Note slide Distributed Systems 2

Abstract Models: Synch vs Asynch

We want to discuss now about possible semantics for communication constructs.

Let's again think of distributed systems from an abstract viewpoint.
There are 2 different abstract models we can consider, which refer to different ways I can think processes are executed over the machines composing the overall distributed system:

- Synchronous model:
  whenever processes takes steps simultaneously. We can imagine the overall execution as carried out in synchronous rounds. This is a very simple abstract model to reason about, but it is not realistic:
  in distributed system we lack a global clock for all our nodes.
  So, we can move to a more realistic model:

- Asynchronous model:
  we make no assumptions of relative timing of processes. The delays experienced by processes carrying out their activity, and the delays experienced in delivering messages, are potentially unbounded. In this case we have to deal with an asynchronous setting, where I cannot rely on any form of shared uniform global clock, and this will be complicated because we will have to find out something else to rely on for the ordering of activities, because I cannot trust any form of clock.

Semantics for Send/Receive

Let's talk about possible semantics for SEND and RECEIVE.
Send and receive are considered, at our abstraction level, as primitives, that is at the high-level language abstraction we are discussing, we can consider them as primitives that is the smallest unit of processing, something that is atomic.

What is the possible behavior of them?

They can be either blocking or non-blocking:

- Blocking:
  suppose a process in execution has to perform a blocking op: it will stop there until the op will be completed. Remember that the finalization of the op depends not only by the process itself but it may depend also on what is done by other processes involved in the communication. (waiting op are possible)

- Non-blocking: the op is just fired, it starts and who cares about the completion of the whole op as it regards activities to be carried out by the other parties, I did my part and now I go on, it is up to other parties to do what they have to perform.
  If I want to use this last semantic (non-blocking), from a practical point of view we have to

resort in some kind of buffering (incoming buffer = “mailboxes” in our jargon).
Non-blocking implies the need to implement buffering in the implementation of communication primitives, both for incoming and outcoming messages.

Blocking vs Non-Blocking Send

BLOCKING SEND:
the message has to be sent to the destination, and the sending process can resume the execution only upon getting sure that the message has been received.

How can we be sure of this? Getting back a message.
The implementation of a communication construct like this one will contain a sort of handshake:
I have to be informed by the other that everything has been received correctly. So I hide some complexity in the implementation of the construct and I just present the general semantic of the construct to the programmer.

NON-BLOCKING SEND:
the sender just send the message to the destination and does not wait for any kind of acknowledgment.
After performing the send op, the process will execute its next op and so on.
We talked about the need of message buffering: in this context what does it means “a message has been sent”?
It may mean several different things and we have to be sure under the specific semantics or a certain middleware platform I’m using what it means:
f.i. a message has been sent when it has been placed in the outgoing buffer; or when the buffer has been flushed because the daemon in my system has took care of taking the message and sending it out over the NW; etc.
Depending on the specific platform and system we will use, a single term may mean different things.

Blocking vs Non-Blocking Receive

- Blocking Receive:
  it checks the incoming queue (the mailbox) and until a message has been found there, it will stop the execution.
  As soon as a message is available in the mailbox, the message is taken from there, will be actually received and the process can go on.

- Non-blocking Receive:
  the check is done as soon as the construct is executed. If the message is there it will be actually received, otherwise the computation goes on without any kind of blocking.

Consider that from the performance point of view having blocking construct means making the program less efficient (you waste time in waiting), it depends on the kind of program you are developing.

Another important point is that the use of specific semantics in blocking/non-blocking pairs may lead to dead lock situation, so take care.

We can recall a famous construct that has been proposed at the beginning of the computer science era known as the Guarded Command:
It has a guard at the beginning, then an arrow and then an instruction. The instruction contains a receive op; the guard corresponds to a condition plus a blocking receive op. In case some condition is verified and the receive op can be completed, the part named instruction will be executed. This is a way to mix conditions about checking some property and getting some info once it is available from other processes.

Example: Scheme for Bounded Buffer

How these constructs can be used to solve a classical problem in distributed systems.

The problem is the so called BOUNDED BUFFER.
The idea is:
I have some space to keep some information, but the size is finite. This block of information can be inserted in the buffer by other processes (named producers) and can be extracted from there by consumers (processes).
I want to organize the overall communication so that insertion and extraction of information blocks in/from the buffer will not turn into an overflow, I cannot insert an information block when the buffer is full, I cannot extract a block when it is empty.
I can imagine that the producer, before inserting an information block has to ask for a permit because it may happen that the buffer is full so it will have to wait for the buffer to have enough space before inserting the block. So, it has to request a permit to the buffer process.
It will receive the reply as soon as the permit can be granted, and once the permit has been obtained the producer can go on inserting the block by sending a message with that piece of information. So, in the interaction between producer and the buffer we can identify different reasons to exchange messages:

1. ask permit
2. give permit
3. insert the information block
   So I can identifies 3 different endpoints: 2 on buffer side (getting the permit, getting the information block), 1 at producer side (to receive the permit).(the 3 interaction on the left). The 3 dots are the ports.

Interaction between consumer and buffer it is much simpler:
the consumer must only ask for a block, if the block is not there no problem: I have to wait for a message from the buffer, there is no need for any further handshake between the 2 --> 2 ports: 1 on consumer side to obtain the requested block, 1 on buffer side to receive the request coming from the consumer.
If I do not use ports, or if I modify the semantics of send and receive I will have to figure out other solution.

Programming Paradigms

When we say paradigm, what do we means ?
A paradigm is a sort of a pattern, a way of doing something, typically we consider it as a way of thinking, and since we are talking about Programming Paradigm it is a way of programming .

A way of programming does not mean just knowing special constructs to do something, but instead is a way to look at problems and structuring possible solution for that problems.
A programming paradigm leads to the solution of the problem according to a certain number of principles.

We know C and Java --> these leads you to find different kinds of solutions.

We are interested in 2 very broad programming paradigms:

- Imperative:
  we have sequences of commands, and these sequences drive the control flow of the program.

- Functional:
  there is no state mutation, but the computation is seen as the process of evaluating expressions or a sequence of expressions.

Programming paradigms

When we say paradigm, what do we means ?
A paradigm is a sort of a pattern, a way of doing something, typically we consider it as a way of thinking, and since we are talking about Programming Paradigm it is a way of programming .

A way of programming does not mean just knowing special constructs to do something, but instead is a way to look at problems and structuring possible solution for that problems.
A programming paradigm leads to the solution of the problem according to a certain number of principles.

We know C and Java --> these leads you to find different kinds of solutions.

We are interested in 2 very broad programming paradigms:

- Imperative:
  we have sequences of commands, and these sequences drive the control flow of the program.

- Functional:
  there is no state mutation, but the computation is seen as the process of evaluating expressions or a sequence of expressions.

Imperative Programming

Let’s try to compare the 2 paradigms from a practical standpoint.
In imperative programming we have typically STATE information, that is an information related

to the description of how the system can be described in a certain point of time.
The execution of a statement results in a modification of the program state.

We can organize our program following different types of rules --> this led to the definition and adoption of more specific different paradigms:

- structured programming
- object oriented programming
- procedural programming
  so on…

See the example in the bottom of the slide:
going on with the computation means providing updates to the variable result (this is pseudo code).

Functional Programming

In the case of functional programming, the typical trait is the so-called lack of state during the computation:

there is no way to use a variable in a similar way to the one described in the previous slide.

Here a computation is seen as a sequence of expression resulting from the evaluation of sub expression.
How this evaluation takes place?

You have to identify and evaluate sub expressions;
once the sub expression has been evaluated the result is substituted in the external expression to get the final result.
The computation goes on exactly this way, by evaluating sub expressions.

In an ideal computation there is no SIDE EFFECT:
the effect of the computation is just the final result (the only effect will be the RETURN value in a function) --> meaningless to use global variables and so on..
This is both a strength but also a limitation because in practical life side effects are useful, f.i. IO is a side effect but it is crucial, so this pure functional programming cannot be implemented in practical systems, some side effects up to a certain extent have to be inserted in our functional language.

Now, let's compare the execution of a functional program to the previous execution:
consider the snippet in the bottom: the first line is an expression, we have to find sub expression and evaluate them and so on...
First line: 15/3 and 23 *are evaluated, and their result will be substituted instead of the sub expression.*
15/3 --> 5, 23 --> 6, x = 7.

Functional Programming Adoption

This paradigm has a lot of positive and interesting aspects so it has been adopted by many languages:
there are some languages that are pure functional, but also some languages that take some convenient ideas from functional languages and insert them into the overall picture of their programming paradigm that might be an imperative paradigm.

Haskell maybe the most paradigmatic one (it is just for pure functional people).

Erlang has several interesting features that are of interest in this course, it is suited to developing robust distributed apps;
but we have to say that there are other languages that contain (in their last versions) functional parts like python, java 8, go, c#, javascript and so on…

Concepts in Functional Programming

Some basic concepts in functional programming.
We will refer to that concepts that are not present in imperative languages and may sound unusual.

- Referential integrity (or pure functions).
- lack of state
- use of eager or lazy evaluation; often lazy evaluation of expression is used in functional programs
- functions are considered as first-class citizens in the language: you can deal with functions as with any other type. We make use of higher order functions
- recursion plays a very primary role, there is no possibility for looping but using recursion (forget about constructs like for, while and so on, they are not available)
- of course, data structures whose internal organization can be seen as recursive place an extremely important role. Among them the most important one is the LIST. (even in python the primary data structure is the list)

Expression & Referential Transparency

Let’s talk about each of those points:

Expression & Referential Transparency
As we said, computations are seen as evaluation of expressions.
An expression is said referentially transparent if it can be replaced by its own value with no change in the program behavior.
For this reason, such an expression has no side effect.
This concept can be moved to the domain of functions and so a function is said pure if its calls are referentially transparent --> this means that it has no SIDE EFFECT.

This is particularly important because this opens the possibility to a wide adoption of the concept of memo-ization:
taking notes of values that have already been computed, so that the next time you find an

execution of a function with certain values as parameters, you do not have to re-execute everything, you just have to take a look to your memo-notes, take the value and substitute it. This is only possible when no side effects are there.

One famous example is a program for the computation of the values of the Fibonacci's series.

If your functions are not pure functional (write some external variable and so on) this is not possible.

Whenever referential transparency holds, it is possible to formally reason on the program as a rewriting system (objects + rules to modify them), so we can apply formal methods in an efficient way to reason about the behavior of programs. This is crucial for optimization, for parallelize the program and so on.

Lack of State

In principle, state information is never kept. So, there is no need to have a mutable variable in your program.
Thus, the classical assignment op is not supported in functional program, so you need to provide something else because you do not have the possibility to change the content of a variable but initialization.
You put a value in a variable and then you cannot change it anymore, so a variable is IMMUTABLE.
How can we take care of something that somehow evolves?
You can create new variables, but you cannot modify values of existing ones.
At some point you will have variables holding values but they will not be used anymore so they are useless, and they corresponds to memory locations useless, so typically these kind of languages make use of a special component: garbage collector --> it frees up memory from no more usable variables. It is quite useful; the program must not take care of these stuffs.

Eager vs. Lazy Evaluation

Let's compare Eager and Lazy evaluation.

They are 2 different approaches in the evaluation of expressions, mainly for the arguments of functions.
This apply to languages in general.

- Eager:
  an expression is evaluated as soon as it is encountered in the program. This is what is done in function argument passing in the so called "call by value/reference".

- Lazy:
  the expression evaluation is postponed at the time when the result is really needed. This leads to the so called "call by need" semantics for function argument passing.
  On one side this could be a good thing: you avoid performing computation where they are not needed, but at the same time computation could accumulate at a certain point where everything has to be calculated.

First-class & Higher-order Functions

Functions can be handled as any other value in the program, so a function can be passed as an argument to other functions. Moreover, they can be also returned by functions, that is the reason they are called first-class citizens.

Higher order functions are those that can accept function as arguments and can return functions as result.

There is an important point, Closures:
what if in python you return a function as the final result of the invocation of another function and the returned function actually depends on the values of the parameters of the generating function?
In that case something strange happens: the final result is a function that depends on a value which is a local variable.
Local variable it means that it will be allocated on the stack, when that particular function is called there will be a frame inserted on the stack that corresponds to that invocation, and from there you can access the local variable, you return the result after the function call is finalized, what happens to the local variables ? they are referenced no more, so they will be possible targets for the garbage collector.
What about the returned functions that has to rely on that particular values?
That is a problem.

The trick is to make use of the Closure: it is a sort of special closed environment, created so that there the function can find the value of the parameters it depends on. It is a sort of primitive way to support the encapsulation principle.

Use of Recursion

Let us talk about recursion.
We know that recursion and iteration lead to the same result (same expression power).
Without the need to relay on state variables we can only use recursion to support the repetition of specific computations.

Look at the example in python to develop a function to compute the factorial of a given number given as input to the function:

- Left snippet, iterative --> we update the content of a local variable res and return it at the end.

- Right snippet: recursive --> we accomplish exactly the same result. We use 2 arguments: n and res that, if not differently stated, it will take value 1 (default value).
  This res acts as a sort of accumulator, but it is an argument so every time this function will be called the idea is to consider the previous values of this argument to calculate a new value so that that value will be assigned to the value for the new call for that function.

If n<=1 is the base case as usual in recursive function, otherwise there will be a call to the same function with different arguments (n-1, res*n).

There is no update in the variables, they are assigned just at the beginning.

There might be some problems:

every time we call a function, the system, to support such a call, will make use of a portion of memory on the stack, each call corresponds to using a frame on the stack;
if I will have recursive calls with a very high number of calls, it will happen that my stack will be grow and grow until my memory gets full.
This is why in many languages there is a limit in recursive calls (like in python).
What could be a trick to address this kind of performance problem?

TAIL RECURSION:
it can be applied any time the structure of the recursive function is exactly as indicated here on the right, where the recursive call is exactly in the very last op in the program.

So, you can imagine what happens to the stack:

It will be filled up with many frames, one exactly identical to the other, so at the last possible call when you fall in the base case, all these frames will be freed up in cascade one after another, in these cases there is a simple trick:

instead of placing every time yet another frame identical to the previous one, we replace only the values of the parameters, so we reuse the same memory of the last frame --> if you have 1 million calls one after the other, on the stack they will be just 1 single frame to support these 1 million calls.

What could be a bad aspect of this trick?

A powerful tool is the inspection of the stack trace.
By inspecting it you can really identify possible bugs and fix them, but in this case there is no possibility to do this because what has be done in previous calls is overwritten by the following calls and only the data relative to the last call is kept in the stack, this is why designers of python decided not to apply TAILORED RECURSION.

Use of Recursive Data Structures

We have seen that recursion is widely used.
As we have to do with data structures whose structure is recursive itself, it will be easy to manipulate information organized in this way, by means of recursive functions.

Consider another aspect:
in functional programming we do not want to emphasize side effects (ideally, no side effects), so why using array?

They are used to keep mutable info, so they are not built in in these languages.

We have to deal with immutable data, isn’t it really inefficient?

we could have a performance penalty from managing everything in this way, but we have to say that this applies only to the logical way.

In the implementation of such languages different tricks are available so that data reuse is applied to keep performance at an acceptable level.

So, the basic data structure is the LIST:
it can be seen as recursive because you can split it into 2 portions:

- The head (orange)
- The tail:
  it is itself a list which in turn has a head and a tail, and so on and so forth.

So, its structure is intrinsically recursive.

Addressing Concurrency (I)

We are interested in concurrent programming, and we can ask our self “is it really interesting for us to use functional programming for concurrent programs?”. We will see that actually it is.

What is the main problem in concurrent computations?
It is related to data RACES.
Look at the example:
We have x = 0 and you have 2 concurrent activities,
x = x+1 and x = x+1.

You have to perform them in parallel, concurrently (||), and then you want to print the value: depending on the way the ops are performed, the final result will be different.

This activity (the sum) is composed of 2 ops:

- 1 read (read the value of x)
- 1 write (write x+1).

If both read the value x=0, the final result will be 1 because both of them will make x= 0+1 = 1.
If instead one of them reads x = 0 first and increment it and then the other action read x = 1 the final result will be 2.

The very reason of DATA RACE is related to data MUTABILITY:
if I have to deal with something whose value can be changed;
but if I have to deal with something whose value cannot be updated we cannot have such a situation.

Even if in the future you will not make use of functional programming you will need to think differently to solve such situations.

Similar problems apply also to visibility of updates:
We studied different models for memory consistency, these kinds of problems are related, in a transitive way, to the possibilities of having updates and in particular to the visibility of updates to the other parties in the computation.

Addressing Concurrency (II)

Dealing with mutable state requires synchronization.

How to perform synchronization? You have to use construct for locking f.i., and synchronization is a way to limit utilization of resources because in this case when you have to lock a resource you have to spend time for this op and for unlocking, and more importantly if you have many potential users asking for the same resource at the same time you will have some of them to have to wait.

So, applying synchronization is a way to limit the performance of your programs. The suggestion is: if performance is an issue you have to limit synchronization as much as possible.
If you have to use pure functional programming you don't have state, you don't have side effects, so you do not have a shared mutable state, so, accessing something that is shared is not a problem because such accesses will be read accesses (no problems), and this leads to thread safe computations, with no critical races.

Moreover, the approach for the evaluation of expressions in functional programming is well suited to be parallelized:
an expression is made by several different sub expressions --> they can be parallelized since each sub-expression has no side effect --> better performance can be obtained.

Addressing Concurrency (III)

So these ideas at the base of functional languages could be very useful also in concurrent programming.

But we have to be honest:
here there is a citation from a paper:
we will always have to do with shared state in one way or another.

We have to do our best to limit the need for shared state, we have to use it only when we can't avoid it.

In this vision, a process can be considered as a basic component of a program and at the programming language level, we have to provide a way to make processes interact, and this can be done by using message passing.

What about Message Passing?

Message passing can be used also in a functional landscape, and the basic component of the computation will be processes.

Processes here mean abstract entities able to carry out an activity.
These can be mapped in different ways onto CPUs, onto different nodes in a nw and so on.
So we can reason about the computation that looks like a distributed one, regardless of the fact that different processes are hosted on a single machine or maybe are scheduled within a single CPU or are spread geographically across many different machines.

The Erlang language adopts exactly this vision of concurrency: in Erlang we have processes that exchange messages and possibly reacts to the reception of messages.

In principle the concurrent programs that are written according to the message passing paradigm could be run over parallel platforms (a multi-processor machine) or maybe over a network; the way to organize the computation is exactly the same, because the information passing is based on the exchange of messages