Effective Actors

Jamie Allen

Who Am I?

- Consultant at Typesafe
- Actor & Scala developer since 2009

jamie.allen@typesafe.com

@jamie_allen

Typesafe 社でコンサルタントしています

Effective Actors

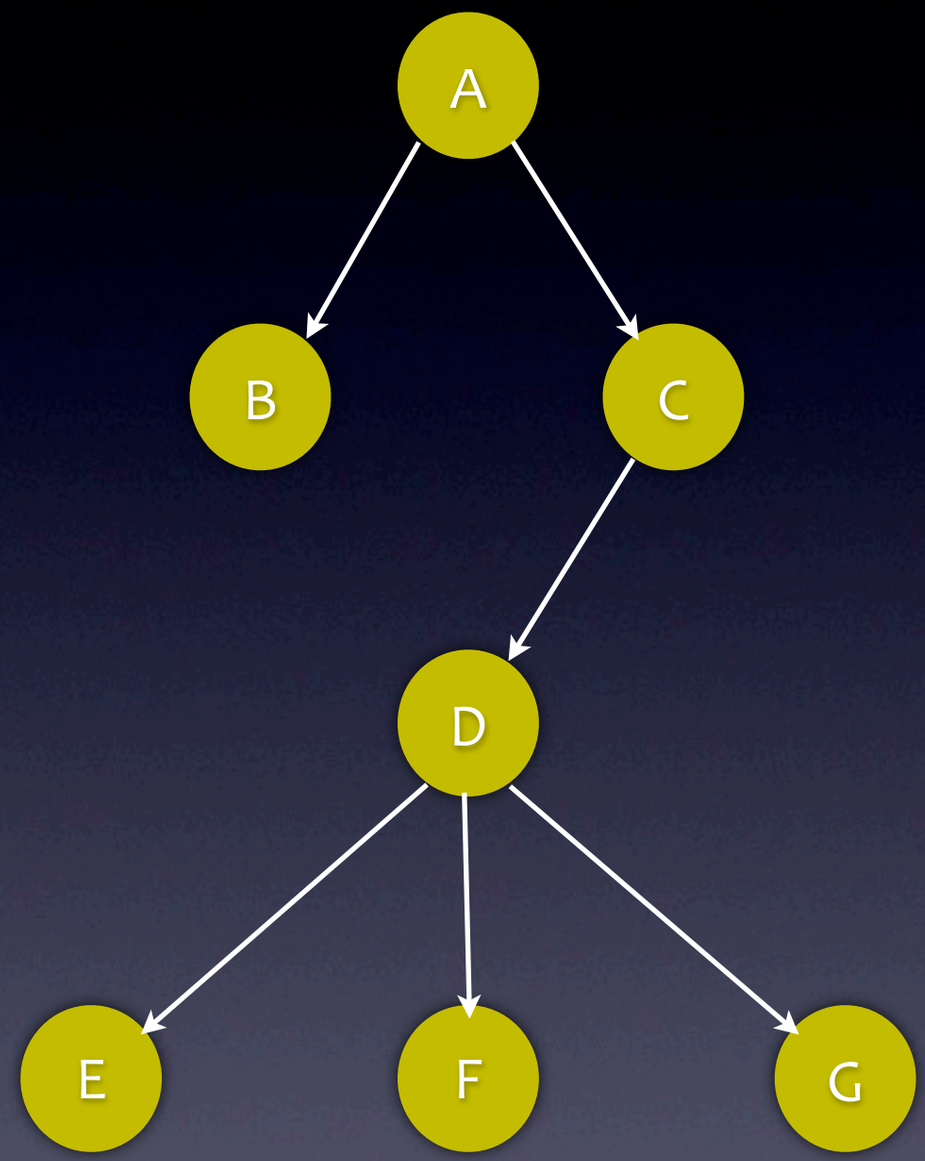
- Best practices based on several years of actor development
- Helpful hints for reasoning about actors at runtime

@jamie_allen

何年かの開発に基づいたベスト・プラクティスとヒント

Actors

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



非同期なメッセージを用いて対話する並行プロセス
状態の隔離

Akka Actors

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

class Ponger extends Actor {
  def receive = {
    case "Ping" => 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("Pong!", ponger)
  Thread.sleep(1000)
  system.shutdown
}
```

@jamie_allen

Typesafe

Akka Actors

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

class Ponger extends Actor {
  def receive = {
    case "Ping" => 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("Pong!", ponger)
  Thread.sleep(1000)
  system.shutdown
}
```

@jamie_allen

Typesafe

Akka Actors

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

class Ponger extends Actor {
  def receive = {
    case "Ping" => 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("Pong!", ponger)
  Thread.sleep(1000)
  system.shutdown
}
```

@jamie_allen

Typesafe

Akka Actors

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

class Ponger extends Actor {
  def receive = {
    case "Ping" => 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("Pong!", ponger)
  Thread.sleep(1000)
  system.shutdown
}
```

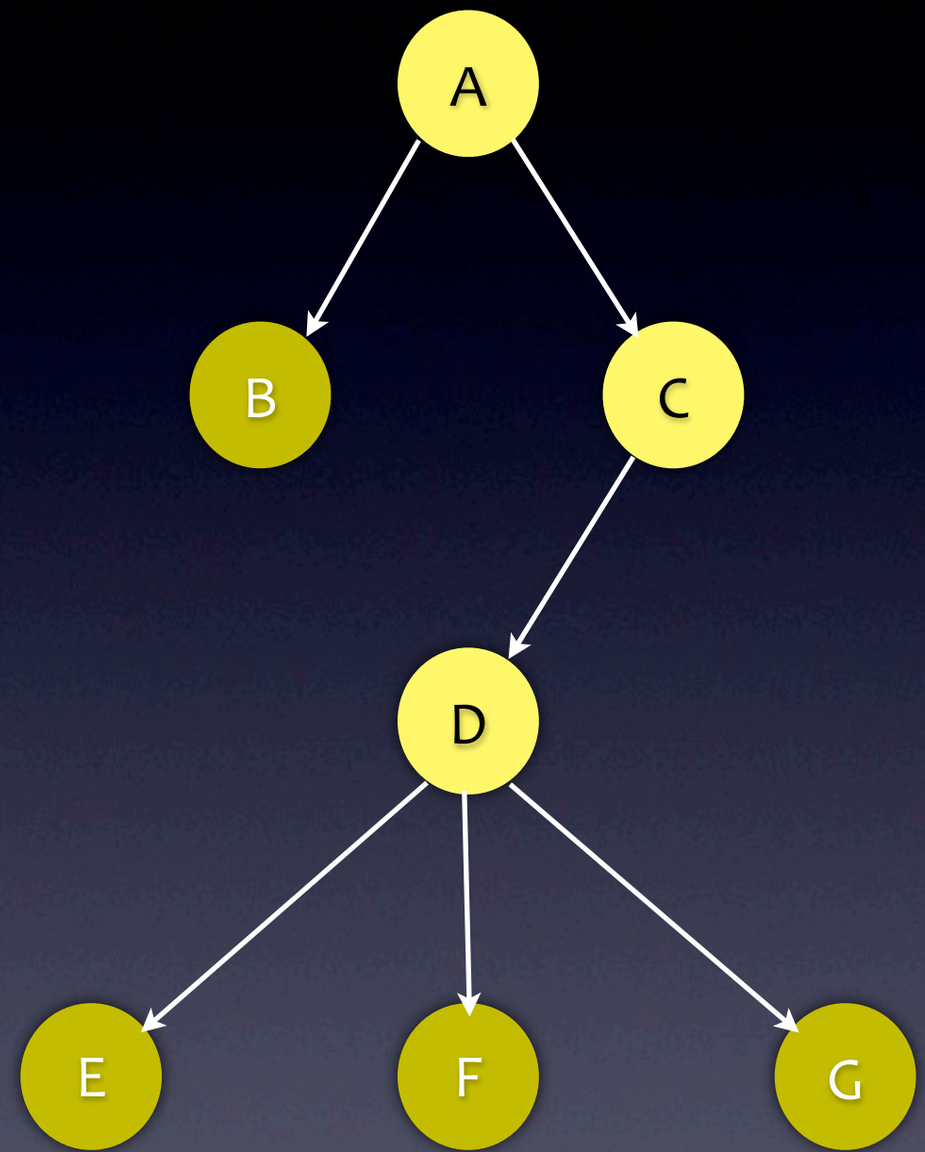
Overriding the “sender”

@jamie_allen

Typesafe

Supervisor Hierarchies

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



@jamie_allen

Supervisor はアクター階層の処理を定義する

Akka Supervisors

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

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

@jamie_allen

Typesafe

Akka Supervisors

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

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

@jamie_allen

Typesafe

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”

@jamie_allen

Typesafe

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

@jamie_allen

Supervisor 階層を利用しドメインをモデルする

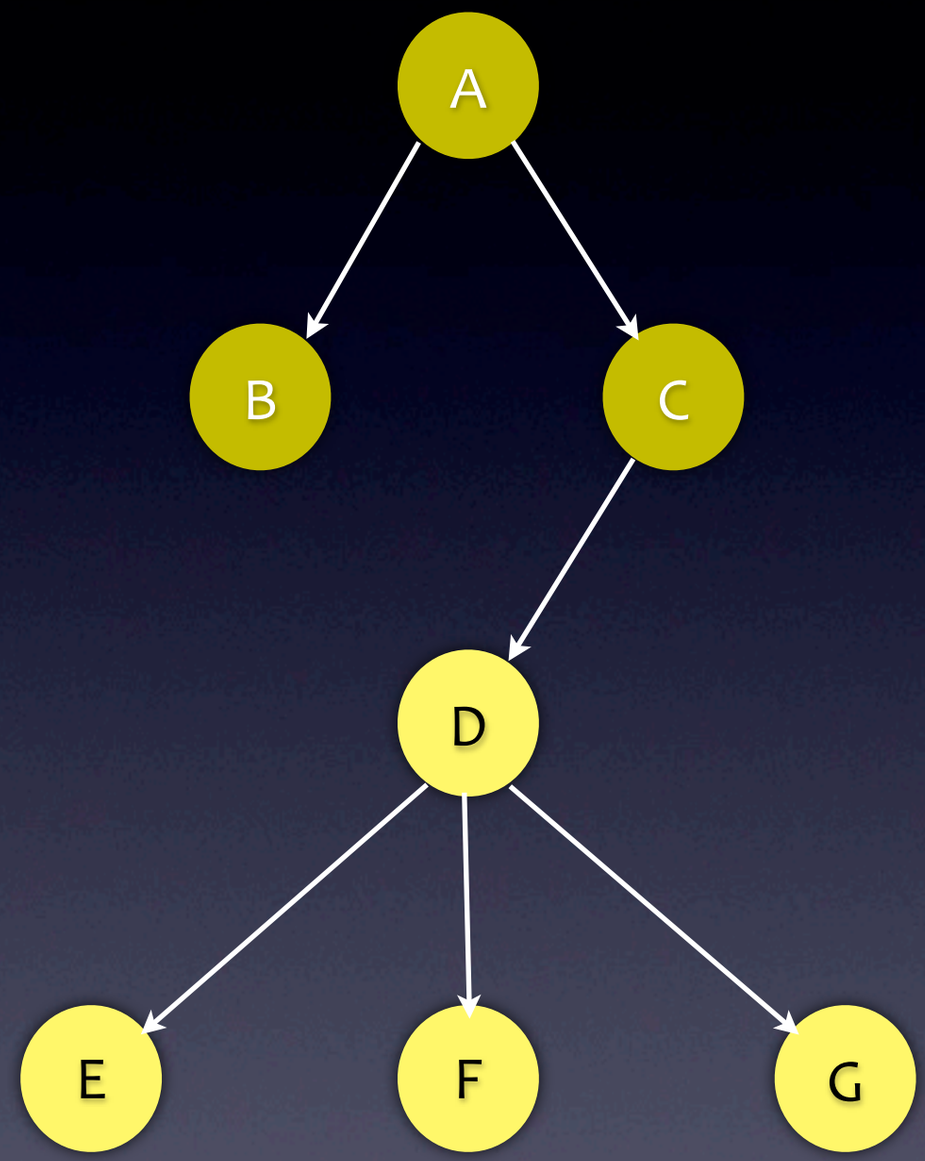
Worker Supervision

- 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

Supervisor がデータを保持し、メッセージでワーカーに渡す
ワーカーは死んでも構わない

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



@jamie_allen

複数のアクターを作成することでタスクをスケール

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
}
```

@jamie_allen

Typesafe

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
}
```

@jamie_allen

Typesafe

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

@jamie_allen

Typesafe

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
}
```

@jamie_allen

Typesafe

RULE:

Actors Should Only Do One Thing

@jamie_allen

アクターは1つの事に特化すること

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

@jamie_allen

単一責任の原則: なんでもかんでも手を出さない

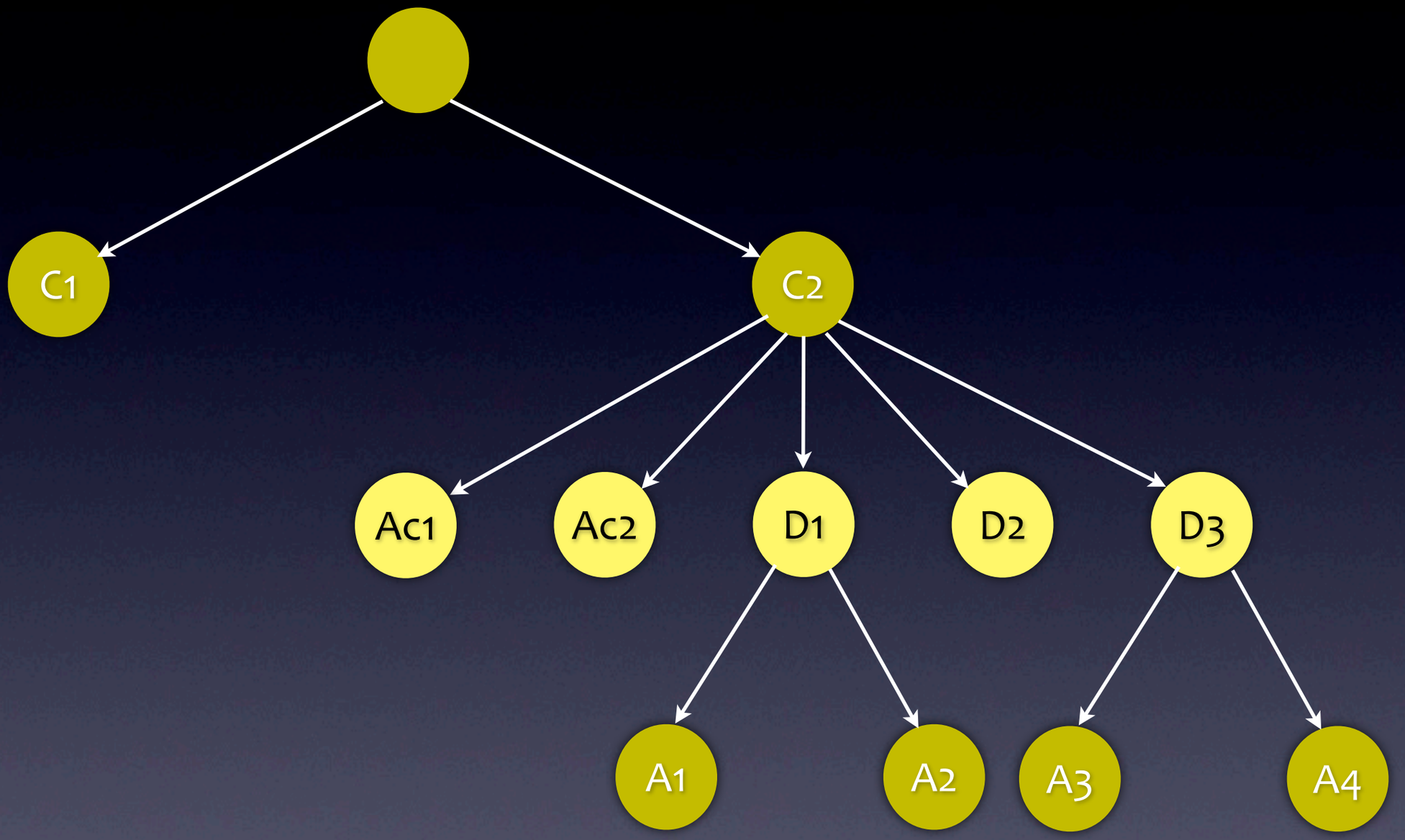
Supervision

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

@jamie_allen

アクターの種類ごとに Supervisor を分けること

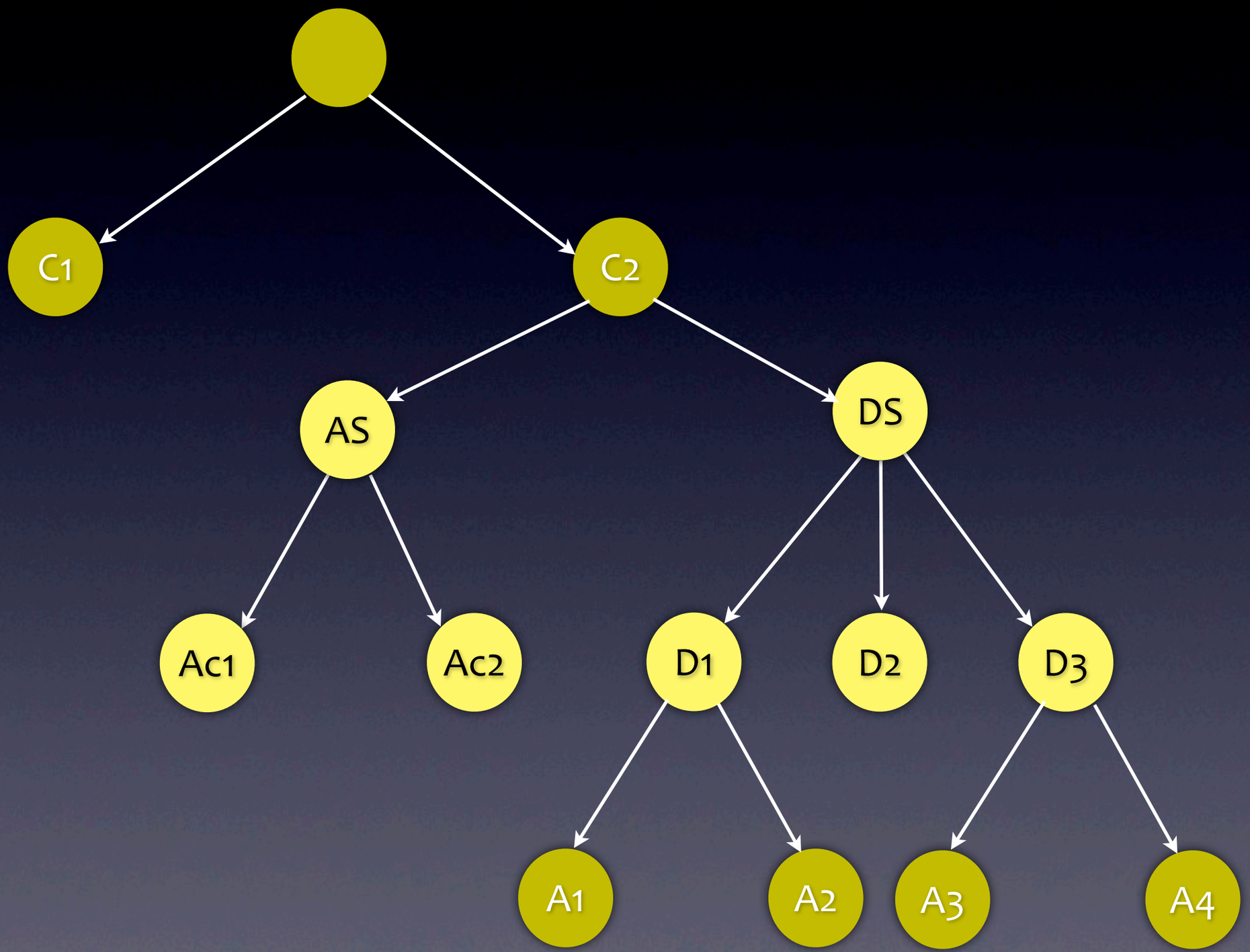
Conflated Supervision



@jamie_allen

混ざってる Supervisor

Explicit Supervision



Customers

Accounts & Devices

Applications

@jamie_allen

明示的な Supervisor

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

@jamie_allen

Error Kernel はシンプルに

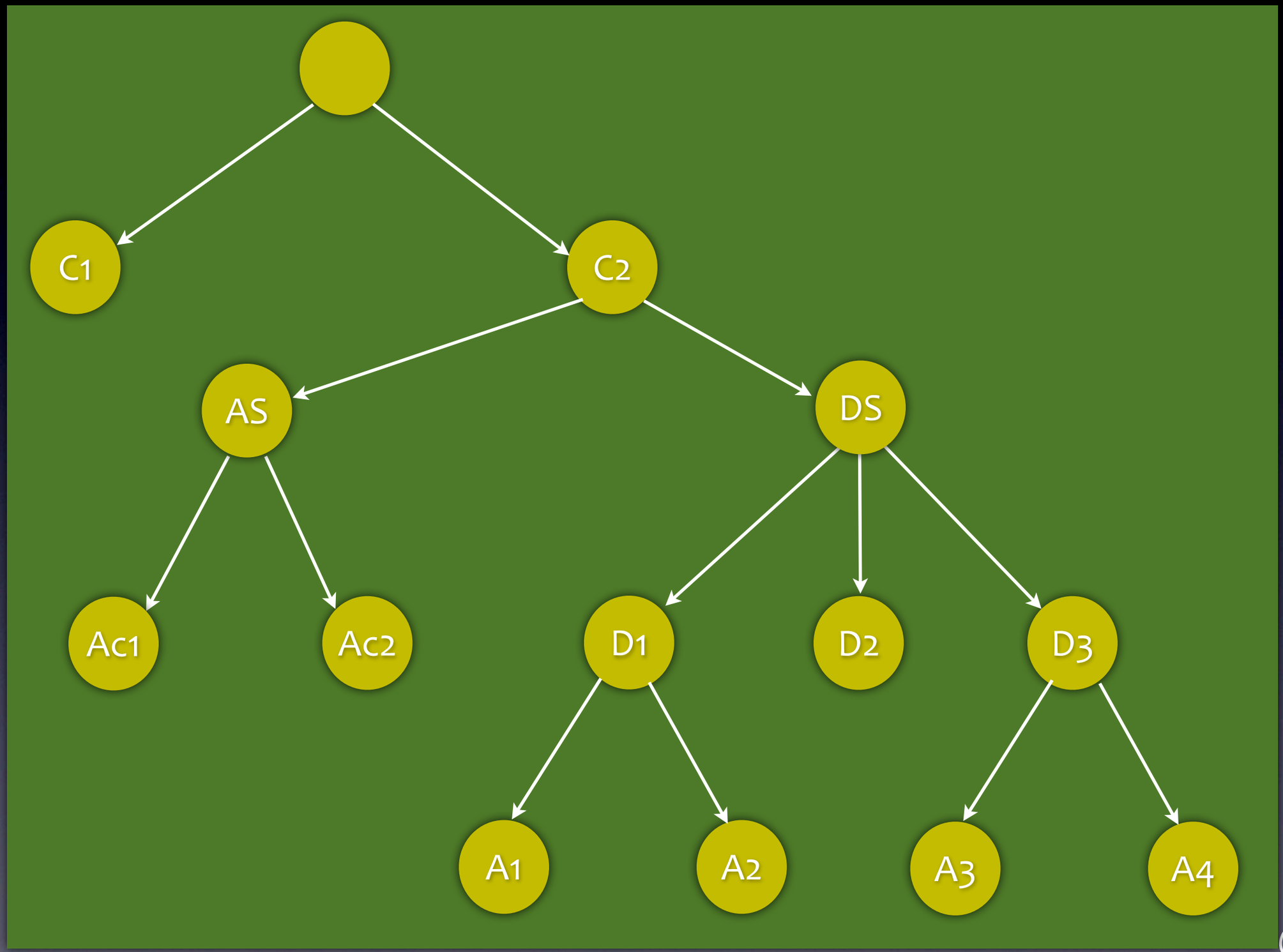
Use Failure Zones

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

@jamie_allen

Failure Zone を使ってリソースを隔離する

Failure Zones



Customers

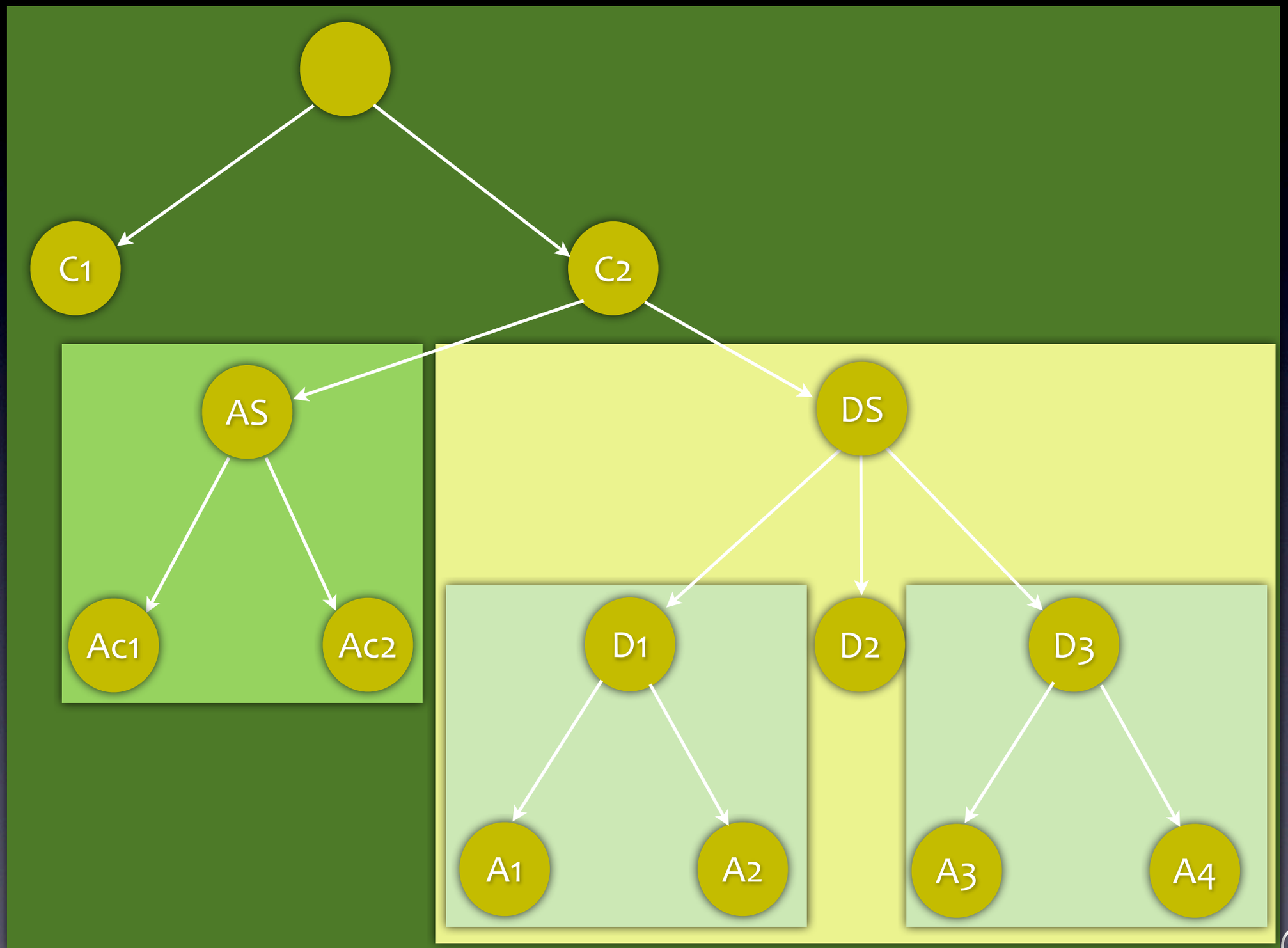
Accounts & Devices

Applications

@jamie_allen

Typesafe

Failure Zones



Customers

Accounts & Devices

Applications

@jamie_allen

Typesafe

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

@jamie_allen

アクターのグループを隔離することで障害の影響を抑える

RULE:

Block Only When and Where You Must

@jamie_allen

必要な時に、必要な場所のみでブロックすること

Consequences of Blocking

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

ブロッキングの行く末: 無くなるスレッド、飢えるアクター、落ちる性能

Futures

```
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])
  def receive = {
    case "Start" =>
      worker ? (1 to 100) map
        (x => println("Got value: %d".format(x)))
  }
}

object Bootstrapper extends App {
  val system = ActorSystem()
  val delegator = system.actorOf(Props[Delegator])
  delegator ! "Start"
  Thread.sleep(1000)
  system.shutdown
}
```

@jamie_allen

Typesafe

Futures

```
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])
  def receive = {
    case "Start" =>
      worker ? (1 to 100) map
        (x => println("Got value: %d".format(x)))
  }
}

object Bootstrapper extends App {
  val system = ActorSystem()
  val delegator = system.actorOf(Props[Delegator])
  delegator ! "Start"
  Thread.sleep(1000)
  system.shutdown
}
```

@jamie_allen

Typesafe

Futures

```
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])
  def receive = {
    case "Start" =>
      worker ? (1 to 100) map
        (x => println("Got value: %d".format(x)))
  }
}

object Bootstrapper extends App {
  val system = ActorSystem()
  val delegator = system.actorOf(Props[Delegator])
  delegator ! "Start"
  Thread.sleep(1000)
  system.shutdown
}
```

@jamie_allen

Typesafe

Futures

```
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])
  def receive = {
    case "Start" =>
      worker ? (1 to 100) map
        (x => println("Got value: %d".format(x)))
  }
}

object Bootstrapper extends App {
  val system = ActorSystem()
  val delegator = system.actorOf(Props[Delegator])
  delegator ! "Start"
  Thread.sleep(1000)
  system.shutdown
}
```

@jamie_allen

Typesafe

Futures

```
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])
  def receive = {
    case "Start" =>
      worker ? (1 to 100) map
        (x => println("Got value: %d".format(x)))
  }
}

object Bootstrapper extends App {
  val system = ActorSystem()
  val delegator = system.actorOf(Props[Delegator])
  delegator ! "Start"
  Thread.sleep(1000)
  system.shutdown
}
```

@jamie_allen

Typesafe

Sequential Futures

```
val r: Future[Int] = for {
  a <- (service1 ? GetResult).mapTo[Int]
  b <- (service2 ? GetResult(a)).mapTo[Int]
} yield a * b
```

@jamie_allen

順次 Future

Sequential Futures

Type Safety

```
val r: Future[Int] = for {
  a <- (service1 ? GetResult).mapTo[Int]
  b <- (service2 ? GetResult(a)).mapTo[Int]
} yield a * b
```

@jamie_allen

順次 Future

Parallel Futures

First way: Define futures first

```
val f1: Future[Int] = service1 ? GetResult
val f2: Future[Int] = service2 ? GetResult

val r: Future[Int] = for {
  a <- f1
  b <- f2
} yield a * b
```

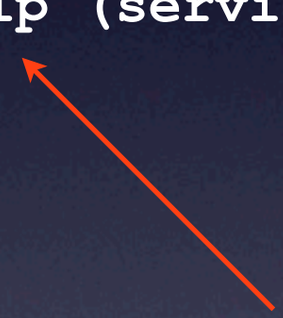
@jamie_allen

並列 Future 方法1: 事前に Future を定義

Parallel Futures

Second way: Define in one line of for comprehension

```
val r: Future[Int] = for {
  (a: Int, b: Int) <- (service1 ? GetResult) zip (service2 ? GetResult)
} yield a * b
```



Note the “zip”

@jamie_allen

並列 Future 方法2: zip を使って 1行で定義

Blocking

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

@jamie_allen

ブロッキング専門のアクターを作る

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

@jamie_allen

Typesafe

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

@jamie_allen

配達時間はまず保証無しから始める

Takeaway

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

@jamie_allen

アクターが非同期で非ブロッキングであるように努めること

RULE:

Do Not Optimize Prematurely

@jamie_allen

早すぎる最適化はダメ

Start Simple

- Make Donald Knuth happy
- Start with a simple configuration and profile

- Do not parallelize until you know you need to and where

@jamie_allen

プロファイルも取らずに並列化しないこと

Initial Focus

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

@jamie_allen

まずは決定的、宣言型、不変など関数型の基本から

Advice From Jonas Bonér

- Layer in complexity
- Add indeterminism
- Add mutability in hot spots (CAS and STM)
- Add explicit locking and threads as a last resort

Photo courtesy of Brian Clapper, NE Scala 2011

段階的に複雑さを導入する: 1. 非決定性 2. 可変性
3. スレッドと明示的なロックは最終手段

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

@jamie_allen

アクターは時間と順序に無関心であるべき

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

メッセージの「嵐」に注意
複数のアクターに作用する汎用メッセージは避ける

Takeaway

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

@jamie_allen

まずはアクターを使わない決定的な実装から

RULE:

Be Explicit In Your Intent

@jamie_allen

意図をハッキリさせること

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")
```

@jamie_allen

アクターには名前を付けること

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)
```

@jamie_allen

専用メッセージを作る。汎用イベントは危険。

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)

@jamie_allen

専用例外を作る

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

@jamie_allen

全てにおいて特定であること。本番環境でも役立つ。

RULE:

Do Not Expose Your Actors

@jamie_allen

アクターを公開しないこと

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

@jamie_allen

絶対にアクターを直接参照してはいけない

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

@jamie_allen

絶対に this を返してはいけない。代わりに self を使う。

Use Immutable Messages

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

@jamie_allen

不変メッセージを使うこと

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

@jamie_allen

可変データはコピーを送る

Avoid Sending Behavior

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

@jamie_allen

振る舞いを送ることは避ける

Takeaway

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

@jamie_allen

アクターの内部構造を外に見せないこと

RULE:

Make Debugging Easy On Yourself

@jamie_allen

デバッギングが簡単にできるようにする

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

@jamie_allen

ビジネス・ロジックは外に書く

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

@jamie_allen

読んで意味のあるログを書く

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

@jamie_allen

メッセージにユニークIDをふる

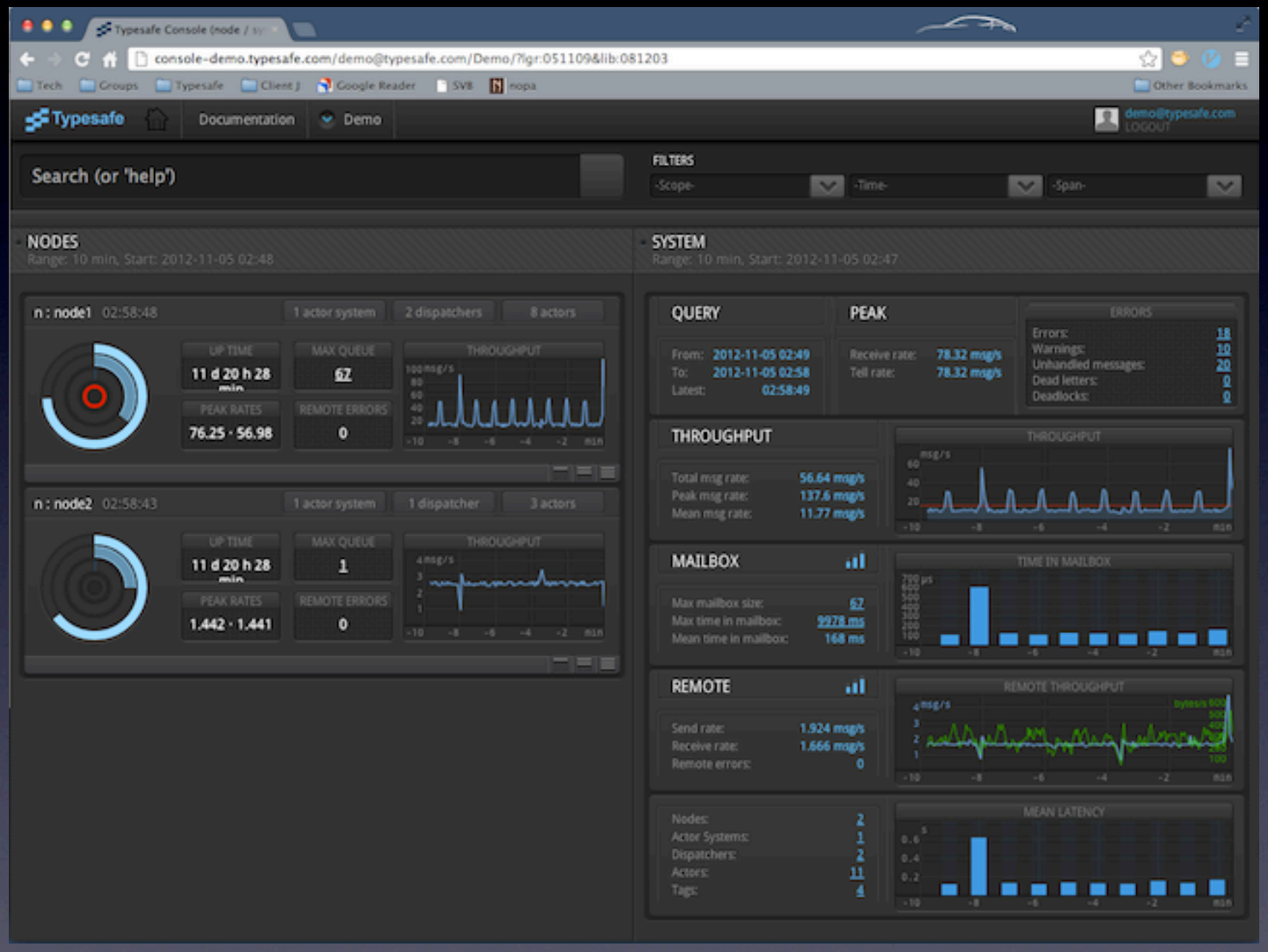
Monitor Everything

- 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

@jamie_allen

全てを監視する

Typesafe Console



Typesafe

@jamie_allen

Takeaway

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

@jamie_allen

メンテが容易なアクターシステムを構築する

Thank You!

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

@jamie_allen

ご清聴ありがとうございました