Erlang Concurrency and Distribution

Lars-Åke Fredlund
2022

Master in Metodos Formales

Universidad Politécnica de Madrid

Concurrency and Distribution?

According to Wikipedia:

- **Concurrent computing** is a form of computing in which several computations are executed concurrently – during overlapping time periods – instead of sequentially, with one completing before the next starts.

POLITÉCNICA

Concurrency and Distribution?

According to Wikipedia:

- **Concurrent computing** is a form of computing in which several computations are executed concurrently – during overlapping time periods – instead of sequentially, with one completing before the next starts.
  - *Multiple activities happening at the same time*

POLITÉCNICA

Concurrency and Distribution?

According to Wikipedia:

- **Concurrent computing** is a form of computing in which several computations are executed concurrently – during overlapping time periods – instead of sequentially, with one completing before the next starts.
  - *Multiple activities happening at the same time*
- A **distributed system** is a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another from any system.

Concurrency and Distribution?

According to Wikipedia:

- **Concurrent computing** is a form of computing in which several computations are executed concurrently – during overlapping time periods – instead of sequentially, with one completing before the next starts.
  - *Multiple activities happening at the same time*
- A **distributed system** is a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another from any system.
  - *Multiple activities happening at different places*

POLITÉCNICA

Concurrency and Distribution in Erlang: Highlights

1. An **intuitive programming model** for concurrency – matching some common sense intuitions we have as humans
2. Using **concurrency is efficient**: concurrent programs can use available processing power well (e.g. making efficient use of multiple processor cores in a CPU)
3. **Minimizing the differences between distributed and concurrent programming** – the same programming constructs are used for both

An intuitive programming model: **Actors** (Carl Hewitt)

- A concurrent system is composed of *Actors*
- An actor is a computation entity that can:
  - create new actors
  - send messages to other actors
  - act upon messages received from other actors, when the actor itself chooses to do so
- Compare with normal human behaviour. . .

POLITÉCNICA

An intuitive programming model: **Actors** (Carl Hewitt)

- A concurrent system is composed of *Actors*
- An actor is a computation entity that can:
  - create new actors
  - send messages to other actors
  - act upon messages received from other actors, when the actor itself chooses to do so
- Compare with normal human behaviour...
- And compare with Java

POLITÉCNICA

An intuitive programming model: **Actors** (Carl Hewitt)

- A concurrent system is composed of *Actors*
- An actor is a computation entity that can:
  - create new actors
  - send messages to other actors
  - act upon messages received from other actors, when the actor itself chooses to do so
- Compare with normal human behaviour...
- And compare with Java

The basic ideas of Actors have been implemented in different programming languages and libraries, for instance:

- *Akka* for Java and Scala
- *CAF* for C++
- *Thespian* for Python
- ...

POLITÉCNICA

Actors in Erlang

An actor is a....

POLITÉCNICA

Actors in Erlang

An actor is a.... **process**

- A process has a unique *name* – its **process identifier** or **pid**
- A process executes a function call $f(e_1, \ldots, e_n)$. When the function call terminates, the process terminates.
- A process has a **mailbox** where messages sent to the process are stored. The mailbox is a queue – messages received earlier are placed before messages received later.

Note that the Erlang shell erl executes all code inside a process.

Interacting with processes in Erlang

- **What is my pid** (process identifier)?

```
self()
```

- **Create a new process** executing 1+2

```
spawn(fun () -> 1+2 end)
```

- **Sending** a message `{hello, world}` to a process with pid Pid:

```
Pid ! {hello, world}
```

  - sending is *asynchronous, non-blocking*

- **Retrieving a message** from my mailbox:

```
receive
  X -> X
end
```

POLITÉCNICA

The receive statement

receive
  pat_1 **when** $guard_1$ **->** $expr_1$
  ...
  pat_n **when** $guard_n$ **->** $expr_n$
  after *time* **->** *expr*
end

- The **receive** statement removes a message from the mailbox
- A removed message has to:
  - match a clause pattern,
  - with the clause guard true,
  - *and no older message can be removed*
- **receive** returns the value computed by the expression in the **first** matching clause
- If no message can be removed the receive statement waits (potentially forever) for the reception of a removable message by the process
- The optional timeout clause labelled with **after** specifies a wait timeout

POLITÉCNICA

Receive – examples

Given a receive statement

```
receive
  {inc, X} -> X+1;
  Other -> wrong
end
```

and the queue is `{inc,3}` · `"a"` where `{inc,3}` is the oldest message – what happens?

POLITÉCNICA

Receive – examples

Given a receive statement

```
receive
  {inc, X} -> X+1;
  Other -> wrong
end
```

and the queue is `{inc,3}` · `"a"` where `{inc,3}` is the oldest message – what happens?

- 4 is returned
- The resulting queue is `"a"`

POLITÉCNICA

Receive – examples

Given a receive statement

```
receive
  {inc, X} -> X+1;
  Other -> wrong
end
```

and the queue is `"b"` · `{inc,3}` · `"a"` where `"b"` is the oldest message – what happens?

POLITÉCNICA

Receive – examples

Given a receive statement

```
receive
  {inc, X} -> X+1;
  Other -> wrong
end
```

and the queue is "b" · {inc,3} · "a" where "b" is the oldest message – what happens?

- wrong is returned
- The resulting queue is {inc,3} · "a"

POLITÉCNICA

Receive – examples

Given a receive statement

```
receive
  {inc, X} -> X+1;
end
```

and the queue is "b" · {inc,3} · "a" where "b" is the oldest message – what happens?

POLITÉCNICA

Receive – examples

Given a receive statement

```
receive
  {inc, X} -> X+1;
end
```

and the queue is `"b"` · `{inc,3}` · `"a"` where `"b"` is the oldest message – what happens?

- 4 is returned
- The resulting queue is `"b"` · `"a"`

Receive – examples

Given a receive statement

```
receive
  {inc, X} -> X+1;
end
```

and the queue is "b" · "a" where "b" is the oldest message – what happens?

POLITÉCNICA

Receive – examples

Given a receive statement

```
receive
  {inc, X} -> X+1;
end
```

and the queue is "b" · "a" where "b" is the oldest message – what happens?

- The receive statement waits until a matching message is received (potentially forever)

POLITÉCNICA

Receive – examples

Given a receive statement

```
receive
  {inc, X} -> X+1;
  after 1000 -> {wrong, timeout}
end
```

and the queue is "b" · "a" where "b" is the oldest message – what happens?

POLITÉCNICA

Receive – examples

Given a receive statement

```
receive
  {inc, X} -> X+1;
  after 1000 -> {wrong, timeout}
end
```

and the queue is "b" · "a" where "b" is the oldest message – what happens?

- The receive statement waits at most 1000 milliseconds until a matching message is received, and if no message is received returns `{wrong, timeout}`

POLITÉCNICA

Exercise

How can we implement a function `sleep(time)` which sleeps for `time` milliseconds?

POLITÉCNICA

Exercise

How can we implement a function `sleep(time)` which sleeps for `time` milliseconds?

```
sleep(time) ->
  receive
    after Time -> ok   % or some other return value...
  end
end.
```

POLITÉCNICA

Exercise

- Read either the message `"a"` or the message `"b"` from the mailbox.
- If there is a message `"a"` always read that first.
- Only if there is no message `"a"` the message `"b"` may be read.
- If neither `"a"` nor `"b"` can be read wait until they can.

Exercise

- Read either the message "a" or the message "b" from the mailbox.
- If there is a message "a" always read that first.
- Only if there is no message "a" the message "b" may be read.
- If neither "a" nor "b" can be read wait until they can.

```
receive
  a -> ...
  after 0 ->
    receive
      a -> ...
      b -> ...
    end
end
```

POLITÉCNICA

Concurrency – Extra Features: Naming a Process

- We can register a symbolic name for a process using **register**(Pid,Name)
- The symbolic name can be used when sending messages:

```
spawn(fun () ->
            register(server,self()),
            receive
              Msg -> io:format("Hola ~p",[Msg])
            end
          end).

 server ! "pablo".
```

POLITÉCNICA

Communication Guarantees

- If a process P first sends the message m_1 to Q, and then sends the message m_2 to Q, what can we as programmers assume?

POLITÉCNICA

Communication Guarantees

- If a process P first sends the message m_1 to Q, and then sends the message m_2 to Q, what can we as programmers assume?
  - can the messages be dropped (i.e., never arrive at Q)?

POLITÉCNICA

Communication Guarantees

- If a process P first sends the message m_1 to Q, and then sends the message m_2 to Q, what can we as programmers assume?
  - can the messages be dropped (i.e., never arrive at Q)?
  - can the messages be duplicated (i.e., m_1 arrives twice at Q)?

POLITÉCNICA

Communication Guarantees

- If a process P first sends the message m_1 to Q, and then sends the message m_2 to Q, what can we as programmers assume?
  - can the messages be dropped (i.e., never arrive at Q)?
  - can the messages be duplicated (i.e., m_1 arrives twice at Q)?
  - can they be corrupted? (i.e., Q receives ${m_1}'$ instead of m_1)

POLITÉCNICA

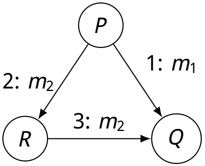
Communication Guarantees

- If a process P first sends the message m_1 to Q, and then sends the message m_2 to Q, what can we as programmers assume?
  - can the messages be dropped (i.e., never arrive at Q)?
  - can the messages be duplicated (i.e., m_1 arrives twice at Q)?
  - can they be corrupted? (i.e., Q receives ${m_1}'$ instead of m_1)
  - what is the order in which Q receives these messages?
- Guarantee: **messages sent from a process P to a process Q are delivered in order (or P or Q crashes)**
- Concretely: if P sends first m_1 and then m_2 to Q, then first m_1 will be stored in the mailbox of Q, and then m_2 will be stored in its mailbox.

Communication Guarantees II

Consider the following situation:

1. P sends a message m_1 to Q
2. P next sends a message m_2 to R
3. R forwards the message m_2 to Q



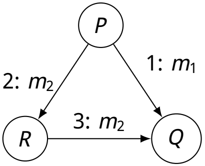
In which order will Q receive the messages m_1 and m_2?

POLITÉCNICA

Communication Guarantees II

Consider the following situation:

1. P sends a message m_1 to Q
2. P next sends a message m_2 to R
3. R forwards the message m_2 to Q



In which order will Q receive the messages m_1 and m_2?

- We don't know - either m_1 or m_2 will arrive first in the mailbox of Q
- Mimics TCP/IP communication guarantees

POLITÉCNICA

Handling Errors – Process Termination

A process can die for different reasons:

- It can terminate *normally* – e.g.,

```
spawn(fun () -> fib(10) end)
```

- It can terminate *abnormally* because of an exception which was not handled:
  - Common mistakes: `2/0`
  - Signalling errors: `error("do not do that")`
  - Non-caught throws: `throw(hola)`

- Or it can terminate *abnormally* if the process was ordered to die:

```
exit(Pid,Reason)
```

POLITÉCNICA

Handling Errors: Links

- Suppose P spawns a new process Q, and Q dies *abnormally*, what happens to P?

POLITÉCNICA

Handling Errors: Links

- Suppose P spawns a new process Q, and Q dies *abnormally*, what happens to P?
  - By default absolutely nothing

POLITÉCNICA

Handling Errors: Links

- Suppose P spawns a new process Q, and Q dies *abnormally*, what happens to P?
  - By default absolutely nothing
- Processes can be *linked* together using a call **link**(Pid):

```
Pid = spawn(fun () -> ... end),
link(Pid).
```

- The process executing the spawn call will be *linked* to the new spawned process

POLITÉCNICA

Handling Errors: Links

- Suppose P spawns a new process Q, and Q dies *abnormally*, what happens to P?
  - By default absolutely nothing
- Processes can be *linked* together using a call **link**(Pid):

```
Pid = spawn(fun () -> ... end),
link(Pid).
```

- The process executing the spawn call will be *linked* to the new spawned process
- We can spawn and link at the same time – **atomically** – using

```
Pid = spawn_link(fun () -> ... end)
```

POLITÉCNICA

Handling Errors: Links and Abnormal Termination

Suppose we execute the following in process P:

```
Pid = spawn_link(fun () -> ... end)
```

- What happens if the spawned process dies *abnormally*?

Handling Errors: Links and Abnormal Termination

Suppose we execute the following in process P:

```
Pid = spawn_link(fun () -> ... end)
```

- What happens if the spawned process dies *abnormally*?
  - **By default** P **dies** ***abnormally*** **too!**

POLITÉCNICA

Handling Errors: Links and Abnormal Termination

Suppose we execute the following in process P:

```
Pid = spawn_link(fun () -> ... end)
```

- What happens if the spawned process dies *abnormally*?
  - **By default P dies *abnormally* too!**
- What happens if the spawned process dies *normally*?

POLITÉCNICA

Handling Errors: Links and Abnormal Termination

Suppose we execute the following in process P:

```
Pid = spawn_link(fun () -> ... end)
```

- What happens if the spawned process dies *abnormally*?
  - **By default** P **dies *abnormally* too!**
- What happens if the spawned process dies *normally*?
  - By default nothing happens to P.

POLITÉCNICA

Handling Errors: Links and Abnormal Termination

Suppose we execute the following in process *P*:

```
Pid = spawn_link(fun () -> ... end)
```

- What happens if the spawned process dies *abnormally*?
  - **By default *P* dies *abnormally* too!**
- What happens if the spawned process dies *normally*?
  - By default nothing happens to *P*.
- Note that links are *bidirectional*. Suppose *P* and *Q* are linked:
  - if *P* dies abnormally so does *Q*
  - if *Q* dies abnormally so does *P*!

Handling Errors: Links and Abnormal Termination as Messages

- As an option the spawned process can be *notified* instead of terminated when a linked process terminates

POLITÉCNICA

Handling Errors: Links and Abnormal Termination as Messages

- As an option the spawned process can be *notified* instead of terminated when a linked process terminates
- Notifications are received as normal messages in the mailbox on the format

```
{'EXIT', Pid, Reason}
```

  where `pid` is the pid that terminated and `reason` is the reason for termination
- The notification behaviour is chosen by calling `process_flag(trap_exit,true)`:

```
process_flag(trap_exit,true),
Pid = spawn_link(fun () -> ... end).
```

POLITÉCNICA

Concurrency – Extra Features: Monitors

- *Monitors* behave like links, **except** they:
  - are unidirectional
  - **always** notifies (sends termination messages) – **never** propagates abnormal termination

```
Pid = spawn(fun () -> 2 end),
monitor(Pid)
```

POLITÉCNICA

Concurrency in Erlang is Efficient

- Pros:
  - It is cheap (quick, uses little memory) to spawn processes
  - It is quick to switch which process is executing on a processor core
  - The concurrency model maps well onto todays hardware: makes use of multiple processor cores with *no programming overhead* and *little computing overhead*

Concurrency in Erlang is Efficient

- Pros:
  - It is cheap (quick, uses little memory) to spawn processes
  - It is quick to switch which process is executing on a processor core
  - The concurrency model maps well onto todays hardware: makes use of multiple processor cores with *no programming overhead* and *little computing overhead*
- Cons:
  - Sending large messages between processes is not cheap (copying is necessary)
  - Thus data structures shared between multiple processes are sometimes used (**the horror!**). Example: the `ets` term storage library.

POLITÉCNICA

Concurrency in Erlang is Efficient

- Pros:
  - It is cheap (quick, uses little memory) to spawn processes
  - It is quick to switch which process is executing on a processor core
  - The concurrency model maps well onto todays hardware: makes use of multiple processor cores with *no programming overhead* and *little computing overhead*
- Cons:
  - Sending large messages between processes is not cheap (copying is necessary)
  - Thus data structures shared between multiple processes are sometimes used (**the horror!**). Example: the `ets` term storage library.
- In practice is realistic to have systems with up to 100,000 alive processes

Support for Distribution in Erlang

- Erlang "nodes" – Erlang runtime systems – can be connected. API:
  - Provide a name for the erl runtime at startup: "`erl -sname nodename`"
  - **`node`**`()` – returns the node name
  - `net_adm:ping(nodename)` – connect to the remote node `nodename`
- New processes can be spawned at a connected remote node:
  `Node:spawn(nodename, fun ()->... end)`
- **Process-to-process communication, links, etc, work between processes at different nodes with no syntactic differences!**

POLITÉCNICA

Support for Distribution in Erlang: Pros and Cons

- Positive
  - a very simple and powerful programming model

Support for Distribution in Erlang: Pros and Cons

- Positive
  - a very simple and powerful programming model
- Negative
  - communication between nodes is not encrypted
  - performance is realistic for only small networks of nodes. The network is *fully connected*: every node has an explicit connection to every other node in the network
  - difficulty of integrating non-Erlang nodes in the network

Support for Distribution in Erlang: Pros and Cons

- Positive
  - a very simple and powerful programming model
- Negative
  - communication between nodes is not encrypted
  - performance is realistic for only small networks of nodes. The network is *fully connected*: every node has an explicit connection to every other node in the network
  - difficulty of integrating non-Erlang nodes in the network
- In practice
  - used only for **small internal networks**, e.g., to distribute load between a group of servers
  - often non-Erlang "middlewares" are used to enable communication between Erlang programmed nodes (RabbitMQ, . . .)

POLITÉCNICA