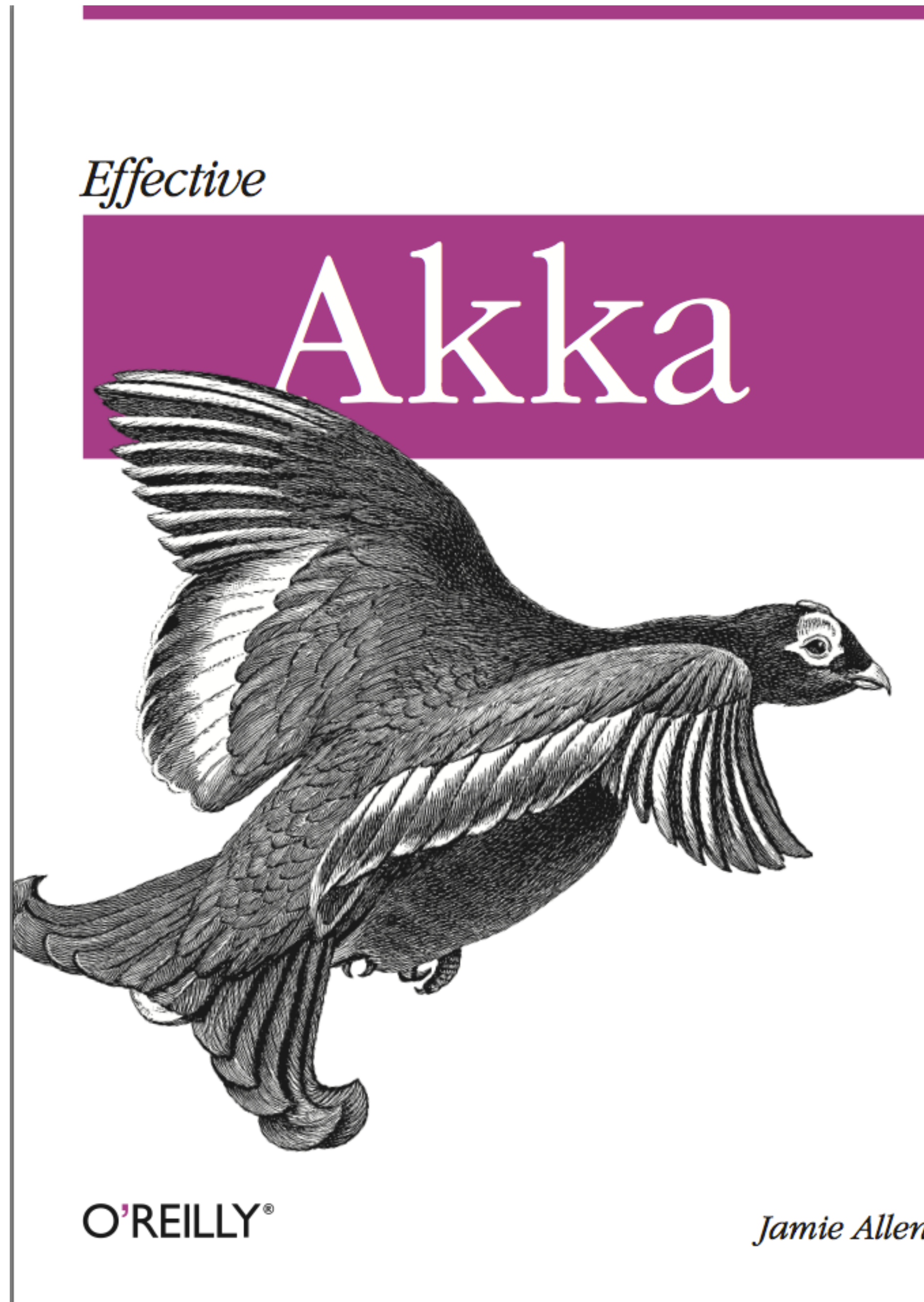
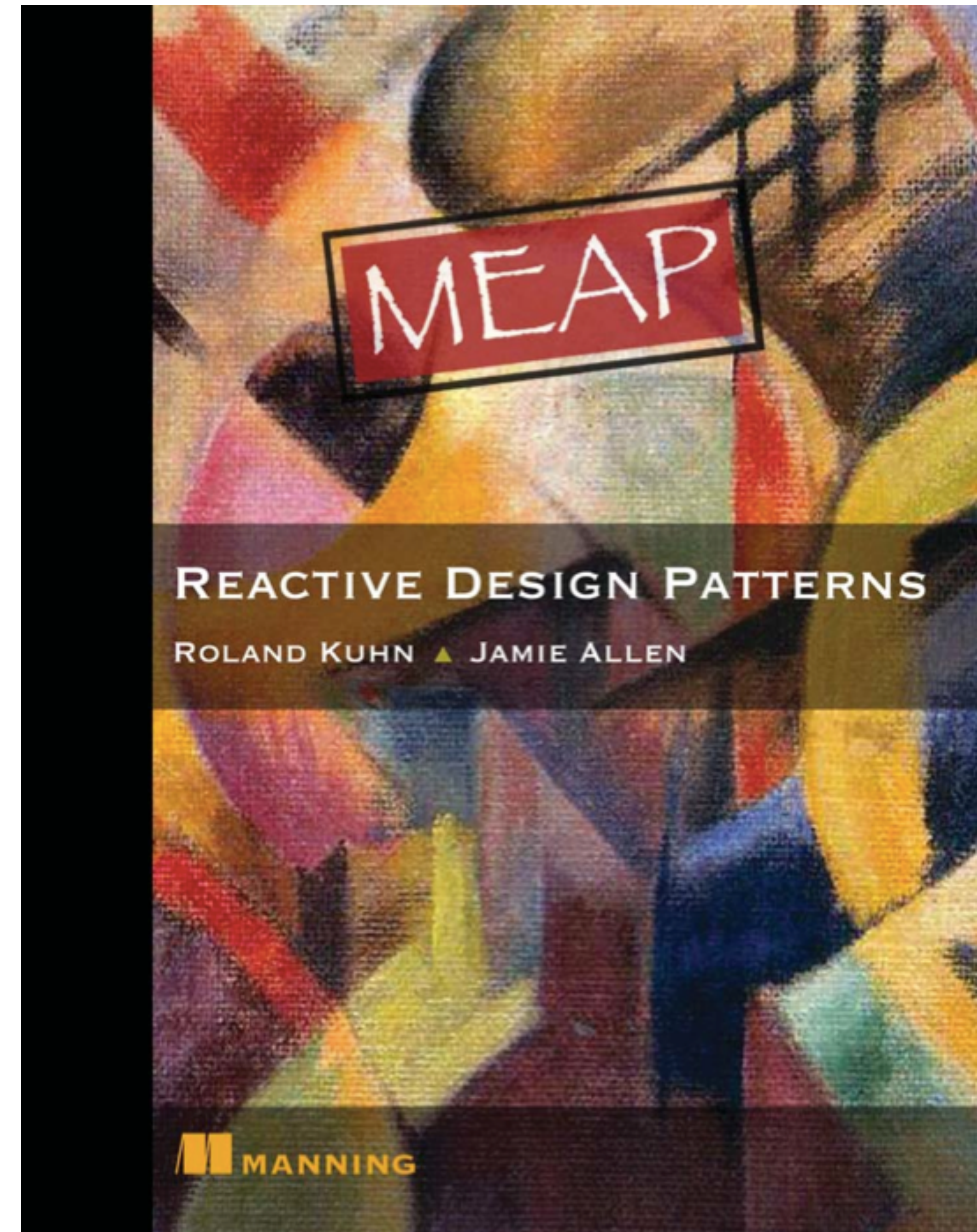
Jamie Allen

Sr. Director of Global Services

@jamie_allen





Goal

Communicate as much about what I've learned in 6+ years of actor development within one hour

We will start with a cursory overview of what Actors are and the problems they solve, then move into some real-world use cases and how to test them

What are Actors?

- An abstraction over the primitives of concurrency, asynchrony and resilience
- The embodiment of single-threaded interactions happening concurrently and asynchronously across all of the resources available to your application
- Finer-grained fault tolerance than thread pool utilities like UncaughtExceptionHandler
- Location transparency in support of greater asynchrony and resilience

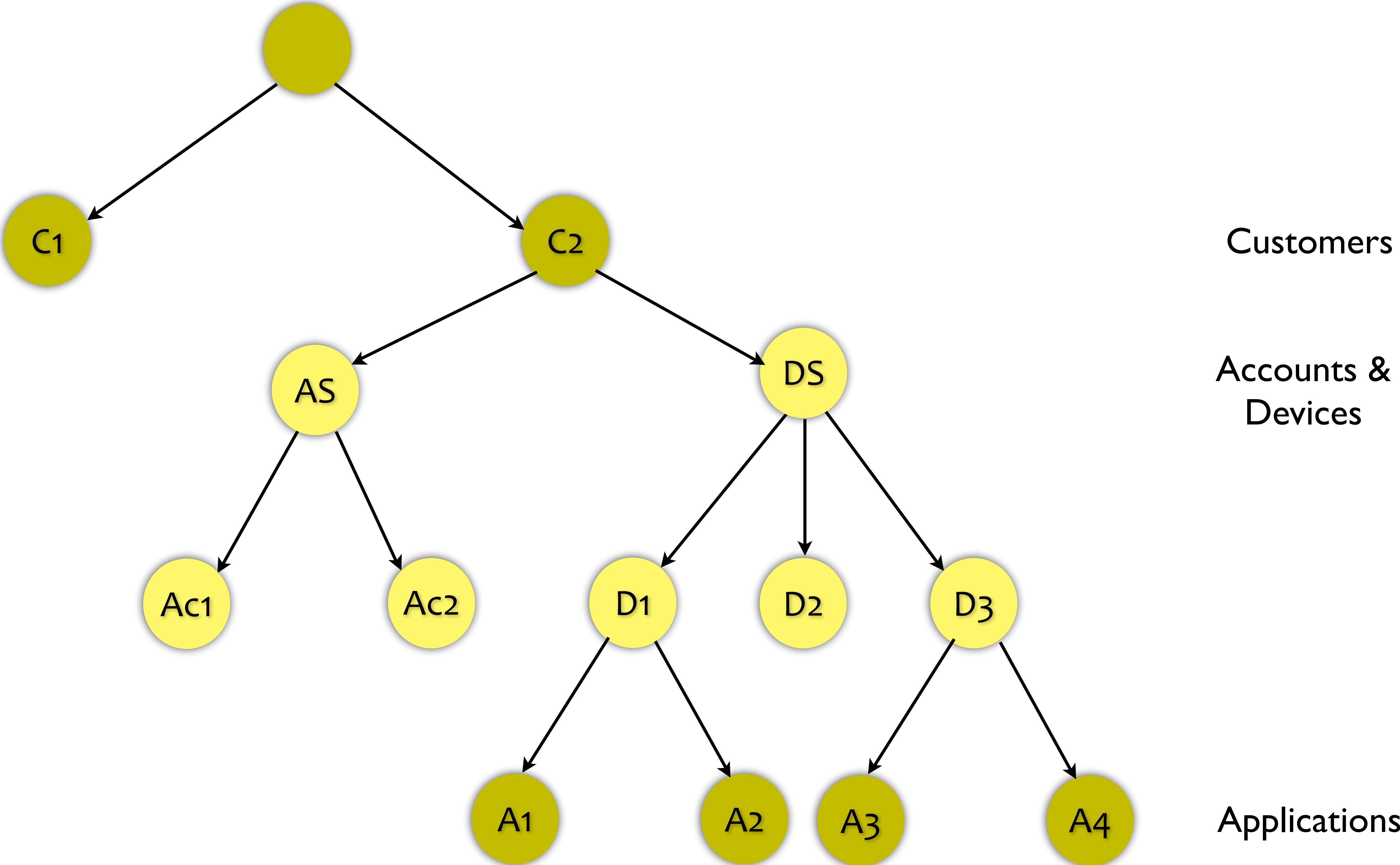
What are Actors?

- Actors never interact with each other via synchronous method calls, only messages
- Actors can be domain instances:
  - A “customer” representation, live in memory
  - Messages arrive and tell them how the world is changing around them
- Actors can be workers:
  - Encapsulate no state, they just know what to do when they get messages that have data in them

What are Actors?

- Should actors be used everywhere?
  - Probably not, but they make excellent “boundaries”
    - Across physical nodes
    - Across services
- They’re also excellent for defining strategies to handle failures you understand, as well as those you do not

What are Actors?



Actors and Pure Functional Programming are NOT Mutually Exclusive

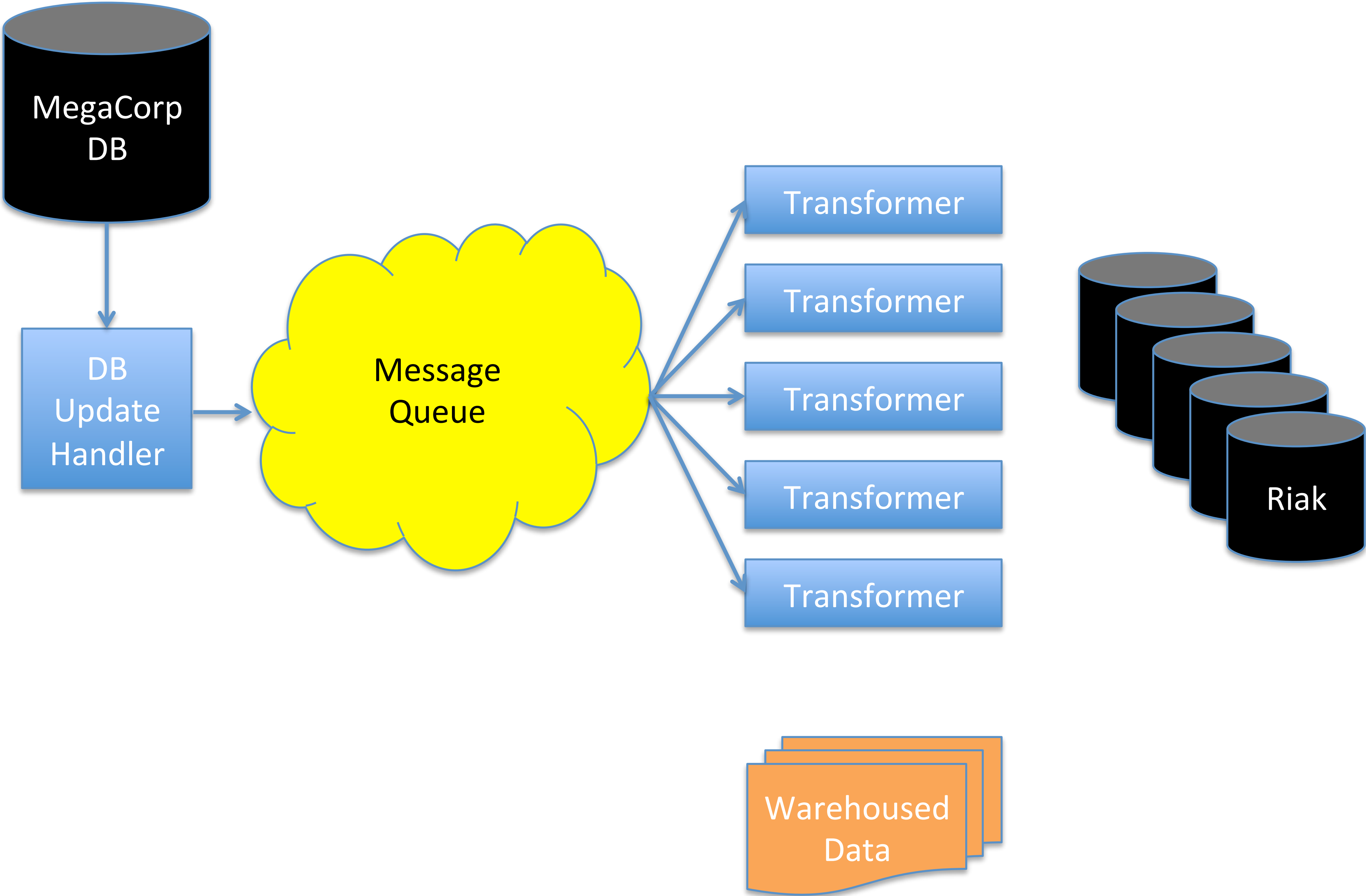
- Pure FP is all about statically-checked correctness of logic, particularly with type safety
- Actors are about resilience to failure beyond the type system
- Distributed systems offer no such guarantees except at the protocol level - how do you verify types of what messages you can send/receive through a message broker?
- **There is no type checking for hardware failures or network split brains**
- Actors help you cope with problems at this level

Typesafe

Use Case 1

- A large cable company needs to stop pinging its massive database for every On Demand web service request from someone using their remote control
- A live cache of “entitlement” data is required that is updated quickly when a customer tries to change their service
- Minimal downtime is required as On Demand viewing is one of the corporation’s largest profit centers

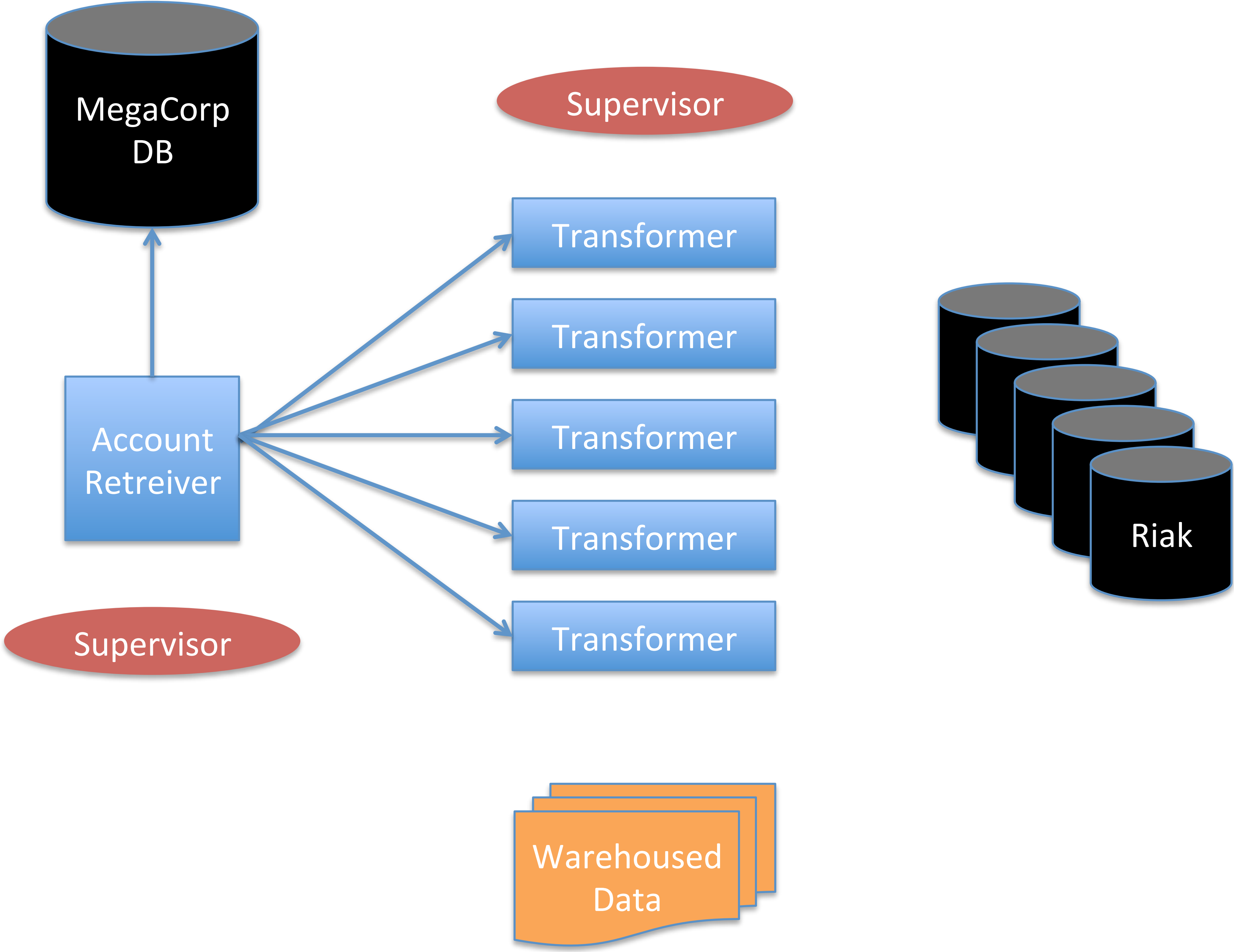
The Time-Based Approach



Issues

- A missed event means the cache is now out of synch with the database
  - Assuming we even know a failure has occurred
- A reload of the cache for all customers would be 2.5 hours
- Latency is harder to track and dependent on “burstiness" of updates
- How do you represent deleted accounts?

The Self-Healing Approach



Wins

- Fixed and tunable latency for updates depending on number of workers (and the size of their buckets of accounts)
- Resilience via supervision
- Simpler architecture with less moving parts
- Never out of synch with primary database for longer than the time it takes to handle the maximum size of a bucket of accounts
- Riak handles accounts to "delete" automatically by tombstoning records that have not been updated within a time window (session length setting)

Use Case 2

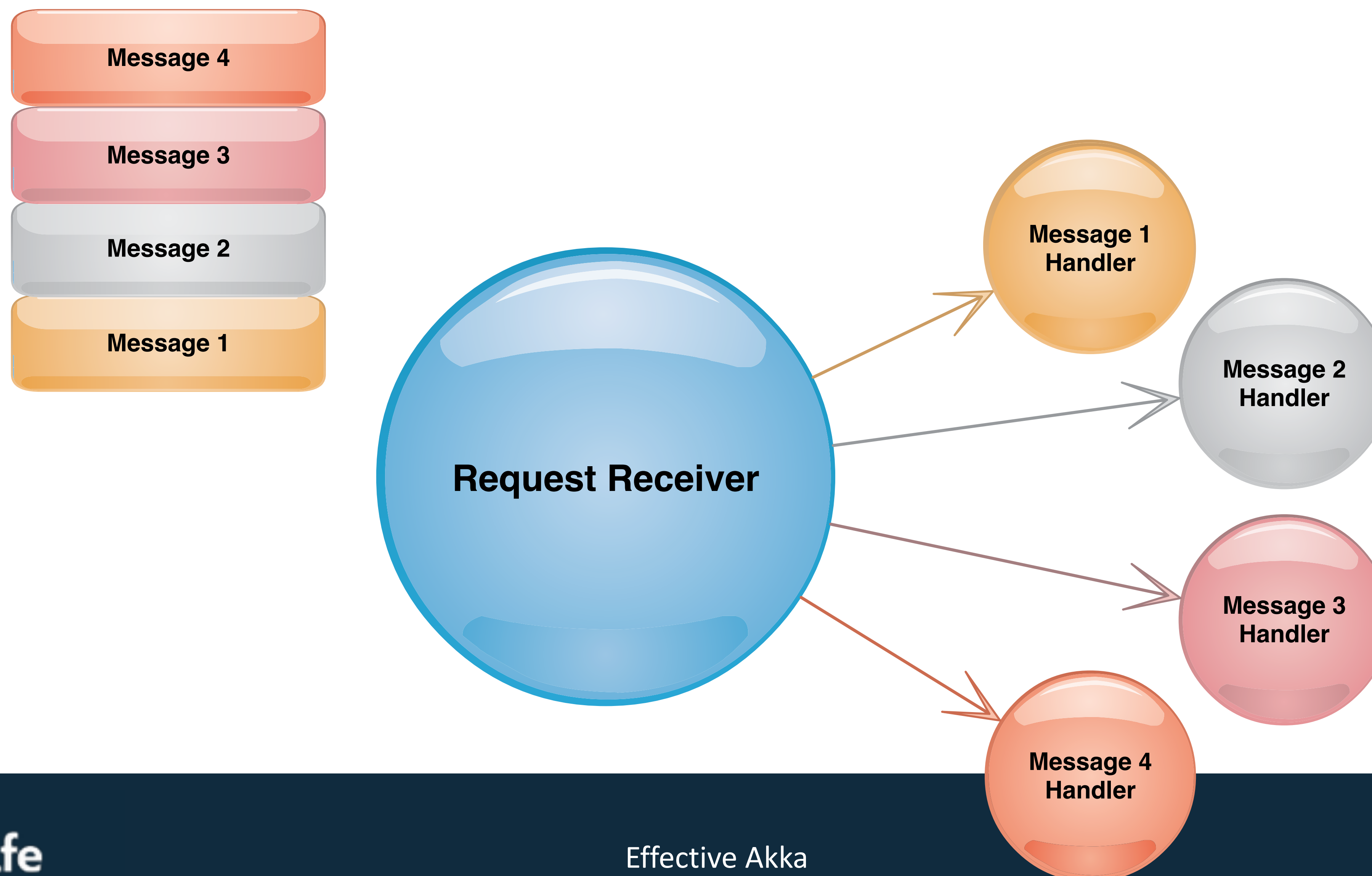
- An actor will receive a request to get all of the account balances for a customer (savings, checking and money market)
- Actor should not wait to finish handling one request before handling another
- Actor should receive service proxies from which it can retrieve each account's info
- Actor should either get responses from all three within a specified time, or send a timeout response back to the requestor

Cameo Pattern

- How to handle individual messages to an actor without making it do all of the work before handling the next message
- Similar to the Saga Pattern, but with less overhead and rules

Request Aggregation

- When an actor handles a message, it frequently has to perform multiple interactions with other services to provide a valid response
- We do not want the actor that received the message to be tied up performing that work



Transactions?

- Could be!
- You have to write the logic of how to roll back if anything fails
- But you have the control and the context to do it, especially if your effects are going to multiple external places or data stores

All Code is on GitHub

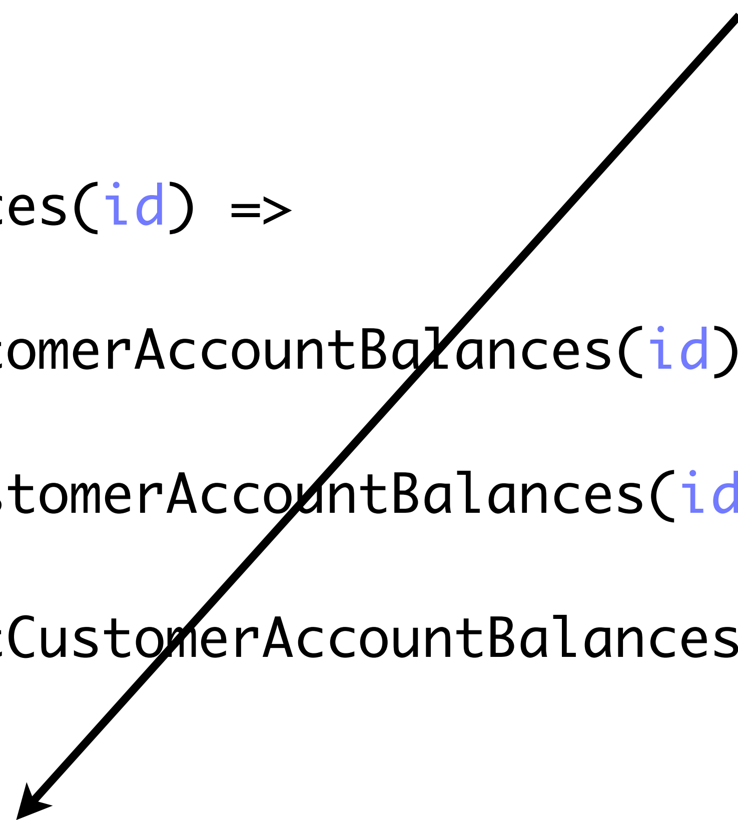
- I'm going to be showing some reasonably complex examples
- Don't worry about trying to copy the code from the slides

http://github.com/jamie-allen/effective_akka

Use Futures and Promises?

- I prefer not to. Use another actor responsible for:
  - Capturing the context (original requestor)
  - Defining how to handle responses from other services
  - Defining the timeout that will race against your
- Each Future and the resulting AskSupport have an additional cost that we do not always need to pay
- Futures do make an excellent mechanism for calling into an actor world from a non-actor context
- Futures are also more "composable" and can help define a problem in more simple terms
- Note that Future failure handling via callbacks is no more composable than Try/Catch

Use Futures and Promises?




```
def receive = {
  case GetCustomerAccountBalances(id) =>
    val futSavings =
      savingsAccounts ? GetCustomerAccountBalances(id)
    val futChecking =
      checkingAccounts ? GetCustomerAccountBalances(id)
    val futMM =
      moneyMarketAccounts ? GetCustomerAccountBalances(id)

    val futBalances = for {
      savings <- futSavings.mapTo[Option[List[(Long, BigDecimal)]]]
      checking <- futChecking.mapTo[Option[List[(Long, BigDecimal)]]]
      mm <- futMM.mapTo[Option[List[(Long, BigDecimal)]]]
    } yield AccountBalances(savings, checking, mm)
    futBalances.map(sender ! _)
  }
}
```

Capturing the Sender

- This is trickier than it sounds
- You need the “sender” value from the actor that received the original request, not the sender inside of the actor handling it!
- One of the biggest sources of actor bugs

Use Futures and Promises?

```
def receive = {
  case GetCustomerAccountBalances(id) =>
    val futSavings =
            savingsAccounts ? GetCustomerAccountBalances(id)
    val futChecking =
            checkingAccounts ? GetCustomerAccountBalances(id)
    val futMM =
            moneyMarketAccounts ? GetCustomerAccountBalances(id)

    val futBalances = for {
      savings <- futSavings.mapTo[Option[List[(Long, BigDecimal)]]]
      checking <- futChecking.mapTo[Option[List[(Long, BigDecimal)]]]
      mm <- futMM.mapTo[Option[List[(Long, BigDecimal)]]]
    } yield AccountBalances(savings, checking, mm)
    futBalances.map(sender ! _)
}
```

Use Futures and Promises?

```
def receive = {
  case GetCustomerAccountBalances(id) =>
    val futSavings =
            savingsAccounts ? GetCustomerAccountBalances(id)
    val futChecking =
            checkingAccounts ? GetCustomerAccountBalances(id)
    val futMM =
            moneyMarketAccounts ? GetCustomerAccountBalances(id)

    val futBalances = for {
      savings <- futSavings.mapTo[Option[List[(Long, BigDecimal)]]]
      checking <- futChecking.mapTo[Option[List[(Long, BigDecimal)]]]
      mm <- futMM.mapTo[Option[List[(Long, BigDecimal)]]]
    } yield AccountBalances(savings, checking, mm)
    futBalances.pipeTo(sender)
}
```

The Actor Approach

- Use an actor to encapsulate a context, such as a specific user request
- Define the values you need to retrieve/transform
- Define the behavior for what to do when you get them, or if you only get partial responses (transaction management)
- Define the response to the original requester and SHUT DOWN THE ACTOR
- Send the messages to start the work
- Set up the single competing timeout message send

Use an Anonymous Actor?

```scala
class OuterActor extends Actor
  def receive = LoggingReceive {
    case DoWork => {
      val originalSender = sender

      context.actorOf(Props(new Actor() {
        def receive = LoggingReceive {
          case HandleResponse(value) =>
            timeoutMessager.cancel
            sendResponseAndShutdown(Response(value))
          case WorkTimeout =>
            sendResponseAndShutdown(WorkTimeout)
        }

        def sendResponseAndShutdown(response: Any) = {
          originalSender ! response
          context.stop(self)
        }

        someService ! DoWork

        import context.dispatcher
        val timeoutMessager = context.system.scheduler.scheduleOnce(250 m     seconds) {
          self ! WorkTimeout
        }
      }))
    }
  }
}
```

Use an Anonymous Actor?

- I call this the "Extra" Pattern
- It's not bad, but it has drawbacks:
  - Poor stack traces due to "name mangled" actor names, like **`$a`**
  - More difficult to maintain the cluttered code, and developers have to read through the body of the anonymous actor to figure out what it is doing
  - More likely to "close over" state

Create a “Cameo” Actor

- Externalize the behavior of such an anonymous actor into a specific type
- Anyone maintaining the code now has a type name from which they can infer what the actor is doing
- Most importantly, you can’t close over state from an enclosing actor - it must be passed explicitly

Create a “Cameo” Actor

```
class WorkerActor(dependency: ActorRef) extends Actor {
  def receive = LoggingReceive {
    case HandleResponse(value) =>
      timeoutMessager.cancel
      sendResponseAndShutdown(Response(value))
    case WorkTimeout =>
      sendResponseAndShutdown(WorkTimeout)
  }

  def sendResponseAndShutdown(response: Any) = {
    originalSender ! response
    context.stop(self)
  }

  // Send request(s) required
  dependency ! GetData(1L)

  import context.dispatcher
  val timeoutMessager = context.system.scheduler.scheduleOnce(
    250 milliseconds, self, WorkTimeout)
}

class DelegatingActor extends Actor
  def receive = LoggingReceive {
    case DoWork => {
      val originalSender = sender
      val worker = context.actorOf(WorkerActor.props(), “worker”)
      someService.tell(DoWork, worker)
    }
  }
}
```

Remember to Stop the Actor

- When you are finished handling a request, ensure that the actor used is shutdown
- This is a big memory leak if you don't

```
def sendResponseAndShutdown(response: Any) = {
  originalSender ! response
  log.debug("Stopping context capturing actor")
  context.stop(self)
}
```

Write Tests!

- Always remember to write tests with your actors
- Create unit tests that check the functionality of method calls without actor interactions using TestActorRef
- Create integration tests that send messages to actors and check the aggregated results

```scala
"An AccountBalanceRetriever" should {
  "return a list of account balances" in {
    val savingsAccountsProxy = system.actorOf(Props(new SavingsAccountsProxyStub()), "svg")
    val checkingAccountsProxy = system.actorOf(Props(new CheckingAccountsProxyStub()), "chk")
    val moneyMarketAccountsProxy = system.actorOf(Props(new MoneyMarketAccountsProxyStub()), "mm")
    val accountBalanceRetriever = system.actorOf(
                                Props(new AccountBalanceRetriever(savingsAccountsProxy,
                                                                  checkingAccountsProxy,
                                                                  moneyMarketAccountsProxy)),
                                                                 "cameo-1")
    val probe1 = TestProbe()
    val probe2 = TestProbe()

    within(300 milliseconds) {
      probe1.send(accountBalanceRetriever, GetCustomerAccountBalances(1L))
      val result = probe1.expectMsgType[AccountBalances]
      result must equal(AccountBalances(Some(List((3, 15000))),
                                    Some(List((1, 150000), (2, 29000))),
                                    Some(List())))
    }
    within(300 milliseconds) {
      probe2.send(accountBalanceRetriever, GetCustomerAccountBalances(2L))
      val result = probe2.expectMsgType[AccountBalances]
      result must equal(AccountBalances(
        Some(List((6, 640000), (7, 1125000), (8, 40000))),
        Some(List((5, 80000))),
        Some(List((9, 640000), (10, 1125000), (11, 40000)))))
    }
  }
}
```

Add Non-Functional Requirements

```
within(300 milliseconds) {
  probe1.send(accountBalanceRetriever,
              GetCustomerAccountBalances(1L))
  val result = probe1.expectMsgType[AccountBalances]
  result must equal(AccountBalances(
                        Some(List((3, 15000))),
                        Some(List((1, 150000), (2, 29000))),
                        Some(List())))
}
within(300 milliseconds) {
  probe2.send(accountBalanceRetriever,
              GetCustomerAccountBalances(2L))
  val result = probe2.expectMsgType[AccountBalances]
  result must equal(AccountBalances(
    Some(List((6, 640000), (7, 1125000), (8, 40000))),
    Some(List((5, 80000))),
    Some(List((9, 640000), (10, 1125000), (11, 40000)))))
}
```

Write Moar Tests!

```
"return a TimeoutException when timeout is exceeded" in {
  val checkingAccountsProxy =
    system.actorOf(Props(new CheckingAccountsProxyStub()),“timeout-chk")

  within(250 milliseconds, 500 milliseconds) {
    probe.send(accountBalanceRetriever, GetCustomerAccountBalances(1L))
    probe.expectMsg(AccountRetrievalTimeout)
  }
}
```

Best Practices

Avoid Complexity of Coordination

- If your implementation can be accomplished with no coordination, you don't need Remoting or Cluster and are linearly scalable
- Use Remoting if:
  - You need to scale across nodes but don't need awareness of nodes going down
  - You don't need to scale "tiers" or roles in the cluster independently of one another
  - You can get away with DeathWatch and simple multi-node routing
- Use Cluster if:
  - You need to know if nodes went down to spin up an actor elsewhere
  - You need independent, managed scalability across tiers

Don’t Create Actors By Type Signature

- Akka Actors can be created by passing a type to the Props constructor
- If you add parameters to the actor later, you don’t get a compile time error

```
val myActor = context.actorOf(Props[AccountBalanceResponseHandler])
```

Create a Props Factory

- Creating an actor within another actor implicitly closes over "this"
- Necessary until spores (SIP-21) are part of Scala, always necessary from Java
- Create a Props factory in a companion object

```
object AccountBalanceResponseHandler {
  def props(savingsAccounts: ActorRef,
            checkingAccounts: ActorRef,
            moneyMarketAccounts: ActorRef,
            originalSender: ActorRef): Props = {

    Props(new AccountBalanceResponseHandler(savingsAccounts, checkingAccounts,
      moneyMarketAccounts, originalSender))
  }
}
```

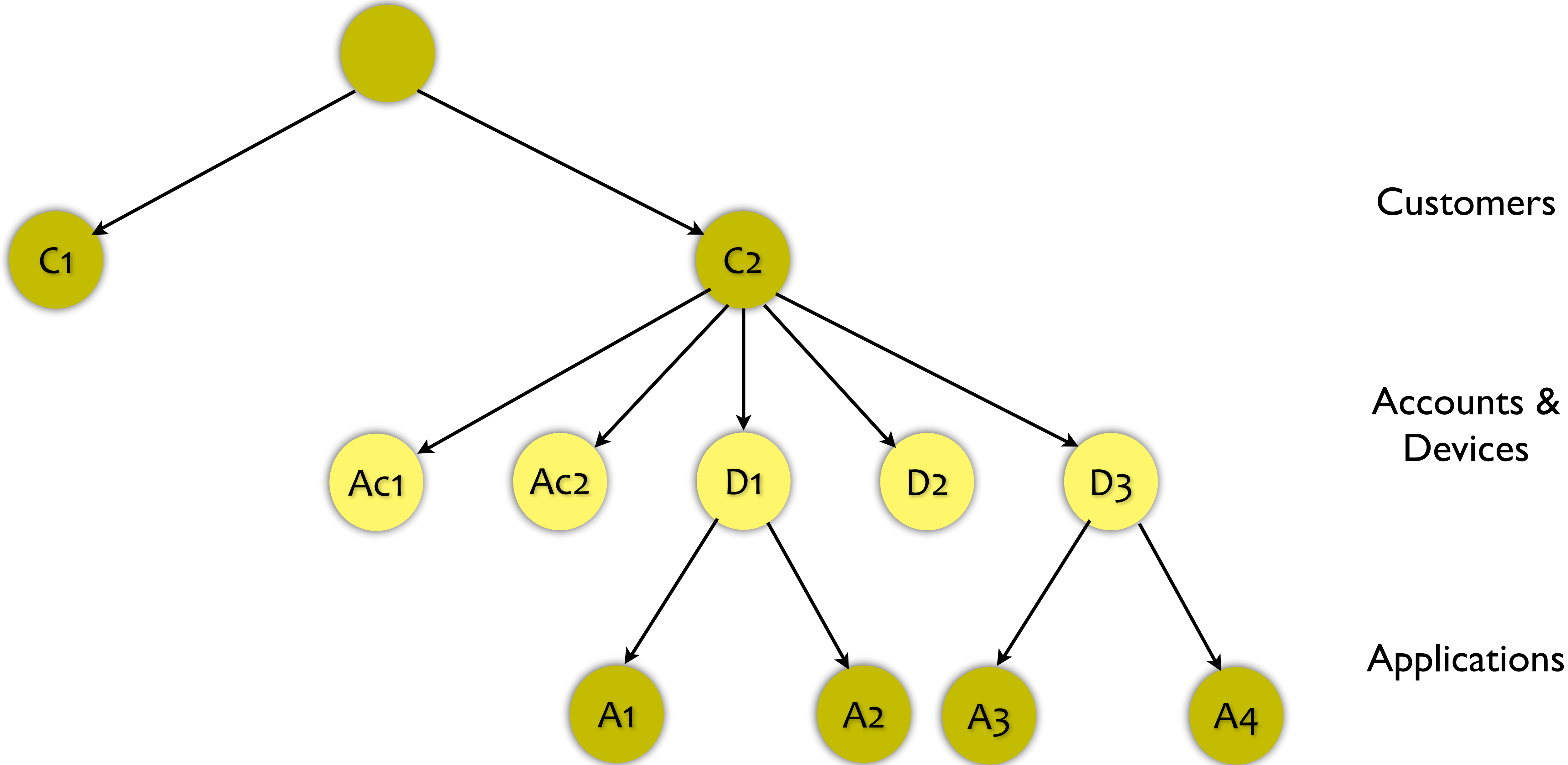
Keep Your Actors Simple

- 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

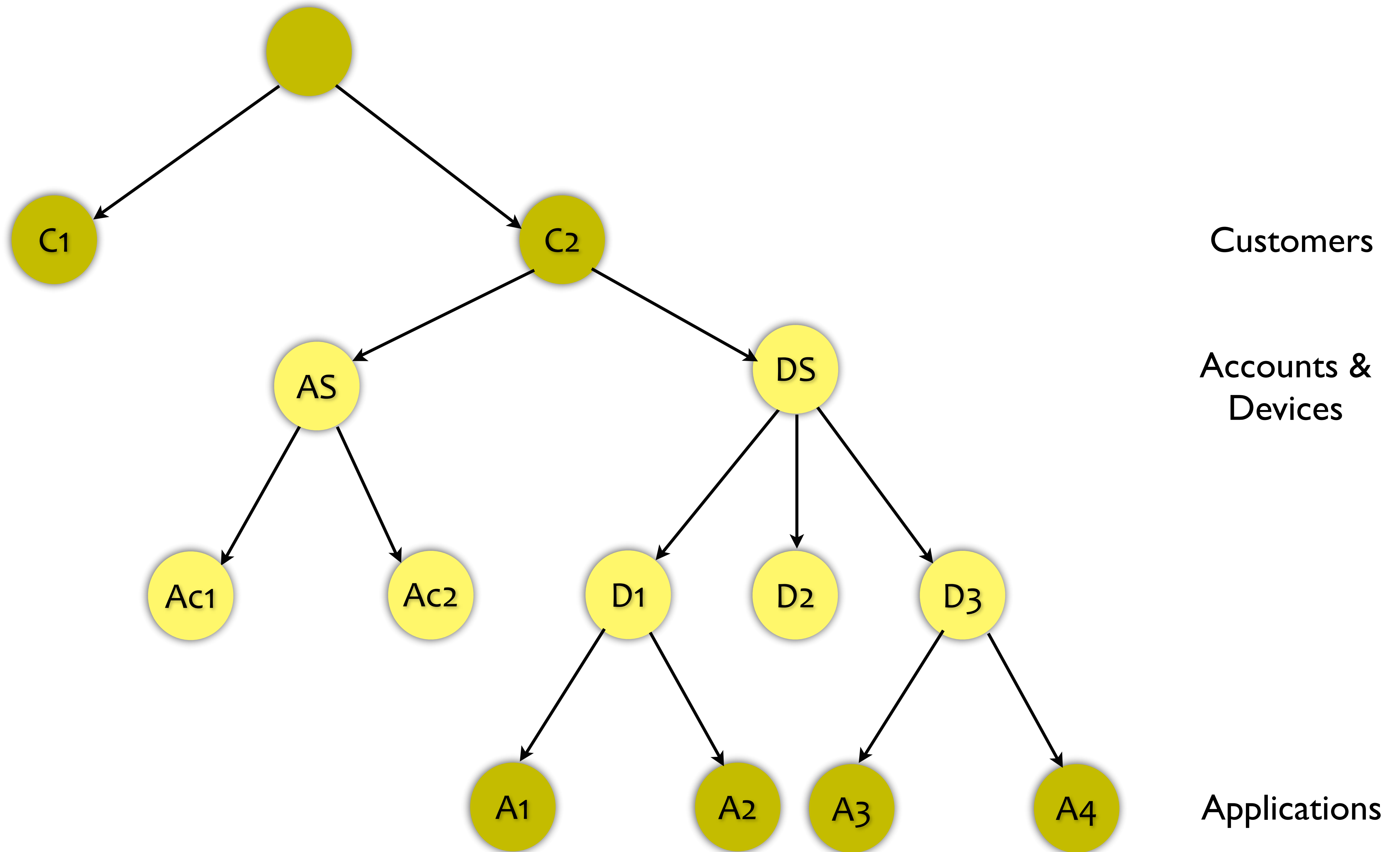
Be Explicit in Supervision

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

Conflated Supervision



Explicit Supervision

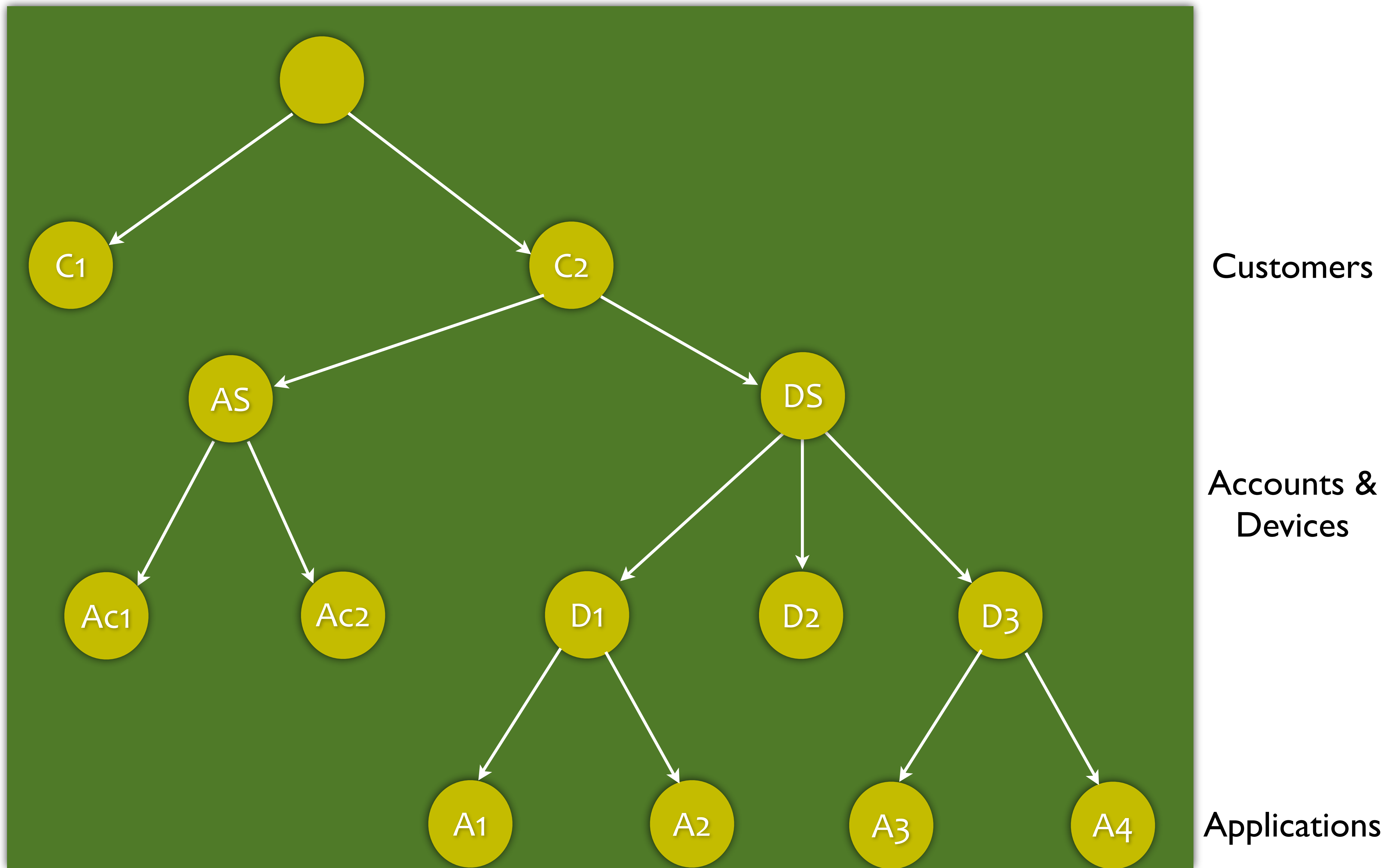


Use Failure Zones

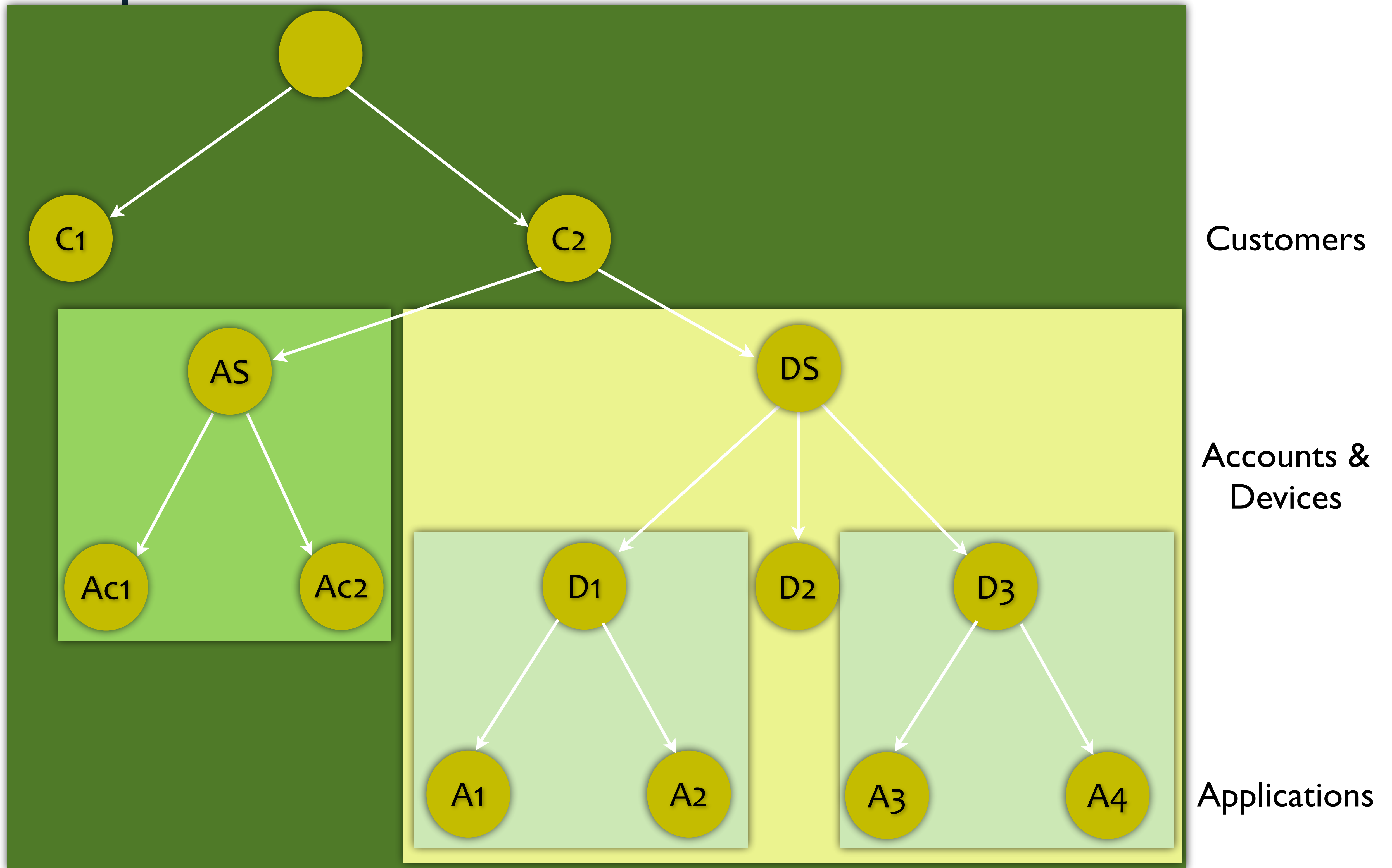
- Multiple isolated zones with their own resources (thread pools, etc)
- Prevents starvation of actors
- Prevents issues in one branch from affecting another

```
val responseHandler = system.actorOf(
      AccountBalanceResponseHandler.props(),
      "cameo-handler").withDispatcher(
      "handler-dispatcher")
```

No Failure Zones



Explicit Failure Zones



Push, Pull or Backpressure?

- If using Reactive Streams (Akka Streams/RxJava/etc), you get back pressure for free
- If not, you have to choose the model and pain you want to endure
  - Pull can load up the producer
  - Push can load up the consumer(s)
  - Rules:
    - Start with no guarantees about delivery
    - Add guarantees only where you need them
    - Retry until you get the answer you expect, or timeout
    - Switch your actor to a "nominal" state if successful

Create Granular 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)
```

- Don’t reuse messages, even if they have similar usages!
- Hurts refactoring

Create Specialized Exceptions

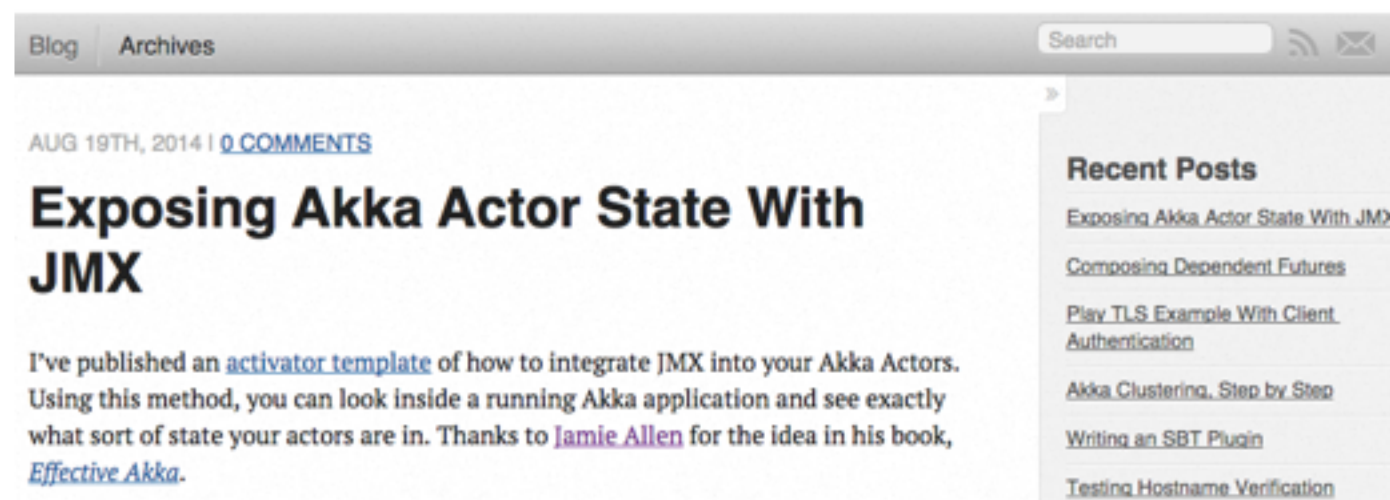
- Don't use java.lang.Exception to represent failure in an actor
- Specific exceptions can be handled explicitly
- State can be transferred between actor incarnations in Akka (if need be)

Never Reference “this”

- Actors die
- Doesn't prevent someone from calling into an actor with another thread
- Akka solves this with the ActorRef abstraction
- Never expose/publish “**this**”
- Loop by sending messages to “**self**”
- Register by sending references to your “**self**”

Never Reference "this" - **Exception is JMX**

- Instrument every actor containing state with JMX MxBeans
- Only use accessors, do not use "operations"
- Akka Actor Paths are a natural MxBean ObjectName
- Gives you visibility into state that no monitoring tool can provide
- See Will Sargent's excellent blog post about this at tersesystems.com



Immutable Interactions

- All messages passed between actors should be immutable
- All mutable data to be passed with messages should be copied before sending
- You don’t want to accidentally expose the very state you’re trying to protect

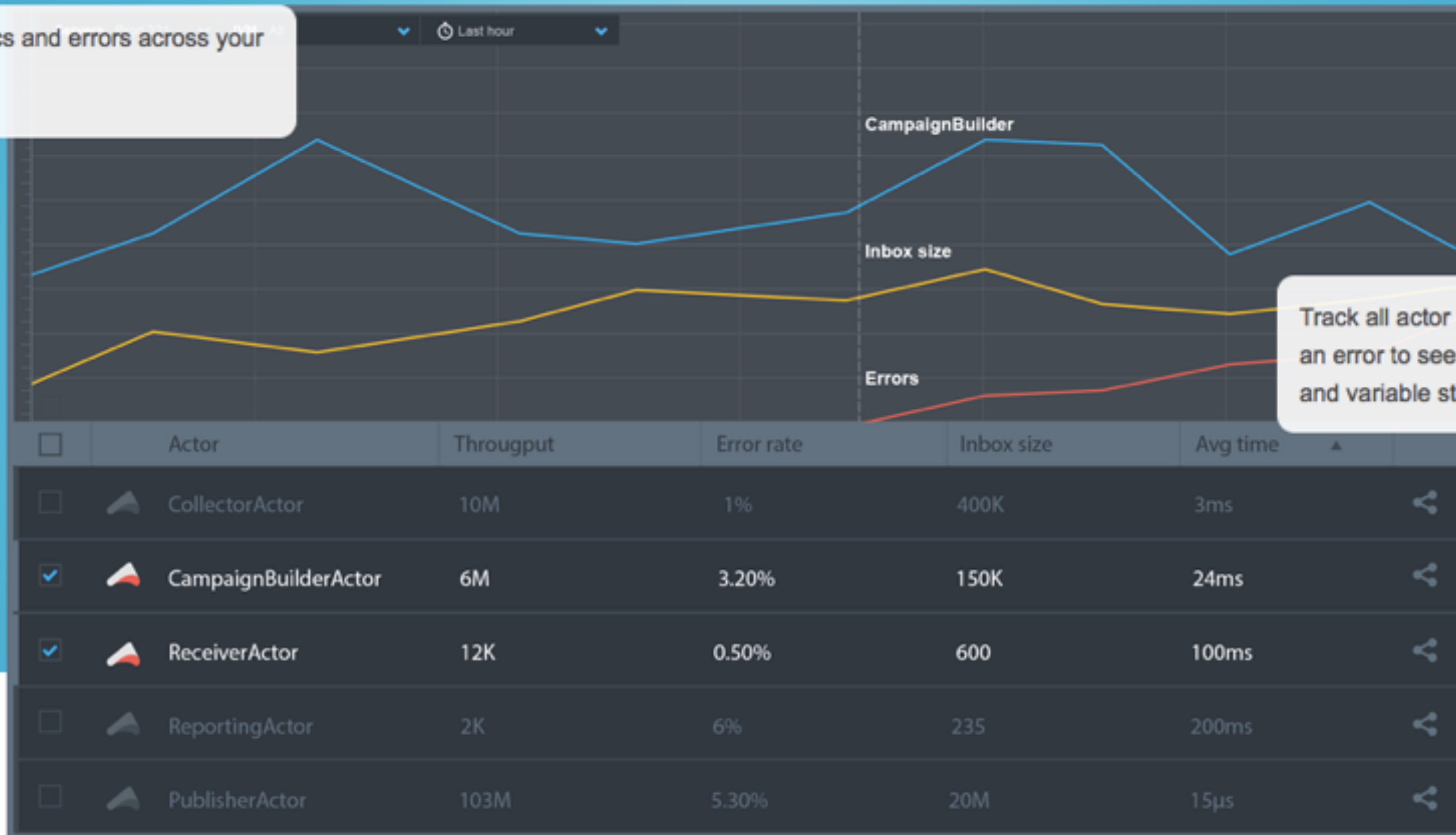
Externalize Logic

- Consider using external functions in objects to encapsulate complex business logic
  - Now only data passed in as operands can be accessed, supports purity
- 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

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

Monitoring is Coming Back!



Monitoring is Coming Back!

- Visual representations of actors at runtime are invaluable tools
- Keep an eye out for actors whose mailboxes never drain and keep getting bigger
- Look out for message handling latency that keeps going higher
- These are signs that the actors cannot handle their load
  - Optimize with routers
  - Rethink usage of Dispatchers
  - Look at “throughput” setting for some groups of actors to batch message handling

Monitoring will be a SPI

- You can tap into the stream and work with it as well
- Will work with Graphite, Coda Hale Metrics, statsd and more
- Will require a Production Success Subscription from Typesafe

```
[trace] actor received: Actor[akka://example/system/log1-Logging$DefaultLogger#355895703] ! Info(akka://example/user/a/b,class example.B,rec
eived: Message(count: 5)) (sender = Actor[akka://example/deadLetters], context = MessageSentAt(1434443582124649000))
[INFO] [06/16/2015 10:33:02.124] [example-akka.actor.default-dispatcher-2] [akka://example/user/a/b] received: Message(count: 5)
[trace] actor completed: Actor[akka://example/system/log1-Logging$DefaultLogger#355895703] ! Info(akka://example/user/a/b,class example.B,re
ceived: Message(count: 5)) (sender = Actor[akka://example/deadLetters], context = MessageSentAt(1434443582124649000))
[trace] clear context
[trace] actor told: Actor[akka://example/user/a#1855574778] ! Tick (sender = null, context = MessageSentAt(1434443583146600000))
[trace] actor received: Actor[akka://example/user/a#1855574778] ! Tick (sender = Actor[akka://example/deadLetters], context = MessageSentAt(
1434443583146600000))
[trace] actor completed: Actor[akka://example/user/a#1855574778] ! Tick (sender = Actor[akka://example/deadLetters], context = MessageSentAt
(1434443583146600000))
[trace] clear context
[trace] actor stopped: Actor[akka://example/user/a/b#-881303314]
[trace] actor stopped: Actor[akka://example/user/a#1855574778]
[trace] actor stopped: Actor[akka://example/user]
[trace] actor told: Actor[akka://example/system] ! Terminated(Actor[akka://example/user]) (sender = Actor[akka://example/user], context = Me
ssageSentAt(1434443583159331000))
[trace] actor received: Actor[akka://example/system] ! Terminated(Actor[akka://example/user]) (sender = Actor[akka://example/user], context
= MessageSentAt(1434443583159331000))
[trace] actor completed: Actor[akka://example/system] ! Terminated(Actor[akka://example/user]) (sender = Actor[akka://example/user], context
 = MessageSentAt(1434443583159331000))
[trace] actor stopped: Actor[akka://example/system/log1-Logging$DefaultLogger#355895703]
[trace] clear context
```

AsyncDebugger is Now Here in Scala IDE v4.2!

- Feature of the FOSS Scala IDE
- Ability to “walk” an actor message send and see where it goes
- Ability to retrace backwards where a message came from, and see the state of the actor at that time
- Ability to “walk” into Futures as well
- Documentation: http://scala-ide.org/docs/current-user-doc/features/async-debugger/index.html

©Typesafe 2015 – All Rights Reserved