Note slide Distributed Systems 1

From Shared Memory to Msg Passing

On the left you can see the typical model for systems that use shared memory:

typically you have one single shared memory, and different processors can access it through a shared bus. Possibly some cache memory can be placed close to each processor.
So, all the communication between processes happens because a shared memory is used.
If you want to communicate something you place somewhere it in the shared memory, and the other one will retrieve it in that position.

In message passing systems the situation is completely different:

each process has its own local memory.
Theoretically, nothing is shared, but different processors on different nodes can communicate by means of an interconnection nw (please, do not think about TCP/IP or something like that, but in this case it is a general context).

So, there exists a way for one processor to communicate with another one, and this communication is carried out by means of messages.

Interconnections: Channels

How can the different nodes communicate ?

we have to say that in distributed systems when we say node and we say process, more or less we mean the same thing, why ?
I can imagine that a node is just a place where some process can be hosted.
If you have in practice many different processes over the same node you can imagine them as processes that can be communicating between them, but you can deal with them as if each of them would placed in a different position.

What about interconnection ?

The interconnection could be imagined as fully connected or partially connected.
The connection between a process and another process can be imagined as a channel.
Channels could be mono-directionals, or bi-directionals.
you can imagine a channel as a FIFO one or a non FIFO one, with respect to the property that messages over the channel can be ordered in a certain way or not.
FIFO = if you send a message A over the channel and over the same channel connecting 2 different processes you send later another message B, you are sure that A will be received before B, otherwise if it is not a FIFO channel, B might be received before A even if A was sent out before.

The hypothesis on message ordering are particularly meaningful when you have to design a distributed system.

Please, you have to be aware that when talking about distributed systems we are not talking about networking.
Networking refers to specific technologies that have been developed to support communication over different nodes.

Now, we try to take advantage of this communication infrastructure and we want to design the overall distributed app applying specific concepts that let us guarantee that some properties of the developed system will be satisfied, that's the reason we try to investigate the structure of an application by means of formal methods, mathematical methods.

Types of Models

Actually, there are different types of model that can be used to describe a distributed system.

F.i. we can have:

- Physical models:

we have to deal with the representation of the underlying hw elements supporting the communication across the different elements of a distributed system, but we will abstract away the details about the technologies used for developing each computer node and the networking protocols used to guarantee the communication.

- Architectural models:

in this case, we can see the overall system in terms of components, and the relationship between components.
In this case, f.i., we can identify a system as a client server system, or a peer to peer system, so emphasizing the role played by each component within the whole app.

Typically, in this case we have to rely on special softwares used to abstract away all the details of the communication at the networking level, but referring to special actions that are meaningful for that architectural pattern.
This special software support is knowns as middleware.

- Fundamental (abstract) models:

they are a sort of mathematical model, that account for the basic properties that the system has to satisfy.

More importantly, in a distributed context we have to reason about possible failures of nodes and failures of the communication infrastructure.

we will talk especially of fundamental models.

Fundamental Models

They contain only the essential ingredients to understand and reason about specific aspects of the behavior of a system.

All this is done with the aim of making explicit all the assumptions that are relevant in the system.
Moreover, it is important to use this kind of models for finding general solutions (generalizations) to understand what is possible or impossible to achieve, given some hypothesis.

What about generalizations ?
they are relative to general purpose algorithms, or properties that can be desirable to be guaranteed.
If you need to guarantee some property, you have most of the time to make use of mathematical proofs over the model at least.

What are the basic behavioral aspects ?

- the interaction
- the failure
- security

we will talk about the first 2.

How to Pass Information?

Now the problem is:

how to pass info from one node to another.

The idea is to make use of messages.

- In a message we can identify a content, payload:

what it is passed from the local memory of one process to the local memory of another.

Moreover, we can add something else to one single message:

- we can add further semantic, f.i. using message types to identify a particular message, or maybe adding tags to a message, so that by inspecting the tags you can understand something about what to do with the info received through that message;
- we can add metadata to the single message, f.i. a description of the content, the way to interpret it and so on;
- other metadata, like who is the sender, and so on.

There are 2 different primitives for message exchange:

- send

- receive

If you want to use send and receive primitives you have to know where to send the message (the process to which the message has to be sent), and so we need an addressing system.

Please note that we are not talking about addressing at the networking level, but we are talking about addressing, let's say, at the app level.
This means that if it will be deemed reasonable, the underlying networking addressing system could be used, but that's just an implementation detail.

Resource Management

What about resource management in a distributed context ?

when we are using messages to exchange info a resource can be paired with one single process, which is able to interact with the resource.
The situation is here on the dotted box, you can see Pn which is the process that interacts with a resource and with its own local memory.
Suppose that the others want to make use of that resource, of course in this setting the only way is to ask process Pn, so Pn acts as a manager for that specific resource, so the access for resources is always mediated by the communication with the responsible manager.

Problem: Process/Node Addressing

Let's talk about process addressing.

The problem is how to express the address of a process (or a node) to be identified in a unique way, but just for the purposes of the app, so forget about IPs.

Moreover, it has also to be found so that if I write a message for a certain process the underlying communication infrastructure has to be able to deliver such a message.

So, the communication infrastructure requires an addressing system.

The most straightforward way from a logical point of view to identify a process in a distributed context is using unique IDs.
But, we know that under the hood, the management of the communication is much more complex, so there will be some software that will take care of mapping the logical process IDs on actual addresses to find where a message has to be delivered.

Overlay Networks

In practice, we can even define an addressing system over the regular standard networking infrastructure.
This kind of nws are known as overlay nws.

So, each node will be identified by a special address, nothing to do with the IPs, but there will be a layer of sw that will take care of mapping the logical overlay nw over the real nw.

Explicit vs Implicit Addressing

It is important to distinguish between explicit and implicit addressing.

- Explicit addressing: I can identify one process and it resides on a single node (single machine), and in case I want to communicate with that process I need to use its own address. That process will be the worker I will exploit to carry out my computation.
- Implicit addressing: What if f.i. I perform this communication just for getting some result? for having some job done? In this case I can further abstract from the notion of process and I can identify not one single process but a service I want to exploit: I do not want to communicate with one specific piece of my app which is there, I only want my job done, so I can identify the provider of the service. In this case I will make use of IMPLICIT ADDRESSING, because by invoking this kind of service, I do not want to know exactly who will carry out the job I am asking for. Using this kind of addressing requires much more support from the underlying infrastructure, because now the supporting SW has to understand what my request is, and it has to map the request for a service onto one specific process which is able to carry out the job, then it has to collect the result from that process and has to pass me back such a result.
  Systems of this kind actually ask for a communication support which is quite complicated.

Addresses, Endpoints, and Ports

Let us talk about possible ways to address processes using Explicit Addressing.

We can just point out one important question: Why do I communicate with a certain process?
I may have many different reasons; I may send out different types of messages to one single process because I want to get back something different. So, how to consider this specific point at the level of the addressing system? F.i. I can identify different COMMUNICATION ENDPOINTS associated to the same process or node. This is just because I want to have a better organization, a better structuring of my app. So, an endpoint corresponds to the unique identification of a process plus something else. Usually this something else is known as port.
We talked about processing one process, then we move to the identification of endpoints and these let us to the use of PORTS. A PORT is an additional mean for the specification of the semantic of the message, making easier the handling of the message sent just on that port.
But this is only one possible way to add semantics to message, there are others that do not necessarily involve addressing but make use of different tricks. F.i. Erlang adopts a different way to add semantics to messages, it adopts the trick of PATTERN MATCHING applied to the special constructs used for receiving messages.

Getting More Complicated

Actually the situation can become much more complicated: we can identify further challenges in addressing.
So far we consider the identification in addressing of one single process, but we might be interested in many different processes to be contacted all at the same time, so we could be

interested in broadcasting: sending a message to all the processes belonging to the same system.
Sometimes I can partition all the processes in different groups, maybe each group of processes could be dedicated to carry out some special activity, another group is dedicated to another activity and so on, and in this case I have to use multicast communication.
Now, I do not have to identify one single process, but I have to group up processes and messages will be addressed to groups. If the membership to a group could change over time, this would make the overall activity of supporting the communication much more complicated.
So, even if the addressing problem could look quite trivial, as you put further challenges in our communication system, everything become much more complicated

Introducing Middleware

How to deal with this complexity in a reasonable way? Do I have to write my own SW for this in my app?
It is much better to build up dedicated SW just to support the functionalities I need, just for the communications, at a more abstract level.
This kind of SW layer is typically known as Middleware:
it can be imagined as placed in middle position between the OS and the app.
You can see in this picture a schematic representation, and f.i. you can suppose that on the left you have one single node, on the right another node in our system, and the app SW is represented just on top of the other SW layers; I can add a middleware layer whose purpose is to enable the communication between the two machines according to the abstraction that the piece of app on the top actually require. What you see as a programmer is just a sort of API, and you can just make use of special construct, special PRIMITIVES provided by the middleware to support communication and information sharing. The same middleware can show to the upper SW the same interface, regardless on the specific machine and OS you are using on a node, of course the implementation of the middleware layer will be different from one OS and another.
So, middlewares support INTEROPERABILITY across app components.

Middleware: How?

How to provide this support to our computations?
There are 2 different possible ways to do this:

we can rely on libraries, so pure SW, but sometimes we have to exploit the services provided by special processes that will run on each node to provide special services for us. We call this kind of support runtime support for the application.

We can have a question:
is it really a good idea to make use of middleware components? Or maybe it could be much better to do everything by hand?

Typically a SW solution that has to be cost effective, has to rely on some SW that has been already developed, debugged, tested and so on, and which is reliable; writing everything by yourself is not often a good idea mainly because of these reasons, but also because if you are asked to develop a SW app, you have time pressure by your client. Of course there are some exceptions, when you have to exploit at best the computing power of your specific machines, or simulation apps.

So, using middleware can be considered the most effective solution to support integration of SW parts.
When we will have to develop projects, we have to consider middleware solutions.