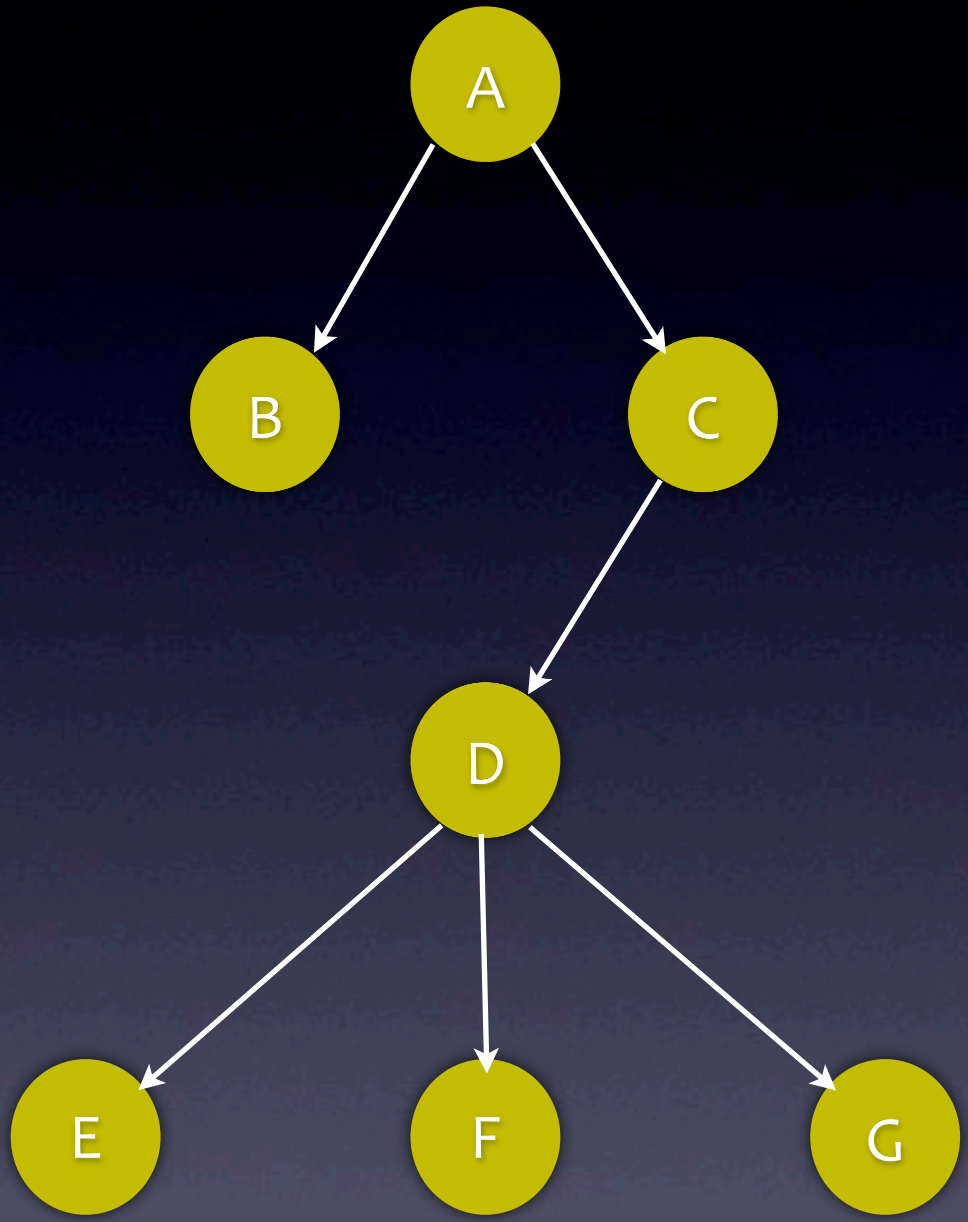
@jamie_allen

Effective Actors

- Best practices based on several years of actor development
- Helpful hints for reasoning about actors at runtime

Actors

- Concurrent, lightweight processes that communicate through asynchronous message passing
- Isolation of state, no internal concurrency



Akka Actors

```
class Pinger extends Actor {
  def receive = {
    case _ => println("Pinging!"); sender ! "Ping!"
  }
}

class Ponger extends Actor {
  def receive = {
    case _ => println("Ponging!"); sender ! "Pong!"
  }
}

object PingPong extends App {
  val system = ActorSystem()
  val pinger = system.actorOf(Props[Pinger])
  val ponger = system.actorOf(Props[Ponger])
  pinger.tell("Ping!", ponger)
  Thread.sleep(1000)
  system.shutdown
}
```

Akka Actors

```
class Pinger extends Actor {
  def receive = {
    case _ => println("Pinging!"); sender ! "Ping!"
  }
}

class Ponger extends Actor {
  def receive = {
    case _ => println("Ponging!"); sender ! "Pong!"
  }
}

object PingPong extends App {
  val system = ActorSystem()
  val pinger = system.actorOf(Props[Pinger])
  val ponger = system.actorOf(Props[Ponger])
  pinger.tell("Ping!", ponger)
  Thread.sleep(1000)
  system.shutdown
}
```

Akka Actors

```
class Pinger extends Actor {
  def receive = {
    case _ => println("Pinging!"); sender ! "Ping!"
  }
}

class Ponger extends Actor {
  def receive = {
    case _ => println("Ponging!"); sender ! "Pong!"
  }
}

object PingPong extends App {
  val system = ActorSystem()
  val pinger = system.actorOf(Props[Pinger])
  val ponger = system.actorOf(Props[Ponger])
  pinger.tell("Ping!", ponger)
  Thread.sleep(1000)
  system.shutdown
}
```

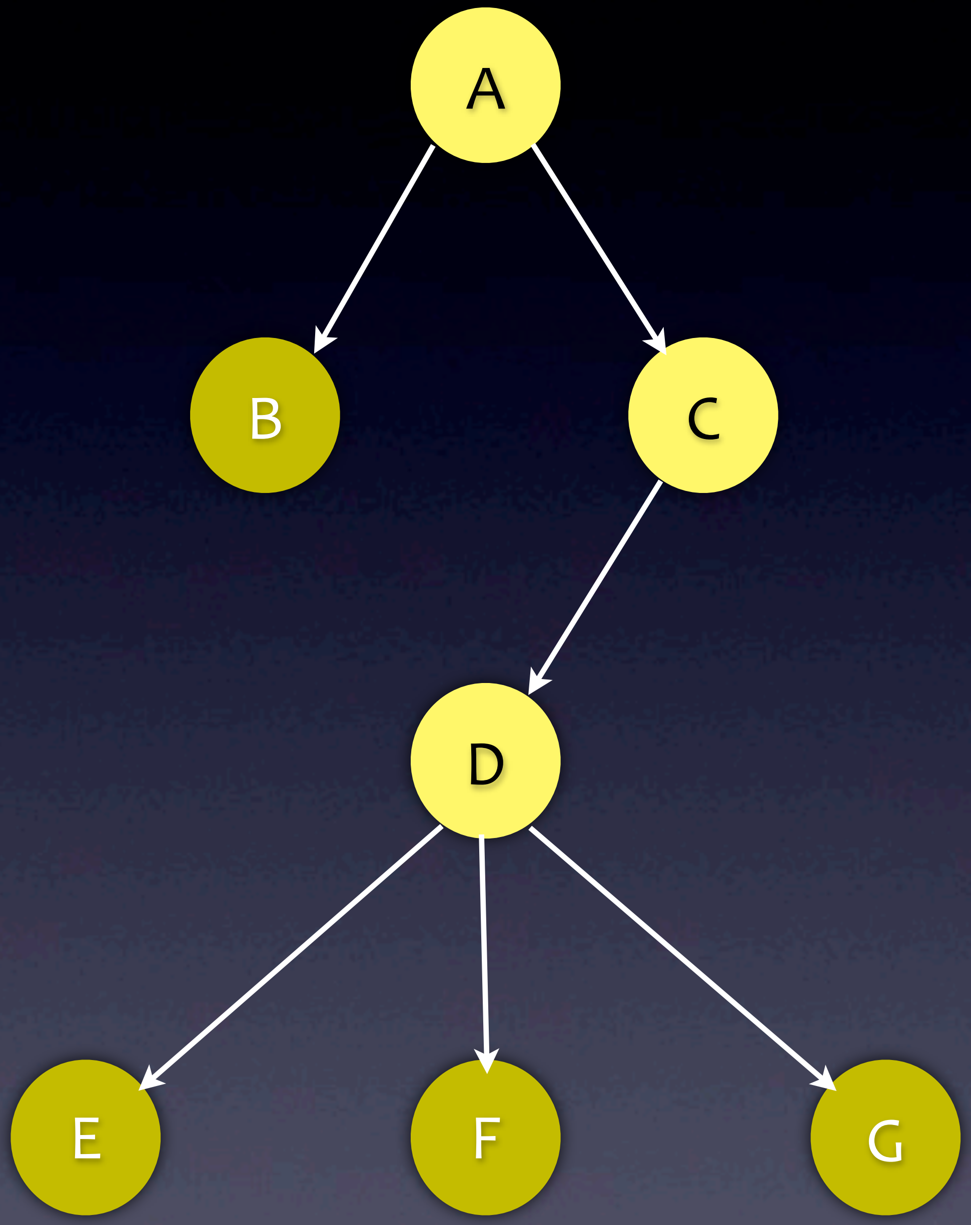
Akka Actors

```scala
class Pinger extends Actor {
  def receive = {
    case _ => println("Pinging!"); sender ! "Ping!"
  }
}

class Ponger extends Actor {
  def receive = {
    case _ => println("Ponging!"); sender ! "Pong!"
  }
}

object PingPong extends App {
  val system = ActorSystem()
  val pinger = system.actorOf(Props[Pinger])
  val ponger = system.actorOf(Props[Ponger])
  pinger.tell("Ping!", ponger)
  Thread.sleep(1000)
  system.shutdown
}
```

Overriding the “sender”

Supervisor Hierarchies

- Specifies handling mechanisms for groupings of actors in parent/child relationship



Typesafe

@jamie_allen

Akka Supervisors

```
class MySupervisor extends Actor {
  override val supervisorStrategy =
    OneForOneStrategy() {
      case ae: ArithmeticException => Resume
      case np: NullPointerException => Restart
    }

  context.actorOf(Props[MyActor])
}
```

Akka Supervisors

```
class MySupervisor extends Actor {
  override val supervisorStrategy =
    OneForOneStrategy() {
      case ae: ArithmeticException => Resume
      case np: NullPointerException => Restart
    }

  context.actorOf(Props[MyActor])
}
```

Akka Supervisors

```
class MySupervisor extends Actor {
  override val supervisorStrategy =
    OneForOneStrategy() {
      case ae: ArithmeticException => Resume
      case np: NullPointerException => Restart
    }

  context.actorOf(Props[MyActor])
}
```

Note the “context”

Domain Supervision

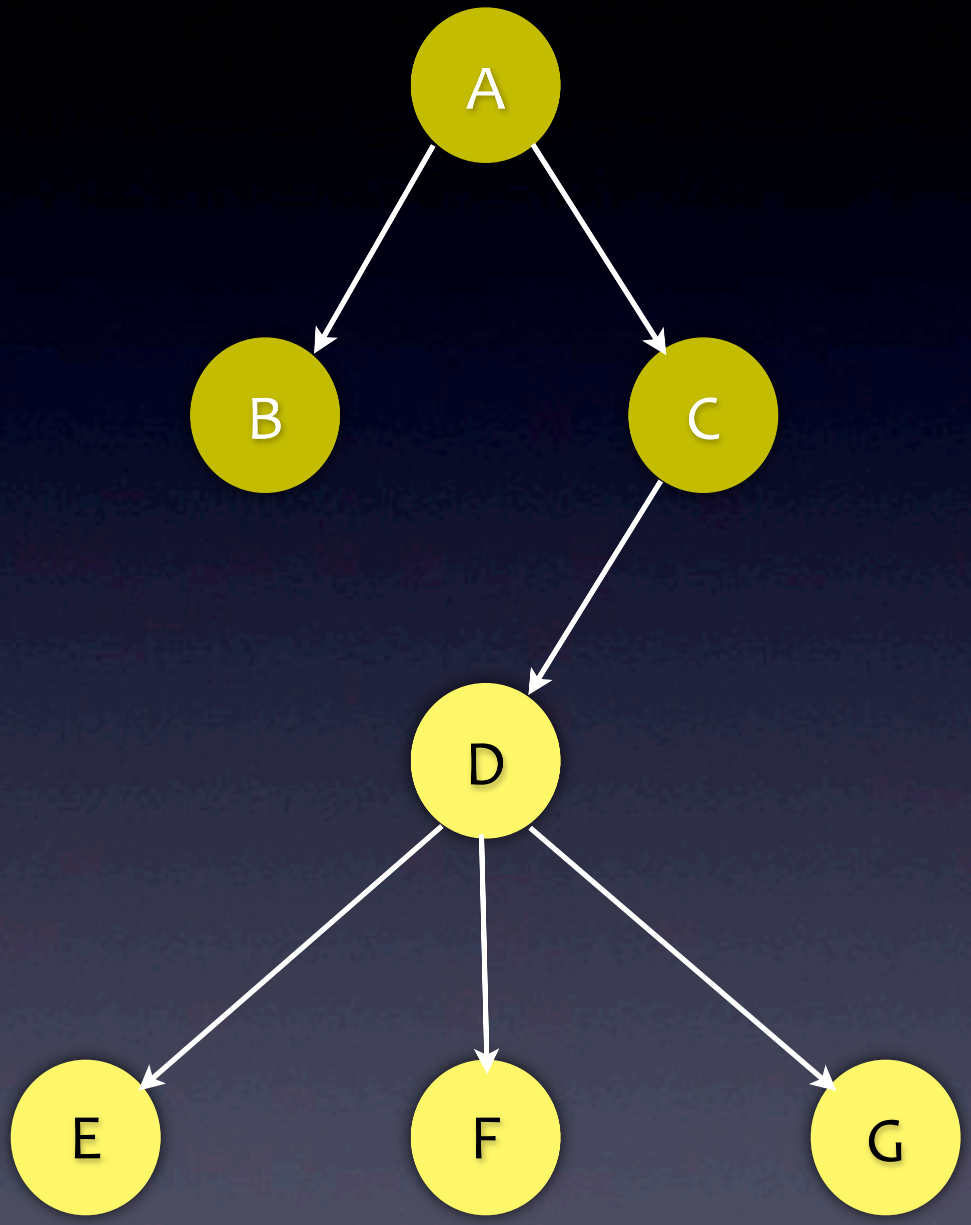
- Each supervisor manages a grouping of types in a domain
- Actors persist to represent existence of instances and contain their own state
- Actors constantly resolve the world as it should be against the world as it is

Worker Supervision

- Supervisors should hold all critical data
- Workers should receive data for tasks in messages
- Workers being supervised should perform dangerous tasks
- Supervisor should know how to handle failures in workers in order to retry appropriately

Parallelism

- Easily scale a task by creating multiple instances of an actor and applying work using various strategies
- Order is not guaranteed, nor should it be



Akka Routing

```
class MyActor extends Actor {
  def receive = { case x => println(x) }
}

object Parallelizer extends App {
  val system = ActorSystem()
  val router: ActorRef = system.actorOf(Props[MyActor].
    withRouter(RoundRobinRouter(nrOfInstances = 5)))

  for (i <- 1 to 10) router ! i
}
```

Akka Routing

```
class MyActor extends Actor {
  def receive = { case x => println(x) }
}

object Parallelizer extends App {
  val system = ActorSystem()
  val router: ActorRef = system.actorOf(Props[MyActor].
    withRouter(RoundRobinRouter(nrOfInstances = 5)))

  for (i <- 1 to 10) router ! i
}
```

Akka Routing

```
class MyActor extends Actor {
  def receive = { case x => println(x) }
}

object Parallelizer extends App {
  val system = ActorSystem()
  val router: ActorRef = system.actorOf(Props[MyActor].
    withRouter(RoundRobinRouter(nrOfInstances = 5)))

  for (i <- 1 to 10) router ! i
}
```

Should be configured externally

Akka Routing

```
class MyActor extends Actor {
  def receive = { case x => println(x) }
}

object Parallelizer extends App {
  val system = ActorSystem()
  val router: ActorRef = system.actorOf(Props[MyActor].
    withRouter(RoundRobinRouter(nrOfInstances = 5)))

  for (i <- 1 to 10) router ! i
}
```

RULE:
Actors Should Only Do One Thing

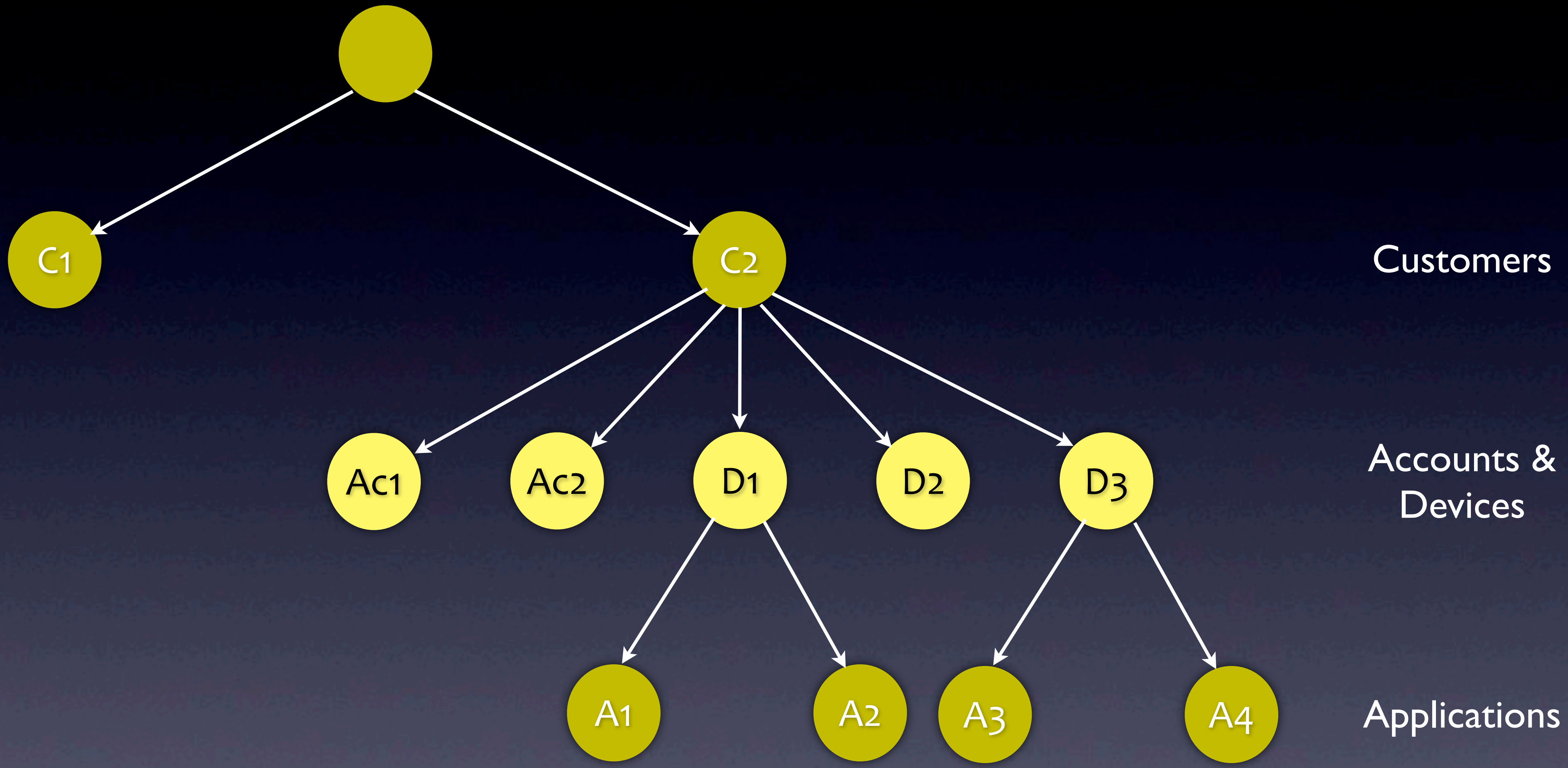
Single Responsibility Principle

- Do not conflate responsibilities in actors
- Becomes hard to define the boundaries of responsibility
- Supervision becomes more difficult as you handle more possibilities
- Debugging becomes very difficult

Supervision

- Every non-leaf node is technically a supervisor
- Create explicit supervisors under each node for each type of child to be managed

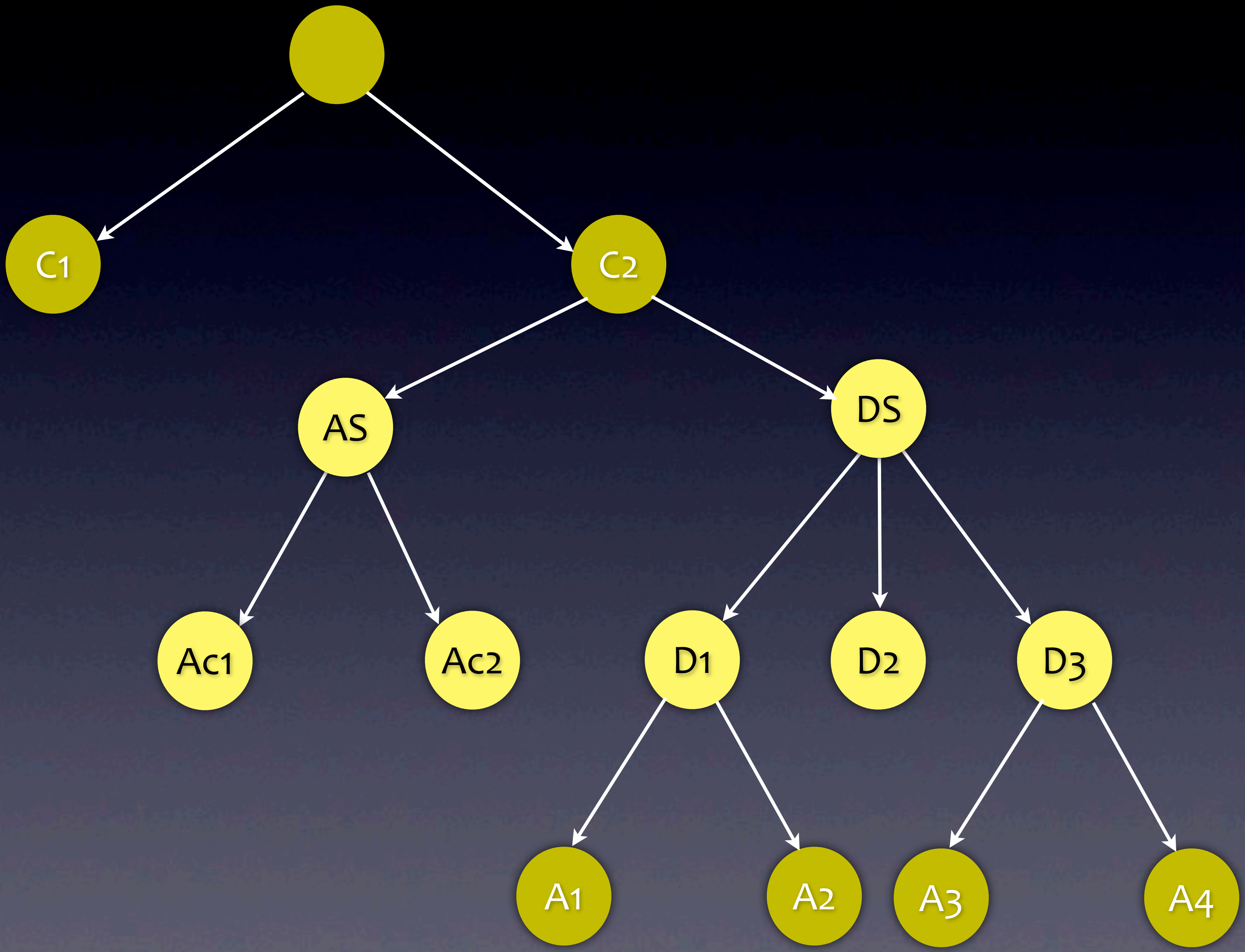
Conflated Supervision



Typesafe

@jamie_allen

Explicit Supervision



Customers

Accounts &
Devices

Applications

Typesafe

@jamie_allen

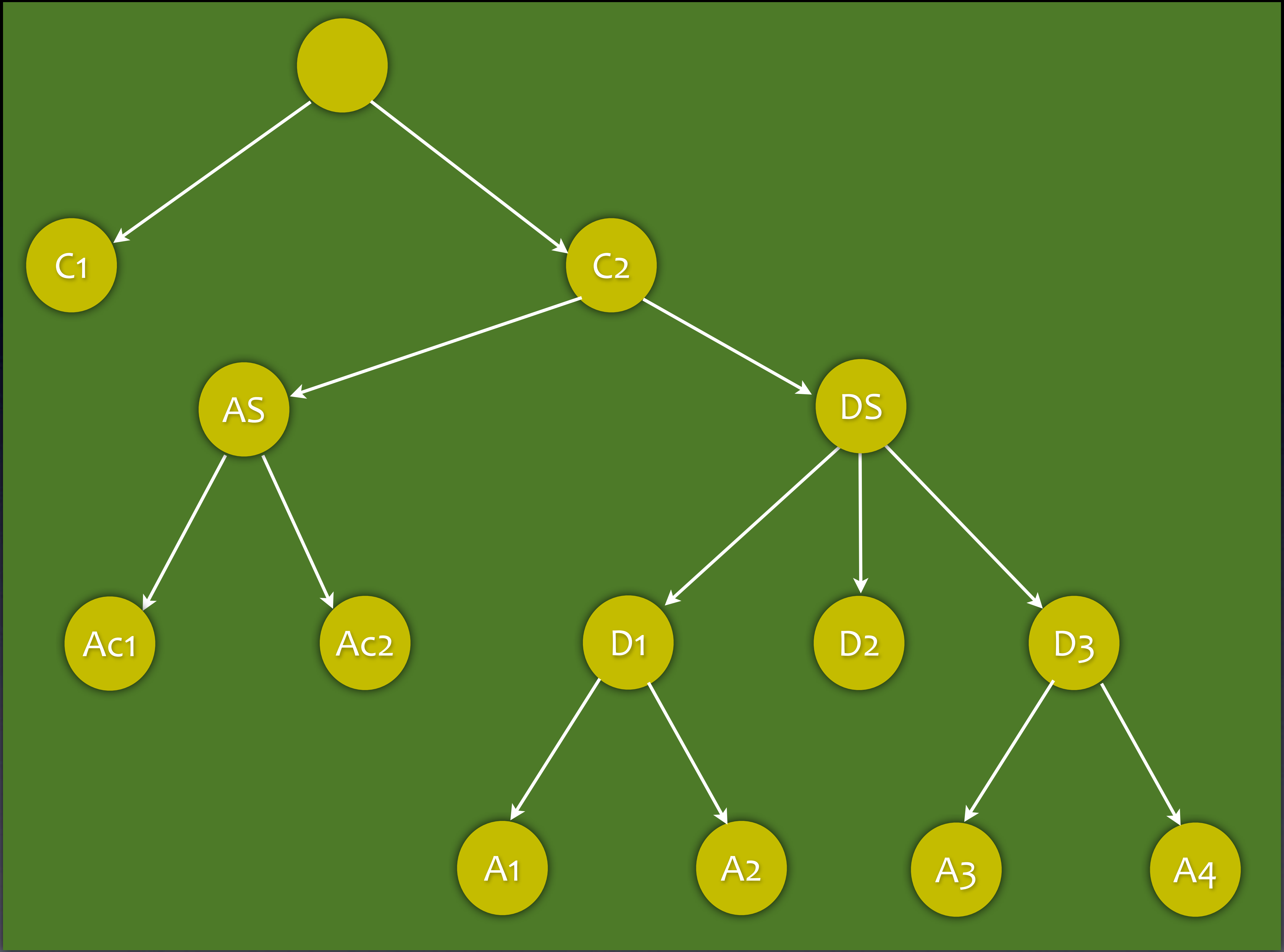
Keep the Error Kernel Simple

- Limit the number of supervisors you create at this level
- Helps with fault tolerance and explicit handling of errors through the hierarchy
- Akka uses synchronous messaging to create top-level actors

Use Failure Zones

- Multiple isolated zones with their own resources (thread pools, etc)
- Prevents starvation of actors
- Prevents issues in one branch from affecting another

Failure Zones

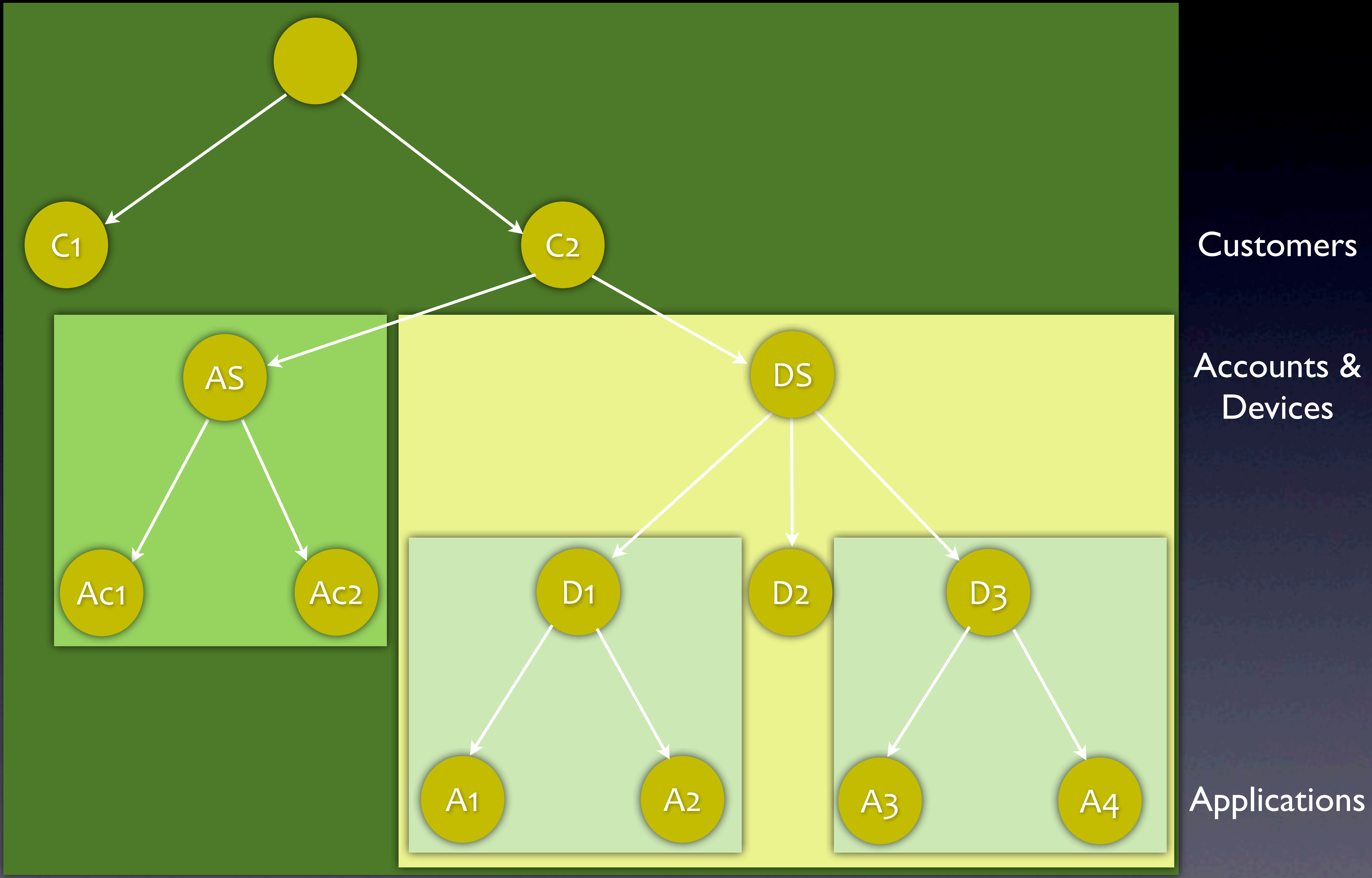


Customers

Accounts & Devices

Applications

Failure Zones



Typesafe

@jamie_allen

Takeaway

- Isolation of groups of actors limits the effects of failure
- For reasonably complex actor systems, shallow trees are a smell test
- Actors are cheap - use them

RULE:
Block Only When and Where You Must

Consequences of Blocking

- Eventually results in actor starvation as thread pool dries up
- Horrible performance
- Massive waste of system resources

Futures in Actors?

- Not the best solution
  - More heavyweight than fire and forget
- Better to use “tell” and introduce a transient child actor to handle the response

Transient Actor

```
class Worker extends Actor {
  def receive = {
    case s: Seq[Int] => sender ! s.reduce(_ + _)
  }
}

class Delegator extends Actor {
  val worker = context.actorOf(Props[Worker])
  def receive = {
    case _ =>
      context.actorOf(Props(new Actor() {
        case x =>
          println("Got value: %d".format(x))
          context.shutdown(self)
      }
  }
}
```

Transient Actor

```
class Worker extends Actor {
 def receive = {
  case s: Seq[Int] => sender ! s.reduce(_ + _)
 }
}

class Delegator extends Actor {
 val worker = context.actorOf(Props[Worker])
 def receive = {
  case _ =>
   context.actorOf(Props(new Actor() {
    case x =>
     println("Got value: %d".format(x))
     context.shutdown(self)
   }
 }
}
```

Remember to shut it down when done!

Blocking

- An example is database access
- Use a specialized actor with its own resources
- Pass messages to other actors to handle the result
- Akka provides “Managed Blocking” to limit the number of blocking operations taking place in actors at one time

Akka Dispatcher

```
class Worker extends Actor {
  def receive = {
    case s: Seq[Int] => sender ! s.reduce(_ + _)
  }
}

class Delegator extends Actor {
  implicit val timeout: Timeout = 2 seconds
  val worker = context.actorOf(Props[Worker]).
    withDispatcher(“my-dispatcher”)
  def receive = {
    case _ =>
      blocking {
        val futResult = worker ? (1 to 100)
        val result = Await.result(futResult, 2 seconds)
      }
  }
}
```

Failure
Zone

Push, Not Pull

- Start with no guarantees about delivery
- Add guarantees only where you need them
- Retry until you get the answer you expect
- Switch your actor to a "nominal" state at that point

Takeaway

- Find ways to ensure that your actors remain asynchronous and non-blocking
- Avoid making your actors wait for anything while handling a message

RULE:
Do Not Optimize Prematurely

Start Simple

- Make Tony Hoare happy
- Some things you know up front will help your performance
  - Algorithm
  - Data Structure
- Start with a simple configuration and profile
- Do not parallelize until you know you need to and where

Initial Focus

- Deterministic
- Declarative
- Immutable
- Start with functional programming and go from there

Advice From Jonas Bonér

- Layer in complexity
- Add indeterminism
- Add mutability in hot spots via CAS, non-locking data structures
- Use STM only if not high-contention
- Add explicit locking and threads as a last resort

Photo courtesy of Brian Clapper, NE Scala 2011

Prepare for Race Conditions

- Write actor code to be agnostic of time and order
- Actors should only care about now, not that something happened before it
- Actors can "become" or represent state machines to represent transitions

Beware the Thundering Herd

- Actor systems can be overwhelmed by "storms" of messages flying about
- Do not pass generic messages that apply to many actors
- Dampen actor messages if the exact same message is being handled repeatedly within a certain timeframe
- Tune your dispatchers and mailboxes via back-off policies and queue sizes
- Akka now has Circuit Breakers

Takeaway

- Start by thinking in terms of an implementation that is deterministic and not actor based
- Layer in complexity as you go

RULE:
Be Explicit In Your Intent

Props Factory in Companion Objects

- There is currently an issue with creating a new Akka actor instance inside of another one, where "this" is closed over
- To avoid this, define a Props factory method in an actor's own Companion Object
- SIP-21's implementation of Spores for defining what is explicitly closed over in an API will fix this

Props Factory

```
object Worker {
  def props = Props[Worker]
}

class Worker extends Actor {
  def receive = {
    case s: Seq[Int] => sender ! s.reduce(_ + _)
  }
}

class Delegator extends Actor {
  implicit val timeout: Timeout = 2 seconds
  val worker = context.actorOf(Worker.props).
    withDispatcher(“my-dispatcher”)
  def receive = {
    case _ =>
      context.actorOf(Props(new Actor() {
        case x =>
          println("Got value: %d".format(x))
          context.shutdown(self)
      }
    }
}
```

Props Factory

```
object Worker {
  def props = Props[Worker]
}

class Worker extends Actor {
  def receive = {
    case s: Seq[Int] => sender ! s.reduce(_ + _)
  }
}

class Delegator extends Actor {
  implicit val timeout: Timeout = 2 seconds
  val worker = context.actorOf(Worker.props).
    withDispatcher(“my-dispatcher”)
  def receive = {
    case _ =>
      context.actorOf(Props(new Actor() {
        case x =>
          println("Got value: %d".format(x))
          context.shutdown(self)
      }
  }
}
```

Props Factory?

Anonymous Actors

- They’re actor “literals”, just like a lambda
- Have similar limitations
  - Tough to debug due to “name mangling”
  - Not testable in isolation
  - Intent not as clear - must be read
- Cannot use a Props factory

Define Specific Actor Types

- You'll be happy you did in production
  - Better stack traces
  - More information about message flow
- Testable in isolation
- Can use a Companion Object Props factory

Name Your Actors

- Allows for external configuration
- Allows for lookup
- Better semantic logging

```
val system = ActorSystem("pingpong")
val pinger = system.actorOf(Props[Pinger], "pinger")
val ponger = system.actorOf(Props[Ponger], "ponger")
```

Create Specialized Messages

- Non-specific messages about general events are dangerous

```
AccountsUpdated
```

- Can result in "event storms" as all actors react to them
- Use specific messages forwarded to actors for handling

```
AccountDeviceAdded(acctNum, deviceNum)
```

Create Specialized Exceptions

- Don't use java.lang.Exception to represent failure in an actor
- Specific exceptions can be handled explicitly
- State can be transferred between actor incarnations in Akka (if need be)

Takeaway

- Be specific in everything you do
- Makes everything that occurs in your actor system more clear to other developers maintaining the code
- Makes everything more clear in production

RULE:
Do Not Expose Your Actors

No Direct References

- Actors die
- Doesn't prevent someone from calling into an actor with another thread
- Akka solves this with the ActorRef abstraction
- Erlang solves this with PIDs

Never Publish “this”

- Don't send it anywhere
- Don't register it anywhere
- Particularly with future callbacks
- Publish “self” instead, which is an ActorRef
- Avoid closing over "sender" in Akka, it will change with the next message

Use Immutable Messages

- Enforces which actor owns the data
- If mutable state can escape, what is the point of using an actor?

Pass Copies of Mutable Data

- Mutable data in actors is fine
- But data can escape your scope
- Copy the data and pass that, as Erlang does (COW)
- Akka has STM references

Avoid Sending Behavior

- Unless using Agents, of course
- Closures make this possible (and easy)
- Also makes it easy for state to escape

Takeaway

- Keep everything about an actor internal to that actor
- Be vary wary of data passed in closures to anyone else

RULE:
Make Debugging Easy On Yourself

Externalize Business Logic

- Consider using external functions to encapsulate complex business logic
- Easier to unit test outside of actor context
- Not a rule of thumb, but something to consider as complexity increases
- Not as big of an issue with Akka's TestKit

Use Semantically Useful Logging

- Trace-level logs should have output that you can read easily
- Use line-breaks and indentation
- Both Akka and Erlang support hooking in multiple listeners to the event log stream

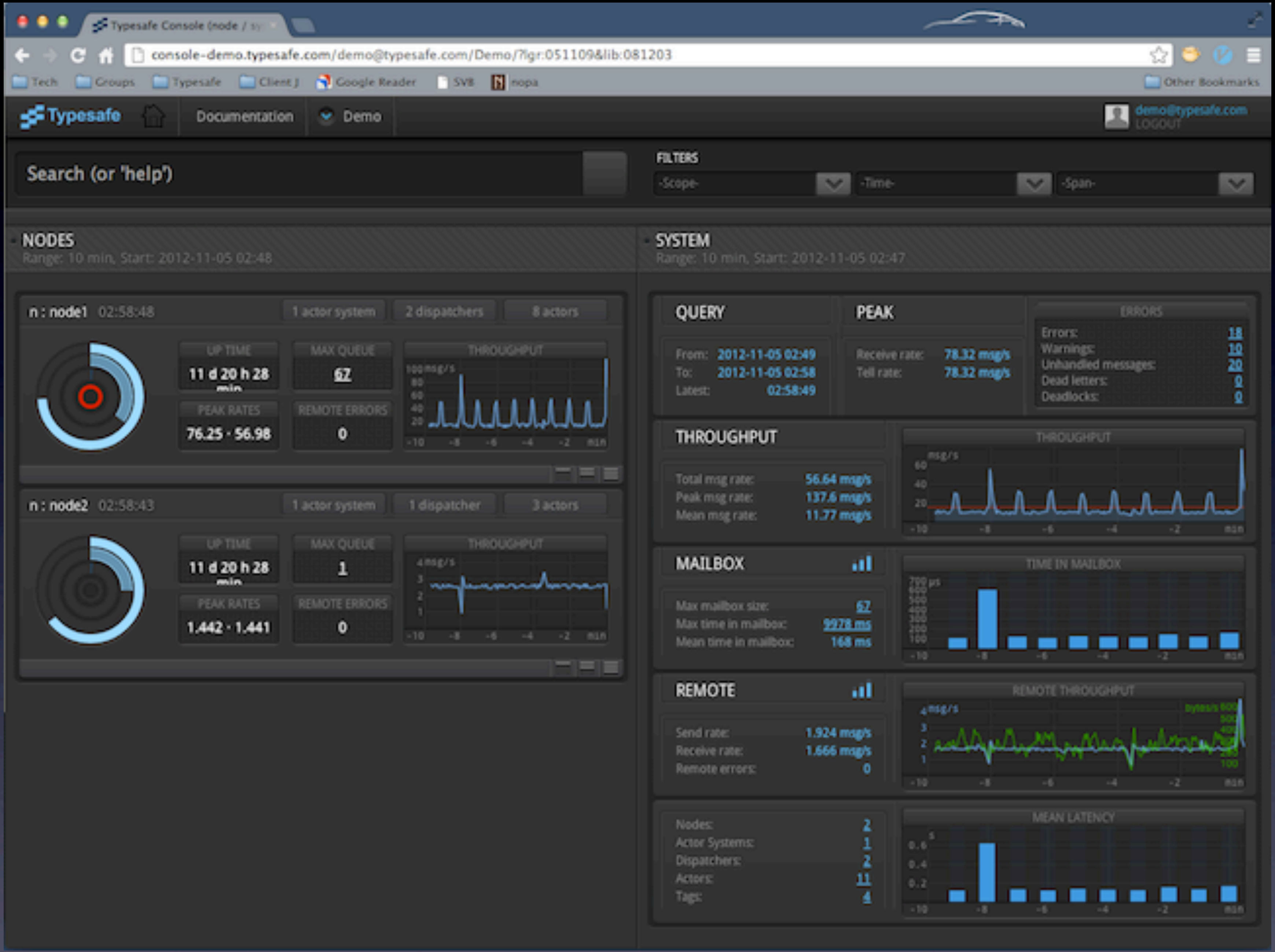
Unique IDs for Messages

- Allows you to track message flow
- When you find a problem, get the ID of the message that led to it
- Use the ID to grep your logs and display output just for that message flow
- Akka ensures ordering on a per actor basis, also in logging

Monitor Everything

- Do it from the start
- Use tools like JMX MBeans to visualize actor realization
- The Typesafe Console is a great tool to visualize actor systems, doesn't require you to do anything up front
- Visual representations of actor systems at runtime are invaluable

Typesafe Console



console-demo.typesafe.com/demo@typesafe.com/Demo/?lgr:051109&lib:081203

demo@typesafe.com

NODES
Range: 10 min, Start: 2012-11-05 02:48

n : node1 02:58:48 — 1 actor system — 2 dispatchers — 8 actors

UP TIME: 11 d 20 h 28 min
MAX QUEUE: 67
PEAK RATES: 76.25 · 56.98
REMOTE ERRORS: 0
THROUGHPUT

n : node2 02:58:43 — 1 actor system — 1 dispatcher — 3 actors

UP TIME: 11 d 20 h 28 min
MAX QUEUE: 1
PEAK RATES: 1.442 · 1.441
REMOTE ERRORS: 0
THROUGHPUT

SYSTEM
Range: 10 min, Start: 2012-11-05 02:47

QUERY
From: 2012-11-05 02:49
To: 2012-11-05 02:58
Latest: 02:58:49

PEAK
Receive rate: 78.32 msg/s
Tell rate: 78.32 msg/s

ERRORS
Errors: 18
Warnings: 10
Unhandled messages: 20
Dead letters: 0
Deadlocks: 0

THROUGHPUT
Total msg rate: 56.64 msg/s
Peak msg rate: 137.6 msg/s
Mean msg rate: 11.77 msg/s

MAILBOX
Max mailbox size: 67
Max time in mailbox: 9978 ms
Mean time in mailbox: 168 ms
TIME IN MAILBOX

REMOTE
Send rate: 1.924 msg/s
Receive rate: 1.666 msg/s
Remote errors: 0
REMOTE THROUGHPUT

Nodes: 2
Actor Systems: 1
Dispatchers: 2
Actors: 11
Tags: 4
MEAN LATENCY

To download: http://typesafe.com/platform/runtime/console

Takeaway

- Build your actor system to be maintainable from the outset
- Utilize all of the tools at your disposal

Thank You!

- Some content provided by members of the Typesafe team, including:
  - Jonas Bonér
  - Viktor Klang
  - Roland Kuhn
  - Havoc Pennington