POLITECNICO
MILANO 1863

Actor Model - Akka

Luca Mottola
Credits: Alessandro Margara
luca.mottola@polimi.it
http://mottola.faculty.polimi.it

Outline

- Fundamentals
- Communication
- Fault-tolerance

Fault-tolerance

Managing faults

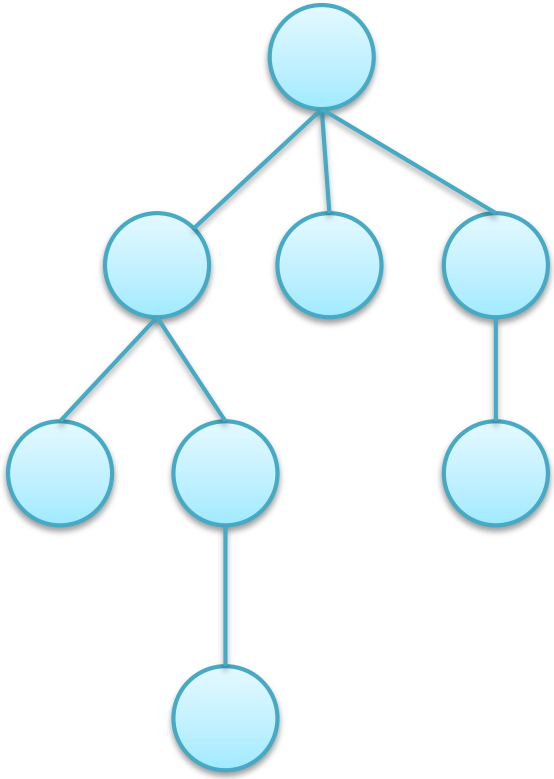
- Faults and exceptional behaviors are managed through the concept/pattern of **supervision**

- Each actor has a supervisor that monitors its execution state
  - …which is plainly another actor

- A supervisor can decide to terminate and possibly restart a supervised faulty actor
  - Restart from scratch or from the latest available state

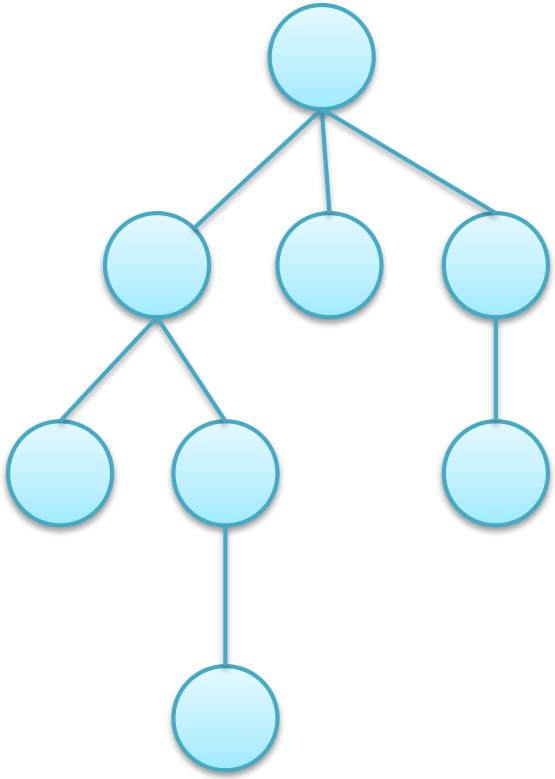
Supervisor hierarchy

- Applications are typically organized in **supervision trees**
- Each supervisor knows how to handle failures in its directly supervised nodes
- If a supervisor cannot handle a problem locally, it propagates the fault to the upper layer
- On the top of the hierarchy there are typically standard supervisors offered by the actor framework



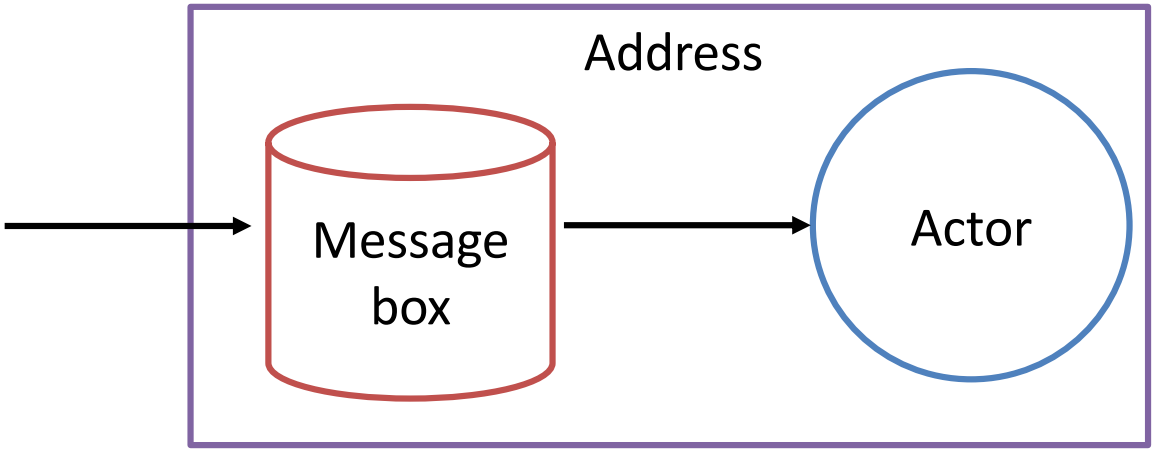
Supervisor hierarchy (again)

- The model favors the definition of hierarchies of responsibility
- If a **failing actor** contains important data that shall not be lost, the **supervisor actor** sources any possibly dangerous subtask to children



Supervisor

- The framework handles the lifecycle of an actor
- When an actor is restarted
  - Its address does not change
  - Its message box is retrieved from persistent state
    - Which lives outside of the actor memory



Hands-on: Akka

when an actor creates another actor with its own local context, it becomes its supervisor

Fault tolerance

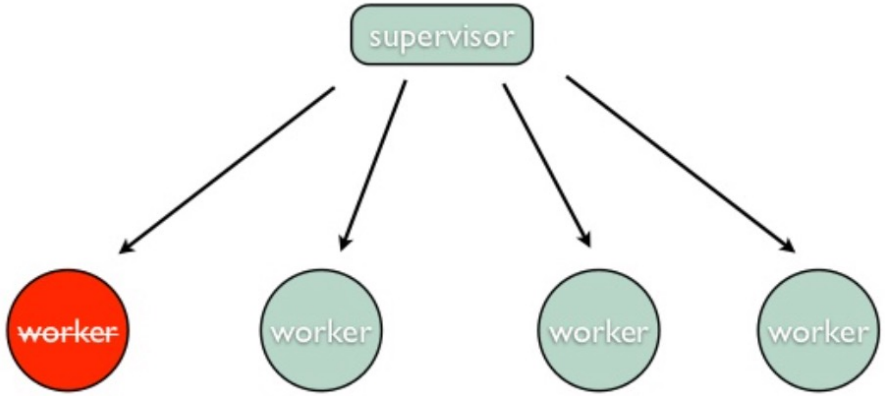
- An actor is responsible for all actors it creates in its context
  - Created through the `getContext().actorOf()` method
  - Stopped through the `getContext().stop()` method

- The supervision strategy can be customized by overriding the **`supervisorStrategy()`** method
  - Depending on the type of exception
  - Depending on the number of errors
  - Independent of the supervised child
    - Philosophy: if you need more flexibility, build a deeper hierarchy!
    - An actor should normally supervise actors of the same type, that is, providing the same or similar functionality

how we do supervision need to be independet to whom we

Fault tolerance

- Different supervision strategies exist
  - One-for-one
  - One-for-all



- Similarly, different actions may be applied when a fault occurs, and the supervisor is asked to manage the situation
  - `stop()`, `restart()`, `resume()`, `escalate()`

Example

```
public class CounterSupervisorActor extends AbstractActor {

    private static SupervisorStrategy strategy =
        new OneForOneStrategy(
            10,
            Duration.ofMinutes(1),
            DeciderBuilder.match(Exception.class, e ->
                                 SupervisorStrategy.restart())
                .build());

    @Override
    public SupervisorStrategy supervisorStrategy() {
      return strategy;

…
```

10 = maximum number of faults that may be managed with this strategy within a

what we need to do upon recognising a fault is specified with the match() clause: that says whenever a g

Example

```
public class CounterActor extends AbstractActor {

…

  void onMessage(DataMessage msg) throws Exception {
    if (msg.getCode() == Counter.NORMAL_OP) {
      System.out.println("I am executing a NORMAL operation...counter is now " + (++counter));
    } else if (msg.getCode() == Counter.FAULT_OP) {
      System.out.println("I am emulating a FAULT!");
      throw new Exception("Actor fault!");
    }
  }

  @Override
  public void preRestart(Throwable reason, Optional<Object> message) {
    System.out.print("Preparing to restart...");
  }

  @Override
  public void postRestart(Throwable reason) {
    System.out.println("...now restarted!");
  }

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

Example

```
public class CounterSupervisor {

  public static final int NORMAL_OP = 0;
  public static final int FAULT_OP = -1;

  public static final int FAULTS = 1;

  public static void main(String[] args) {
    scala.concurrent.duration.Duration timeout = scala.concurrent.duration.
                                                    Duration.create(5, SECONDS);

    final ActorSystem sys = ActorSystem.create("System");
    final ActorRef supervisor = sys.actorOf(CounterSupervisorActor.props(), "supervisor");

    ActorRef counter;
    try {
    scala.concurrent.Future<Object> waitingForCounter = ask(supervisor,
                                                    Props.create(CounterActor.class), 5000);
    counter = (ActorRef) waitingForCounter.result(timeout, null);

    counter.tell(new DataMessage(NORMAL_OP), ActorRef.noSender());
    for (int i = 0; i < FAULTS; i++)
      counter.tell(new DataMessage(FAULT_OP), ActorRef.noSender());
    counter.tell(new DataMessage(NORMAL_OP), ActorRef.noSender());

    sys.terminate();
…
```

clustering is a way to group actors and actor system to coordinate together in a total decentralized way

Clustering

- Akka clustering offers a membership service
  - Decentralized
  - No single point of failure/bottleneck

- Implementation
  - Peer to peer
  - Gossip protocol
  - Automatic failure detection

Clustering terminology

- **Node**: a logical member of a cluster
  - There can be multiple nodes on each physical machine
  - Each node is identified by a tuple hostname:port:uid
  - You can think of each node as a process / actor system

- **Cluster**: a set of nodes joined together through the membership service

- **Leader**: a role in the cluster
  - A single node in the cluster acts as a leader
  - The leader manages cluster convergence

Clustering basics

- An Akka application can be distributed over a cluster
- Each node hosts some part of the application
- Cluster membership and the actors running on a member node are decoupled

- A node can be a member of a cluster hosting any number of actors
- An actor system/node joins a cluster by sending a **join command** to one of the nodes in the cluster

Clustering protocol

- Nodes organize themselves into an overlay network

- They distribute information about cluster members using a gossip protocol
  - Nodes propagate messages containing their current view of the membership
  - Nodes update their view based on the received messages
    - Messages are designed in such a way that the state of nodes eventually converges
    - They contain **vector clocks** to record a partial order of the events (nodes joining, leaving, …) observed in the environment

Clustering protocol

- Information about the cluster converges at a given node when the node can prove that the cluster state it is observing **is seen the same** by all other nodes in the same cluster
- Gossip convergence cannot occur while some node is “unreachable”
  - A state in the lifecycle of nodes indicating that it is not currently possible to communicate with the node
  - The node need to become reachable again or be removed from the cluster

Seed nodes

- Seed nodes are configured contact points for new nodes that want to join the cluster

- When a new node wants to join the cluster
  - It contacts all the seed nodes
    - At least one needs to be active
  - It sends a join command to the first seed node that answers

Cluster tools

- Akka offers higher-level tools that build on top of clustering
  - Cluster singleton: to ensure that a single actor of a certain type exists in the cluster
  - Cluster sharding: distributes actors across nodes of the cluster
    - Ensuring that they can communicate without knowing their physical location
  - Distributed data: creates a distributed key-value store across the nodes of the cluster
  - Cluster metrics: to publish and collect health metrics about cluster members

Further readings

- Akka actors official documentation
  - https://doc.akka.io/docs/akka/current/index-actors.html