Questions on the syllabus

Core interoperability standards

1) What is SOAP? (Which are the types of communication in SOAP?)

*"**SOAP** is a stateless, one-way network application protocol that is used to transfer messages between service instances described by WSDL interfaces. It relies on HTTP as a transfer protocol and XML as a message format. Each SOAP message consists of an Envelope containing an optional Header and a mandatory Body. The SOAP header contains information relevant to how the message must be processed. It may contain information about the source, intermediates, and the destination. When a node receives a message, it determines what role it will assume by inspecting the role attribute, which identifies the intended target. The SOAP Body is the immediate child of the envelope. It contains either a payload or a fault message.*

Soap supports two communication styles

*• **Remote Procedure Call (RPC)**: clients express their request as a method call with its parameters and get back a response envelope, enclosing the requested result or a fault message*

*• **Document-Style**: the SOAP Body can be structured as wish because the content is any arbitrary XML instance. The Document-Style is also referred to as Message-Oriented style."*

Simple Object Access Protocol. XML-based, over HTTP.

It allows requesters to invoke services. It's composed with a Header (control info, optional) and a Body (payload or fault msg).

PRO: machine-readable, interoperability

CONS: XML overhead, REST is used more

2) What is WSDL? (What is a request-response operation in WSDL? How do abstract and concrete WSDL interface differ?)

*"It is de-facto **a standard XML based language for describing the public interface of a web-service**. A public interface is a kind of "contract" between provider and consumer, describes only information relevant to both parties. WSDL is used to describe:*

• What a service does

• Where it resides

• How to invoke it

The WSDL description consists of two parts:

• an abstract service description with the operations supported by the service, their parameters, and abstract data types

• a concrete endpoint implementation which binds the abstract interface to an actual network address, to a specific protocol, and concrete data structures"

Web Service Description Language.

To send a message to a service, the sender must know **many** things, such as names/parameters for the operation, address of the service, which transport protocol to use, etc. We need a *(machine-readable)* description of servers -> WSDL. It's like a contract between the provider and the consumer, describing ***what*** but not ***how*** (no implementation details, only relevant Infos).

There are 2 types of interfaces so that I can have one abstract and many different implementations. The ***abstract*** one is about operations, parameters and abstract data types. The ***concrete*** one is instead about bindings to a network address, specific protocol, and concrete data structures.

NOTE: Binding defines how operations are mapped via concrete protocols, messaging styles and encoding styles. The ***msg exchange patterns*** are *One-Way, Request-Response (*typical procedure call), Notification and Solicit-Response.

3) What is REST? (How can we create/update/access resources in REST? Which are the pros and cons of REST?)

*"REST stands for Representational State Transfer and it is **an architectural style for message exchange**, lighter than SOAP. Each service is viewed as a resource and it is identified with a URI (that stands for Uniform Resource Identifier). Each request contains enough context information to process the message. REST has a uniform and simple interface. Resources are manipulated using a fixed set of operations:*

• PUT: creates a new resource (IDEMPOTENT)

• POST: modify the state of a resource (NOT IDEMPOTENT)

• GET: retrieve the state of a resource

• DELETE: delete an existing resource

Every interaction in REST is stateless, but using hyperlinks can be built a stateful connection. In this case, the client must hold the state of the session. In REST all resources are decoupled from their representation. The content of a resource can be accessed in a variety of formats (XML, JSON, HTML)"

Representational State Transfer. An alternative to the WSDL+SOAP.

It's ***resource-centric*** (SOAP is msg centric). The main idea is that the WS is a set of web pages and each action is to either get or work on the information (resources). The ***state transfer*** is *partial* to clients but *consistent*.

PRO: Simplicity (low learning curve, needs minimal tooling), efficiency (lightweight protocol), scalability (stateless, can serve tons of clients).

CONS: confusions on best-practices (REST, Lo-REST, ..).

4) What is OpenAPI?

*"OpenAPI is a **simple description language** that lists HTTP endpoints, their usage, and the structure of input/output data. The OpenAPI defines a standard, programming language for the description of the interface for REST APIs, which allows both humans and computers to discover and understand the capabilities of a service without requiring access to source code. When properly defined via OpenAPI, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. OpenAPI documents describe an API's services and represented in either YAML or JSON formats."*

A.k.a. Swagger, to create an open standard to describe REST APIs. It's JSON-based (so it's lightweight), used to specify HTTP endpoints, how to use them, etc.

Microservices

5) Why microservices?

To shorten the lead time for new features/updates, to change part of the application without redeploying everything and to reduce chords across functional silos. The direct consequences are that we get independent things that can work with each other without knowing how others do their stuff, and finally that we can scale effectively.

6) Which are the main characteristics of microservice-based architectures?

"The microservice-based architecture is an approach to developing a single application as a suite of small services, each running in its own process or container, and communicating with lightweight mechanisms. These services are built around business capabilities and independently deployable by a fully automated deployment mechanism. The main characteristics are:

*• **Componentization via Services**: develop an application as a set of services. The services are independently deployable so if you change a single service, you'll redeploy only that service. Communication with lightweight*

mechanisms (HTTP, REST-ish protocols or dumb message bus). Possibility to develop component with different languages and different storage techniques

• ***Services organized around Business Capabilities**: Conway's law- = "any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure". The microservice approach to division is splitting up into services organizes around business capability.*

• ***Products not Projects**: preferring the notion that a team should own a product instead of a project. This brings developers into day-to-day contact with how their software behaves in production and increases contact with their users*

• ***Decentralized Data Management**: prefer letting each service manage its own database. Decentralizing responsibility for data across microservices has implications for managing updates consistently. Distributed transactions are difficult to implement and as a consequence microservice architectures emphasize transaction-less coordination between services with explicit recognition that consistency may only be eventual consistency*

• ***Independently deployable services***

• ***Horizontally scalable services**: possibility to deploy as many replicas of a service in order to, for example, increase the bandwidth or the audience as needed.*

• ***Design for failure**: need to be designed so that they can tolerate the failure of services. Any service call can fail, the client has to respond to this as gracefully as possible"*

Shorter lead time for new features/updates, change part of the application without redeploying everything and reduce chords across functional silos.

7) Which are the main pros and cons of microservices?

*"**PRO:***

• ***failure** of service can have a **small impact** on the whole application*

• ***short time** to develop and deploy means services are updated and added fast*

• ***scale fast***

• *reduce **chords** across teams*

CON:

• *communication **overhead** and latency*

• ***complexity when** the number of microservices is very **large** (orchestration vs choreography)*

• ***wrong cuts** of the monolith **translates in a distributed version** of it, that is worse than the monolith itself*

• *distributed database is difficult to manage if not properly split*

• ***hard to monitor***

Things to do to deploy a very good microservice-based architecture:

• *investment in automation so continuous delivery*

• *a realistic approach to faults and failures (fault-tolerant)*

• *fail fast*

• *microservice has a private data store. Team can change the structure of the database without coordinate with other teams"*

Service Orientation, organize services around business capabilities (cross-functional teams), decentralized data management (each service its DB -> EVENTUAL CONSISTENCY, better to fix mistakes than losing business), independently deployability, horizontally scalability, fault resilience.

8) How can architectural smells affecting design principles of microservices be detected and resolved via refactoring?

Using some tools like Microfreshener.

Independent Deployability: *one container per service*

Horizontal scalability: endpoint-based service interactions

service discovery, msg router, msg broker

no API Gateway

add one

Isolation of failures: Wobbly Service Interactions

add MB, bulkheads (like circuit breaker), use timeouts

Decentralization: Shared Persistence (multiple services accessing the same DB)

split DB, add data manager, merge services (☹)

ESB misuse

Smart services, dumb pipes!

Single-layer teams

Follow the AGILE idea, full-stack devs, …

9) Which refactoring can be applied to resolve architectural smell X?

….. ?

10) What are Flask and Celery?

*"**Flask** is a microframework for Python to deploy easily and fast a web-application. It has a built-in web-server and debugger, unit testing support, RESTful request dispatcher, Jinja2 as UI templating. It is very light and with no dependencies. It is well documented. It interoperates with other frameworks and plug-ins.*

***Celery** is an asynchronous task queue which is based on distributed message passing. It is focused on operations in real-time, but support scheduling as well. The execution units, called tasks, are executed concurrently, on a single or more workers servers using multiprocessing."*

Flask is a micro web framework written in Python to easily create web services.

Celery is an open-source asynchronous task queue used in production systems (like Instagram).

Software testing

11) What is the development/release/user testing?

*"• **Development testing** is when the system is tested during development to discover bugs and defects*

*• **Release testing** is when the complete version of the system is tested before the release. The aim is to check that the system meets the requirements of stakeholders and it is good enough for external use. Usually, this test is made by a separate team*

*• **User testing** is made by users or potential users in a real environment*

Development Testing:

*• **UNIT Testing:** make individual program units test. Tests should call all methods with different input parameters and provide coverage of all object features*

*• **COMPONENT Testing:** components are made up of several interacting objects, accessible through an interface. Testing composite components should focus on showing that the interface behaves according to its specification*

*• **SYSTEM Testing:** components are integrated to create a version of a system. We focus on checking that components are compatible, interact correctly and transfer the right data at the right time across their interfaces. Also testing the so-called "emergent behavior" of a system"*

These 3 are the main different phases in testing.

Development: test *while* developing. It's split in:

Unit test: test individual program units (Partition Testing, Guideline-based testing)

Component test: many units together, test interfaces (Parameters, Procedural, Msg Passing, Shared Memory)

System test: Test component interfaces (like the whole system)

Release: different teams test before giving the software to the user.

User: testing in their environment.

12) What is partition/load testing?

"It is a software testing technique that divides the input test data of a method into partitions. An advantage of this approach is it reduces the time required to perform testing of a software due to less number of test cases. A partition is a set of inputs that should be processed in the same way, because for example, they have the same characteristics (all positive/negative numbers, all odd/even numbers, etc). A good way to test partitions is to choose test cases on the boundaries and in the midpoint of each partition.

***Load Test** checks if the system is capable to operate in a given workload. This is also useful to find bottlenecks and see how it behaves in peak loads."*

Partition testing is about identifying a group of inputs that should be processed the same way *(negative numbers, ..)*. We then choose tests from within these groups, taking bound and the middle value.

13) What is test-driven development?

*"TDD is **an approach to program development in which you interleave testing and code development**. You develop code incrementally along with a set of tests for that increment, so tests are written before code. The steps for TDD process are:*

• Identify the increment of functionality

• Write a test and implement it as an automated test

• Run the test to see if it fails. You did not implement the functionality

• Implement the functionality and re-run the test

• Once all tests pass, you start again

Benefits of TDD:

• Code Coverage: since you write automated tests before the implementation of functionality, every code line should have tested

• Regression Testing: with automated tests, you can run again the tests on the codebase. This is useful when you insert a new feature that can introduce new bugs

• Simplified Debugging: when a test fails, you know where the problem lies

• System Documentation: tests act as a form of documentation"

Interleave testing and code development. **Incremental,** wait until the test is OK to proceed. This implies:

Code coverage: each line has at least one associated test.

Regression testing: new changes don't break older and already stable versions.

Simplified debugging: we work on one functionality at a time, so…

System documentation: each test is a form of documentation.

User stories

14) What is a user story (for)?

"A user story is a very high-level definition of a requirement, containing just enough information so that the developers can produce a reasonable estimate of the effort to implement it. A user story is a short description of a feature or behavior that the application must have for the user. Important considerations are:

• Stakeholders write user stories not developers

• Use the simplest tool like card

• Remember non-functional requirements

• Indicate the estimated size. A good story should take 1 or 2 days of work. Split long stories into several shortest stories

• Indicate the priority in a simple manner, for example using numbers from 1 to 10 or use the low/medium/high discriminator. Making them simple you can add, remove and re-prioritizing cards according to the new requirements

Example: AS A [USER ROLE] I WANT TO [GOAL] SO THAT [BENEFIT]

• Scheduling: the priority affects when the work will be done to implement that requirement. Stakeholders are responsible for prioritizing requirements, define new requirements and change the priority

• Estimating: developers are responsible for estimating the effort required to implement the stories There are three common phases when stories will be worked during an agile project:

• Inception (beginning): create a stack of user stories to identify the scope of your system

• Construction: identify new stories, split stories that cannot be implemented in a single iteration, re-

prioritize and remove stories

• Transition: sometimes new stories are identified during this phase. This is not very common as the

focus of the release is on hardening the system and not to introduce new functionalities."

In short, user stories are very slim and high-level requirements artifacts, containing just enough information so that the developers can produce a reasonable estimate of the effort to implement it. They're produced as a result of the requirement elicitation phase, where the interviewed user provides the requirements for the product.

Its template is **AS A… I WANT TO… SO I CAN …**

15) What is the priority/effort/size of a user story?

*"The **priority** of a user story is the **importance of the functionality** to be implemented. User stories are usually sorted in a decreasing order (higher first, lower last) to prioritize the implementation of feature/functionality. Priority can be changed during the development to meet the new requirements imposed by stakeholders. Indicate the priority in a simple manner, for example using numbers from 1 to 10 or use stars from 1 to 5. Making them simple you can add, remove and re-prioritizing cards.*

*The **effort** indicates the **time that a pair of programmers will spend to implement the story**. If a story will take too much time, the team can split that story in smaller and easier stories."*

Priority: the importance of the story from the POV of the project stakeholder

Effort: an estimation of the needed work from the POV of the developer

Size: The time needed to complete a User Story

Splitting the monolith

16) When and where to start splitting a monolith codebase?

*"**WHEN** the monolith starts to be **too large for introduce new features**. It has lost high cohesion and low coupling when you cannot deploy some change without potentially impacting the rest of the monolith, so you have to redeploy the entire system.*

***WHERE**. From a portion of **code that can be treated in isolation and work without impacting the rest of the codebase** (seam). It is possible to group similar code identifying high-level bounded contexts. Some languages have the notion of namespace, others like java have the notion of packages. By definition, they represent a cohesive and loosely coupled bounded context. So a good start is to create namespaces or packages that representing these contexts and analyze dependencies between packages. Splitting a monolith can be a challenge, so it must be done incrementally and carefully to minimize problems during this procedure. Start to split where we can get the most benefit. Consider some drivers in the split:*

*• **Pace of Change**: keep in mind that there will be lots of changes*

*• **Team Structure**: splits codebase according to team location or skills*

*• **Security**: splitting the monolith can improve the security level of the application*

*• **Tangled Dependency**: remove the seam that has the least dependency and turn it into a microservice*

*• **Technology Split**: think to adopt a new technology/language that fits the needs of a particular seam*

*• **Database**: think about the seam also in the database."*

When: since the monolith grows over time (new functionalities/updates) it could be difficult to maintain it as a unique codebase (high coupling -> little changes have an impact on the whole thing), so that's when we should aim to split it.

Where: we have to find the seams so the portions of code can be treated in isolation and working on it doesn't impact the rest of the codebase. **Seams == service boundaries**

17) How to split databases? (e.g., how to break foreign key relationships?)

*"Understand which parts of code read to and write from the database. **Detect database-level constraints like foreign-key relationships.***

***Foreign-key relationship**. Remove the relationship and expose data via API. This introduces overhead, the relationship and consistency have to be managed by the service and not by the DBMS, data has to be moved over the network.*

Shared Static Data

• Duplicate tables: potentially consistency problems

• Data as code: embed data into code

• Data as service: create a microservice to store and retrieve data (overkill?)

Shared Mutable Data

Recognize namespace/package and expose data via API."

We first need to understand which parts of code read from and write to, in order to detect database-level constraints, like FK relationships.

FK relationships: Instead of having two services that access the same table, one of the services will request via APIs to the other one the needed pieces of information. This introduces overhead and may need to implement consistency checks across services.

Others:

Shared STATIC data: duplicate (☹), treat it as a config file (😊) or push data into a service (overkill?)

Shared MUTABLE data: The mutable data become a service that can be invoked by other services via its API

Shared TABLES: If the same table is shared between services but there are different concepts, split it into 2 tables.

18) What about transactions when you split?

"Transactions are atomic and powerful tools in a database. if it fails, it rollback the state and maintain consistency in the database. Distributed transactions can be a solution. It can be done with a Transaction Manager that orchestrates the various transactions being done. It can be also done with Two-Phase Commit:

• Voting phase: participants tell the manager whether its local transaction can go ahead or not

• Commit: if the manager gets all yes vote from all participants, it tells them all to go ahead and perform

their commits, otherwise send a rollback action

Vulnerabilities

• if the transaction manager goes down, transactions never complete and resources are locked

• if one participant fails to vote, transaction blocks

• if one participant fails after the vote, consistency of data is lost

*Another solution is **eventual consistency**.*

Try Again Later. Simply retry the transaction later. Queue up requests and accept that the system could be in an inconsistent state, but soon or later will be consistent."

When we split the DB, the #calls increases so we lose the "atomic flavor", which implies the use of *Transactions*. The problem is when one of these calls fails: the possible solutions are

Rollback: from a consistent state to another, which implies compensations, retry, etc.

Distributed transactions: a transaction manager who orchestrates many actors + voting phase

EVENTUAL CONSISTENCY

19) What is the SAGA pattern?

*"The SAGA pattern is a pattern for business distributed transactions. A SAGA **represents a high-level business process that consists of several low-level requests that each update data within a single service**. Each request has a Compensating Request that is executed when request fails. Note that some actions are not undo-able in the conventional sense. Compensating Requests semantically undoes a request by restoring the application's state to the original state. Requests and Compensating Requests have to be Idempotent so that repeated identically action must provide the same outcome. Compensating Requests must be commutative. Requests can abort, which triggers a Compensating Request but Compensating Requests CANNOT abort.*

The SAGA pattern can be implemented in:

• Event-driven choreography: where there is no central coordination. Each service produces and listen to events to decide if an action should be taken or not

• Orchestration: a coordinator service (SAGA Execution Controller) is responsible for centralizing the SAGA decision. The coordinator attempts transactions in concurrent or risk-centric order.

There are two recovery models:

• Backward: if a transaction fails, undo all successful transaction

• Forwards: retry every transaction until all are successful."

SAGA pattern is a way to implement transactions that involve multiple microservices. A SAGA is a sequence of local transactions and for each transaction, an event is generated. If one of the transaction fails, all the Saga executes a series of compensating transactions. It must be decided if it will be a backward or forward recovery and the choice between orchestration vs choreography-based sagas.

20) What is eventual consistency?

*"Eventual consistency **is a consistency model** used in distributed computing to achieve high availability that guarantees that if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value. Accept that the system will get itself in a consistent state at some point in the future. Use fault-tolerant message queues and SAGA pattern."*

It's a way to manage consistency in distributed environments *(CAP Theorem: distribution -> Availability XOR Consistency).* We accept that the system will get itself into a consistent state at some point in the future. Informally, it guarantees that if there are no new updates to a given item all accesses to that item will be returned to the last updated value. We introduced this because the cost of fixing mistakes is smaller than losing business.

21) What is an (event) data pump?

*"In a reporting problem scenario, a Data Pump **is a way to collect data in order to make a report on it**. The Data Pump service is responsible for getting the data from a database and push it to the reporting system. Data Pump should be managed and version-controlled by the team managing the data-source service. Usually, the data is pushed in a scheduled manner. The drawback is that is a coupled system, but in this case, the reporting is much easier.*

*The **Event Data Pump** is a Data Pump that works in a microservice architecture and make use of events. Each time a data changes its state, a microservice can emit an event. Event subscriber pumps data into the reporting system. This means that pushing data could be more efficient because it is sent only deltas and they are sent only when an event occurs. Since it uses a provider-subscriber pattern, there is no tight coupling. A drawback is that, since events are broadcasted, this could not scale well"*

Data pump: Rather than have the reporting system pull the data, we could instead have the data pushed to the reporting system. It's one of the two ways (either pull data [not scalable] or this one, where you push data into a reporting system). The Data Pump modifies the data so that they're readable from the reporting schema. This allows doing reporting when we do backup operations (piggybacking).

Event data pump: each state change generates an event and each customer subscribed to that event, when generated, will pump data into the reporting database. It's faster, because in the *Data Pump* we do it periodically, here we follow the temporal nature of events. Drawback: may not scale well for large volumes of data because all required information must be broadcasted as events.

Cloud-based software engineering

22) What is a container/image/volume?

*"Images and containers can be seen as classes and objects respectively in an OO programming view. Multiple containers can be created from a single image. **An image is an executable package that includes everything needed to run an application. A container is a run-time instance of an image**. A container is a lightweight, isolated environment consists of a Docker image, an execution environment and a standard set of instructions. A volume is a specially-designated directory within one or more containers.*

***Volumes** are designed to persist data, independent of the container's life cycle."*

Container: a lightweight (can run dozens at the same time) and portable (they hold an isolated version of an OS) store for an application and its dependencies (no need to configure and install *docker run*). It's a running system defined by images.

Image: it's a collection of more layers and some meta-data produced by Docker. A layer is a collection of changes to files.

Volume: are the mechanism for persisting data generated by and used by Docker containers.

23) Which are the differences between a virtual machine and a container?

*"Containers and virtual machines are two ways to deploy multiple isolated services on a single platform. A **container** contains the necessary to run the application, making **use** of the **kernel host**, the least resources needed, **making it lightweight**. Container systems have a **lower overhead** than VMs but it is **less secure** and provides **least isolation**.*

*A **virtual machine runs a full OS** with virtual access to host resources. In general, VMs provide an environment with **more resources than most applications need**, but provides a **greater deal of security**. Also, Virtual Machines can operate at the hardware level already, whereas a container engine must operate starting from a shared OS."*

Two main differences:

1) a container system requires an underlying operating system, while hypervisors (VM) have their own OS. So on the container, you only have the parts of the OS you need (dependencies). So containers are lighter than VMs because they don't have to activate a dedicated OS.
2) Containers are simpler to build
3) Containers are less secure because there's less isolation than VMs. The Hypervisor is the separation layer among the different VMs, the Docker Engine is on top of the Host OS.

24) What is image layering in Docker?

*"A docker image is built up from a series of layers. **Each layer except the very last one is read-only**. Each layer is only a set of differences from the layer before it. The layers are stacked on top of each other."*

Each update to any file in the container produces a layer. This way I can create newer images starting from one of these layers.

25) What is the effect of docker run/commit?

"docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

*• creates a write-able **container** of the specified image*

• starts the container

• run the command provided

docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]

*Create a new **image** from a container's changes.*

It is possible to create a new image for debug purpose passing –change as an option providing some Dockerfile instructions."

Docker run: launches containers. The following arguments are the name of the image we want to use *(Debian)*. The image is downloaded *(from DockerHub)* if there's no local copy, and that image is then turned into a (new) running container.

Documentation: it first creates a writeable container layer over the specified image, then starts it using the specified command.

Docker commit: create a new image from a container's changes. The returned value is the unique ID for our image. *Documentation: the commit operation will not include any data contained in volumes mounted inside the container. By default, the container being committed and its processes will be paused while the image is committed (to prevent data corruption).*

26) What is Docker Compose for?

It's a tool for defining and running multi-container Docker applications. 3-step process:

Define your app's environment with a *Dockerfile* so it can be reproduced anywhere. Define the services that make up your app in *docker-compose.yml* so they can be run together in an isolated environment. Finally, run *docker-compose up* and Compose starts and runs the entire app.

Business process modeling

27) What is a parallel/exclusive/inclusive gateway in BPMN?

"Gateways are used to control how the Process flows through Sequence Flows as they converge and diverge within a Process. A gateway is useful only if the flow must be controlled. The term gateway implies that there is a gating mechanism that either allows or disallows passage through the Gateways. As tokens arrive at a Gateway they can be merged together on input and/or split apart on output. BPMN gateways are decision points that can adjust the path of a flow based on certain conditions.

*• **Parallel Gateway** (A Jesus Cross inside a Diamond): is used to create and synchronize parallel flows. It creates paths without checking any conditions. A token traverse each outgoing path. For incoming flows, it waits for all incoming flows before triggering the outgoing Sequence Flows.*

*• **Exclusive Gateway** (Empty Diamond): is used to create alternative paths within a Process flow. Only one of the paths can be taken. Each alternative is associated with a condition (expression). A default path can optionally be identified and it is taken if none of the conditions are satisfied. If no condition is satisfied and there is no default path, an exception is raised. A Converging Exclusive Gateway is used to merge alternative paths. Each incoming Sequence Flow token is routed to the outgoing Sequence Flow without synchronization.*

*• **Inclusive Gateway** (Circle inside a Diamond): is used to create an alternative but also parallel paths within a Process Flow. All condition expressions are evaluated, and for each one that is true, a token traverse the corresponding path. Also here a default path can be designed. A default path can optionally be identified and it is taken if none of the conditions are satisfied. If no condition is satisfied and there is no default path, an exception is raised. A Converging Inclusive Gateway is used to merge a combination of alternative and parallel paths. A control flow token arriving at an Inclusive Gateway MAY be synchronized with some other tokens that arrive later at this Gateway."*

Gateways are used to control how the process flows.

Parallel [+]: it's an AND gateway, which implies that all the outgoing flows will be activated

Exclusive [ ]: it's a XOR gateway, which implies that only one of the outgoing flows will

Inclusive [O]: it's an OR gateway, which implies that one or more branches are activated depending on formula in each flow.

Each of these gateways will have it's "counter-part" *(the Join*), where all active inputs branches must be completed.

28) What is an error event in BPMN?

*"Error Events (lightning inside a circle) are used to handle errors during the execution of a certain activity or at a certain point in the flow of a process. Error Event **always interrupts the Activity** to which it is attached. The Error Event can be:*

*• **Error Intermediate Events**: can only be attached to the boundary of an activity. That the error can*

only be caught once (it is similar to the try/catch mechanisms of programming languages).

*• **Error End Events**: are used to indicate that a certain process path ends with an error. This error will*

propagate to the parent processes in the same way as in Error Intermediate Events."

It's a possible termination state of a sub-process, when something goes wrong.

29) What is Camunda?

*"Camunda **is a framework supporting BPMN for workflow and process automation**. It provides a RESTful API that allows you to use your language of choice. After defining a BPMN process, Camunda can directly call services via built-in connectors. It supports RESTful and SOAP services. However, it only allows scaling on process instances NOT on microservices, because each microservice can handle more than one process instance.*

A more interesting pattern is known as External Task. Units of work (Task) are provided in a Topic Queue that can be polled by RESTful workers, possibly interacting with microservices. You can see it as a Farm Skeleton, in which the Emitter is a queue of tasks for each topic and workers poll a task from a topic queue when it's able to compute a new one of that topic. It allows the scaling of process instances, of workers and microservices. We have three steps:

• Through process engine, creation of external task instance

• External worker fetches and locks externals tasks

• Process engine and external worker complete external task instance"

It's an open-source platform for workflow and decision automation that brings business users and software developers together. It provides a BPMN standard-compliant workflow engine. It's a lightweight, Java-based framework that provides a REST API and dedicated client libraries to build applications connecting to a remote workflow engine.

30) Can you describe the two usage patterns of Camunda?

Pattern A: It's the one where after we have defined a BPMN process, Camunda calls services via *connectors* (the arrows). It won't scale on the microservices but the processes' instances, so all the instances will refer to the same microservice.

Pattern B: *(external task)* each task is put in a queue and this queue is pulled by *workers* that refer to a microservice. In this way we can scale on process instances (they refer to the same queue), we can scale workers and every worker instance is associated with a microservice instance, so we can scale them too.

Note: Camunda won't call the workers, they'll offer voluntarily.

31) What is a workflow net?

*"**An extension of Petri-nets: one of the best-known techniques for formally specifying business processes.***
Workflow nets focus on the control flow behavior of a process. Main characteristics:

• Graphical representation eases communications between different stakeholders

• Process properties can be formally analyzed

• Various supporting tools are available

Like Petri-nets:

• Transitions represent activities

• Places represent conditions

• Tokens represent process instances

A Petri-net is a workflow net if and only if:

• There is an initial place with no incoming edge

• There is a final place with no outgoing edge

• All places and transitions are located on some path from the initial place to the final place"

A Petri Net is a Workflow Net if and only if there's an initial place *(no incoming edges)*, a final place (*no outgoing edges)*, and all places and transitions are located in some path from the initial to the final one.

32) What is a sound workflow net? What is a live/bounded Petri net?

Theorem: A Workflow N is sound if and only if (N^) is *Live* and *Bounded*. [N^ = N + transition from F to S]

A Workflow Net is sound if and only if, for each execution of the network, at some point, I reach the final marking *(one token only)* **and** there are no ghosts transitions. This can be automated by checking if a Petri Net is Live and Bounded.

Live (PN): if and only if, for each reachable state *M'* and for each transition *t*, it exists a state *M''* reachable from *M'* which enables *t*. *Very strong, wherever you may stop, for each transition, we can fire that transition!*

Bounded (PN): if and only if, for each place *p* there's a number *n* such that for each reachable state, the number of tokens in *p* is lower than *n*.

33) How can we model BPMN parallel/exclusive/inclusive gateways with workflow nets?

Figure 2: Parallel Gateway

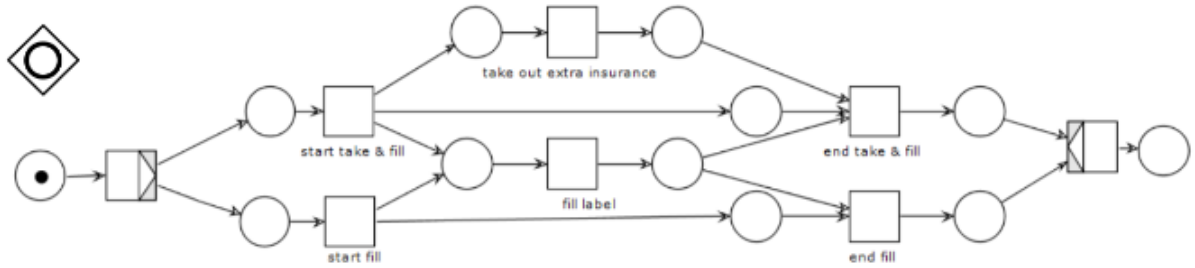


Figure 3: Exclusive Gateway

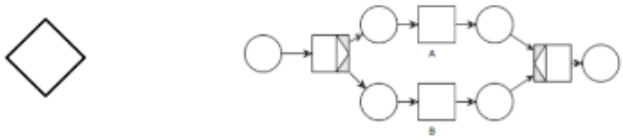
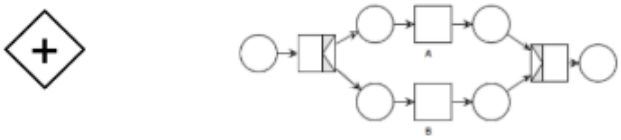


Figure 4: Inclusive Gateway



Note: the XOR is a user decision of where to go, the OR is whoever comes first.

Note by Bruni: the XOR can "scale" *(if-else)*, OR can't *(if, if, if, …)*

Security

34) What are authentication and authorization?

Authentication: the act of confirming the truth of an attribute of a single piece of data.

Authorization: to specify access rights to resources.

35) What are LDAP/SAML/OIDC/OAuth 2.0 for?

LDAP: *Lightweight Directory Access Protocol*, it's a protocol for accessing and maintaining distributed directory information services over an Internet protocol network. Figuratively, it induces a tree structure where the leaves are the users associated with a specific role, which gives some access rights (following access policies).

SAML and OIDC are standards to exchange security Infos in a Federated AuthN

SAML: *Simple Assertion Mark-up Language.* XML-based

OIDC: *OpenID Connect.* REST APIs, JSON

OAuth 2.0: It's a standard for AuthZ *(not backward-compatible with OAuth 1.0)*. It's a framework that allows third-party applications to obtain limited access to an HTTP service. **How:** the authorization is an *access token*, convey the authorization to a third-party application and use the access token on a protected resource.

36) What is vulnerability analysis (with Bandit)?

It's an analysis made on the intermediate representations of the source code, by checking: *Names Database/Symbol Table, the Abstract Syntax Tree, the Control Flow Graph, the Call Graph.* **Note:** the vulnerabilities that static analysis can find are not all the possible vulnerabilities, plus there may be some detected vulnerabilities which are false positives.

We used in lab *Bandit*, a static analysis tool designed to find common security issues in Python code, by exploiting known patterns (plugins).

Fog computing

37) What is Fog computing?

"Fog Computing integrates Edge devices and cloud resources. Fog systems generally use the sensor-process-actuate and stream-processing programming models. Sensor stream data to IoT networks, applications running on fog devices subscribe to and process the information, and the obtained insights are translating into actions sent to actuators. Edge and cloud resources communicate using machine to machine standards. Fog computing is a distributed paradigm that provides cloud-like services to the network edge. The deployment model IoT + Cloud can exploit huge computing power but suffers from high latencies and bandwidth bottleneck. Fog computing extends the cloud towards the IoT to overcome these problems. Fog computing is a horizontal architecture that distributes resources and services of computing, storage, control and networking anywhere along the continuum from Cloud to Things, thereby accelerating the velocity of decision making."

Fog Computing aims at extending the Cloud towards the IoT to better support ***latency-sensitive*** and ***bandwidth-hungry*** IoT applications. It's like a middle layer between the Cloud and IoT devices, this because the number of devices is increasing very quickly in time *(estimated 50bln devices in 2020)*, and this means that there will be too much data to be processed that the Cloud alone cannot support, so we need to filter and process data *before* the Cloud.

38) What is / how difficult is it to deploy applications to Fog infrastructures? (Could you sketch the proof of

NP-hardness of CDP?)

"Modern applications usually consist of many independently deploy-able components that interact together in a distributed way. Such interactions may have stringent QoS requirements, e.g. latency, bandwidth, security. When deciding where to deploy application components and which device has to run it, we should check their hardware, software, IoT and QoS requirements against the offering available infrastructure. We can define CDP (Component Deployment Problem) as an NP-Complete problem, given:

• A multi-component application A with requirements R

• A distributed Fog infrastructure I (nodes and links)

• A set of objective metrics M

Consider the Subgraph Isomorphism Problem (SIP). Given 2 graphs G and H, a solution to the SIP answers the question: is there a subgraph G' such that H can be mapped, nodes and links to G'? To reduce SIP in CDP in poly-time we need:

1. vertices V in G are Fog nodes with available resources set to 1

2. edges (u, v) in G are node-to-node links in I with available bandwidth set to 1

3. vertices V in H are components of A with resource requirements set to 1

4. edges (u, v) in H are component-component interactions in A with bandwidth requirements set to 1

Since we reduced the CDP into SIP and being SIP NP-Complete, CDP is also NP-Complete"

It's an NP-hard problem, and we prove this by taking another known NP-problem and reduce it to the previous one in polynomial time *(our problem is at least as hard as the other one)*.

Subgraph Isomorphism Problem (SIP) as our known problem: *Given two graphs G and H, a solution to the SIP answers the question "Is there a subgraph G' in G such that H can be mapped one-to-one, nodes, and links, to G'?*

Dim: **(step 1)** change all vertices *v* of *G* into Fog nodes in *I* with available resources set to 1.

(step 2) change all edges in *(u, v)* of *G* into node-to-node links in *I* with available bandwidth set to 1.

(step 3) change all *v* of *H* into components of *A* with resource requirements set to 1.

(step 4) change all edges *(u, v)* of *H* into component-component interactions in *A* with bandwidth requirements set to 1.

39) How can we assess the security level of an application deployment? How can we model trust?

SecFog: it's a declarative methodology to assess the security level of multi-component application deployments in Fog scenarios, whilst considering trust relations among involved stakeholders. Once you have the Fog structure, you "take" the *security* capabilities. From the Application, you get the c*omponent* requirements and the *application requirements,* plus some *Custom Security Policies,* the *deployment,* and the *trust* degree. SecFog takes all of this, matches it with some of its custom policies and produces a security assessment. *(Problog2)*

Model:

*trusts(*X, X).

*trusts**2**(*A,B) *:-*

*trusts(*A,B).

trusts**2**(*A,B) :-*

*trusts(*A,C), *trusts**2**(*C,B). // A trusts C and there's a path from C to B

40) Could you please read rule X of the operational semantics of Fog Director?

[https://elearning.di.unipi.it/pluginfile.php/29214/mod_folder/content/0/b%20Mimicking%20FogDirector%20application%20management.pdf?forcedownload=1]

Add Node: If the Client wants to add a new node that is not already in the nodes of the infrastructure (such that the new nodes will be the old ones disjoint union with n), then I'll move to another state where C has changed and N includes the new node.

Delete Node: If the Client wants to remove a node, that node must be in the present nodes at that time, then I'll move to another state where C has changed and N is the old N minus the node n.

Publish App: If the Client wants to publish an application A with idA and the idA is not already present in the published apps (such that the new P will be the old one plus the pair (idA, A), then we have a new set of published apps that includes the new one.

New Deployment: If the Client decides to start a new deployment of an already published application *(idA, A belongs to P)* identified by idA and the deployment of idD must not be already present in the Under Deployment Apps nor Running Apps *(not in D u R)*, then I move from P D F to a new set of under deployment apps that includes this new deployment with "null" deploying node and empty requirements.

Deploy App: If the Client decides to deploy the application indicated by idD on the node n, we must check that in the under deployment apps there must be that app, n must be in N and the free resources must be enough to satisfy the requirements of the application, I then move in a new state where the application under deployment I've changed the parameters of idD assigning to him the node n and V as the requirements.

Bind Thing: If I want to bind *t* to a specific deploy of a requirement *t_r* and the deployment must exists, the node must not be null, that the requirement that I want to assign is effectively an application's requirements, that there's not already something assigned to that requirement, that t is effectively a Thing accepted by my infrastructure, that the type of t and the type of the requirement are the same, then I will move to a state where the set of Under Deployment Apps is edited updating the quintuple identified idD (the one I bind) adding to the requirements the pair (t_r, t) (assigning to t_r the Thing t).

Start App: If I want to start an application under deployment, I must be sure that that application is under deployment, that the node where I deploy it is not null, that all the requirements of the application there is a thing associated to that requirement, then I'll move to a new state where I moved the tuple identified by idD from D to R

Resource Alert: if C receives an Alert for the deploy D and the quintuple identified by D is running and the node where the app is running doesn't satisfy the requirements of the app, then the status won't change but I receive an alert

A2T Alert: If I receive a getAlert on a specific Thing, I have to be sure that the application is a running application, that exists a Thing called t_r that is a requirement of the application and that there's something bound to t_r and there's a link (n, t) with quality q and q doesn't satisfy the quality of service for which I received the alert, then same as before

Things Info: *getters*

Published Apps Info: *getters*

Nodes Info: *getters*

Deploying Apps Info: *getters*

Links Info: *getters*

Running Apps Info: *getters*

No name rule: “You’re not alone in the infrastructure”

I -> I’ represents changes in the infrastructure (new things, new links, new resource allocations, etc).

41) How can we use a tool like FogDirSim for app management?

Writing a management script, try it in FogDirSim, predict the KPI (Key Performance Indicator), refine the management based on the KPI, use the management script in production and repeat. This way we can have a simulation of the system performance, getting an estimation of the uptime/downtime, energy consumption, robustness to failures, capacity to adapt to changing workload and much more.

42) How can we monitor the Fog “gently” / in a scalable manner?

We saw FogMon, a lightweight fault-resilient monitoring technique for Fog infrastructures. The main idea is that there are two types of distributed P2P agents, the **followers** measuring monitored metrics and **leaders**, aggregating metrics from a group of followers and gossiping them to other leaders.

> -> Leaders collect measurements from followers in their groups and spread data to other leaders O(log L) rounds to spread information on avg, O(L log L) messages exchanged overall.

This induces scalability because, with N nodes and L leaders, there will be N/L nodes per group (leader), so N^2/L^2 e2e measurements for bandwidth and latency. If L is nearly the sqrt(N) then O(N) e2e measurements.

Talking about fault-tolerance, the data replication between Leaders guarantees tolerance: when a Leader fails, the followers rearrange into other groups and they keep working in case of network interruption between Leaders.