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

Communication

Communication

- Sharing of information takes place exclusively through message passing

- Thus, it is fundamental to define the **semantics** and properties of the communication model

Communication

- Communication is **not mediated** by any entity
  - Unlike other programming models where channels can add semantics
    - Unidirectional vs bidirectional
    - Reliable vs unreliable (with/without loss)
    - With/without duplicates
    - With different ordering guarantees
    - …
- In a typical implementation of the actor model, the channel only offers **best-effort communication**
  - At most once (no duplicates, but messages can be lost)
  - No ordering guarantees
    - In Akka, for example, we enjoy FIFO order between any sender-receiver pair only when using TCP for network communication

Being Best-effort

- [Some say] is difficult to program, but has some good reasons

- In distributed settings, it is difficult to offer better guarantees
  - IP is a best-effort protocol
  - TCP cannot mask network partitions

- The actor model **forces to reason about the worst case**…
- … to build application that are robust
  - On a single machine, as well as…
  - …on multiple machines

Best-effort

- Stronger guarantees can be
  - Expensive
  - Not always necessary

- "Less is more"
- Developers can **implement them on top of the basic** programming model

Communication

- Each actor is identified by **an address**
  - Abstracts away the physical location and network configurations
  - The developer **needs not know** where an actor is located to communicate with it
    - Same machine
    - Same cluster (more on this later)
    - Geographically remote site

- An address can also represent more than one actor
  - For load balancing, proxying, …

- An actor can possibly have more than one address

Stash

- It is possible to stash a message that cannot be processed in the current state (behavior) for later processing

- To do so, it is necessary to inherit from **AbstractActorWithStash**
  - The `stash()` method saves the message for later processing in a different state
  - The `unstashAll()` method extracts all the messages from the stash, in the same order in which they were added

Motivation for the Ask Pattern

- The tell operation is asynchronous
  - The sender sends the message and does not wait for a reply

- Waiting for a specific reply is not trivial
  - A workaround
    - Change the behavior to match the expected reply and stash all other messages
    - Upon receiving the expected message, unstash all stashed messages and revert to the normal behavior

Ask Pattern

- An alternative is offered by the Ask Pattern

- The Ask Pattern sends a message and returns a **future**, which will contain the reply when it becomes available
  - `Patterns.ask(receiver, msg, timeout)`
  - The receiver replies as usual `sender().tell(reply, self())`
  - The sender can block on the future to obtain a blocking/synchronous behavior `Await.result(future, timeout)`

Hands-on: Akka

Distribution

- One of the advantages of using Akka is that actors can be **transparently** distributed across multiple machines
  - The only thing that we need to know is the address of a remote actor
  - Then, we can exchange messages with that actor as if it was local
    - This includes the propagation of **ActorRefs** that refer to other actors in any remote machine

Distribution

- Let’s implement a client/server application in Akka
- Akka supports multiple communication protocols
  - Including TCP, which we will use in the following

Distribution

- First, we need to configure the system to listen to a given TCP port

```
akka {
  actor {
    provider = remote
  }
  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "127.0.0.1"
      port = 6123
    }
  }
}
```

Distribution

- The server instantiates a new actor

```java
public static void main(String[] args) {
   Config conf =
     ConfigFactory.parseFile(new File("conf"));
   ActorSystem sys = ActorSystem.create("Server", conf);
   sys.actorOf(ServerActor.props(), "ServerActor");
}
```

Distribution

- The client will retrieve the remote server by name
- Within the client actor

```
String serverAddr =
"akka.tcp://Server@127.0.0.1:6123/user/serverActor";

ActorSelection server =
      getContext().actorSelection(serverAddr);
```

- …and everything else remains the **same**!