Scala Actors Library

Robert Hilbrich

Foreword and Disclaimer

I am not going to teach you Scala.

However, I want to:

- Introduce a library
- Explain what I use it for

My Goal is to:

- Give you a basic idea about Scala Actors
- Tell you about ist underlying thread magic

Agenda

1. Introduction
2. Scala Actors Library
3. Good Actors Style
4. Actors and Threads
5. Actors in My Life

References

Fraunhofer
FIRST

1 INTRODUCTION

Why should you care?

Fraunhofer
FIRST

Introduction

About me: SPES Project → Multicore / Manycore Processors

Multicore Programming Today:

- Manually deal with Threads
- Often Shared Memory approaches
- Synchronization is hard, slow and error prone
- „Thread"-level = = „Assembler"-level (?)

Future Challenges:

- Manycore → Rise in Complexity
- Want: Less sharing, less synchronization
- Want: Message Passing instead of Shared Memory
- Want: Abstraction of Threads

Tilera: 100 Cores (2010)

2 SCALA ACTORS LIBRARY

An introduction and brief overview

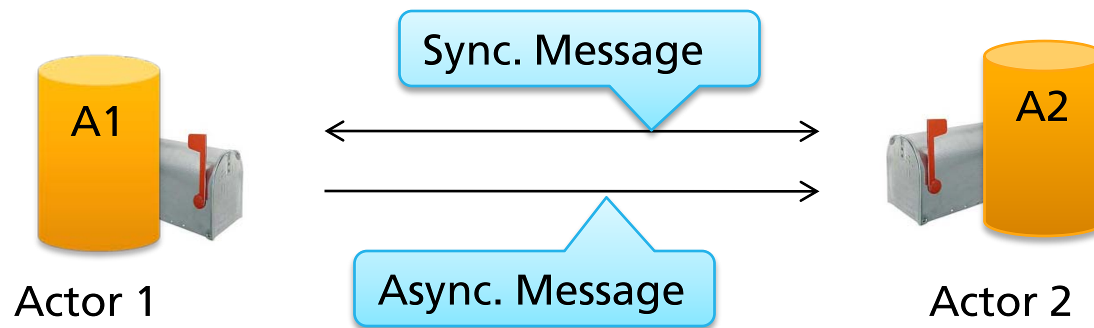
Actors and Concurrency

Classical Java Thread Programming

- Shared data and locks

Scala Actors Library

- Simplify JVM thread programming
- Uses flexibility of Scala to create an abstraction layer that looks „native“
- „Share-nothing“ and „Message Passing“ approach (no locks!)
- Add some runtime „Thread-Magic“

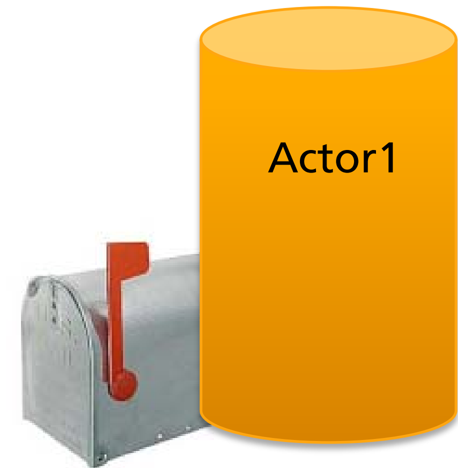


Fraunhofer
FIRST

A simple actor

```
import scala.actors._
object actor1 extends Actor {
  def act() {
      println(„I am acting“)
  }
}
```

```
scala> actor1.start()
I am acting
scala>
```



Fraunhofer FIRST

Message Passing: an echo actor

```
import scala.actors._
object echoActor extends Actor {
   def act() {
      receive {
         case msg => println(msg)
      }
   }
}
```

```
scala> echoActor.start()
scala> echoActor ! "Hallo FIRST"
Hallo FIRST
scala>
```

Fraunhofer FIRST

Details about Message Passing

– Sending an asynchronous message is done via "!"

```
actor ! msg
```

– Sending a synchronous message is done via "!?"

```
actor !? msg
```

– Retrieving a single message from the mailbox (blocks!)

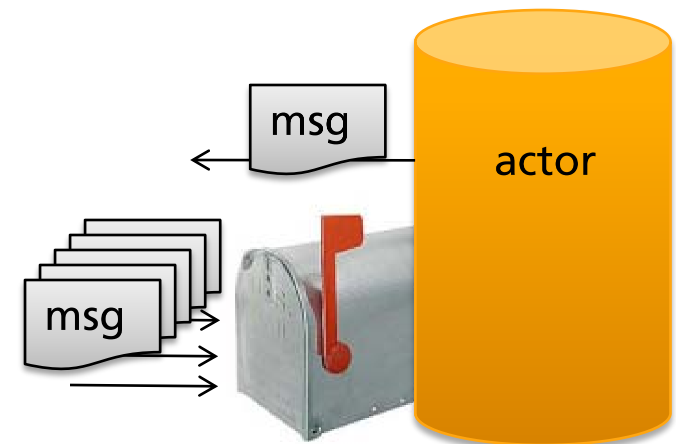
```
receive { … }
```

– Filter messages (returns the first message that matches)

```
receive { case (msg: Type) => … }
```

– Reply to a synchronous message with `reply`

```
receive { case (req) => reply(rep) }
```

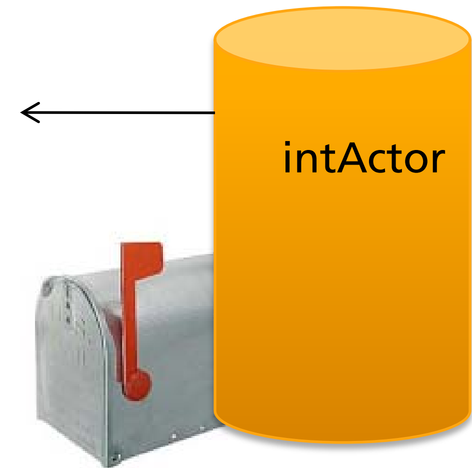


Fraunhofer
FIRST

Message Passing: async with case types example

```
object intActor extends Actor {
   def act() {
      receive { case x:Int => println(x)
      }
   }
}
```

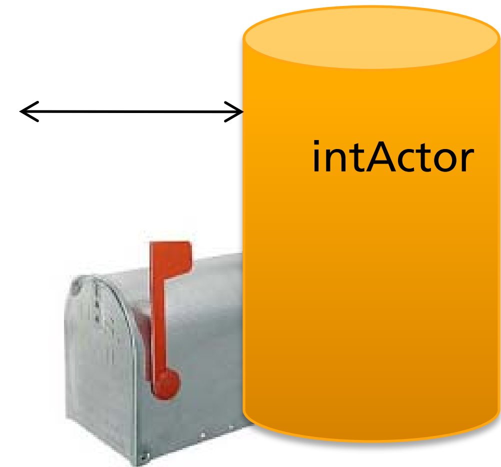
```
scala> intActor.start()
scala> intActor ! "hello"
scala> intActor ! Math.Pi
scala> intActor ! 12
12
scala>
```



Fraunhofer
FIRST

Message Passing: sync example

```
object intActor extends Actor {
   def act() {
      receive { case x:Int => reply(x*x)
      }
   }
}
```




```
scala> intActor.start()
scala> intActor !? 12
144
scala>
```

← Blocks!

Background: Actors model

- "Actors" is a generic computational model for concurrent and distributed computations
- Has (first?) been implemented in Erlang – "processes"
  - Erlang was developed at Ericson
  - Often used at Telco providers, especially for reliable (!) line switching

- Its goodness has been proven in many Real-Time Control Systems

- Until now: no similar abstraction to Threads has been available for popular virtual machines

3 GOOD ACTORS STYLE

Some ideas about good programming practice with actors

Style Tips

- Actors should not block
  - … because it may block the processing of another message
  - … instead of `Thread.sleep()` create a seperate sleepActor
- Communicate only via immutable message
  - … going back to shared memory and locks requires thinking!
  - … message objects may be shared between different actors
    → thread safe implementation required
- Make messages self-contained
  - "Fire-and-Forget" demands for redundant information in messages

4 ACTORS AND THREADS

An actor is not just a thread!

Mapping Actors to Threads

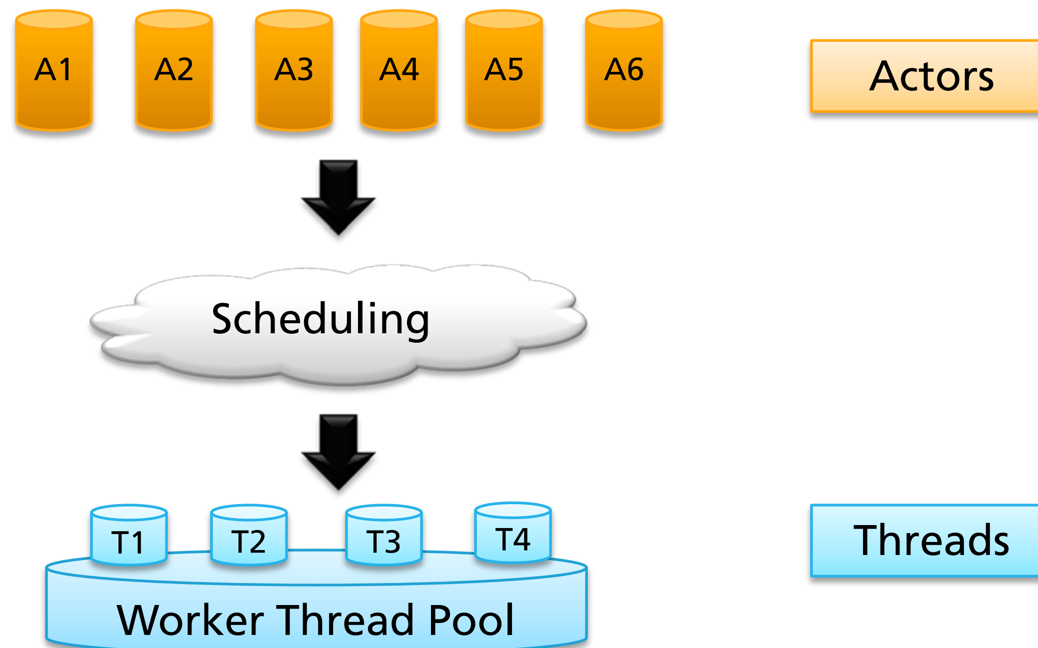
- Until now: Actors only as concurrency abstraction to Java Threads
  - N receiving Actors → N Java Threads
  - Heavy weight → exhaustion of virtual address space
  - Context switch is expensive (spawn 500'000'000 Actors for a computation?!)

- Event-driven programming models circumvent this overhead
- But:
  - "Inversion of control" – register a handler for an event
  - Fragmentation of logic
  - Control flow is implicitly expressed by modifying shared state

- Design goal for Actors: make them thread-less

Actors and Threads

- Actors are implemented as lightweight event objects
- Scheduled and executed on an underlying worker thread pool
- Worker Thread Pool gets automatically resized



Actors and Suspension

Actors can thus be used thread-based and event-based!

Two Actor suspension mechanisms exist:

- Via `receive` → thread-based
- Via `react` → execution is "piggy-backed" onto the senders thread

```
object echoActor extends Actor
{
  def act() {
   react {
        case msg => println(msg)
   }
}
```

Why?

- With `react` all actors could be implemented with a single worker thread
- Control flow is explicitely expressed
- Reduced overhead for context switches

Fraunhofer FIRST

More on `React`

- A wait on `react` is represented by a *continuation closure* (= a closure capturing the rest of the actor's computation)
- Closure is executed by the senders thread once a matching message arrived
- When closure terminates:
  - control is returned to the sender
- When closure blocks again (another `react`):
  - control is returned to the sender (→ special exception)
  - unwinding of receivers call stack

Fraunhofer
FIRST

React Example I

```
object echoActor extends Actor
{
  def act() {
    react {
      case msg =>
                  println(msg)
                  act()
    }
}
```


Fraunhofer
FIRST

React Example II

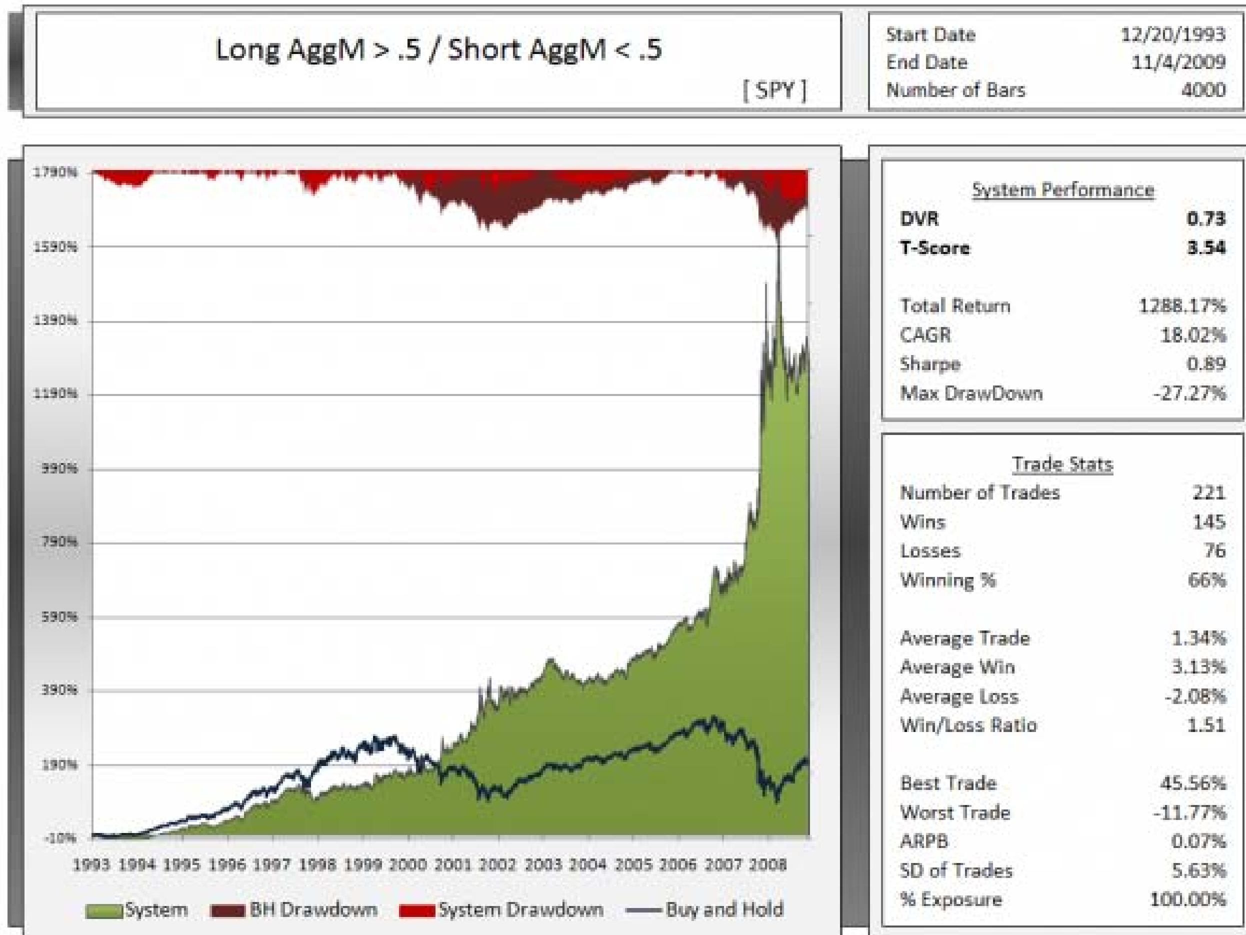
```
object echoActor extends Actor
{
  def act() {
    loop {
      react {
        case msg => println(msg)
      }
    }
}
```


Fraunhofer
FIRST

5 ACTORS IN MY LIFE

Why did I stumble upon actors?

Trading and financial speculation

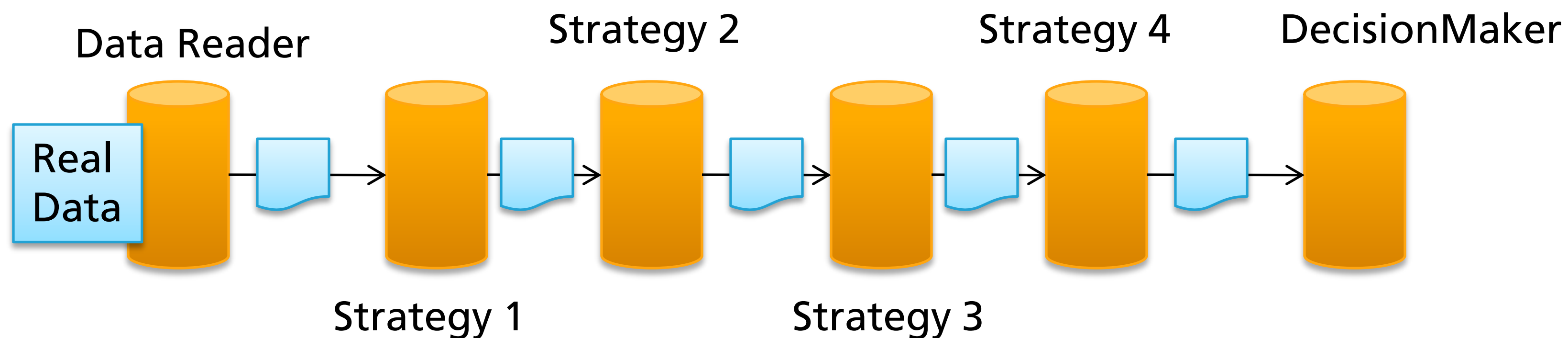


Fraunhofer FIRST

Implementing a Strategy Framework

- Ultimate goal: get rich! ☺ … (at least die trying!)

- Immediate goal: a framework to study different investment strategies
- Underlying model: data pipeline
- Each actor receives stock data and decisions from the previous actor
- Some actors may have no memory, others may memorize decisions of other actors

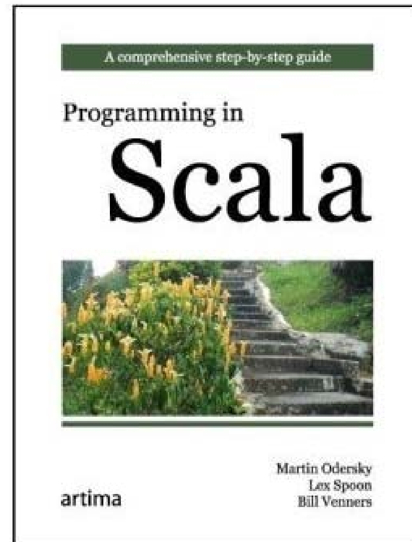


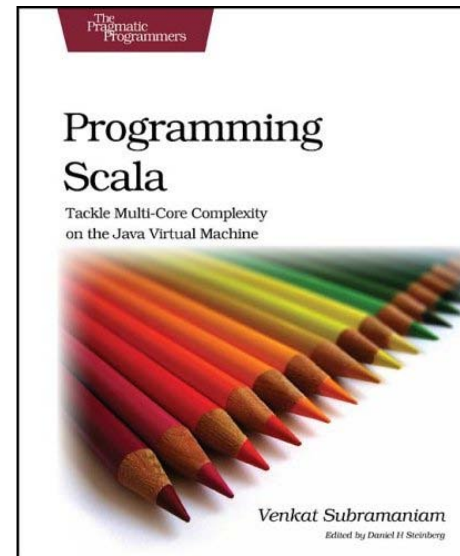
Fraunhofer FIRST

REFERENCES

Where to look for more?

References





– "**Event-Based Programming without Inversion of Control**", P. Haller and M. Odersky, in *Proceedings JMLC 2006*.

– "**Actors that Unify Threads and Events**", P. Haller and M. Odersky, in *Proceedings COORDINATION 2007*.

– Overview: **http://java.dzone.com/articles/scala-threadless-concurrent**