Integration of the Actor Model into Mainstream Tech

PHILIPP HALLER
What is Mainstream?
What is Mainstream?

ONE ANSWER:
What is Mainstream?

ONE ANSWER:

PLATFORM/EXECUTION MODEL:
- JIT compiled bytecode
- Threading based on OS processes or native POSIX threads
What is Mainstream?

ONE ANSWER:

PLATFORM/EXECUTION MODEL:
- JIT compiled bytecode
- Threading based on OS processes or native POSIX threads

STATIC TYPING
What is Mainstream?

ONE ANSWER:

PLATFORM/EXECUTION MODEL:
- JIT compiled bytecode
- Threading based on OS processes or native POSIX threads

STATIC TYPING

In our case

THE JVM + SCALA
SCALA
SCALA

What is it?
SCALA

What is it?

- OBJECT-ORIENTED
- FUNCTIONAL
- AGILE, LIGHTWEIGHT SYNTAX
- SAFE, PERFORMANT

with strong static typing
SCALA

Where does it come from?
SCALA

Where does it come from?

1996-2000
Pizza, GJ, Java generics, javac

2003-2006
The Scala “Experiment”
Who's Using Scala?

[Logos of various companies using Scala]
Scala Actors

**LONGTIME CORE CONCURRENCY LIB**
In the stdlib from early-on, (since Scala 2.1.7)

**ERLANG-LIKE**
Very close to Erlang’s actor-like processes

```scala
val shop = actor {
  while (true) {
    receive {
      case Order(item) =>
        val order = handleOrder(item, sender)
        sender ! Ack(order)
      case Cancel(order) =>
        cancelOrder(order)
        sender ! Cancelled(order)
    }
  }
}
```
SCALA ACTORS

Early Goals
SCALA ACTORS
Early Goals

- Library-based Design
  - Unclear which concurrency paradigm will “win”
  - Scalability: enable flexible concurrency libraries
SCALA ACTORS

Early Goals

LIBRARY-BASED DESIGN
- Unclear which concurrency paradigm will “win”
- Scalability: enable flexible concurrency libraries

EMBRACE THE HOST LANGUAGE
- “Competitive” programming interface
SCALA ACTORS

Early Goals

LIBRARY-BASED DESIGN
- Unclear which concurrency paradigm will “win”
- Scalability: enable flexible concurrency libraries

EMBRACE THE HOST LANGUAGE
- “Competitive” programming interface

LIGHTWEIGHT EXECUTION ENVIRONMENT
- Event-based actors much more lightweight
- Integration with JVM threads
**IDEA:** Introduce an event-based `react` operation which takes a continuation closure:

```plaintext
loop {
  react {
    case Order(item) => ... 
    case Cancel(order) => ...
  }
}
```

Actor detached from a thread while waiting to receive a message

- Scales to much larger numbers of actors
- Uses work-stealing thread pool for message processing
Integrating EVENTS & THREADS

EVENT-BASED & BLOCKING
Actors support both event-based react and blocking operations

MANAGED BLOCKING
Thread pool resizing

SEND/RECEIVE ANYWHERE
Message send and receive also available on regular, non-actor threads of the JVM

SCALA ACTORS: Experience
SCALA ACTORS: Experience

LIBRARY-BASED DESIGN WORKS WELL
SCALA ACTORS: Experience

Library-based design works well

Scalability
Through work-stealing thread pool

Proven in production!
For example, at Twitter during Obama inauguration
Scala Actors: Experience

- Library-based design works well
- Scalability: Through work-stealing thread pool
- Proven in production! For example, at Twitter during Obama inauguration
- Adoption: By many commercial users
Scala Actors: Experience

Library-based design works well

Scalability
Through work-stealing thread pool

Proven in production!
For example, at Twitter during Obama inauguration

Adoption
By many commercial users

Robust!
Only a handful of known issues even after years of low maintenance
SCALA ACTORS: Challenges
SCALA ACTORS: Challenges

Isolation

- Actors are objects => direct access to its methods/state possible unless precautions are taken
- Exchange of mutable messages by reference
SCALA ACTORS: Challenges

**ISOLATION**
- Actors are objects => direct access to its methods/state possible unless precautions are taken
- Exchange of mutable messages by reference

**FAULT TOLERANCE**
Restarting an actor is impractical, since it requires updating all references to that same logical actor in the entire system
SCALA ACTORS: Challenges

**ISOLATION**
- Actors are objects => direct access to its methods/state possible unless precautions are taken
- Exchange of mutable messages by reference

**FAULT TOLERANCE**
Restarting an actor is impractical, since it requires updating all references to that same logical actor in the entire system

**REMITING ONLY RUDIMENTARY**
SCALA ACTORS: Challenges

**ISOLATION**
- Actors are objects => direct access to its methods/state possible unless precautions are taken
- Exchange of mutable messages by reference

**FAULT TOLERANCE**
Restarting an actor is impractical, since it requires updating all references to that same logical actor in the entire system

**REMITING ONLY RUDIMENTARY**

**MESSAGE PILE-UP**
Erlang’s queue model can lead to message pile-up, linear performance degradation
ISOLATION through Uniqueness
ISOLATION through Uniqueness

Avoiding data races when exchanging mutable objects

No need for full ownership types
ISOLATION through Uniqueness

Avoiding data races when exchanging mutable objects
No need for full ownership types

FOUNDATIONS AND SOUNDNESS PROOF:

Philipp Haller, Martin Odersky. Capabilities for uniqueness and borrowing.
ECOOP 2010
**ISOLATION through Uniqueness**

Avoiding data races when exchanging mutable objects

No need for full ownership types

**FOUNDATIONS AND SOUNDNESS PROOF:**
Philipp Haller, Martin Odersky. Capabilities for uniqueness and borrowing. ECOOP 2010

**EXAMPLE:** using the prototype of a Scala compiler plug-in:

```scala
actor {
  val buf: ArrayBuffer[Int] @unique =
    new ArrayBuffer[Int](3)
  buf ++= Array(0, 1, 2)
  someActor ! buf
}

actor {
  val buf: ArrayBuffer[Int] @unique =
    new ArrayBuffer[Int](3)
  buf ++= Array(0, 1, 2)
  someActor ! buf
  println(buf.remove(0))
}
```

ok! illegal!
Requirements of Industry

EARLY GOALS NOT ENOUGH, NEED ALSO:

HIGH PERFORMANCE

EXTENSIVE REMOTING CAPABILITIES
- Support for third party remote transports
- Flexible configuration

PRAGMATIC SOLUTIONS TO CHALLENGES

SHORT RELEASE CYCLES
- Until 2.10.0 only infrequent releases of Scala distribution
Enter: AKKA
AKKA: Actors Reloaded

Main Differences:

- Distinction between actors and `ActorRefs` to avoid direct access to actor instances
- Actor-global event loop replaces blocking-style `react`
- Unhandled messages not kept in mailbox
AKKA: Actors Reloaded

Benefits

- Simpler implementation
- Higher performance
- Simplified fault-tolerance (actor restarts made easy)
- ActorRefs enable transparent remoting
AKKA’s Actor API

Similar to scala.actors API

Example:

class Shop extends Actor {
  def receive = {
    case Order(item) =>
      val order = handleOrder(item, sender)
      sender ! Ack(order)
    case Cancel(order) =>
      cancelOrder(order)
      sender ! Cancelled(order)
  }
}

val shop: ActorRef = system.actorOf(Props[Shop])
Partial Functions

**Block with Pattern Matching Cases**

**Type Definitions:**

```scala
trait Function1[-A, +B] {
  def apply(x: A): B
}

trait PartialFunction[-A, +B]
  extends Function1[A, B] {
    def isDefinedAt(x: A): Boolean
    ...
  }
```
**receive** returns global message handler

- handler activated when message can be removed from mailbox
- will never leave a message in the mailbox
- if no pattern matches removed message, an event is published to the enclosing container ("actor system"), signaling an unhandled message
- works well with case class instances: matching on receiver’s side
- use of partial functions as message handlers as well as case classes for message types introduced by Scala Actors
Sending Messages

Like Scala Actors, Akka adopts the principal message send operator from Erlang:
- \texttt{a ! msg} asynchronously sends \texttt{msg} to \texttt{a}

Other constructs adopted from Scala Actors:
- \texttt{a forward msg, sender ! msg}
- \texttt{a ? msg} asynchronously sends \texttt{msg} to \texttt{a} and immediately returns a future (\texttt{a !! msg} in \texttt{scala.actors})

A future is a placeholder for a response that may eventually be received
scala.concurrent.

FUTURE
&PROMISE
Futures & Promises

Can be thought of as a single concurrency model.

Diagram:

- Future
- Promise

Sunday, October 21, 12
Futures & Promises

Can be thought of as a single concurrency model

- **READ-MANY**
- **WRITE-ONCE**

**FUTURE**

**PROMISE**
Futures & Promises

CAN BE THOUGHT OF AS A SINGLE CONCURRENCY MODEL

IMPORTANT OPS

✔ Start async computation
✔ Wait for result
✔ Assign result value
✔ Obtain associated future object
A promise $p$ of type `Promise[T]` can be completed in two ways...

**Success**

```scala
val result: T = ...
p.success(result)
```

**Failure**

```scala
val exc = new Exception("something went wrong")
p.failure(exc)
```
Async & Non Blocking
Async & Non-Blocking

**GOAL:** Do not block current thread while waiting for result of future
Async & NonBlocking

**GOAL:** Do not block current thread while waiting for result of future

Callbacks

- **REGISTER CALLBACK** which is invoked (asynchronously) when future is completed
- **ASYNC COMPUTATIONS NEVER BLOCK** (except for managed blocking)
Async & Non-Blocking

**GOAL:** Do not block current thread while waiting for result of future

Callbacks

- **REGISTER CALLBACK** which is invoked (asynchronously) when future is completed
- **ASYNC COMPUTATIONS NEVER BLOCK** (except for managed blocking)

**USER DOESN’T HAVE TO EXPLICITLY MANAGE CALLBACKS. HIGHER-ORDER FUNCTIONS INSTEAD!**
Futures & Promises

Example

Thread 1  Thread 2  Thread 3
Futures & Promises

EXAMPLE

val p = Promise[Int]() // Thread 1

(Thread 1, Thread 2, Thread 3)
val p = Promise[Int]() // Thread 1
val f = p.future // Thread 1
val p = Promise[Int]() // Thread 1
val f = p.future // Thread 1
f onSuccess { // Thread 2
  case x: Int => println("Successful!")
}

hread1 Thread2 Thread3

val p = Promise[Int]() // Thread 1
val f = p.future // Thread 1
f onSuccess {
  case x: Int => println("Successful!")
}
val p = Promise[Int]() // Thread 1
val f = p.future // Thread 1
f onSuccess {
  case x: Int => println("Successful!")
}
p.success(42) // Thread 1

(val create promise)
(val get reference to future)
(val register callback)
(val write to promise)
Futures & Promises

**EXAMPLE**

```scala
val p = Promise[Int]() // Thread 1
val f = p.future // Thread 1
f onSuccess {
  // Thread 2
  case x: Int => println("Successful!")
}
p.success(42) // Thread 1
```

**NOTE:** onSuccess CALLBACK EXECUTED EVEN IF f HAS ALREADY BEEN COMPLETED AT TIME OF REGISTRATION
Combinators

COMPOSABILITY THRU HIGHER-ORDER FUNCs

STANDARD MONADIC COMBINATORS

```scala
def map[S](f: T => S): Future[S]

val purchase: Future[Int] = rateQuote map {
  quote => connection.buy(amount, quote)
}

def filter(pred: T => Boolean): Future[T]

val postBySmith: Future[Post] =
post.filter(_.author == "Smith")
```
Combinators

COMPOSABILITY THRU HIGHER-ORDER FUNCS

STANDARD MONADIC COMBINATORS

def map[S](f: T => S): Future[S]

val purchase: Future[Int] = rateQuote map {
  quote => connection.buy(amount, quote)
}

IF MAP FAILS: purchase is completed with unhandled exception

def filter(pred: T => Boolean): Future[T]

val postBySmith: Future[Post] =
  post.filter(_.author == "Smith")

IF FILTER FAILS: postBySmith completed with NoSuchElementException
Future
THE IMPLEMENTATION

Many operations implemented in terms of promises

SIMPLIFIED EXAMPLE

def map[S](f: T => S): Future[S] = {
  val p = Promise[S]()

  onComplete {
    case result =>
      try {
        result match {
          case Success(r) => p success f(r)
          case Failure(t) => p failure t
        }
      } catch {
        case t: Throwable => p failure t
      }
  }
  p.future
}
The real implementation (a) adds an implicit `ExecutionContext`, (b) avoids extra object creations, and (c) catches only non-fatal exceptions:

```scala
def map[S](f: T => S)(implicit executor: ExecutionContext): Future[S] = {
val p = Promise[S]()

onComplete {
  case result =>
    try {
      result match {
        case Success(r) => p success f(r)
        case f: Failure[_] => p complete f.asInstanceOf[Failure[S]]
      }
    } catch {
      case NonFatal(t) => p failure t
    }
}

p.future
```
scala.concurrent.

EXECUTION CONTEXT
Threadpools... are needed by:

- **FUTURES** for executing callbacks and function arguments
- **ACTORS** for executing message handlers, scheduled tasks, etc.
- **PARALLEL COLLECTIONS** for executing data-parallel operations
Scala 2.10 introduces EXECUTION CONTEXTS
Scala 2.10 introduces

EXECUTION CONTEXTS

Goal

PROVIDE GLOBAL THREADPOOL AS
PLATFORM SERVICE TO BE SHARED BY
ALL PARALLEL FRAMEWORKS
Scala 2.10 introduces EXECUTION CONTEXTS

Goal

PROVIDE GLOBAL THREADPOOL AS PLATFORM SERVICE TO BE SHARED BY ALL PARALLEL FRAMEWORKS

scala.concurrent package provides global ExecutionContext

Default ExecutionContext backed by the most recent fork join pool (collaboration with Doug Lea, SUNY Oswego)
Implicit Execution Ctxs

Asynchronous computations are executed on an ExecutionContext which is provided implicitly.

```scala
def map[S](f: T => S)(implicit executor: ExecutionContext): Future[S]
def onSuccess[U](pf: PartialFunction[T, U]) (implicit executor: ExecutionContext): Unit
```

Implicit parameters enable fine-grained selection of the ExecutionContext:

```scala
implicit val context: ExecutionContext = customExecutionContext
val fut2 = fut1.filter(pred)
       .map(fun)
```
Implicit Execution Ctxs

**IMPLICIT** ExecutionContexts **ALLOW SHARING ECS BETWEEN FRAMEWORKS**

```scala
def map[S](f: T => S)(implicit executor: ExecutionContext): Future[S]
def onSuccess[U](pf: PartialFunction[T, U])
  (implicit executor: ExecutionContext): Unit
```

**ENABLES FLEXIBLE SELECTION OF EXECUTION POLICY**

```scala
implicit val context: ExecutionContext = customExecutionContext
val fut2 = fut1.filter(pred)
  .map(fun)
```
ThreadPoolExecutor

Throughput (msg/s) vs. number of actors

msg/s
1500000
1200000
900000
600000
300000
0

# actors
0 4 8 12 16 20 24

- thread-pool
ForkJoinPool

Throughput (msg/s) vs. number of actors

msg/s
20,000,000
17,500,000
15,000,000
12,500,000
10,000,000
7,500,000
5,000,000
2,500,000

# actors
0 16 32 48 64 80 96 112 128

fork-join
thread-pool
After some tweaks... 

+MILLIONS MESSAGES PER SECOND!
What about Fault Tolerance?
Akka embraces...

LET IT CRASH

FAULT TOLERANCE
// from within an actor
val child = context.actorOf(Props[MyActor], "A")

TRANSPARENT AND AUTOMATIC FAULT HANDLING BY DESIGN.
Actors can form hierarchies...
Actors can form hierarchies...

Guardian System Actor

```
system.actorOf(Props[Greeter], "Greeter")
```
ACTORS can form hierarchies...

Guardian System Actor

system.actorOf(Props[Greeter], "Greeter")
Actors can form hierarchies...

Guardian System Actor

Greeter

context.actorOf(Props[A], "A")
Actors can form hierarchies...

Greeter

Guardian System Actor

context.actorOf(Props[A], “A”)
Actors can form hierarchies...

Guardian System Actor

- Greeter
  - A
    - B
    - D
  - C
    - E
- Curser
  - A
    - B
    - C
NAME RESOLUTION
like a file system...

Guardian System Actor

Greeter

Curser

A

B

C

D

E

A

B

C

Sunday, October 21, 12
NAME RESOLUTION
like a file system...

Guardian System Actor

/Greeter

- Greeter
  - A
    - B
    - D
  - C
    - E

- Curser
  - A
    - B
    - C
NAME RESOLUTION
like a file system...

Guardian System Actor

/Greeter

/Greeter/A

A

B

D

E

B

A

C

Sunday, October 21, 12
NAME RESOLUTION
like a file system...

Guardian System Actor

/Greeter
/Greeter/A
/Greeter/A/B
NAME RESOLUTION like a file system...

Guardian System Actor

/Greeter

/Greeter/A

/Greeter/A/B

/Greeter/A/D

A

B

D

E

C

A

B

C
Find Actors

```scala
val actorRef = system.actorFor("/user/Greeter/A")
val parent = context.actorFor("..")
val sibling = context.actorFor("../B")
val selection = system.actorSelection("/user/Greeter/*")
```
ERLANG-style Actors

In Scala Actors, Erlang-style receive/react is the default

**Issues**

- Implementation more expensive than Akka’s global message handler
- Queue model can lead to message pile-up

**But...**

- Most real-world actor programs written in Erlang (probably)
- Erlang style can simplify complex messaging protocols

AKKA 2.0 INTRODUCES A Stash TRAIT FOR THIS
Erlang-style **react** of Scala actors makes it easy to express certain messaging protocols through nested **reacts**:

```scala
actor {
  react {
    case "open" =>
      var done = false
      loopWhile (!done) {
        react {
          case "read" => ...  
          case "close" => done = true
        }
      }
  }
}
```
Using the "stash" to model the previous example using an Akka actor’s global event loop:

```scala
class ActorWithProtocol extends Actor with Stash {
  def receive = {
    case "open" => unstashAll()
    context.become {
      case "read" => // do reading...
      case "close" => unstashAll(); context.unbecome()
      case msg => stash()
    }
    case msg => stash()
  }
}
```
CONCLUSION

SCALA IS A GROWABLE LANGUAGE
invaluable for establishing actors as one of its principle concurrency models

EMBRACING UNIQUE SCALA FEATURES
supports adoption in Scala community (but can provide Java API)

TIGHT INTEGRATION
with execution environment ensures scalability and high performance

FIND OUT MORE
Akka: http://akka.io Futures in Scala 2.10: http://docs.scala-lang.org
The Typesafe Stack: http://www.typesafe.com/stack/
QUESTIONS?