ARCHITECTURE REPORT

Software Architecture

Last updated on 26/02/22

João **BRILHANTE**

Quentin **LAROSE**

Ludovic **MARTI**

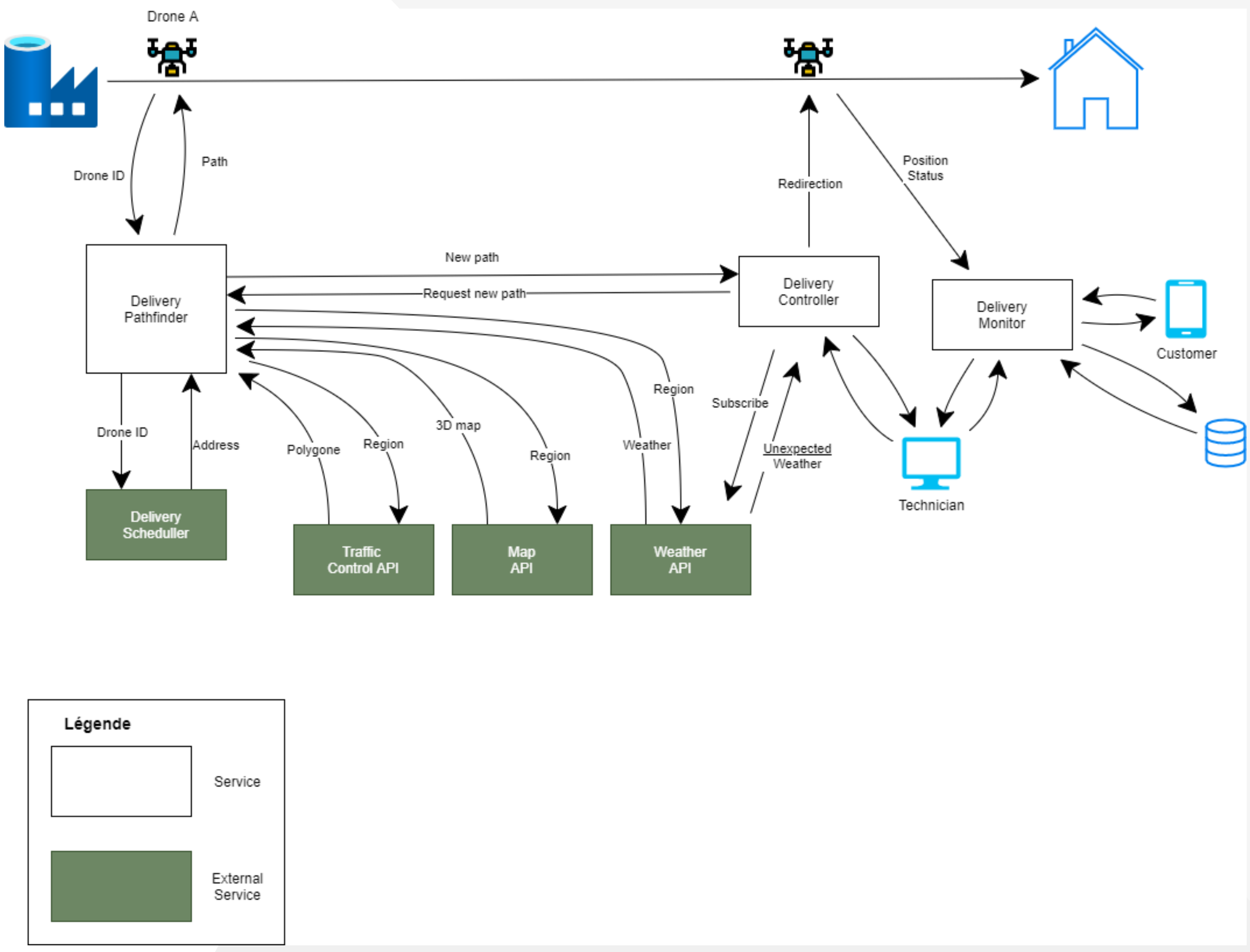
Enzo **BRIZIARELLI**

Charly **DUCROCQ**

I. Table of contents

First assignment: Scope definition, main scenarios (28/09/21)

First architecture



Delivery Pathfinder

The delivery pathfinder allows a drone to retrieve its flight plan before leaving the warehouse. It computes the flight plan of each drone using a 3D mapping service, an air traffic control service, and a weather service.

Delivery Controller

The delivery controller can redirect the drone in case of a problem during the delivery. It also regularly consults the weather service to make sure the delivery is going well. This service also allows a technician to take manual control of the drone.

Delivery Monitor

The delivery monitor allows drones to regularly report their position and other useful metrics used for tracking the order. This service allows customers to view the status of the delivery, as well as allowing technicians to track deliveries in real time and quickly identify problems.

Delivery Scheduler

The delivery scheduler is a service of the project out of our scope. It will provide a schedule of the drone’s allocations.

Traffic Control API

This external API provides a set of 3D airline points where a drone is allowed or prohibited to fly.

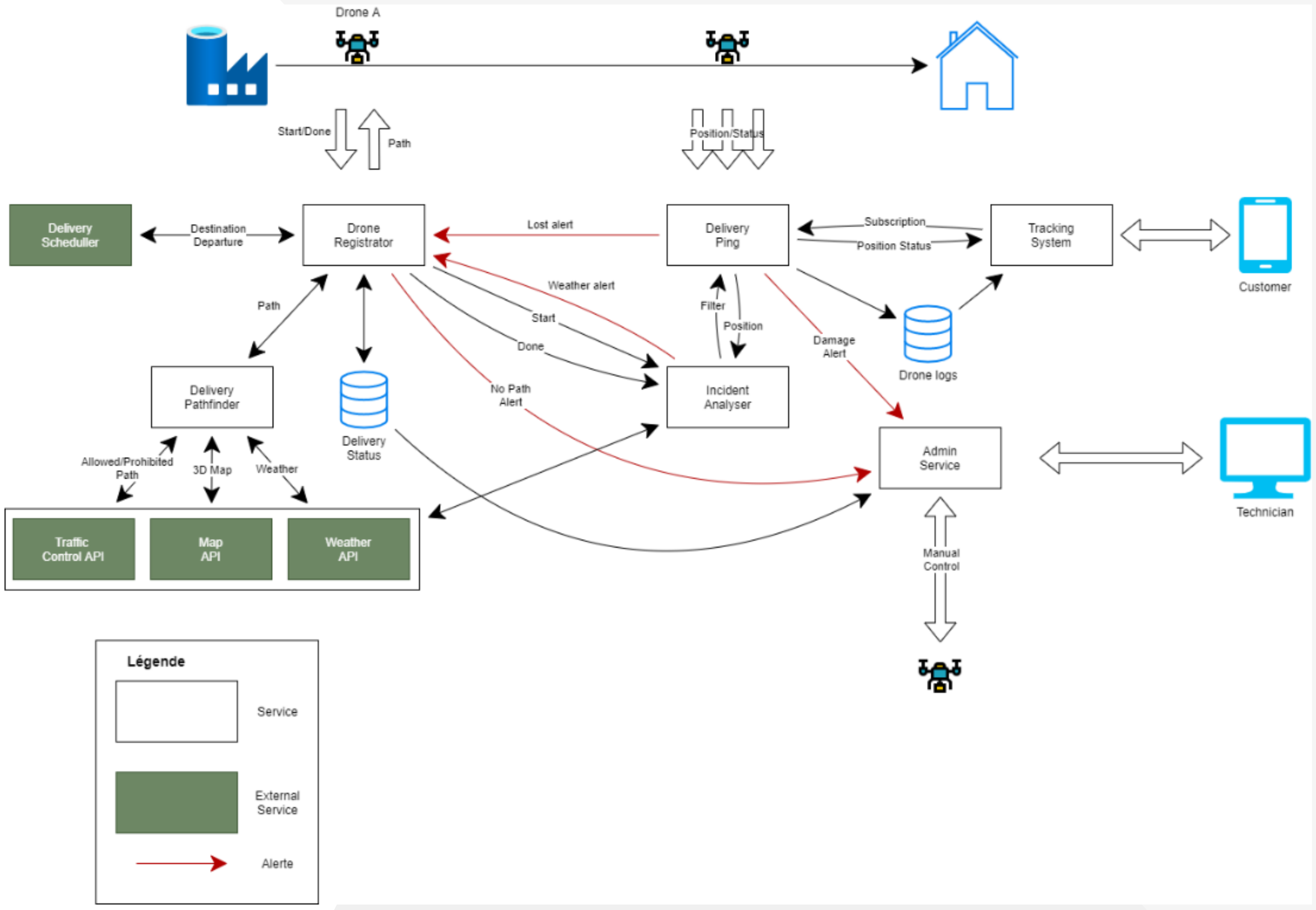
Map API

This external API provides map information. It will provide us a 3D view of a city, allowing us to avoid any registered building and obstacle.

Weather API

To avoid bad storm or rain, we’ll need to know the weather. The weather API will provide us weather information and allow us to subscribe to be alerted by any unexpected weather changes.

Second architecture



Following our reflections, we produced 2 diagrams representing our architecture. A first one that seemed good but with some small inconsistencies and a second one where we redesigned the architecture to overcome our previous problems. We also reconsidered the responsibility of each of our components.

The flaws corrected by our new architecture are the following:

- The *Delivery Monitor* component was a SPOF (Single Point of Failure) that was a critical issue. Indeed, this component was receiving calls from drones, customers (to view the drones), technicians (to see all the drones in real time), etc...
- The *Delivery Pathfinder* component was also a SPOF, overloaded with responsibilities that were not intended for him. For instance, it was in charge of managing drone startup and shutdown, although its initial job is to calculate the path of the drones. So, we split our *Delivery Pathfinder* component into 2 other components: *Drone Registry* (to manage startup and shutdown) and Delivery Scheduler.
- Moreover, in our previous architecture our drones never signaled the end of their deliveries. This feature was hidden by our previous *Delivery Pathfinder* component

- The *Delivery Controller* component also had too many responsibilities. It was communicating with the drones to know their positions, send them redirects, communicate with the Weather API and finally manage incidents.

Personas

- **Brian:** Customer living in a building.
  Brian wants the drone to find his place and to make the delivery through his window.
- **Emilie:** Customer living in a countryside house.
  Emilie wants to know the progress of her delivery at any time and the package to be left in front of her door.
- **Zoe:** Drone maintenance technician.
  Zoe wants to know every drone position and status at any time to be able to intervene quickly in case of an issue. She also wants to be able to manually control the drone if needed.
  She also wants the drones to avoid risky weather and not fly in the worst cases.

User Stories

1. **As a** customer,
   I want my delivery drone to follow a path that considers the landscape and the buildings in the city,
   So it should reach my place. **[MUST]**

2. **As a** technician,
   I want to visualize the real time position and followed path of many drones at a time,
   So I can ensure they follow a right path without conflicts. **[MUST]**

3. **As a** technician,
   I want the delivery drones to get a new path to follow when they are deflected (e.g.: by wind) or lost,
   So they can reach their goal destination. **[SHOULD]**

4. **As a** customer,
   I want a graphical view to track the drone in charge of my package,
   So I can make sure my delivery is going well. **[SHOULD]**

5. **As a** technician,
   I want to be notified the last position of a crashed drone,
   So I can quickly go retrieve and repair it. **[SHOULD]**

6. **As a** customer,
   I want my delivery drone to update its path when it is suddenly raining on its route,
   So my package (and the drone) is not damaged and come safely at my place or go back to the warehouse. **[COULD]**

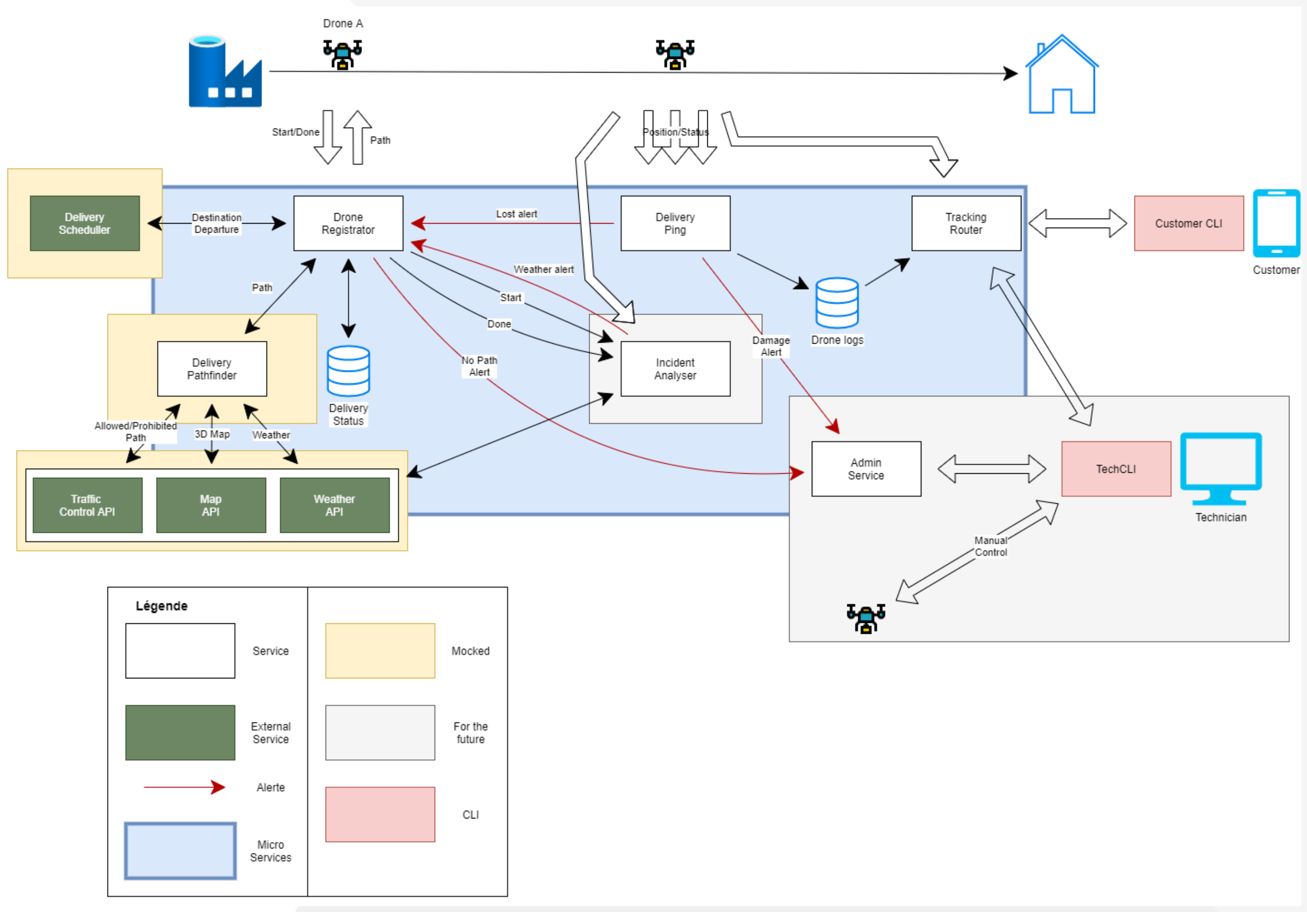
7. **As a** technician,
   I want to be able to take control of a stuck drone,
   So I can make it go over the obstacle. **[COULD]**

MVP Hypotheses

- All drones are identical.
- The drones have embedded programs to avoid small obstacles and follows a given path (birds, trees, ...).
- Only one warehouse.
- The drones are ready to go with the orders.
- A drone only takes care of one delivery at a time.
- Drone flights are authorized in most aerial spaces.
- We have different APIs:
    - Traffic Control: to avoid aerial spaces with traffic.
    - Weather: to know the weather around a geographical point at any time.
    - 3D map: to match landforms and avoid known buildings.

Second assignment: Component diagram, roadmap (05/10/21)

New choice of architecture



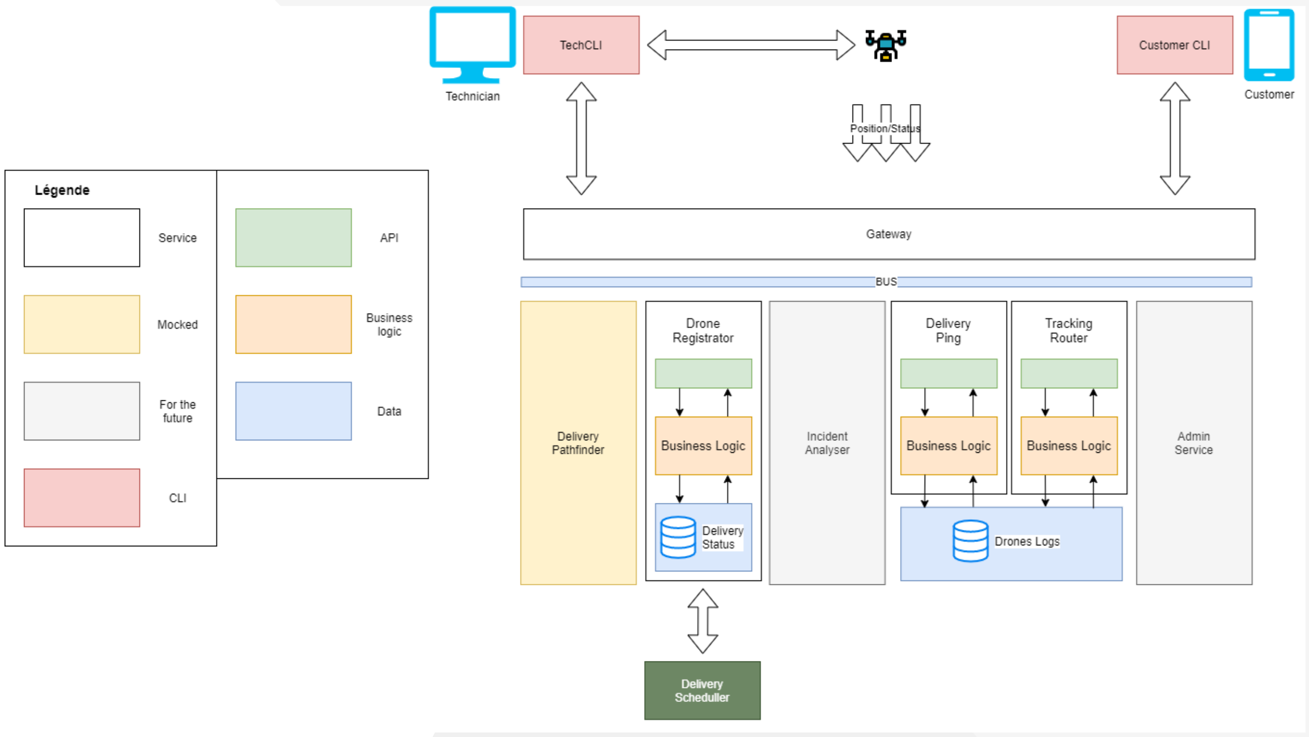
Since our last architecture, we have added the consideration of CLI and the interactions they will have with our services.

For example, the technician's CLI will communicate directly with the drone, to be able to control it after having requested access to the *Admin Service*.

Then, we thought about what type of architecture to implement between a monolith, services or microservices.

Finally, we decided what will be our priorities with a MuSCoW breakdown and our technologies need, in order to define a POC (Proof of Concept).

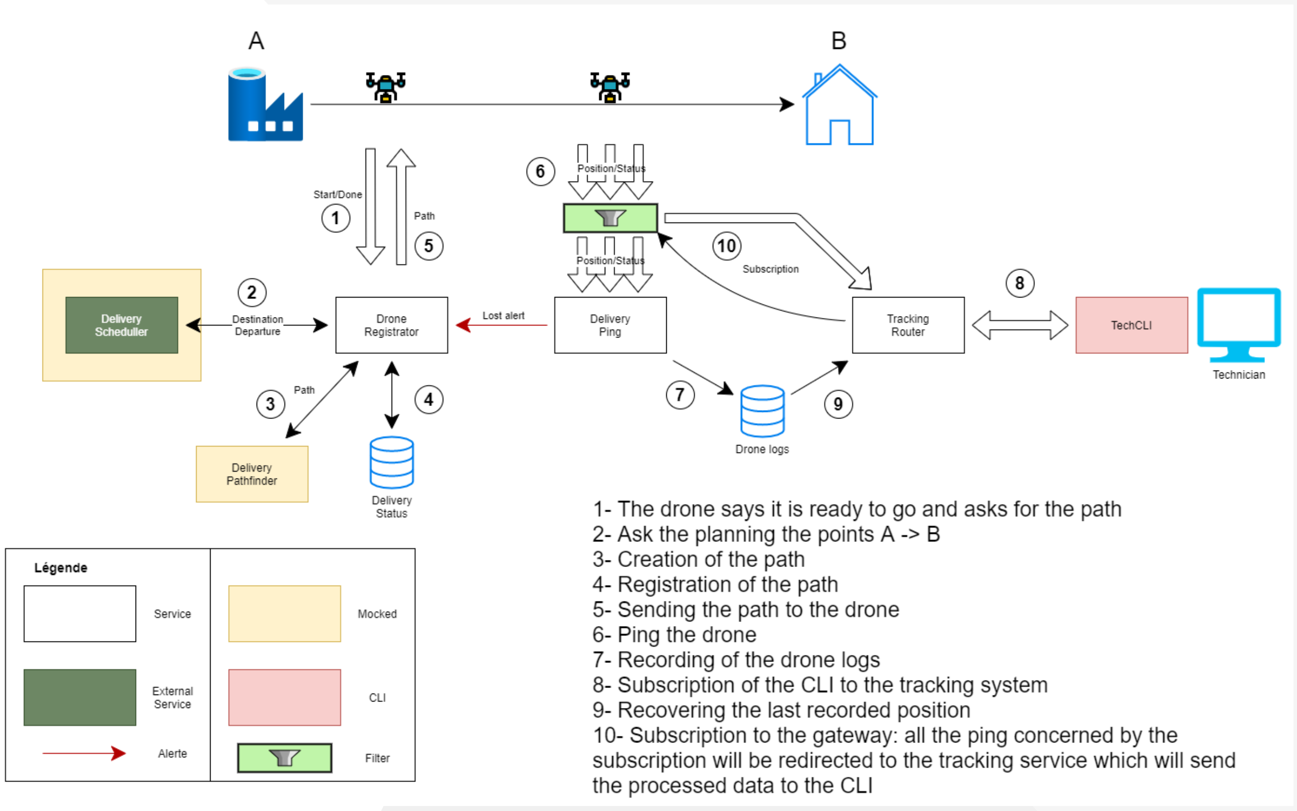
A view into micro-services system



First of all, we are thinking of using a micro-services architecture oriented around a message bus. Indeed, this seems to be an interesting approach from the point of view of alerts that several of our services must send/receive. A bus would simply allow us to subscribe to them. In the same way, our *Delivery Ping* service is currently in charge of redirecting the flow of information received from the drones to the *Tracking Router* service, which is not optimal given that with a bus, the ping message would only be sent once.

Secondly, the fact of splitting into micro-services could make it possible to create several instances of the same service in order to process tasks simultaneously to lighten the load and reduce response times. This is the case of the *Delivery Pathfinder*, whose algorithm is quite substantial. In short, the very large number of transactions and alerts sent justifies the use of a message bus and microservices.

POC: Proof of Concept



1- The drone says it is ready to go and asks for the path
2- Ask the planning the points A -> B
3- Creation of the path
4- Registration of the path
5- Sending the path to the drone
6- Ping the drone
7- Recording of the drone logs
8- Subscription of the CLI to the tracking system
9- Recovering the last recorded position
10- Subscription to the gateway: all the ping concerned by the subscription will be redirected to the tracking service which will send the processed data to the CLI

For the POC of the project, we moved the *Delivery Pathfinder* into a "black box" by making it a mock, so we avoid working on complex algorithms that doesn’t demonstrate functionality. We also decided to not focus our efforts on the *Incident Analyzer*, the *Admin Service* or the *Customer CLI* because they are not a priority in our user stories and represents a lot of work.

We also decided to keep an alert message between the *Delivery Ping* and the *Drone Registry* in order to demonstrate the lost drone alert functionality and our ability to handle this kind of messages that we might have to implement again in the future.

Thus, we can say that our "MUST" scenarios will be covered and that we will be able to cover most others by extension.

Roadmap

- 08 October: Backlog and "hello world" services with dockers and CI
    - Services: just a skeleton but with minimal implementation
    - Backlog: defining the issues, definition of done/ready, ...
    - CI: choosing a techno and set it up (Jenkins?)

    Therefore, we should have a project well prepared and ready to be developed.

- 15 October: Functional services with direct communication
    - Services communicating with each other
    - MUST (Moscow) scenarios covered

    In order to have a simple functional system and to start thinking about the message bus implementation.

- 22 October: Services communication through a message bus with a gateway
    - Techno chosen and set up (Kafka? RabbitMQ?)
    - Message bus and gateway implementation started

    So, we match the defined scop and constraints.

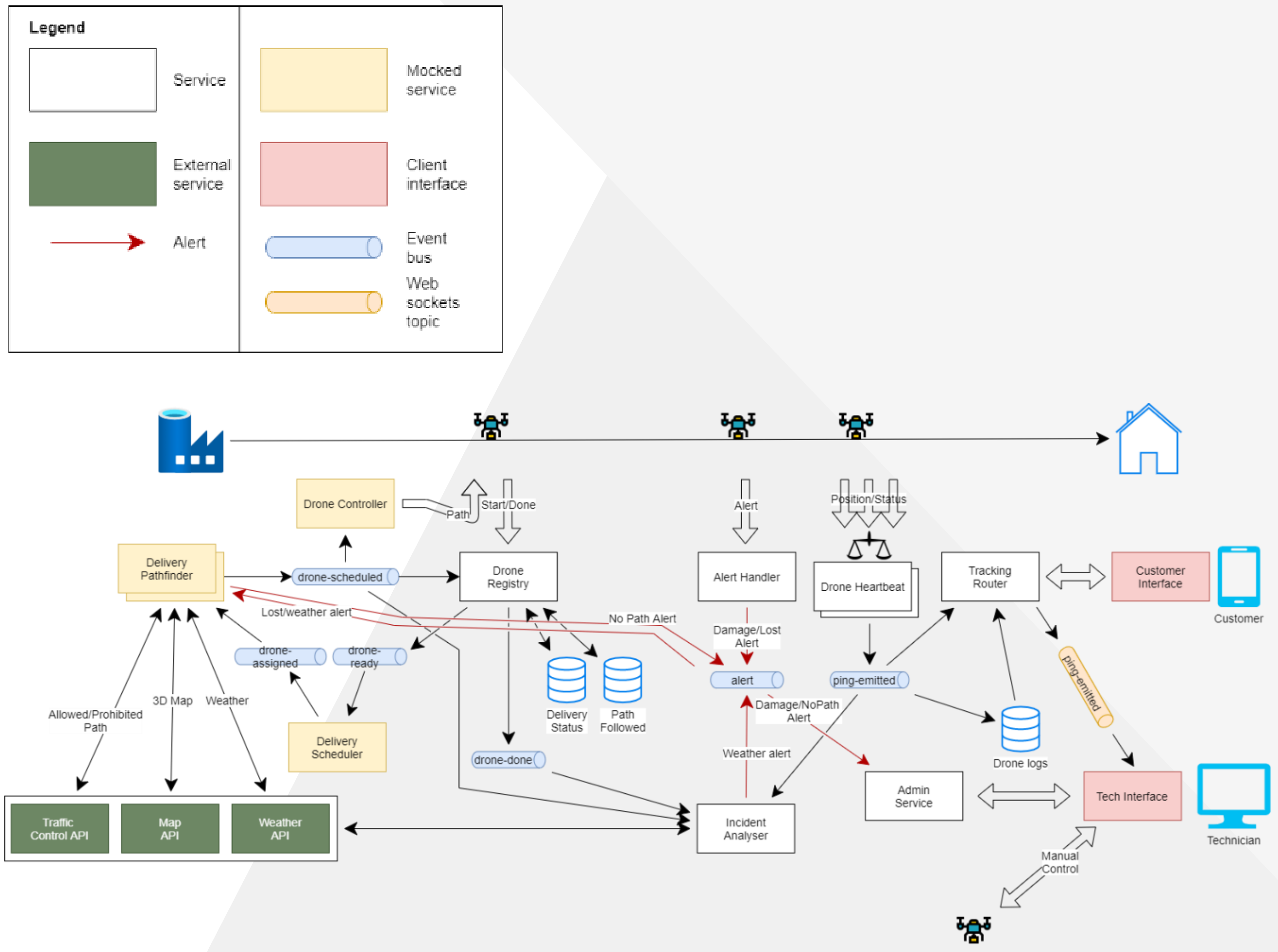
- 29 October: MVP working with message bus, gateway and simple front ends
    - Finishing the message bus and gateway implementation
    - Make simple front ends (CLIs?)

    To prepare the POC.

- 05 November: POC ready!
    - Demonstration ready
    - Every scenario covered and tested

Post-assignments: Event driven implementation (05/11/21)

New choice of architecture



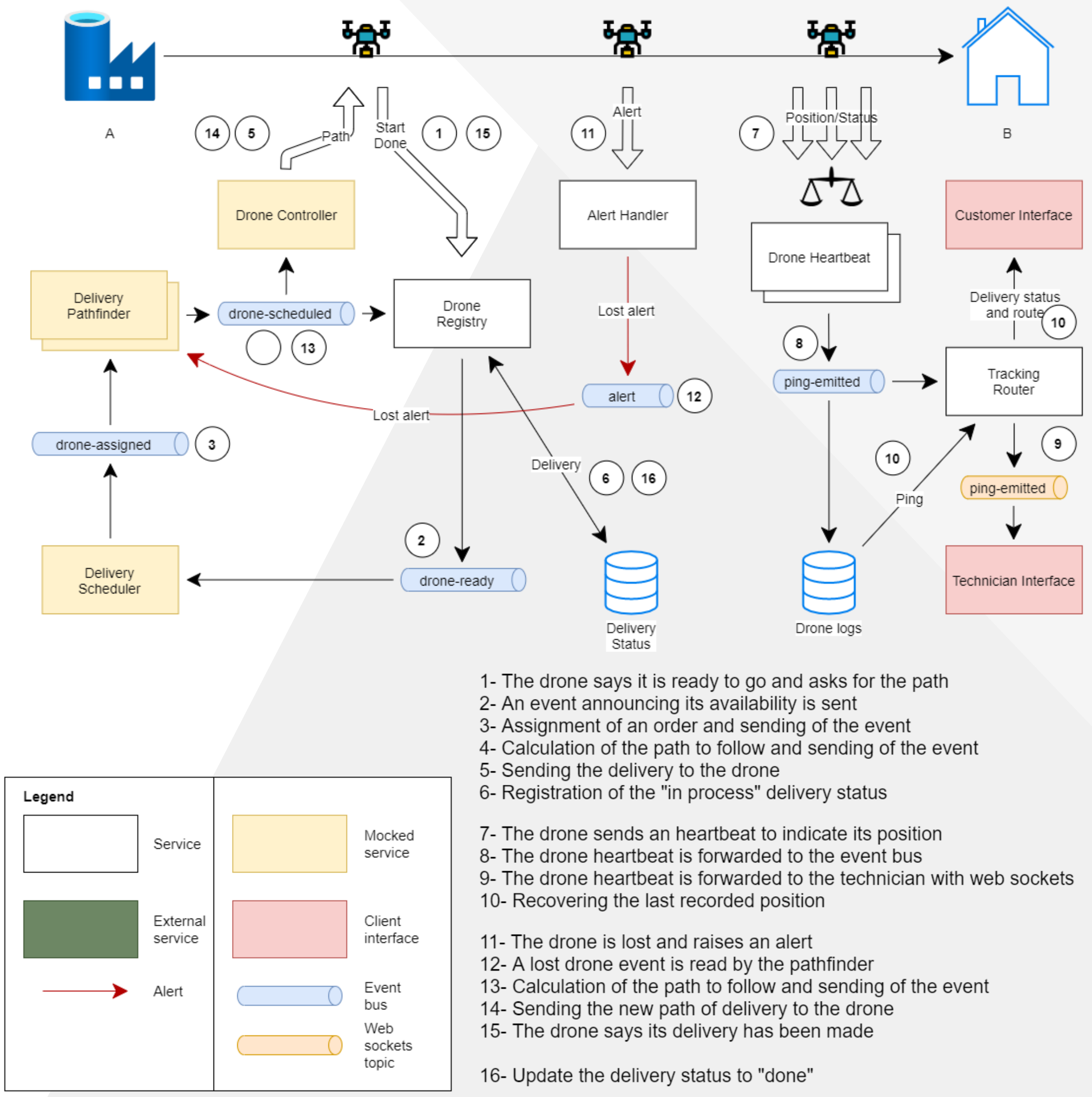
Our choice to make an event-driven based architecture led us to include our bus topics in our diagram. In doing so, we realized that the "view into micro-services system" was not well represented. Indeed, the bus is not placed between the entry point and the services as we thought. It is more coherent to put it between the concerned services. After this reflection, we have changed the diagram above consequently.

Our new architecture includes different message buses (or topics). These buses make the architecture easier to read and understand. Indeed, each service is now independent of the others, and several subscribe to the same events, which avoids the multiplication of data transfers between several services.

We have also chosen to separate the processing of errors from the pings by adding the Alert Handler service. The drone will communicate to this service in case of error (Lost, Failure, etc..).

This event-driven architecture led us to add a service responsible for contacting the drones: the *Drone Controller* service. Indeed, before the implementation of buses, we used to communicate the path to the drone in response to its request. However, this is no longer the case here.

POC: bus topics view



1- The drone says it is ready to go and asks for the path
2- An event announcing its availability is sent
3- Assignment of an order and sending of the event
4- Calculation of the path to follow and sending of the event
5- Sending the delivery to the drone
6- Registration of the "in process" delivery status

7- The drone sends an heartbeat to indicate its position
8- The drone heartbeat is forwarded to the event bus
9- The drone heartbeat is forwarded to the technician with web sockets
10- Recovering the last recorded position

11- The drone is lost and raises an alert
12- A lost drone event is read by the pathfinder
13- Calculation of the path to follow and sending of the event
14- Sending the new path of delivery to the drone
15- The drone says its delivery has been made

16- Update the delivery status to "done"

Finally, here's the final architecture representing our POC.

Here as well, the only main difference is the addition of event buses. Interactions with the *Drone Registry* no longer happen sequentially but circularly through the two mocked services. In addition, the *Delivery Pathfinder* can now subscribe directly to error messages requesting a new path calculation.

Also, when implementing the user interface used by the technician, we set up a web socket allowing the dynamic update of the drone positions.

Technical explanations

By reviewing the subject and its key points, we have converged towards a greater consideration of Resilience, Performance and Scalability:

- **Resilience**: We chose to use Kafka to handle the events received and to allow multiple services to subscribe to a drone ping. Moreover, we can store the pings from drones when the service crashes and retrieve the missed data (positions and status) sent while the service was down. We need to know if a delivery has been completed, or draw the full path with accuracy, even if an error occurred. In addition, if a drone crashes when the *Delivery Ping* service is down, we need to be able to determine its last position in order to find it.
- **Performance**: One of the main points of our architecture is the path computing which is a CPU-intensive task. We must be able to compute more than one path at a time to handle many deliveries. Therefore, we will be using multiple instances of our Pathfinder service.
- **Scalability**: The *Delivery Pathfinder* and *Delivery Ping* services must be stateless to allow the creation of multiple instances of themselves and handle strong affluence. The event-driven architecture will be allowing messages to be consumed only once by a group of consumers.

Transvers scenarios

Here are the new through scenarios we have chosen to show to demonstrate our architectural choices:

Given a set of orders waiting to be delivered
When a drone indicates that it is ready to deliver an order
Then the scheduler assigns the drone to an order,
And the pathfinder computes the path between the source and destination,
And the registry sends the drone its path and schedule.

Given a few drones assigned to deliveries and flying to their destinations,
When the technician wants to see the status of the drones,
Then the tracking router sends him the latest locations.

Given a drone assigned to a delivery and flying to its destination,
When the customer wants to see its delivery status,
Then the tracking router sends him the drone's current location.

Given a drone assigned to a delivery and flying to its destination,
When the drone got lost,
Then the alert handler receives this information

And the pathfinder computes a new path between the drone position and its destination,
And the drone controller sends the drone its new path.

Given a drone assigned to a delivery and flying to its destination,
When the drone heartbeat service crashes and is restarted,
Then the customer can see the last location of the drone before the service went down,
And the technician can see the latest location.

Second Semester: New features (25/02/22)

Introduction

For the continuation of our software architecture course, new features are added to confront our current architecture with new requirements.

These features are the following:

- Real-time tracking of "critical" deliveries, even if they need to cross areas partially covered by the network (poor network coverage/quality).
- Real-time feedback of the tracking needs to be provided to an external air-management authority for all urban areas.

Personas

- **Martine:** CEO of the drone company
  When a drone crosses an urban area, Martine must provide its tracking information to the local authorities to comply with the law.

User Stories

8. **As a** customer,
   I want to know the position of my delivery even when it crosses an area with poor network coverage,
   In order to follow the progress of my packages in all circumstances. **[MUST]**

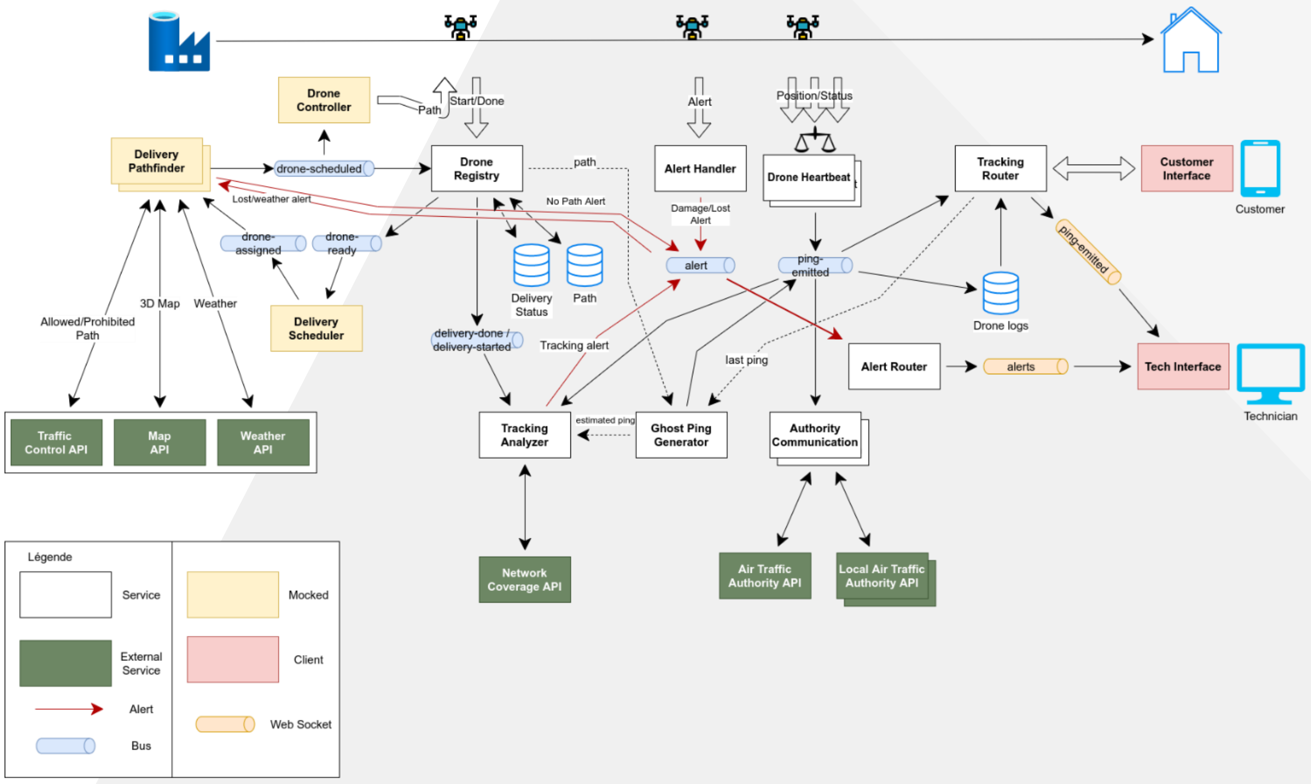
9. **As a** technician,
   I want to know the position of every drone even when they cross an area with poor network coverage,
   In order to have an accurate view of the position of the drones at any time. **[MUST]**

10. **As the** CEO of the drone company,
I want the local air management authority to know about every drone flying in their area,
In order for my company to comply with the law. **[MUST]**

Hypotheses

- We have access to three new external APIs:
    - **Network Coverage API**: to be aware of areas with poor network coverage.
    - **Air Traffic Management API**: to know all Local Air Traffic Management APIs.
    - **Local Air Traffic Management API** (one for each locality): to inform the local authority of drone traffic in its area.

New architecture



Technical explanations

Among the new constraints we must face, our architecture is already able to respond to one of them: providing real-time tracking of drones. Indeed, we had already assumed that the drones were sending us their position at regular intervals. These position reports are then sent to a front-end via a web-socket allowing for a near real-time tracking.

The next issue though, tracking the drones in areas with little to no network coverage, requires updates. So, we have divided the *Incident Analyser* service into two sub-services (*Weather Analyzer* and *Tracking Analyzer*) to better distribute responsibility for incident detection. The *Tracking Analyser* service will oversee the heartbeats of the drones in the process of delivery and will have to interpret and emit their position if none are received. After a certain number of missing heartbeats, it will reach the *Network Coverage API* to find out if the drone is currently flying over an area with poor network coverage. If this is the case, the missing heartbeat counter will simply be reset since it is normal for the drone to fail to contact us in the current area. If this is not the case, an alert will be raised to indicate that the drone is lost. This service will also be notified when a drone delivery is done to stop tracking it.

For the second issue, we need to be able to provide every drone tracking in urban areas to a local authority. For this purpose, we created a service, named *Authority Communication,* that will simply scan the latest known positions of the drones and will transmit them to the corresponding local authority when flying in urban areas. Our service will communicate with the external services *Air Traffic Management API* and *Local Air Traffic Management API*.

New services

Authority Communication

The *Authority Communication* service subscribes to the drone heartbeats sent in the bus. Each time the service receives a drone heartbeat, it contacts the external *Air Traffic Authority* service to find out if the drone is flying over an urban area. If it is the case, the external *Air Traffic Authority* service indicates the URL of the *Local Air Traffic Authority* server corresponding to the overflown urban area. We are aware that this service can be under heavy load since it must process all drone heartbeats, therefore we designed this service to be stateless, so it can be easily scaled horizontally. It can be achieved with several instances of this service, as well as increasing the partitions for the *ping-emitted* Kafka topic.

Tracking Analyser

The *Tracking Analyser* service also subscribes to the drone heartbeats sent in the bus. This service verifies that each drone currently doing their deliveries sends a drone heartbeat on a regular basis. When a drone fails to send a drone heartbeat while delivering, this service invokes the *Ghost Ping Generator* service to interpolate the position of the drone. This ensures that we always have a rough idea of the position of the company's drones. After a certain number of drone heartbeats not received from the drone, this service will call an external API (we mocked it in our project: *Network Coverage API* service) in order to get information about the current network coverage level of the drone. If the drone is currently flying over an area with poor network coverage, the service considers the lack of heartbeats as normal, and so resets the cycle of generating ghost-heartbeats and waiting for drone's real-heartbeats. However, if the drone is currently flying over an area with good network coverage, the drone's silence is worrying, so the service issues an alert so the technicians are kept informed on the situation in the field.

Ghost Ping Generator

The *Ghost Ping Generator* works with the *Drone Registry* and the *Tracking Analyser*. In the first place, the *Ghost Ping Generator* is called by the *Tracking Analyser* to provide a simulation of the possible current position of a given drone: a ghost-heartbeat (a.k.a. ghost-ping). To achieve it, this service will call the *Drone Registry* to get the path of the drone, then it will use the elapsed time (the amount of time since a drone stopped emitting) and the path to determine the current position of the drone. Finally, the determined position is emitted through the *ping-emitted* topic as a ghost-ping and can be used to simulate the position of the drone in the client, etc. On the long-term perspective, this service will be scaled horizontally by multiplying its number of instances and adding a load-balancer to divide the load fairly.

Alert Router

The *Alert Router* service is quite simple, it subscribes to the alerts issued in the bus and transmits them to the connected clients through a web socket. This allows technicians to review alerts from the drones in near real time. Since this service only retransmits messages, there is no need to scale it, however it may be useful to deploy this service with enough resources to avoid alerts being transmitted with latency.

Issues encountered

The biggest issue we faced throughout the whole project was Kafka. Indeed, we had a lot of connection problems between our services and the Kafka broker. Moreover, randomly our services could get disconnected from the topics, for no apparent reason. The image we used for almost the whole project was Bitnami's Kafka image, although while discussing other groups, we noticed that our image could reasonably be the source of the previous problems. That's why we decided to face this issue by replacing our previous Bitnami's image with the Confluent's image. It turns out that this image was indeed the root cause of these problems, and now it works as we expected it to work in the first place, we no longer face troublesome issues like services randomly disconnecting from Kafka.

Limits and Future perspectives

Weather Analyser

One of our desired features from the start was the possibility to reroute drones out of bad weather zones to avoid crashing them and losing packages. Either directly during the path computation, if a "danger-zone" is already known, or directly while flying in case of unforeseen circumstances. The bad weather zones are identified with a call to an external API to provides such information, such as "Météo France".

We started the implementation of this service early during the second semester, however we had to quickly put it asides because our 2 main features were to be prioritized. We had plans to continue the implementation if we had extra time to spare, which unfortunately we hadn't.

Tracking Analyser

Although this service is crucial to our ghost-ping feature, it is stateful, so it's impossible to scale up. It represents a flaw in our architecture because we currently cannot answer to a heavy-load requirement on this feature, leading to added latency on the tracking. Fortunately, thanks to the bus the resilience aspect of this feature is still present.

Conclusion

To sum up our experience, we honestly learned a lot throughout this project. In particular, we learned that it is important to have a quality validation process for project dependencies. Indeed, in the case of the Kafka image, we should have understood that the source of the connection issues was the Kafka Docker image that we were using. To avoid this kind of issues, we should first test the Docker images used by the project in a POC and keep the Docker images we use in a private Docker repository. This would allow us to ensure the quality of the images used by the entire team.

Moreover, we learned about the pros and cons of an event-driven and microservices architecture. In our case, we think it was the best thing to do both in terms of the need for real time information and the organization of the team. However, this type of architecture inevitably comes with some testability and operation issues. Indeed, we found that it was more complex to set up solid integration tests compared to a monolithic architecture. Moreover, our solution would definitely require a good team for the operation. Indeed, whether it is for the microservices or for the Kafka brokers, there are a lot of elements to configure and monitor when deploying our solution.