Actor model

• Model of concurrent computation

• Carl Hewitt and others 1973

• An actor receives messages, performs computation, creates more actors, sends more messages

• Like objects, but inherently concurrent
Erlang

- Dynamically typed functional programming language
- Developed by Joe Armstrong and others at Ericsson in 1986-7
- Designed to run telephony systems
  - fault-tolerant – down times of seconds/year
  - hot code replacement – cannot bring system down for upgrades
  - concurrent – actors
  - distributed
  - soft real-time
  - non-stop
Erlang processes

• Very lightweight concurrent activities
  • fast to create and destroy (~10 microseconds)
  • fast to send messages
  • small (~1KB each)
  • share no memory, completely independent
  • interact only through message passing
Primitives

• spawn
• send
• receive
Spawn

- Pid = spawn(Fun)

  - create a new process that evaluations function Fun
  - returns a process identifier Pid
Message send

- Pid ! M
  - Send message M to Pid

- Sending a message is asynchronous
  - Sender does not wait for a response

- Pid1 ! Pid2 ! Pid3 ! ... ! M
  - Send message to Pid1, Pid2, Pid3, ...
Message receive

receive

    Pattern1 [when Guard1] -> Expressions1;
    Pattern2 [when Guard2] -> Expressions2;
    ...
end

• Receive a message sent to this process.
• Does a pattern match to choose which case to run
Implementation

- Every process has a **mailbox**
  - a queue of messages sent to it

- receive removes the next matching message from the mailbox

- messages can contain any type of data
Example

```erlang
loop() ->
    receive
        {rectangle, Width, Ht} ->
            io:format("Area of rect is ~p~n", [Width * Ht]),
            loop();
        {circle, R} ->
            io:format("Area of circle is ~p~n", [3.14 * R * R]),
            loop();
        Other ->
            io:format("Huh?"),
            loop()
    end.

> Pid ! {rectangle, 6, 10}
Area of rect is 60
```
loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! Width * Ht,
            loop();
        {From, {circle, R}} ->
            From ! 3.14 * R * R,
            loop();
        Other ->
            From ! {error, Other}
            loop()
    end.

> Pid ! {self(), {rectangle, 6, 10}}
> receive Response -> Response end.
60
SMS

- Used for processing SMS messages
  - 10000s of messages per second
- Spawn a new process for every message
- Bug in UK: whenever a message containing the euro symbol (€) received, process crashes
  - But since all processes independent, only these messages were lost
How well does Erlang scale

- Yaws is a web server written in Erlang
- Apache is a web server written in C
Scalability: Apache (C) vs. Yaws (Erlang)

Throughput (KB/s) vs. Connections

Apache (local)
Apache (NFS)
Yaws (NFS)

Apache dies at 4,000 connections. YAWS? 80000+…
Scalability: Apache (C) vs. Yaws (Erlang)

Apache dies at 4,000 connections; Yaws dies at 80,000 connections.
Scala actors

- Scala supports an actor model also
  - asynchronous and synchronous message sends
  - futures
Ping pong

- Message types:
  case object Ping
  case object Pong
  case object Stop

- An actor:
  import scala.actors.Actor
  import scala.actors.Actor._
  class Ping(count: Int, pong: Actor) extends Actor {
    def act = {
      var pingsLeft = count - 1
      pong ! Ping
      while (true) {
        receive {
          case Pong => /* handle the Pong message */
        }
      }
    }
  }
Pass to the Ping constructor the number of Ping messages to send and the receiver

Ping loops, waiting for Pong messages

When Pong arrives, send back Ping or Stop
class Ping(count: Int, pong: Actor) extends Actor {
    def act = {
        var pingsLeft = count - 1
        pong ! Ping
        while (true) {
            receive {
                case Pong =>
                    if (pingsLeft % 1000 == 0) println("Ping: pong")
                    if (pingsLeft > 0) {
                        pong ! Ping
                        pingsLeft -= 1
                    } else {
                        println("Ping: stop")
                        pong ! Stop
                        exit
                    }
            }
        }
    }
}
class Pong extends Actor {
    def act = {
        var pongCount = 0
        while (true) {
            receive {
                case Ping =>
                    if (pongCount % 1000 == 0)
                        println("Pong: ping " + pongCount)
                    sender ! Pong
                    pongCount += 1
                case Stop =>
                    println("Pong: stop")
                    exit
            }
        }
    }
}
object PingPong extends Application {
    val pong = new Pong
    val ping = new Ping(100000, pong)
    ping.start
    pong.start
}
Implementation

- Uses a thread pool
- Actors don’t block each other
- But doesn’t scale to more than a few thousand actors
Event-based actors

- Scala also support event-based actors
- Similar to Erlang processes
- Scales to millions of actors

- But: one actor can block all other actors running in the same OS thread
Event-based actors

- react { ... }

- event-based version of receive

- react never returns
  - must call another function (it’s tail recursive)

- Wrap in loop
  loop { react { ... } }

  ==

  f = { ... }

  react { f; loop { react f } }
Receive vs. react

• An actor waiting on `receive` occupies a thread
• If it never receives anything, that thread will never do anything (except take up resources)

• An actor on `react` does not occupy any thread until it receives something
• Once it receives something, a thread gets allocated to it, and it is initialized in it
Receive vs. react

• A receiving thread is expected to return something, a reacting thread is not.
  • So the previous stack state at the end of the last react can be, and is, wholly discarded.
  • Not needing to either save or restore the stack state makes the thread faster to start.

• Having too many threads in Java is not a good idea.
• But, faster to receive than to react
  • So if you have actors that receive many messages but do very little with it, the additional delay of react might make it too slow
class Pong extends Actor {
    def act = {
        var pongCount = 0
        loop {
            react {
                case Ping =>
                    if (pongCount % 1000 == 0)
                        println("Pong: ping " + pongCount)
                    sender ! Pong
                    pongCount += 1
                case Stop =>
                    println("Pong: stop")
                    exit
            }
        }
    }
}
Summary

• Actors
  • message-passing based concurrency model
  • actors are isolated
• Erlang
  • functional language with actor model
• Scala actors
  • “port” of Erlang actors to Scala
  • two implementations: thread-based, event-based