Note slide Distributed systems 5

Moving from Sequential to Concurrent

We already spoke about how to make sequential an execution which is inherently concurrent (topological sorting, Kahn's algorithm).
Now we will do exactly the opposite.

So far we imagined our programs as sequential ones. So, we will try to decompose such programs in separate units and we will try to take advantage of the fact that different actions could be carried out by independent ideal programs that communicate among each other, with one another.
So, we are going to do the opposite of topological sorting.
We start with a first idea of a sequential program, and we will try to break it down into different pieces.
What we have to do is to understand how these pieces could coordinate to obtain the final result.

Erlang is basically a concurrent language, and when you think about how to solve a problem, it should be natural to decomposing it in different portions and to identify a way to make the different parts of our concurrent program communicate and coordinate.

Erlang adopts the message passing paradigm --> the information is passed by means of messages, nothing is shared, everything is passed.
In this perspective we may have several advantages in organizing our solution this way, because we do not have to deal with locking or something like that, or any kind of concurrent access.

Erlang Concurrency & Distribution

We will see how to deal with concurrency and distribution.
What does it mean distribution?

We can imagine our programs as hosted by one single machine able to implement the abstraction of different computing units, but in practice these different computing units could be spread over many different nodes, geographically separated, relying on the communication infrastructure these machines typically use to communicate.

We will see the programming abstraction, so we will not deal with networking problems, we have just to think about how to solve our programming problems, relying on the means that the language gives us, on the provided conceptual tools it gives us.

Concurrent, but How? The Actor Model

Of course, there are many different possible ways to organize a concurrent program, and historically many models have been proposed, there has been a lot of theoretical investigation on this field.
It has been shown that some models have different power in terms of what can be done, and many others have been shown to have exactly the same expressive power.

We want to present one particular model proposed for concurrent computation which is known as the Actor Model (proposed at the beginning at the 70ths).

The concurrency model adopted by Erlang is really similar to this one.

- In this model the fundamental computational unit is named ACTOR.
- Any computation involves at least one actor, but typically more than one.
- The underlying principle is: NOTHING IS SHARED.
  So, one actor has its own pieces of information, and such information is not shared with any other actor, but an actor can communicate with others by sending messages any time it wants.
- Messages are exchanged in an asynchronous fashion --> no notion of shared clock or shared time: you just know that you send a message and you have no idea when it will be received, for sure it will be received later, this is the only guarantee we have.

What about the time it takes for a message to be delivered ?

We do not know. We know that eventually it will be delivered. We don't even know about possible upper bounds of this delay time.
Of course, this is a sort of ideal situation, often in practice we will have to deal with a reasonable upper bound for an app to be practically usable. This is the reason why additional constraints in practice have been used to make the model more useful.

One problem of the ideal model in any type of model is the addressing problem:
how to identify the different actors in the system ?

from a theoretical point of view it is pretty simple, because we have just ID's, typically Actor IDs are integer numbers, or something which is equivalent.
There is a problem:
for one actor to send a message to another actor, the first one has to know the identity of the other one.

How is it possible to create actors?

Actors are created by other actors --> in the actor model we do not have a fixed number of computational units. We may have more or less computational units along a single computation because actors can be created and some others may terminate before the overall computation is finalized.
So, the overall system has a variable topology in terms of computational units, where topology means the graph whose nodes are the actors, and the edges are the communication channels (let's call them this way).

Moreover, the actors follow a reactive behavior --> actors, to do something, have to receive a message and, upon the inspection of the content of the message, may decide, according to a certain program logic, to do perform some actions. So, all the actions are a consequence of the reception of a message.

Re-acting Actors

The behavior of an actor is what an actor actually does in processing a received message.
There are several possibilities in the theoretical model that an actor could do upon receiving a message:

- Send a finite number of messages to other actors.
- Create a finite number of new actors.
- Decide to change the behavior to be used for the next message(s) that will come.
  Note that the current behavior depends only on what happened in the previous step, not in, say, 10 steps before.

This model is the inspiring one for Erlang.
Now we have to see how the receive operation is actually structured.

Erlang Ingredients for Concurrency

We have seen the actor model saying that it inspired Erlang, but in practice when the Erlang concurrency model was proposed, the designers of the language did not know the actor model. So, Erlang follows a version of the actor model:

in Erlang actors are named processes, even if processes here have not the same mean as the processes of operating systems, they are just computational units handled directly by the Erlang virtual machine.

To deal with concurrency in the language we have only 3 separate constructs:

- Create a process --> spawn() BIF.

When creating a process, one of the most important things is to define what a process is required to do, so the behavior of a process.
Erlang is a functional language --> if you want to describe what a process has to do, you can do it by specifying a function.
So, what a process will do is just an evaluation of a function, and you have to provide as an argument to the spawn() BIF a function.

When you call a function, the evaluation of a function call corresponds to some term.
In the case of the spawn() function, what it is returned is the so called PID, that is the identity of the process, and this PID is what it is required in addressing it, so it is very important, because we will use this value of a special data type to identify the process and to send it some message.

- Once we are able to crate different processes and we know how to identify them (PIDs), we need to understand how to support the sending of a message.
  It is really simple:

Sending message --> ”!” Called the bang operator.

Let’s try to send a message:

self() ! emptymessage:

this is an operation that corresponds to send a message to ourselves (self())).
Suppose you are a process, you are able to spot out your identity by calling the BIF named self(), which will give you your own PID. emptymessage is an atom, so writing this operation the system will take the term after the bang operator (!), and this will be what will be sent in the message to, in this case, self().

We still need something that let us to receive a message:
we want to take the message and we want to be able to inspect what has been inserted in that message.

- Receiving messages --> receive … end.

As usual, at the language level this is seen as an expression, so receiving a message corresponds to the evaluation of an expression.
In receive we will exploit pattern matching exactly in the same way we did for the case expression, thus it is possible, thanks pattern matching, to select the message to be received and processed, because at a certain point in time you may want to process some messages instead of others.

How can you distinguish between different messages?

Just by inspecting their structure. The content of a message actually is an Erlang term, maybe it could be a tuple containing some values and such values may be required to be in some ranges and so on.

Some messages reach the destination process, but they are not received because no matching actually occurred in the receive operation. These messages will stay there waiting to be received, since maybe later for some reasons they may become eligible to be received.

Spawning of Processes

We will now describe each of the Erlang ingredients for concurrency in detail.

Spawning of process.

See the example:

We have a module test, and we have a simple function body printing on the screen the 2 arguments and their sum.
The function will be used as the body of an Erlang process.

There are 2 different ways to specify a process body:

1. A version of spawn with arity 1:

In this case the argument has to be a fun.

2. MFA: Module Function Arguments. In this case we have a version of spawn() with arity 3.

The arguments are the module and the function, indicated as atoms, and a list of arguments for the function.

Why we have these 2 different ways?

because these 2 different ways correspond to 2 different mechanisms used by the underlying system to deal with the process spawning.

The second way has been introduced to support the so called dynamic code loading:
f.i. if you decide to change the implementation of a function that corresponds to the body of a process, and you have your program running, you may decide to insert in the system the new version of your process body, so the actual code inside is different, is an updated version, and if you specify that the processes from that point on are spawned using the MFA method, the system will look in the beam file with the new version and all the processes with that body from that point on will be spawned with the new updated version of the body.
So, you do not need to stop your system to insert updates, modifications and so on.

We said that spawn returns a PID, which is a special data type in Erlang containing 3 fields:

- Indication of the node, in case your program is actually a distributed one, where the process is located.

- Then you have a numerical ID split into 2 different fields.
  Usually with a small amount of processes only the first of these 2 fields is actually used, the last one will be 0. Of course the system will take care of handling automatically the values within a process ID.

How to specify the end of a process?

A process life ends as the function call will end, so it is up to you to specify, by means of that function, what the process is required to do and how.
You could also specify a never-ending process, if you specify a function that as the last step will call itself. In this case the process will loop and, otherwise not specifically written in the program, it will never stop.
Once a process reaches the termination, it is no more handled by the system.

Let’s look at the example on how to spawn a process:

- Line 1:
  we define a fun that corresponds exactly to the body function.

- Line 2:
  I can spawn a certain number of processes using a fun as we said. In this line we spotted a

trick:
we create a list comprehension, so all the element in this list will be obtained by working on what is reported after "||".
So, this list comprehension says: "consider the list containing 3 and 4, and let the variable A assume the values of the elements in that list (first 3, then 4).".

We want, at each step, do what is specified before the "||" --> we call spawn --> an element of this list will correspond to the PID of the spawned process.
We have to specify the body of the process, so it will be a fun corresponding to the call of F (previously specified at line 1) with as arguments A and A+1.
So, 2 different processes will be spawned:
-- the first one will show the result of summing 3 and 3+1;

-- the second one the sum of 4 and 4+1.

The 2 dots with the circular arrow mean that in this operation 2 different processes have been spawned and actually run.
Do they still run?

No, because they just performed one operation and they came to the end.
But we know what their PIDs were (0.134.0 and 0.135.0).

- Line 3:
  I spawn another process.
  Here we adopt the second way: MFA.

Module is test, Function is body, Arguments will be 1 and 2.
The PID of the spawned process was 0.137.0.

This process could be interested in keeping some information useful for its own computation.
Is there any way to easily do this kind of information storing?

Yes. In Erlang each process has its own dictionary (an associative array), and you can insert values by specifying a key to retrieve a single value, and there exist a predefined BIF to work with process dictionary: get, put, get_keys, erase.
According to the basic principle of functional programming it should be avoided to keep state in variables, so even if you can do it, please try to use the dictionary as a place where to store values that are written just once, according to the general way Erlang variables are used.

Messages, and Sending thereof

We will talk now about communication among processes.
In Erlang communication is performed by asynchronous signaling.
Signals are exchanged between processes.
The most common type of signal is the message.

A message has both a recipient, identified by a PID, and a content.
The content of a message could be any Erlang term.

One important point is the specification of the guarantees about the delivering order for messages between processes. There is only one simple ordering guarantee in Erlang: Point to Point FIFO --> if we have 2 different processes A and B, and A sends to B first message M1 and then M2, the guarantee states that M1 will be received before M2, supposing that both of them are eligible to be received at B.

How to perform a non-blocking send?

We have to specify in the expression the PID then the bang operator “!”, and then the message.

The first part is the specification of the recipient, the second part is the specification of the content.
This ("Pid ! Msg") in Erlang is just an expression (Remember: an expression must evaluate to a term).
In this case, it evaluates to the message itself.
So, the evaluation of this expression corresponds to sending the message and returning the term corresponding to the evaluation of the message itself (on the right side of the bang operator).

Send is right-associative:
if I find PID2 ! PID1 ! PID0 ! examplemsg --> the message is sent to PID0, this is evaluated and the evaluation of this one corresponds to examplemessage, and now it will be sent to PID1, and so on.

...and Other Signals…

There are other signals, but this is not the right time to talk about them since they are used for the coordination of particular activities between processes.

- Exit:
  used to send special notification to processes and also for killing processes.
- Link/unlink:
  used for setting up a bidirectional link between processes.
- Monitor/Demonitor:
  used to handling other kind of monitoring operations with unidirectional channels

Receive Machinery

How to receive a message?

We said the send operation is non-blocking.

Instead receive is a blocking operation.

Every process has a special queue named mailbox to hold incoming messages.

The picture indicates that all the incoming messages are ordered according to the relative arrival time.

The content of a mailbox can be inspected by using a special BIF named process_info():
used to get information on a process and you have first to specify what process you want to get information on through the PID, and then what type of information through an atom(in the picture case we are interested in the messages in the mailbox).

To extract messages from the mailbox we have to use the receive expression.

See the example:

Inside we have several clauses, and each one is made of a pattern (P1, Pn), you can optionally add guards, finally you have an expression to be evaluated.

So, you go through the different patterns and check them sequentially if they match some message.
All the messages are explored from the oldest to one the newest one, and in case the match is actually present, the guard is evaluated;
if both are satisfied --> the corresponding expression Ei is evaluated and the evaluation of the overall receive expression, corresponds to the evaluation of the chosen expression Ei --> the message will be extracted from the mailbox and it will be used, it is actually received.

Suppose that no messages in the mailbox is eligible to be received:
we stop here, we wait for a message to come so that will match a pattern in the receive expression and the corresponding guard.

Example: Ping Pong (code)

Let's see a simple example where these constructs are used.

The idea is:
setup 2 different processes (Alice and Bob) that exchange messages back and forth.
We call Alice and Bob the functions corresponding to the body of such processes.

This program is a way to show how to exchange N messages, where N is a parameter.

We want Alice to start the operations.

We want to specify the behavior of each process. We can do it by writing down a function for each process.

Alice has to send out a certain number N of messages, and this number will be a parameter for the function. But Alice might do not know the counterpart, she must know how to find Bob to send messages to it. We need to provide Alice with the PID of Bob, so this is another parameter of the function.

Suppose we want to send N messages, immediately Alice send a message to Bob. We need to specify how to structure the message itself, what info to insert in the message.
In this case we just insert an atom, for example ping_msg, but moreover I may be interested in

letting the other know who I am. This is very important in general, you may decide to accept messages only from some special processes. One possible way to exploit pattern matching in the receive is just to select the good messages to receive.
We have to organize the message as a simple Erlang term. The simplest way to do it is to group up both the ping_msg atom and the PID of the sender in one single tuple.

The receiver has to know the structure of the message, there should be a sort of agreement at the program level between the 2.

After sending the message, Alice expects to receive it back and to send back a reply --> the next operation to do is receiving the reply.

We said that in the ping message there is the atom ping_msg;
in the pong message we suppose that they contain the pong_msg atom, so I want to receive something that is structured as a single atom named pong_msg, writing then on the screen something.

After this has been done, we want a certain number of messages to be exchanged in N rounds: the simplest way to do this is to make Alice repeat again this function but specifying as argument N-1.

At a certain point of time this N-1 will become 0 and this situation has to be catch because in that case we want Alice to terminate, that's the reason we have the first part of the function alice/2 that matches 0 as the first argument and the Other_PID as the second.
In this case Alice sends Bob a special message (finished atom) and writes something on the screen.

On the right we have what Bob should do:
It, as it is spawned will block waiting for a message to come.
The possibilities are the following:

- he may receive a message containing just the atom finished --> also Bob has to terminate and maybe writing something on the screen.

- He may receive a tuple where the first part contains ping_msg atom, the other one Bob will keep it in a variable named Other_PID.
  In this case Bob has to reply with a pong message, so he does Other_PID ! pong_msg.

After replying, it has to go back at the beginning.
Bob will reply to any message by any process, provided that the message is structured this way.

How can Bob know the address to send the reply message?

He knows it because it can retrieve it just from inside the message.

How can Alice know where to send her message?

She knows It because it is an argument of the function.
In order to try spawning the 2 processes, we defined a start() function.
By calling it:
Bob is spawned. We used the spawn/3 function, so we specify the module, the function and the

arguments list.
The module is specified through a MACRO, ?MODULE, just letting the system substitute this one with the name of the current module. By evaluating spawn we will get a PID, that is the PID for bob, which will be kept in the variable Pong_PID.
This can be used as an argument for spawning Alice
Alice is spawned again through the spawn/3. Here we have 2 arguments:

2 (the number of rounds), and the ID of bob got from the previous line of code.

Example: Ping Pong (run)

Now we start everything from the shell supposing that we already compiled the module, so we call the start() function.

In this case 2 processes are started, first Bob and then Alice.

At the end the system prints <0.79.0>, what PID is this?
When we have an expression made of subsequent sub expressions, the overall expression evaluates to the term corresponding to the evaluation of the last subexpression, that is the last line, so that is the Alice address returned by the last spawn in the start() function.

Basic Client-Server (I)

We can use this very simple syntax to introduce Client Server computing.

We want to catch the basic notion of a client server system using just few lines and few constructs.

A server process is intended to perform maybe repeatedly, the operation listed here:

- Wait for a request
- Suppose a request comes, at that time it has to compute the answer.
- Once the answer has been computed, it has to be used to reply back to the process that asked for that kind of service.

The basic code for a server is shown in the box in the slide.

We have to agree with the client on the way we structure our messages, and we have to understand what kind of information has to be inserted in the messages that must be exchanged.

First of all, we don’t need just one single piece of information but more, so the most trivial way is to make use of tuples.

Moreover, when a process asks for something to a server, the server has to be able to reply back so it has to know where to send the reply.
How can the server know about this ?

The client has to specify its own PID as part of the message content, so we suppose that our message will be organized as a tuple whose first element is the PID of the client, and the second the payload organized as a tuple.
We will see why we want the second element to be organized as a tuple again.

We write the code in a module called server and we organize it such that every time a message is processed, at the end it will become available again to process yet another request, so we will call the body of the server loop(), because usually a server behave like this.

Let's see how to organize the loop:

- Wait for a request:
  write a receive expression. We know that receive exploits pattern matching.

We want to organize the server to provide not just 1 operation but maybe 2.
This server is able to perform not only addition, but also subtraction.

How can we subscribe what service we want as a client ?

We have just to write it in the message, so the second part of the message should specify what service we want to have from the server, so in the first element of the tuple we specify the operation (plus or minus), and then the operands of the operation.

The server receives a message. It will only process messages structured the way specified in the receive operation, the others will be left there in the mailbox.

So, first of all it has to retrieve the info about the address of the client requiring the service, then it has to understand whether the requested service is plus or minus --> there are 2 different patterns, so by using pattern matching the correct clause corresponding to the required service will be chosen.

The computation of the answer is pretty simple in this case:
Reply back with the answer.

First of all I need to know the address of the client, but I know it because it was in the incoming request, and it is in variable From, so I will send to From a message made this way:
a tuple with the indication of my self and the result.

Why specify self()?

It depends, because for example the client should not be fooled by other processes that try to send a message to it instead of the real server.

What to do once the service has been provided?

It has to become ready again to answer to another request, so I call loop() again in both cases.

In these few lines you have the essence of a client-server system.
Here you see the reason the body of a server is typically the server loop.

Basic Client-Server (II)

What about the client?

We will develop also a module for the client.

Let's see the body() of the client:
in this case I just do a specification of the body function with as arguments the server ID, the operation ID and the arguments to be passed to the server.
This will be the body of a client who in his life will only ask 1 question and get the reply, nothing else.

The client has to send the request.
As for the address of the server, it must be known just from the beginning, so it will send the request, where it specify self() (its own PID), the operation ID and the arguments (all these come from the function arguments).

Once the request has been sent, it has to block on the receive for a reply.
So, it will check that the reply comes exactly from the contacted server and not from other processes.

Once it gets the result it only prints a message on the screen.

First of all, we have to start the server (77 is the PID of the server), then we start the client telling the operation and the arguments and the server ID (79 is the PID of the client).

If I want to stop the server I can make use of a signaling mechanism towards that process by executing exit towards S (PID of server) and specifying the operation I'm asking, kill.

Basic Client-Server (III)

Let's try to make everything simpler, trying to abstract away some details in the communication on the client side, you can do it only on client side.

Let's think about what happens:
there is an operation which is a request of service, and that's a communication;
and then another communication from the server back to the client.

I can imagine grouping up these 2 operations into one operation, and this kind of abstraction will end up with the definition of a sort of synchronous communication construct.

Why synchronous?

Because I stop there, and I wait for all the communication to be done according to the specified rules.

I will call this abstracted operation RPC (Remote Procedure Call) for simplicity.
I insert here what is required to send the request and to have back the answer, and grouping up this I have to specify as arguments the server ID and the message, nothing else.

By defining rpc() we can rewrite the body() this way:

the information here is the same but I can immediately get the result by calling rpc of server ID and what I have to do.
This is much simpler to read and to understand, and this turns to be a better structuring for our program.

From now on, if you want to take advantage of this service you have just to call one function, rpc.

Basic Client-Server (IV)

We have just another issue to deal with:

how to deal at the server side with incoming messages with unexpected format ?

Suppose that someone else send messages to my server with a different format. If all the messages are kept in the mailbox for possible future evaluation, the content of my mailbox will grow and grow, and at the end this will result in a sort of DoS, all my memory will be filled up. So it is important to get rid of unwanted messages.

Of course, we can add also another pattern in my receive matching to any possible message. Since all the patterns are checked sequentially, if you insert at the end a pattern that will match everything, once a message arrived there all the previous one have already been checked you are sure the message is not an allowed one.
In this case, since the message will match this last inserted pattern match, the message is removed from the mailbox, but we don't want to do anything with it so we simply go back calling loop again.

Instead of having to kill the server in order to stop it, which is not very polite, I can arrange the behavior of the server to deal also with some special message whose purpose is just to indicate the server that the termination time has come, so I "gently invite it to suicide".

This can be done by choosing a message whose format is a special one, in jargon it is known as Poison pill.

Receive Glitches & Timeouts

So fare we have seen the basic constructs for making processes communicate, but there are some situations that requires additional functionalities from the basic constructs, and in particular we have to say that the blocking behavior of the receive expression may cause some problem because f.i. we may wait for a message that will never come for many different reasons --> we will never go on with our process that waits for this message to come, and thus we would like to be able to guarantee the progression of our computation also in situation like this.

Let's take a look at what might happen (box top right):

I start the server, then I keep in the variable S its PID, then I kill the server. The server is down, but the PID is a well formed one, and it can be used to send messages, so I can try to call rpc()

with S as argument. The server is no more there so the message is lost, but the client waits for a reply that will never come back --> the shell blocks.

This is not so pleasant in many situations.

A possible solution is to adopt a timed version of the receive operation:

let's wait for a reply, but not forever, I will wait for a certain amount of time after which I will do something else.
This behavior can be specified in the receive operation with the after keyword --> after a certain delay (in millisecond) if I have to wait more than this delay the following expression will be evaluated and will be returned as the value corresponding to the evaluation to the overall receive operation.

Under the hood a timeout is used.
After the expiration of that timeout, I will go on returning what is specified in ExprDel.

Example with Timeout = 0

It is possible to take advantage of this new semantic also making use of timeouts whose value is 0.

Look at the example:

I want to write a function to flush all the messages in my own mailbox: I can organize this function, flush_messages(), this way:

I perform a receive, in case a message is in the mailbox I want to take it away.
I need to provide a pattern that will match with any message.
This pattern will be named f.i. _Any.

Why the underscore?

This variable is not used in any other part of the program, so this might be seen a sort of strange situation, but I actually want to do this.

So, just to avoid warning messages from the compiler I name it with an initial underscore, and it is just a way to show in the code that this variable is not used anymore.
So, after matching it I call again this function.

At the end I specify "after 0 --> true", and this will happen when no more messages are in the mailbox.

There are some cases where we can insert infinity as value for the timeout --> the behavior will correspond to the ordinary one, so why inserting it and not simply omitting it?

Actually there are some cases where the time to wait has to be calculated and there might be situations where the variable which is used after the after keyword might assume also the value infinity (so having no timeout at all).

Semantics of Receive, Detailed (I)

Now we have to redefine the semantic of the receive operation, to understand how everything works.

Entering a receive statement will start a timer, in the case the after part is present.

Take the first message in the mailbox and let's try to match it against the first pattern:

if it doesn't match we will try the second pattern and so on;

- suppose a match occurs: as soon as a match occurs, the message is removed from the mailbox and the corresponding expression is evaluated.
- In case of no match for the first message in the mailbox, it is removed from the mailbox and put in another data structure known as save queue.

Then, if present, the second message of the mailbox is tried, and it is processed exactly the same way.
This procedure is repeated until:
a. a matching message is found

or

b. all the message in the mailbox have been examined.

Semantics of Receive, Detailed (II)

If none of the messages matches, the process is suspended, and will be rescheduled for the execution at the time another message is put in the mailbox.
When a new message arrives, the messages in the save queue are not rematched again, only the new one is matched.

As a message match occurs, something has changed, so all the messages in the save queue have moved back in the mailbox, in the same order of before (same order of arrival). In case a timer was started it is cleared.
If the timer elapses when we are waiting for a message --> we do not have to wait anymore, we evaluate the relative expression and we have to move the message from the save queue to the mailbox.

Publishing of Processes

Let's introduce another important concept: publishing of processes.

In Erlang there exists a system repository where processes can be published --> we give a name to processes so that instead of using PIDs we can make use of names (think about DNS).

Once a process has been registered, it will be called a registered process.

Actually this is not like the DNS service because remember that we are dealing with our own program, not something that has to be necessarily exposed to the external world.

What processes should be registered?

Only those that play a special role within our program, and thus it is sensible to making the access by their names.

We can perform the registration and related operation using 4 different BIFs:

- register(AnAtom, Pid):

register the process
-- AnAtom will be the name of the process;
-- Pid is the PID of the process.

- unregister(AnAtom):

we can remove a registration for process AnAtom.

- whereis(AnAtom):

provides a lookup for process AnAtom, giving you back the PID of the process, or the atom undefined if the process is not present in the repository.

- registered():

take a look at all the registered processes in the registry.

What if a registered process terminate?

In this case it is the system that automatically will cleanup the register, removing the corresponding entry.

Example: Registered Server

A server could be a possible target for registration --> the different clients within our app might find very simple locating the server.

Look at the example:

I start the server and I keep its PID in the variable S.
We can then register this PID with the name calc_server.

Now I can use the name of the server, as specified in the registry, in the send() operation, so I can send to calc_server that message at line 3.

The server will reply back to the shell.

How to inspect whether the reply is in my mailbox or not?

One possibility is to use flush() that flushes the mailbox outputting in a tuple the PID of the sender, and the result.

What about using the name of the server, f.i., in the rpc where I was expecting the PID?

we can use it by passing it to the whereis() BIF, which correspond to a lookup in the registry and which returns the PID of the registered server.

Keeping State at Server, Functionally

Often at server we want to keep some info from one iteration of the loop function to the next one.
We know that the concept of program state is not used in functional programming, but in practice I want to keep trace of what's going on from one iteration to the next one.

How can I simulate somehow a sort of state variable to be updated ?

each iteration corresponds by calling the loop function again and again, so one possibility is to use a parameter in the loop function to keep state information.

Often at server we want to keep some info from one iteration of the loop function to the next one. We know that the concept of program state is not used in functional programming, but in practice I want to keep trace of what's going on from one iteration to the next one. How can I simulate somehow a sort of state variable to be updated ?
Each iteration corresponds by calling the loop function again and again, so one possibility is to use a parameter in the loop function to keep state information.

Look at the example:
We have a server that keeps the count of the number of executed loops, so I start it with an argument = 0.
loop() takes an argument which is intended to keep trace of the progression of the execution.
when we do the recursive call to loop(), we call it with N+1. The value N is used also by other instruction within the loop.

Going Distributed, Actually

We talked about processes and how to exchange messages, but all we did so far is confined within one single Erlang virtual machine.
If we are interested in distributed computing we should become able to deploy different portions of our app in different places, and moreover we need to make these separate virtual machines running in the same node or in different nodes cooperate for the purposes of our app.

So we need to understand how to break the border of one single Erlang machine.

Erlang Distributed Applications

There are different ways in Erlang to develop distributed apps.

We will see one of the 2 possible ways, Distributed Erlang, because it is somehow more powerful and at the same time more abstract than the second way:

it let us to apply the same principles seen so far also for actual distributed programming.

The idea is to have the possibility to spawn processes on remote nodes and machines.

In Distributed Erlang programs run on Erlang nodes.

An Erlang node is supposed to be a single full featured Erlang system, and on a single node we will have a specific set of processes, like the ones we seen so far, operating on just one single environment.

There is the possibility to spawn processes also on other nodes, and processes can interact across nodes using the constructs we described last time (send, receive…) .

The problem with this approach is that it is not so secure, and we don’t have so much control on what a remote client can do in a node, that is in the Distributed Erlang we do not take into account security issues.

This can be avoided using socket-based distribution, but as it is closer to the network issues we will skip it.