Distributed and Pervasive Systems - Appunti

Lezione 1 - 02/03/2022

Claudio Bettini. He worked for IBM in New York for one year. He also has been affiliated with idk University. Processes and Threads will return. Networking too (ISO-OSI stack, TCP-IP...).

Friday we have the "apply part" of the course, where we will work and follow what the tutors teach us. The goal is to experiment technologies related to the theory. We will develop and design a project of a distributed system with a pervasive part.

We will start at 8.50 (!!!). We will have a break in the middle. Let's motivate this course. Modern applications are Distributed Systems. All the applications that we run (or at least, a lot of them) like youtube, instagram, gmail... run as DS. They have an infrastructure organized as a DS. It would be good to know the concepts behind them. The course anyway is not only about Distributed Systems, because they will naturally include nodes (definition later). The nodes are phones, smartwatch, IOT devices, etc.., not just network and computers. Also, blockchain could not exist if it was not for an algorithm of DS. We will talk a whole morning of the blockchain. Anyawy, DS expertise is required in a lot of contexts.

Course objectives? Understanding the foundations of modern distributed systems. but we will not learn about about GRPC (in the lab yes though) or technologies in general: we will grasp the problems and the solutions, so that we can understand the future DS. So, foundations. Also, we will learn about transparency, synchronization, fault-tolerance consensus and blockchain, sensor data management and context-awareness. And also, we will be guided to design and program a distributed system. There is also a risk that someone copies our projects from Github and uses them. The project will be done singularly: the code must be self-made. the teacher doesn't want us to publish our project until february, when the last session of this academic year ends (the risk is that someone copies it, and it happened).

What will be talking about in this course?

First, what is a DS. Then, the architectures, like client-server and peer-to-peer systems and their organizations. Then, communication in DS. A DS is a collection of pcs that communicate together after all. This is the most interesting part of the theoretic part of the course: if we have different computers that need to communicate, how do we coordinate them to achieve a goal? An idea can be that the pcs share and idea of time, a clock, so that they can coordiante their actions. There are problems, of course, that we will study. We'll also see about mutual exclusion and election algorithms. In single programs, locks and semaphores are used. But in DS, we need communication. We'll see algorithms to do mutual exclusion in this way. The election algorithm instead tries to find a pc among all the ones that must be elected to act as a moderator.

The second part is about Pervasive Systems and their applications.

The third one is a guidance to project development. We will develop something that deals with, idk, election algorithms. We use Java, and concurrency & multi-threading. We'll learn about gRPC and MQTT.

This last one is a technology used to hadle data coming from sensors.

TEXTBOOK: Distributed Systems: Concepts and Design, 5e, Coulouris, Dollimore, Kindberg & Blair, Addison-Wesley, 2012. Isbn-10: 0132143011. Occhi però che the book does not cover everything.

Distributed Systems, third edition, version 3.03 (2020), Maarten van Steen and Andrew S. Tanenbaum.

Multiple choice questions + 2 open questions, one is usually an exercise about an algorithm idk. 50-50 theory part and project. Ricevimento on appointment.

End of introduction. Tanenbaum's book will have a lot of things of this class. What is a Distributed System? *A collection of independent computers that appears to its users as a single coherent systems*. This means that the single units don't have multiple CPUs, or at least, they are not required to have more than one. The fact is that each PC has its own memory, there is no shared memory. They are phisically independent, but they should appear as a single entity. The user should not know how many computer there are. And how do we build such a system? Of course we need a network that connects them. Each PC has a local OS, that can be different from machine to machine, and each PC may run a certain application. A given application can run on multiple computers! How? Well, thanks to specific softwares, that use TCP-IP technology. Usually, we have to install a middleware, a Distributed System Layer, that hides some details from the layer: in particular, it hides the fact that the application is running on different PCs. For example:

imagine opening a folder in your PC that shows you contents that are not really on the machine that you are using. The resources are shared. Another thing that can happen is that, when i start a process, it really starts on another computer, even if it looks like it started on my local computer. Also in Multiplayer Online Games we can have different components distributed on different machines. And of course the WWW, even if this is not a precise example. But somehow, the WWW has distributed resources (it has different independent computers that are connected and work together). But why it is not exactly a Distributed System? Because, if i look at the website of the course (for example), i do because i know what i'm looking for. There is no location transparency: i know where my resource is locate, i have its IP address! But in a DS i don't need to know where it is.

An ironic definition of Leslie Lampor is: "*you know you have one (DS) when the crash of a computer you've never heard of stops you from getting any work done*". This is a problem we'll have when developing, since debugging will be difficult. We will have to coordinate different machines, but we'll see some tricks to have an easier life.

What are the goeals of a distributed system, its properties? Well, we have some resources that we want to make accessible. And we would also like Distribution transparency. We have different levels of transparency. In ALL EXAMS there is a question of transparency. There are different kinds of transparency that we'd like DS to have. I.e, location trasparency is the ability of the DS to hide where a resource is located. Or, access transparency is the ability of the system to hide different data representation and

how a resource is accessed. The component of a DS are independent computers, that can have different architectures, file systems, and so the way we access to the resources in those File Systems is different. A DS should have a layer that hides how the user accesses those datas. A network File System (NFS) is an abstraction layer that standardizes the access to the different FSs.

Migration: the user doesn't have to know that a resource is located in some PC. And doesn't need to know if that is moved from a PC to another. Relocation, instead, is moving a resource while it is used (is migration but more complex).

Oh, n.b.: not all DS have ALL those properties.

Replication transparency: the user should not be aware of the fact that a resource is replicated in different places of the world. I'd like to replicate a reource in roder to have fault tolerance and better performances (if i have a resource closer to me, i'll acces it faster). Concurrency transparency is that a resource may be shared by several competitive users, but this is none of their problem: for the users, the resource they accessed is just accessed by them. Failure transparency is that if something fails, the system should be able to recover by itself, and the users shouldn't note that a failure happened. For example, in an FSM, if a machine is trying to access a resource but it doesn't have that resource, that it asks to another machine to pass it. If this fails, it asks to someone else, and so on until a timeout or the resource if found.

A DS also needs to be open. But what does it mean? An open DS should offer Interoperability, Portability and Extensibility. Interoperate means being able to connect and work with another system. Portability means that an application can run on different DS that have minimal or no differences. Extensibility means it can be extended. How do we achieve opnenness? A general rule to guarantee this property is to not design our own protocol and keep them secret. if we do so, we'll be the only one to know how our program works. We should use standard protocols. We should also publish key interfaces with specific languages, like Java. But there are a lot of frameworks (like corba) that allows different processes to run on different computers and in different languages, but they can still communicate one another. So, we need standard interfaces, or API, to make different processes communicate one another. And last, we need testing and verifying the conformance of components to published standars.

The last property we'd like a DS to have is scalability: what is scalability? We want to be scalable in terms of what? One example is size scalability: if is increased the number of nodes/users, i have to scale the resources that support them. Can the system do that? Other scalabilities are the geographical one, that require the computers to stay "close" geographically speaking, and understanding if the administration is centralized or distributed. Here are some problems related to scalability, or implementation choices that can make scalability an issue:

-Centralized services: a single server for all users. If a single mahine handles all the users, and they grow too much, the machine fails. In this case, a load balancing technique in a DS context would have been better. We need replication and things like this.

-Centralized data: of example, a single on-line telephone book. Or, the old way to do the name resolution

in internet (symbolic addresses to numeric addresses). The Domain Name Service is nowdays distributed, but it was not always like this. There was once a file on a machine that was queried when a resolution had to be done. This caused problems of cause.

-Centralized algorithms: example, doing routing based on complete information. Routing is decing in which way a message has to go. An algorithm that decides the whole path, it is not efficient. If each router handles local informations, btw, the load balance is better handled. In decentralized algorithms, we have that: No machine has complete information about the system state. So, a router knows the links to neighbor routers, no more. Machines make decisions based only on local information. Ok, this is self-explanatory. Failure of one machine does not ruin the algorithm. Of course, if a machine fails, the whole system still stays alive. If a "moderator" fails, the system should be able to elect someone else to moderate the system. Finally, there is non implicit assumtpion that a global clock exists. In a DS, the computers are not always perfectly syncronized. But we usually need perfect syncronization. Well, we'll see how to deal with this problem, and how the machines don't assume a perfecto shared clock. We'll see centralized and decentralized versions.

Imagine that a server has to handle the checking of forms as they are being filled, or a client that has to do the same thing. In this last case, i take computations away from the server.

Also the DNS nowdays uses a more scalable way to do name resolution. The DNS is organized in domains, zones, and it has multiple servers on the edge of the internet. The name resoultion is performend through communication and collaboration of different nodes spread in the world. This way of distributing the name resolution is a way to make the WWW scalable.

When dealing with DS, we should not keep some thing in mind. The following are false assumptions in DS:

-The network is reliable. False.

-The network is secure. Of course not.

-The network is homogeneous. No.

-The topology does not change. of course it does.

-Latency is zero.

-Bandwidth is infinite.

-Transport cost is zero.

-There is one administrator.

-Debugging distributed applications is analogous to standard applications.

Now we have an idea about what a Distributed System is, its properties and problems. Now, what kind of DS exist? Well, for example, Distributed computing systems and information systems and pervasive

systems.

Let's focus on Distributed Computing Systems. What are clusters? A collection of similar servers closely connected by high-speed local-area network and usually running the same operating system. The idea was: having a high-performance single computer is difficult. What if we have hundreds or thousands of PCs connected together that simulate a high-performance computer? The goal was high performance and availability. There are two kind of clusters: symmetric and asymmetric.The symmetric is: all the nodes are the same. There is no master, all nodes have the same software installed, and run the same program. All of them run just one code, the same one, but it is executed in different processes. One of them is elected as coordinator anyway. This is kind of challenging, and has not been the most used type of cluster. In the asymmetric approach, we have a master node instead. We have a lot of compute node with their own local OS and a component of a parallel application. On the mster node, however, we have the local OS, a management application and parallel libraries (?). The case of Google Borg: Google designed this system in 2003, and it had a master, a certain number of compute nodes, and all of them communicated. The strange thing is that there were multiple BorgMasters. Why? For fault tolerance and performance reasons, in order to not have pitfalls or bottlenecks. The five masters elect one of them (the one that is doing the job), and that is the working one. the other fourt though follow the work of the first one (they clone the state). In this way, if the super-master fails, the other can take its place. So, each of them stores the state, but we need a way to ensure their state is the same. So we run the Paxos protocol that makes this possible. After 10 years of google running boh.

Nowdays we have Kubernetis, that Google made open source (?). It is widely used because it was donated by the Linus Foundation and everybody can use. Kubernetis is a platform for managing containerized computer applications. It can be seen as an evolution of Borg, and is an example of Asymmetric cluster. [Compito per casa: leggi articoli riguardo borg. Lui ha pullicato un articolo su Borg con le slides].

So, cluster computing is a kind of Distributed System. Cloud computing is another kind of DS, that allow users to access resources without knowing exactly where they are. In cloud, we have that the nodes are etherogeneous in hardware and operating systems (not the same as clusters i think). In this case, different cloud clusters can have different technologies, OS, different networks, and so on. The Cloud Service Models are Iaas, PaaS and SaaS.

IaaS: I just want this much of memory, computer power, and so on.

PaaS: I want more things. If the cloud has a distributed database, i'd like to use it. So i use some higher-level tools of the cloud. An example is Google App engine.

SaaS: google docs, Gmail, Youtube... all applications that we use as services.

There are also different deployment models, like private cloud (exclusive of a single organization comprising multiple consumers), Community Cloud, fuck. Other but i didn't read.

Edge or Fog Computing: introduced to take processing closer to where data is really produced. Like, we

have a lot of IOT devices nowdays, and we want to process their datas. The important part is: behind all those systems, we have DSs.

Pervasive Computing: *"the most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinnguishable from it"*. The idea is to make everyday life things "intelligent" by putting a limited computationl power inside of them, and let them be part of a DS. If we do so, we can retrive and share a lot of datas. In general, a system is "smart" if it can adapt the way it behaves depending on the context. if the system can understand the context and change the behavior, it can be considered as smart. But doing so is not easy of course.

Lezione 2 - 09/03/2022

Distributed Systems Architecture. A DS architecture defines the main entities of the system (sensor nodes, processes, threads, object, components, services). It describes also the pattern of communication between those single components. It describes also how do they communicate and the role of the entities, for centralized and decentralized architectures and also hybrid once (server, client), and other things. Some architectures in clud computing are also absed on containers and microarchitectures.

In centralized architectures, we have client and servers architectures and the event bus architectures. In the client-server model, we have a central component, a server computer, that waits for requests from clients, and we can also have servers that act as clients. In this patter of invocation and service answered from a temporal perspective, we have that the client requests something to the server, it provides the service and replies to the client. There is also network delay, that is, the moment the client requests a service is different from the moment in which the server starts working on it. In this case, the client stops until the server has fully processed the request and answered to the client. There are many variants of the client-server model. We, as designers of the system, can make decisions that have an impact on scalability and other things. In a case, the client is very dumb, it just has a UI, and gets all the informations from the server. We can have also models in which the UI is completely client-side, the database ont server-side, but the application is split between the two. This is the case of a web-based application, where we could use javascript for the client side both for UI and the application. In other cases, the application is completely client-side and forwards SQL queries to the server! The server just has the database and is of course capable to fulfill the requests provided by the queries. The right-est extreme is that portion of the database is cached within the client. There are, in all this, a lot of caching systems. How does caching work in those c-s architectures? The web servers communicate with a proxy server, that is in the middle between the server and the client (closer to this one, so faster client-side). This is used to cache some requests in those proxies, so that future accesses are faster. In the DNS, it's a good idea to store some conversions even locally on the laptop. Another common way to organize software components in a client-server architecture is to not have only two tiers (client and server), but on more tiers, like three. A way is to divide the application logic in a server and the database (intended as data management) in another server. This is because, in this way, datas are divided from applications. In this way, application can change while the datas just increase, but can stay without problems. In those

case, we are not only talking of different processes, but different hardwares. This is even an old architecture! In the VERTICAL DISTRIBUTION, we have a different server or node for each functionality of the system. in this case: data management to a node, application logic to another node and presentatio (UI) to another one. It's up to us to decide wheter it's better to distribute an application on more machines or viceversa. In fact, in HORIZONTAL DISTRIBUTION, the same functionality is distributed on multiple servers or nodes with load balancing. For example, we can have a server that accepts the requests and forwards them to other nodes. in this case, the same functionality is distributed on multiple nodes. This helps with fault tolerance, node balancing, etc.. the Round Robin technique can be used for the distribution of the tasks. The VERTICAL and HORIZONTAL distributions can also be combined. In those cases, each functionality is duplicated on a separate group of servers with load balancing. So, some servers are devoted to a specific functionality (vertical), but each functionality is divided in multiple machines (horizontal). So, multiple levels for different functionalities, but at each level we have multiple machines.

Let's talk a bit about the Microservice architecture. We can take to extreme the idea of separating functionalities (as in the vertical distribution). In this case, we run each component/functionality of the application logic in a separate process and possibly a different machine. The idea is: if i have a functionality that is independent from others, i'd like to isolate it in a single process or machine. This of course adds network overhead. But nowdays, with low latencies, it is possible to also follow this approach. microservices must be independently replaceable and upgradable, packaged with all they need to be deployed. Microservices communicate with lightweight mechanisms, like REST API or google RPC (invoking a function that is in another machine). Focused scalability: it means that the system MUST be scalable (a lot of users), and to achieve that i find out what's the functionality that is the most critical and i duplicate it, so that i don't have bottlenecks. And also, microservices can be written in different programming languages. [Java RMI is similar to RPC. The problem was that everything had to be written in Java anyway!]

A container is the abstraction at the application layer that packagesa code and dependencies together. Basically, the idea is: cool the microservices! But how are they run? To increase portability, we want to run those functionalities in an enviroment which has all that is needed to run a software. We wrap, in a *container*, the software that implements the functionality and all the libraries needed to run this piece of code. But where do I run this container? Well, the idea is similar to VMs. In a VM, we have a Guest OS that contains bins and libraries and one ore more applications. What's the different thing in containers? Well, we are at ahigher level. We don't care about OS. We have a basic, Host OS, and on top of it we have Docker. On top of docker, we have a container with the app and its bins and libraries. Docker can be thought ad a container engine. It does some optimizations: for examples, three instances of the same program (MySQL DB) can share the same libraries.

HE WILL NEVER ASK US ABOUT CONTAINERS (they are just a way to run microservices). But the idea of an architecture where each functioanlity is run in a single, separated environment, and the microservice concept, is important. This was the centralized architecture: we always have a server(s) that serve clients.

Another centralized architecture is the Event Bus Architecture. It works following a pattern called publish-subscribe. In this case, we have a certain number of nodes (subscriber), that are interested in getting some informations. And there is another number of nodes (publisher) that publish those informations. The informations, the "news", are published inside an Event Bus, and the subscriber takes them from the Event Bus. The publishers can be seen as source of events. This model is centralized because of the presence of a Broker in the middle, that is similar to a server. We can duplicate it oc.

Now, decentralized model. Many years ago, distributed systems investigated a different way to distribute their nodes. In this architecture, we have different nodes that act both as clients and servers. All nodes have the same functional capabilities. Napster is a well-known ?2? system that started in 1999. It was proposed by two college students, but was not totally decentralized. It was used to exchange audio files. Some years ago, infact, it was shut down. The idea was: hey, friend, you have a file i'd like. Can you share it with me? To do this, a server was reached (this is why it was not fully decentralized). The server gives and index, that is a computer that has the file (it can be a list of indexes). At this point, the client requests the file to the given node (infact, the index is really an IP address and a port). The most important part of the protocol starts now: the computer that got the file adds its IP and port to the index server, to notice that now he also has that file. So, this was Napster.

Problems in developing those systems: how to place data objects across many hosts in order to achieve load balancing while accessing data and ensuring availability avoiding overheads.

Three generations of P2P systems (like in the book): we already talked about the first one, Napster, used for file sharing. Then, protocols were refined to improve scalability, fault tolerance and anonymity, and people started working on other systems with a similar decentralized architecture. Gnutella, for example, that said: let's design a system like napster but without servers. FreeNet said: we want to be anonymous while sharing those files. BitTorrent is kind of doing the same thing, but the idea, the key idea, is that a single file was downloaded from different nodes. Different part of a single file, called chunks, are asked to different nodes, and used to construct the whole file. The last generation is P2P middleware: can we design middleware that achive the objectives said before in a way that is abstract form the data that we are looking for/sharing?

So, once those systems reached a good degree of scalability, we hade P2P middleware. Their goals were to enable clients to transparently locate and communicate with any resource, add and remove resources and add and remove peers (those goals are a QUESTION IN THE EXAM!). So, the objectives are that we can add new partecipants to the system easily and delete nodes also easily. A way to optimize those operations is to use an overlay network, like internet. An overlay network is a logical network, or virtual network, in which we impose a pattern of communication on the nodes that don't correspond to the actual connections of the nodes. The optimization criterias are global scalability, load balancing, locality of interactions, etc.. So, we have an overlay network, like a ring composed of nodes connected as a ring. If we have this kind of overlay, we can think about overlay network as another virtual network.

Ooook, so, the Routing Overlay is a distributed algorithm used to locate nodes and objects in an overlay

network. Routing requests fro clients to hosts holding objects of interest at the application level instead of the network level IP, etc..

We can divide P2P systems in two families: structured and unstructured systems. Most of them will be using a structured P2P system, that is deterministically built in order to obtain efficient routing towards the node containing the required data. In the unstructured, the overlay network is built with randomized algorithms. That is, each peer only knows about its neighbors. Let's see structured overlays first: the idea is to have a structured overlay, like the ring overlay. In this way, we can also optimize the search in the system. For example, we have a certain number of bits for identifying a single node. If we have 16 nodes, we just need 4 bits. A function is used to associate an address to each node. Some nodes are called *actual nodes* and are the nodes that are present in the system. Also, some data keys are associated to each actual node, following a certain algorithm (this is the Chord structured overlay). (the resources have an address). Distributed Has Tabels are used for having a faster search. What are they and how do they work? The idea is: we have a data structure in each peer, that is a Hash Table or Finger Table. It's an extremely simple table in which in the first column we just have an index. The number of rows is the number of bits used for the address. In the other column, we have succ(p + 2^(i-1)). p represents the index of the current index, while i is the value of the first column. succ(x) means the next available node which index is higher than x.

This kind of structured overlay, the ring, and this algorithm, is just an approach. CAN, content addressable network, uses a different kind of separation of ndoes. CAN uses a bi-dimensional address space, where we have different points, and to each of them a certain space is associated. Each node manages all the points in tis region. In this case, the insertion is more complex: we have to divide a rectangle and give half of the resources to a node and other resources to another one.

Limitations of structured overlay? I mean, why somebody came up with the idea of unstructured overlay networks? Well, the finger table has to be mantained. Each time a node goes away from the system, we have to change the finger table This maintenance of complex overlay structures can be difficult and costly to achieve. If our target is a dynamic environment, we might not want distributed has tables, that are too costly to mantain. We'd like something that is resilient to fault tolerance and auto-organizes the nodes.

In unstructured overlays, the routing is based on randomizing algorithms. Each nodes stores a list of neighbots randomly built. That will be its "view" of the overlay. Also, resources are often randomly assigned to nodes. How is the search handled? It starts from a node and propagates according to local views. Also, the search is often limited to a number of hops or timeout, and the resource replication helps improving search success rate. The search must be limited because else I'd ask to a lot of peers for a file, and that is time consuming. That is why we replicate the resources. How do those randomized algorithms work? one of them is the gossiping algorithm (?): the nodes, periodically, exchange, with the nodes with which they are connected, some informations about other nodes. Basically, a peer communicates a fraction of its knowledge (its neighbor peers) to another peer, randomly (circa). But since this is a dynamic system, we can have that a node goes down at some point, and i cannot communicate with it. For this, the nodes will tend to share the last contacted nodes to other peers.

Structured: we are guaranteed to locate objects, if they exist, and provide time and complexity bounds on this operation (hash table -> logarithmic time). Also, relative low message overhead. BUT. the manteinance of this structure is compex.

Unstructured: they self-organize, they are probabilistic and so cannot offer absolute guarantee on sh-

Hybrid architectures: for example Gnutella, that has some superpeers connected one another, and they are the reference for a P2P network of their own.

BitTorrent: another Hybrid architecture, in the sense that a part is centralized (when looking for a torrent file). In a .torrent file, we have a list of trackers. Inside of them, we find the IP address of ?. The point is that getting the address is centralized (like Napster), while the single nodes part is decentralized. [he won't ask exactly how BitTorrent works, but hybrid systems and hash tables yes].

Lezione 4 - 16/03/2022

We'll go back to the chord algorithm. We were talking about structured and unstructured overlay network. Chord is an algorithm that maps any kind of resource to a node. How to quickly find resources? how to remove or add nodes? One idea that was used in chord but not only (also in other structured overlays) was to organize the overlay as a ring, so each node only knows about two neighbours. Each node had a Distributed Hash Table, called finger table. How many entries does each table has, at most? 160 bits (?), so each node has at most this number of nodes/addresses that he knows. The time that the algorithms takes to map a resource to a node is at maximum Olog(n), that in this case is almost constant. How does a new peer get an address in the space? Using a hashing function. [A way to get a unique name for a peer is to get the ID number that identifies the machine on the internet and the port that identifies the process in that machine.] Since we have 160 bits, the address space will never be filled (too large number), and so each node will handle multiple resources. The nodes are much less than the resources. A file with key k (obtained through the hashing funciton) is managed by a node, that is the first node (going along the ring clockwise) with id >= k, called succ(k). The calculus is done % 2^m, where m is the number of bits used.
Chord provides a function LOOKUP(l) to efficiently find the address. Infact, this algorithm provides a middleware that gives us this function, used to know the address of the peer that will handle this resource. [Sta mettendo molta enfasi sul modulo, non so perché. Comunque id >= k ma tieni sempre presente il modulo].
So, chord has this algorithm behind, but the programmer doesn't need to know all this stuff (but if he wants to program new algorithms, he has to know them oc). Now, imagine that we have the finger (hash) table of a certain node with id = p. Its number of entries is m, where m is the number of bits used for addressing all the possible peers in the ring. FT(i) is a function that we use to refer to the content of of the Finger Table, row i (FT(3) = 9 in the image). The value of the function, basically, is the address of a peer. But, as said, we need to know, to communicate with a peer, its IP and PORT. So, together with 9, we have an IP and a Port. The value of the function is calculated in a clever way: succ(p + 2^(i-1)). All the operations are % 2^m. succ(j) means: if j is not a node, take its successor node (the next available peer in

the ring). [In tannenbauns there is the same example of the algorithm of Chord]. The formula says, for resolving a key k: find FT(j) such that FT(j) <= k < FT(j+1). After that, the search is forwarded to the node FT(j). If there is no FT(j+1) in the table, then take the largest address number. We take FT(j) and not FT(j+1) because between those two there may be a lot of nodes. However, there is a case in which i can go directly to FT(j+1): when k = FT(j+1).

Benefits of having unstructured peer 2 peer network with superpeers? Keeping an index of data for a subnetwork and optimizations I didn't read.

Let's see some questions we might get in the final exam.

Saw them lol.

COMMUNICATION MODELS IN DISTRIBUTED SYSTEMS.

Communication is the core of distributed system. We have hetereogeneous machines, nodes etc.. in those systems, and we need some general mechanisms that abstract those differences. What will we cover? Types of communication (persisntend vs transient, synchronous vs asynchronous), transient message oriented communication like sockets, persistent asynch message oriented communication, that is queuing systems and message brokers, and remote procedure call. In the case of the algorithm seen below, the "arrows" inside the ring were really RPC.
The model for communication in the network is composed of muddlware protocols, at different levels: Physical, Data link, Network, Transport, Middleware and Application. Each of them has its own protocol. The sockets are completely above all of this, they are an abstraction. Remember that this is not a networking course. Http is the most commpon kind of application protocol. We'll never go below transport anyway.

Types of communication. We'll see, on the slides, a lot of drawings with bold and dashed lines representing the fact that a node is doing something (or not) during a certain moment in time. Usually, a client, after having requested something, wants to know, before the response, if the request was sent or not. A client may also want to know if the request was delivered succesfully or not.

What is persistent communication? We have it when there is some infrastructure between the sender and the receiver that conservs the messages. For example, a mail server: it keeps our mails until we read and delete them. This is a persistent system. We can have persistent and asynchronous systems. For example, a client may be sending a message to a server that is not up and running in that moment. So, we have a persistent system in the middle. When the server runs, he fetches the message from the intermediate level. In a persistent synchronous communication, the receiver gets the message, but doesn't handle it immediately. In this case, the receiver waits until this confirmation. In the Transient Asynchronous communication, the sender tries to contact the receiver. Independently on wheter it responds or not, the sender continues processing. Note that there is non intermediate level: the server either recieves the message or it doesn't. In the Receipt-based transient synchronous communication,

the client sends a request and waits until the request is received. He doesn't do anything until the server responds. WE ALSO HAVE delivery-based transient synchronous communication at message delivery. In this case, the server receives the request, accepts it, and puts the client in a queue. When he starts processing it, he sends an "accepted" message to the client. Then there is Response.based transient synchronous communication but idk.

How can we do Message oriented Communication? Let's see the mechanisms. How can we classify sockets? They are generally not used in systems that use persistency. In fact, with sockets, theorically both computers are present during the communication (even if there are ways to use them asynchronously). How the sockets work, however, is hidden from our eyes. Let's see Berkeley Sockets. Suppose that a client wants to communicate with a server. What does it do? he calls a system call, that is a socket. [A socket it's like the plug of a cable. You need two of them for the communication to actually work.] When we call the socket primitive, the same thing as "opening a file" happens: the system returns us a socket descriptor for that specific socket. Each process in unix has three file descriptor, always: stdin, stout and stderror. When asking the system to open a socket, three file descriptors are returned. They are used to read and write on the "cable". Something else has to be done in the system: the descriptors are numbers really, so we can't use them for communication. So, after opening a socket, we perform a **bind**. It associated to the number of the socket a port, a free one. We are literally "binding on a port", a free one. In general, the bind operation will take as argument a specific port, if we want to specify it. We have then the **listen** function. It allows to open the connection by communicating with the TCP protocol (for example). It basically says: I'm a server, and i can receive many requests from different clients. So, please, OS, organize your memory so that I can receive different requests and process them (put them in a queue if necessary). This is NOT a persistent mechanism: remember that anyway the server needs to be up and running to do this. So, now the client calls a function called **connect**, that takes as arguments the IP and the port of the server. The connect, through the stack TCP-IP, is able to connect with the TCP of the receiving machine, that delivers the request to the server (if there is space to do it). If there is space from the listen, the request is queued. If there is an error, anyway, the connect fails. If it doesn't fail, anyway, the connect is accepted (**accept**), and **read** and **write** operations can be performed. Note: in the client, there is still a bind, but it's hidden. The application, the programmer even, doesn0t need to know the port used by the client for communicating with the server. We just want a free port. In fact, we just need the IP and Port of both parts of the communication to fully describe it. At this point, it's all about our choice, that is, we can define the format of the messages and similar things. This is a mechanism that doesn't work if the server is not up and running anyway. Sometimes, we would like to have asynchronous mechanisms or some kind of persistent system. In those cases, we have a queuing layer. The queuing layer is between the sender and the local OS, both in the client and server side. It receives messages and keeps them until they can be processed. But what happens if the whoule receiver node is down? That the server can't get the message, so, we have no service. So, what those systems do usually is to use different nodes on the network to conserve the queues. In the message-queuing model, we have some primitives and their meaning in the slide. Put, Get, Poll (check in a queue if there is a message and remove it, but don't block) and another one.

The intermediate nodes that implement the queuing system, have also another role. In some

intermediate processes, we have a so-called Broker program. He not only has to keep the messages when the destination is down. He also needs to convert some informations: if the two machines "talk in a different way", the broker needs to convert a message into another language when needed. These kind of systems, with a queue, are useful when we want asynchronous communication, and scalability can be increased by admitting delays in requests and responses. But also when the procucer is faster than the consumer, and when implementing a publish-subscribe communication pattern. The messaging protocols we will be seeing are XMPP (maybe?), MQTT, that is a very lightweight/efficient protocol for machine-to-machine communication (also used for IOT) and AMQP, that is more complete and belongs to the ISO standard. This is suitable for a lot of more different infrastructures. Some popular message-broker softwares are Eclipse mosquitto (supports the MQTT protocol, is opensource and lightweight), RabbitMQ (it's very popular and partially opensource, and also supports AMQP and other protocols), Google Cloud Pub/Sub, Amazon MQ and Apache ActiveMQ and Kafka. [All those things like RabbitMQ implement middleware for queue messaging].

In RabbitMQ, there are different modalities of use. In the Topic modality, that is the most commonly used, the messages are put in a queue, and then the consumer receives them. The receiver is just one.

The role of the broker is the important thing: we want to implement persistent communication, and without the broker we can't do this.

REMOTE PROCEDURE CALL.

What is difficult in making this remote? When we work with data structures in a program, sometimens we use functions, that take arguments. When we pass the arguments to a function call, we usually pass some variables that hold a value. The same goes for constant values. In those cases, the RPC is easy: i just forward to the remote machine the function name and the parameters. The problem is that sometimes we pass a memory address to a function. We can't do this directly in RPC. The remote machine doesn't know my local addresses.

Lezione 5 - 30/03/2022

Synchronization and coordination among processes/nodes in a distributed systems. BUT, before that, let's go back to the topics of the previous class.
We saw transinet, persistent, sync, async communication, sockets and queuing systems. The third topic about communication is RPC that works at a more abstract level. All P2P systems we'll discuss, like chord, have a communciation implemented as RPC. A critical point in RPC is the packing and unpacking of parameters. A DS that supprots a RPC needs to have both on the client and server side some technology that hides to the programmer the fact thath the objects are not all on the same node. We have a Stub both on the client and on the server. They just take care of the parameters when passed to the other side. The sender serializes the data, while the receiver deserializes them in order to understand what the sender wanted to communicate. This process is also called marshalling and unmarshalling. Marshalling requires serialization of objects, using a string-based system like JSON or binary format. Since this is a

heterogeneous environment, we have to be careful about how the data is interpreted. The sender and the receiver must agree how to interpret them. This is why we need a standard way of coding this. RPC is sync or async? Well for sure is transient, because when calling a procedure on another nodoe that node must be ready to answer immediately. The client calls the remote procedure, the request is delivered to the server, and a reply is fired to the client. So, sync or async? Usually, both are supported. The standard one is synchronous, but in general can be also asynchronous. We can have also a different point of synchronization, for example, the client waits until the server has recieved the request, he doesn't have to wait for the result.
Deferred synchronous RPC: as soon as the client is sure the server has recieved the task request, it proceeds with his computations. But at a certain points he will recieve the answer from the client, and at that point the client will be interrupted by a call from the server. How do we perform this interrupt? We can have a thread in the client that waits messages from the server (RPC calls more than messages actually). There are also Sys calls to avoid generating a second thread, but we won't cover that.
Parameters: if they are passed by value, no problem. For example two integers. But if I'm using a complex data structure and I pass it by reference to my memory, what do I do? The remote machine doesn't have that reference to my memory. So we can't pass by reference.
Marshalling and Unmarshalling: who writes the code to do that? Some stubs on the client and the server. How do we generate those stubs? Usually they are automatically generated: the system generates it in a C-like manner (?). To generate the stub, we have to provide: the name of the procedure, the kind of parameters it wants. We must also think about the fact that the server and the client might be written in differen languages. Anyway, once we have our Interface definition file, that tells how to format the stubs in different languages, we can use a IDL compiler and use all the languages available to generate the stub (the stub is a reference to the true object on the machine).
A modern RPC example is gRPC. It is largely used to build low latency, scalable distributed systems, and supports multiple languages. It's also open source.
esample of IDL and stub generation in gRPC:
1) Define: define the function name and the parameters.
2) Compile: in the language you prefer
3) Generate the stub.
How do we bind a client to a server? If we have a server, we'd like to make a RPC available to everyone. How can a client in the world know about my server and use my funciton? Well, first of all, the server process might be a daemon process, that is always up and running. What a server can do is "Register an endpoint" in a endpoint talbe present on a DCE daemon process. So, this daemon has a table: a list of all the processes in that machine. So, 1°, register endpoint: tell the daemon that I have this service. But how do I advertice my service? 2°: we have a Directory server, on a different Directory machine, which is notified about the fact that I have my service up and running on this machine. If a client machine is looking for a certain function, but is not interested on the specific machine/process that gives that function, but instead on the function itself, he asks: how do I call this function? 3°: the client contanct a look up server to discover who has the service I'm looking for. 4° He then asks for endpoint, contacting my machine, in particular my daemon process. This process checks if the service server is up and running. If it's not running, that process starts running. And now, 5°, the client can perform the RPC. The daemon in fact communicated to the client the port on which he can find the the server process that

now is running.

So,

Synchronization Problems in Distributed Systems.

We saw the architecture of the DS (how the nodes are connected), we saw the communication part, and now we have to see how do they synchronize to do a more complex task. We synchonize because we are here in class at 9.00. The computers need to do something similar. Another problem we have is mutual exclusions: we have problems even on a single machine, imagine on a DS! But if we have mutual exclusion, how do we decide who can access to them in a certain moment? We need some algorithms to do that. We can also have an algorithm that chooses a node that has to perform those kind of decisions.
Simplest solution about clock synchronization: we use the clock (the physical clock) to synchronize. Example from Tannenbaum (probably). [he's doing a makefile example to explain how a unix systems deals with file modifications, usinig timestamps, to avoid recompiling the whole project]. In a makefile example, what if the compilation is performed through a RPC? The two machines might have a different clock. Unless we have a way to synchronize them, the system won't work. In the image on the slide, basically, even if the output.o was created BEFORE the output.c file in ABSOLUTE TIME, the fact that the two clocks are not synchronized makes the makefile think that output.c (2143) doesn't need to be recompiled (output.o is seen as created at 2144 instant). This is a problem.
S*it we have to understand how physical clocks work.
In computers, we should specify the reference time, UTC time (Coordinated Universal Time). it's a standard time. The time is related to a astronomical phenomena. There are several institutes in the world that study this topic. There is someone that, once in a while, takes away some time (strong powers) to adjust the time. They just observe the universe and take some time away when needed.
What kind of clocks do we have anyway? How do they work? with crystal of quarts that vibrate at a constant frequency, so we kind of compute the vibrations that occour in a second and we measure them.
But a more precise way to measure time is by using atomic time. TAI (International Atomic Time) seconds are of constant length, unlike solar seconds. Leap seconds are introduced (once in a while) to keep our time in phase with the sun. What is a second? A bilion of transitions of a cheesion (???). A SECOND is just an observation made by looking at the universe. The time that passes from when the sun is precisely above the earth and blablabla. If we keep the atomic time, measured by the TAI, we have something very periodic. But really, seconds are not always the same. That is why leap seconds are introduced once in a while to keep the time, our time, in phase with the sun. To get the UTC (TAI + leap seconds), we have to contact one of those institutes.
In reality, we can have a node that perfectly measures time. It doesn't exist really: there are some nodes that go slower, some go a bit faster (it depends on the quartz), and if we let them do so, bye bye, our clock goes in the wrong way. We have to make sure our nodes synchronize with UTC, and the synchronization should be performed frequently. Depending on how imprecise is our quartz, we have to calculate the frequency at which we have to synchronize.
So, how do we synchronize?
Method 1: Using GNSS.
GPS is the most used Global Navigation Satellite System (GNSS), and is used mostly in smartphones.

Since this is a collaboration between multiple satellites, those satellites whould synchronize with each other. To do so, they have atomic clocks on board. If we can get the signal from the satellite and get their position, we can also obtain our time. ERROR: GPS sends signals to satellites. This is a MISTAKE. GPS, that is in our phones, is a GPS *receiver*. It recieves signals, but doesn't send them anywhere. The satellites sends signals, and our phone GPS recieve the signals. What's in those signals? What are those satellites saying? Timestamps. They are telling us what time it is. They also have an ID that identifies them. The GPS is actually measuring the time that the signal took to reach our phone (the phone knows its position and the one of the satellite), and tries to build the right time. There are some approximations used of course. But the time obtained by GNSS is very accurate, in the order of micro/nanoseconds. But it really depends on the accuracy of the GNSS. This is one way in which we can synchronize. This is one possibility, if we have GPS in a DS, we can use satellites. Keep in mind that in DS we can't have the same exact time in all the nodes, we need approximations.
Trilateration idea (slide 10): supposte that 14,14 is the satellite. -6,6 is another satellite. My phone can tell the distance from himself and the satellite. If he queries the distance also from another satellite, he can close the search to just two points. With a third satellite (trilateration), he can know his true position.
Method 2: Christian's algorithm and Network Time Protocol.
There are servers, on the earth, that are connected with atomic clocks, and know the right time. Some computers are directly connected with atomic clocks, other are connected at high speed to them.
When we communicate with another computer, some time passes. All nodes would like to have a precise time. All computers connect to an NTP server to know the right time. If a client asks to one of those precise servers the exact time, some time passes before the response is given, of course. Somehow, we need to estimate the delay, also to understand the true time. How is it done? This communication client-server is done several times. Since we want an estimate of the latency bw client-server to get the right clock time, we make this several time and do an average. When the client communicates with the server, four things, at four points in time, happen:
T1: the message is send to the server.
T2: the message arrives to the server.
T3: the server sends a response to the client.
T4: the client recieves the message. The precision that this entity reaches is bw one and 30 ms (with the GPS we can be more precise), that is generally good. The frequency at which we should run NTP depends on how we want our approximation in DS. We should also ask ourselves which is the worst synchronization we can get, and accordingly decide the number of NTP request we do in a time unit (frequency).
Method 3: (Internal Synchronization) the Berkley Algorithm. If our machines interact with the real world, we'd like the time used inside the system to be precise in respect to the one outside the system. Example: let's say we have three machines. I decide that one of them has a role to be a time daemon (it's always active). The time daemon asks all the other machines for their clock values. It is likely that the times provided are a bit different. The daemon will understand how far behind or in front are the clocks of the other nodes. We are NOT considering the latency between the nodes (we suppose the latency is 0, like in a fast cluster). The daemon, what does it do now? It doesn't force the other ones to have its time: he makes an average out of the different times (his and the ones of the other nodes) and forces

that clock to all the nodes, himself included. This Berkley Algorithm is used when there is no need to syncronize with UTC, but just synchronize the nodes in the system. But why average? The idea is: let's try to not have too big displacements. I want to minimize them. What happens in reality? The daemon tells other nodes the time they must have. But some nodes might be, in this way, "be sent back in time", so we could have inconsistency. I can't have a file that comes from the future. What is done in practice is: the time will always go forward, but since clocks inside a computer are anyway counters, the time is slowed down until we reach the average time we wanted (that is, until all nodes are synchronized).
So: UTC, Berkley, and now...

Lamport's Logical Clocks: very cool idea.
Sometimes, the nodes don't need to agree on their physical clock values. Sometimes, it's enough to make the nodes agree on the order on cerain events (fix the before and after, not on absolute time). There are many cases in which we'll see that we have to find a solution for mutual exclusion. The logical clocks idea is: decide the order of events. An event is an internal event or a message being sent or received. An integer counter is used at each node as a logical clock. It is incremented every time an interesting event occurs. In this way, the makefile problem is solved: the event of modifying a file happens before making its .o file. So, this counter is incremented every time an interesting event occurs.
If a and b are events, a --> b means that a happens before b. It is a transitive relation. C(a) is the logical clock value assigned by the process where a occurs. The value of a logical clock can only increase. So, if a --> b, then C(a) < C(b). Some events can be: send a message, receive a message... Anyway, we can't always say that a certain event occurs before another one, it depends on where the events happened. On two different processes, it is not always possible to order the events.
Idea: each time a process receives a message from another one, the clock instant of that message received is C(message sent) + 1. The logical clock is basically adjusted based on the time of the message received.
So, LAMPORT'S ALGORITHM is cool, and will be in the exam. Slide 25: it is an example in which Lamport does **NOT** apply. Applying Lamport means changing some values. 56 becomes 61. 64 becomes 69 (61 + 8, it has to stay 8 events after).
The algorithm is run in a middleware, that is in the middle (lmao).
We'd like total order in the timestamps. It may happen that two events have the same timestamps. To reach total ordering, we can modify the algorithm by attaching a process number to the timestamp of an event, or anyway a ID that uniquiely identifies a process in a DS. If we attach this number to the timestamp of the logical clock, we can distinguish different values. The timestamp of event e, P_i, is C_i(e), where i is the process number. In this way, if two events have the same timestamps, the one happened before is the one with the lowest process ID.
Exercises till slide 30.

Lezione 6 - 06/04/2022

We'll see an election algorithm for coordination and a mutual exclusion algorithm.
Last time, we saw Lamport's Algorithm, and we said that we have a few ways to synchronize time

between different nodes. If we need UTC, we can use GPS and GPS reciever, or the NTP protocol (Networl Time Protocol). In this case, the time comes from servers on earth connected to an atomic clock directly or indirectly. Sometimes, we don't need to sync with UTC, we can use Berkley's Algorithm for synchronizing. Another idea is Lamport's Algorithms, that just imposes a total order on the events that happen in a distributed system.

So, Lamport: each event has its own timestamp, different from the others. Also, total order of timestamps does NOT mean that we know the actual relationship between any pair of events. Assuming we just have send and recieve, what are the clock values of the events a-g? (slide exercise).

Application: Totally Ordered Multicasting. A famous problem id DS. Don't confuse it with multicasting in networking, we are above TCP/IP, we are in a middleware layer. Suppost we have two replicas of a bank database. Imagine they are in two different places in the world, se the latency is high. Say we start with 1000 euros as balance. We put 100 euros in the account. This is the deposit of 100 euros. Doing so, we update, with two messages, the two replicas. If we don't have a system that decides that the order of operations would be the same in the replicas, i could have some problems. If our money are invested and we earn from interests, if this is applied to 1000 euros or 1100 euros we have inconsistency! So we have to apply an ordering. Here, Lamport is useful because it allows us to solve the problem. Assumptions: the network is reliable, non messages are lost. Messages from the same sender are recieved in the order they were sent. A process P_i sends timestamped messages m_i to all others. The message itself is put in a local queue i. Every process has a queue. All incoming messages are queued in its own queue, according to its timestamp, and ACKed to every other process (send and receive events for messages and acks are totally ordered with Lamport). So, let's consider a process in each node. The person in the slide is P1, the other one is P2. The replicated databases are two other processes, P3 and P4. There are 4 nodes in the system. If I want to deposit, I send a message to both databases. Each message has inside of it also the timestamps. A process P_j passes a message m_i to its application if:
-m_i is at the head of queue j AND
-m_i has been ACKed by each of the other processes.
Eventually, all processes will have the same copy of the local queue, and so all messages are passed to the application in the same order everywhere.
If, by any chance, the messages have the same logical clock, we give priority to the one that has lower ID among the processes. Slide 34: try to apply the algorithm then check the solution.

MUTUAL EXCLUSION

We know monitors, semaphores, etc.. to solve internal concurrency problems. But here we are in a DS, and the problems are different. We have messages as our main tool here. There are also here critical regions that can be accessed by a limited number of processes/entities. To coordinate those critical regions, we need a coordinator, someone that decides who can access to a critical region in a certain moment and who cannot. It also must store the requests of other processes that right now cannot access the shared resource, because another one is using that. The coordinator will put those waiting processes in a queue, with a certain policy like FIFO. Issues: starvation. If process 1, that is accessing the shared resource, crashes, it will never release the resource. Also, this is a centralized solution, that has

different problems. If the coordinator is offline for whatever reason, nobody can handle the resources. Classic Single Point of Failure, since this is centralized. Also, bottleneck.
Now let's see a more interesting solution, a distributed algorithm. (Later we'll see what we can do in the centralized system if the current coordinator crashes: the other nodes will decide the new coordinator by themselves). QUESTION: tell me about algos of mutual exclusions. What are important assumptions?
Assumptions: total order of events and reliable message delivery. If a process P wants to use a resource R it builds a message containing:
1) the name of R
2) the ID of P
3) the current timestamp
Slide 58: P0 uses a queue to remember the other processes that wanted to access the shared region. After it has finished using the shared resource, P0 sends an ok message to ALL processes that are in the queue. So it's not really a queue, it's like a ste. P2, that already recieved OK from all other processes beside from P0, now that he recieves the OK from it, can start using the shared resource.
Probems of the distributed system: another issue is communication. If a Process wants to use a resource, he sends a message to all the other nodes, that could be millions. If we know the resource is of interest of just a subset of processes, we should send this message only to this subset. Another extremely algorithm that tries to avoid this broeadcast to everyone: suppose to organize the processes in a logical ring, where a process can send a message just to the one ini front of him (if one crashes, the ring closes itself, it's fault tolerant). I didn't understand lol. Something about token. **Exam: show one of the algos for mutual exclusion seen in class: the number of messages, problems and so on...).**

Election algorithms. We'll talk next time about fault tolerance and what we can do about it. Sometimes, we just have to do some assumptions even if they are not the reality. It's impossible to keep consistency in a system when the system is asynchronous. Internet connects our computers, and there are delays that cannot have any bound: we don't know when the connection will come back. In some cases, we cannot assume that that the connections will come back. Algorithms for reaching consensus: will be useful to understand Block Chain. But now, let's focus on this. As a practical example, the simplest algorithm for mutual exclusion is one using a coordinator. Up to now, we implemented only synchronized algorithms. It's not btw the best solution in terms of scalability. We can have more coordinators to allow multiple points of failure. Now we consider the problem of having only one coordinator, but when it crashes, the other nodes elect another coordinator. The algorithm we'll see elects as coordinator the process with the highest identifier. But why? Let's define the identifier(Pi) as <1/load(Pi), i>. In this way, the coordinator node is the one with less work. The one less busy will do the coordinator. Basically, our algorithm elects as coordinator the one with the lower amount of work (in this case, highest identifier). But if we know that the coordinator is the one with highset identifier, why do we need an algorithm to elect someone? Because this process might be switched off at any point during the execution.

1) Bully Algorithm.
Suppose that 7, the highest identifier, crashes. 7 was the coordinator btw, so now what? 4 was asking something to the coordinator, but he's dead. 4 doesn't know if 7 is dead or busy. But anyway, suppose we have a timeout. After that timeout, we assume that node is dead. Now, since 4 knows 7 is dead, he

starts an election process. But 4 DOES NOT do a broadcast. 4 knows the IDs of the other processes, actually knows the IDs of the ndoes with an ID higher than his ID. So, he sends a message to 5, 6, and 7. Yes, 7 too, because he might rise from the dead. Supposing it still is dead thought, 5 and 6 reply to 4. But anyway, after 4 recieves an ok, he is content. Because he knows that there will exist a node that will be the new coordinator. Let's say 5 recieved this election request. After the OK message to 4, 5 issues an election request to 6 and 7, and AT THE SAME TIME, 6 might be sending an election request to 7 (we are in a concurrent environment!!!). Ok, 7 doesn't answer, but 6 yes. After recieving the message from 5, he sends an OK message to 5 too (he sent an OK message also to 4 before). 6, after finding out that 7 is dead, sends a broadcast to all other processes to tell them that he is the coordinator. How does 6 understand that he now must be the coordinator? He asks to all the processes with ID higher than his. If no OK comes within a timeout, he assumes to be the process with highest ID, and so he becomes the coordinator.
[it might also happen that the new coordinator dies immediately after he was chosen. But this is no problem, a new election will start].
Another algorithm, slightly more complicated.

2) Chang and Roberts algorithm, or ring-based election.
The previous algorithm needed all the processes to know the addresses of all the other nodes, and this was a heavy assumption. So, in this case, the idea is: can we use a lgocal ring again (in which i know the addresses of the first d nodes in front of me), in order to carry out a new coordinator?
The goal is electing the active process with the largest ID that is alive. The basic idea is very simple: one process understands that the coordinator is not alive anymore, and he has to start an election. So, he sends a message to first available process in front of him. The message is like <Election, ID(P_k)>. The successor recieves the message. When P_m recieves <election, ID(P_k)> the two ids are compared. if k = m the message came back to the sender. So, the sender becomes the coordinator. If instead P_k < P_m, then P_m changes the content of the message from P_k to P_m (the highest) and forwards it. In the end, the coordinator sends a message <ELECTED, ID(P_m)> to the next process, and the reciever now knows that P_m is the coordinator, and, unless k = m, the message is forwarded.
The participant - non participant features are used just for efficiency reasons.
If the communication to the next process fails, i send the message to the next next one.
In the worst case, how many messages are exchanged? The worst case occurs when the process with highest ID is the anti-clockwise neighbor of the one that starts the election. N-1 to reach the highest node, onther N messages to conclude the election and N messages to announce the coordinator. So, 3N-1. All fo this assuming that the IDs cannot change during the execution of the algorithm!

Lezione 7 - 13/04/2022

IL TEMPIO DI ASMETA (by Davide).

Fault Tolerance and Consensus.
Today we'll see those things. With consensus we mean reaching consensus in a DS, among the different

nodes. Also, remember that there are some things that are not possible to solve completely. There are approximate solutions that are effective, and this is related to blockchain (we'll see it in next class). All the algos we've studied together up until now had problems if a node failed. All those algorithms have to be refined in order to resist some kind of failures, being of the nodes themselves or network issues. This topic also relates to fault transparency: no one should notice a failure in our system, expecially the users. Being fault tolerant is related to what is called "dependable suystems". The dependability implies:
-Availability. At any time, the system works correctly with a certain probability.
-Reliability. The ability to run correctly for a long interval of time (uptime). A system can be almost always available, but once a day it restarts. It is available, but not very reliable.
-Safety. Failure to operate correctly does not lead to catastrophic failures.
-Maintainability. The ability to "easily" repair a failed system.

Talking about failures, we have to talk about crashes. If a server crashes, he can't serve anymore clients. How do we deal with crashes? Let's distinguish different kind of failures:
-crash failure: a server halts, but is working correctly until it halts (lol)
-omission failure: can be receive omission or send omission. It's when a server fails to respond to incoming requests, to receive incoming messages or to send messages.
-timing failures: a server's response lies outsiuide the specified time interval. This relates to timeouts.
-response faillure: can be value failure or state transmission failure. It's when a server's responde is incorrect, in particular, the value of the response is wrong or the server deviates from the correct flow of control.
-Arbitrary failure: the most difficult failure to deal with. It's when a server may produce arbitrary responses at arbitrary times. This is called **byzantine failure**.

How can we deal with failures? We can introduce redundancy, usually we use it to deal with failures. If we set up a server, usually, we have multiple copies of it. Not only redundancy on the process: also on the machines, storage, etc.. All those redundancy are Information redundancy, Time redundancy and Physical redundancy.
Information r.: we can use hamming codes or various techniques used to recover transmitted messages.
Time r.: when a transaction is repeated if it was aborted because of a failure of one of its actions. This works for *idempotent* operations, that are ops that have always the same effect, even if issued multiple times (like a get request).
Physical r.: consists in using hardware and software to cope with failures.

Let's see an example of Failure Masking by Redundancy. If we have different redundant nodes, used to deal with fault tolerance, we should make sure that they have the same status. To guarantee that they have the same status, we have to use a consensus algorithm. So, we can make multiple copies of the same processes (this applies of course for different processes/nodes). But let's imagine that the different copies produce, given the same input, different outputs, because one of them crashed (response failure, imagine). Those outputs have to converge in a voter that decides which output is correct. In the slide 6, we can deal, at most, with one of the copies of A that is faulty, and the same goes for one of B and one of C. This is because the voters work on majority. If the majority if faulty, nothing works. This btw is a technique used by most of the consensus algorithms used also in cloud technologies.

how much redundancy there needs to be?
If faulty processes just stop working for a crash failure, k+1 processes provide k-fault tolerance. What does it mean? If the problematic processes just stop working, they won't produce "bad" values: they won't produce them at all. In those cases, it sufficies that one of them survives in order to carry out the right result.
Il faulty processes reply with wrong values (value failure, worse than before), at least 2k+1 processes are needed. In this way, the client can decide by voting. 2k + 1 processes provide k-fault tolerance (at most k processes can be faulty). Suppose we have 3 processes. One of them sends wrong values, the other two are correct. From the outside of the system, what I'm doing is that i receive three messages from the group and, on the client side, I have a piece of code that says "the true answer is the majority of the values you received". This is different from CONSENSUS.
Consensus is when every process reaches an agreement with the others. Consensus is needed for many applications (mutual exclusion, election, and so on). Usually, it is impossible to achieve full consensus (we'll see why and where). We have different kinds of consensus in literature.

Among the agreement problems we have consensus, byzantine generale probelm and interactive consistency.
First, consensus: each process will compute a value, propose it to the non-faulty processes and they all should agree on the same value.
Let's see the byzantine generals problem (Lamport is important also in this case). The original problem has n generals, very far one another, and there is an enemy in the valley. The generals have to decide if they want to retreat or to attack. How can the generals communicate with each other? They can send a guy that transports a message. But what if he dies or is a traitor? What if a general is a traitor, that sends "retreat" when they should attack? Lamport has in mind that the messanger is the network connectors, while the generale are the processes. Another formulation of the process is a bit easier, and we'll see an example. In this second one, there is a commander that proposes a value, say b, and expects that all the other liutenants agree on b. If the commander is not faulty, all processes agree on b. How many faulty processes there can be, at most, in this case?
Third problem, interactive consistency: each process proposes a value. All the correct processes agree on a vector of values where each value comes from one of the processes.

A very important result from over 20 years ago is about reaching agreement in the presence of fault. The result is about Byzantine agreement with synchronous systems. The result says that IF the system is synchronous (we have a reliable network, messages are not lost...), in order to have k-fault tolerance, you need more than the majority to reach consensus. We need 3k + 1. But we won't give a proof about this. Byzantine general problem example for N = 4 and k = 1. To be k-fault tolerance, we need N nodes.
When there is no majority, there is a special value that means "no majority here". In this case, all processes agree on this special value. In a real case scanario, we have multiple rounds of message passing.

This solution proposed by Lamport of the byzantine problem assumes that the system is synchronous. What does it mean, here? That the system waits the response of the server before doing anything? Not really. In a perfectly synchronous system, we have those assumptions:

-execution on each node is bounded in speed and time. Each operation uses a fixed amount of time and space.
-Communication links have bounded transmission delay.
-clock on each node has a bounded drift. This means that the clock doesn't diverge without a bound. We are ensured that they don't diverge for more than, say, 100 milliseconds.
Asynchronous, instead, asusmes that:
-execution on each node can occur at arbitrary speed.
-communication links have different and unbounded transmission delay.
-clock on each node has an unbounded drift.
Usually, our systems will be closer to this last kind of systems. Coordination and agreement in asynchronous systems is hard and often impossible. We often make partial synchronicity assumptions. In fact, to do so, we can use algorithms that achieve a form of synchronization.

What is a partial synchronicity? There is a paper from 1994 that explains it. We basically assume that the message will be delivered, and there is a bound for the delay. Another assumption is that messages will not be lost, and the messages will be delivered in the same order they were sent. The bound for transfer is known, but ???.

Impossibility of agreement in asynchronous systems: it has been proven with the FLP theorem (1985) luckily, we have approximations that work well.

Another theorem: the CAP theorem. It says that only two of the following properties can be achieved, all of the three is impossible.
Consistency: every read receives the most recent write or an error (implies all nodes see the same data).
Availability: every request receives a response (the system is operational at any given time), according to the semantic of the operation.
Partition tolerance: the system tolerates an arbitrary number of messages lost (*or delayed*).

Real systems relax availability or consistency guarantees (or both). You'll receive an answer, or the data will be consistent, but with some delay. Some compromises are needed. Now we'll see some algorithms that can be used for those problems.

byzantine and the bound is known but ?.

Practical solutions for the consensus problems. For the practical consensus, there is the Paxos protocol. Lamport came up with this name because he was on vacation in there. He thought that in greece there was a parliament with different people that come and go from it by time to time, but despite this the parliament must work. There are two versions of this protocol (that we'll see). This protocol, that is not byzantine in one of its variants, can tolerate k failing nodes having the majority of the nodes not faulty. Google was one of the first to introduce NoSQl databases, and wanted both the relational world and powerful query languages. So, they came up with a system that has properties from both those words. Google Spanner is one of these, and Paxos is used to mantain consistency in those databases. All of this to say that this theory we are studing is the basis of many things we use nowdays. There are conditions under which it won't make progress, but they are very unlikely to occur. Considering practical consensus,

we won't see in detail the Paxos algorithm. Lamport, for this algorithm, received the Turing award (Paxos and all the theoretical work he did before). But the probelm is that all of this is difficult to read and understand. So he wrote a paper so that it could be easier to understand. But the paper too is difficult lol. So we'll see a simpler version of Paxos: the Raft protocol, developed by Ongare and ousterhout at Stanford in 2013 as a more understandable and easier version of the Paxos algorithm. He has the same goals of Paxos. How do Paxos and Raft relate with the three properties seen before? They want to guarantee partition tolerance (works under network errors) and works when there are k nodes failures, with N = 2k+1.

Raft: it is based on leader election and log replication. The leader election is something we know: we saw the bully algorithm and the ring algorithm (we'll see another). Why election? Well, first of all, it doesn't cover byzantine failures. The assumption is that a leader can crash, but he can't act in a crazy way (can't be hacked. It's an ASSUMPTION, we ASSUME no hack attacks on the coordinator).

Google file system: it uses the consensus algorithm. We'll use Google RPC in the project, but we won't have to deal with failures. In an RPC many things can happen: the remote machine can crash for example. Another thing we didn't see is Atomic multicast: if I have a group of processes and I have to send a message to all of them, and I want that either all of them receive the message or none of them does, I have to use Atomic multicast.

RAFT: each node can be in either in a state of {follower, candidate, leader}. The leader is the coordinator. Initially, all or them are in the follower state. Now we can apply an election algorithm, and it will work. Here we use the following: if a node doesn't here from a leader, then they can becoma a candidate (don't hear = timeout). The difference from the algos we've seen is that the node we are considering wants to become the leater itself, instead of finding a new leader. The candidate then waits for some time. It can happen that two nodes, at the same time, will be candidate to become leaders. Anyway, a candidate requests votes from other nodes (because he wants to be the leader). The other nodes reply with a message. The candidate becomes the leader if it gets voted by the majority of the nodes. Now, all changes to the system go through the leader. All changes that go through the leader node must be committed: that means, the leader must communicate to the other nodes the changes. The message that the leader sends is the message that the client sent. The leader waits until the majority of nodes confirm to have changed to the new value. The leader then notifies the followers that the entry is committed. Now, the cluster has come to consensus about the system state. This process is called *log replication*, since there is a log that must be committed on multiple nodes.
In Raft, we have two kind of terms (timeouts) which control elections. The first is the election timeout (a parameter of the system), and is the amount of time a follower waits before becoming a candidate. It is usually a random value between 150ms and 300ms. This is used in order to make more probable that just one node becomes the netx candidate. The first candidate starts the second timeout (term), and sends out its requests to the other nodes. If the other nodes have not voted for a candidate yet, he sends a reply, and resets the first timeout.
There is also a heartbeat timeout. This timeout determines how frequently does the leader send a "i'm alive!" message to the other processes.
The followers respond to each Append Entries message, the messages (different from heartbeats) that

tell the followers how to modify a entry.
(A CANDIDATE ALWAYS VOTES FOR HIMSELF, THAT VOTE COUNTS FOR THE MAJORITY OF VOTES). A process can vote for just another process during a term. A term is a timeout, but also a "phase", and in fact each message transmitted for the election has inside of it the current term of the sender, so that each node knows to which term does a message refer to.

GFS - GOOGLE FILE SYSTEM. Let's see this first kind of distributed file system. When Google started acquiring a lot of data, they decided to write an application just to manage those big quantities of datas. Most of the operations required were append operations, so the way the datas were uses were different from old applications. The simplified architecture of this GFS is: a file is completely distributed inside different nodes. A single file is divided in chunks of different parts, and all of them are distributed on different machines, and also replicated of course. [bittorrent: does its job considering chunk of files]. Those machines are chunk servers. If a client requests a certain file and a chunk index, a master needs to know where the chunk is (it could be on different servers). In this case, the master gives an address to the client, so that the client can reach out the chunk he wants (so there is a level of indirection). Where does this has to do with consensus? Well, there is a single point of failure: what if the master crashes? This is similar to kubernetis and Borg (system used to handle clusters with distributed computation, while here is distributed files). Well, usually, we have a group of masters at work, so that if one fails, other ones will take care of its job. But those need to have a copy of the work of the old master. We want a replica of the master, and we need a consensus protocol to ensure that all the master replicas have the same data. (The slides regarding this topic are in the first set of slides of the course).

Lezione 8 - 27/04/2022

Distributed Ledger Technologies and Blockchain.

This class was introduced some years ago when Blockchains started to gain more and more popularity. We will focus on the DS part of course, but there are many possible applications. It's an interesting topic, he says. We'll try to understand what's going on "under the hood".

First thing: is blockchain a buzzword? In part. Nowdays we hear about blockchains and NFTs, but it's nontheless considered a key technology. It is considered by market reasearch among the "essential eight technologies", along with IoT, Augmented Reality, Virtual Reality, AI, etc..
What is a DLT? (since blockchain/bitcoin is a technology based on a DLT) Well, a DS with decentralized control (that is, there is no controller), in which nodes are run by different entities that do not trust each other and may even be malicious (Byzantine behavior). We need fault tolerance of course. And then, the third characteristic of DLT, is that a copy of data records (financial transactions, medical records, sensed data, etc..) is stored at each node. There is a consensus problem in all of this: the node need to agree on a history of data (for example, a "ledger" with a history of financial transactions). If we think about transactions, we have transactions of money from people to people, that must be consistent. So, we need consensus about the ordering of the operations. Usually, we don't have distributed ledger. We trust an institution. Here, no, there is no trust. Reaching consensus among a lot of processes is not easy at all.

Now, let's see the story of blockchain.

In 2008, *a whitepaper appers*, and is entitled "Bitcoin: A Peer to Peer Electronic Cash System", from Satoshi Nakamoto, a person that doesn't exist. It was a peer-to-peer system for electronic cash. This guy/group of people, one month before this paper, registered the name "Bitcoin". Then, people tried to implement what was said in the paper, and in 2009 we had the first client and open source implementation of bitcoin. In 2012-2013, people recognized that this was a brilliant idea that revolutionised the way cryptocurrencies were created. Then, people realized that this technology could be used also to improve DS. So Ethereum, another DLT, was created, which wanted to go beyond cryptocurrencies in order to handlinghandling tokens and contracts (pieces of software).

Indeed, blockchain and DLT go much beyond bitcoin. Nowdays we talk about foodchain, that allows us to know everything about the production of some food. The ledger, in this case, is the history of that food. Also healthcare is a domain in which we can have blockchain. Also art pieces can be subject to blockchain. We can divide the property of an art piece between different people thanks to blockchain. Also gaming is affected, expecially with Multiplayer Online Games.

Blockchain is a ledger (REGISTRO) of transactions. A transaction is a data record. An example of a bitcoin transaction is "Alice transfers 0.15 BitcoinCents(BTC) to Bob". This is represented by input transactions and output transactions in Alice blockchain and Bob blockchain. There is a lot of crypto stuff underneath, since this is bitcoin in particular. Why? Think of Byzantine's general problem: if the General sends a message, there could be problems. But here, we'd like to know that if Alice sends 0.15 cents through the network, then it was really Alice who sent those money. To do that, we use the hash of a public key (actually, the transactions are digitally signed with the private key of the sender). The idea is that: each node/person behind a wallet in bitcoin, has an address when is doing a transaction (like IBAN, a bank code). This is the public key, that is hashed and becomes the address. Also, a person can use many of those addresses. And we can also use pseudonims for those addresses. In some cases, we also have to change the pseudonims, because if too much informations about a pseudonim are public, we can have anonimity issues. Anyway, in a transaction, we have two numbers: the hash of the public key of the sender and of the receiver. The whole transaction is signed with the private key of Alice, so that anyone can check if that message really came from Alice (so that, if someone changes the content, we can find it out). The transaction is then broadcasted to all the nodes in the system, even in different order. When a node receives a transaction, it validates that transaction. What does it mean? In bitcoin, for example, a transaction is validated by checking if the sender of the money actually has that money. The validation is application-dependant. Validating means following business rules and making sure the sender has that money. This is not all. Those transactions go to different nodes, and they are considered pending. They are not, yet, part of the chain.
So, the idea is that I get a group transaction (I actually group the transactoins). There are some nodes that voulonteer and group those transactions forming blocks, and giving them (the blocks) a timestamp. We know that there are problems with timestamps. So, the timestamp of the transaction is not really relevant, but they exists. When n transactions are grouped into a block, we give them a timestamp, valid for all the transactions of the block. The transactions are organized in a tree fashion (Merkle Tree) inside a block, because in this way is easier to find a transaction inside of it, if there actually is that transaction.

Actually, the tree contains the hash of those transactions, organized in this tree data structure. We can say that each transaction is represented by a specific hash. What else is there in a block? The hash of the previous block (otherwise is not a chain). But what is the hash of a block? Well, all the things inside of a block (timestamp, the tree of transactions, a nonce, and its previous hash) are taken as a single object, that is hashed, and that hash represents that block.
So, we have group of transactions that are in a chain: this is blockchain.
Problems: there is unpredictable latency, imprecise clock synchronization and faulty (malevolent) nodes.
So:
-the order of arrivar of transactions may be different at different nodes. This is the same problem we've seen in totally ordered multicast (deposit and interest calculation in different bank databases).
-if concurrently two nodes are grouping transactions, is not given that they give the same group of transactions.
-different nodes may build different blocks.
-different nodes may end up with different chains (no consensus).
So, we'd like consensus, in particular, on the sequence of blocks (each block has a timestamp, remember, this is a TEMPORAL sequence). The goal is that each node should eventually have the same copy of the chain. THIS IS WHAT THEY HAVE TO AGREE ON: ON THE CHAIN, THE HISTORY, THE WHOLE CHAIN. Otherwise we have different ledgers.
So, the main idea is:
-compute the hash of each transaction, and of a block.
-compute the hash of a block including the hash of the previous block in the chain. This is actually smart, because any change to the previous will have an impact on all the following ones.
-include a trick to make the computation of the block hash expensive, but very easy to verify. We want to make it expensive because so we won't desire to change the hash of a block, since it is complex to re-compute the has of all the following nodes. The algorithm is called "Proof of work". It says: the computation of a block hash is expensive, but is very easy to verify if it's correct.
-make the nodes compete on this computation with a reward, and have the winner propagate its computed block to the others (it is like "electing" a node to impose its block). Basically, the first one that computes the hash of a block wins. It can take an average of 10 minutes to calculate those values, but the system is such that, if we look at the distribution of how different nodes solve the puzzle (compute the hash), is such that is very unlikely that two nodes will solve the puzzles at the same time. It can happen, but rarely. So, to recap: a lot of nodes in the network try to solve the puzzle. The first one that computes the hash wins, and communicates this to all the other nodes. They can verify immediately if that is a correct solution, because of how the algorithm works. The rubik cube is an example: it's difficult to solve, but easy to verifiy if a solution is actually a solution.

Let's try to understand a bit more the hashing. We have a cryptographic hash function f (for example SHA-256). f(A) has fixed length (for example, 256 bits independently on A's length). It is also collision resistant, that is, A neq B implies f(A) neq f(B). And it is very difficult to find A from f(A). Also, it's very easy to compute f(A) from A, so it's easy to verify if, given A and B, B = f(A).

The puzzle is: find a Nonce value that will make the result of a hashing value less of a certain number.

The idea is that: let's suppose I'm a malevolent node. I want to change a block of a blockchain. Not the last one, but the 20th before the last one (for example), and I want to to that because, in such a way, it will seem that an individual actually sent to me some money, insteo of sending them to the right receiver. So, I change the content of the block, and mine it, calculating the new hash.
1) This takes me a lot of time, because I not only have to mine that block, but also ALL the following ones.
2) Even if I manage to do so, remember that data are scattered among all nodes: every other node has a copy of the blockchain. But if we are 100 individuals, and 99 of them have the same (right) blockchain while I'm the only impostor with the modified one, my blockchain isn't considered truthful, and so the false transactions that I've inserted are not considered by the others. I'd have, in order to apply them, to convince other 50 individuals (so that we are in majority) to mine the blochchain from the 20th-before block with the exact modifications I applied to it.

[The mining of the blocks of a chain must be sequential, since every block needs the validated hash of the previous one].

In the case of the bitcoin blockchain, how do we validate a TRANSACTION (not a BLOCK)? Well, we can trace back the whole chain in order to find older transactions, in previous blocks, or older coinbases (that are like bank account), and say: well, if Jackson now wants to transfer 15 dollars to Sophie, and he received, in the past blocks, 5 dollars from Amy and 10 dollars from Lucas, he can for sure do this transfer.

The miner that solves the problem sends the block to every body else, who can easily verify that the computed hash for that block, given the generated nonce, is lower than a certain value. This allows other nodes to validate that block.

Every node knows who sent the message because each sent block is signed (private key), so that the sender can rightfully get a reward when he solves/validates a block.

Blockchain PoW algorithm. The algorithm used for consensus is called Proof-Of-Work (POW). We said that, to verify a block, a miner needs to compute its hash that satisfies specific requirements, as being less than a certain value. In order to do so, the miner needs to find a specific nonce value. To solve the puzzle, a brute force approach is required. The problem, infact, is designed in such a way that it will take in average a specific amout of time to solve it, so that it is unlikely that two different miners solve it at the same time. Last, a reward is given to miners that win.

A verified block is sent to all nodes. When a node receives a block, it checks the puzzle solution and then adds it to its local copy of the chain (linked to the one whose hash is in the block). When the puzzle is solved, this news is sent to all other nodes, which can verify the solution, and if it is valid, the block is added to their copy of the chain. We can have branching in blockchain. It happens when in the network two miners solved the problem with two different blocks. So, the blockchain is branched. After a while, the longest branch will win, and the other branch will die. But that dead branch contains nontheless

verified transactions and verified blocks, so we can't just say that all those transactions it this branch never happened. So, the transactions in a dead branch go back to the pool of trasactions that need to be added to a block to be part of a chain, because they WILL eventually become an "official" part of the chain. If different nodes are working on the same chain, the one growing fastest will be the longest, and most trustworthy (because he computes the hashes of the blocks, sends the solution to the other nodes, they verify the solution, and the new block is added to every chain). We said that this is a competitive environment, infact. In order for a malicious node to change a transaction in an intermediate block, it has to re-compute all the subsequent block hashes **and prevail over other nodes in the network**. This means that other nodes should agree with the malicious one's chain. Blockchain is safe as long as more than 50% of the work being put in by the miners is honest.

There are a lot of critics to blockchain. Blockchain is imperfect, also because the consensus algorithm used is imperfect. An enrmous amount of money has been lost because of blockchain. If one loses its key, its money are gone forever. Lost keys/dead hardware (where the wallet was stored) imply lost money. There are also some risks. When transactions are added to the chain, we don't have to consider the chain immutable, because we have to be aware that some people are trying to cheat. It's difficult to change something in the past, but it's easier to change some things changed in the immediate past. A cluster of miners were able to change 6 blocks back in a chain. 6 blocks is not that much, they took one hour of time. It's easier to cheat toward the end of the chain than the remote past. Also, this Proof of Work algorithm is not good for the environment. The computing power consumption in terms of electricity and hardware cost is high, and bitcoins consume a similar amout of power of the Netherlands, in terms of TWh. It's something that is damaging the planet. For this reasons, other consensus algorithms are being studied for bitcoins, like byzantine Paxos (Raft doesn't take into account byzantine failures, btw). Byzantine Paxos and in general Byzantine solutions for consensus are being investigated. Also, bitcoins have a really small number of transactions/sec, compared to Visa, ripple, PayPal, etc.. A lot of researches have gone through consensus algorithms, and different alternatives to the Proof of Work algorithms have been proposed. The Proof of Stake is an example, that is/will be used by Ethereum. Ethereum wants, in the next 6 months, to move from POW to POS, and has been working on this for the last 6 years.

Proof of Stake.
The idea is: if you want to validate a block, or to, better, group transactions and propose to the others this group (so that that group will be added), you have to put a stake (some of your money/currecy), to say "I'll do this job, I'll create this block". They are a guarantee on my work. If I don't do a good job, I'll lose a lot of money. How do the other decide who is gonna do the validation? How do they accept the stake of a possible validator? Well, there is a distributed algorithm that decides (also randomly if there are many who want to participate) who participates to this election, first of all. In this case there isn't a puzzle to solve. A participant must also be a well-known one, must has a reputation and his money there for a long time. Based on all those factors, the algorithm decides who will decide how to group the transactions.

Smart Contracts.
Basically, this contracts may become a distributed general software capable of computing any funciton.

What's the idea? Instead of storing transactions, as seen up until now, we store a piece of software. Initially, this was a contract: an agreement, really. This contract is not stored in a trusted institution. It's just a contract that anybody knows and can verify that it was signed. Not only: a Smart Contract can really be defined as a piece of software code defininig a self-executed contract. As an example, think of crowdfunding platforms, like Kickstarter or Indiegogo. You have an idea, but no money. You publish your idea in one of this platforms and receive money in exchange for privileges. There is a contract between you and whoever participates and gives you money. In this platform, there is clearly someone gaining money through the platform (like Kickstarter), because this is a centralized solution. The idea is similar in smart contracts, but completely distributed. There is no platform. I simply publish on the blockchain my Smart Contract that says: if you, individual, give me this money, I will reward you in this way. If I reach, say, 1M euro, everyone gets his reward. Otherwise, I refund every individual with his money. This is translaved in terms of transactions of course. Anyone can see with my contract and what were my promises. This has an effect on my reputation.

Bitcoin and Ethereum are permissionless.
Ethereum understoon the power of this technology: it goes beyond financial transactions. For example they created NFTs, and in Ethereum we can create our cryptocurrency. Anyway, permissionless DLT are decentralized DL that tracks all transactions. There is no trusted third party and unconditioned access to the ledger. There is also pseudo-autonymity of participants, the transactions validation mechanism and the generation of new "coins" is performed by miners and the transactions are immutable.
Their counterpart is Permissioned DLT. They are still decentralized and track all transactions, but there is one or more trusted third parties and the access to the ledger is conditioned: some can enter, some not. The identity of the participants is known, and the transactions are immutable. Also here, the transactions validation and the generation of new "coins" is performed by miners.

Among these Permissioned DLT there is Hyperledger Fabric, a free-to-use project hosted by the Linux Foundation. In principle, this has a pluggable plugin consensus mechanism, so we can use the consensus algorithm we prefer, but the one used by default is raft.

Lezione 9 - 04/05/2022

Introduction to Pervasive Computing. We will present its topic as an extension to Distributed Systems. It's not a technical talk, but an easier one. we'll see what is a pervasive system, what are interesting problems about it, and then we'll se other topics in the next classes.
"*The most profund technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it*" (Mark Weiser, The Computer for the 21st Century). Computing in general should not be done, for the most part, on a speficic device (this is what this paper says). The idea is that it should be normal that a lot of objects have, among the others, the ability to communicate and compute, each with its interface, or no interface. There is a bit of this in the notion of Distributed Systems. Because ok, there is communication, but there is also transparency: the user should not be aware of the fact that this is a collection of nodes, and shouldn't know where they are. Bettini's

definition of a Distributed Pervasive System is a system with those main features:
-**It includes unconventional nodes**. Possibly mobile objects with computing and communication capabilities, like smartphones, smart appliances, smart meters, sensor networks, etc.. Even plugs nowdays can be part of a distributed system.
-**Adaptivity**. The system logic considers the current context and adapts the system behavior for optimizing the system goal. Every system has a goal. The opportunity to have nodes as part of the system allows them to have sense capability. They are context-aware nzomma, and we as humans are incredibly good at adapting (in general).

Those systems are also *Volatile Systems*. That is, systems that exhibit high volatility:
-**we can have failure of devices and communication systems**, that are much more frequent. A node can be disconnected from the system, and in that case, the partition problem is very frequent.
-**changes in the characteristics of communication such as bandwidth**. Also, the devices should not consume too much electricity and bandwidth. They should also be able to find other sensors/devices when needed.
-**creation and destruction of associations between software components resident on the devices**. When we saw peer-2-peer systems we saw that we needed efficient functions. [We won't deal with accidental failures in the project, the fact that the project will crash suddenly, no, that no].

Pervasive computing is an extension of Distributed Systems, and this is implied from many studies. Usually, the introduction of mobility to DS, generates a Pervasive System. In mobile computing, we have Energy-aware systems (they need energy to work), Location sensitivity (that consumes energy). We can miniaturize a lot of things, but the battery is always a problem. We then have issues related to networking, because we have different operators for different networks. When changing network, how do we keep a file transfer, for example? TCP-IP helps with this. Also adaptivity is a feature.

In mobile computing, we have some main particular issues:
-limited resources, like energy, CPU, memory...
-different types of interfaces, like blackberry's keyboard and touch interfaces.
-high variance in connectivity.
-variable location.
Computer Scientists are mainly reasarching on those topics:
-Networking (Mobile IP, Mobile networks, ad hoc networks, ...)
-Mobile information access (disconnected operations, proxy architectures, bandwidth adaptive access, ...).
-Mobile data management (spazio-temporal data management, LBS, context-awareness, privacy and security, mobile cloud services). When nodes are mobile, we have to deal with space and time (movement), so if we want to analyse how people move in a city, we have to collect all the datas and study them. LBS = Location Based Services. Context-awareness, we'll have an entire class about this. Privacy and security too is a big issue. Another topic is how to delegate some computations to other powerful nodes, like edge-computing.
-Positioning (indoor and outdoor localization, proximity, tracking). Understanding where a device is, that usually means understanding where a person is. Indoor is more difficult, because it is usually difficult to

have indoor maps, and also it's difficult to use gps in closed environments. We also have to understand when two things/persons are close together, so that they can exchange messages for example. -Software (App and mobile services design, development and testing, scalability). Mobile computing is not normal computing. We have issues about scalability, and we need to make applications scalable. There are centralized solutions, that have problems if not supported with replication.

Pervasive is not necessarily mobile. It MAY BE also mobile. The pervasive computing involves embedded computers and affordable Internet connected sensors and actuators, that listen and affect the physical world. We also have wearable devices, like rings and glasses. For the ring: many people don't want to have a watch or a smart watch, and the ring is less invasive. Also, people want fashionable things on them (the aura ring can be either in gold or silver). The glasses have sensing capabilities, they have a way to function as earphones, and can also provide us with a projection of some informations. Google worked a lot on those, but decided not to sell them. Two years ago, they bought a company that produces those glasses, but right now they are not publishing anything. Some examples of Pervasive Systems (other examples) are Smart Environment Systems (like a home, a building, or a city) and e-Health systems for Tele-healthcare, independent living and ageing well, and accessibility. IoT is the set of devices we've been talking about. So-called smart building, home, city, could be pervasive systems. COULD be because they don't always have the properties of a smart distributed system.

In a DPS there is a huge amount of devices that are nodes of the system that don't have an ip address. This can be a problem, even though they of course have ways to make them reachable. How do we contact it? Well, there is a gateway that allows us to reach it with a message. But who is using those IoT devices? Well, nfdaishdb.

A smart thing should be connected to the internet, receive data and understand the context. Then, after that, it should understand what's going on, and elaborate those data depending also on the context. We should get the data, analize them, and get conclusions. Some "Smart" appliances include: smart TVs. But they are not used so much from an interactive point of view. Smart vehicles, that is, vehicles that drive themselves. They have to understand and react to what is happening around them.

Adaptivity: a device can turn himself off when needed, adapt if the environment changes... Some devices are wearables, as we said, like wearable sensors. Some of the are used to do medicaly approved control of the heartbeat. They measure the breathing ability, and are used also to perform sport performances. Vandrico (?) is one of the many companies triyng to invest on this technology. The sensors connected to a vest cannot be connected throgh wires: it goes inside a washing machine! So they have to communicate with a network communication. The system has to adapt to provide us the right functionalities.

Sensor networks: what are they? Once they were everywhere, now a bit less but still used. We have a lot of sensors in our everyday life. A set of sensors can construct a sensor network, that also have their protocols. We will talk about sensor in the next classes and how to optimize them. For example, some problems: how do we dynamically set up an efficient tree in a sensor network? How does aggregation of results take place? Because if i want to average some data, idk if I have to take data from all of them or

distribute the results (temperature for the kitchen, temperature for the evening room...").

So, ok, Pervasive inherits from Distributed but is also different. How? We have new issues:
-Smart spaces need to be effectively used, with adaptiveness, context-awareness, anticipation of needs (like turning up the light of a room in which I'm walking in)...
-Invisibility: the interaction with users should be minimized (system transparency). That is, I put the device on an object, for example, and the user uses the object without knowing that there is the device attached to it.
-All the resources in a pervasive environment should be discoverable and dynamic association/collaboration should be enabled. This is a big issue, because if it is our home, then we set up the system that recognizes the people that usually are at home. But in a public building, how do we discover that there are new devices? This is the idea of plug and play. But, it's a nightmare to have a new device and not having the right drives for it. Nowdays, this problem is common in IoT devices and sensors. This is because those devices are not plug and play nowday.

The current reasearch topics in pervasive computing are:
-Context and Activity Modeling and Recognition. This means how to represent rich context and how to reason with it, and recognize human activities from streams of sensor data.
-Crowd Sensing: distributed acquisition of data from users.
-Energy Analytics.
-Pervasive Health.
-Pervasive Transportation.
-Edge Computing.
-Security and privacy.

Smarthome System Demo.

How much do those devices cost? Boh.

"SECURE": example of eHealth project. "Sistema intelligEnte per diagnosi preCoci e follow-Up domiciliaRE". It was funded by MIUR and Regione Lombardia. It has 4 partners: Health Telematics, Network, FatebeneFratelli IRCCS, EWLab, Università Statale di Milano & 3 caramelle.

Lezione 10 - 11/05/2022

Sensor Data acquisition and management

The first part is again quite """easy""", we'll go more in-depth about sensor devices & networks. in the second part, we'll have a more methodological part. We'll see SLIDING WINDOW! The exam lasts one hour - one and a half at most. And is sequential. THERE IS NO PENALTY FOR A WRONG ANSWER (It's something he regrets to do lol). There are 2 open questions: they can be either the execution of an algorithm (one of them is an algorithm) or discussion about something. What transparency is needed? Which one do you know?

Introduction to sensor devices, sensor-rich devices, how to manage and query sensor data and sensor-based systems and apps. We'll go through this. First of all: we are talking about low-battery devices, so we have to be careful about them and their usage.

Transducer: both sensors and actuators belong to this family of transducers. They are devices that transform one form of energy into another one. For example, a microphone transduces a sound (physical phenomena) into an electronic signal and sends it to speakers in the room. So, the physical phenomenon is passed throught the transducer, and then becomes an electronic signal. The sensors detect physical changes and produce an electrical signal. Many of our everyday objects have sensors, like accelerometers in our smartphones. We need to understand the specifications of the sensors and what the application really needs out of it. There are different types of physical sensors:
-Motion sensors: measure acceleration forces and rotation forces, along the three axes.
-Environmental sensors: they measure environmental parameters suc as ambient air temperature, pressure, illumination and humidity. In the project, we have an environmental sensor.
-Position sensors: measure the physical position of a device. This includes orientation sensors and magnetometers.

Smartphones became useful when MEMS were introduced (Micro Electro-Mechanical Systems). What are they? They were invented by chance, because they were working on printers but came up with those MEMS, and understood that could be used to produce tiny sensors. There are some elastic/mechanical structure where some parts are fixed while others are moving. The ones moving change the magnetic field depending on the speed of the device on which they are mounted. It is then possible to measure the acceleration thanks also to the use of capacitors. Anyway, the idea is that we have micromachines, micro devices, that are very small, and revolutionised the world of smartphones and mobile computing in general. Some environmental sensors are sensors for light, temperature, humidity, magnetic fields, pollution, pressure, electrical fields, sound, chemical elements... In a house there can be a lot of smart-home sensor devices. Most of those devices are related to health, like ECG, body temperature, blood pressure, heart rate... but also oxygen saturation, perspiration, skin conductivity (important for covid), blood clucose, EMG, EEG, etc..
BLE beacons: Bluetooth-Low Energy applications, in which we want to limit the power consumption. Some years ago, there was a big hype about those beacons, and on the phone one could find a person and understand where he/she is, and advertise that person if it gets close to a certain shop for example. The problem? Beacons doesn't work very well. It has not high precision and there are a lot of interferences. Still, it's a considered technology.
A bit more technical: difference between SENSOR and SENSOR DEVICE. A Geiger count sensor measures the radioactivity. It's a DEVICE, and the Geiger count is a part of the whole device. The device is divided in sub-modules, one of which is a sensing subsystem, that collects data from the environment. Another one is a Processing subsystem, or the Wireless communication subsystem, and the power source of course. We also have Local data storage for caching, and a Wired interface and then a Mobilizer, that handles the change of configuration.
Also virtual sensors exist, that are serviecs and apps tht provide context data to rempte clients. For example, a weather web service or Google Places API revealing semantic location. This means: we don't

just want to know the position of a person, we are mostly interested in what streets and shops there are there.
Actuators: they are a particular type of transducers, usually powered. They convert energy into an action (motion, switch...) upon receiving a command from a control system.
All these sensors and actuators need to communicate, but in most cases they have to rely upon wireless communication. Wi-Fi is the most used, and there are researches in this topic because we have different issues, like bandwidth and limited energy. Also, we have to consider the topology. We are passing from a star topology to a mesh topology, in which we don't have anymore a central point of connection. But what's a mesh? Well, the idea is not to communicate to the acces point directly, but to pass through another device that reaches the gateway, in order for a device to talk with the others. Different devices have different roles when powered, and the ones with more power can act as intermediate points.
Zigbee and Z-Wave have been competing for years in order to be the best in terms of networking technology for sensors and actuator devices. But then we also have the Thread technology. The idea is: each device has its own ip address (IPv6), and there can be more than 256 devices in the same network. THread also has an interesting feature in the Mesh: each node can be either in one of two states. Some of them can be considered as leaders, and if someone fails, they have to elect a new one. If I have devices electrically powered (not limited to energy consumption), they should act as leaders.
We should also know that in these networks like Zigbee and Z-Wave there are gateways (routers/controllers/base stations). They are really USB that are connected to antennas (that are very small). The antenna is then connected to an access point/modem with for example a ethernet cable that allows access to the internet. Now, what processing do we need to do with the data we get fro the sensor devices? Where do we store them? In a database, usually. Ok boh. [The home connected over IP changed name to "matter"].

Managing and Querying Sensor Data.
In a smart place, devices can appear adn disappear frequently. When they appear, "network bootstrapping" is needed: registering the device to the network and assigning a unique address, just like in DHCP. For example, if we enter in a hotel, we'd like to use the smart devices immediately with our phone. The network by itself could be extended over the bounds of our room, so we also have to deal with security and privacy. With Z-Wave, association is a pain in the ass. Usually we have to struggle a lot to do it. To associate a device with a new network, we need to de-associate it from the old network, and this is an issue that makes this technology primitive. Anyway, data acquisition.
Let's see basic ways of getting datas from sensors. When we want data from a database, we use queries. But to get data from a sensor, it's a completely different thing. Infact, continuous queries need to be done, because the data change frequently. The batch processing is a first method to acquire data: we buffer on the sensor and do offline processing on the base station, or a remote server. So, we store data on the sensor (in a BUFFER) and send those raw data to the base station, that computes those data and does all the processing required. Clearly, this can work if the communication can support the flow of data coming from the sensor. So, we get a precise answer, because we delegate, and there is no processing on the sensor, but we also have a high communication cost. Also, we have to consider that the base station receives all the data with a delay, due to the fact that this is an asynchronous system. So, this is the simplest basic method, that is rarely used because of high communication => energy

consumptions because of antenna. So, what is done is sampling. We consider only a part of the sensed signal, for example, we read data only at a given frequency. This technique clearly has some error, but we can anyway provide guarantees about the error we introduce, so to estimante the error we produce.
Let's now see the SLIDING WINDOW technique. It doesn't give a precise answer to the query, but an approximate one, based on a group of consecutive readings, for example using a temporal window. A temporal window can be considered a period of time. The size of the window is given in time units. But are there other windows, like windows based on a certain number of data sampled. The cost of this technique? Well, intuitively, we are not throwing away datas as we did before. Here, we usa an aggregation function in the window to condensate the data. We do an average of all the data we collected: if the window is of one hour and we did 100 measurements, we take the average of those measurements. This technique doesn't give a precise answer, since we are approximating, and also some CPU time on the device is needed for those computations to occour, but we save transfer times. So, the base station receies a continuous stream of approximated sensor readings, with a small delay. The delay is because we have to fill the window before we can compute the average and send it to the base station.
So, in, for example, a sliding window of an accelerometers, we have a stream of measurements in consecutive periods of time, and we save those measurements in a window. If the window has 4 positions, we compute a sort-of average out of those values. In general, we want to compute "features" out of these data, like max, min, avg, etc.. The features tell us something about what we are studying. When the window is full however, we compute the features and send them to gateway. There is a computation cost in terms of CPU device, and another cost in terms of the space we need to allocate in order to keep some space in the window. But the prevailing cost is the one of sending the values of the function. However, it is usually much better than the batching processing, because there we send all the data, not just the values of the functions. We can do this function processing also with the batch processing, but we compute the function on the gateway. This, anyway, is not scalable.
Windows can be disjoint (slide 28) and be a partition of the stream of data. The windows are contiguous, but that's not what we want: usually, we want them to overlap. So, we have to know: the size of the window and the overlapping factor.
Given a stream of data, the size of window (in terms of data) and the overlapping factor, we compute the function.
Thanks to overlapping factor, we can consider the correlation between more near values.

Now let's talk about advanced methods: how can we save energy and improve data quality? There can be noise that transforms the datas, and the cost for transmitting 1 bit is the same as doing 1000 CPU operations. Energy-saving protocols like Zigbee and Z-wave have been studied for this reason. (this is not in the textbook).
Another idea is: why don't we distribute computation on multiple nodes that are organized like a tree, where each node communicates with another node or with the base station? This costs less than communicating always directly to the base station.
Duty cycling is what happens in Z-wave. The radio (or whatever it is) is put on sleep mode most of the time, and if I contact it while it is asleep, I have to wait the end of its duty cycle in order for him to receive my message. There is a trade-off between duty cycle and operative time of course.

Mobility-based advanced method. The idea is that the base station may be mobile, and nodes may be carried by people, cars, animals, etc.. So, we have to adjust communication between nodes based on their current location.
Model-based approach: the idea is that this phemonea we are observing usually has a certain regularity. The data are not always random. According to the season and time, we can expect a certain distribution for the temperature data. Not in all cases, but in some of them we can have an expected behaviour. So, once the apporoximation function has been computed, we send the value only if it differs from the model. If I have the sensore node and I understand the data, I look at the value of the average and the model I have: is the average at un valore simile a quello del modello? Sì, allora non faccio niente. But if the node is dead, the base station assumes that the model is being followed, but really the node failed. We can use heartbeat messages to avoid this. In data acquisition, we observe some real data, but if the sensor has the model and its data follow the model, we won't communicate any value. The approximation value delta is also given: if the distance between the measure and the model is less than delta, we don't communicate.

Data acqusition can be done in two ways, BUT THERE IS A PROBLEM! Ok, I, the user, give the query to the gateway/base station, whatever it is. In a pool based approach, the base station makes an explicit query to the pool of sensors. In a push based approach, instead, the base station sends the expected behavior to the pool of sensors (sensor network), that is a function that tells the expected value. If the measured value is too different from the model, it ise sent. Otherwise, no.
The model-based approaches are anyways used also for data cleaning, that want to identify noisy readings. Those readings are probably due to malfunctioning, because they are too much outside from the expected value. Sensor Data Cleaning works as follows: data cleaning is usually done at the GATEWAY, not on the sensor nodes. Also because at the gateway we can compare different values of different sensor devices. It is a complex process anyway, and the gateway has its model and anomaly detector. Thanks to a user interface, a human can visualize those anomalies. There is also a data storage that conserves bot raw sensaor data and cleaned data. This data can be processed online or offline: this means that datas can be cleaned when they are collected or after.
The models we use are not always linear of course. The can be polynomial, and we use polynomial regression.

Lezione 11 - 18/05/2022

Context awareness. Last time we saw how to get data from sensors, while today we understand what do we do with datas coming from sensors. Most of the things we'll be talking about is not on the book, but on a survey paper.
Normal applications can't understand the context of a request, and it is difficult for the user to communicate the context to the machine. Also, what is the context? What is important for the device? Because many statements make sense only within their context, like "How do I get there?". In this case, the system should understand where the user is, where he wants to go, and how (on foot, car, public transport, etc..).

There have been several papers trying to define what exactly is a context: a location? A time? season? surrounding people? Temperature? All of them? Some suggested also location, environment, the identity of the person, the mood of the user, its level of attention, the object or people around him. So, there are a lot of choises. And it's up to us understand the context we need for our application. We also depend on the sensors we have at our disposal: in the future, we might have simpler ways to collect datas.
Anind Dey gave a good definition of context (here we paraphrase):
until now, we gave for granted that the context is something that surronund a request given by a user. But there, the context characterize the situation of an entity, that can be whatever. If a user is involved, all the infos about him are relevant for the context. Even informations about the applications are important. And also, the characteristics of the device the user is using is important: is it a smart speaker? A device in a car? A smartwatch? A normal computer? Because, the interface is very different from case to case. Simplyfing: all the data that are useful to adapt a service.
Some characteristics we are interested in are:
-the ones of the user, like its identity, its emotions, its interests (we know about the user), etc.. It's also important to distinguish the sources for all those informations. Like, the identity of the user is the user itself that must give it to me. Physiological and Emotional datas must come from sensors. Don't take these sources from granted! There might be other ways to get those datas. Also the activity the user is performing, in the moment he's asking for a service, is important. Btw, many of those context properties are taken from sensors or devices.
Also connectivity is immportant. We might have nodes with poor connectivity, so we can't always stream videos, for example. This is another information to consider and adapt the stream based on this.

Time, also, plays an iportant role. It can be the time of day, month, the current season, our time zone... but not only this. Also context history palys an important role. It can be useful to know the context of yesterday, this morning... if we track this informations, we can know how the user moves, the shops he visits, how often you go out to visit friends, and so on. Also because, if you have history, you can predict what the user will do in the future (we humans are predictable). in this way, the system can predict our needs or similar things. We are interested in all of this things because we want the system to adapt and change the behaviour depending on the context. The context, or, a context-ware system, is important for a mobile computing system because the connectivity in the network might change, the energy might change, the user situation 8location, time, activity...), the environment (light, crowding, temperature...) and limited interfaces are used to specify parameters.
Example: adaptive video streaming. If the connection goes down, or is lower, the quality of videos decreases, but the streaming, at least, is still guaranteed.

So, this is context. How do we adapt our service? Well, the system might change its logic depending on the context. Depending on the situation, the system will change its data flow. Also, we could hide some functionalities when the context does not require them. Some examples include: increasing caching, moving more computations server-side, changing interface mode, hiding security sensitive functionalities... another idea is: battery is low, so deactivate the GPS or decrease the precision of the service so that the device won't die.
At the same time, we can adapt data, that is, changing the precision with which we get data from

sensors, use higher/lower quality images, changing the sample rate from sensors...

But how do we get those data? Well, there are two categories:
1) Low-level context: the datas coming from sensors or other sources. All the raw datas, basically. These are low-level datas, because we don't do any processing on them. Actually, we can obtain also a simple processing and/or fusion of raw data. For example, a more sohisticated data could be "Hot&Humid", obtained from sensors that tell us that there are 30°C and humidity is 80%.
2) high-level context: the human activities. But there may be any context property that we derive from low-level context.
Why detecting human activities, that could be more or less complex? [Because we humans are stupid basically, at least this is the meaning of Professor's joke for me]. If we know what the user is doing, we could help him better. For example, we could change the interface.
Our health behaviors have an important impact on our general health, and affects it from 40% to 50%. Behavior recognition is important for supporting older people in order to keep them independent. A system can help them stay at home, and understand if they have difficulties in doing something. In the course of years, they should understand if they need help for something. This is a huge application area. Many projects go in this direction.
One class of disiseas that is having a big impact is halzaimer, and context-aware systems can help those older people. First of all, we have to understant the activities the person is performing, and the if there are problems in performing it.
So, it's important to have an high-level context, because it can help us a lot, also for protecting a person. How do we get this context, btw? One way is putting cameras, analyzing videos with machine learning, and then understanding what is happening. What if someone doesn't want camera in his house? Use a microphone. If not even that can be used? There are other sensors.
There are a lot of researches in this. Also, it's important understanding the difference between smart living and outdoor settings. The context might be very different even if we have to perform human-Activity recognition based on a single individual or a group of them. We should also understand if the activities being performed are isolated, overlapping or concurrent. If I'm cooking an, at the same time, setting up the table for dinner, the system should understand that those two things are being done concurrently.

How do we do all of this? Machine learning! ...yeah, but it's not the solutioj to everything. A first approach is data-driven, where we collect a lot of datas. Another approach is knowledge-driven. The deep learning approach: how does it work? We have a bunch of sensors, and we get datas from them. Then we apply a deep-learning model to process those datas (neural networks are used), and this is a very expensive processing. The system is doing feature-extraction, trying to understand what is relevant. First, we understand what is relevant. And then, we map the features to different activities. To do so, we have to provide a lot of examples. If the fridge is opening and the stove is turning on. Given enough examples, the system will map those signals with the cooking activity, and understand when the user will start to cook. The problem? We need to provide a LOT of examples. And those examples must be given by the context itself. And someone must manually map the signals input to the activities the user is performing. This works, of course, but it's complex, because user act in different ways, people

think in different ways... So, producing a new labelled dataset is expensive (labeling is mapping signals to activities). Of course, there are also privacy issues in getting those data, along with differences between different environments. This approach needs a lot of examples (Data-Driven approach: you collect datas and label them). The Knowledge-Based approach, instead, starts from a model you give. In this approach, we observe sensors outputs and derive some sematic states, that is, we give a meaning to what happened.

{flashback about the slide of the last time: among the model-based approaches, we've seen the data-acquisition idea of distributing the model aong the nodes, and there is no communication if the value measured is similar to the one expected from the model. In data-cleaning, instead, the model is possessed by just the gateway and should understand if a value is an outlier or not. This module tries to understand if there is an outlier or not. Anyway, another use of model-based approaches is data compression. It is used to eliminate redundancy. In some cases, we want to store some datas, and this can be done also on the sensor. If the sensor knows that the data acquired follows the distribution of the model, he can just store two values, and all the other intermediate values are calculated from the model. Markov is not important, they just say that we can use different kind of functions for the model We have linear models, where each sensor is considered independent, while in some other cases the sensors can be considered dependent one-another.
Poor Man's Compression: an even simpler model than the linear one, because is a segment with a constant value. Along one axis, we have a sensor value, like temperature. In a given amount of time, all the measures have a maximum distance of 2 epsilon, and the compression says: I just give a constant value and epsilon. So, if I ask the temperature at a specific point in time, it gives me the constant value of that time. What the system needs to store is just a fixed epsilon, and for each segment its time interval and the constant value of that segment. If this sensors produces a lot of samples frequently, this technique helps improving the store. If the sensor got crazy and we have an outsider (that forms a segment by itself), the gateway that gets this information understand the crazy values present and removes them.
We can also use different functions at different times. For some time, we use the constant function. Then, I sense that a linear function would fit better, and so i change the function. In this case, we have a SegmentTable in which the entries point to a ModelTable, describing: from instant a to instant b, this model was used.
[Acquisition and query models (DIFFERENT FROM THE MODELS WE ARE DISCUSSING NOW) are used together, the data cleaning model can or not be used].
[Remember the difference in hwo model is used for data cleaning and data acquisition: in data acquisition, the model is sent to sensors, while in data cleaning just the gateway knows it].}

Back to context awareness. A health monitoring system needs to be unobtrusive, usable and "good-looking" (there could be aesthetic issues). Other issues are security, privacy and operational lifetime. HIGH-LEVEL CONTEXT: what can we use? AI methods, as we said. By AI methods we mean not only deep/machine learning. Also statistical tools are good. Let's see a knowledge based approach, that is based on mathematical logic. In the very large spectrum of logics existent, the Description logic is the theoretical fundation of the "semantic web". This is interesting because it has nice computational

proeprties. In this way, anyway, we have automatic ways of reasoning. We describe with logic what is for us the activity of cooking, so we describe the domain with logic, and then we use automatic tools for reasoning and understand if the semnatic coming from sensors match the logic we specified. But, logic is lousy in representing uncertainty. Describing cooking in logic is difficult because there isn't just one definition, and if a cooking activity we are observing doesn't perfectly math the definition we gave before, the system doesn't understand that we are cooking.
Now, let's present two ways of storing and representing this high-level context. How do we reason about this high-level context? There is a list of requirements that are of interest for context models. But anyway: we can store those data as key-value pairs. In this way (the simples one) we have no structure: it's jsut a bunch of pairs. This is aldo called "Flat Model". By using XML, however, we can have a bit of structure. Flat models are efficient and highly usable, but don't support reasoning, uncertainty (we just have values), historical data, expression of relations and dependencies. In some way, there is heterogeneity and extensibility, but not that much.
Another model is the DB-based model(s). Why don't we use graphical tools to represent the context? Why don't we model also relationships between entities? those models do exactly that. This model also tries to introduce some uncertainty, but not that much. So? Well, trying to use a relational database to represent context is a bit better for expressing relations and dependencies, there is a bit of uncertainty, but we can't do any reasoning, just wueries. There is not even support for historical data (in some recent models maybe yes).
Last model: Ontological Model, based on the knowledge-nbased approach. In Computer Science, ontology means a formal specification of a shared conceptualization. So, were trying to define a complex, based on the concepts of classes, properties, objects... a language we can use is OWL 2, used for defining ontologies. This model is used to represent common knowledge and reason about it. We can express, for example, that a Carnival Party is a Friendly Meeting, and so on.
The support for automatic reasoning is based on:
-substumption: is this concept a specialization of this other concept?
-realization: which concept captures this set of observations? For example, if I'm using the fridge and the oven, I'm cooking.
-Consistency (didn't read lol)
This models, anyway have heterogeneity and extensibility, can express relations and dependencies, and MOST OF ALL, they can do reasoning, that NO ONE could do before. The cons? Efficiency. It's extremely time-consuming doing all this reasoning.
The rest is not coveren.
ONE. LAST. COMMENT.
There is possibility for hybrid models. In general, if you need to undersantd basic states from intertial sensors, like the steps you perform, how do you get datas? We don't know (lol), but we can collect multiple datas from multiple users that are then processed by machine learning models. Then, after collecting datas, we can use those examples (data-driven approach) in order to understand high-level activities. But after understanding that you are walking, for example, I want to know if you used a public transport. We sum all of those datas, and we see if the user is performinf an action that is reasonable for us.
Example: observe the activity of an user from a wrist accelerometer. But, if the user uses a blackboard,

the sensor might think the user is brushing its teeth. A bit of external knowledge can help this. So, colelcting context data from many sources and putting them all together can help in understanding more precisely the situation.

Lezione 12 - 25/05/2022

Last time we saw reasoning methods.
One issue to mention, that is still kind of open as a problem: sensors are very often malfunctioning, giving crazy values. Data inference is also prone to uncertainty, in fact, and ontological models support also uncertainty. We can have different sources of context.
Context data could be old. We might need really fresh datas or we could be good with older datas.

It would be nice to have a middleware that allows to easily acquire context from different sources and reason abbout it. Better if it takes into account privacy issues. Context infact is a private information about what the user is doing, his preferences, etc.. Here we have a trade-off: the user might want to give a lot of private informations to have the best possible experience, while it might also want to be extremely conservative and give almost no infos.

Data privacy in mobile and pervasive system. It's a large topic required by a lot of cose. He has been working for 20 years in privacy and pervasive systems. We'll see what is data privacy, the privacy threats there are in mobile apps and pervasive systems, and techniques to mitigate privacy threats.
Privacy is "the right to be let alone". Most important thing: privacy is a right of an individual, so we are talking about datas regarding an individual, not an organization or confidential informations. So, what is personal data? We will mention a loto the GDPR, that deals with this topic. Personal data is any information which are related to an identified or identifialbe natural person. Data privacy, instead, is the right to be let alone in terms of the data that regard me and I need to control, I don't want to give them to someone. So, it's the ability to control the release, use and distribution of own personal data. In this class we should think about ourselves as people with personal datas, but also engeneers that are using datas of people. Why should we be careful about data privacy? Because of discrimination, deprivation of civil rights, stalking, spam...

What happens? We have data privacy violation in case of internet service requests when people are issuing requests (not a single request, each of the people is probably making multiple requests, so it's a stream of requests) through a mobile operator that go to an internet service. Where is the privacy threat? There might be adversaries that might look at these requests and do sensitive association. That is, an unauthorized individual gets those sensitive associations, that combines private informations with the identity of a person. If we have private informations about, for example, the religion of a person, but we don't know the person that has this religion, there is no privacy violation. Same thing with the position: we don't want to know where a person is. If we know where a person is exactly, we could offuscate this information by saying: ok, he's not right there, he is somewhere in Milan. Attacckers usually use also external knowledge.
Re-identifying means understanding who a person is from some informations, for example those given in

a request. But when we re-identify someone, that person usually wouldn't like to be re-identified. The location can be a sensitive information: we might do re-identification using frequent patterns of location. So, location informations can be very powerful for finding out who a person is.
Geo-Social network: a lot of social networks are geo-tagged, that means, they save location informations about their users. Examples include Facebook and Twitter. In solcial networks, the adversaries might be other users of the social network. This is a surface attack (this is the name). But there is another tricky thing is geo-social network. If a friend of mine is publishing a photo of me and him together, with a geo-tag, perople will know I'm there even if I'm not using Facebook anymore, and this is a problem.
Location privacy has two other sligthly different notions: co-location privacy means having the right to be somewhere with somebody else. I just care that people know that I'm somewhere with somebody. In pratica la gente scopre che sono con qualcuno, e non vorrei che si sapesse. Absence privacy: people knows that I'm not in some place (like, at home, so people can steal). This is different from asking where a person IS, but is dangerous as well. So, location privacy, co-location privacy and absence privacy.

In mobile computing, space and time datas are the most important. By using sensors, we get a lot of informations. Regarding health, we have to be even more careful: if someone knows I'm ill in the USA, they might rise the price for an operation for me. There are in fact some "Well being" apps that can know how we are. Even the accelerometer might understand how I'm walking, and that can be a distinctive trait.
There are a lot of gadgets nowdays that can keep track of what we do. They usually connect to an app, that might ask us to transmit those datas to a service provider, even if usually there would be no need to do that, because the app itself has the computing power it needs to provide the service we want. They want to collect those datas just for reasons outside of our direct benefit. Bettini published a paper, a survey about privacy issues in pervasive computing.
LBS (Location Based Service), Mobile Advertisements, GeoSN and Participatory sensing are all category of infrastructures that collect our private data. Other categories are eHealth, Quantifies Self, Vehicular apps and smartcity services and SmartHome and Smart utilities. All of them have adversaries and sensitive datas.
Both designers and users should know what datas will be at stake. If we have more personal datas, we have more exposure and need for scalability, and new type fo data impy new ways tor e-identify someone. Continuous streams should make us use some forms of protection from sequential/continuous release. If we have a continuous stream of informations, the adversary might have collected previous informations that he could use for identifying us in the future, based on new datas he collect that are similar to the ones we sent in the past. Other things: interfaces to express privacy preferences might be absent because of the system we are using (that does not provide them).

This was about threats. What can we do about this? There are people saying that you should do anything to protect, because we are getting used to the fact that this informations go out.
Be very careful about privacy protection: to do so, we have regulation and techniques. Expecially if we are using context data, we have to be careful or we might get in trouble. So, a few things about regulations.
GDPR, the EU general data protection regulation, has extablished that there are fires up to 22M of euros

for data privacy crimes. In particular, for context data, we need to do the Data Protection Impact Assessment, that is a test for understanding the effects that there would be if the privacy of our users were to be violated. In those cases, we have to analyze the specific data and service we offer, using also a lawyer. The analysis needed also requires more protection if the datas are extremely sensitive. More technical aspects: security. All things that apply to security for communication and network security (also encrypting datas) also apply here. We must use state of the art methods to protect the data. But privacy requires more effort than security. If we offer to the user some options for privacy, the user must be able to say "yes, I want to give more data to have a better service", but at the start of the app the app should be the most conservative possible. Privacy by design: it's not a specific technique that we have to apply. If we design a system, at the same time, we should also think about the datas we are collecting: are they private? Who collects them? We have to think about this immediately, or we might pay a price after the implementation has been done. But how can people chack I'm dealing with those issues? If someone comes for an inspection of my company, I must show them that all the processes that involve personal data are such that I did an analysis for each of them, and the kind of countermeasures I took to respect the regulation. Data Minimization is collecting datas just at the necesary precision, the one strictly needed to do the work. Tools for minimization/anonimization can be used. This has a lot to do with techniques we'll see in a minute. Minimization says that if we provide a service and we want the context to adapt to id, we should get the context data only at the minimum resolution required for the service. If I'm a forecast service, I just need your area, not your exact position. This is important, because sometimes organizations try to collect more datas than needed. But this is not allowed: we should specify to the users what datas we want for them and why we want to use them.

Security is part of the problem. We need different techniques to ensure security. Security principles include:
-Confidentiality. Only authorized parties can access data, so we can use authentication, encryption of data in transfer...
-Integrity: data should not be alteren without authorization, so we use cryptographic hashing, checksum...
-???
-Transparency: we show what we do with the data, and we can prove that we have a specifici pipeline of data processing.
-Unlinkability: What we'll be talking about. It has to do with sensitive association: it's the idea that we want to unlink certain data parts in order not to reveal sensitive data associations.
-Intervenability: emergency button to push when we need informations about someone. We violate privacy to save someone, for example.

Data minimization: not every app requires precise location (and time). Some Services require high precision, others do not. Depending on the requirements of the applciation, we might choose the granularity of the data. In order not to make explicit the time where you are in a specific place, boh. The GDPR explicitly says that pseudonymization is a good technique to protect personal data. If correctly implemented, it separates sensitive data from the person that produced that data. The mapping between them, nzomma, is accessible only to selected authorities. Of course we need to store the

association between those entities, but this has a very restricted access. Only a few authorized entities can really access this mapping, and it will be protected with extra protection. Do not confuse anonimization with pseudonymization. Anonimization tries to separate sensitive informations from an individual. A bank may store data about a customer.
Anonymize: no way for anybody to understand the individual from the data.
Pseudonymize: the mapping exists but is available only to some people.
k-anonimity: way to avoid re-identification. The k-anonimity says: suppose I am the adversary and I see the data. But these data are offuscated. If I take a specific record, I cannot really understand the mapping between the person and the datas regarding him. If I take a specific record, I cannot understand many informations. Zip and Age, in the slide, are Quasi-Identifiers. In any anonimization technique we have to undestand what can be used to identify people. "Out of this data, these can identify a person (or not)". Location can act as quasi-identifier. Location k-anonimity solution: use pseudonyms and spatio-temporal cloacking. If I want to be 5-anonymous, 5-location anonymous, when i do a request, I should give a region that comprehends at least 5 other people that request the service I'm requesting [They don't have to make the same request in the same moment, it just suffices that they are there].
In an anonimity group we should include different values for different datas, otherwise it's easy to understand the mapping between the user and the datas.
Historical k-anonimity: Alice makes a request, and after some time she makes the same request, with the same pseudonym. If the adversary knows that in a certain moment the searched user issued a particular request, while in a second moment he did the same, he might as well find out who is the user (Alice). Bottom Line: how do I protect from this attack? "Never walk alone": in the rectangle I should include people that go where I'm going, so that, in all rectangles, we are almost always the same people. Another thing we can do is change the pseudonym. this is about re-identification.

What about data location? It is a sensitive information. In this case, the technique is the same. If you want to confuse the adversary and not let him know that we are in a cancer facitily, we enlarge the area (cloaking). But now the question is: how much should I enlarge the area? Before the criteria was based on how many people were in the area (k-anonimity). In this case we are looking for other facilities. We enlarge the area in order to inlcude other places that are not sensitive. If i incluse a Pub in the area, the adversary won't know if I'm in the cancer facility or in the pub. i enlarge the area to make less sensitive the information. Semantic place is offuscated basically: the adversary doesn't know where we are. So, the criteria for enlargement is this.
Temporal cloacking: instead of giving the exact timestamp, we give an interval of time. So, uncertainty even in time. Example: Alice is at the Uni (not specific location) in a time between 5 and 6 (not an exact time). Then she goes to a pub in Downtown: again, a large area, not an exact one, and, for example, a precise time, like 6:10. But it is not possible that Alice was at 6:00 at Uni... so beware, an adversary could find out that these things don't make sense and discover something.

Both apple and Google have used some techniques to get statistical datas from individuals, but only because they want aggregated data: what people prefer and so on. But we don't talk here about statistics of people data, but about precise informations about individuals (microdata of a specific individual). So, apple introduced the enlargement of region, and mobile developers have to take into

account this change in their apps. Another thing we can do: use a fake location. How to we pick it? There are theorems for it.
In 2020 iOS14 was introduced, and the idea was to give the users an option for having more privacy. "Do you want to tell your position? You can also offuscate your region, if you want a via di mezzo". An app in use can use the position, but it will never be exact. This requires us, anyway, to change the application, of course, because we don't know where the user is exactly, so it's a challenge for the developer. When apple came out with this solution, Bettini was eager to see if they did it the right way: did they respect a particular property? YES! (this proerty is of Bettini). The property is resistent to inversion. many researchers thought to build the uncertainty region starting from the position of the user. In fact, an adversary could do the intersection of the regions to understand possible movements. But iOS is resistant to inversion, and they use fixed regions, and the areas are determined by a lot of factors (context-aware size, the size changes in respect to what happens around the user). Frequent updates and the change of region would reveal we are close to a certain point, so the updates are unfrequent.