# Scala, Akka, and Actors: Overview

## Contents

## 1 Actors

- an **actor** is an object which encapsulate **state** and **behavior**
- communication between actors are exclusively by exchanging messages which are placed into the recipient's mailbox
- **hierarchy of actors**:
  - an actor can **supervise** child actors
  - each actor has exactly only one **supervisor**
- **actor best practices**
  - do not pass **mutable objects** between actors – prefer **immutable messages**
  - actors are made to be containers for beahvior and state
    * try NOT to send "behavior" within messages (tempting – e.g. process migration) but may be very difficult to debug, etc.
  - top-level actors are expensive – try to minimize their numbers
- **supervisor**
  - supervisor (parent) delegates tasks to subordinates (child)
  - **when child fails, parent must handle the failure**
    * when failure; suspend itself and all its subordinates; sends a message to parent; signaling parent of failure
- **actor reference**: subtype of `ActorRef`
  - purpose: support sending meessages to the actor it represents
  - each actor has its local reference through `self`
  - each receptor can access sender's reference by `sender` field
  - `Router inherits ActorRef`
    * sending message to `Router` will cause its message to be routed to children of the "Router"
- **actor path**: since actors are created in hierarchical fashion, actor names can be given in a hier-path
  - `"[ActorSystem]://[RootGuardian]/actor1/actor2"`

```
"akka://my-sys/user/service-a/worker1" // purely local
"akka.tcp://my-sys@host.example.com:5678/user/service-b" // remote
"cluster://my-cluster/service-c" // clustered (Future Extension)
```

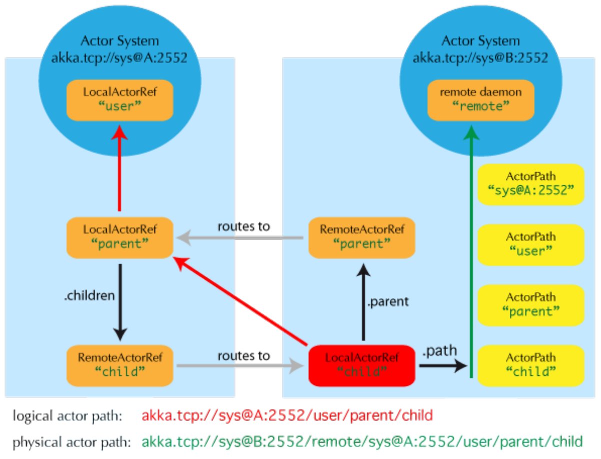
- **OBTAINING Actor references**
  - **creating actors**

    ```
    ActorSystem.actorOf
    ActorContext.actorOf
    ```

  - **looking up actors by concrete path**

    ```
    ActorSystem.actorSelection

    context.actorSelection("../brother") ! msg
    context.actorSelection("/usr/serviceA") ! msg
    ```



## 2 Actors and Java Memory Model

- two ways multiple threads can execute actions on shared memory (in Actors implementation in Akka)
  - **if a message is sent to an actor (e.g. by another actor)**. In most cases messages are immutable, but if that message is not a properly constructed immutable object, without a "happens-before" rule, it would be possible for the receiver to see partially initialized data structures and possibly even values out of thin air (longs/doubles).
  - **if an actor makes changes to its internal state while processing a message, and accesses that state while processing another message moments later**: It is important to realize that with the actor model you don't get any guarantee that the same thread will be executing the same actor for different messages.

  To prevent visibility and reordering problems on actors, Akka guarantees the following two "happens before" rules:

  - **The actor send rule**: the send of the message to an actor happens before the receive of that message by the same actor.

– **The actor subsequent processing rule**: processing of one message happens before processing of the next message by the same actor.