Modello ad Attori e Concorrenza

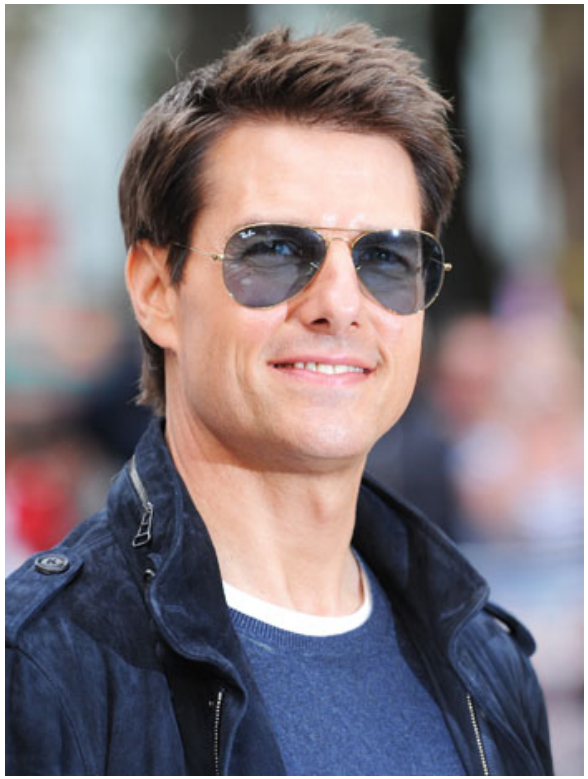
Mirko Bez, Matteo Lisotto, Tobia Tesan

7 Novembre 2016

Università degli Studi di Padova

Overview

Cos'è un attore?



Modello ad attori

- Nasce nel campo dell'IA negli anni '70
  - 1973: A Universal Modular ACTOR Formalism for Artificial Intelligence" (Carl Hewitt, Peter Bishop, Richard Steiger)
  - 1986: Actors: a model of concurrent computation in distributed systems (Gul Agha)
- Come:
  - framework per lo studio teorico della concorrenza
  - base per implementazioni pratiche di sistemi concorrenti
- Implementato in: Erlang, Scala, F#...

In teoria...

Hewitt et al. (1973)

Intuitively, an ACTOR is an active agent which plays a role on cue according to a script.

Hewitt (2010)

Actor are the foundamental of computation that **embodies three foundamental concepts of computation***: processing, storage, communication [...] An actor can create more actors, send messages to other actors and designate the behaviour to be applied to the next message.*

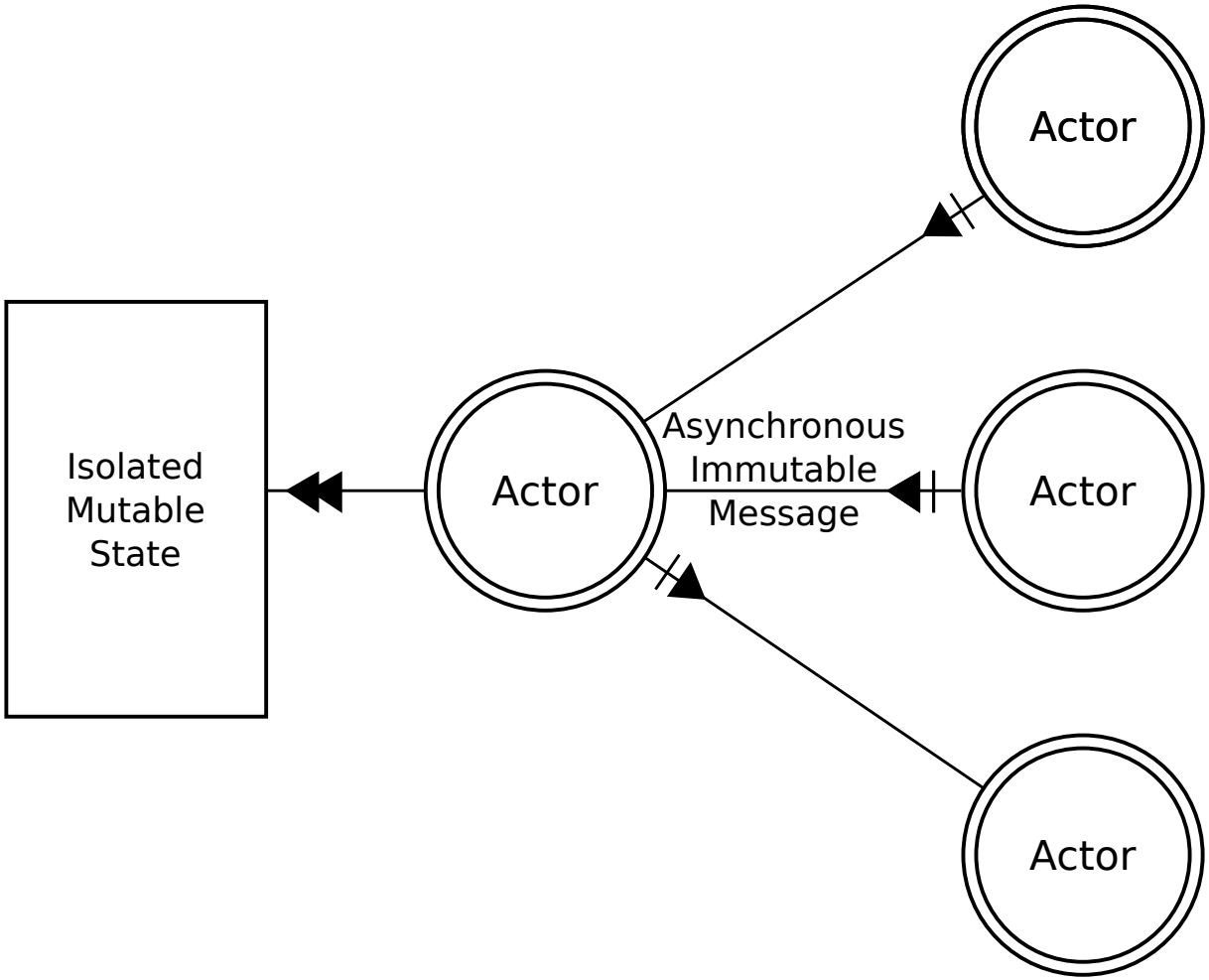
...e in pratica


```
actor Greeter
    loop n do
        receive
            (:greet, name) ⇒ print ("Hello " + name)
            (:praise, name) ⇒ print (name + "rocks")
            (:count) ⇒ print ("Greeted" + n + "people")
            (:stop) ⇒ die()
        end
        loop (n+1)
    end
end
```

...e in pratica (2)

- Loop
- Manda e riceve messaggi
  - Asincronia
  - Indeterminismo
- Pattern matching per selezionare alternative
- Pure functional (in Hewitt)
  - No global state

...e in pratica (3)



Principi e caratteristiche del modello ad attori

No global state

Hewitt et al. (1973)

Global state considered harmful

Hewitt @ Lang.NEXT 2012

Here's a sense in which Dijkstra got it completely wrong: for an Actor point of view GOTO is completely innocent! In the actor model it is a parameterless procedure call, pure and simple. The **assignment command**, *now that is harmful; because it's extremely* **expensive**!

No global state (2)

No global state $\Rightarrow$ ...

... stato sempre consistente all'interno del corpo!

- Il corpo è purely functional in Hewitt
  - o comunque eseguito in modo deterministico
- Nell'esempio *n* ha *un* valore per la durata dell'esecuzione del corpo

Asincronia e indeterminismo (1)

- La ricezione dei messaggi è **asincrona**, **non bloccante**
  - CSP (Hoare 1978) è esplicitamente sincrono
- I messaggi possono arrivare in qualsiasi ordine

Asincronia e indeterminismo (2)

Hewitt et al. (1973)

Messages can arrive in any order.

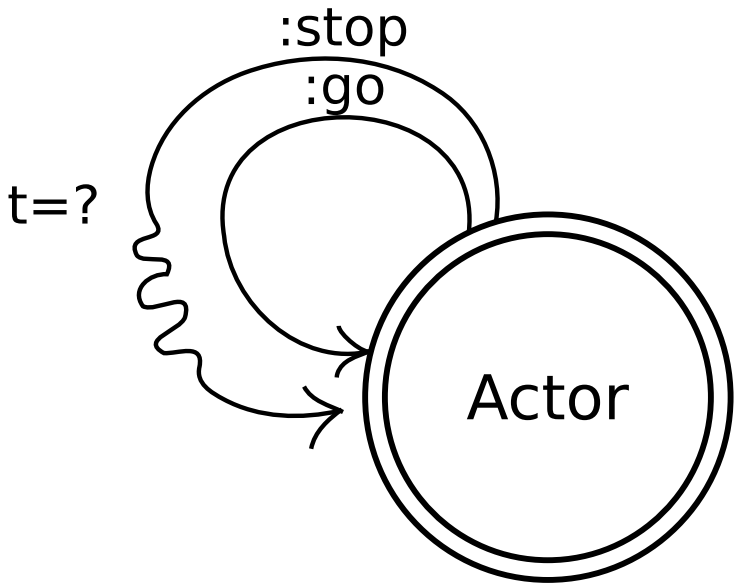
Hewitt @ Lang.NEXT 2012

We want to distinguish between **non-determinism** *and* **indeterminism**. *Nondeterminism is what we got with the original TM: it is when you flip a coin, heads or tails comes out and you go one way or another. Indeterminism is what happens when things work themselves out.*

Asincronia e indeterminismo (3)


```
actor Example
    loop count do
        receive
            (:go) ⇒ send (self, :go)
            (:stop) ⇒ die()
        end
        loop (count+1)
    end
end
```

Asincronia e indeterminismo (4)



n attori *m* indirizzi

Hewitt @ Lang.NEXT 2012

There is a many-to-many relationship between actors and addresses.

Hewitt @ Lang.NEXT 2012

Address $\neq$ *identity*

Hoare (1978)

My design insists that every input or output command must name its source or destination explicitly

Conseguenze per la concorrenza

Ma allora...

Non ci sono lock e primitive di sincronizzazione!

Ma allora...

Deadlock e deadlock detection

Agha (1986)

There is **no syntactic (or low-level) deadlock possible** *in any actor system, in the sense of it being impossible to communicate*

Agha (1986)

Since in the actor model every actor must specify a replacement, and mail addresses may be communicated, **it is possible to detect** *[logical]* **deadlock**.

Deadlock e deadlock detection

Agha (1986)

At the same time, actors **avoid sequential bottlenecks** *caused by* **assignments** *to a store. The concept of a store, in the context of parallel processing has been the nemesis of the von Neuman architectures.*

Esplosione combinatoria

Agha (1986)

The open and interactive nature of actors implies that any description of actor behavior will necessarily involve a **combinatorial explosion** *of the exact configurations possible in a system. However, by* **establishing invariants** *in the behavior of an actor, we can satisfy our self as to its correctness. The importance of* **proving correctness** *in concurrent systems is underscored by the fact that it is not possible to adequately test such systems in practice. In particular,* **arrival order nondeterminism** *implies that any particular sequence of message delivery need never be repeated regardless of the number of tests carried out.*

Ricapitolando

- No lock
- No low-level deadlock
- Più facile deadlock detection
- No bottleneck per il parallelismo
- Esplosione combinatoria

Svantaggi

Not composable

"Actors are not a good concurrency model", Paul Chiusano

So what makes actors not composable? Well, an actor (and an Erlang process) is constructed from a function A $=\rangle$ Unit. That this function returns Unit rather than a meaningful type means that it must hardcode some action to take once the result (whose type is not even shown) is available. It is this unfortunate property that destroys any chance we have at combining actors in any sort of general way like we would in a typical combinator library.

No inheritance or hierarchy

"Actors are not a good concurrency model", Paul Chiusano

The actor model has **no direct notion of inheritance or hierarchy**, *which means it is time consuming and confusing to program actors with trends of common behaviour.*

Hard to analyze

"Why has the actor model not succeeded", Paul Mackay

Another problem is behaviour replacement.
The behaviour is **dynamic**, *and very difficult to perform using a static language (e.g. C).*
Static analysis *cannot support reflection, or reconfiguration of the runtime system.*

Esempio: filosofi a cena (1)


```
actor Philospher
    loop (name, left_fork, right_fork) do
        print (name + "is thinking")
        sleep()
        print (name + "is hungry")
        send(waiter, (:waiting, self(), left_fork, right_fork)) // Sit
        receive
            (:served)⇒
                // grab forks
                send(waiter,
                (:eating, (self(), left_fork, right_fork)), // Start eating
                print (name + "is eating")
        end
        sleep()
        // release forks
        send(waiter, (:finished))
        end
    loop(name, left_fork, right_fork)
    end
end
```

Esempio: filosofi a cena (2)


```
actor Waiter
    loop (waiting_list, client_count, eating_count, busy)
        receive
            (:waiting, client) ⇒
                if (/*forks available*/)
                    send(client, :served)
                else
                    // add to waiting list
                    // let waiting_list', busy' be updated
                waiter(waiting_list', client_count, eating_count, busy');
            (:eating, Client) ⇒
                // let waiting_list' = waiting_list - client
                waiter(waiting_list', client_count, eating_count+1, false);
            (:finished) ⇒
                if (/*forks available for queued client*/)
                    send (queued_client, :served)
                waiter(waiting_list, client_count, eating_count-1, busy');
            (:leaving) ⇒
                waiter(waiting_list, client_count-1, eating_count, busy)
        end
    end
end
```

Il modello ad attori in Erlang: processi, link...

Erlang

Creato nel 1986 da Joe Armstrong per Ericsson
Open source nel 1998

- Funzionale (non puro)
- Concorrente
- Distribuito
- Fault-tolerant
- Soft Real-time
- Highly available
- Hot swapping

Actors

In Erlang un attore è un processo

- Ha un suo stato
- Una **sola** mailbox
- Comunica tramite messaggi
- Si può collegare ad un altro processo (link)
- Termina tramite
  - Messaggio di terminazione
  - Crash

Da: `http://learnyousomeerlang.com`

Let it Crash!

Erlang adotta la filosofia “Let It Crash”
Quando un processo fallisce

- Si lascia terminare
- Si esegue un restart

Let it Crash!

Processi chiamati Supervisors

- Monitorano i processi collegati
- Notificano l'avvenuto crash di un processo
- Riavviano i processi

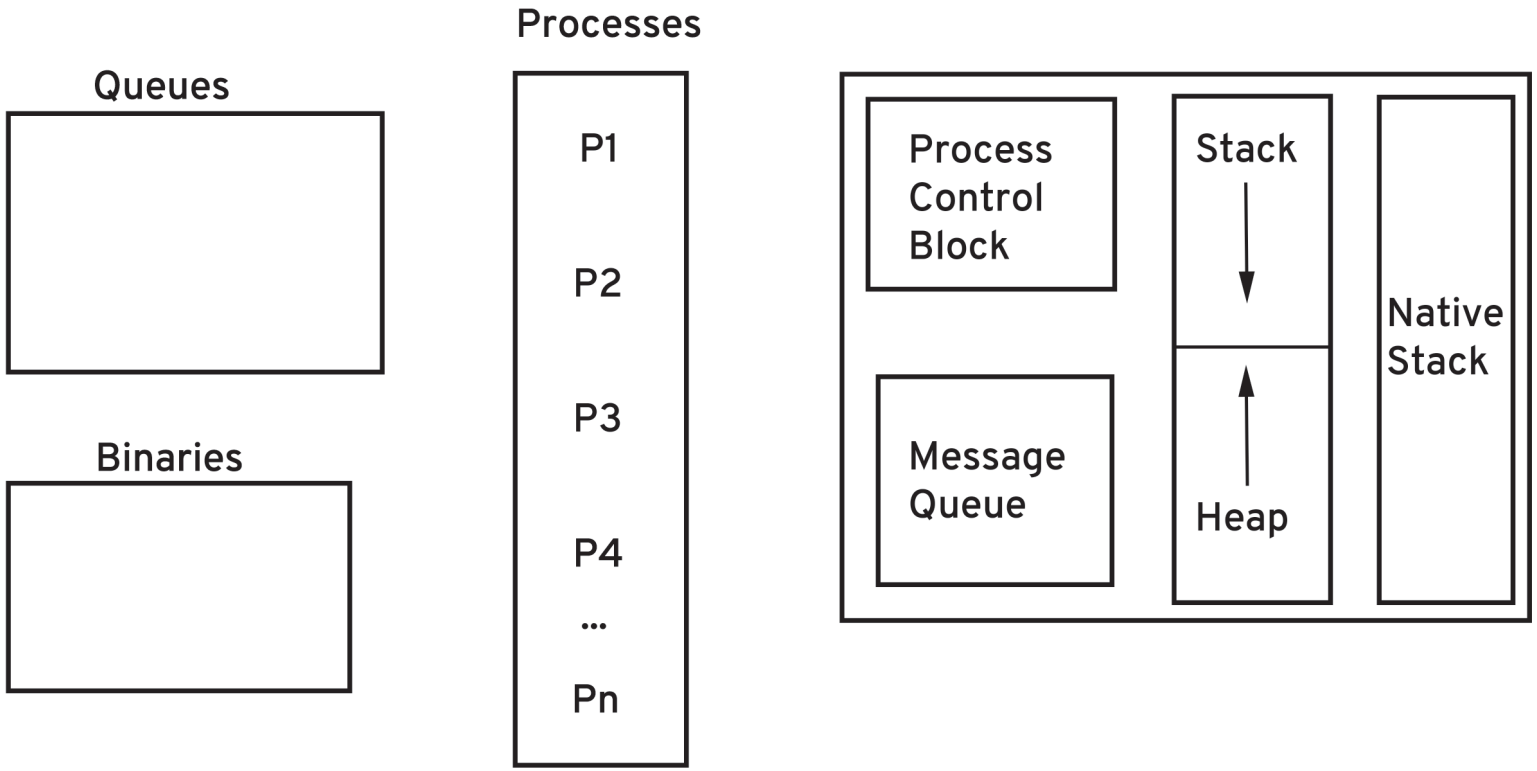
BEAM: La Virtual Machine di Erlang

Il processo all'interno di BEAM

- Un processo è uno spazio in memoria all'interno di BEAM. **Non sono OS-thread**
- N processi in M thread
- Ogni core ha un unico thread (grazie a SMTP)

Il costo di creazione e terminazione di un processo è minimo

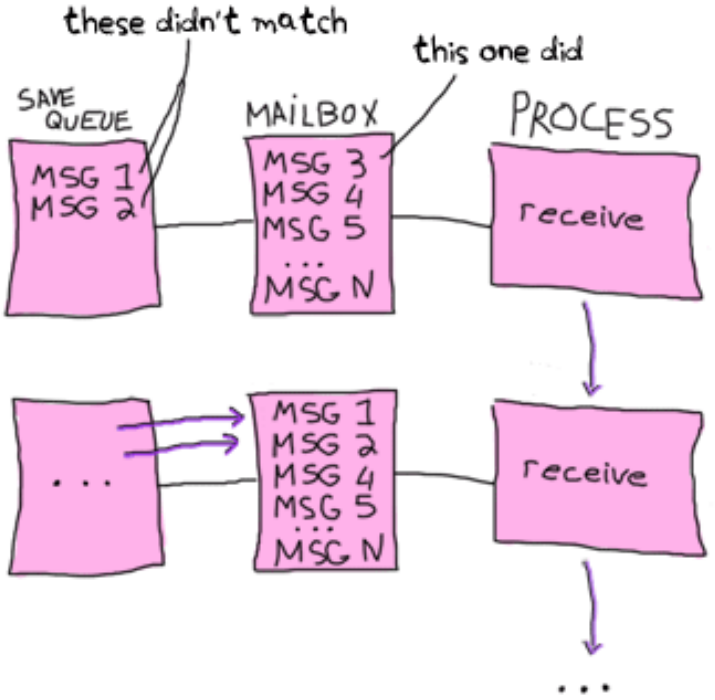
Il processo all'interno di BEAM



Un processo appena creato pesa **2.5KB** contro i **1024KB** di Java

Mailbox

- Realizzata con una coda FIFO
- Pattern matching sulle condizioni della receive
- Messaggi inviati vengono copiati nella coda del processo destinatario
- **Non vengono utilizzati puntatori**



Da: `http://learnyousomeerlang.com`

Scheduler

- Pre-emptive
- Ogni processo ha un reduction budget
- Ogni core ha uno scheduler
- Massimo 1024 scheduler per VM
- Soft Real-time system

Scheduler

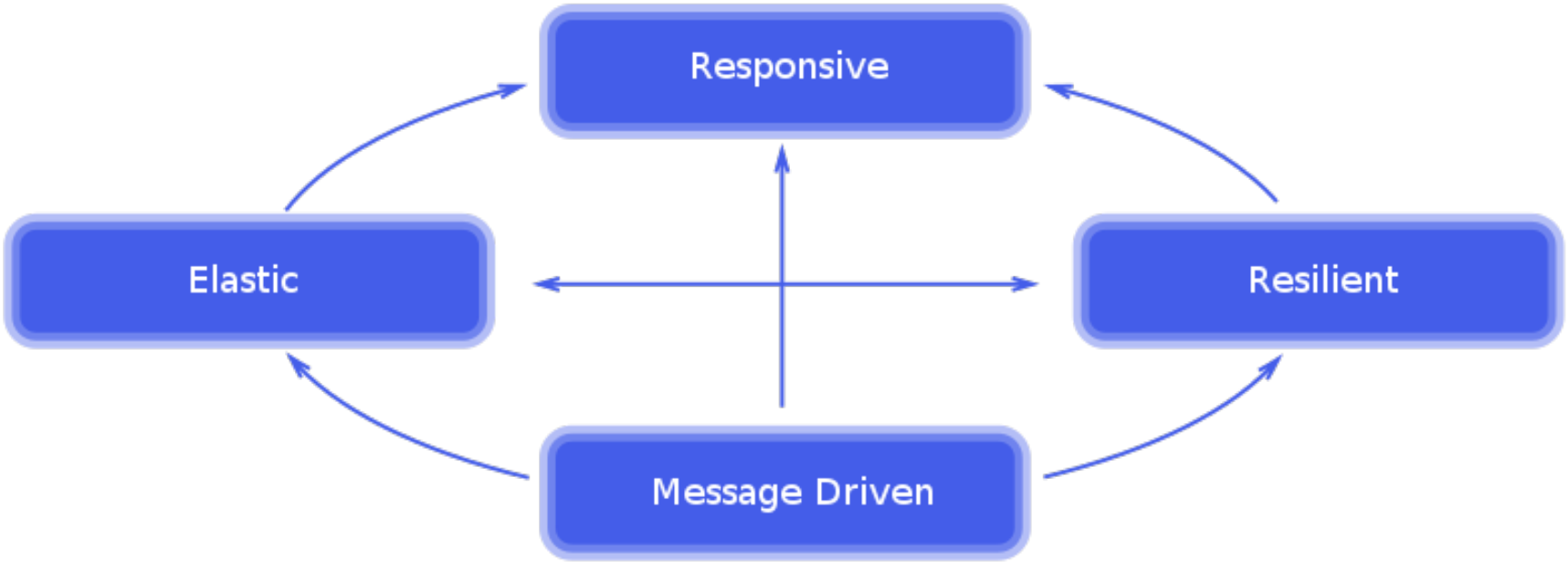
Ogni processo

- Ha 2000 reduction point (1ms)
- Ogni operazione del e sul processo ha un costo
- Lascia lo stato di esecuzione
  - Al termine dei reduction point
  - All'esecuzione di una receive

Il modello ad attori in Akka

- "Akka is a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM"
- Una libreria scritta in Scala, compatibile anche con Java, che implementa il modello ad attori
- Prende ispirazione da Erlang
- Creato da Jonas Bonèr primo firmatario del Reactive Manifesto

Reactive Manifesto



Da: `http://www.reactivemanifesto.org/`

Concettualmente

Un Attore è un container per:

- **State**
- **Behavior**
- **Mailbox**
- **Children**
- **Supervision Strategy**

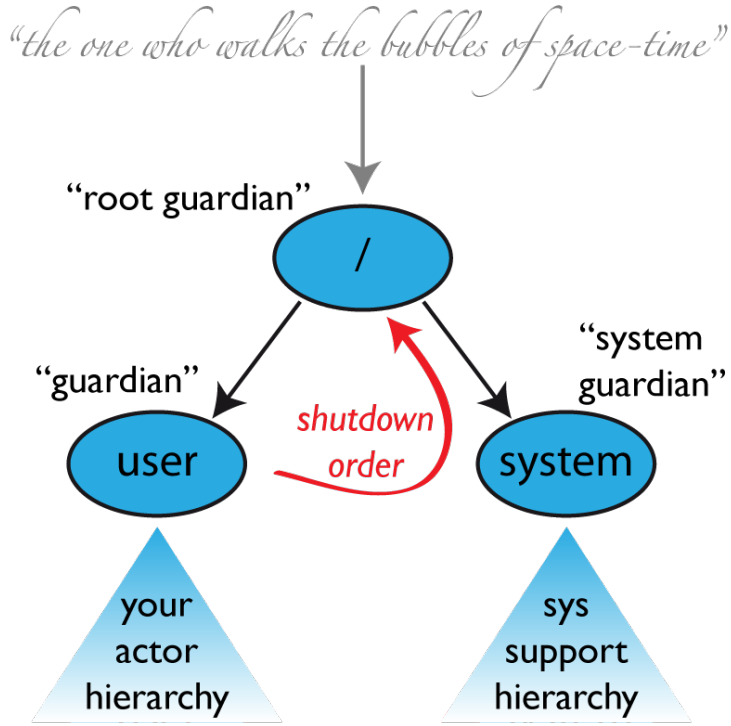
Tutto questo è incapsulato in una Actor Reference (`ActorRef`)

Concretamente

Un Attore è una classe che estende la classe astratta `Actor` che deve sovrascrivere il metodo `receive()`

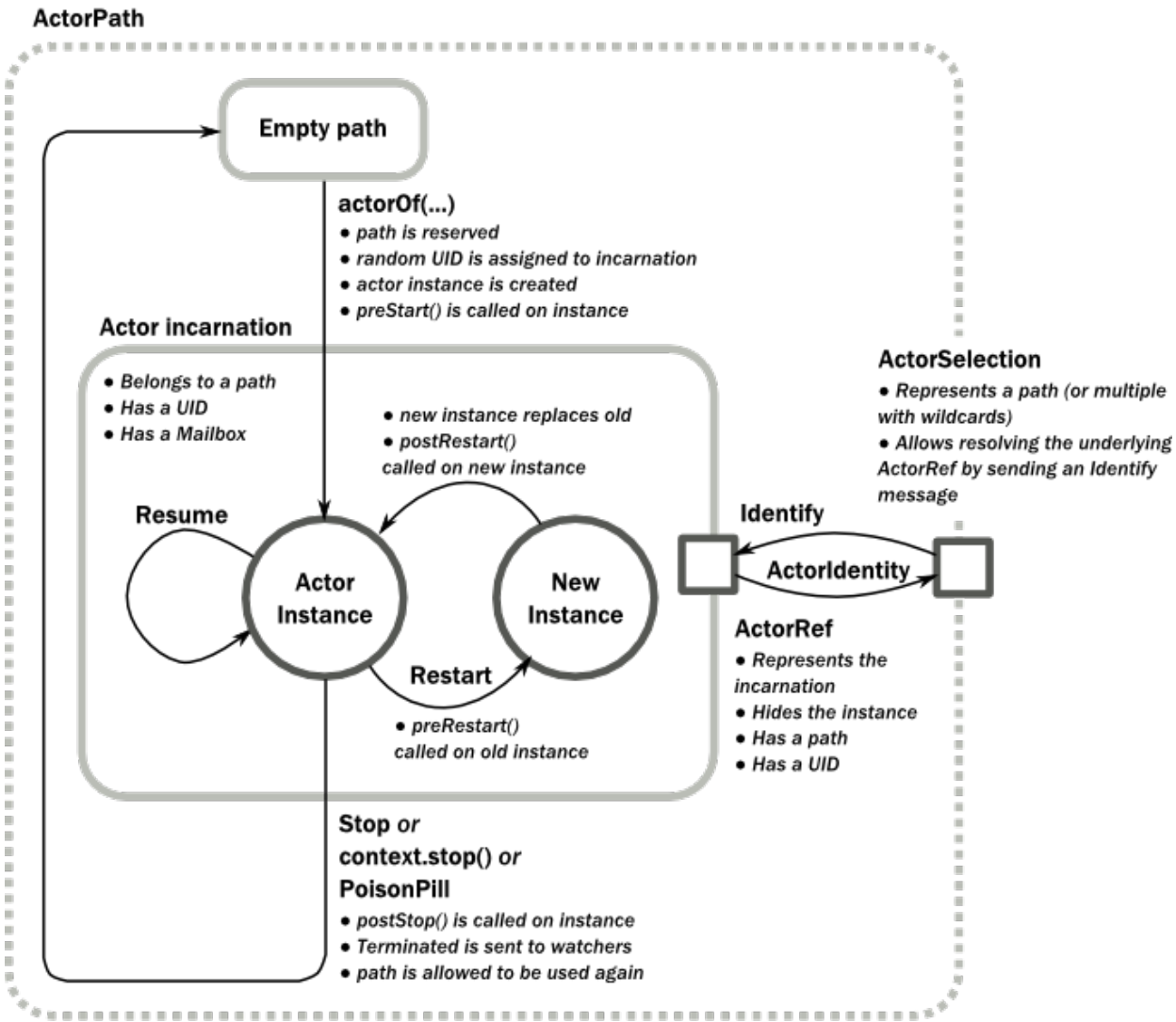
Actor System

L'Actor System è un container di Actor con una struttura ad albero



Da: `http://doc.akka.io/docs/akka/current/general/supervision.html`

Actor's Lifecycle



Da: `http://doc.akka.io/docs/akka/snapshot/scala/actors.html`

Come Termina un attore?

- Un attore sospende la propria mailbox e invia un messaggio di stop a tutti i propri figli, aspetta che questi abbiano terminato e infine termina anche lui.
- Il programmatore deve stabilire quando stoppare l'attore ma i suoi figli vengono terminati trasparentemente da Akka
- Quando un attore termina libera le risorse e la sua mailbox viene rimpiazzata da una mailbox di sistema, che userà la stessa `ActorRef` (utile per il debug)

Consistenza dello Stato (1)

Concettualmente

Ogni Attore ha il proprio light-weight thread (ca. 300 Bytes + stato) che è protetto dal resto del sistema.

Sotto il cofano

- Akka esegue insiemi di attori su insiemi di thread. Piú attori condividono un thread e successive chiamate di un attore possono essere processati da thread differenti.
- Akka assicura che le singole istanze degli attori non possano essere eseguite da più di un thread contemporaneamente (quindi lo stato sarà sempre consistente)

Consistenza dello Stato (2)

- Gli attori hanno accesso **solo** all'`ActorRef` di un altro attore
- L'ActorRef è immutabile ed ha una corrispondenza univoca con l'attore che rappresenta
- L'utilizzo di queste referenze (immutabili) previene che le interazioni tra gli attori avvengano su basi diverse rispetto al message passing.
- Altrimenti si romperebbe il disaccoppiamento dell'actor model

Message Passing Asincrono - Generale

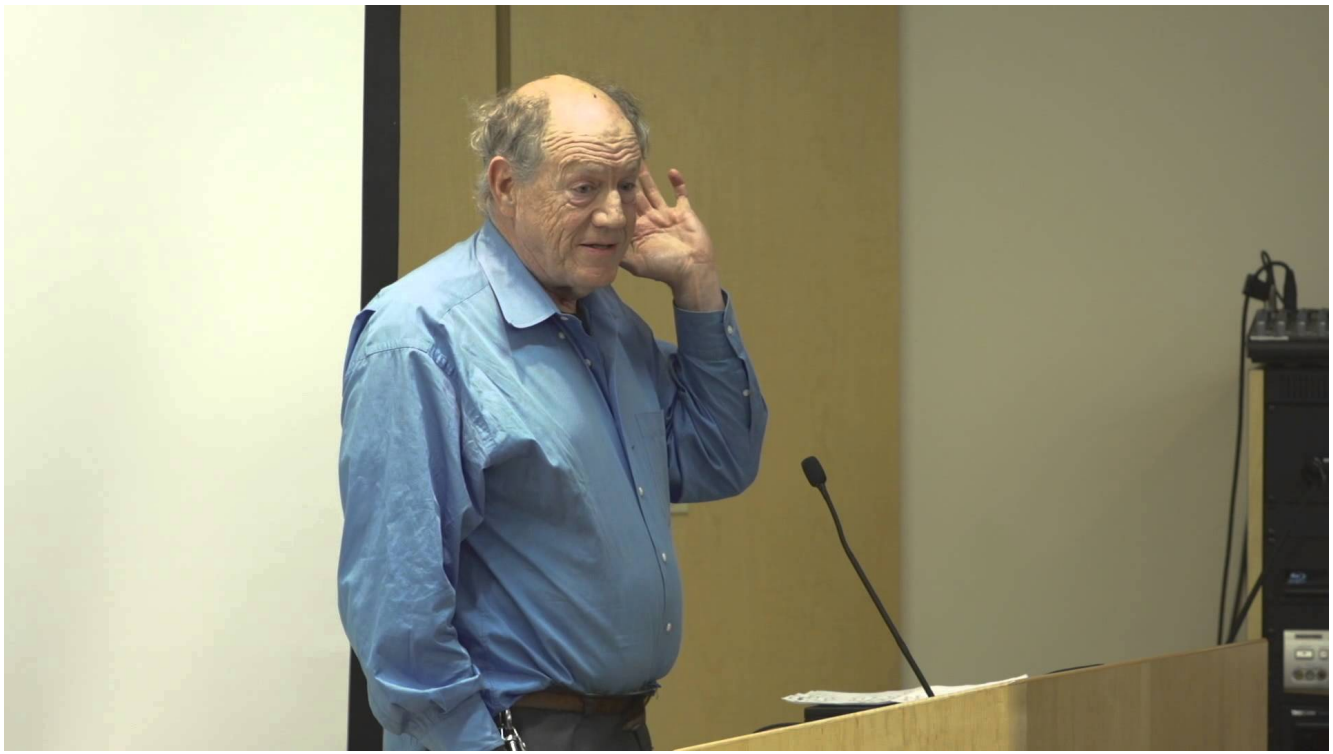
- In Akka i messaggi in arrivo devono essere immutabili per evitare contaminazioni di altri attori
- La mailbox è quindi un buffer in cui i messaggi vengono salvati finchè non sono processati.
- L'immutabilità dei messaggi significa che il programmatore non deve preoccuparsi dei problemi dovuti alla condivisione di dati tra attori
- Siccome gli unici dati condivisi sono immutabili la sincronizzazione non è necessaria.

Message Passing Asincrono - Come avviene?

- Gli attori in Akka eseguono asincronamente quindi anche se il destinatario si trova sulla stessa JVM del mittente l'esecuzione del destinatario non avverrà mai immediatamente.
- Il thread che esegue l'invio del messaggio aggiunge il messaggio alla mailbox del destinatario.
- Ogni qual volta un messaggio viene aggiunto ad una mailbox scatena l'esecuzione di un thread per prendere il messaggio dalla coda e farlo processare, evocando il metodo receive del destinatario
- In generale questi due thread non corrispondono

Grazie per l'attenzione

Domande?



Riferimenti I

Agha, G. (1986). *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA.

Andersen, J. L. (2013). How Erlang does scheduling. `http://jlouisramblings.blogspot.com/2013/01/how-erlang-does-scheduling.html`. [Online].

Chiusano, P. (2010). Actors are not a good concurrency model. `http://pchiusano.blogspot.it/2010/01/actors-are-not-good-concurrency-model.html`. [Online; accessed 12/11/2016].

Erlang/OTP Team (2016). Erlang Documentation 19.1. `http://erlang.org/doc/search/`. [Online].

Hebert, F. (2013). *Learn You Some Erlang for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA.

Hewitt, C. (2010). Actor model for discretionary, adaptive concurrency. *CoRR*, abs/1008.1459.

Riferimenti II

Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Hewitt, M. and Szyperski (2012). The actor model (everything you wanted to know, but were afraid to ask). `https://channel9.msdn.com/Shows/Going+Deep/Hewitt-Meijer-and-Szyperski-The-Actor-Model-everything-you-wanted-to` [Talk at Lang.NEXT 2012, Redmond, Washington].

Hoare, C. A. R. (1978). Communicating sequential processes. *Commun. ACM*, 21(8):666–677.

Lightbend, Inc. (2016). Akka Scala Documentation Release 2.4.12. `http://doc.akka.io/docs/akka/2.4/AkkaScala.pdf`. [Online; accessed 12/11/2016].

Mackay, P. (1997). Why has the actor model not succeeded? `http://www.doc.ic.ac.uk/~nd/surprise_97/journal/vol2/pjm2/`. [Online; accessed 12/11/2016].

Riferimenti III

Pool, T. (2015). Comparison of Erlang Runtime System and Java Virtual Machine. http://ds.cs.ut.ee/courses/course-files/To303nis%20Pool%20.pdf.

Sosnoski, D. (2015). JVM concurrency: Acting asynchronously with Akka. https://www.ibm.com/developerworks/library/j-jvmc5/. [Online; accessed 12/11/2016].

.