System Architectures
Actors: Akka

Jonathan Thaler

Department of Computer Science

FH Vorarlberg
University of Applied Sciences

Introduction

- **Most** modern programming languages have an **actor library** nowadays,
- Some languages have built actors into their **core language**, such as the functional programming languages *F#*, *Elixir Erlang*.
- All actor model implementations **differ** in subtle ways from the original concept, for example messages arrive in **sequence** (at least when sent on the same machine) and are **reliable**.

Akka

Akka

Akka is a free and open-source toolkit implemented in **Scala** with bindings for **Java** as both run in the **VM**. Akka was inspired by **Erlang's** actor model implementation and therefore implements the actor model as well. There are **two** different actor system implementations in Akka:

1. **Classic** is the original, first, implementation of Akka, which did **not** exploit typesafety per se.
2. **Typed** enforces **type safety** of messaging and behaviors of actors to avoid certain classes of run-time exception. Is implemented using the classic system under the hood.

Akka is a full-blown, mature toolkit, which provides a number of advanced features not covered in this course, such as *Clustering, gRPC (Google Remote Procedure Call), HTTP, Data Streaming, Microservices, REST*.

Akka

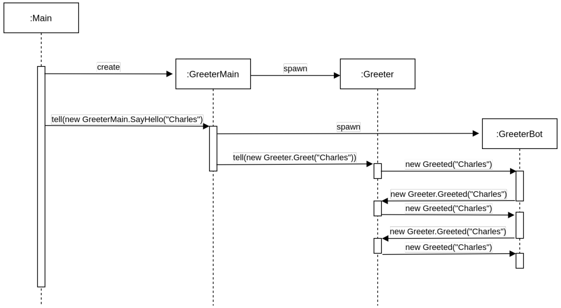
- **Event-driven model**: Actors perform work in response to messages. Communication between Actors is asynchronous, allowing Actors to send messages and continue their own work without blocking to wait for a reply.
- **Strong isolation principles**: unlike regular objects in Java, an Actor does not have a public API in terms of methods that you can invoke. Instead, its public API is defined through messages that the actor handles. This prevents any sharing of state between Actors; the only way to observe another actor's state is by sending it a message asking for it.
- **Location transparency**: the system constructs Actors from a factory and returns references to the instances. Because location doesn't matter, Actor instances can start, stop, move, and restart to scale up and down as well as recover from unexpected failures.
- **Lightweight**: each instance consumes only a few hundred bytes, which realistically allows millions of concurrent Actors to exist in a single application.

Akka

Hello World in Akka [1]

[1] Based on `https://developer.lightbend.com/guides/akka-quickstart-java/index.html`

Akka


:Main
create
:GreeterMain
spawn
:Greeter
tell(new GreeterMain.SayHello("Charles"))
spawn
:GreeterBot
tell(new Greeter.Greet("Charles"))
new Greeted("Charles")
new Greeter.Greeted("Charles")
new Greeted("Charles")
new Greeter.Greeted("Charles")
new Greeted("Charles")

1. `GreeterMain`: the guardian actor that **bootstraps** everything.
2. `Greeter`: receives commands to `Greet` someone and responds with a `Greeted` to confirm the greeting has taken place.
3. `GreeterBot` receives the reply from the `Greeter` and sends a number of additional greeting messages and collect the replies until a given max number of messages have been reached.

Akka

Live coding...

Akka

Akka in relation to the Actor Model:

1. **Send** a finite number of messages to other Actors it knows using the `tell` method from `ActorRef`.
2. **Create** a finite number of Actors provided by the `ActorContext.spawn(Behavior<U>, String)`.
3. **Designate** the **behavior** for the next message implicit in the signature of `Behavior` in that the next behavior is always returned from the message processing logic.
4. An `ActorContext` in addition provides access to the Actor's own **identity** with `getSelf`.

Hello World in Erlang

We see that a lot of **ceremony** is necessary in this implementation of actors. This is due to Java's **verbosity**, the **difficulty** of object-oriented programming to deal with pure data and the fact that the actor model was simply **not** built into the Java core language. In Erlang this example is much **shorter** and concise, involves not nearly as much ceremony and is therefore much more **readable**.

Hello World Erlang

Greeter

```erlang
create() ->
  Pid = spawn(fun() -> process() end),
  Pid.

process() ->
  receive
    {greet, Whom, ReplyTo} ->
      io:fwrite("Hello ~s ~n", [Whom]),
      ReplyTo ! {greeted, Whom, self()},
      process()
  end.
```

Hello World Erlang

Greeter Bot

```erlang
create(Max) ->
  Pid = spawn(fun() -> process(1, Max) end),
  Pid.

process(Counter, Max) when Counter > Max ->
  ok;
process(Counter, Max) ->
  receive
    {greeted, Whom, ReplyTo} ->
      io:fwrite("Greeting ~w for ~s ~n", [Counter, Whom]),
      ReplyTo ! {greet, Whom, self()},
      process(Counter + 1, Max)
  end.
```

Hello World Erlang

GreeterMain

```erlang
create() ->
  Greeter = greeter:create(),
  Pid = spawn(fun() -> process(Greeter) end),
  Pid.

process(Greeter) ->
  receive
    {sayHello, Name} ->
      GreeterBotRef = greeterbot:create(3),
      Greeter ! {greet, Name, GreeterBotRef},
      process(Greeter)
  end.

GreeterMain = greetermain:create(),
GreeterMain ! {sayHello, "Charles"},
receive
    _ ->
        ok
end.
```

Akka

Factorial in Akka [2]

[2] https://homepages.fhv.at/thjo/lecturenotes/sysarch/actors.html#recursive-factorial