POLITECNICO
MILANO 1863

# Software Engineering 2

## Software Design Exercises

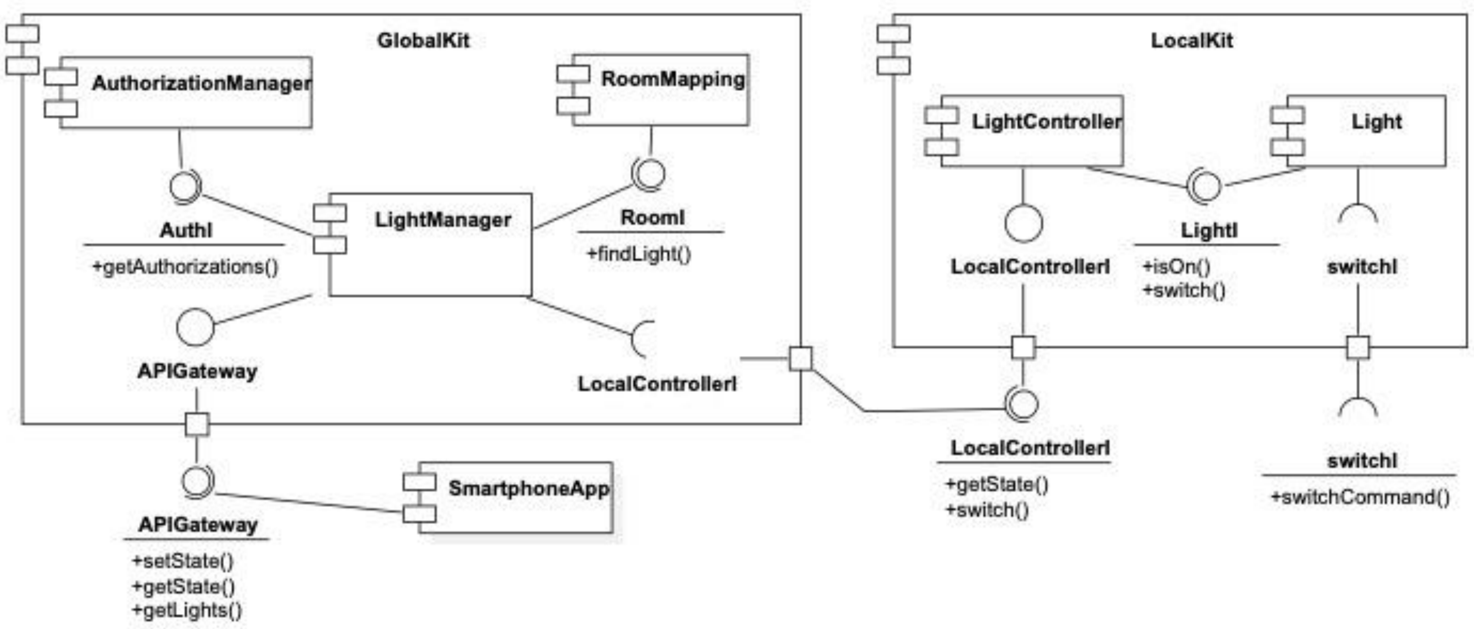
# SmartLightingKit

# SmartLightingKit

- SmartLightingKit is a system expected to manage the lights of a potentially big building composed of many rooms (e.g., an office space).
- Each room of the building, may have one or more lights.
- The system shall allow the lights to be controlled – either locally or remotely – by authorized users. Local control is achieved through terminals installed in the rooms (one terminal per room). Remote control is realized through a smartphone application, or through a central terminal installed in the control room of the building.
- While controlling the lights of a room, the user can execute one or more of the following actions: turn a light on/off at the current time, at a specified time, or when certain events happen (e.g., a person enters the building or a specific room). Moreover, users can create routines, that is, scripts containing a set of actions.
- SmartLightingKit manages remote control by adopting a fine-grained authorization mechanism. This means there are multiple levels of permissions that may even change dynamically.
- System administrators can control the lights of every room, install new lights, and remove existing ones. Administrators can also change the authorizations assigned to regular users. These last ones can control the lights of specific rooms. When an administrator installs a new light in a room, he/she triggers a pairing process between the light and the terminal located in that room. After pairing, the light can be managed by the system.

# SmartLightingKit – part 1

- This diagram describes the portion of the SmartLightingKit system supporting lights turning on/off and lights status checking.



- What can we infer from the analysis of this diagram?

# SmartLightingKit – part 2

- The diagram is complemented by the following description
  - `GlobalKit` is the component installed in the central terminal. It can be contacted through the `APIGateway` interface by the `SmartphoneApp` component representing the application used for remote control. `GlobalKit` includes:
    - `RoomMapping`, which keeps track, in a persistent way, of lights' locations within the building's rooms;
    - `AuthorizationManager`, which keeps track, in a persistent way, of the authorizations associated with each user (for simplicity, we assume that users exploiting the operations offered by the `APIGateway` are already authenticated through an external system and include in their operation calls a proper token that identifies them univocally); and
    - `LightManager`, which coordinates the interaction with all `LocalKit` components.
  - Each `LocalKit` component runs on top of a local terminal. Also, each `LocalKit` exposes the `LocalControllerI` interface that is implemented by its internal component `LightController`, which controls the lights in the room. Each light is represented in the system by a `Light` component which interacts with the external system operating the light through the `switchCommand` operation offered by the latter.

# SmartLightingKit – part 2 (cont.)

- Q1:
  Analyze the operations offered by the components shown in the diagram and identify proper input and output parameters for each of them.
- Q2:
  Write a UML Sequence Diagram illustrating the interaction between the software components when a regular user wants to use the smartphone application to check whether he/she left some lights on (among the lights he/she can control).

# Operations

- **APIGateway**

- *getLights*: input = userID; output = list[lightID]
- *getState*: input = userID; output = list[(lightID, state)]
- *setState*: input = userID, lightID, state; output = none

- **AuthI**

- *getAuthorizations*: input = userID; output = list[(roomID, localKitID)]

- **RoomI**

- *findLight*: input = roomID; output = list[lightID]

- **LocalControllerI**

- *switch*: input = lightID; output = none
- *getState*: input = lightID; output = state

- **LightI**

- *isOn*: input = none, output = True/False
- *switch*: input = none, output = none

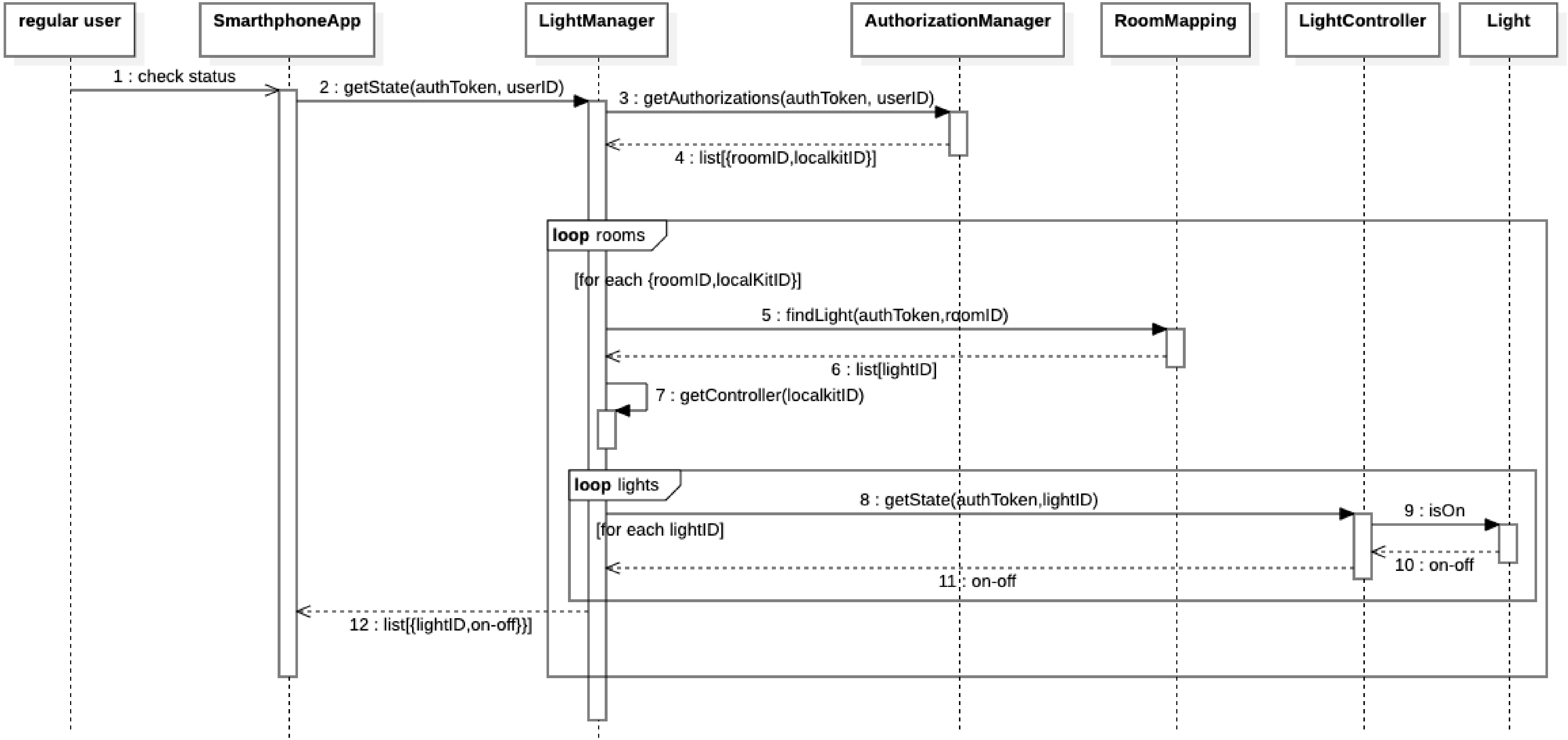
- **SwitchI**

- *switchCommand*: input = none; output = none

- **Note**

- State = On/Off;
- All the operations of `APIGateway`, `AuthI`, `RoomI`, `LocalKitI` receive as input also the authentication token.

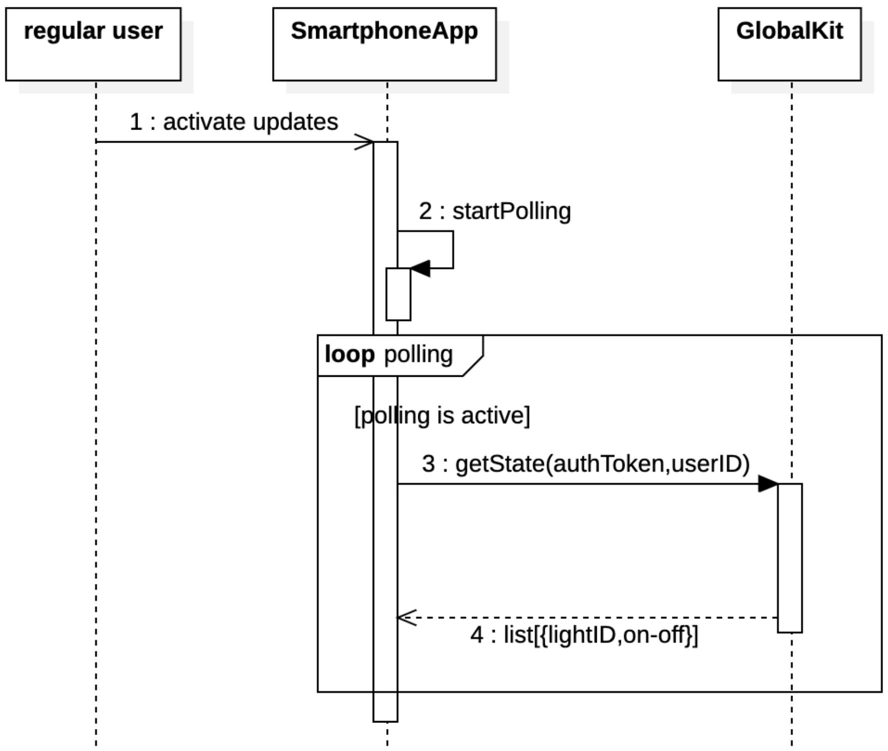
# Sequence diagram


regular user
SmarthphoneApp
LightManager
AuthorizationManager
RoomMapping
LightController
Light
1 : check status
2 : getState(authToken, userID)
3 : getAuthorizations(authToken, userID)
4 : list[{roomID,localkitID}]
loop rooms
[for each {roomID,localKitID}]
5 : findLight(authToken,roomID)
6 : list[lightID]
7 : getController(localkitID)
loop lights
8 : getState(authToken,lightID)
[for each lightID]
9 : isOn
10 : on-off
11 : on-off
12 : list[{lightID,on-off}]

# SmartLightingKit – part 3

- Assume that, at a certain point, the following new requirement is defined:
    - **NewReq**: *“The smartphone application should allow users to activate/deactivate the receival of real-time updates about the state (on/off) of all the lights they can control.”*
- Define a high-level UML Sequence Diagram to describe how the current architecture could accommodate this requirement.
- Highlight the main disadvantage emerging from the sequence diagram.

# Realizing NewReq



- Problem: the current architecture does not support updates in push mode.
- The `SmartphoneApp` carries out a continuous polling process to retrieve the status of all the lights even in case it does not change.
- This propagates also internally to `GlobalKit` and the involved `LocalKits`, thus resulting in a potentially significant communication overhead.

# KitchenDesigner

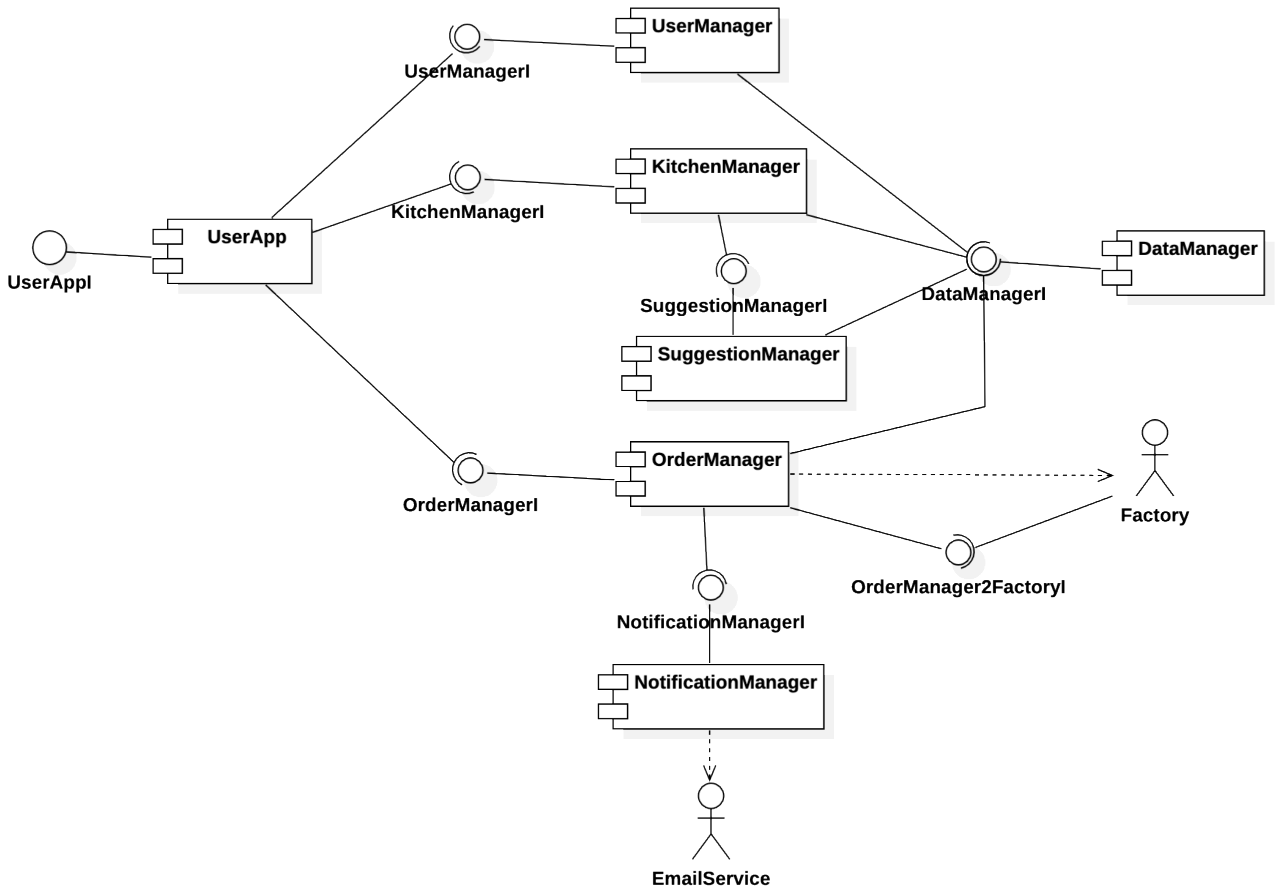
# June 24th, 2021 exam

- We want to build an application, KitchenDesigner, that allows users to define the layout of kitchens and to insert in such layouts the furniture and appliances (refrigerators, stoves, dishwashers, etc.) that go in them.
- For simplicity, we consider only rooms that have rectangular shape. Users can define the physical features of the room (length, width, height). Moreover, they can add pieces of furniture and appliances, and move them around. The position of each piece of furniture/appliance is given by the 3D coordinates of the lower left corner of the bottom side of the item, and by its orientation (which is the angle with respect to the x axis, and which can only be a multiple of 90 degrees).
- Users register with the application to be able to store and retrieve their designs. After a user finalizes his/her kitchen design, he/she can ask to have the kitchen delivered to a desired address; in this case, the kitchen is sent to production, and the user is given a probable date of delivery (producing a kitchen can take a few days, or even weeks). For simplicity, we do not consider payment. When the kitchen is ready to be delivered, the user is notified of the confirmed date of delivery.
- The system keeps track of the designs created by users, to identify the most common combinations of pieces of furniture and appliances. Hence, upon request by a user, given a draft layout for the kitchen, the system returns a list of possible pieces of furniture and appliances that might be added to that kitchen.

# June 24th, 2021 exam

- Assuming you need to implement system KitchenDesigner analyzed above, identify the most relevant components and interfaces describing them through UML Component or Class Diagrams.
- Provide a brief description of each component. For each interface identified in the previous point, list the operations that it provides. You do not need to precisely specify operation parameters; however, you should give each operation a meaningful enough name to understand what it does; you can also briefly describe what information operations use/produce.
- Define a runtime-level Sequence Diagram describing the interaction that occurs among the KitchenDesigner components when the user asks for a list of suggested elements to be added to the kitchen. If useful (optional), provide a brief description of the defined Sequence Diagram.

# Solution – component diagram



# Solution – description of components and interfaces

- **UserApp**: This is the front-end for users. It allows them to interact with the system by offering the following functions through interface UserAppI:
  - Register (input: user data)
  - Login (input: userid and password)
  - Create a new kitchen project (input: name)
  - Set the dimensions of the kitchen (input: dimensions)
  - Add an item to kitchen (input: item to be added)
  - Move item in kitchen (input: item to be moved, new position/orientation)
  - Remove item from kitchen (input: item to be removed)
  - Ask for a suggestion (returns suggested elements)
  - Finalize kitchen
  - Place order
- The module can keep track of the current kitchen being designed, so the user operates on the "open project", and the information does not need to be included in the calls each time.

# Solution – description of components and interfaces

- **UserManager**: This component offer, through interface UserManagerI, the basic functions for handling users:
  - Register (input: user info)
  - Login (input: user id and password)
- **KitchenManager**: This component provides interface KitchenManagerI, which handles functions related to the management of kitchens (excluding placing the order, which is handled by another component):
  - Create a new kitchen project (input: name)
  - Set the dimensions of the kitchen (input: kitchen id, dimensions)
  - Add an item to kitchen (input: kitchen id, item to be added)
  - Move item in kitchen (input: kitchen id, item to be moved, new position/orientation)
  - Remove item from kitchen (input: kitchen id, item to be removed)
  - Ask for a suggestion (input: kitchen id, returns suggested elements)
  - Finalize kitchen (input: kitchen id)
  - These operations are similar to those offered by the UserApp, but they also include the id of the kitchen which should be modified.

# Solution – description of components and interfaces

- **SuggestionManager**: This component provides, through interface SuggestionsManagerI, functions related to the retrieval of suggestions:
  - Get suggestion (input: kitched id)
- The idea is that the component periodically retrieves kitchen designs from the DataManager, mines them, and identifies which combinations of items are most common. Hence, it only provides a single function, for getting the outcome of this mining. Hence, the computation is mostly offline, the “get suggestion” function compares what is present in the kitchen with what is most common, and suggests additional elements.

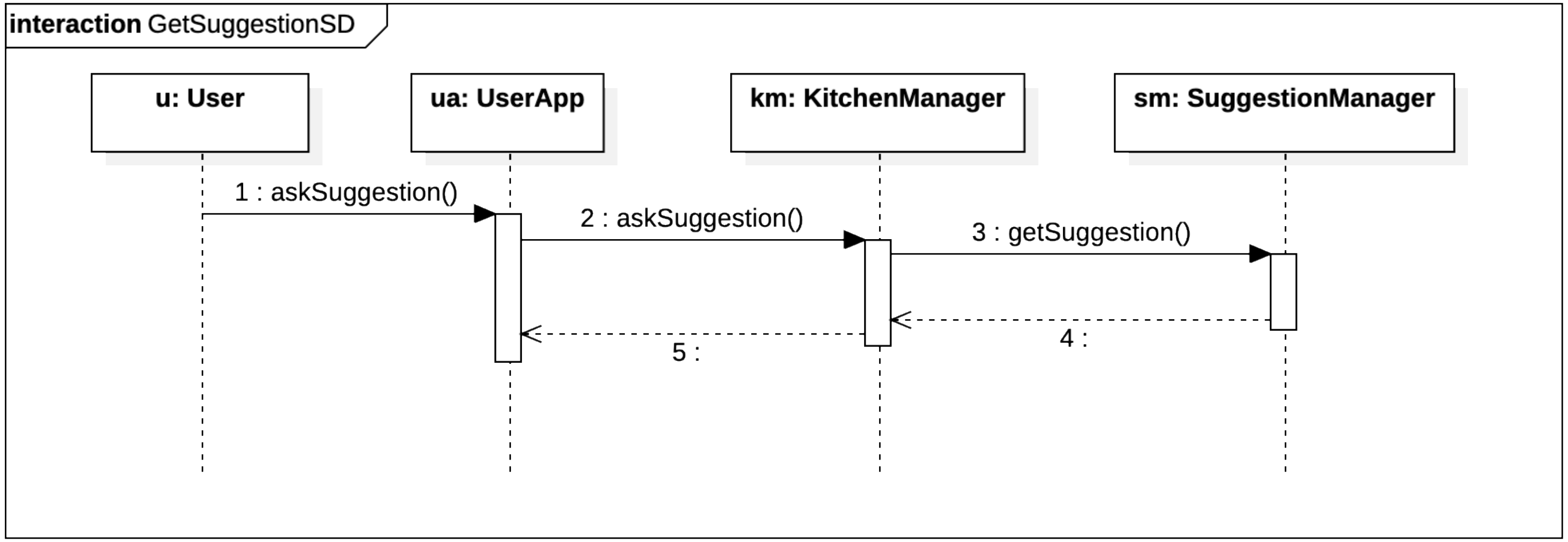
# Solution – description of components and interfaces

- **OrderManager**: This component provides, through interfaces OrderManagerI and OrderManager2FactoryI functions related to the management of orders. These are used by 2 different clients. Interface OrderManagerI is used by UserApp; it provides the following function
  - Place order (input: kitchen id)
- which is used to start the process to produce a kitchen. To handle the order the OrderManager needs to notify the factory of the new order. The handling of the order, and in particular its creation, is outside of the scope of the KitchenDesigner application; this is represented in the diagram by the fact that there is an interaction with the factory. When the order is complete, the OrderManager is informed of this through interface OrderManager2FactoryI (to be used by the external actor “factory”) which provides the following operation:
  - Kitchen completed (input: kitchen id)
- Which simply informs the OrderManager that the kitchen is indeed ready (hence the user can be notified of the date of its actual delivery).

# Solution – description of components and interfaces

- **NotificationManager**: This component handles the notification to users, in particular updates on when the kitchen will be delivered. It provides the following function through interface NotificationManagerI:
  - Notify user (input: message to be sent, recipient)
- The notification is sent as an email, and for this reason it goes through an email server, which is an external component, outside of the system.
- **DataManager**: This component handles the data of the system, which consists essentially of Users and Kitchens. It provides interface DataManagerI, which includes all necessary functions to handle CRUD operations on data.
- The only backend component that does not need to interact with DataManager is NotificationManager, because it relies on information provided by OrderManager.

# Solution - sequence diagram



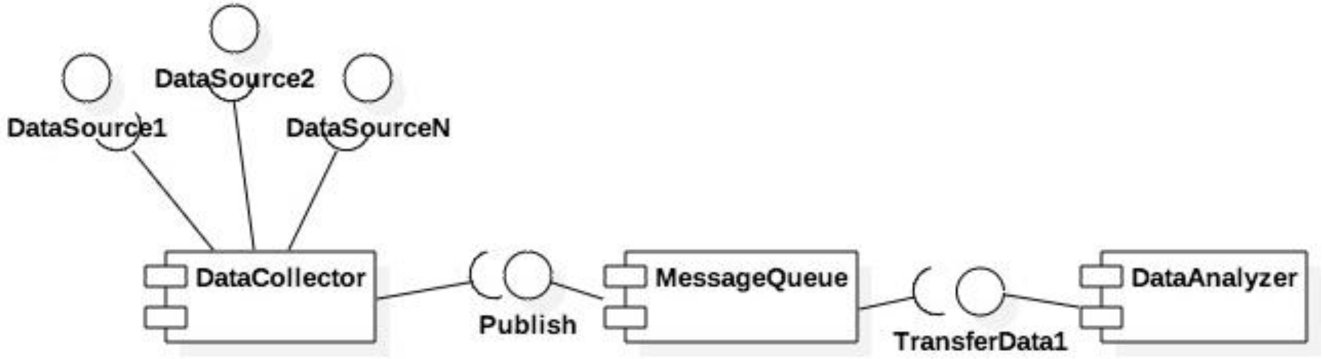
- Mining of the designs is done asynchronously, not when a suggestion is requested, but offline, so it is not represented in this interaction.

# Data Analysis Architecture

# Exercise on Alloy and architectures

(exam of July 13th, 2018)

- Consider the following UML component diagram.



This diagram describes a software system that acquires and elaborates information from a number of different sources by polling them periodically. The DataCollector component is exploiting the interfaces offered by some data sources to acquire data and the interface of the MessageQueue to pass collected data to the other components. The MessageQueue exploits the interface offered by the DataAnalyzer to pass the data to this component.

# Exercise (cont.)

- Write in Alloy the signatures that model a DataSource, a DataCollector, a MessageQueue and a DataAnalyzer. Make sure that you represent in the model the connections between components that are highlighted in the UML component diagram.
- Assume that we decide to replicate the DataCollector component. Model in Alloy the following possible configurations of the system:
  - **Configuration 1**: Each DataCollector replica is connected to a disjoint subset of DataSource components.
  - **Configuration 2**: All DataCollector replicas are connected to all DataSource components.
  - **Configuration 3**: DataCollector components are classified in master and slaves. There is always one DataCollector that acts as *master*.

# A possible solution

```
sig DataSource {}

sig DataCollector {
  sources: set DataSource,
  queue : MessageQueue
}

sig MessageQueue {
  analyzer: DataAnalyzer

}


sig DataAnalyzer {}
```

# A possible solution (cont.)

```
sig Configuration {
  sources: set DataSource,
  collectors: set DataCollector,

  queue: MessageQueue,
  analyzer: DataAnalyzer
}
{ // all DataCollector components are connected to the
  // same MessageQueue, which is connected to the
  // DataAnalyzer of the configuration
  all coll : collectors | coll.queue = queue
  queue.analyzer = analyzer

  // also, the DataSource components used by the
  // DataCollector ones are exactly
  // those of the configuration

  collectors.sources = sources
}
```

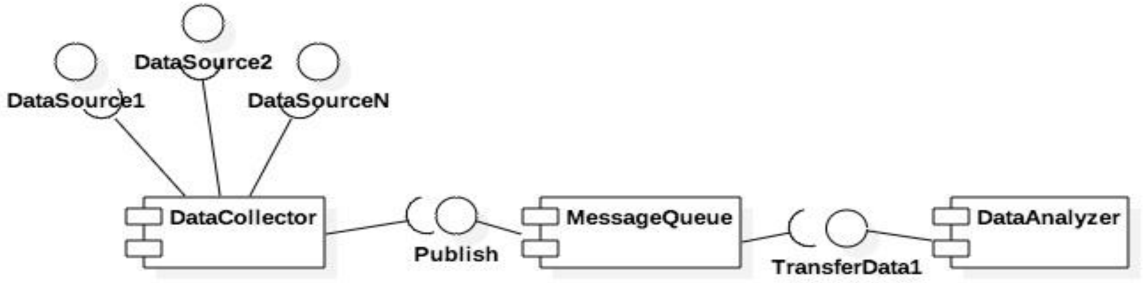
# A possible solution (cont.)

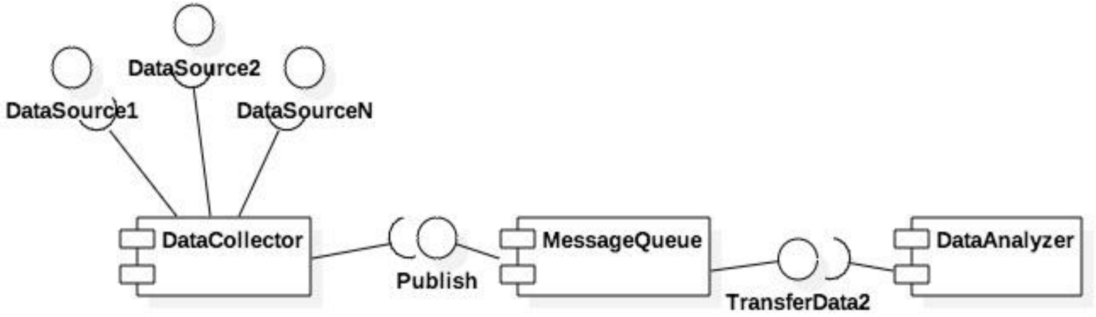
```
// We capture the different configurations through
// extensions of the Configuration
// signature above; they add the necessary constraints

sig Configuration1 extends Configuration{}
{ all disj coll1, coll2 : collectors |
        coll1.sources & coll2.sources = none }

sig Configuration2 extends Configuration{}
{ all coll : collectors | coll.sources = sources }

sig MasterDataCollector extends DataCollector {}
sig SlaveDataCollector extends DataCollector {}

sig Configuration3 extends Configuration{}
{ all coll : collectors |
   coll in (MasterDataCollector | SlaveDataCollector)
  one coll : collectors | coll in MasterDataCollector
}
```

# More on the data analysis example

(from the exam of June, 27th 2018)

- Consider the following two versions of the same system





- Q1: What is the difference between the two?

# More on the data analysis example

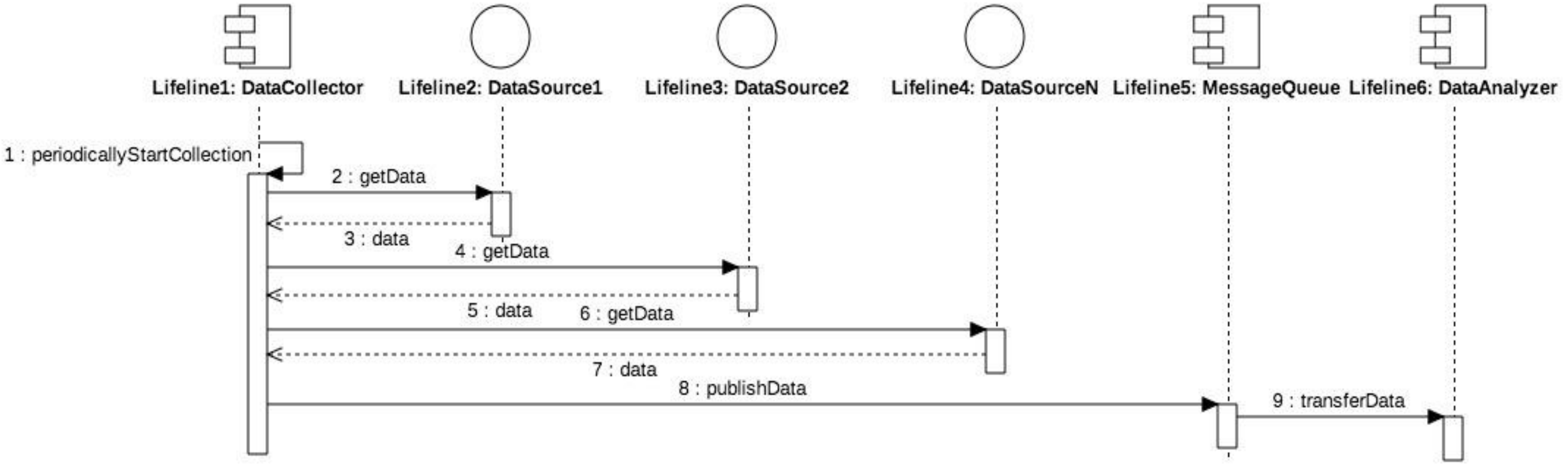
(from the exam of June, 27th 2018)

- Solution
  - In the second case, the MessageQueue does not actively push the data to the DataAnalyzer, but it offers interface TransferData2 so that the DataAnalyzer can pull data as soon as it is ready to process them. Also in this case, both a batch or a per data approach is possible. The rest of the system behaves as first one.
- Q2: Define two sequence diagrams that describe how data flow through the system in the two versions of the architecture

# More on the data analysis example

(from the exam of June, 27th 2018)

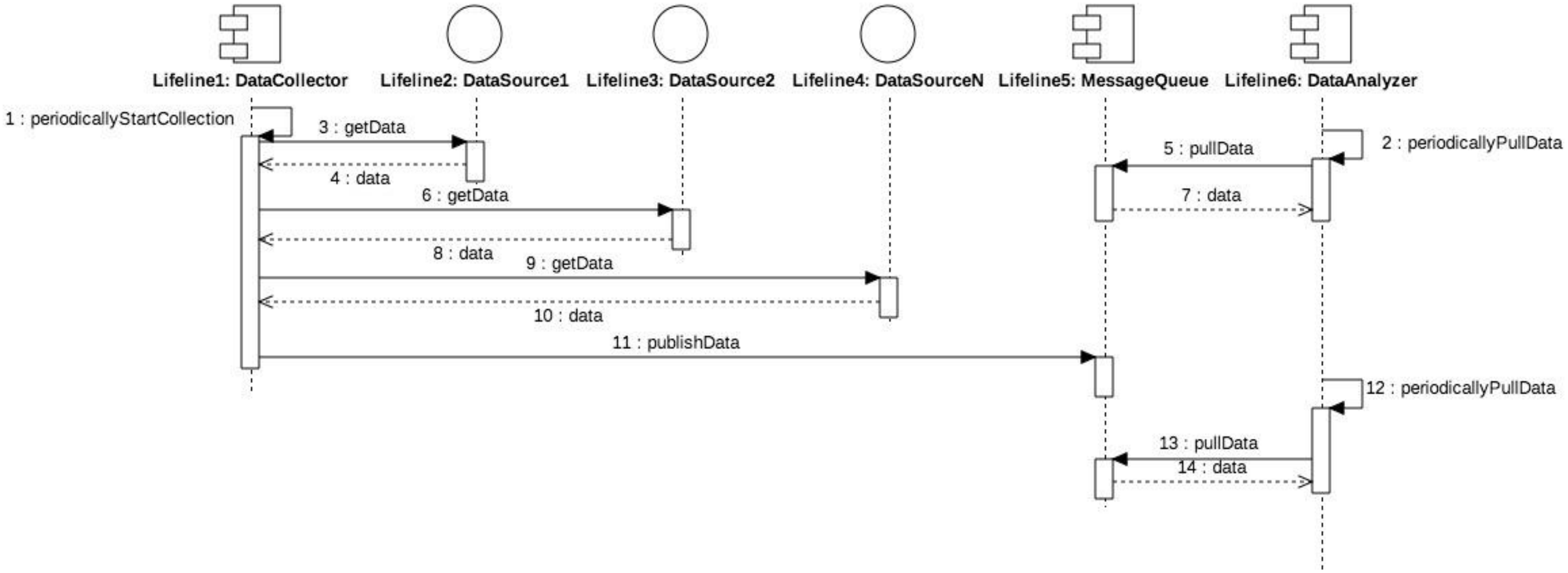
- Solution to Q2
  - Sequence diagram compatible with the first component diagram



# More on the data analysis example

(from the exam of June, 27th 2018)

- Solution to Q2
  - Sequence diagram compatible with the second component diagram



# More on the data analysis example

(from the exam of June, 27th 2018)

- Assume that the components of your system offer the following availability:
  - DataCollector: 99%
  - MessageQueue: 99.99%
  - DataAnalyzer: 99.5%
- Provide an estimation of the total availability of your system (you can provide a raw estimation of the availability without computing it completely).

# More on the data analysis example

(from the exam of June, 27th 2018)

- Data flow through the whole chain of components to be processed => series of component.
- The total availability of the system is determined by the weakest element, that is, the DataCollector.
  - $A_{Total}$ = 0.99*0.9999*0.995 = 0.985
- Assuming that you wanted to improve this total availability by exploiting replication, which component(s) would you replicate? Please provide an argument for your answer.

# More on the data analysis example

(from the exam of June, 27th 2018)

- If we parallelize the data collector adding a new replica, we can achieve the following availability:
  - $(1-(1-0.99)^2)$ *0.9999*0.995 = 0.995
- if we increase the number of DataCollector replica, we do not achieve an improvement as the weakest component becomes the DataAnalyzer.
- We can parallelize this component as well to further improve the availability of our system.

- How would such replication impact on the way the system works and is designed?

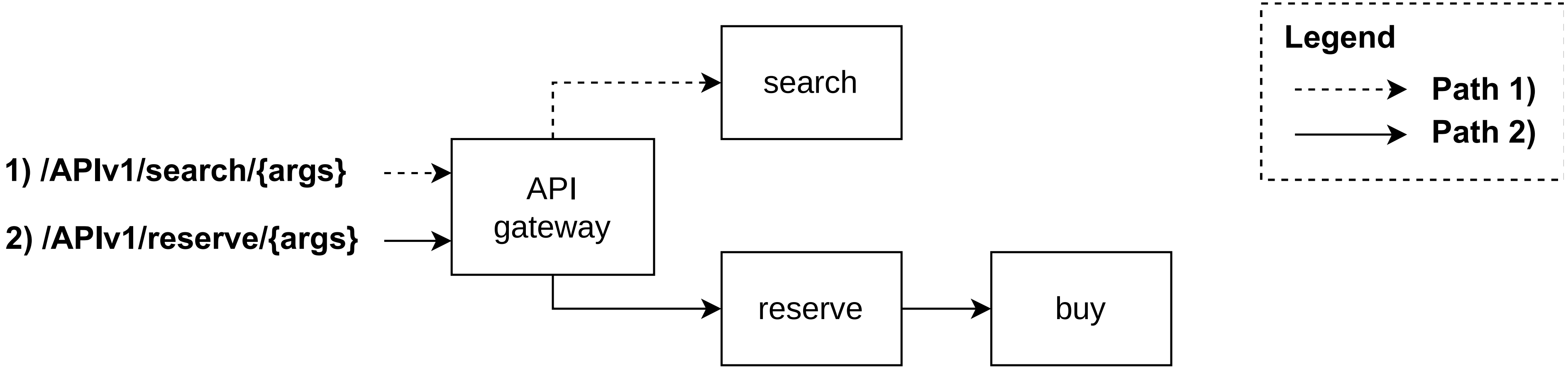
# TrainTicket

# TrainTicket (Sept 7th 2023 - WE1 exam)

- Consider a microservices application called TrainTicket composed of 3 domain microservices (search, reserve, buy) and 1 additional microservice that acts as the API gateway. TrainTicket supports two basic operations invoked using the exposed RESTFul APIs:
    1. search: /APIv1/search/{args}
    2. reserve: /APIv1/reserve/{args}
- Requests (both search and reserve) are received and then dispatched by the API gateway.

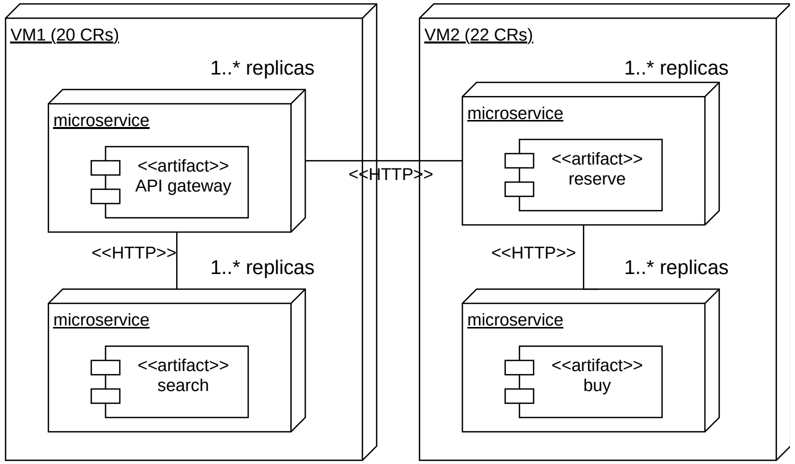
# TrainTicket (Sept 7th 2023 - WE1 exam)

- In particular, the following high-level schema shows how requests propagate from the gateway to internal microservices. Note that in this example reserve includes also the purchase of reserved items.



# TrainTicket (Sept 7th 2023 - WE1 exam)

- Microservices run in units deployed onto 2 different Virtual Machines, VM1 and VM2 as shown in the following UML deployment diagram.
- The available VMs have Computational Resources (CRs) that can be allocated to run microservices. Each VM has a maximum number of CRs and each microservice requires a certain number of CRs, according to the executed artifact. As shown in the schema, available CRs are as follows:
  - VM1: 20 CRs
  - VM2: 22 CRs



# TrainTicket (Sept 7th 2023 - WE1 exam)

- The mapping between microservices and required CRs is as follows:
  - API gateway: 2 CRs, search: 5 CRs, reserve: 4 CRs, buy: 5 CRs
- The deployment diagram shows that each microservice can be replicated to have redundant business-critical components. In the latter case, requests are directed to all the replicas rather than to an individual instance and the first answer received from a replica is returned to the caller, while the others are simply ignored. The number of replicas for each microservice shall be defined so that the following nonfunctional requirement is satisfied and the deployment constraints defined in the deployment diagram and above are fulfilled.
- R1: “Both search and reserve services exposed through API gateway shall have availability greater than or equal to 0.99”.

# TrainTicket (Sept 7th 2023 - WE1 exam)

- **Availability_1 (3 points)** Considering the constraints of the execution environment represented above, determine whether requirement R1 can be satisfied or not assuming the following availability estimates for each microservice:
  - API gateway: 0.99
  - search: 0.98
  - reserve: 0.95
  - buy: 0.91
- Justify your answer.

# TrainTicket (Sept 7th 2023 - WE1 exam)

- **Answer**: Considering the execution environment, we can derive the following inequations constraining the number of replicas:
  - Constraints extracted from environment:
    - $2x + 5y <= 20$
    - $4u + 5z <= 22$
  - Constraints extracted from requirement R1:
    - $(1 - (1 - 0.99)^x) * (1 - (1 - 0.98)^y) >= 0.99$
    - $(1 - (1 - 0.99)^x) * (1 - (1 - 0.95)^u) * (1 - (1 - 0.91)^z) >= 0.99$
- Where variables x, y, u, and z represent the number of replicas for the microservices *API gateway*, *search*, *reserve*, and *buy*, respectively.
- The requirement R1 can be satisfied since there exists a valid assignment to variables that satisfies all constraints. For instance:
  - $x = 2, y = 2, u = 3, z = 2$

# TrainTicket (Sept 7th 2023 - WE1 exam)

- **Availability_2 (2 points)** Consider the problem of resource allocation taking into account the operational profile, that is, the behavior of the users. Assume the following workload in terms of average number of concurrent users for each request:
  - search: 50 users
  - reserve: 90 users
- Assume also that for reserve, only 20% of users complete the purchase at reservation time. This means that 20% of reserve requests get through and reach the buy microservice, while 80% of them terminate the execution without calling buy.

# TrainTicket (Sept 7th 2023 - WE1 exam)

- After a preliminary analysis, we realize that availability depends on the workload according to the following new estimates:

| LOW workload: 0-60 concurrent users | | HIGH workload: 60-150 concurrent users | |
|---|---|---|---|
| **Microservice** | Availability | Microservice | Availability |
| **API gateway** | 0.99 | API gateway | 0.98 |
| **search** | 0.98 | search | 0.95 |
| **reserve** | 0.95 | reserve | 0.93 |
| **buy** | 0.91 | buy | 0.90 |

- Does the execution environment have enough computational resources to support the workload defined above still fulfilling requirement R1 and the defined constraints? Justify your answer.

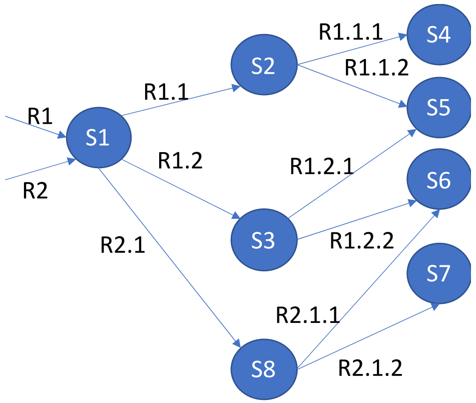
# TrainTicket (Sept 7th 2023 - WE1 exam)

- The expected workload for each microservice is as follows:
  - API gateway: 140 users (HIGH)
  - search: 50 users (LOW)
  - reserve: 90 users (HIGH)
  - buy: 18 users (LOW)
- The constraints extracted from requirement R1 become as follows:
  - $(1 - (1 - 0.98)^x) * (1 - (1 - 0.98)^y) >= 0.99$
  - $(1 - (1 - 0.98)^x) * (1 - (1 - 0.93)^u) * (1 - (1 - 0.91)^z) >= 0.99$
- An optimal resource allocation is represented by the assignment $x = 2$, $y = 2$, $u = 3$, $z = 2$ that is again feasible according to environment constraints.

# Microservices and availability

# Sept 7th, 2022 exam

- Consider the microservice-based architecture shown in the figure below. The architecture is organized in eight stateless microservices that collaborate to fulfill requests R1 and R2. S1 is the front-end service that receives both requests. The fulfillment of request R1 requires the interaction with services S2 and S3 (through sub-requests R1.1 and R1.2, respectively), which, in turn, need to interact with other services. In particular, S2 interacts with S4 and S5 and S3 with S5 and S6.
- The fulfillment of R2 requires that S1 interacts with S8 which, in turn, interacts with S6 and S7.



| Service | Avail |
|---|---|
| S1 | 0.99 |
| S2 | 0.9 |
| S3 | 0.9 |
| S4 | 0.95 |
| S5 | 0.999 |
| S6 | 0.99 |
| S7 | 0.99 |
| S8 | 0.95 |

# Sept 7th, 2022 exam

- Assuming that the availability of services S1-S8 is the one reported in the table above, what is the availability of the system when answering to request R1?
- If each of the services S1-S8 is duplicated, what is the new value of the availability computed at point 1?

# Solution (1/2)

- Note that, regardless of the way the interaction between the MessageQueue and the DataAnalyzer works, data have to flow through the whole chain of components to be processed. This implies that we can model the system as a series of component.
- The total availability of the system is determined by the weakest element, that is, the DataCollector.
  - ATotal = 0.99*0.9999*0.995 = 0.985
- If we parallelize the data collector adding a new replica, we can achieve the following availability:
  - (1-(1-0.99)2) *0.9999*0.995 = 0.995
- At this point, even if we increase the number of DataCollector replica, we do not achieve an improvement as the weakest component becomes the DataAnalyzer. We can parallelize this component as well to further improve the availability of our system.

# Solution (2/2)

- Let’s consider the impact of the DataCollector parallelization on the rest of the system. If both replicas acquire information from the same sources in order to guarantee that all data are offered to the rest of the system, then the other components will see all data duplicated and will have to be developed considering this situation. For instance, the MessageQueue could discard all duplicates. Another aspect to be considered is that both DataSources and MessageQueue have to implement mutual exclusion mechanisms that ensure the communication between them and the two DataCollector replicas does not raise concurrency issues. Another option could be that only one DataCollector replica at a time is available and the other is activated only when needed (for instance, if the first one does not send feedback within a certain timeout).