Akka Concurrency Works

by Duncan K. DeVore,

Viridity Energy, Inc.

About Viridity: Energy Software Company

- Industrials, data centers, universities, etc.
- Help customers manage
  - Renewables & storage
  - Controllable load
  - Forecasting
  - Energy assets

About Viridity: VPower Software Platform

- Suite of applications
- Distributed & cloud based
- Micro service architecture
- Reactive philosophy
    - Event-driven, responsive, resilient, scalable
- Transform energy profiles into financial returns

About Me: VP, Software Engineering

- 25 years
- Enterprise applications
- Distributed computing
- Reactive applications
- Open source - Akka Persistence Mongo
- Scala, Akka, Testing, Agile
- Book: Manning, Building Reactive Applications

Outline

- How many with concurrency experience?
- How many with Scala/Akka experience?
- Concurrency
- Java
- Reactive
- Scala
- Akka

Concurrency: Definition

In computer science, **concurrency** is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other.

— *Google*

Concurrency: The Early Days

- Computers ran one program at a time
- From start to end
- Had access to all of the machines resources
- Sequential computing model
- This was very inefficient and expensive

Concurrency: The Process

- More than one program could run at once (not concurrently)
- Isolated independent execution of programs
- OS would allocate resources (memory, file handles, etc.)
- Communication (sockets, shared memory, semaphores, etc.)
- Process schedulers
- Multi-tasking, time sharing

Concurrency: The Thread

- Multiple program control flow
- Coexist within the same process
- Path to hardware parallelism
- Simultaneous scheduling
- Run on multiple CPU's
- Non-sequential computing model
- Awesome, multiple things at once!
- But there are challenges...

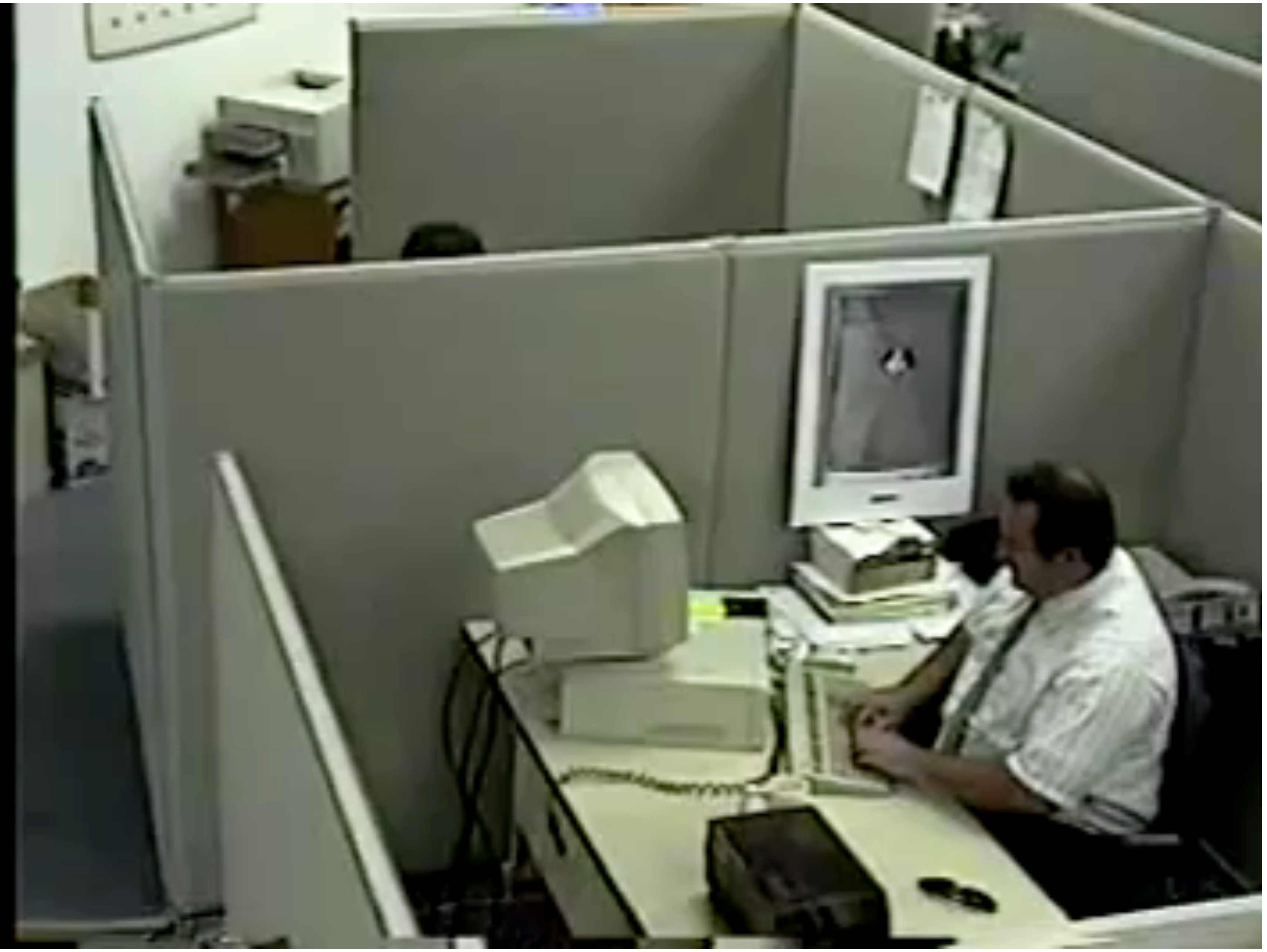
Concurrency: Not Easy!

- Non-determinism
- Shared Mutable State
- Amdahl's Law
- Exponential growth of problem

Concurrency: Non-Determinism

Although threads seem to be a small step from sequential computation, in fact, they represent a **huge step**. They discard the most essential and appealing properties of sequential computation: **understandability**, **predictability**, and **determinism**. Threads, as a model of computation, are wildly non-deterministic, and the job of the programmer becomes one of **pruning** that **nondeterminism**.

— The Problem with Threads, Edward A. Lee, Berkeley 2006



Concurrency: Non-Determinism

- What is going on?
- Try using a debugger
- Ok, I'll use a print statement
- Ok, I'll use logging

Imagine a man walking down a path in a forest and, every time he steps further, he must pick which fork in the road he wishes to take.

— *Wikipedia*

Concurrency: Shared State

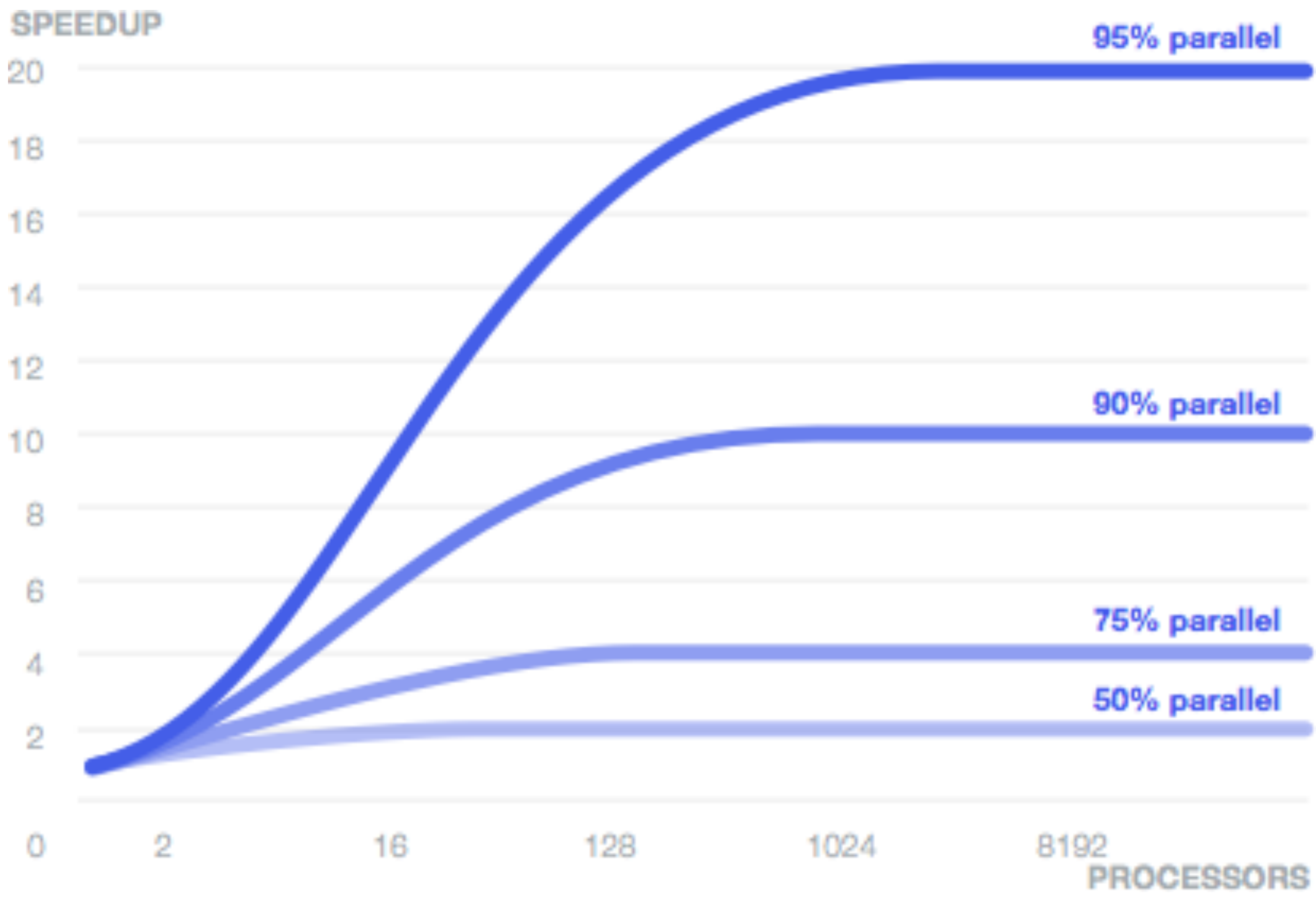
Imperative programming, the most popular form of structured programming, is centered around the notion of **sequential execution** and **mutable state**.

- Derived from the Von Neuman architecture
- Works great in a sequential single threaded environment
- Not fun in a multi-threaded environment
- Not fun trying to parallelize
- Locking, blocking, call-back hell

Concurrency: Amdahl's Law

The speedup of a program using multiple processors in parallel computing is **limited by the sequential** fraction of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20× as shown in the diagram, no matter how many processors are used.

— *Wikipedia*



Concurrency: Exponential Growth

- The days of increasing clock speed are over
- Faster switches will not help
- Multi-core systems are common place
  - Four or more cores are now common
  - 10 or more cores are coming soon!
- Performance is based on concurrency and multiple cores

Concurrency: Exponential Growth

- Programmers must embrace concurrent programming
- Local = multi-core, multi-core = distributed
- Distributed systems are the future
  - Resilience (not just fault tolerance)
  - Scaling for load (both in and out)
  - Responsiveness (users don't care)

Concurrency: Definition (Real One)

Madness, mayhem, **heisenbug**, bohrbug, mandelbug and general all around pain an suffering.

— *me*

Concurrency: Solutions?

- Solutions Exist
- Some Hard
- Some not so Hard
- Java
- Scala
- Akka

Java

- Imperative Style
- Shared State (the elephant in the room)
- Atomic Variables
- Locking
- Executors & Thread Pools
- ExecutorService & Futures

Java: Imperative Style

Characteristic	How its Handled
Focus	How to perform tasks and track state changes
State Changes	Important
Order of Execution	Important
Flow Control	Loops, conditionals and methods
Manipulation Units	Instances of structures or classes

Java: Imperative Style

The better argument for functional programming is that, in modern applications involving highly concurrent computing on multicore machines, **state is the problem**. All imperative languages, including object-oriented languages, involve multiple threads changing the shared state of objects. This is where deadlocks, stack traces, and low-level processor cache misses all take place. **If there is no state, there is no problem**.

– *JavaWorld*, 2012

Java: Shared State

If multiple threads access the same mutable state variable without appropriate synchronization, **your program is broken**. There are three ways to fix it:
* **Don't share** the state variable across threads;
* Make the state variable **immutable**; or
* Use synchronization when accessing state

— *Java Concurrency In Practice*

Java: Atomic Variables

- Implement low level machine instructions
- Atomic and non-blocking
- Scalable & performant
- compare-and-swap operation (CAS)
- `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, etc.

Java: Atomic Variables

- Limited number of atomic variables
- Shared state is often represented by a complex compositions
- Often compound actions are required for state mutation
- Will not work for compound actions

To preserve state consistency, update related state variables in a **single atomic operation**.

— *Java Concurrency In Practice*

Java: Locking

- Built in locking mechanism for enforcing atomicity
- Locks automatically acquired by executing thread upon entry
- Locks automatically released upon exit
- Reentrant - per-thread rather than per-invocation basis
- `synchronized`, `Lock`, `ReadWriteLock`, `Condition`

Java: Locking

- Deadlocks
- Livelocks
- Lock starvation
- Race conditions

The more complex the **shared state** composition and the more **compound actions** required to **mutate** that state, the more likely a concurrency bug.

Java: Locking

- Requires great vigilence!
- Must be used anywhere threads cross paths
- Must reason about mutable state
- Must reason about compound actions
- Must reason about deadlocks, livelocks, race conditions, etc.
- Act as *mutexes* (mutual exclusion locks) - they block - Yuck!

Java: Executors

- Simple interface for execution of logical units of work (tasks)
- Single method `execute`, replacement for thread creation
- `execute` is based on the **executor implementation**
  - Some create a new thread and launch immediately
  - Others may use an existing worker thread to run `r`
  - Others place `r` in a queue and wait for a worker thread to become available

Java: Thread Pools

- Most executor implementations use thread pools
- They consist of worker threads
- They minimize overhead due to thread creation
- Fixed thread pools
- Cached thread pools

Java: ExecutorService

- An extension of `Executor` that provides termination and a `Future` for tracking asynchronous progress
- Can be shutdown and will reject new tasks
- Has `submit` method that extends `Executor.execute` that returns a `Future`
- The `Future` can be used to cancel execution or wait for completion

Java: Futures

- Represents the result of an asynchronous computation
- `cancel` method for stopping execution
- `get` methods for waiting and returning the result
- Methods to determine if completion was normal or cancelled
- Cannot be cancelled after completion
- `get` methods are **blocking**

Reactive

Merriam-Webster defines reactive as *“readily responsive to a stimulus”*, i.e. its components are *“active”* and always ready to receive events. This definition captures the essence of reactive applications, focusing on systems that: **react to events**, **react to load**, **react to failure**, **react to users**

— *Reactive Manifesto*

Reactive

How Does this Relate to Concurrency?

Why do We Build Concurrent Applications?

Performance & Scalability!!

Reactive

Techniques to Achieve Performance & Scalability

- Asynchronous
- Non-blocking
- Message Passing
- Share Nothing

Reactive: Asynchronous

- Use async message/event passing
- Think workflow, how events flow
- This will give you
  - A more loosely coupled system
  - Easier to reason about and evolve
  - Lower latency
  - Higher throughput

Reactive: Non-Blocking

- ...unless you have **absolutely no** other choice
- Blocking kills scalability
- Use non-blocking I/O
- Use concurrency paradigms that are **lock free**

Reactive: Message Passing

- The asynchronous passing of events
- Concurrent apps equal multi-core without changes
- Naturally asynchronous and non-blocking
- Increase in parallelization opportunities
- Tend to rely on push rather than pull or poll

Reactive: Share Nothing

A share nothing architecture (SN) is a distributed computing architecture in which each node is **independent** and **self-sufficient**, and there is no **single point** of contention across the system. More specifically, **none** of the **nodes** share memory or disk storage.

— *Wikipedia*

This means **no shared mutable state**.

Reactive: Share Nothing

What Happens?

```
class SharedMutableState(stuff: Any)
class NonDeterministic(sms: SharedMutableState)

class MultiThreadedEnvironment {
    def whatHappens(sms: SharedMtableState): NonDeterministic = new NonDeterministic(sms)
}
```

In a concurrent environment, let alone a distributed system, **mutable state** is the essence of **BAD MOJO**.

Reactive: Share Nothing

Reactive: Share Nothing

Instead Use Immutable State!

```
case class ImmutableState(stuff: Any)
case class Deterministic(is: ImmutableState)

class ImmutableStateActor extends Actor {
    def receive = { # <=== workflow allows us to reason deterministically
        case msg: ImmutableState => Deterministic(msg)
    }
}
```

Reactive: Share Nothing

If multiple threads access the same mutable state variable without appropriate synchronization, **your program is broken**. There are three ways to fix it:
* **Don't share** the state variable across threads;
* Make the state variable **immutable**; or
* Use synchronization whenever accessing the state variable.

— *Java Concurrency In Practice*

Scala

- What is Scala?
- Functional style
- Future
- Promise

Scala: What is Scala?

Have the best of both worlds. Construct elegant class hierarchies for maximum code reuse and extensibility, implement their behavior using higher-order functions. Or anything in-between.

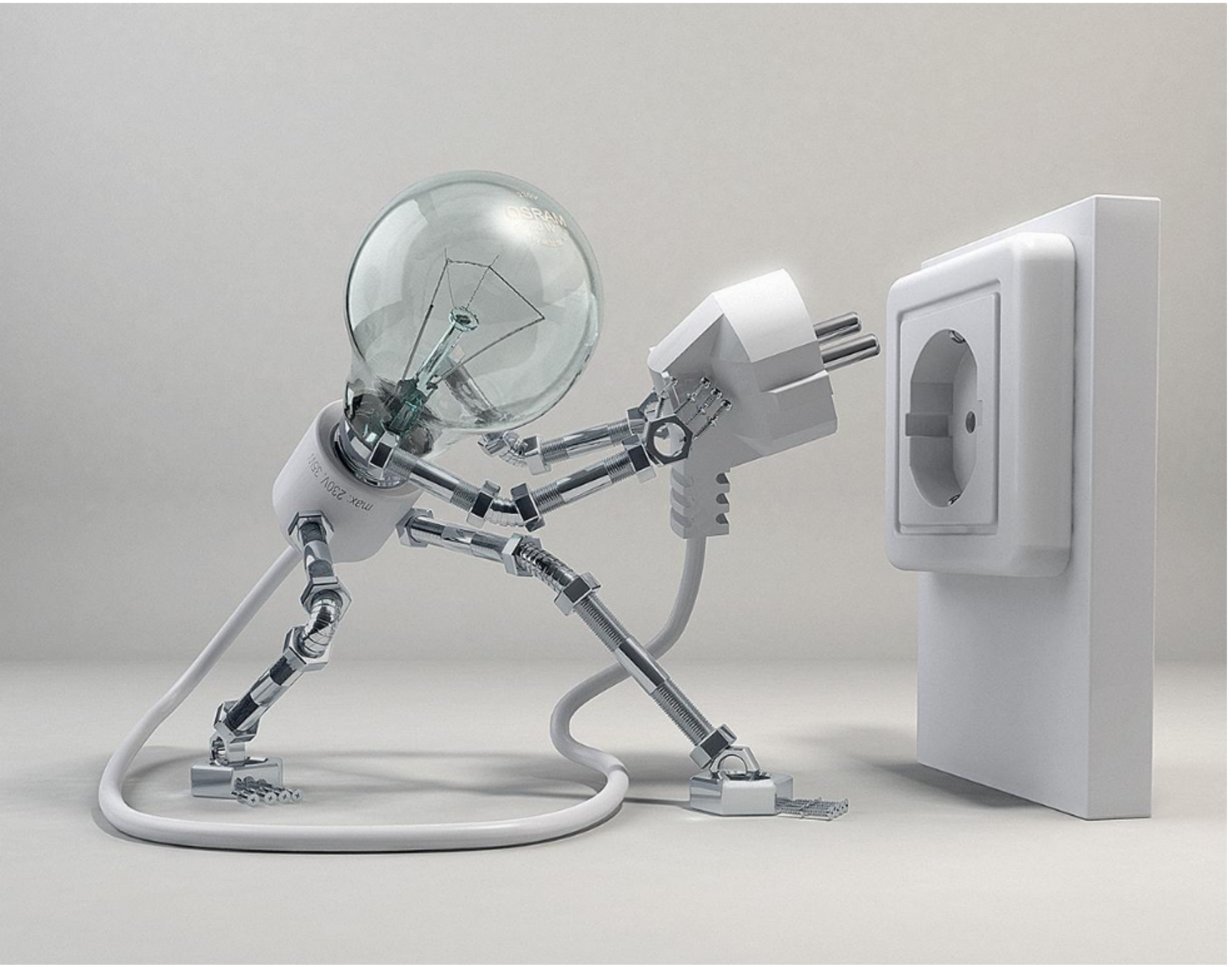
— *Typesafe*

- Acronym for “Scalable Language”.
- Object-Oriented
- Functional, Functions are objects
- Seamless Java interop

Scala: Functional Style

Characteristic	How its Handled
Focus	What information is desired, what transform is required
State Changes	Non-existent
Order of Execution	Low importance
Flow Control	Function calls, recursion
Manipulation Units	Functions are first class objects

Scala: Future



- A way to reason about many concurrent operations
- A placeholder for a result that is yet to occur
- Can be composed for sequential reasoning
- Combinators and callbacks for non-blocking
- May only be assigned once, effectively immutable

Scala: Future

Example with Callback

```
import scala.util.{ Success, Failure }

val greeting: Future[String] = future {
  session.getLastGreeting
}

...
```

Scala: Future

Example with Callback

```
import scala.util.{ Success, Failure }

val greeting: Future[String] = future {
    session.getLastGreeting
}

greeting onComplete { # <==== callback when future completes
  case Success(greet) => println("Last greeting was " + greet)
  case Failure(e)     => println("Error: " + e.getMessage)
}
```

Scala: Future

Composition with Combinators

```
val pizzaStore: Future[PizzaStore] = future {
  pizzaService.getClosestStore(zipCode)
}

...
```

Scala: Future

Composition with Combinators

```
val pizzaStore: Future[PizzaStore] = future {
  pizzaService.getClosestStore(zipCode)
}

val pizza: Future[Option[Pizza]] = pizzaStore map { # <==== produces a new future
  store => Some(pizzaService.buy(store, "pepporoni"))
} recover {
  case NonFatal(e) => None
}
```

Scala: Future

Composition with Combinators

```
val pizzaStore: Future[PizzaStore] = future {
  pizzaService.getClosestStore(zipCode)
}

val pizza: Future[Option[Pizza]] = pizzaStore map { # <==== produces a new future
  store => Some(pizzaService.buy(store, "pepporoni"))
} recover { # <==== produces a new future, if error, applies partial function
  case NonFatal(e) => None
}
```

Scala: Promise

- Promises can create a future
- Writable single-assigment container
- Completes a future with `success`
- Fails a futre with `failure`
- It's the writing side of the `Future`

Scala: Promise

```
val pss = new PizzaStoreService
val hs = new HomeService
val p = promise[Pizza]()
val f = p.future

val orderFood = future {
  val pizza = pss.orderPizza() # <==== they told me it would only be 30 minutes ;-(
  p success pizza
  hs.setTable()
}

val eat = future {
  hs.findMovie()
  f onSuccess {
    case pizza => hs.eat()
  }
}
```

Scala: Promise

```
val pss = new PizzaStoreService
val hs = new HomeService
val p = promise[Pizza]()
val f = p.future

val orderFood = future {
  val pizza = pss.orderPizza() # <==== they told me it would only be 30 minutes ;-(
  p success pizza # <==== when the pizza arrives complete the future
  hs.setTable() # <==== don't wait for the pizza, set the table in the meantime
}

val eat = future {
  hs.findMovie()
  f onSuccess {
    case pizza => hs.eat()
  }
}
```

Scala: Promise

```
val pss = new PizzaStoreService
val hs = new HomeService
val p = promise[Pizza]()
val f = p.future

val orderFood = future {
  val pizza = pss.orderPizza() # <==== they told me it would only be 30 minutes ;-(
  p success pizza # <==== when the pizza arrives complete the future
  hs.setTable() # <==== don't wait for the pizza, set the table in the meantime
}

val eat = future {
  hs.findMovie() # <==== still waiting, lets find a good movie!
  f onSuccess {
    case pizza => hs.eat()
  }
}
```

Scala: Promise

```
val pss = new PizzaStoreService
val hs = new HomeService
val p = promise[Pizza]()
val f = p.future

val orderFood = future {
  val pizza = pss.orderPizza() # <==== they told me it would only be 30 minutes ;-(
  p success pizza # <==== when the pizza arrives complete the future
  hs.setTable() # <==== don't wait for the pizza, set the table in the meantime
}

val eat = future {
  hs.findMovie() # <==== still waiting, lets find a good movie!
  f onSuccess {
    case pizza => hs.eat() # <==== Yeah! Pizza is here, lets eat!
  }
}
```

Akka

- What is Akka?
- Actor System
- Distributed Model

Akka: What is Akka?

Akka is a toolkit and runtime for building highly concurrent, distributed, and fault tolerant event-driven applications on the JVM.

— *Typesafe*

- Simple Concurrency & Distribution
- Resilient by Design
- High Performance
- Elastic & Decentralized
- Extensible

Akka: Actors

- Lightweight concurrent entities - 2.5m / GB mem
- Uses asynchronous event-driven receive loop
- Much easier to reason about concurrent code
- Focus is on **workflow** rather than **concurrency**
- Supports both Scala & Java

Akka: Actors

```
case class Pizza(kind: String)

class PizzaActor extends Actor with ActorLogging {
  def receive = {
    case Pizza(kind) ⇒ log.info("You want a " + kind + " Pizza!")
  }
}

val system = ActorSystem("MySystem")
val PizzaEater = system.actorOf(Props[PizzaActor], name = "pizzaeater")
PizzaEater ! Pizza("Pepporoni")
```

Akka: Actors

Fault Tolerance

- Supervisor hierarchies with "let-it-crash" semantics
- Supervisor hierarchies can span multiple JVM's
- Self-healing semantics
- Never stop philosophy

Fault Tolerance

- Actor's supervise actors they create
- When failure occurs the supervisor can:
  - Resume the failed actor
  - Stop or Restart the failed actor
  - Escalate the problem up the chain
- Supervisor strategy can be overridden

Fault Tolerance

```
import akka.actor.OneForOneStrategy
import akka.actor.SupervisorStrategy._
import scala.concurrent.duration._

override val supervisorStrategy =
  OneForOneStrategy(maxNrOfRetries = 5, withinTimeRange = 1 minute) {
    case _: ArithmeticException      => Resume
    case _: NullPointerException     => Restart
    case _: IllegalArgumentException => Stop
    case _: Exception                => Escalate
  }
```

Akka: Actors

Location Transparency

- Distributed workflow environment
- Purely with messages passing
- Asynchronous in nature
- Local model = distributed model
- Purely driven by configuration

Akka: Actors

Location Transparency

```
# Message sent to local actor
ActorRef localWorld = system.actorOf(
    new Props(WorldActor.class), "world");

localWorld ! "Hello!"
```

Akka: Actors

Location Transparency

```
# Message sent to remote actor
ActorRef remoteWorld = system.actorOf(
    new Props(WorldActor.class), "world");

remoteWorld ! "Hello!"
```

Akka: Actors

Location Transparency

```
ActorRef localWorld = system.actorOf(
    new Props(WorldActor.class), "world");

localWorld ! "Hello!"
   |
 # No Difference in Semantics
   |
ActorRef remoteWorld = system.actorOf(
    new Props(WorldActor.class), "world");

remoteWorld ! "Hello!"
```

Akka: Actors

Persistence

- Messages can be optionally persisted and replayed
- Actors can recover their state
  - even after JVM crashes
  - even after node migration
- Supports snapshots

Akka: Actors

Persistence

```
class ExampleProcessor extends PersistentActor {
  var state = ExampleState() # <--- mutable state, but NOT shared = OK!

  def updateState(event: Evt): Unit =
    state = state.update(event)

  ...

}
```

Akka: Actors

Persistence

```
class ExamplePersistentActor extends PersistentActor {
  ...

  val receiveRecover: Receive = { # <=== process persisted events on boostrap
    case evt: Evt                                    => updateState(evt)
    case SnapshotOffer(_, snapshot: ExampleState) => state = snapshot
  }

  ...
}
```

Akka: Actors

Persistence

```
class ExamplePersistentActor extends PersistentActor {
  ...

    val receiveCommand: Receive = { # <=== process commands, if valid persist events
    case Cmd(data) =>
      persist(Evt(s"{data}")) { event =>
        updateState(event)
        context.system.eventStream.publish(event)
      }
    ...
  }
}
```

Akka Concurrency Works

Thank You!