



POLITECNICO
MILANO 1863

Actor Model - Akka

Luca Mottola

Credits: Alessandro Margara

luca.mottola@polimi.it

<http://mottola.faculty.polimi.it>

Philosophy

«**The world is parallel.** If we want to write programs that behave as other objects behave in the real world, then these programs will have a concurrent structure.

Use a language that was designed for writing concurrent applications, and development becomes a lot easier. Erlang (actor-oriented language) programs model how we think and interact.»

Joe Armstrong

Outline

- Fundamentals
- Communication
- Fault-tolerance

Fundamentals

Why?

- We present the actor-based model for several reasons
 - Different approach to **concurrency** compared to lock-based primitives
 - **Low-level:** does not hide the complexities of building distributed applications, but forces the developers to take care of the concerns related to distribution
 - **Blur the distinction** between local and remote processing
- Many services implemented on top of actor-oriented systems

Why?

- Used in practice to build complex distributed applications
- **Erlang**
 - Ericsson network infrastructure software
 - WhatsApp
 - highscalability.com/blog/2014/2/26/the-whatsapp-architecture-facebook-bought-for-19-billion.html
 - CouchDB
 - RabbitMQ
- .Net **Orleans**
 - Halo multiplayer backend
- Java / Scala **Akka**
 - Customers: BMW, VW Group, Bosch, Siemens, Volvo, ...

Actor model

«The actor model abstraction allows you to **think about your code in terms of communication**, not unlike the exchanges that occur between people in a large organization.»

Akka documentation

Actor model

«The actor model in computer science is a mathematical model of concurrent computation that treats actors as the universal primitives of concurrent computation. In response to a message that it receives, an actor can: **make local decisions, create more actors, send more messages**, and determine how to respond to the next message received. Actors may modify their own private state, but **can only affect each other through messages, avoiding the need for any locks.**»

Wikipedia

Actor model

- Concurrency model where the concept of actor is the **universal primitive**
- Properties of actors
 - Stateful
 - Asynchronous
 - Actors run concurrently and communicate through asynchronous message passing
 - Persistent
 - The state can be saved to persistent storage

Actors: what do they do?

- They receive messages and react by
 - Taking **local decisions**
 - Change their internal state
 - Change their behavior
 - **Send messages**
 - They can reply with 0, 1, or more messages
 - To the sender
 - To other actors they know of
 - **Start new actors**

Basic idea of communication

« Do not communicate by sharing memory; instead, share memory by communicating.»

Effective GO

- Since we do not rely on shared memory, programs can run in distributed settings with no changes compared to a local setup
 - Similarly, the model makes **no assumptions on reliability, message orders, ...**

Example

- We want to implement a bank management application
 - Account balance cannot be negative
 - So we need to avoid violations due to concurrent accesses
- In a **shared memory** programming model we need to
 1. Create an account
 2. Explicitly create a thread to manage one or more accounts
 3. Add code to prevent undesired concurrent accesses
 - E.g., synchronized blocks, locks, ...

Example

- The actor model offers a **different design opportunity** for point 2 and 3
 - Creating a new actor to manage a new account automatically makes it run asynchronously with respect to other actors
 - The internal state of the actor contains the managed account
 - Messages are **processed atomically**
 - As if the entire processing of a message was executed in a synchronized method of a class where all methods are synchronized
 - Messages received meanwhile are queued up and do not affect the currently running receive functionality

Hands-on: Akka

Akka

- Implementation of **the actor model for the JVM**
 - Java and Scala API
- Also provides additional abstractions built on top of the basic actor model
 - Akka clusters
 - Akka sharding
 - Akka streams
 - Akka HTTP
 - ...
- We will present the primitives for
 - Creating actors
 - Exchanging messages
 - Changing behavior
 - Fault tolerance
 - Clustering
 - ...

Create actors

- The easiest way to define a new type of actor is to inherit from the **AbstractActor** class
 - You need to define the `createReceive()` method, which defines how each message is processed
 - It is also possible to override other methods to customize the behavior when entering different states in the lifecycle of the actor
 - `preStart()`
 - `preRestart()`
 - `postRestart()`
 - `preStop()`,
 - ...

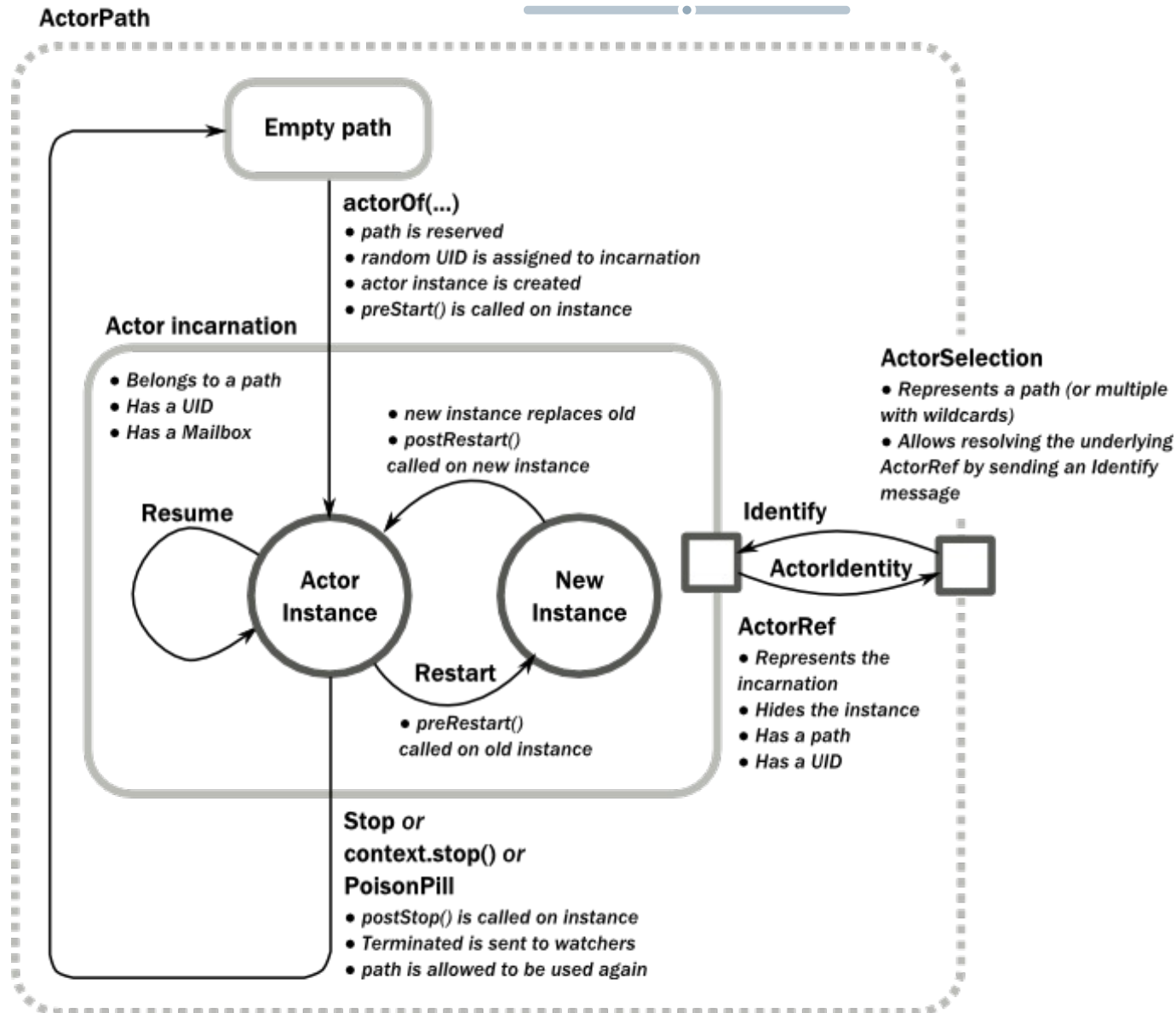
Create actors

- To create an actor you need to
 - Instantiate an ActorSystem

```
ActorSystem sys = ActorSystem.create("sys name");
```
 - Instantiate a new actor starting from a property object that defines the class of the actor

```
Props p = Props.create(MyActor.class);  
sys.actorOf(p, "actor name");
```
 - Often, the property is created and returned within a static method of the actor class

Actors lifecycle



Send messages to actors

- Upon creation, the `actorOf` method returns a reference to an actor (**ActorRef**)
 - The `ActorRef` remains valid throughout the actor lifecycle
 - Even if the actor stops and restarts
- `ActorRefs` can be passed as part of messages to inform other actors
- `ActorRefs` can be used to send messages to actors using `tell()`

Actors basics

```
public class CounterActor extends AbstractActor {
    private int counter;

    public CounterActor() {
        this.counter = 0;
    }

    @Override
    public Receive createReceive() {
        return receiveBuilder() //
            .match(SimpleMessage.class, this::onMessage) //
            .build();
    }

    void onMessage(SimpleMessage message) {
        ++counter;
        System.out.println("Counter increased to: " + counter);
    }

    static Props props() {
        return Props.create(CounterActor.class);
    }
}
```

Actors basics

```
public class Counter {  
    private static final int numThreads = 16;  
    private static final int numMessages = 1000;  
  
    public static void main(String[] args) throws InterruptedException, IOException {  
        final ActorSystem sys = ActorSystem.create("System");  
        final ActorRef counter = sys.actorOf(CounterActor.props(), "counter");  
  
        // Send messages from multiple threads in parallel  
        final ExecutorService exec = Executors.newFixedThreadPool(numThreads);  
        for (int i = 0; i < numMessages; i++) {  
            exec.submit(() -> counter.tell(new SimpleMessage(), ActorRef.noSender()));  
        }  
        // Wait for all messages to be sent and received  
        System.in.read();  
        exec.shutdown();  
        sys.terminate();  
    }  
}
```

Mailbox semantics

- The receiving behavior of the actor is defined by the `createReceive()` method
- When an actor receives a message
 - It checks all the match clauses in the order in which they are defined
 - It uses the method associated to the first matching clause
 - If no matching clause exists, the message is discarded

Mailbox semantics (again)

- The default mailbox policy is **FIFO**
- Messages are processed in the same order in which they are received
 - **FIFO order** from a single sender
 - **Non-deterministic** from different senders
- Messages must be immutable
 - This cannot be statically checked and enforced by the framework...
 - ... but using mutable messages might lead to unexpected behaviors!

Mailbox semantics (again)

- An actor can dynamically change its behavior using the `getContext().become()` method
- This method takes as input a `Receive`, with a new set of match clauses

Mailbox semantics

```
public class AlarmActor extends AbstractActor {
  @Override
  public Receive createReceive() {
    return disabled();
  }

  private final Receive enabled() {
    return receiveBuilder(). //
      match(TriggerMsg.class, this::onTrigger). //
      match(DeactivateMsg.class, this::onDeactivate). //
      build();
  }

  private final Receive disabled() {
    return receiveBuilder(). //
      match(ActivateMsg.class, this::onActivate). //
      build();
  }
}
```

```
private void onTrigger(TriggerMsg msg) {
  System.out.println("Alarm!!!!");
}

private void onActivate(ActivateMsg msg) {
  System.out.println("Becoming enabled");
  getContext().become(enabled());
}

private void onDeactivate(DeactivateMsg msg) {
  System.out.println("Becoming disabled");
  getContext().become(disabled());
}

static Props props() {
  return Props.create(AlarmActor.class);
}
```

Replying to messages

- When processing a message, an actor can
 - Obtain a reference (`ActorRef`) to itself with the `self()` method
 - Obtain a reference (`ActorRef`) to the sender with the `sender()` method
- **To reply** to the sender with a message `msg`, an actor simply invoke

```
sender().tell(msg, self());
```