

(先看一下後面的幾種 software architecture 例子(pipes-and-filter, event-driven 等), 就知道什麼是 software architecture 了)

1. Hello, and welcome to the third part of our software engineering course. In this mini-course, we will discuss software design. We will also introduce the Unified Software Process. And we will work on a more complex project, in which we will develop a distributed software system that involves multiple different platforms. In our first lesson of this mini-course, in particular, we will talk about software architecture. A software engineering discipline whose goal is to lay the foundation on which to build successful and long lasting software systems. So let's begin.

2. Because the topic of today's lecture is software architecture, it seemed appropriate to start the lesson by asking a world expert on this topic, what is software architecture, and why it is important. To do that, let's fly to California, and more precisely, Los Angeles, and visit Professor Nenad Medvidovic. Hi, I'm here visiting Professor Nenad Medvidovic from the University of Southern California. And Neno is one of the world experts in software architecture, actually one of the authors of, of a recent book which is, sort of the book in software architecture. What I would like to discuss with Neno is the concept of software architecture and its importance. Because [people are very familiar with the idea and the cost of the design. And software architecture is something is very related to that, but is less known](#). So I would like for Nenad to elaborate on that, and tell us why is it important to focus also on this specific you know, architectural aspects of the software. >> When you build any software system, even a simple, relatively simple one. You're going to go through, a process of making many, many design decisions. Hundreds or thousands sometimes even tens of thousands of design decisions, so any program that you write at some point you get to deciding what the interface of a particular method is going to be. Are you're going to put in a parameter that is an integer or a float. When you're writing your routine about some sort you have to decide whether you're going to use a static data structure or a dynamic data structure. All these things are design decisions. Many of them however, will typically, in the average case, not really impact the success of your system and the long term well-being of your system. But typically the things that software engineers start struggling with are other design decisions. Design decisions that are the equivalent of load bearing walls in a building >> Mm hm >> These are the things that, if you don't get them right, or if you compromise them, will in fact potentially impact how the system operates. They might result in failures of different kinds. They may result in a system that is not easily maintainable and so forth. In a sense, to make a long story short, architectural design decisions are really the principle design decisions in your system. These are the things that are very important. All of the other design decisions you could sort of tag with being important, but they're sort of below this very important or highly important threshold. >> So if you need to change a low level design decision, sometimes it's kind of easy to do. It might change a little structure. Is it the case that you know, being the architecture is sort of the pillar of the software, is that going to be much more difficult to change an architectural decision? And architecture is deemed to be you know, say if you start with the wrong architecture the software is going to, you know, necessarily be unsuccessful. Or you can also do something that is better. >> A system could be successful and very poorly architected. Just like a building or an airplane or a car, any other engineering artifact could be successful but poorly architected. So success we can separated from this, but the, the point that you make in asking this question is an important one. The known architectural design decisions, should be on the average, there are exceptions and we need to acknowledge that there is no one size fits all type of solution for anything in software engineering really. But on the average, the non-architectural design decisions, should be much easier to make. So the scale of the consequences of making such a change. Really can vary from very minor, highly localized to very important and sometimes, even system wide. >> To conclude, I just like to ask you about some concept that is we here about a lot. Which is architectural erosion. Since, we're talking about in with fine architecture and self-revolution. So, what is, exactly, an

architectural erosion and why does that happen? So, to go back to our non software metaphors. Imagine you buy a car. And your car has four wheels, it has a steering wheel, it has a nice chassis, it looks pretty nice. At one point, you end up replacing its 150 horsepower engine with a 250 horsepower engine because that's what you want. And you start putting a spoiler on the back of the car and then you replace the headlights. And then you replace the side view mirrors with smaller ones because you want your car to be more aerodynamic. And then you start tinkering with other things, like you cut the, maybe the roof of the car because you want to turn it into a convertible, et cetera. And in the end, what you have is a car that is still your car. Looks very different, Its structural and behavioral properties are very different And, what you might find is that the car doesn't handle nearly as well. For example, in a very sharp turn it might not be able to negotiate a steep hill as well. Because you pretty much changed it all along the way. Architectural erosion in the case of a software system is the exact same thing with one huge caveat. Very few, if any of us, will ever put a new engine into our car or tinker with the structural soundness of the car by cutting off the roof etc. In a software system we do it all the time. We'll add a feature. We'll change one bit of the user interface here. We'll port it to a new platform, some kind of a, a mobile platform, for example. And pretty soon, what you end up with is really a software system that, that is maybe a distant relative of your original system. It is a mutant in many ways, because often times these little tinkering happen on a one off basis. There is no over-arching vision of how you should do this. So, you are basically going through a subsequent set of steps where you are making locally optimal decisions for any one of these changes and what you might end up finding is that the globally optimal behavior of the system is badly compromised. The structural sound is in a sense of the system badly compromised. The non-functional properties of the system could be seriously affected. This is how security flaws creep into systems. This is how reliability flaws. This is how we use the usability of a system often times suffers. And most importantly for software engineers, the people who actually build the software, the maintainability of the system becomes a huge problem. Because now you're looking at this thing, it's got all these various appendages, its original design has pretty badly eroded and yet somehow you have to figure out how to keep fixing it. Making sure that it operates in a continuous fashion because many of these systems for 20, 30, 40 years. >> Thank you so much for your insight; it is a perfect introduction for our lesson. So we'll get to the lesson now. And. >> Thank you very much. >> Thank you.

3. After this interesting conversation with Nino, let me start the lesson by defining what a software architecture is. And to do that, I'm going to use two seminal definitions. The first one is from Dewayne Perry and Alex Wolf. And they define a software architecture as elements, form and rationale. In this definition, the elements are the what, which means the processes, data, and connectors that compose a software architecture. The form is the how, the set of properties off in relationships among these elements. And, finally, the rationale is the why, the justification for the elements and their relationships. The second definition I want to use is from Mary Shaw and David Garland. And they defined a software architecture as a level of design that involves four main things, a description of elements from which these systems are built, the interactions among those elements, the patterns that guide their composition, and finally, the constraints on these patterns. As you can see, these definitions are fairly similar and there are many more alternative definitions of software architecture. In fact, if we try to search the term software architecture, we get over two million entries. And if we look at the images in the results of the search this is what we get. And I like this sort of graphical depiction because it gives you a clear idea the software architecture are prevalent concept, given the number of results. But they also show you clearly, that software architecture are complex entities, if you look at some of these pictures. And ultimately, they show that software architecture are presented in all kinds of ways including 3D, if you look at this picture. We cannot clearly cover all of these definitions in one lesson. So what I will do instead, is to introduce a very general definition that encompasses(包含, 圍繞) most of the existing ones.

WHAT IS SOFTWARE ARCHITECTURE ?



Perry and Wolf

SWA = { Elements, Form, Rationale }
 what how why



Shaw and Garland

SWA [is a level of design that] involves

- description of elements from which systems are built
- interactions among those elements
- patterns that guide their composition
- constraints on these patterns

4. I'm going to define a software systems architecture as the set of principal design decisions about the system. Where principle here, implies a degree of importance, that grants a design decision architectural status. And the point here, as we discussed with Nino early on, is that **when building a system, we make tons of design decisions, and most of them do not affect the architecture of the system. For example, the effect of choosing a for loop, instead of a while loop, in the code, or the fact of deciding that we are going to use data structure a instead of data structure b. Some decisions however, do affect the architecture of the system. And in some cases the distinction between these two kinds of design decisions is clear. In some other cases it is much fuzzier and it depends on the context. The bottom line here, is that if you believe that something is an important design decision, that becomes an architectural decision.** That is a decision that impacts a system's architecture. In this spirit, we can see a software architecture as the blueprint for a software system, that we can use to construct and evolve the system. And the key point about software architecture is that this blueprint encompasses every facet of the system under development. It encompasses its structure, of course, but not only. It also involves the behavior of the system, the interactions within the system, and the non-functional properties of the system. And we will see how this happens in the rest of the lesson. Another important point about software architecture is that there is a temporal aspect to it. And the point here is that you don't build the software architecture in a single shot, but you do it iteratively, over time. So, basically, you go from having no architecture to your final architecture. So, at any point in time, there is a software architecture, but it will change over time. And this happens because design decisions are made, unmade and changed over a system's lifetime.

A GENERAL DEFINITION OF SWA

Set of principal design decisions about the system



Blueprint of a software system

- Structure
- Behavior
- Interaction
- Non-functional properties

TEMPORAL ASPECT



A SWA is not defined at once, but iteratively, over time

At any point in time there is a SWA, but it will change over time

Design decisions are made, unmade, and changed over a system's lifetime

5. We can look at the software architecture from two main standpoints. There are prescriptive and descriptive software architectures. So what does that mean? A prescriptive architecture captured the design decisions that are made prior to the system's construction. This is what we normally call the as-conceived software architecture. Conversely, a descriptive architecture describes how the system has actually been built. So it's based on observing the system as it is and extracting the architecture from the observation. This is what we call the as-implemented software architecture. And one key point here is that [often, these two architectures, the prescriptive and the descriptive architectures end up being different. So let's see why that happens.](#)

PRESCRIPTIVE VS DESCRIPTIVE ARCHITECTURE



A prescriptive architecture captures the design decisions made prior to the system's construction

⇒ as-conceived SWA



A descriptive architecture describes how the system has actually been built

⇒ as-implemented SWA

6. To do that let's look at how architectural evolution occurs in practice. Ideally when a system evolves, its prescriptive architecture should be modified first. Just like when you modify a building. You change the blueprint and then you change the actual building. You don't go the other way around. In software, unfortunately this rarely ever happens in practice. In practice the system, and therefore its descriptive architecture are often directly modified. Like in this case that I'm showing here. So what happens is that the architecture as conceived does not change. Whereas the architecture as implemented, does change. And therefore these two things start diverging. And this really happens for a number of reasons. So I'm just going to list a few of those reasons here. In some cases it just happens for plain sloppiness. I need to make this modification and I don't really want to go back and look at the prescriptive architecture modified. I'm just going to make the change, and maybe I'll fix the description later. And then you never really get to it. In other cases you do this because of the perception of short deadlines. If you have to do something by this afternoon, you're not going through a four month software architecture review, you normally just get to it, and do it. In some cases a prescriptive architecture is not even present, so there's a lack of documentation. So in these cases, clearly, you cannot go and modify something that does not even exist, and so you jump directly to the code and start modifying that. And as I said there's a many, many more other reasons why that happen. But important point here is that it does happen and it does happen often and the result is that prescriptive and descriptive architectures diverge.

ARCHITECTURAL EVOLUTION

When a system evolves, ideally its prescriptive architecture should be modified first



In practice, this rarely happens

- Developers' sloppiness
- Short deadlines
- Lack of documented prescriptive architecture
- ...

7. (degradation: 降格,退化) And there are two important and related concepts that have to do with the way a soft architecture evolves. The first one is Architectural Drift, which is the introduction of architectural design decisions that are orthogonal(正交的) to a system's prescriptive architecture. That is, they're not included in, encompassed by, or implied by the prescriptive architecture. And the result of Architectural Drift is that you start from a clean architecture, like the one that I'm showing here, and then you start adding pieces without following a clear plan. Like, for example, here, we add an additional room here, but we don't really do it in the right way so we need to add something else to keep it stable. And then maybe we want some more room so we add a tent. And then another side of the house, it doesn't really follow the same architecture but it doesn't matter, we just put it there because we want to expand. And maybe then we want to put something classic there, even though it doesn't really fit the overall design and the overall architecture. So I think you get my point, the fact that the architecture then becomes unnecessarily complex, hard to understand and ultimately awkward, just like the one that I'm showing here, that goes from the original build into this final atrocity. The second concept is Architectural Erosion, which is the introduction of architectural design decisions that violate a system prescriptive architecture. So in this case, that we were introducing decisions that were orthogonal, here, were introducing this decisions that don't comply with the prescriptive architecture. And the result of Architectural Erosion is typically a poor architecture an architecture that is going to have problems in the future. So both Architectural Drift and Architectural Erosion take you away in different ways from what you think your software architecture is or should be.

ARCHITECTURAL DEGRADATION



Architectural drift - introduction of architectural design decisions orthogonal to a system's prescriptive architecture



Architectural erosion - introduction of architectural design decisions that violate a system's prescriptive architecture

8. And sometimes, architectural drift and erosion gets you so far away from the point where your software architecture should be, that your architecture is completely degraded. And at this point, you have two main options. The first option is to keep frantically tweaking the code. And this normally leads to disaster. Why? Because you only make things worse. You don't know exactly what you are changing and therefore, you're basically stabbing in the dark, trying to fix your system. The other possibility is that you can try to determine the software system architecture from its implementation level artifacts, so you try to derive what the architecture is and try to fix it, once you have derived the architecture. And this is what is normally called, architectural recovery, determining a software architecture from an implementation and fixing it. And as you can imagine, this is normally a more recommended way to go than the first solution.

ARCHITECTURAL RECOVERY

Drift and erosion \Rightarrow degraded architecture



Keep tweaking the code
(typically disastrous)



Architectural recovery: determine SWA from implementation and fix it

9. Now that we discussed some important concepts about software architectures, I would like for you to tell me which of the following sentences is true. Prescriptive architecture and descriptive architecture are typically the same. Architectural drift results in unnecessarily complex architectures. Architectural erosion is less problematic than architectural drift. And the best way to improve a degraded architecture, is to keep fixing the code until the system starts looking and behaving as expected. Which of these sentences is true?



Which of the following sentences is true?

☐ Prescriptive architecture and descriptive architecture are typically the same

☒ Architectural drift results in unnecessarily complex architectures

☐ Architectural erosion is less problematic than architectural drift

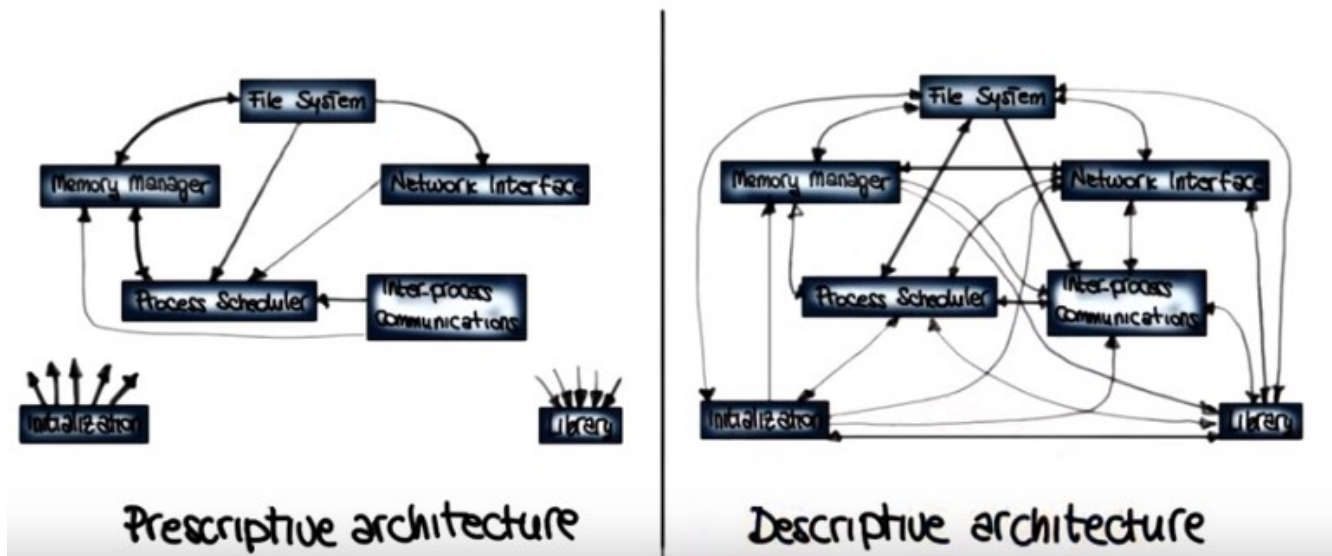
☐ The best way to improve a degraded architecture is to keep fixing the code until the system starts looking and behaving as expected

10. The first sentence is definitely false. Prescriptive architecture and descriptive architecture tend to diverge as systems evolve, and sometimes, even when the system is first developed, as we will see in some of the upcoming examples. Conversely, the second sentence is true. By adding unnecessary elements to the architecture, architectural drift can transform an otherwise clean architecture into a complex sub-optimal, and often ugly, architecture. The third sentence is false. Architectural erosion and architectural drift are, indeed, different phenomena. But they both result in a less than ideal, and in some cases, highly degraded architecture. And the fourth sentence is also false, as we discussed a minute ago. Just tweaking at the code is very unlikely to improve the code. Quite the opposite, actually. The best way to repair a degraded architectural design is to first, understand the current architecture, and then, try to fix it in a more principled way.

11. Now to drive home some of the points that I just made, [I would like to show you a few real world examples of architectures that kind of went astray](#)(迷途地,入歧途地). The first example I want to use is an example from the Linux kernel. Actually, from an earlier version of the Linux kernel. A research group studied the documentation of Linux, and also interviewed several Linux developers. And by doing that, they were able to come up with a software architecture of Linux at different levels of obstruction(障礙). So the one that I'm showing you here on the left, is the software architecture at the level of Linux's main subsystems. So this is the prescriptive architecture of Linux at the level of Linux's main subsystems. So the researchers, after identifying this architecture, they showed it to the developers, and the developers agreed that, that was indeed the architecture of the system. The researchers then studied the source code of Linux and reverse engineered its actual architecture. So the architecture as implemented, it's descriptive architecture. And this one here, on the right, is the result. And as you can see, they found a number of differences or violations between the prescriptive

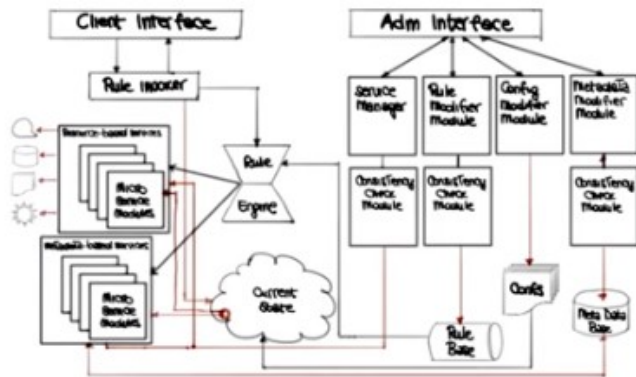
architecture and the descriptive architecture. In particular, if we look at this architecture, we can see that [pretty much everything talks to everything else, which is, in general, not a good thing](#). And in addition to that, [there are also several things that don't really make much sense](#). For example [the library calls the file system and also the network interface](#) which doesn't make much sense. Another thing that is kind of weird is the fact that [file system cause the kernel initialization code](#). Which is also a little bit weird. So basically, the bottom line here is that not even the developers realized how the actual architecture of the system was, and how it was different from the architecture they have conceived. And in fact another interesting thing here is the reaction of the developers when they were shown the actual architecture. So basically they just defined the differences by saying things such as, well you know it had to be done fast and, and therefore I changed it and then I didn't have time to go back and update the documentation and things of this sort. And by the way these are exactly some of the reasons that we mentioned early on in the lesson for the discrepancy between prescriptive and descriptive software architecture. So one last thing that I want to mention here as an aside and we can get back to that later is the fact that you can probably clearly show how representing software architectures graphically can be extremely useful, because it allows for easily seeing the structure of the system. Look at different views identify problematic points and so on. And we will see how that can be useful in many cases also later on.

AN EXAMPLE FROM THE LINUX KERNEL



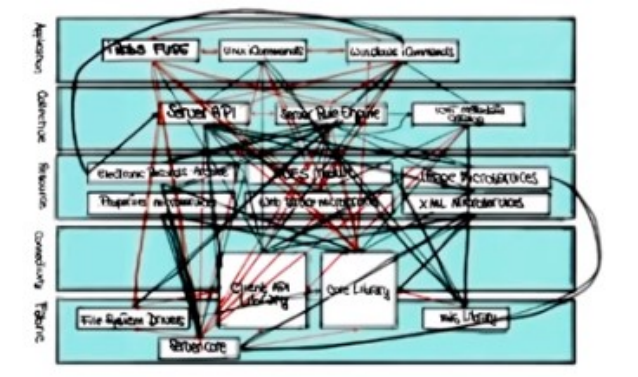
12. As another example, I want to show you the architecture of the iRods system. This is a data grid system that was built by a biologist. And it's a system for storing and accessing big data. So what I'm going to do, I'm going to do the same thing that I did for the Linux system. I'm going to show you here, on the left hand side, this clean prescriptive architecture for the iRODS system. And I'm going to show you here on the right the actual architecture of the system. The descriptive architecture of iRODS. So here, even if we don't go in and look at the details, you can see very easily that the system is badly drifted and eroded. With us back to the way it was supposed to be. Continue with the examples. What I want to show you now is the view of the complete architecture of HADOOP. As many of you probably already know, [HADOOP is an open source software framework for storage and large scale processing of data sets](#). It's very proudly used. And here is a picture of the architecture, and I hope you can see it because the architecture is so complex and so broad and so intertwined, and in order to be able to

ANOTHER EXAMPLE: iRODS



The diagram illustrates the Prescriptive architecture of iRODS. It is divided into two main sections: Client Interface and Admin Interface. The Client Interface includes a Rule Manager, which interacts with a Rule Engine. The Admin Interface includes a Service Manager, Rule Manager Module, Config Manager Module, Metadata Manager Module, Consistency Check Module, and a Rule Base. The Rule Engine interacts with the Rule Base and a Current Queue. The Service Manager interacts with the Rule Base and a Meta Data Base. The Consistency Check Module interacts with the Rule Base and a Meta Data Base. The Metadata Manager Module interacts with the Rule Base and a Meta Data Base. The Rule Base interacts with the Meta Data Base. The Meta Data Base interacts with the Rule Base. The Rule Base interacts with the Meta Data Base. The Rule Base interacts with the Meta Data Base.


Prescriptive architecture



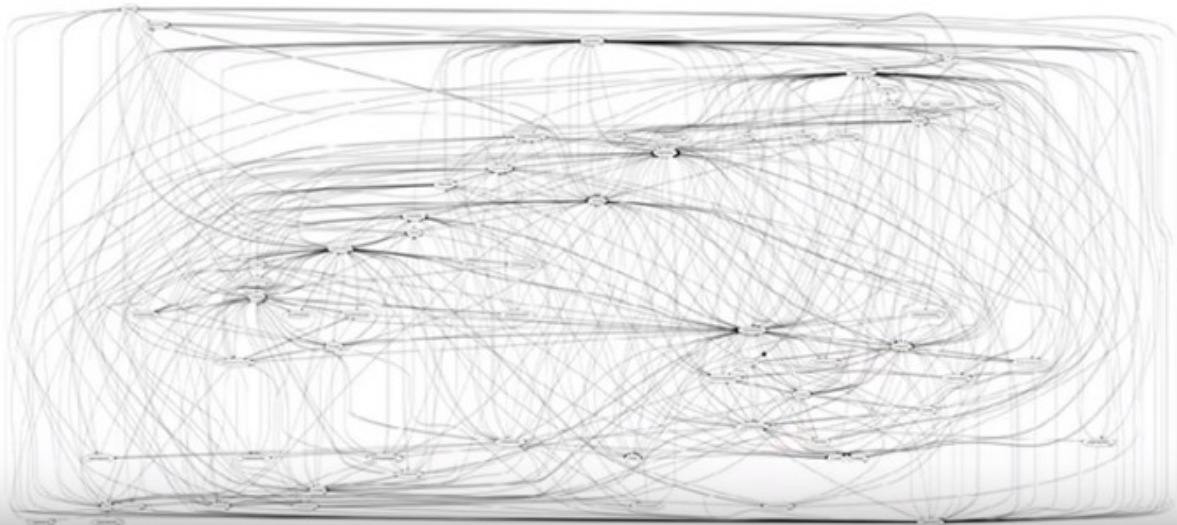
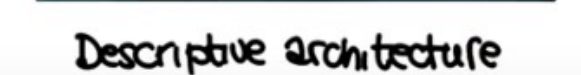
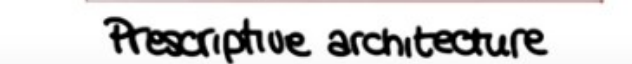
The diagram illustrates the Descriptive architecture of iRODS. It shows a complex network of components and their interactions. The components are organized into layers: Application, Collective, Resource, and Computing. The Application layer includes iRODS RUSER, Unix Commands, and Windows Commands. The Collective layer includes Server API, Server Rule Engine, and iRODS Metadata. The Resource layer includes iRODS Microservices, iRODS Microservices, and iRODS Microservices. The Computing layer includes File System Drivers, iRODS Core, and iRODS Core. The interactions are represented by a dense web of lines connecting the components across the layers.

Descriptive architecture

MORE EXAMPLES : HADOOP

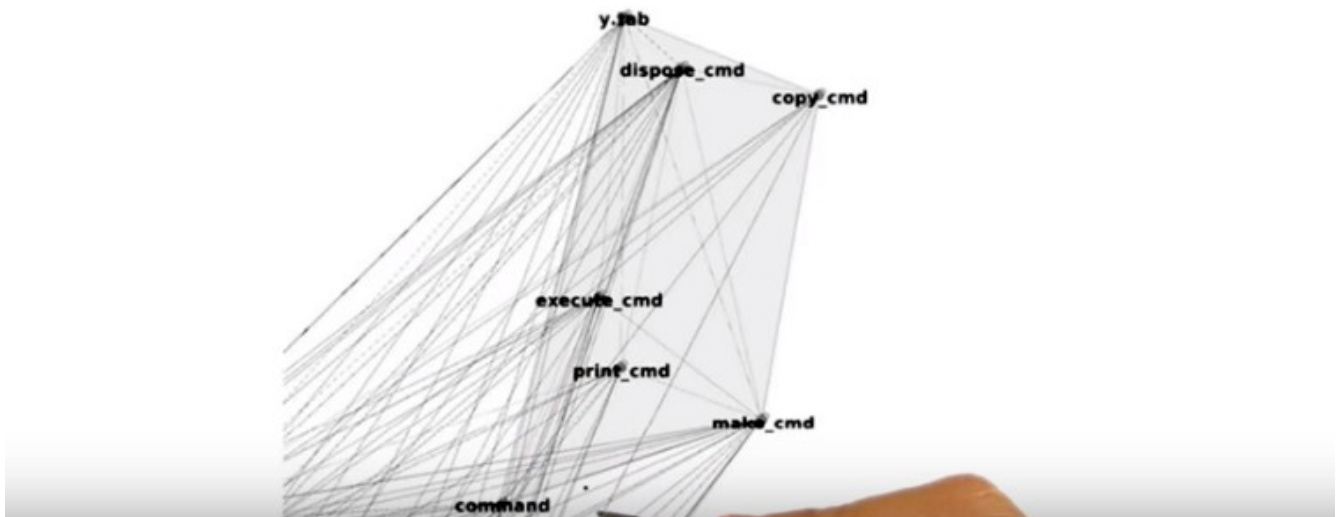


The diagram illustrates the Hadoop architecture, showing a complex network of nodes and connections. The nodes are organized into a grid-like structure, with horizontal and vertical connections. The connections are represented by a dense web of lines, indicating a highly distributed and interconnected system. The nodes are labeled with various components, including NameNode, DataNode, and TaskNode. The connections represent the flow of data and tasks between the nodes.



component of Bash. So, is the architecture, as implemented, of the command component of Bash. And the component is the one here sort of highlighted in gray. And what you can see here, these names are the sub components of the command component. And if we look at this architecture, **two design problems of the component can kind of jump at us. The first one is the lack of cohesion within the component.** So, if you look here, you can see that only a few connections exist between the sub-components. And having a low cohesion is normally not a good thing for a design. **The second thing that we can note is the high coupling.** The component has tons of connections with other components. They're, **these edges that are leaving the components and going towards other parts of the system.** So basically, **this component has low cohesion and high coupling, which is exactly the opposite of how a good design should be.** Given the structure, it is clear that anytime you change this component you might need to change a bunch of other components in the system. And of course, when changing other components in the system, you might also need to change the command component as well. And along similar lines, to understand this component you probably need to look at many other parts of the system, which is also less than ideal. And one important point here is that with all these examples, I'm not really trying to criticize any specific system, what I'm trying to show instead, is how complex software architectures can be, and how much they can degrade over time. And this is true for most systems, not just the ones that I showed you.

A FINAL EXAMPLE: BASH



14. At this point, we have seen some examples of things that might go wrong with the software architecture. So I'd like to ask you also to recap some of the concepts that we've touched upon. What are ideal characteristics of an architectural design? And I'm showing you three possibilities here: scalability, low cohesion, and low coupling. And some of these concepts we did not explicitly define, but we talked about it when discussing the examples that we just saw.



What are ideal characteristics of an architectural design?

- ☒ Scalability
- ☐ Low cohesion
- ☒ Low coupling

15. So, let's look at these three characteristics one by one. Scalability(規模可伸縮性) for software architecture is its ability to handle the growth of the software system. For example, for a web based system, scalability could be the ability to handle a larger workload by adding new servers to the system. Scalability is therefore an important characteristic of a software architecture, especially for the kinds of systems that can grow over time. So, we're going to mark it as an ideal characteristic. Cohesion is a measure of how strongly related are the elements of a module. Clearly, we should shoot for high and not low cohesion when developing a system. We want to develop modules whose elements cooperate to provide the specific piece of functionality rather than modules consisting of a bunch of elements that provide different unrelated pieces of functionality. Therefore, low cohesion is definitely not something that we want. We want high cohesion instead. As for coupling, coupling is a concept related to cohesion and is also a measure. In this case though, it is a measure of how strongly related are the different modules in a system. Low coupling, which is often correlated with high cohesion, is an important and ideal characteristic of a software architecture as it indicates that the different modules in the system are independent from one another. Each module provides a specific piece of functionality and it can provide it without relying too much on other modules. Basically, systems characterized by low coupling and high cohesion, are systems that are easier to understand, and to maintain.

16. Now that we have discussed a few foundational aspects of software architectures, and we have looked at some real world examples that help us to illustrate some of these points, to discuss some of these aspects. I want to introduce and define the different elements that compose a software architecture and also talk about architectural styles. So let's start by discussing a software architecture's elements. A software systems architecture typically is not, and should not be, a uniform monolith(巨型獨石). On the contrary, an architecture should be a composition and interplay of different elements. In particular, as we quickly mentioned at the beginning of the lesson, there are three main types of elements in an architecture. Processing elements, data elements, and interaction elements. Processing elements are those elements that implement the business logic and perform transformations on data. Data elements, also called information or state, are those elements that contain the information that is used and transformed by the processing elements. And finally, the interaction elements are the glue that holds the different pieces of the architecture together. Now, the processing elements and the data are contained into the system components, whereas the interaction elements are maintained and controlled by the system connectors. And components and connectors get all cooked together into a systems

configuration, which models components, connectors and their relationships. So now, let's look at components, connectors and configurations in a little more detail.

SOFTWARE ARCHITECTURE'S ELEMENTS

A software architecture typically is not a monolith

Composition and interplay of different elements



Processing elements



Data elements



Interaction elements



+



⇒

components

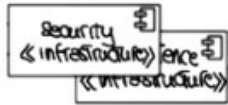


⇒ connectors

} configuration

17. And let's start with software components. A software component is an architectural entity that encapsulates a subset of the system's functionality and or the system's data. So basically components typically provide application specific services. In addition to that, a software component also restricts access to that subset via an explicitly defined interface. And, in addition, which I'm not sure in here, a component can also have explicitly defined dependencies on its required execution environment. In complex systems, interactions might become more important and challenging than functionality. And this is why connectors are very important architectural elements. A software connector is an architectural building block tasked with effecting and regulating interactions among components. So basically, connectors typically provide application independent interaction facilities. And it's worth noting here that in many software systems, connectors might simply be procedure calls or shared data accesses. So all constants that we're familiar with. But consider that much more sophisticated and complex connectors are also possible. And components and connectors are composed in a specific way in a given system architecture to accomplish that system's objective And this is expressed through an architectural configuration. More precisely, an architectural configuration, or topology, is a set of specific associations between the components and connectors of a software system's architecture. So now, let's look at an example that brings all of this together.

COMPONENTS, CONNECTORS, AND CONFIGURATIONS

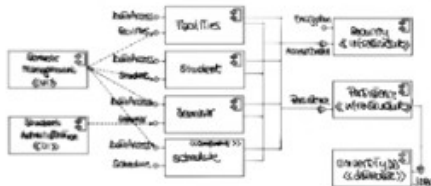


Software component : architectural entity that

- encapsulates a subset of the system's functionality and/or data
- restricts access to that subset via an explicitly defined interface



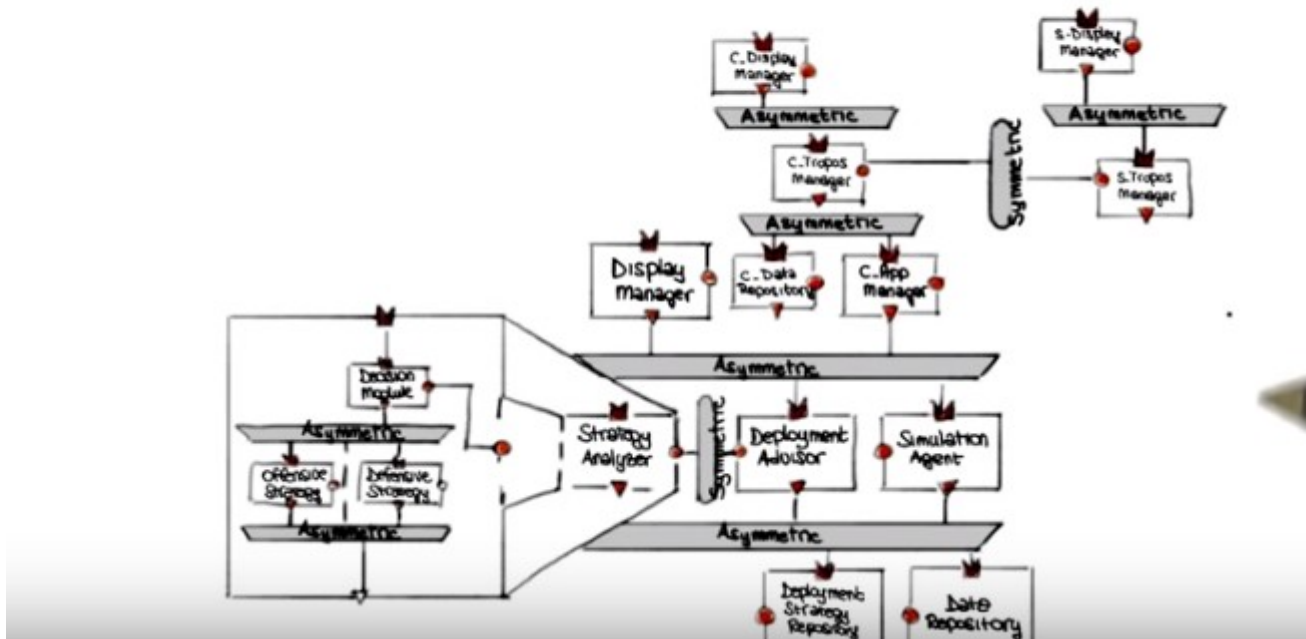
Software connector : architectural entity effecting and regulating interaction



Architectural configuration : association between components and connectors of a software architecture

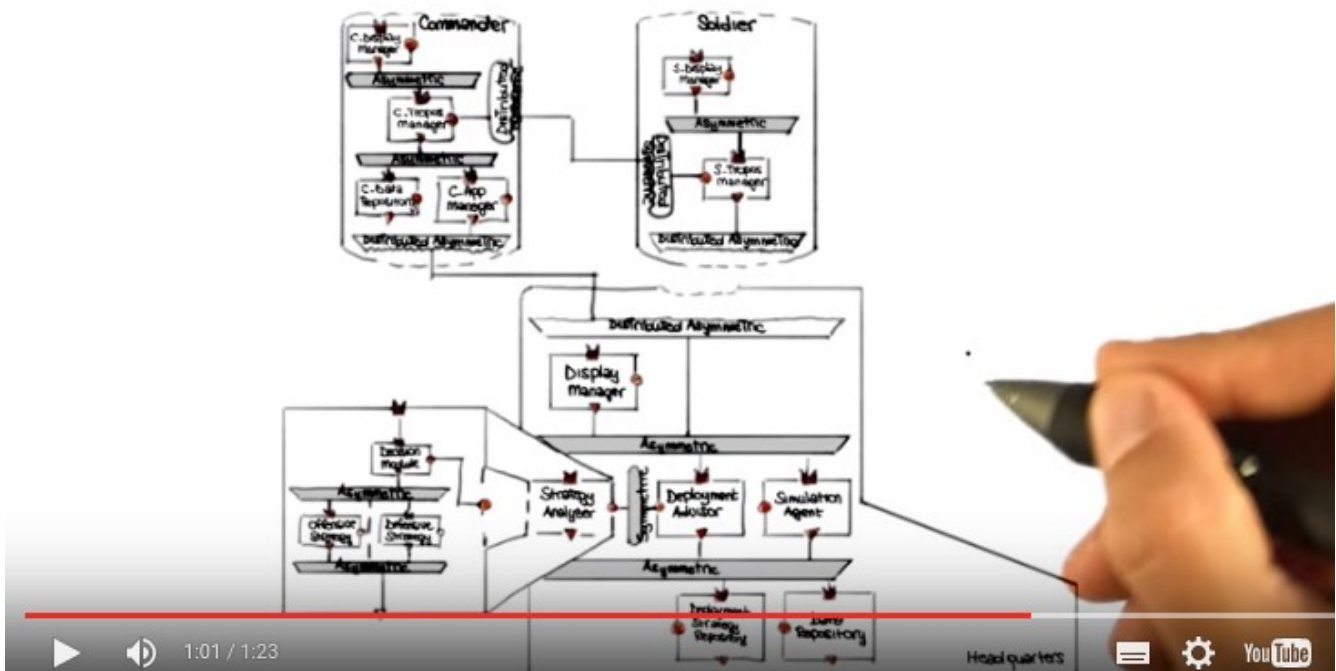
18. What I'm showing here is what an architectural configuration of a system might look like in practice. And as you can see, the configuration includes a set of **components, which are these rectangles** over here. The components have various kinds of ports, which are the ones marked here on the components with different graphical representations. And the components communicate through various types of **connectors, which are the grey elements** here which as you can see are used to connect the different components. And something else that you can notice by looking at this configuration is the fact that you can also have hierarchically decomposable. **For example, if you look at the strategy analyzer component, you can see that it has three subcomponents:** one, two, and three and two internal connectors as part of it. And it is worth recalling here that a component diagram as we said when first discussed in UML in the course, can also be used to represent an architectural configuration. So sometimes you will see architectural configurations represented as UML component diagrams.

AN EXAMPLE CONFIGURATION



19. A system cannot fulfill its purpose until it is deployed(配置,展開). And deploying a system involves physically placing the system's executable modules on the hardware devices on which they are supposed to run. So when you do that, you're basically mapping your components and connectors to specific hardware elements. Here in this diagram, for instance, I'm showing the same components that we saw in the previous diagram, but we see them deployed on a laptop(下面的大框), which is depicted here in this way, and on two smartphones that are represented here(上面的兩個小框), or PDAs, if you wish. So why do we this, why do we create a, a deployment perspective for our architecture? Well, because the deployment view of an architecture can be critical in assessing whether the system will be able to satisfy its requirement. Because doing this mapping allows you to discover and assess other characteristics of your system that you might not have considered up to now. For instance, using a deployment view like this one and knowing the characteristics of the hardware devices, one might be able to assess the system in terms of available memory. Is there going to be enough memory available to run the system, for example, on this device? Power consumption. Is the power consumption profile going to be larger than what the device can handle? Or again the required network bandwidth. Does the system have enough network bandwidth to enable the required interactions? And so on. So all of these characteristics, all of these qualities, you can assess when you do this final mapping of the components to the hardware elements.

DEPLOYMENT ARCHITECTURAL PERSPECTIVE



20. The last topic I want to cover in this lesson is [architectural styles](#). So, let's see what those architectural styles are. There are certain design choices that when applied in a given context regularly result in solutions with superior properties. What this means is that, [compared to other possible alternatives, these solutions are more elegant, effective, efficient, dependable, evolve-able, scale-able, and so on](#). Architectural styles capture exactly these solutions. They capture idioms that we can use when designing a system. For a more formal definition, let's look how Mary Shaw and David Garlan define a architectural style. They say that an architectural style defines a family of systems in terms of a pattern of structural organization; a vocabulary of components and connectors and constraints on how this components and connectors can be combined. So in summary we can say that an architectural style is a named collection of architectural design decisions applicable in a given context. [And I want to stress that it is important to study and know these architectural styles for several reasons. Because knowing them allows us to avoid reinventing the wheel.](#) It also allows us to choose the right solution to a known problem and in some cases it even allows us to move on and discover even more advanced styles if we know the basic ones. [So we should be familiar with architectural styles, what they are, and in which context they work, and in which context they do not work. So as to be able to apply them in the right situations.](#)

ARCHITECTURAL STYLES



An architectural style defines "a family of systems in terms of a pattern of structural organization; a vocabulary of components and connectors, with constraints on how they can be combined"

M. Shaw and D. Garlan, 1996

Basically, named collection of architectural design decisions applicable in a given context

21. So what does it mean to know architectural styles? There are many many, many architectural styles. So we cannot cover them all here. What I want to do instead is, I want to mention a few of those. And then I want to go in more depth, on one of them. So the first item I want to mention is pipes and filters. And pipes and filters indicate an architectural style in which a chain of processing elements, which can be processes, threads, co-routines, is arranged so that the output of each element is the input. Of the next one and usually with some buffering in between consecutive elements. A typical example of this, if you're familiar with Unix are Unix pipes, that you can use to concatenate Unix commands. Another set I want to mention is the event driven one. An even driven system, typically consists of event and mirrors, like the alarm over here, and event consumers, like the fire truck, down here, and consumers are notified when events of interests occurred and have the responsibility of reacting to those events. A typical example will be a GUI, in which widgets generate events and listeners listen to those events and react to them. For example, they react to the push of a button. A very commonly used architectural style is Publish-subscribe, represented by the paper boy. Over here. And this is an architectural style in which senders of messages, they're called publishers, do not send messages directly to specific receivers. Instead, they publish messages with one or more associated texts without knowledge of who will receive such messages. Similarly subscribers will express interest in one or more tags. And will all receive messages of interest according to such tags. A typical example of a publish-subscribe system, will be Twitter. And I'm pretty sure that most of you are familiar with the client-server architecture. In which computers in a network, assume one of two roles. The server provides the resources and functionality. And the client initiates contact with the server, and requests the use of those resources and functionality. Also in this case, a typical example would be email, in which an email server provides email storage and management capabilities, and an email client will use those capabilities. You may also be familiar with peer-to-peer, or P2P, systems. A P2P system is a type of decentralized and distributed network system in which individual nodes in the network, that are called peers, act as independent agents that are both suppliers and consumers of resources. This is in contrast to the centralized client-server model, where client nodes interact with the central authority. And I'm not going to say anything more about peer-to-peer, because I'm going to show you two examples, of peer-to-peer systems in the rest of the lesson. And you probably have at least heard of REST. Which in this case is not an invitation to relax as the graphic might indicate. But rather stands for Representational

State Transfer. REST is a hybrid architectural style for distributed hypermedia systems, that is derived from several other network based architectural styles. And that is characterized by uniform connector interface, and even if I'm not going to say anything else about the rest, I wanted to mention it, because it is an extremely well known architectural style. And the reason for this is that REST is very widely used, because it is basically the architectural style that governs the world wide web. So we use it all the time when we browse the internet, for instance.



22. Consider now the following architectural styles that we just saw: pipes and filters, event-driven, publish-subscribe, client-server, peer-to-peer, and rest. I'm showing you here, a list of four different systems, and I would like for you to mark here which architectural style, or styles, characterize Each of these systems. Again, mark all that apply.



Consider the following architectural styles that we just saw: pipes and filters (A), event-driven (B), publish-subscribe (C), client-server (D), peer-to-peer (E), REST (F). Mark which architectural style(s) characterizes the following systems. (Mark the corresponding letter(s) next to the systems)

Android OS	A[]	B[X]	C[X]	D[]	E[]	F[]
Skype	A[]	B[]	C[]	D[X]	E[X]	F[]
World-Wide Web	A[]	B[]	C[]	D[X]	E[]	F[X]
Drop Box	A[]	B[]	C[]	D[X]	E[]	F[]

以上選項寫到這裡更清楚:

- A. Pipes and filters
- B. Event-driven
- C. Publish-subscribe
- D. Client-server
- E. Peer-to-peer
- F. REST

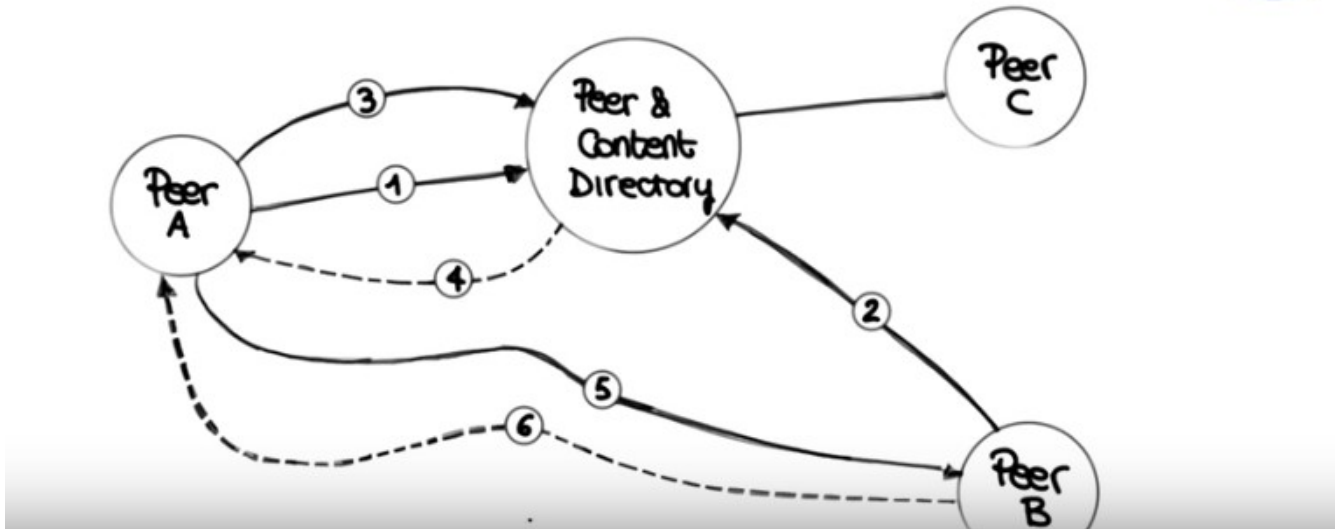
23. Okay, let's start with the Android Operating System. The Android system, heavily based on the generation and handling of events, so it is mostly an event driven system. However, it also has some elements of publish, subscribe, in the way elements in the system can register for elements of interest. So we can mark both styles here. So what about Skype? We haven't discussed Skype yet. So here we probably had to take a little bit of a wild guess. But as we will see in more detail in the rest of the lesson. Skype is mainly a peer to peer architecture, with some minimal elements of a client server architecture. For example, when you start Skype and sign in to a conceptually centralized server. So let's move to the World Wide Web. As we just discussed, the World Wide Web is based on a rest architecture. And because rest style, is a hybrid derived from other architectural styles, including the client server architectural styles. Both of those styles apply here. And finally Dropbox is by and large, a client server architecture. As conceptually, we upload our documents to a Dropbox central server, and get the files from the same server.

24. We're not going to be able to study in that, any of the architectural styles that we just discussed. However, [I want to at least discuss two representative examples of P2P architectures](#). Because, these are systems that you probably used, or at least, you used one of them. And, they will allow me to highlight some interesting points. So, as we just mentioned, P2P systems are decentralized resource

sharing and discovery systems. And the two systems that I want to discuss, and that are representative of this kind of architectures, are [Napster and Skype](#). And you may or may not be familiar with Napster, but I'm pretty sure that you know Skype.

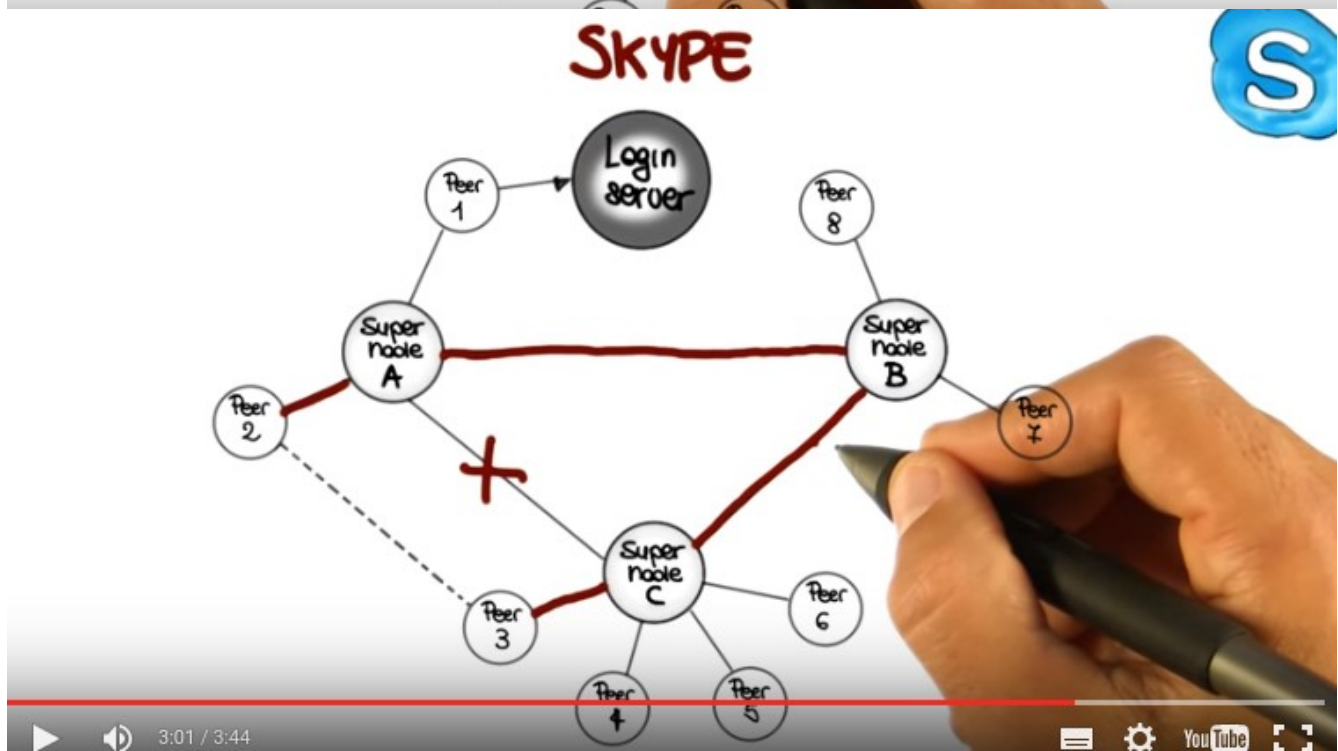
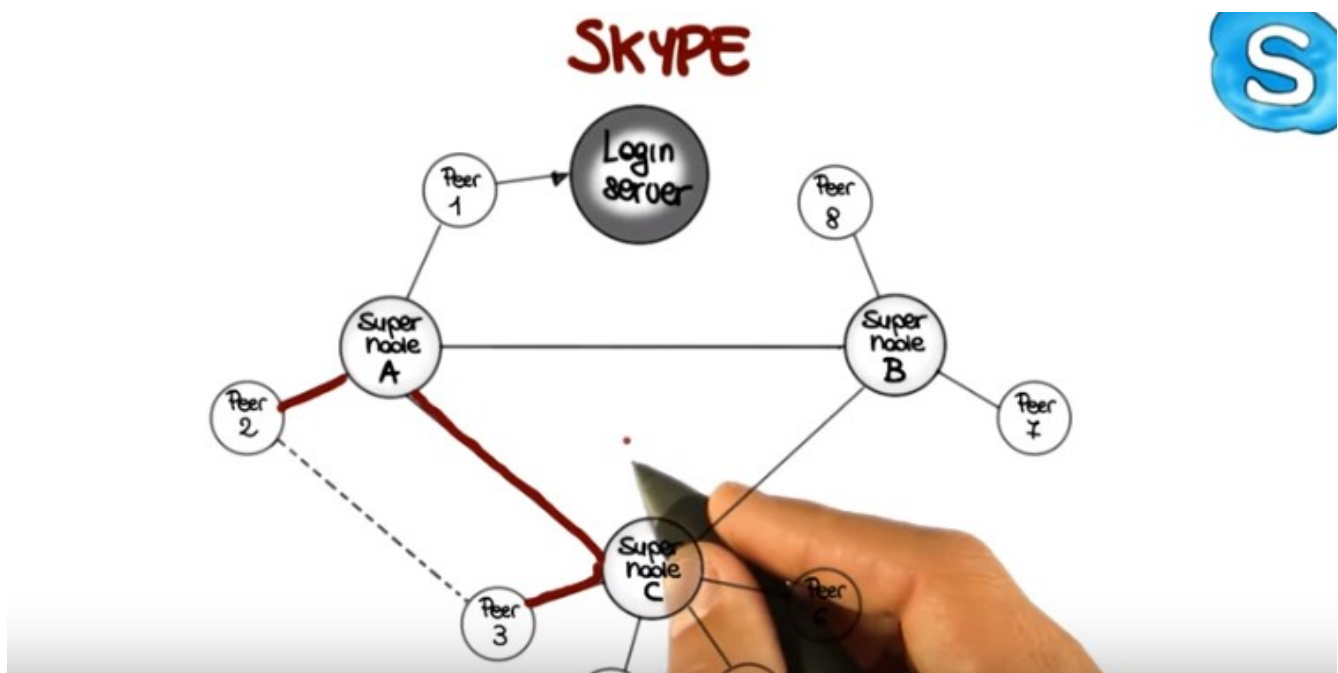
25. So let's start by considering Napster. In its first incarnation, [Napster was a peer-to-peer file sharing system. And it was mostly used, actually, to illegally share mp3s. Which is also why it got sued and later on, it basically ceased operations.](#) But nevertheless, I think Napster is an interesting example of mixed architecture. And I'm going to illustrate the way Napster works by showing you, here, the basic configuration of Napster and the interactions between its elements. So let's look at how such interaction can take place for the three peers shown here. And in this case [Peer A and B are the only ones really involved in the action.](#) So let's look at a typical sequence of events for the Napster system. We have Peer A that will start by registering, here, with the content directory. Peer B will also register with the content directory. [And when these two peers register, the content directory will know what kind of content they can provide.](#) Later on, Peer A will request a song. And one first observation that we can make, based on this interaction, is the fact that, [up to now, this is a purely client-server system.](#) This is the client. This is the client. And this is the server. And the interaction is a typical client-server interaction. But now we're at the point in which things start to change a little bit. At this point, [after Peer A has requested the song, the peer and content directory will look up its gigantic index and will see that Peer B actually has the song that Peer A requested. So it will send to Peer A a handle that Peer A can use to connect directly to Peer B. So this is where the system is no longer a client-server system. Because at this point, the two peers are connected directly. So at this point, we have a peer-to-peer interaction.](#) And, after getting the request from Peer A, then Peer B will start sending the content to Peer A. And I said earlier that one of the useful things about representing an architecture and interaction within an architecture graphically, is the fact that it allows you to spot possible problems. And in this case, by representing the Napster architecture in this way, and by studying how things work, we can see that there's an issue with the architecture of Napster that will not make this architecture scale. As some of you might have already noticed, this peer and content directory is a single point of failure, and is very likely to cause problems when the number of peers grows too large. Because at that point, there are going to be too many requests to the peer and content directory, and the peer and content directory is unlikely to be able to keep up with all the requests. So some changes in the architecture will have to be made. In the case of Napster, we didn't see this problem occurring because, as I said earlier, Napster got sued and ceased operation before the problem actually manifested. Now looking at the system for an architecture-style perspective, we can see that Napster was a hybrid architecture with both client-server and peer-to-peer elements. And something I would like to stress here, is that this is not at all uncommon. So in real world nontrivial architectures, it is very common to see multiple styles used in the same system. The next system that we will consider, Skype, is instead, an example of a well-designed, almost purely peer-to-peer system.

NAPSTER



26. So even if you're too young to have used Napster, I'm pretty sure that most of you know and use Skype, a Voice Over IP and instant messaging service. Many of you, however, probably don't know how Skype works. To understand that, [let's have a look at Skype's architecture](#), which I'm sketching here, and which is a peer-to-peer architecture with a small twist. So first of all, by looking at the architecture we can see that whereas Napster was a client-server system with an element of peer-to-peer, Skype is a much more decentralized system. Why is that? Well, if we look here, we can see that there is a login server, okay? This node over here and that means that every Skype user has to register with this centralized service. But that's the only interaction of this kind within Skype. [After you log in, all you get is a connection through a super node like this one. So, what are super nodes? Super nodes are highly reliable nodes with high bandwidth that are not behind a firewall and that runs Skype regularly, which means that nodes that shut down Skype occasionally will not qualify as super nodes. And one interesting thing about super nodes is that they're not owned by Skype. They're just regular nodes that get promoted by Skype to super nodes, and that know about each other.](#) So basically Skype has an algorithm that looks at the nodes in the system and decides whether a node can be a super node or not based on its characteristics. So now that we've discussed super nodes, let's see what will happen if peer two wanted to communicate with peer three. So let's represent this by creating a dashed line between peer two and peer three. In this case, peer two will contact this super node, which is super node A. And super node A, based on its knowledge of the Skype network and the position of the super nodes, will contact and route the communication through super node C, which will in turn route the communication to peer three. And in that way peer two and peer three will be able to communicate with each other. And this will happen just as if peer two and peer three were connected directly, as peers, even though the communication goes through two super nodes. Another thing that is important to know about the behavior of Skype is that, if the link between super nodes A and C were to go down. So let's assume that there is a problem with this link, then Skype will automatically, or automagically reroute the communication through super node B, which will in turn reroute it super node C, which will again reroute to peer three. So peer two and three will still be connected, but this time they will be going through three super nodes. And just in case you wondered, this is exactly what happens when you are talking over Skype. The quality of the communication degrades, and you are reconnected. So

there is this rerouting going on through different nodes. So although this architecture is more effective than the Napster's one, it is not without problems. For example, you might remember that a few years ago, Skype went down for about 36 hours. And later on it was discovered that the cause was the algorithm used by Skype to determine which nodes could be super nodes. And remember, as I said, that one requirement for these nodes is that have to up all the time. So what happened is most of the super nodes were running on Windows machines, and Microsoft pushed a critical patch that required a reboot to be installed. So a large number of machines, and therefore a large number of super nodes were down roughly at the same time throughout the globe. And Skype's algorithm for determining super nodes didn't have enough nodes to work with. So the whole system crashed and burned. So the message I want to give here, is that when you have a large peer to peer distributed system, such as this one, such as Skype, these kind of perfect storms can happen. Because you are not really in control. Because the control is distributed. So the algorithms become more complex. So to wrap up our Skype example, in case you are interested, Skype then fixed the issue by changing the algorithm for identifying super nodes. And more recently actually, Skype ditched peer-to-peer super nodes altogether.



27. And I want to conclude this lesson with three takeaway messages. The first one is that having an effective architecture is fundamental in a software project. Or as I say here, a great architecture is a ticket to a successful project. To put it in a different way, although a great architecture does not guarantee that your project will be successful, having a poor architecture will make it much more difficult for your project to be successful. The second message is that an architecture cannot come about in a vacuum. You need to understand the domain of the problem that you're trying to solve in order to define an architectural solution that fits the characteristics of the problem. So a great architecture reflects a deep understanding of the problem domain. And finally, a great architecture is likely to combine aspects of several simpler architectures. It is typical for engineers to see problems that are new, but such that parts of the problems have already been solved by someone else. An effective engineer should therefore, first of all, know what is out there, know the solution space. Second, an engineer should understand what has worked well and what has failed miserably in similar occasions in the past. And finally, an effective engineer should be able to suitably combine existing solutions appropriately to come up with an effective overall solution for the specific problem at hand. And this is just as true in the context of software architectures. When designing a software architecture, you should innovate only as much as you need to and reuse as much as you can. As we said early in the lesson, by doing so, that is, by innovating only as much as you need to and reusing as much as you can, you will be able to avoid reinventing the wheel. You will be able to choose the right solution to known problems. And identify suitable solutions for new problems. So ultimately, you will be able to realize an effective software architecture that will help the success of your project.

TAKEAWAYS



A great architecture is a ticket to success



A great architecture reflects deep understanding of the problem domain



A great architecture normally combines aspects of several simpler architectures