Programming Reactive Systems in Scala: Principles and Abstractions

Philipp Haller

KTH Royal Institute of Technology
Stockholm, Sweden

Entwicklertag Frankfurt, Germany, 21 February, 2018
What are reactive systems?

- Multiple definitions proposed previously, e.g. by Gérard Berry [1] and by the Reactive Manifesto [2]

- Common among definitions: reactive systems

  - react to events or messages from their environment

  - react (typically) "at a speed which is determined by the environment, not the program itself" [1]

- Thus, reactive systems are:

  - responsive

  - scalable
What makes it so difficult to build reactive systems?

1. Workloads require massive scalability

- Steam, a digital distribution service, delivers 16.9 PB per week to users in Germany (USA: 46.9 PB) [3]
- CERN amassed about 200 PB of data from over 800 trillion collisions looking for the Higgs boson. [4]
- Twitter has about 330 million monthly active users [5]
Steam delivers **16.9 PB per week** to users in Germany (USA: 46.9 PB) [3]
What makes it so difficult to build reactive systems?

1. Workloads require massive scalability
   • Steam, a digital distribution service, delivers \textbf{16.9 PB per week} to users in Germany (USA: 46.9 PB) [3]
   • CERN amassed about \textbf{200 PB of data} from over 800 trillion collisions looking for the Higgs boson. [4]
   • Twitter has about \textbf{330 million monthly active users} [5]

2. Reacting \textbf{at the speed of the environment} (guaranteed timely responses)
Example: Twitter during Obama's inauguration

“We saw 5x normal tweets-per-second and about 4x tweets-per-minute as this chart illustrates.” [6]
Implications

- Massive scalability $\rightarrow$ large-scale distribution
- Timely responses + distribution $\rightarrow$ resiliency

"To make a fault-tolerant system you need at least two computers." - Joe Armstrong [7]
How to program reactive systems?

Want to build systems responding to events emitted by their environment in a way that enables scalability, distribution, and resiliency

• We're looking for *programming abstractions*!

• How did we approach this *in the Scala project*?
Example

• Chat service
  • Many long-lived connections
  • Usually idle, with short bursts of traffic
Chat service: first try

- Thread per user session
- **Huge overheads stemming from heavyweight threads**
- **Does not scale to large numbers of users**
Chat service: second try

• Asynchronous I/O and thread pool
• Session state maintained in regular objects (e.g., POJOs)
• Much more scalable
• Problems:
  • **Code difficult to maintain** ➞ "callback hell" [8]
  • **Blocking calls fatal**
The trouble with **blocking ops**

**Example**

Function for creating a Future that is completed with `value` after `delay` milliseconds

```scala
def after[T](delay: Long, value: T): Future[T]
```
"after", version 1

def after1[T](delay: Long, value: T) =
  Future {
    Thread.sleep(delay)
    value
  }
"after", version 1

How does it behave?

```scala
assert(Runtime.getRuntime()
        .availableProcessors() == 8)

for (_ <- 1 to 8) yield
  after1(1000, true)

val later = after1(1000, true)
```

Quiz: when is “later” completed?

Answer: after either ~1 s or ~2 s (most often)
Promises

object Promise {
    def apply[T](): Promise[T]
}

trait Promise[T] {
    def success(value: T): Promise[T]
    def failure(cause: Throwable): Promise[T]

    def future: Future[T]
}
"after", version 2

```scala
def after2[T](delay: Long, value: T) = {
  val promise = Promise[T]()

  timer.schedule(new TimerTask {
    def run(): Unit = promise.success(value)
  }, delay)

  promise.future
}
```

Much better behaved!
Chat service example

• Neither of the shown approaches is satisfactory
  • Thread-based approach induces huge overheads, does not scale
  • Event-driven approach suffers from callback hell and blocking operations are troublesome

We need better programming abstractions which *reconcile scalability and productivity*
Better programming abstractions

• At the end of 2005, our main influence was the *Erlang* programming language

• One of *very few success stories* in the concurrent programming

• Had been used successfully to build the influential *Ericsson AXD301* switch providing an availability of nine nines

• ... and there was a really great movie about Erlang [9] ;-)

• Additional influences, including *Argus* [10], the *join-calculus* [11], and other seminal languages and systems
**Erlang and the actor model**

- Erlang: a dynamic, functional, distributed, concurrency-oriented programming language
- Provides an implementation of the *actor model of concurrency* \[12\]
- Actors = concurrent "processes" communicating via message passing
  - No shared state
  - Senders decoupled from receivers ➞ *asynchronous messaging*
- Upon receiving a message, an actor may
  - change its behavior/state
  - send messages to actors (including itself)
  - create new actors

Sender does not fail if receiver fails!
Actors in Scala (using Akka)

Definition of an actor class:

```scala
class Counter extends Actor with ActorLogging {
  var sum = 0

  def receive = {
    case AddAll(values) =>
      sum += values.reduce((x, y) => x + y)
    case PrintSum() =>
      log.info(s"the sum is: $sum")
  }
}
```

case class AddAll(values: Array[Int])
case class PrintSum()
Client of an actor

Creating and using an actor:

```scala
object Main {
  def main(args: Array[String]): Unit = {
    val system = ActorSystem("system")
    val counter: ActorRef = system.actorOf(Counter.props, "counter")
    counter ! AddAll(Array(1, 2, 3))
    counter ! AddAll(Array(4, 5))
    counter ! PrintSum()
  }
}

object Counter {
  def props: Props = Props(new Counter)
}
```

Asynchronous message sends

Actor creation properties
Actors: important features

- Actors are **isolated**
- Field `sum` not accessible from outside
- Ensured by exposing only an `ActorRef` to clients
  - `ActorRef` provides an **extremely simple interface**
- Messages in actor's mailbox are **processed sequentially**
- No concurrency control necessary within an actor
- Messaging is **location-transparent**
- `ActorRefs` may be remote; can be sent in messages
Resiliency using actors

• Erlang's approach to fault handling: "let it crash!"

• **Do not:**
  • try to avoid failure
  • attempt to repair program state/data in case of failure

• **Do:**
  • let faulty actors crash
  • manage crashed actors via *supervision*
Actor supervision: strategy 1
Actor supervision: strategy 2
Actor supervision: strategy 3
Resiliency (continued)

How to restart a fresh actor from some previous state?

• Supervisor initializes its state, or

• Fresh actor obtains initial state from elsewhere, or

• Fresh actor replays received messages from persistent log

  ➟ event sourcing: Akka Persistence
Actors in Scala

• Q: *Is all of this built into Scala?*

• A: *Not quite.*
Defining a receive method returns a `partial function` defined by the block of cases `{ ... }`
Deconstructing actors

```scala
object Actor {
  // Type alias for receive blocks
  type Receive = PartialFunction[Any, Unit]

  // ...
}

trait Actor {

  def receive: Actor.Receive

  // ...
}
```
Partial functions

- Partial functions have a type `PartialFunction[A, B]`
- `PartialFunction[A, B]` is a subtype of `Function1[A, B]`

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

trait PartialFunction[A, B] extends Function1[A, B] {
  def isDefinedAt(x: A): Boolean
  ..
}
```
Pattern matching

```scala
{ case AddAll(values) =>
    sum += values.reduce((x, y) => x + y)
  case PrintSum() =>
    log.info(s"the sum is: $sum")
}
```

The case clauses are just regular pattern matching in Scala:

```scala
val opt: Option[Int] = thisgetOption()
opt match {
  case Some(x) => // full optional object
    // use `x` of type `Int`
  case None =>    // empty optional object
    // no value available
}
```
The `!` operator is just a method written using infix syntax:

```scala
abstract class ActorRef extends .. { 
  def !(message: Any): Unit
  // ..
}
```

"Aha! Built-in support for messaging!!"
Summary

• Actors not built into Scala
  • Rely only on shared-memory threads of the JVM
• Scala as a "growable" language [13]
• Programming models as libraries
  • Akka actors = domain-specific language (DSL) embedded in Scala
  • Many of the patterns and techniques first implemented in Scala Actors [14]
Philipp Haller creates Scala Actors (the original standard library actors). His work becomes a major influence to Akka and the main reason for Jonas Bonér to choose Scala as the platform for Akka.

First commit: https://github.com/scala/scala/commit/0d8b14c6055e76c0b93b65d0f428d711abe1f5a

Philipp Haller publishes his influential paper on Scala's Actors, "Actors that Unify Threads and Events".
http://infoscience.epfl.ch/record/104549

Jonas gets seriously interested in Erlang.

Jonas starts tinkering with Scala Actors.

https://www.lightbend.com/akka-five-year-anniversary
There is more

• Q: *Actors are clearly awesome! All problems solved?*

• A: *Not quite.*
Example

Image processing pipeline:

1. Filter 1
2. Filter 2

Pipeline stages run concurrently
Implementation

• Assumptions:
  • Image data large
  • Main memory expensive

• Approach for high performance:
  • *In-place update* of image buffers
  • Pass mutable buffers *by-reference*
Problem

Easy to produce data races:

1. Stage 1 sends a reference to a buffer to stage 2
2. Following the send, both stages have a reference to the same buffer
3. Stages can concurrently access the buffer
Preventing data races

• Approach: safe transfer of ownership
  • Sending stage loses ownership
  • Compiler prevents sender from accessing objects that have been transferred

• Advantages:
  • No run-time overhead
    • Safety does not compromise performance
  • Errors caught at compile time
Ownership transfer in Scala

• Active research project: *LaCasa* [15]
• LaCasa: Scala extension for *affine references*
  • "Transferable" references
• At most one owner per transferable reference
Affine references in LaCasa

- LaCasa provides affine references by combining two concepts:
  - Access permissions
  - Encapsulated boxes
Access permissions

- Access to transferable objects controlled by implicit permissions

  \[
  \text{CanAccess} \{ \text{type} \ C \}
  \]

- Type member C uniquely identifies box

  \[
  \text{Box}[T] \{ \text{type} \ C \}\]
Creating boxes and permissions

```scala
sealed trait Packed[+T] {
  val box: Box[T]
  val access: CanAccess { type C = box.C }
}

implicit val access = packed.access
val box = packed.box
```

```scala
mkBox[Message] { packed =>
  implicit val access = packed.access
  val box = packed.box
  ...
  LaCasa library
  ...
```
Accessing boxes

- Boxes are encapsulated
- Boxes must be opened for access

```scala
mkBox[Message] { packed =>
  implicit val access = packed.access
  val box = packed.box

  box.open { msg =>
    msg.arr = Array(1, 2, 3, 4)
  }
}
```

Requires implicit access permission
Consuming permissions

**Example:** transferring a box from one actor to another consumes its access permission

```scala
mkBox[Message] { packed =>
  implicit val access = packed.access
  val box = packed.box

  ...
  someActor.send(box) {
    // make `access` unavailable
    ...
  }
}
```

Leverage *spores* [1]
Encapsulation

Problem: not all types safe to transfer!

class Message {
    var arr: Array[Int] = _
    def leak(): Unit = {
        SomeObject.fld = arr
    }
}

object SomeObject {
    var fld: Array[Int] = _
}
Encapsulation

• Ensuring absence of data races requires restricting types put into boxes

• Requirements for “safe” classes:
  • Methods only access parameters and this
  • Method parameter types are “safe”
  • Methods only instantiate “safe” classes
  • Types of fields are “safe”

“Safe” = conforms to object capability model [17]
Object capabilities in Scala

• How common is object-capability safe code in Scala?

• Empirical study of over 75,000 SLOC of open-source Scala code:

<table>
<thead>
<tr>
<th>Project</th>
<th>Version</th>
<th>SLOC</th>
<th>GitHub stats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scala stdlib</td>
<td>2.11.7</td>
<td>33,107</td>
<td>✭5,795</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>★257</td>
</tr>
<tr>
<td>Signal/Collect</td>
<td>8.0.6</td>
<td>10,159</td>
<td>✭123</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>★11</td>
</tr>
<tr>
<td>GeoTrellis</td>
<td>0.10.0-RC2</td>
<td>35,351</td>
<td>★400</td>
</tr>
<tr>
<td>-engine</td>
<td></td>
<td>3,868</td>
<td>★38</td>
</tr>
<tr>
<td>-raster</td>
<td></td>
<td>22,291</td>
<td></td>
</tr>
<tr>
<td>-spark</td>
<td></td>
<td>9,192</td>
<td></td>
</tr>
</tbody>
</table>
# Object capabilities in Scala

## Results of empirical study:

<table>
<thead>
<tr>
<th>Project</th>
<th>#classes/traits</th>
<th>#ocap (%)</th>
<th>#dir. insec. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scala stdlib</td>
<td>1,505</td>
<td>644 (43%)</td>
<td>212/861 (25%)</td>
</tr>
<tr>
<td>Signal/Collect</td>
<td>236</td>
<td>159 (67%)</td>
<td>60/77 (78%)</td>
</tr>
<tr>
<td>GeoTrellis</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-engine</td>
<td>190</td>
<td>40 (21%)</td>
<td>124/150 (83%)</td>
</tr>
<tr>
<td>-raster</td>
<td>670</td>
<td>233 (35%)</td>
<td>325/437 (74%)</td>
</tr>
<tr>
<td>-spark</td>
<td>326</td>
<td>101 (31%)</td>
<td>167/225 (74%)</td>
</tr>
<tr>
<td>Total</td>
<td>2,927</td>
<td>1,177 (40%)</td>
<td>888/1,750 (51%)</td>
</tr>
</tbody>
</table>

Immutability inference increases these percentages!
Ongoing work

- Flow-sensitive type checking
  - "Don't indent when consuming permission"
- Empirical studies
  - How much effort to change existing code?
- Language support for immutable types [18]
- Complete mechanization in Coq proof assistant
Conclusion

• **Scala enables powerful libraries for reactive programming**
  
  • Akka actors representative example
  
  • There are many others: Akka Streams, Spark Streaming, REScala [19] etc.

• Not all concurrency hazards can be prevented by Scala's current type system.

• In ongoing research projects, such as LaCasa and Reactive Async [20], we are exploring ways to rule out *data races* and *non-determinism*
References (1)

- [2]: https://www.reactivemanifesto.org/
- [3]: http://store.steampowered.com/stats/content/
- [6]: https://blog.twitter.com/2009/inauguration-day-twitter
- [8]: http://static.usenix.org/publications/library/proceedings/usenix02/full_papers/adyahowell/adyahowell_html/
- [9]: https://www.youtube.com/watch?v=uKfKtXYLg78
References (2)


- [15]: https://github.com/phaller/lacasa


- [18]: https://www.youtube.com/watch?v=IiCt4nZfQfg

- [19]: http://guidosalva.github.io/REScala/

- [20]: https://github.com/phaller/reactive-async