Jamie Allen

# Effective Actors

Typesafe

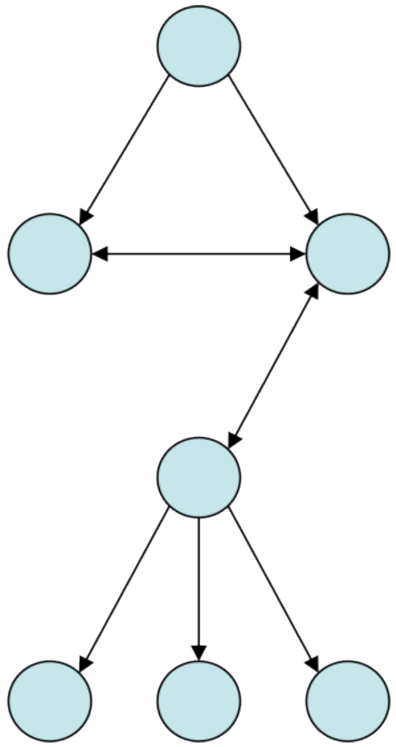
# Who Am I?

- Consultant at Typesafe
- Actor developer since 2009

jamie.allen@typesafe.com
@jamie_allen
github.com/jamie-allen

# Groundwork: What Are Actors?

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



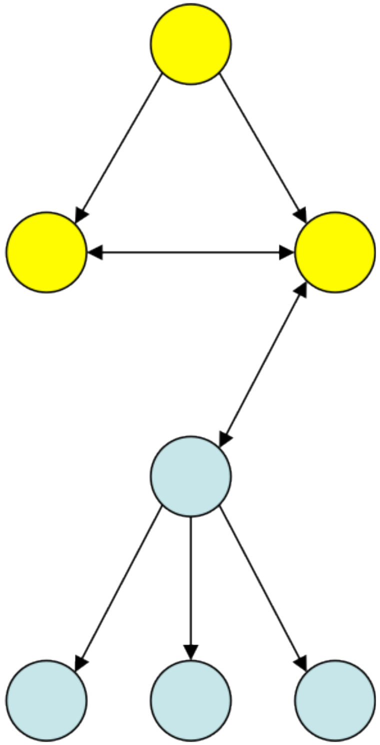
# Akka Actor Code

```
// Definitions
case class Start(thePonger: ActorRef)
class Pinger extends Actor {
    def receive = {
      case Start(x) => x ! "Ping!"
      case x => println("Pinger: " + x); sender ! "Ping!"
    }
}
class Ponger extends Actor {
  def receive = { case x => println("Ponger: " + x); sender ! "Pong!" }
}

// Execution
object PingPong extends App {
  val system = ActorSystem()
  val ponger = system.actorOf(Props[Ponger], “pinger”)
  val pinger = system.actorOf(Props[Pinger], “ponger”)
  pinger ! Start(ponger)
}
```

# Groundwork: Supervisor Hierarchy

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



# Akka Supervision Code

```
class MySupervisor extends Actor {
  override val supervisorStrategy =
    OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 minute) {
      case ae: ArithmeticException => Resume
      case np: NullPointerException => Restart
     }

  context.actorOf(Props[MyActor], "my-actor")
}
```

# Groundwork - 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

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

  val system = ActorSystem()
  val router: ActorRef = system.actorOf(Props[MyActor].
        withRouter(RoundRobinRouter(nrOfInstances = 5)), "my-router")
  for (i <- 1 to 10) router ! i
}
```

# Effective Actors

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

# 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

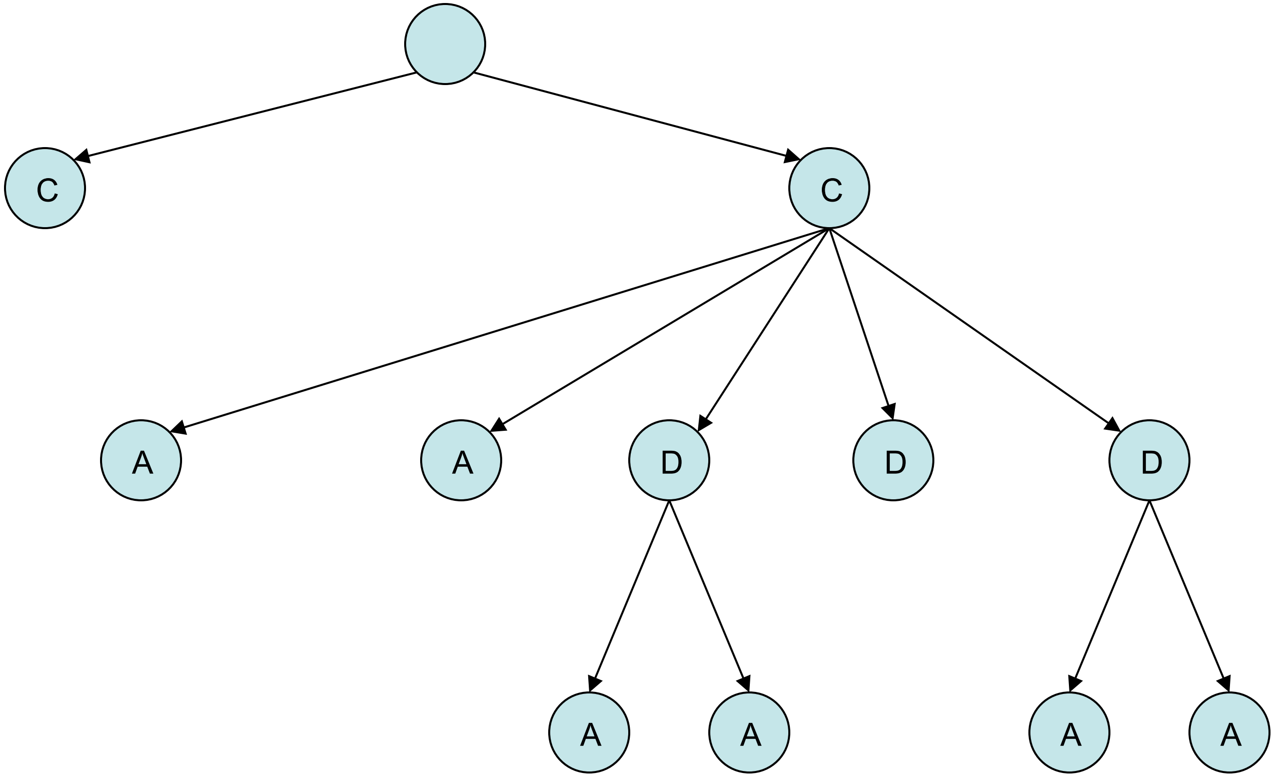
# Actor Behavior

- Handle messages
- Delegate messages
- Forward messages
- Supervise supervisors under them

# Supervision

- Every non-leaf node is technically a supervisor
- Create explicit supervisors under each node for each type of child to be managed
- Supervisors should do nothing except manage their actors

# Conflated Supervisors

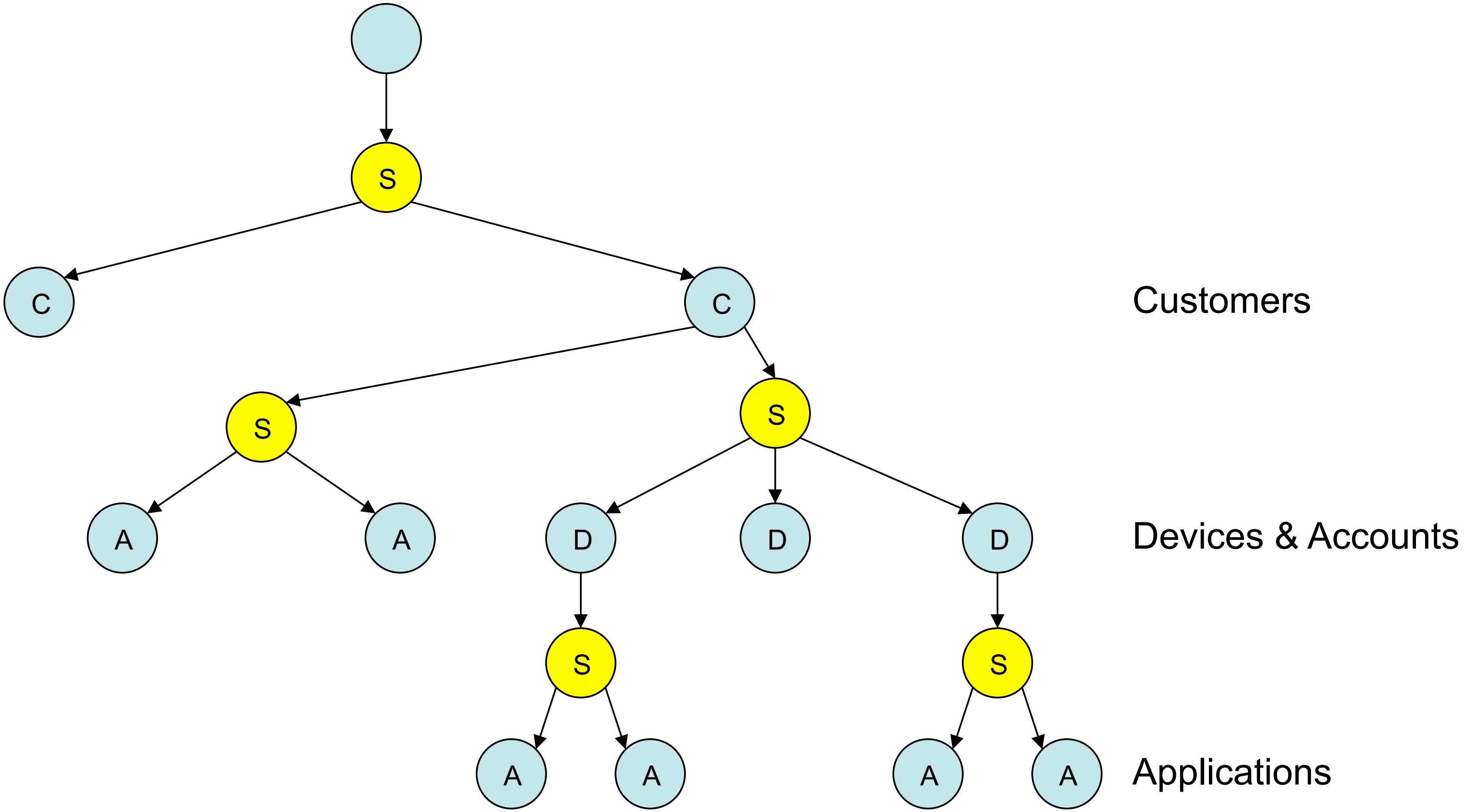



Customers

Devices & Accounts

Applications

# Explicit Supervisors



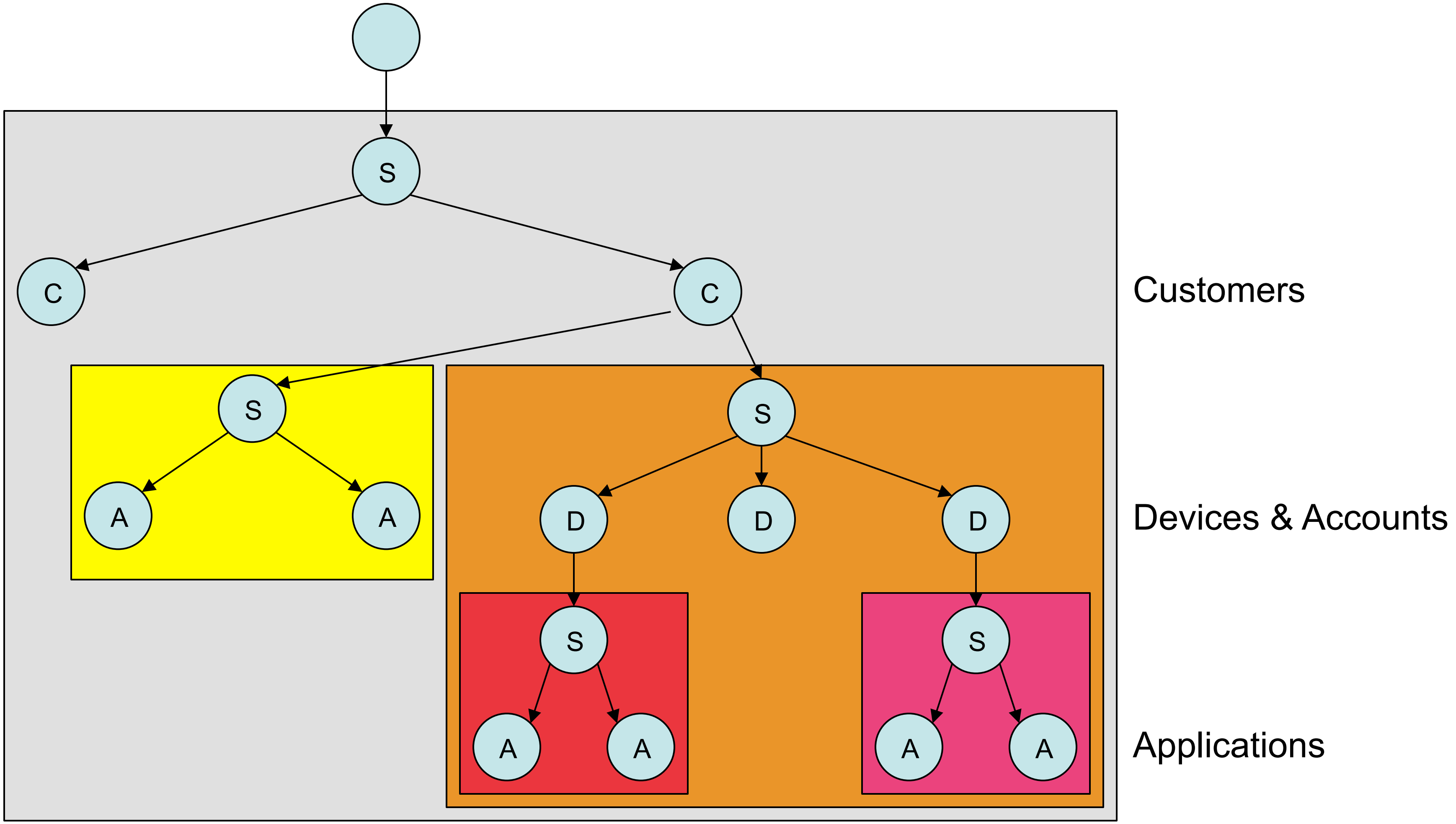
# Keep the Error Kernel Simple

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

# Failure Zones

- Multiple isolated zones with their own dispatcher
- Protects thread pools to prevent starvation
- Prevents issues in one branch from affecting another

# Failure Zones



# Takeaway

- For reasonably complex actor systems, shallow trees are a smell test
- Actors are cheap - use them

# RULE

THOU SHALT NOT BLOCK

# Consequences of Blocking

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

# Futures and Timeouts

- Futures are composable tasks to be performed asynchronously
- Futures can easily be performed sequentially or in parallel in Scala
- Timeout within a reasonable period of time
- Exponential backoff
- Handle failure

# Futures

```
case class Start(ref: ActorRef)
case class SumSequence(ints: Seq[Int])
class Worker extends Actor {
  def receive = {
    case s: SumSequence => sender ! (
     try { s.ints.reduce(_ + _) }
     catch { case NonFatal(e) => println("Non-fatal exception") })
  }
}
class Delegator extends Actor {
  implicit val timeout: Timeout = 2 seconds
  def receive = {
    case Start(worker) =>
      val workFut = worker ? SumSequence(1 to 100)
      workFut.onComplete {
        case Left(x: Throwable) => {
          println("Exception: %s".format(x.getMessage))
          self ! Start(worker)
        }
        case Right(y) => println("Got a result: " + y)
      }
  }
}

object Bootstrapper extends App {
  val system = ActorSystem()
  val worker = system.actorOf(Props[Worker], "my-worker")
  val delegator = system.actorOf(Props[Delegator], "my-delegator")

  delegator ! Start(worker)
}
```

# Sequential vs Parallel Futures

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


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

# What If I Must Block?

- An example is database access
- Use a specialized actor with its own dispatcher
- Pass messages to other actors to handle

# Using Dispatchers

```
case class SumSequence(ints: Seq[Int])
class Worker extends Actor {
  def receive = {
    case s: SumSequence => sender ! (
      try { s.ints.reduce(_ + _) }
      catch { case NonFatal(e) => log.error(e, "Non-fatal exception")
  }
}

object Bootstrapper extends App {
  val system = ActorSystem()
  val worker = system.actorOf(Props[Worker].withDispatcher("my-dispatcher"), "my-actor1)
  implicit val timeout: Timeout = 2 seconds

  try {
    val workFut = worker ? SumSequence(1 to 100)
    workFut.onComplete {
      case Left(x: Throwable) => println("Exception: %s".format(x.getMessage))
      case Right(y) => println("Got a result: " + y)
    }
  } finally { system.shutdown }
}

my-dispatcher {
  executor = "thread-pool-executor"
  throughput = 100
}
```

# Handling I/O

- Akka provides the IOManager
- Uses an “Iteratee” for handling a stream of data without waiting for all of the data to arrive
- Great for non-blocking I/O

# 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

**THOU SHALT NOT OPTIMIZE PREMATURELY**

# Start Simple

- Make Donald Knuth happy
- Start with a simple configuration and profile
- Do not parallelize until you know you need to and where

# Initial Focus

- Deterministic
- Declarative
- Immutable

# Advice from Jonas Bonér

- Layer in complexity
- Add indeterminism (actors and agents)
- Add mutability in hot spots (CAS and STM)
- Add explicit locking and threads

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, be specific
- Dampen actor messages if the exact same message is being handled repeatedly within a certain timeframe
- Tune your dispatchers and mailboxes
  - Back-off policies
  - Queue sizes

# Takeaway

- Start by creating code that is not actor based
- Layer in complexity as you go

# RULE

## BE EXPLICIT IN YOUR INTENT

# Name Your Actors

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

# Create Specialized Messages

- Non-specific messages about general events are dangerous

  Example: `AccountsUpdated`

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

  Example: `AccountDeviceAdded(acctNum, deviceNum)`

# Create Specialized Exceptions

- Don't use 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 and at runtime

# RULE

**THOU SHALT NOT EXPOSE YOUR ACTORS**

# No Direct References to Other Actors

- 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
- 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

MAKETH DEBUGGING EASIER ON THYSELF

# 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

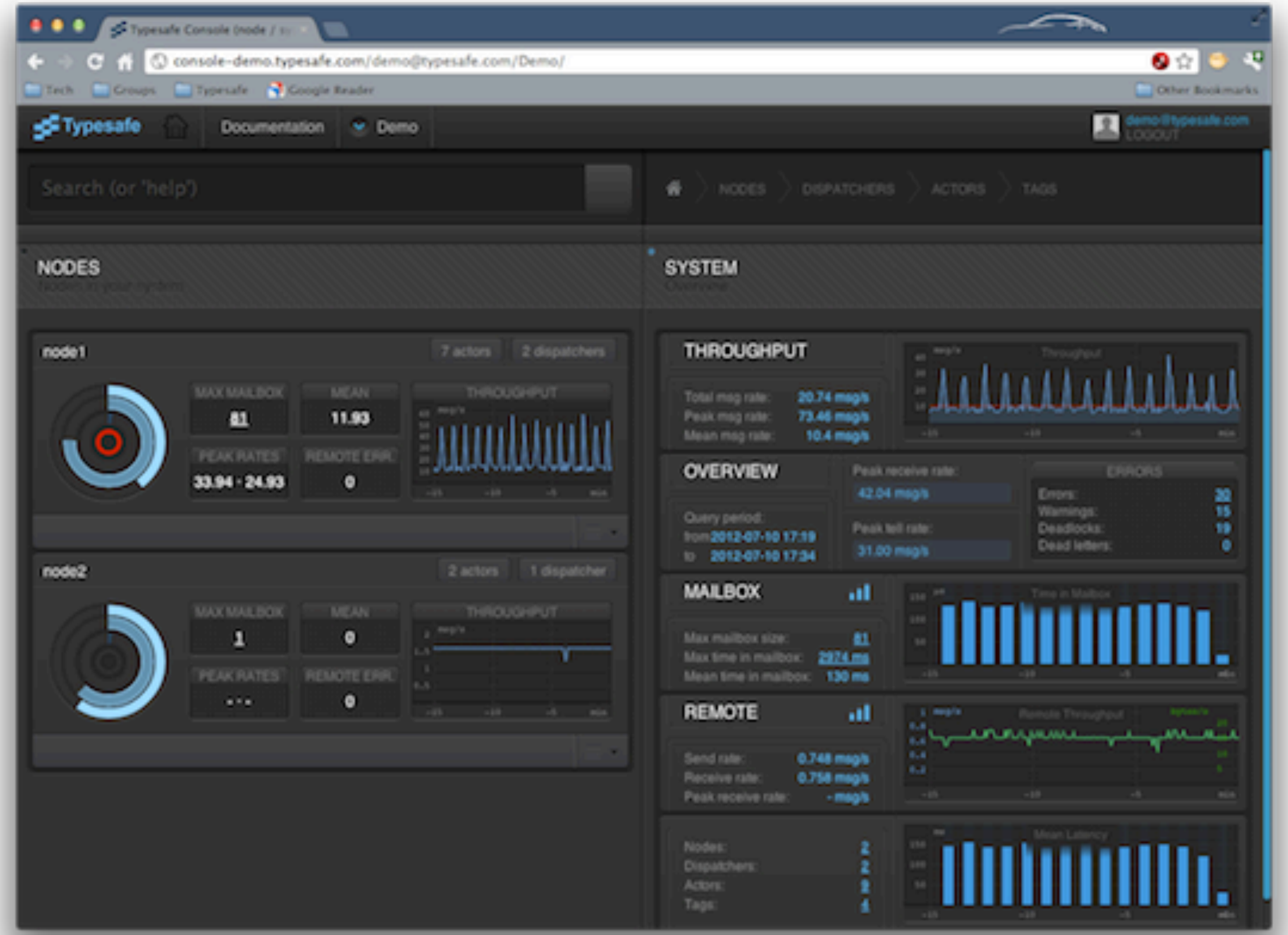
# Unique IDs for Messages

- 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 Atmos/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



# 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