Corso di Programmazione Concorrente

Modello ad Attori

Valter Crescenzi
http://crescenzi.inf.uniroma3.it

Sommario

- Motivazioni
- Origini del Modello ad Attori
- Ciclo di Vita degli Attori
  - Attori *vs* Task *vs* f.d.e. *vs* Oggetti
- Prevenzione dell'interferenza ad Attori
- Vantaggi/Svantaggi
- Akka
  - Classi e motodi fondamentali
  - Java pre-8 *vs* Java 8 *vs* Scala
  - Remote deployment
  - Clustering

Motivazioni

- La PC è difficile!
- La scrittura e la manutenzione di codice concorrente continua a richiedere competenze specifiche a costi elevati
- Gli strumenti studiati in questo corso, pur costituendo delle soluzioni, sono:
  - talvolta di basso livello di astrazione
  - talvolta difficili da comporre
  - sempre ostici per il debugging ed il testing

Origini del Modello ad Attori

- Profonde ma ormai lontane radici accademiche
  - Carl Hewitt; Peter Bishop; Richard Steiger (1973).
    A Universal Modular Actor Formalism for Artificial Intelligence. IJCAI.
- Ha avuto un momento di chiaro successo industriale
  - a fine ‘90 nello sviluppo di applicazioni in ambito telecom assieme al linguaggio Erlang
  - N.B. decisivo anche un addizionale ingrediente rispetto la proposta iniziale
    - ✔ meccanismi per la fault-tolerance
      (oltre gli obiettivi formativi di questo corso)

Attori e PC

- E' chiaramente un modello ispirato dalle necessità che scaturiscono con la programmazione di attività
  - *concorrenti* e *distribuite*

Its development was "motivated by the prospect of highly parallel computing machines consisting of dozens, hundreds or even thousands of independent microprocessors, each with its own local memory and communications processor, communicating via a high-performance communications network"

Foundations of Actor Semantics. Clinger, William Douglas. Mathematics Doctoral Dissertation. MIT. (1981)

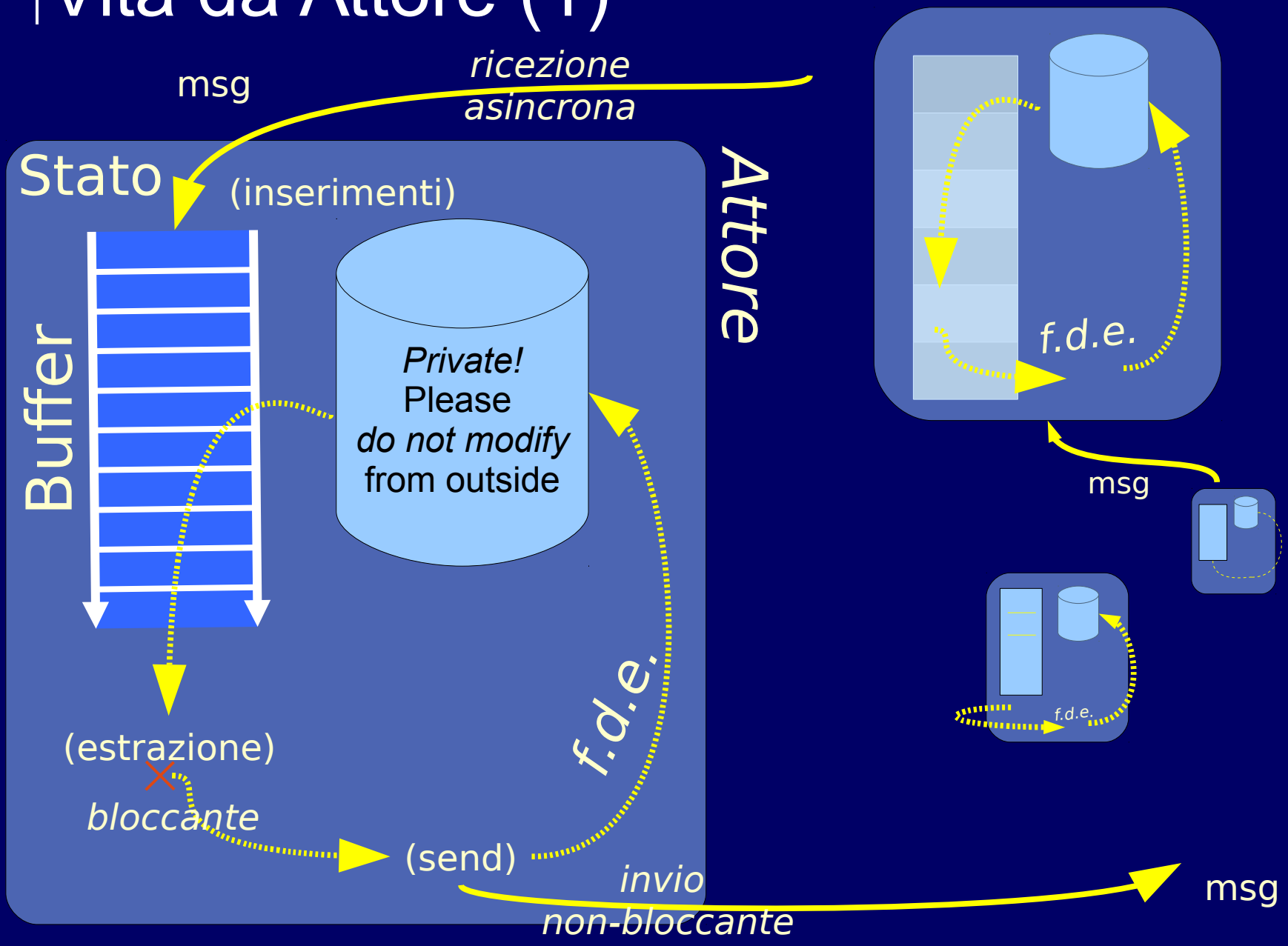
PC/Distribuita e nuovi Paradigmi di Programmazione

- La diffusione architetture multi-core/distribuite favorisce:
  - riscoperta dei tradizionali linguaggi funzionali
    - Haskell, Erlang (purché dotati di compilatore che le sfruttino: OCaml!)
  - diffusione di nuovi linguaggi a paradigma ibrido
    - Scala
  - "ibridizzazione" di linguaggi tradizionalmente OO
    - Java 8+, C++ (entrambi hanno introdotto il lambda calcolo)
  - riscoperta e diffusione di paradigmi di sviluppo "concorrenti"
    - il modello ad attori
      - akka è una implementazione Java/Scala (ma ne esistono altre…)
    - *Reactive Streams*
- Il mercato fa il resto per il crescente interesse verso applicazioni
  - distribuite e scalabili
  - fault-tolerance
  - elastiche
- E la didattica...?

Il Modello ad Attori

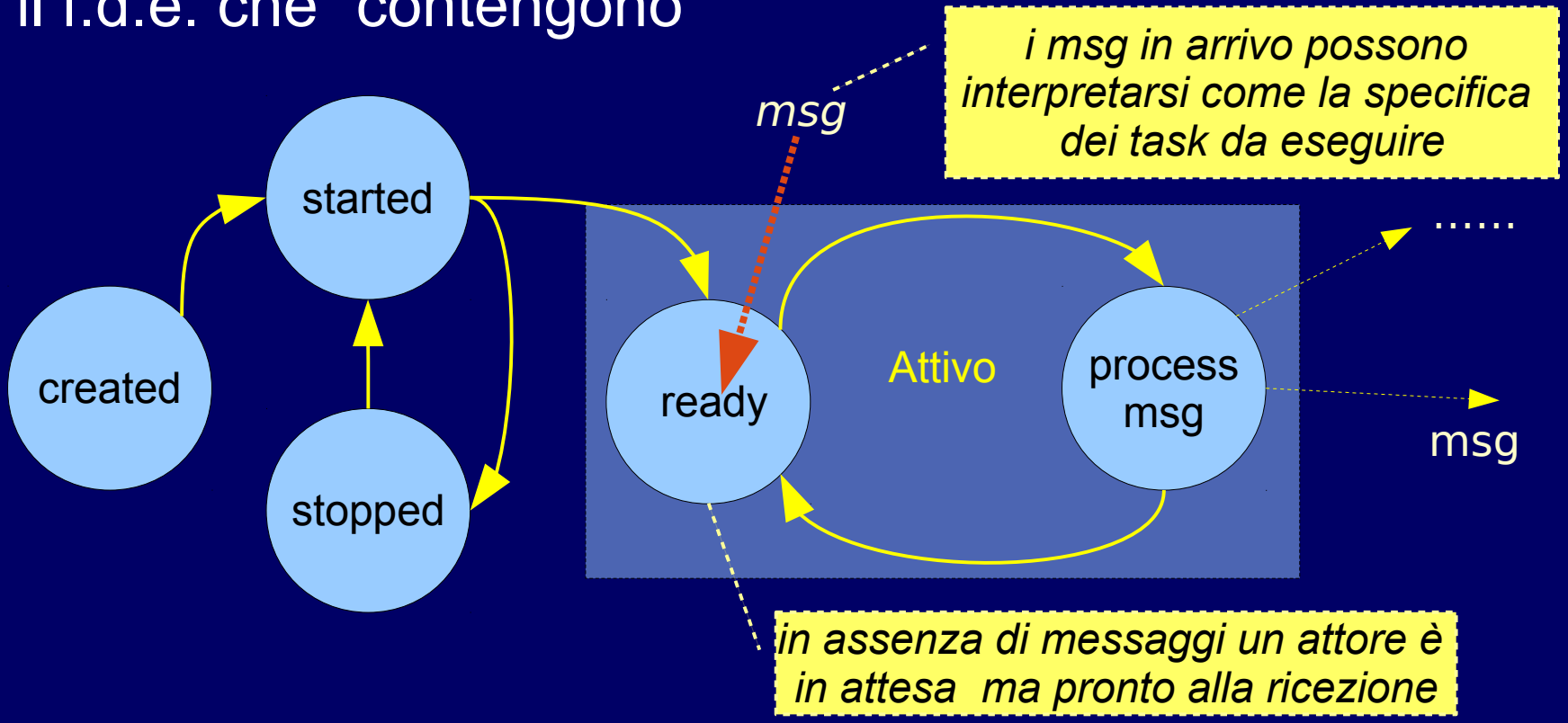
- Applicazioni sviluppate attorno a delle *unità di processamento* chiamate *Attori*
- Fusione tutto sommato naturale del concetto di
  - oggetto (tipica della POO)
  - f.d.e. (tipica della PC)
- Un attore (reso come singolo oggetto):
  - possiede un f.d.e. con accesso *esclusivo* allo suo stato
  - comunica con gli altri attori scambiando messaggi
    - inviandoli – in maniera *non-bloccante*
    - ricevendoli – in un buffer locale di processamento da cui effettua un'estrazione *bloccante*
- Il processamento degli attori è descrivibile come un ciclo indefinito di ricezione e processamento di nuovi messaggi…

Vita da Attore (1)



Vita da Attore (2)

- Viene naturale accostare gli attori ai *Worker-Thread* per il f.d.e. che “contengono”



- Dalla loro implementazione, e dal loro ciclo di vita, si intuisce che invece sono *vicini* ad un concetto a livello intensionale come quello di *task*

Gestione dell'Interferenza

- Per evitare fenomeni di interferenza, il paradigma di programmazione ad attori implica un oculato e mirato utilizzo delle tecniche di gestione dell'interferenza
  - *messaggi:* immutabilità
    - i messaggi scambiati sono immutabili
      - ✔ n.b. Java 8 non supporta costrutti per forzare questo vincolo a tempo di compilazione (Scala si: vedi case-classes)
  - *stato:* confinamento per f.d.e./oggetto (attore)
    - lo stato degli attori non è condiviso
    - solo il f.d.e. dell'attore può accedervi
  - processamento seriale dei messaggi ricevuti
- Complessivamente è un modello di elaborazione semplice ed adatto ad una piattaforma distribuita, prima ancora che multi-processore! (>>)

Tecnica di Confinamento per f.d.e./oggetto

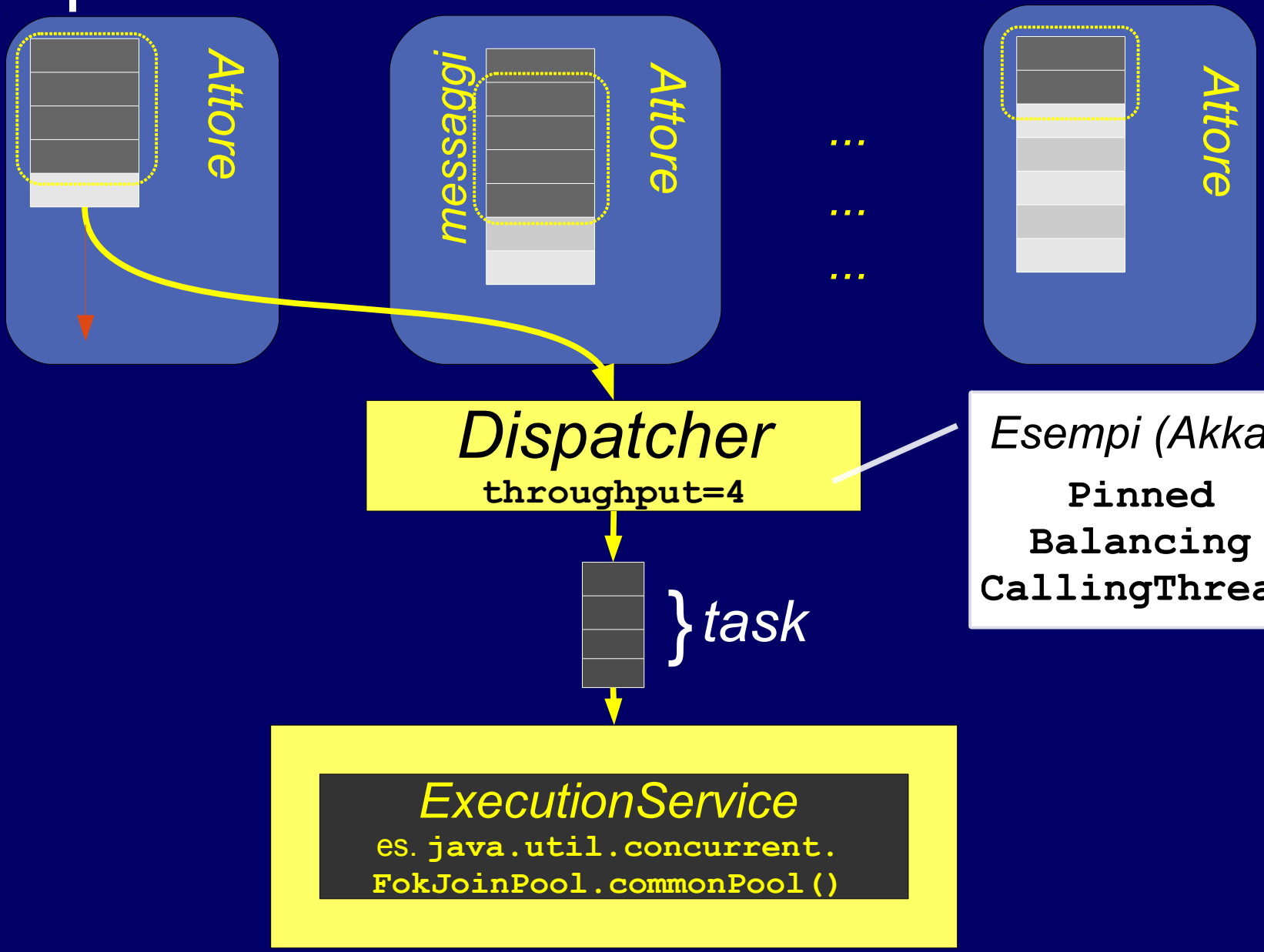
- Si previene l'interferenza sullo stato degli attori adottando semplici *tecniche di confinamento*
- Gli attori sono oggetti con stato
  - mutabile
  - non condiviso (privato)
- Per tutto il suo ciclo di vita esiste un solo f.d.e. che può accedere lo stato dell'attore
  - quello associato all'attore stesso

- ✔ Il f.d.e. associato ad attore in generale è realizzato in momenti diversi da diversi worker-thread >>
  - ✔ trattasi comunque di un dettaglio implementativo

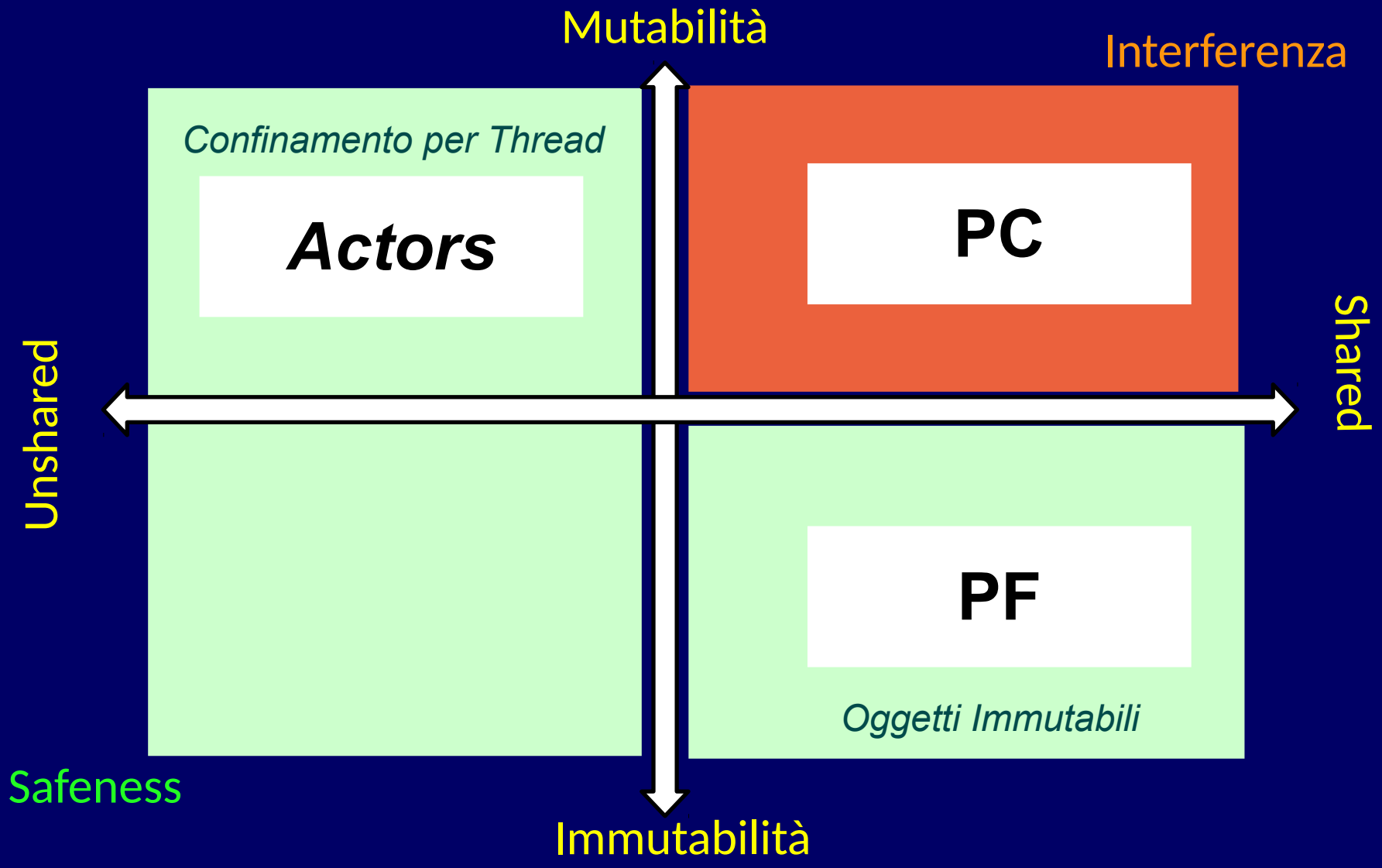
Configurazione di un Dispatcher (Akka)

```
my-dispatcher {
  # Dispatcher is the name of the event-based dispatcher
  type = Dispatcher
  # What kind of ExecutionService to use
  executor = "fork-join-executor"
  # Configuration for the fork join pool
  fork-join-executor {
    # Min number of threads to cap factor-based parallelism number to
    parallelism-min = 2
    # Parallelism (threads) ... ceil(available processors * factor)
    parallelism-factor = 2.0
    # Max number of threads to cap factor-based parallelism number to
    parallelism-max = 10
  }
  # Throughput defines the maximum number of messages to be
  # processed per actor before the thread jumps to the next actor.
  # Set to 1 for as fair as possible.
  throughput = 100
}
```

Dispatcher


Attore
messaggi
Attore
...
...
...
Attore
Dispatcher
throughput=4
task
ExecutionService
es. java.util.concurrent.
FokJoinPool.commonPool()
Esempi (Akka):
Pinned
Balancing
CallingThread

La Programmazione e la Thread-Safeness



Cambio di Paradigma

- E' un modello di programmazione semplice, forse “sorprendente” quando utilizzato per lo sviluppo di codice concorrente ma *NON* distribuito...
  - Paradossalmente spinge a programmare applicazioni concorrenti ma locali come fossero distribute
- L'approccio *opposto* è stato già ripetutamente tentato
  - ovvero, molte proposte hanno provata a rendere trasparente la distribuzione dell'esecuzione (ad es. *RPC*)
  - facendo leva sulle competenze più diffuse
- E’ ormai unanimanente accettato che cercare di nascondere completamente la distribuzione sia comunque utopico, oltre che inutile
  - Tanto vale renderne alcune conseguenze esplicite!

Vantaggi del Modello ad Attori

- Evita molti problemi della PC alla radice
- Rende più accessibile agli sviluppatori la PC distribuita
  - coadiuvigato da modelli di gestione dei fallimenti dimostratosi efficaci
- Alcuni domini sono più naturalmente traducibili ad attori
  - ad es. per il naturale utilizzo dei f.d.e. nella modellazione (es. simulazioni)
- Si prestano a costruire applicazioni di caratteristiche particolarmente richieste dal mercato attuale
  - piattaforme cloud
  - fault-tolerance
  - distribuzione/scalabilità/elasticità
  - semplicità di sviluppo

  trainate dall'evoluzione delle architetture hw moderne

Svantaggi del Modello ad Attori (1)

- Impone un cambiamento rispetto alle abitudini di programmazione più consolidate, ovvero
  - *seriali* (non concorrenti)
  - *locali* (non distribuite)
- Alcuni domini (anche importanti) sono più facilmente modellabili condividendo stato e mal si prestano a modellazioni basati sugli attori
  - finiscono per ripresentarsi i soliti e noti problemi per accedere allo stato condiviso
    - scalabilità
    - thread-safeness

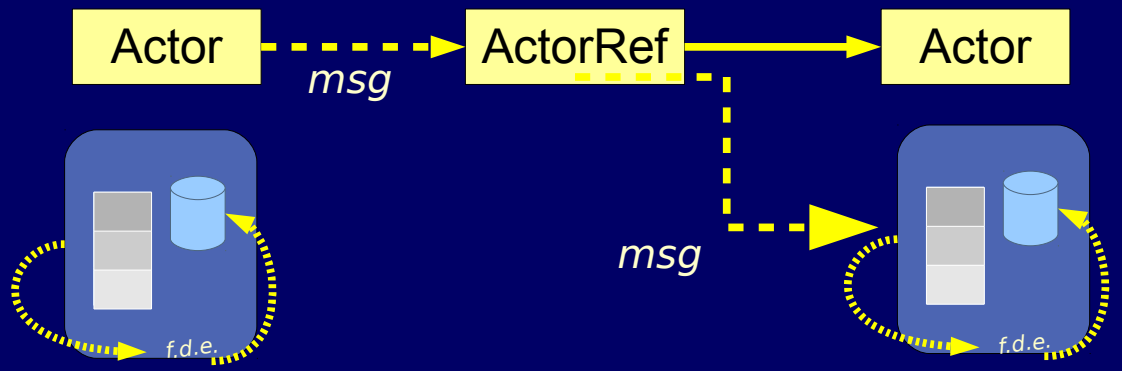
Svantaggi del Modello ad Attori (2)

- Al momento il modello ad attori è *NON-tipato*
  - la definizione di un attore non costringe il programmatore a definire il tipo dei messaggi che può ricevere
    - i messaggi usano il sistema dei tipi del linguaggio di programmazione ospitante
    - sono tipati con il più generico tipo...
  - esistono resistenze e difficoltà di verso l'adozione di un modello più fortemente tipato
    - talune legate a motivazioni quasi "storiche"
      - il modello ad attori "originale" è non tipato

      altre che ricordano le diatribe sui ling.
        - tipati staticamente vs dinamicamente
    - altre ancora di natura squisitamente tecnica
      - ✔ non è affatto semplice, come dimostrano i ripetuti fallimenti
  - discussione tuttora attuale (ed oggetto di attiva ricerca)
    - modulo sperimentale: *akka-typed*

Location Transparancy degli Attori

- Le implementazioni del modello ad attori moderne (*Akka*) possiedono interessanti caratteristiche:
  - un meccanismo di *naming* stabilisce un livello di indirezione usando un riferimento *(ActorRef)* verso un attore *(Actor)*
  - gli attori sono *serializzabili*
- Queste due caratterische, assieme, permettono la *Location Transparancy*
  - gli attori sono referenziabili indipendentemente dalla macchina su cui sono in esecuzione
- Ma attenzione: l'assoluta trasparenza della distribuzione rimane un'utopia
  - alcuni dettagli del ciclo di vita degli attori possono differire tra attori *locali* e *remoti*



Ancora sui Vantaggi del Modello ad Attori

- Gli attori sono serializzabili e trasparenti rispetto alla loro ubicazione su un certa macchina. Si possono
  - *muovere:* spostando anche il carico tra macchine
  - *sdoppiare*: per lo speed-up
  - *terminare*: per non sprecare risorse altrimenti inutilizzate
  - *rimpiazzare:* per politiche di gestione dei fallimenti od aggiornamenti a runtime del loro codice
- Questo consente l'ideazione di soluzioni concorrenti e distribuite che, rispetto a più consolidate alternative, scalano più facilmente e flessibilmente sulle nuove architetture:
  - con più core (architetture multi-core)
  - con più macchine (architetture cloud)
- ✔ Gli approcci tradizionali implicitamente sono orientati verso lo sviluppo *mono-f.d.e. e locale* e sono stati solo successivamente riadattati alle architetture hw moderne

Il Framework Akka

- Scritto in Scala
  - Linguaggio multi-paradigma (funzionale/oggetti)
  - Permette di scrivere codice molto più coinciso
    - ...e risolve la maggior parte dei problemi che Java ha inevitabilmente accumulato in 20+ anni e 10+ versioni
  - Eseguito dalla JVM e compatibile con Java
    - i metodi Java/Scala si possono invocare *bidirezionalmente*
- E' tuttora oggetto di fervente sviluppo
  - ...ma probabilmente il suo successo per alcune tipologie di applicazioni rimane indistricabilmente legato al successo del sottostante modello di elaborazione
- Utilizzabile in maniera piuttosto diversa (a livello di sintassi) in ciascuno di questi ambienti
  - Scala
  - Pre-Java 8
  - Java 8+ (usando lambda function)

Akka in Java

- Dal punto di vista della programmazione ad attori mediante Akka, la versione Java soffre di due principali mancanze (del linguaggio di programmazione)
  - assenza di meccanismi di pattern matching
    - gli attori ricevono generici `Object` come messaggi
      - come prima cosa devono spesso capirne il tipo ed il codice Java per farlo non è mai troppo elegante...>>
  - assenza di un vero tipo funzione tra i tipi di Java 8
    - le *FunctionalInterface* non aiutano in questo senso
- Nonostante questi oggettivi problemi, gli sviluppatori del framework sono molto attenti (ed interessati...) al parco di sviluppatori Java.

akka.actor.*UntypedActor*

- SAM con metodo **`onReceive()`**
  - l'estrazione dal buffer locale dei messaggi *è operata dall'ambiente di esecuzione*, che invoca il metodo astratto passandogli il nuovo messaggio estratto da processare
  - si inviano messaggi con il metodo *non-bloccante* **`tell()`**

```
import akka.actor.UntypedActor;
public class MyUntypedActor extends UntypedActor {
  public void onReceive(Object msg)
                throws Exception {
     if (message instanceof String) {
        System.out.println("Ricevuto: " + msg);
        getSender().tell(message, getSelf());
     } else unhandled(message);
  }
}
```

non-bloccante

mittente

akka.actor.*UntypedActor*

- Se il messaggio fosse di tipo diverso da **`java.lang.String`** risulterebbe necessario anche un inelegante downcasting:

```
public class Greeting implements Serializable {
  public final String who;
  public Greeting(String who) { this.who = who; }
}
```

```
import akka.actor.UntypedActor;
public class Greeter extends UntypedActor {
  public void onReceive(Object msg)
                throws Exception {
    if (message instanceof Greeting) {
        System.out.println("Ricevuto: " +
                           ((Greeting) message).who);
        getSender().tell(message, getSelf());
    } else unhandled(message);
}}}
```

Java 8 ed Akka

- Problema parzialmente risolto in Java 8
- Si sfruttano le nuove possibilità offerte con l'introduzione di una forma di lambda calcolo; a sua volta questo consente l'implementazione di una forma limitata di pattern matching

```
import akka.actor.UntypedActor;
public class Greeter extends AbstractActor {{
  receive(ReceiveBuilder.
    match(Greeting.class, m -> {
      System.out.println("Hello " + m.who);
    }).
    matchAny(unknown -> {
      System.out.println("Unknown message " + unknown);
    }).build());
}}
```

Akka in Scala

- ...senza tuttavia raggiungere l'eleganza della soluzione Scala

```
case class Greeting(who: String);
class Greeter extends Actor {
  def receive = {
    case Greeting(who) => System.out.println("Hello {}", who)
  }
}
```

- Il problema diventerebbe ancora più evidente se lo stesso attore ricevesse messaggi di molti tipi diversi
- Servirebbero addizionali:
  - clausole **`case`** in Scala
  - invocazioni dei metodi **`match`** in Java 8+
  - condizionali con **`instanceof`** in pre Java 8
- A ben vedere è il risultato di due circostanze
  - natura *untyped* degli attori>>
  - assenza di un meccanismo di pattern-matching in Java

Invio di Messaggi: `tell()`

- `tell()`: semantica *non-bloccante*. >Da preferire<

```
public final void tell(java.lang.Object msg,
                       ActorRef sender)
```

Sends the specified message to this ActorRef, i.e. fire-and-forget semantics, including the sender reference if possible.

Pass `ActorRef` `noSender` or `null` as sender if there is nobody to reply to

- Si invia un messaggio e si è liberi di passare ad altro senza introdurre attese
- Akka garantisce che la consegna avvenga al massimo una sola volta e che l'ordine dei messaggi inviati sia rispettato ma solo per ciascuna coppia di attori (mittente, ricevente) coinvolti
- Si tratta di garanzie non particolarmente forti, anzi!
  - è una scelta coerente con la visione del framework, che fornisce un dedicato e separato supporto per la gestione dei fallimenti
  - garanzie più forti comporterebbero costi non sempre sopportabili e non da tutti...

L'Astio verso le Attese Bloccanti di Akka...

- Ovunque nella documentazione di Akka, si consiglia di non effettuare attese *bloccanti* nel codice dei propri attori
- Perché?
  - Perché i momenti in cui si fanno attese bloccanti (ovvero le estrazioni bloccanti dalla *mailbox* dell'attore) sono quelli in cui il controllo passa *naturalmente* all'ambiente di esecuzione
  - Questo si addossa la responsabilità di gestire lo scheduling dei worker-thread imponendo la sua *visione* sulla gestione di *tutte* le attese e l'esecuzione di tutti gli attori che le eseguono
- Introducendo proprie attese bloccanti, fuori dal "controllo" di Akka stesso, risulta poi *necessario*
  - prendere consapevolezza di come funziona l'ambiente di esec.
  - evitare di *contraddirlo* rendendo il suo scheduling inefficace
- Gli stessi consigli valgono per il Framework Fork/Join
  - ✔ l'***`ExecutorService`*** di default del Framework Fork/Join è il *dispatcher* di default anche di Akka

Actor + Future (1)

- Tuttavia alcune chiamate bloccanti possono far comodo
  - ad esempio le chiamate bloccanti per leggere il contenuto di *Future* restituito da una chiamata non-bloccante
  - per questo, alcune tipologie di attese bloccanti hanno meritato un apposito supporto (>>)
- Akka/Scala offrono la loro versione dei *Future*
  - **`scala.concurrent.Future`** (sola lettura)
  - **`scala.concurrent.Promise`** (scrivibile)
    - N.B. l'interoperabilità con Java è motivo di tuttora frequenti aggiustamenti delle librerie Akka
  - E' possibile specificare un'operazione da eseguire al completamento del future invocando i metodi
  - **`Future.onComplete()/onSuccess()/onFailure()`** che ricevono implementazioni di corrispondenti SAM **`OnComplete<T>/OnSuccess<T>/OnFailure<T>`**

Actor + Future (2)

- Es. di codice di processamento dei msg da parte di un attore, esplicitando l'***ExecutorService*** da utilizzare:

```
import akka.dispatch.*;
import scala.concurrent.ExecutionContext;
import scala.concurrent.Future;
…
Future<String> f = future(new Callable<String>() {
    public String call() {
       return aLongComputationResult();
    }                          // task sottomesso
}, system.dispatcher());       // all'ExecutorService/Dispatcher
                               // di default
f.onSuccess(new PrintResult<String>(), system.dispatcher());
…
```

Factory method di un future Scala

- Dove:

```
public final static class PrintResult<T> extends OnSuccess<T> {
    @Override public final void onSuccess(T t) {
        System.out.println(t);
    }
}
```

Invio di Messaggi: `ask()`

- **`ask()`**: semantica *non-bloccante* del metodo ma lettura *bloccante* sulla **`Future<Object>`** fornito come risposta.
- Chiamate bloccanti >Da evitare<

```
public static scala.concurrent.Future<java.lang.Object> ask(ActorRef actor,
                                                            java.lang.Object message,
                                                            long timeoutMillis)
```

Java API for `akka.pattern.ask`: Sends a message asynchronously and returns a `Future` holding the eventual reply message; this means that the target actor needs to send the result to the `sender` reference provided. The Future will be completed with an `AskTimeoutException` after the given timeout has expired; this is independent from any timeout applied while awaiting a result for this future (i.e. in `Await.result(..., timeout)`).

- Implementato tramite un metodo factory statico

  `import static akka.pattern.Patterns.ask;`
- Gli attori *devono* rimanere liberi di processare msg in arrivo
  - ✔ altrimenti occupano il *worker-thread* che li ha presi in carico
  - ❑ compromettendo la reattività del framework
    - in un certo senso, le chiamate bloccanti comportano una *violazione* del modello di programmazione offerto/imposto dal framework

Pipe Pattern (1)

- Problema
  - **`ask()`** restituisce uno **`scala.concurrent.Future<Object>`**
  - ma NON possiamo fare attese bloccanti per conoscerne il risultato!
- Soluzione: il *Pipe Pattern*
  - al completamento del *Future* (esecuzione di **`onSuccess()`**) si invia (con il metodo non bloccante **`tell()`**) un msg ad altro attore (anche se stesso...)

```
…
final ActorRef target = …
Timeout timeout = new Timeout(Duration.create(5, "seconds"));
final Future<Object> f = Patterns.ask(actor, msg, timeout);
f.onSuccess(new OnSuccess<Object>() {
        @Override
        public final void onSuccess(Object res) {
            target.tell(res, getSender());
        }
    }, system.dispatcher());
}
```

Attore a cui è destinata la risposta quando disponibile

Pipe Pattern (2)

- E' un problema abbastanza frequente da aver meritato un supporto apposito per velocizzare la scrittura del codice

```
import akka.dispatch.*;
import scala.concurrent.ExecutionContext;
import scala.concurrent.Future;
import akka.pattern.Patterns;
import akka.util.Timeout;
Timeout timeout = new Timeout(Duration.create(5, "seconds"));
Future<Object> future = Patterns.ask(actor, msg, timeout);
// ...
akka.pattern.Patterns.pipe(future, system.dispatcher()).to(target);
```

pipe

```
public static <T> PipeToSupport.PipeableFuture<T> pipe(scala.concurrent.Future<T> future,
                                                       scala.concurrent.ExecutionContext context)
```

Register an onComplete callback on this Future to send the result to the given ActorRef or ActorSelection. Returns the original Future to allow method chaining. If the future was completed with failure it is sent as a Status.Failure to the recipient.

to

```
public PipeToSupport.PipeableFuture<T> to(ActorRef recipient)
```

Creazione di Attori

- Un **`akka.actor.ActorSystem`** è un gruppo di attori che condividono una configurazione
  - specificata con un oggetto **`akka.actor.Props`**
    - ad es. può contenere anche i dettagli sulla distribuzione degli attori su un cluster di macchine (>>)
  - E' il punto di ingresso per creare e raggiungere gli attori

```
ActorSystem system = ActorSystem.create("MySystem");

ActorRef greeter = system.actorOf(
                       Props.create(Greeter.class),
                       "greeter"
                   );
greeter.tell(new Greeting("Charlie Parker”));
```

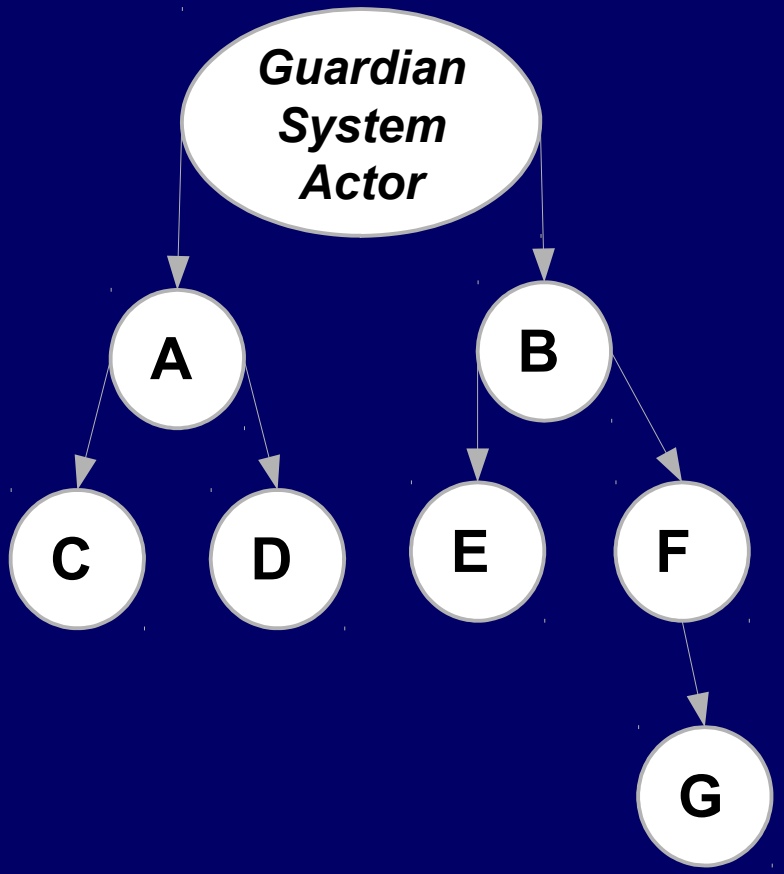
Creazione di Attori Figli

- Un **`akka.actor.ActorSystem`** è un punto di ingresso per la creazione di gerarchie di attori
- La creazione di un attore è molto meno costosa della creazione di un thread (costo simile a quello di un oggetto)
  - il numero di attori che si possono creare con una certa quantità di memoria è diversi ordini di grandezza superiore al numero di thread che si possono creare nella stessa quantità di memoria
- E' prassi normale creare attori per raggiungere un livello di suddivisione a grana *fine* delle responsabilità

```
class AnActor extends UntypedActor {
  final ActorRef child =
    getContext().actorOf(
        Props.create(MyChildActor.class),
        "myChild"
    );
}
```

Naming Gerarchico

- Classico sistema di nomi gerarchico. Ad es.
  - /A
  - /A/C
  - /B/F/G
- **`akka.actor.ActorPath`**
- I padri “supervisionano” i figli
  - la gerarchia ha un importante ruolo nei meccanismi di gestione dei fallimenti (non discussi qui)



- ✔ In fase di *modellazione* la richiesta di un ordinamento gerarchico (essenzialmente legata ai meccanismi di supervisione) può risultare invece limitante...

Distribuzione Remota

- Gli attori di Akka possono essere distribuiti su macchine remote opportunamente configurate per “riceverli”

```
akka {
  actor {
    provider = remote
    deployment {
      /Greeter {
        remote = akka://MySystem@machine1:2552
      }
    }
  }
}
```

Attori Senza Tipo

- Gli attori previsti sono non tipati
  - ovvero, un attore non specifica (a tempo statico) l'elenco dei tipi di messaggio che sa processare
- Principali motivazioni:
  - era così previsto sin dalla proposta originale
  - a tempo dinamico risulta più semplice sfruttare la location-transparancy degli attori per
    - soluzioni architetturalmente più flessibili>>
    - gestione dei fallimenti...
- Al momento, anche se è pacifico che la specifica dei tipi
  - semplificherebbe la scrittura di attori in Java
  - aiuterebbe l'interazione con sistemi legacy

  ...è oggetto di discussioni e di attiva ricerca sino a che punto il modello ad attori debba essere rivisto in tal senso

  - ✔ **`akka-typed`** è correntemente un modulo sperimentale

Attori Senza Tipo e `become()`

- E' possibile *reinstallare* il codice di processamento dei messaggi a tempo di esecuzione

```
public class Greeter extends AbstractActor {
  public Greeter {
    receive(ReceiveBuilder.
      match(Greeting.class, m -> {
        println("Hello " + m.who);
      }).
      matchEquals("stop" -> {
        context().become(ReceiveBuilder.
          match(Greeting.class, m -> {
            println("Go Away!");
          }).build());
      }).build();
    }
}
```

- ✔ Sulla base dello stesso meccanismo è possibile ad es. sostituire dinamicamente un attore con un pool di attori

Attori Speciali: i Router

- Instradano i messaggi verso un gruppo di attori organizzati in un pool
- Risoluzione quasi indolore di alcuni *colli-di-bottiglia*
  - a tempo dinamico:

```
ActorRef router = context().actorOf(
  new RoundRobinPool(5).props(
        Props.create(Worker.class)), "router" );
```

  - direttamente da file di configurazione:

```
akka.actor.deployment {
  /service/router {
    router = round-robin-pool
    resizer {
      lower-bound = 12
      upper-bound = 15
    }
  }
}
```

Clustering ed Attori

- Gli stessi meccanismi si prestano a gestire un sistema di attori che sia distribuito su un cluster di più macchine
  - costruito come estensione dei servizi di remoting

```
… class ClusterActorRefProvider … extends RemoteActorRefProvider
```

  - interessanti servizi aggiuntivi
    - cluster-wide failure-detection
    - routing su cluster (load balancing) ; pool capabilities

```
akka {
  actor {
    provider = cluster
    ...
  }

  cluster {
    seed-nodes = [
      "akka.tcp://ClusterSystem@127.0.0.1:2551",
      "akka.tcp://ClusterSystem@127.0.0.1:2552"
    ]
    auto-down = off
  }
}
```

Considerazioni sul Modello ad Attori

- Difficile predirre la diffusione del modello conseguente alla sua riscoperta
- Alcune *sensazioni*:
  - per alcuni aspetti, legati principalmente alla distribuzione, i benefici nell'uso degli attori ripagano ampiamente degli sforzi per il cambio di paradigma
    - La *location-transparancy* è un'importante caratteristica
  - ma per la "tipica" programmazione *locale* il paradigma appare di troppo basso livello rispetto ad alcune alternative moderne
- Soluzioni più potenti per la specifica di codice concorrente sembrano affacciarsi prepotentemente
- Come confermato dallo stesso ecosistema Akka…

Ecosistema Akka

- Akka sta rapidamente evolvendo verso un'intero ecosistema di soluzioni
  - con evidenti ed ampi sconfinamenti rispetto agli obiettivi formativi di questo corso
- Tra i vari progetti si segnala *Akka Stream*
  - per il processamento di stream di dati
  - consentono la specifica di pipeline di processamento *tipate* e risolvono i limiti del paradigma ad attori
- Interessante notare che lo sviluppo di **`akka-stream`** è stato motivato anche dall'esigenza di basarvici sopra altri fondamentali moduli Akka come ad esempio **`akka-http`**
  - ✔ Quasi che all'interno dello stesso progetto si reputasse troppo limitante il livello di astrazione imposto dal modello ad attori!

Akka Stream

- *Akka Stream* consente il processamento di *stream* di dati
- ✔ Similarmente ai Java Stream di Java 8+ già visti in questo corso

- Consentono la specifica di *pipeline* di processamento *tipate*
  - offrono un livello di astrazione decisamente superiore
  - automaticamente tradotte in attori dal motore di esecuzione
  - il parallelismo è implicito nel modello di esecuzione
  - in realtà il termine *pipeline* è riduttivo perché si possono creare complessi grafi di processamento
  - implementano la *back-pressure* con *bounded buffer*
    - meccanismo necessario per sfruttare le architetture hw multi-core distribuite (anche se il remoting non è ancora supportato)

Akka Stream: Esempio

```
val source: Source[Int, NotUsed] =
   Source(0 to 200000)
val flow: Flow[Int, String, NotUsed] =
   Flow[Int].map(_.toString)
val sink: Sink[String, Future[Done]] =
   Sink.foreach(println)
val runnableGraph =
   source.via(flow).to(sink)
runnableGraph.run()
```

Akka Stream: Conclusioni

- Reintroducono una tipizzazione più forte
  - ...accantonata dal modello ad attori
- Aumentano notevolmente il livello di astrazione a cui si programma
- Un linguaggio dichiarativo per il processamento di stream di dati
  - Un po' come l'SQL lascia al motore di traduzione/esecuzione le ottimizzazioni...
  - ...lo stesso accade con gli stream: la parallellizzazione è lasciata al motore di esecuzione
    - è consentita dalla traduzione in un sistema di attori
    - prendendo consapevolezza di come avviene la traduzione, è possibile fornire indicazioni decisive per l'efficienza della valutazione

Riferimenti

- *http://akka.io/*
- *Raymond Roestenburg, Rob Bakker, Rob Williams.* Akka 8 in Action - *Manning*
- Akka Concurrency - Building reliable software in a multi-core world. *Derek Wyatt. Artima.*