Systemy rozproszone

Agentowy system monitorowania ruchu lotniczego

Skład zespołu:

Michał Motyl, numer albumu 401943

Joanna Nużka, numer albumu 400561

Kamil Pieprzycki, numer albumu 402037

Spis treści

1. Założenia projektu

Celem projektu jest stworzenie systemu symulującego monitorowanie ruchu lotniczego. System posiada agentów, którzy udostępniają informacje o ruchu lotniczym na kontrolowanym przez siebie terenie. Każdy z nich posiada dane o samolotach na swoim obszarze oraz ich lotniskach docelowych. Jeśli samolot przelatuje z jednego obszaru do drugiego, agenci przekazują sobie odpowiednie informacje. Każdy z nich może komunikować się z agentami ze swojego otoczenia w sposób współbieżny. Sumarycznie wszyscy agenci posiadają informację o całym ruchu lotniczym. Celem jest jego monitorowanie i wizualizacja.

Samoloty mają zdefiniowane trasy lotu między lotniskami. W przypadku zbyt dużej zajętości danego sektora mogą jednak zostać skierowane na trasę alternatywną lub otrzymać polecenie krążenia nad bieżącym obszarem. Każdy samolot porusza się z określoną, stałą prędkością.

2. Scenariusze

2.1. Opis

Każdy agent ma własny system GPS pozwalający mu namierzać samoloty znajdujące się na jego obszarze. Może także komunikować się z sąsiednimi agentami. Podejmuje on następujące działania w standardowych sytuacjach:

- odbiór informacji od sąsiada o tym, że samolot znajdzie się na jego obszarze i monitorowanie tego samolotu;
- namierzenie, w stronę którego sąsiada przemieszcza się samolot opuszczający jego obszar i przekazanie tej informacji;
- obsługa wyjątkowych przypadków i awarii w zależności od sytuacji (opisane niżej).

Agenci mogą również komunikować się z samolotami będącymi na ich obszarze, aby odpytać je o szczegóły lotu czy przekazać zlecenia krążenia nad swoim sektorem lub wybór innego w przypadku zbyt dużego zajęcia kolejnego planowanego. Podczas tworzenia wizualizacji system odpytuje agentów o znajdujące się w ich obszarze samoloty. Sprawdza też, czy wystąpiły jakieś alarmy.

Awaria może wystąpić w kilku przypadkach, które należy rozwiązywać w różny sposób:

- Agent dostał informację planowanym przylocie samolotu, ale samolot nie przyleciał i nie dostał odpowiedzialności za samolot:
    - należy odpytać lotnisko, czy samolot faktycznie wyleciał. Jeśli nie – poprzedni agent wysłał błędną informację – ostrzeżenie, że jest problem z agentem i likwidacja samolotu.
    - Jeśli samolot wyleciał z lotniska – próba jego znalezienia poprzez odpytanie sąsiadów sektora poprzedniego:
        - jeśli samolot zostanie znaleziony – zgłoszenie błędu trasy samolotu, komunikacja z samolotem i skierowanie go na poprawną trasę;
        - jeśli samolot nie zostanie znaleziony – po czasie określonym na podstawie prędkości samolotu odpytanie sąsiadów bieżącego sektora. Jeśli samolot zostanie znaleziony u następnika – zgłoszenie awarii GPS bieżącego agenta. Jeśli u innego sąsiada – awaria GPS agenta oraz błąd trasy samolotu.

- Agent nie dostał informacji o samolocie i odpowiedzialności, ale namierzył samolot – odpytanie samolotu o jego trasę. Jeśli samolot powinien pojawić się w bieżącym sektorze – ostrzeżenie o błędzie komunikacji z agentem poprzednim. Jeśli samolot nie powinien pojawić się w sektorze – błąd trasy samolotu.
- Agent dostał informację o zbliżającym się samolocie, nie dostał odpowiedzialności, ale ma samolot – ostrzeżenie o błędzie komunikacji z poprzednim agentem
- Samolot zniknie z radarów agentów i nie pojawi się na lotnisku – nie udało się lądowanie lub nastąpił wypadek samolotu – włączenie lampki ostrzegawczej, alarm do nadzorcy lotów.

2.2. Formalizacja

W tabelach 1 i 2 przedstawiono dwa scenariusze biznesowe występujące w projekcie.

Tabela 1. Pierwszy scenariusz

Identyfikator i nazwa przypadku użycia:	**Agentowy system monitorowania ruchu lotniczego**		
Utworzony przez:	Joanna Nużka Michał Motyl Kamil Pieprzycki	Data utworzenia:	28.03.2023r.
Aktor główny	Kontroler lotów	Aktor drugorzędny:	Nadzorca krajowy lotów
Wyzwalacz:	otrzymanie sygnału, że do kontrolowanego obszaru zbliża się samolot		
Opis:	Kontroler lotów siedzi w wieży monitorującej obszar. Kiedy otrzyma informację, że do jego obszaru zbliża się samolot musi przygotować się do przejęcia odpowiedzialności za niego		
Warunki początkowe	Na kontrolowanym obszarze nie ma wlatującego samolotu, mogą być inne		
Warunki końcowe	Samolot opuścił przestrzeń powietrzną obszaru (wyleciał/ wylądował)		
Przepływ normalny	**1. Normalna kontrola lotu samolotu** 1. Agent otrzymuje informację o tym, że w jego obszarze pojawi się samolot. 2. Agent po pojawieniu się samolotu w jego strefie odpytuje go o jego stan techniczny i przesyła informacje do pozostałych aktorów. 3. Namierzenie, w stronę którego sąsiada przemieszcza się samolot opuszczający jego obszar i przekazanie tej informacji 4. Wylot samolotu z obszaru kontroli 1. Pozbycie się informacji o samolocie 2. Dodanie informacji o obecności samolotu w obszarze do archiwum agenta		

Przepływy alternatywne:	1. **Samolot wylatuje z naszego lotniska** - 1. przekazanie informacji do lotniska docelowego o locie 2. wydanie pozwolenia na start samolotu 3. monitorowanie go i wysłanie sygnału do następnego agenta, kiedy samolot opuszcza obszar 2. **Samolot nie może opuścić obszaru, ponieważ kolejny jest pełny** - 1. komunikacja z sąsiednim obszarem i otrzymanie informacji, że nie można przekazać samolotu\ 2. przekazanie informacji do samolotu, aby krążył po obszarze do czasu otrzymania informacji 3. komunikacja z pozostałymi sąsiadami w celu wyboru trasy alternatywnej 4. wybór trasy alternatywnej lub zwolnienie pierwotnego obszaru 5. przekazanie informacji o przekazaniu samolotu do odpowiedniego agenta oraz informacji do samolotu o możliwości opuszczenia obszaru oraz kierunku lotu
Wyjątki:	Awarie: • Agent dostał informację planowanym przylocie samolotu, ale samolot nie przyleciał i nie dostał odpowiedzialności za samolot. • Agent nie dostał informacji o samolocie i odpowiedzialności, ale namierzył samolot. • Agent dostał informację o zbliżającym się samolocie, nie dostał odpowiedzialności, ale ma samolot. • Samolot zniknie z radarów agentów i nie pojawi się na lotnisku – nie udało się lądowanie lub nastąpił wypadek samolotu.
Rozszerzenie scenariusza bazowego	Potrzeba kontaktu kontroler → samolot Wyłączenie danego obszaru z przestrzeni powietrznej
Priorytet	bardzo wysoki
Częstotliwość użycia:	Praca w czasie rzeczywistym
Reguły biznesowe:	"Czas to pieniądz" (chcemy żeby samoloty jak najszybciej dolatywały do celu, po najkrótszej drodze)
Inne informacje:	Klient drugorzędny chciałby mieć dostęp do mapy wizualizacji pozycji samolotów na obszarze kontrolowanej przestrzeni powietrznej
Założenia wstępne:	1. Baza danych informacji o modelach samolotów 2. Zdefiniowana mapa obszarów kontroli kontrolerów 3. Nie ma obszarów na mapie, które nie należą do jakiegoś kontrolera

Tabela 2. Drugi scenariusz

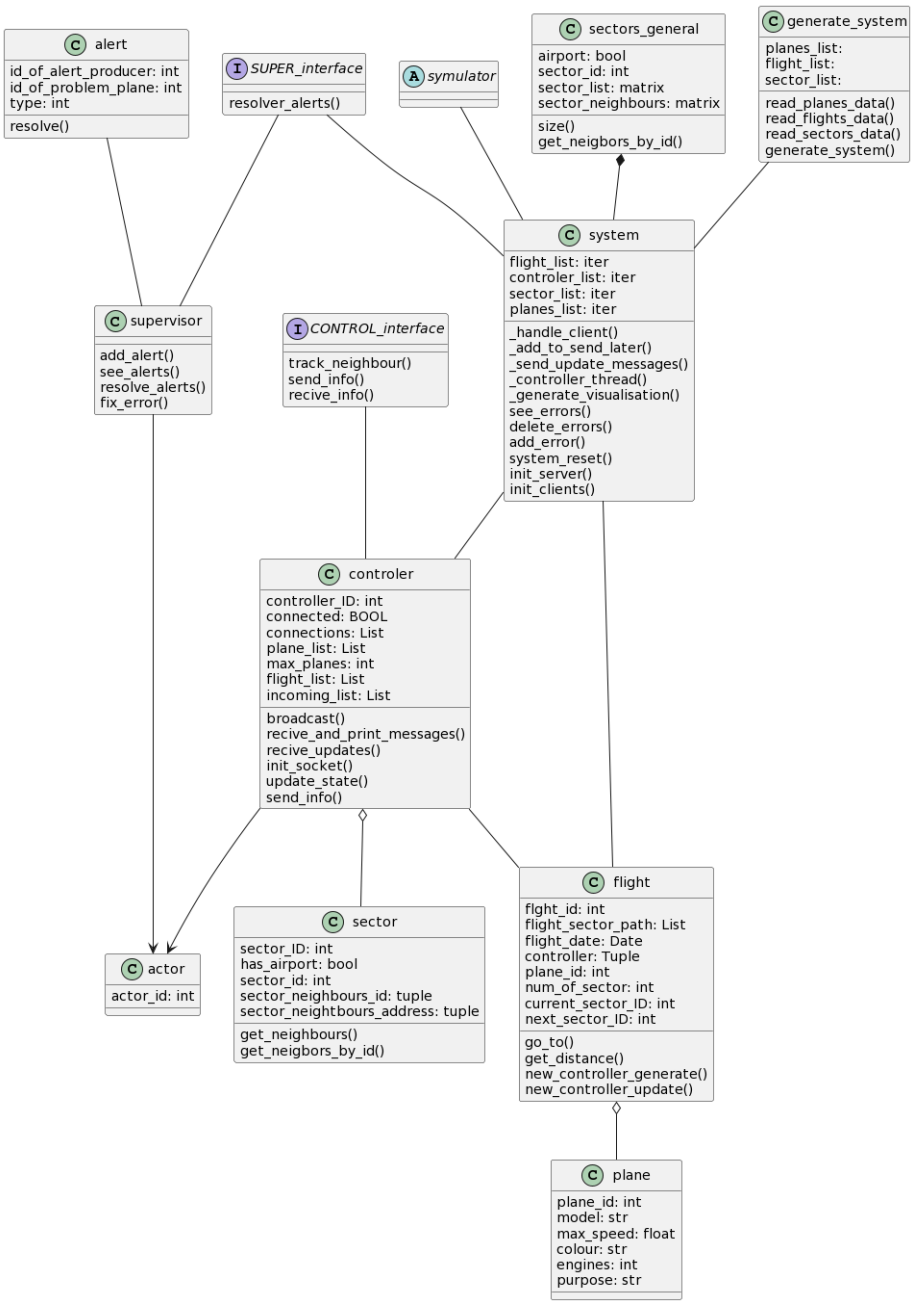
Identyfikator i nazwa przypadku użycia:	**Agentowy system monitorowania ruchu lotniczego**		
Utworzony przez:	Joanna Nużka Michał Motyl Kamil Pieprzycki	Data utworzenia:	28.03.2023r.
Aktor główny	Nadzorca krajowy lotów	Aktor drugorzędny:	Kontroler lotów
Wyzwalacz:	"zalogowanie się" do systemu przez nadzorcę		
Opis:	Nadzorca lotów wchodzi do systemu, aby zobaczyć jak działają kontrolerzy, zrewidować awarie oraz rozpatrzyć błędy związane z działaniem kontroli lotów		
Warunki początkowe	Nadzorca otwiera aplikacje z dostępem do wizualizacji		
Warunki końcowe	Nadzorca zamyka aplikacje z dostępem do wizualizacji		
Przepływ normalny	1. Wysłanie prośby o wizualizację - kliknięcie przycisku. 2. Wysłanie odpowiednich zapytań do bazy przez system. 3. Zwrócenie i wyświetlanie wizualizacji.		
Przepływy alternatywne:	**Obsługa błędów:** 1. Otrzymanie informacji o błędzie 2. Zatwierdzenie przeczytania informacji o błędzie. 3. Obsługa błędów zgodnie z krokami opisanymi w poprzednim scenariuszu (w sekcji wyjątki)		
Wyjątki:	1. Błąd komunikacji z systemem - aplikacja nie ma dostępu do danych potrzebnych do wizualizacji 2. Aplikacja nie odświeża się w czasie rzeczywistym		
Rozszerzenie scenariusza bazowego	—		
Priorytet	Wysoki		
Częstotliwość użycia:	3 do 5 razy dziennie		
Reguły biznesowe:	1. "Czas to pieniądz" (chcemy żeby samoloty jak najszybciej dolatywały do celu, po najkrótszej drodze)		

Inne informacje:	Jedyną formą komunikacji nadzorca-kontroler powinno być obsługiwanie błędów
Założenia wstępne:	Kontroler posiada dostęp do bazy danych zawierającej informacje aktualizowane przez agentów

3. Diagramy

3.1. Diagram klas

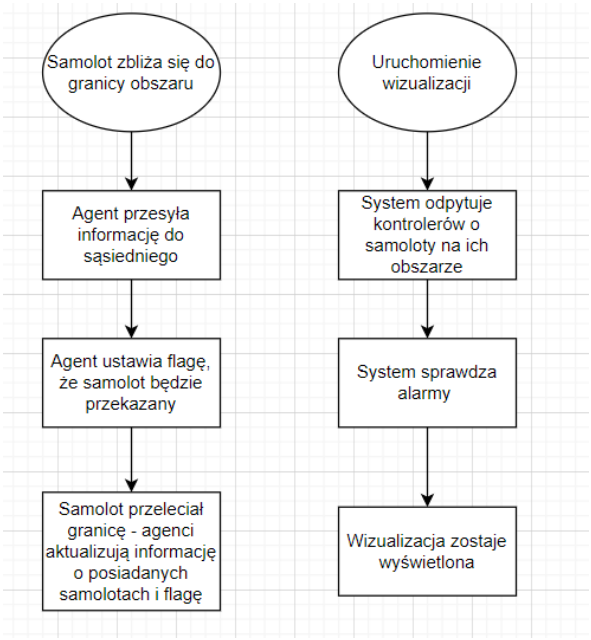
Na rysunku 1. Przedstawiono diagram klas występujących w projekcie.



Rysunek 1. Diagram klas

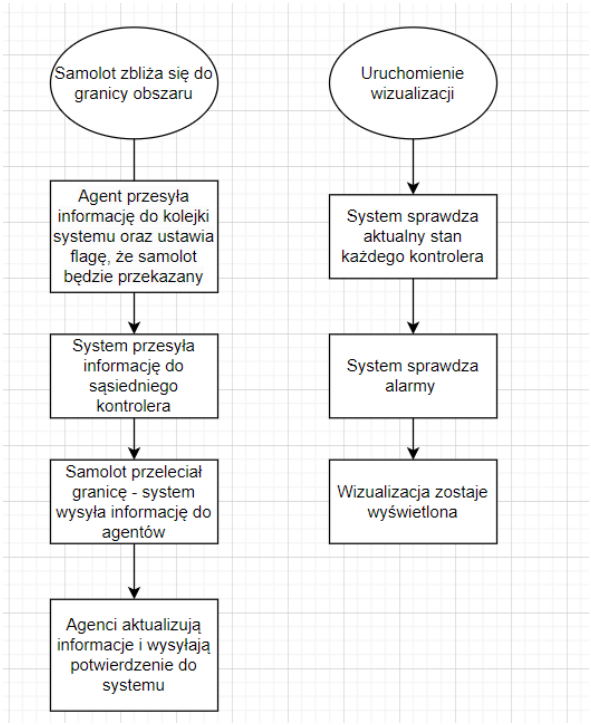
3.2. Przepływ danych

Na rysunku 2 przedstawiono planowane diagramy przepływu danych w przypadku uruchomienia wizualizacji oraz zbliżania się samolotu do granicy obszaru.



Rysunek 2. Planowany diagram przepływu danych

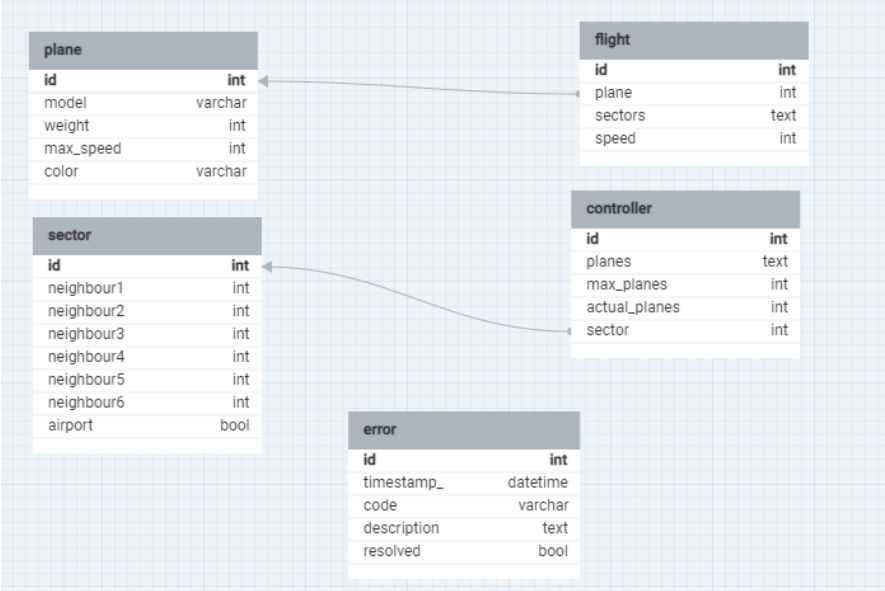
Niestety z powodu stopnia skomplikowania nie udało nam się zrealizować komunikacji bezpośrednio między kontrolerami i przesył danych odbywa się za pośrednictwem systemu. Aktualnie wygląda on więc w sposób przedstawiony na rysunku 3.



Rysunek 3. Aktualny diagram przepływu danych

3.3. Model danych

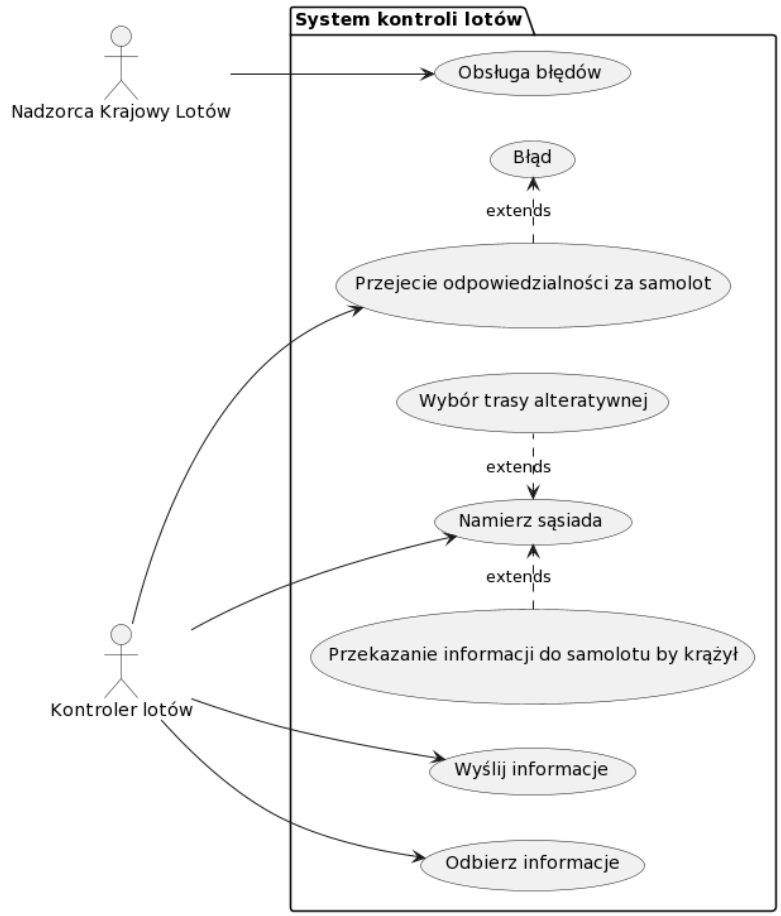
Na rysunku 4 przedstawiono model danych występujących w projekcie.



Rysunek 4. Model danych

3.4. Przypadki użycia

Na rysunku 5 przedstawiono diagram przypadków użycia projektu.



Rysunek 5. Diagram przypadków użycia

4. Aplikacja

4.1. Środowisko

Aplikacja została napisana w języku Python w wersji 3.8 z użyciem następujących bibliotek:

- socket – obsługuje komunikację między kontrolerami i lotami;
- threading – do obsługi kilku wątków;
- pickle – do serializacji i deserializacji przesyłanych danych;
- time i datetime – do obsługi czasu;
- os – do zwrócenia aktualnego folderu projektu
- OpenCV – do wizualizacji wyników symulacji;
- pandas – do obsługi wczytywana danych z plików .csv przy generacji systemu.

4.2. Realizacja komunikacji między kontrolerami

Początkowo chcieliśmy zrealizować komunikację peer-to-peer, aby kontrolerzy mogli komunikować się bezpośrednio między sobą. Niestety okazało się to zbyt trudne do zaimplementowania przy użyciu biblioteki socket. Ostatecznie komunikacja zrealizowana jest w architekturze klient-serwer, gdzie system jest serwerem, zaś kontrolerzy klientami. Podczas komunikacji wszyscy kontrolerzy zapisują informacje do wspólnej listy znajdującej się w systemie. Następnie system przechodzi przez wszystkie wiadomości, procesuje je i wysyła komunikaty do odpowiednich kontrolerów. Jeśli dany kontroler musi dokonać aktualizacji swoich danych, wysyła odpowiednią informację zwrotną do systemu. System zapisuje je w jednym miejscu, co jest następnie wykorzystywane np. podczas tworzenia wizualizacji.

4.3. Opis plików w projekcie

Projekt składa się z następujących plików:

- **system_generator.py** – zawiera funkcje służące do wygenerowania systemu na podstawie plików csv. Wczytują one dane o sektorach, planowanych lotach i samolotach i tworzą obiekt klasy System.
- **system.py** – odpowiedzialny za reprezentację systemu kontroli lotów. Posiada klasę:
  - System :
    - *"__init__(self, flights, controllers, sectors, planes, update_interval=10) "*: Konstruktor klasy System. Przyjmuje listy lotów (flights), kontrolerów (controllers), sektorów (sectors) i samolotów (planes). update_interval to opcjonalny parametr określający interwał aktualizacji systemu. Tworzy instancje klasy Supervisor i inicjalizuje zmienne.
    - *"main()":* Metoda prywatna, która ustawia gniazdo serwera i tworzy wątki dla obsługi klientów i wysyłania aktualizacji.
    - *"_controller_thread(sock, controller_sockets)* ": Metoda prywatna używana jako wątek do akceptowania nowych klientów (kontrolerów) i uruchamiania dla nich osobnych wątków.
    - *"_send_update_messages(controller_sockets)* ": Metoda prywatna używana do wysyłania aktualizacji do kontrolerów. Przyjmuje listę gniazd kontrolerów controller_sockets. W nieskończonej pętli wysyła wiadomości UPDATE do wszystkich kontrolerów i obsługuje wysyłanie oczekujących wiadomości z messages_to_send.
    - *"_add_to_send_later(msg_tuple) "*: Metoda prywatna używana do dodawania wiadomości do kolejki oczekujących na wysłanie. Przyjmuje krotkę msg_tuple, która

zawiera dane do wysłania. Dodaje tę krotkę do słownika messages_to_send z kolejnym kluczem, aby zachować kolejność.

- *"_handle_clientconnection(controller_id, controller_sockets):* Metoda prywatna używana do obsługi komunikacji między klientami (kontrolerami) a serwerem. Przyjmuje połączenie connection, identyfikator kontrolera controller_id i listę gniazd kontrolerów controller_sockets. W zależności od otrzymanych danych przetwarza i reaguje na odpowiednie komunikaty.

Dodatkowo klasa odpowiada zainicjalizację i za generację wizualizacji w formacie PNG na podstawie źródłowego obrazu "*simulation_map.png*" i zapisuje go w folderze *"simulation_visualisations*". Zapisuje plik z oznaczeniem czasu, w którym został zapisany, oraz krokiem symulacji, przy którym został zapisany

- **flight.py** zawiera dwie klasy:
  - **Flight** odpowiedzialną za reprezentowanie informacji o locie, takie jak identyfikator, ścieżka lotu, data, oraz identyfikator samolotu.

    Klasa Flight posiada również metody:
    - *"go_to(ID):* metoda do zmiany trasy lotu.
    - *"new_controller_generate(list_of_controllers)"*: metoda do generowania odpowiedniego kontrolera, do którego zostanie wysłana informacja, BEZ zmiany aktualnego kontrolera lotu. Metoda przyjmuje listę kontrolerów w systemie i zwraca krotkę (ID, Adres) odpowiedniego kontrolera.
    - *"new_controller_update(list_of_controllers)"*: metoda do aktualizacji kontrolera i sektora bieżącego lotu. Metoda przyjmuje listę kontrolerów w systemie i nie zwraca żadnej wartości.
    - *"update():* metoda służąca do aktualizuji stanu lotu. Oblicza czas od ostatniej aktualizacji, odległość pokonaną przez lot w tym czasie, oraz sprawdza, czy lot jest blisko granicy sektora lub opuszcza sektor.

  - **Plane** reprezentuje informacje o samolocie, takie jak identyfikator, model, maksymalna prędkość, kolor, liczba silników i przeznaczenie.

- **controllers.py** zawiera dwie klasy:
  - **Sector**: reprezentuje sektor, stanowiący obszar odpowiedzialności przydzielony odpowiedniemu kontrolerowi lotów.

    Przechowuje ona informacje o sektorach, takie jak identyfikator, czy posiadają one na swoim obszarze lotnisko, listę sąsiadujących ze sobą nawzajem sektorów

  - **Controller**: reprezentuje kontrolera lotów. Oto jej metody i pola:
    - *__init__(self, id, plane_list, flight_list):* Konstruktor klasy, który inicjalizuje pola obiektu. Przyjmuje identyfikator kontrolera (id), listę obiektów typu Plane oraz listę obiektów typu Flight
    - *„Broadcast(data)":* Metoda używana do wysyłania danych do serwera.
    - *„receive_and_print_messages()"*: Metoda używana do odbierania i wyświetlania wiadomości od innych klientów (kontrolerów).
    - *„receive_updates()":* Metoda używana do odbierania aktualizacji od serwera systemu.

- *„init_socket()":* Metoda inicjująca gniazdo klienta.
- *„main_socket_start()":* Metoda inicjująca gniazdo klienta i łącząca je z serwerem. Tworzy osobne wątki do odbierania aktualizacji i odbierania/wyświetlania wiadomości.
- *„main_socket_stop()"*: Metoda zamykająca gniazdo klienta.
- *„update_state()":* Metoda aktualizująca stan kontrolera i lotów. Aktualizuje każdy lot w flight_list_flights i podejmuje odpowiednie działania w zależności od stanu lotu.
- *„_send_info"(flight_nearing: Flight, controllers_list: List):* Metoda prywatna używana do wysyłania informacji o locie, który wkrótce będzie gotowy do zmiany kontrolera.
- *„send_plane"(flight_over: Flight, controller_list: List):* Metoda używana do wysyłania samolotu, który jest gotowy do opuszczenia przestrzeni powietrznej kontrolera i przekroczenia do następnego kontrolera.

❖ **supervisor.py**: zawiera klasę Supervisor, odpowiedzialną za monitorowanie i zarządzanie błędami i ostrzeżeniami generowanymi przez aplikację.

Składa się ona z metod:

- *„add_alert"*: Służy do dodawania nowego alertu do listy list_of_alerts
- *„see_alerts"*: służy do wyświetlania wszystkich alertów znajdujących się na liście list_of_alerts. Zawiera ona informacje o typie alertu, kontrolerze, który go wygenerował, informacji czy alert został rozwiązany
- *„resolve_alerts"*: służy do zatwierdzania alertów.

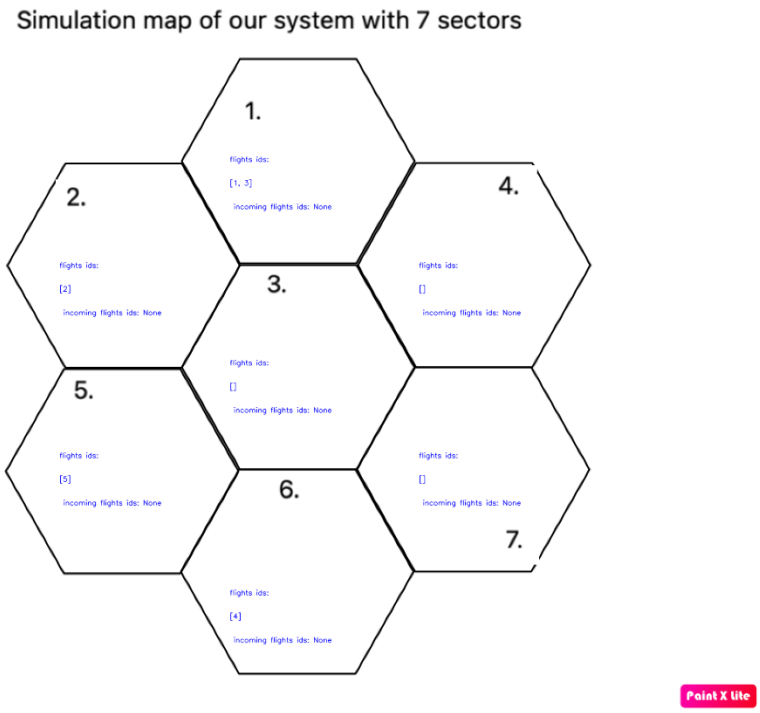
4.4. Uruchomienie programu

Przed uruchomieniem programu należy posiadać zainstalowanego pythona w wersji minimum 3.8 wraz z wszystkimi bibliotekami opisanymi w punkcie 4.1. Należy zadbać o to, aby w folderze classes/data znajdowały się pliki konfiguracyjne: controllers.csv, flights.csv, planes.csv i sectors.csv, zaś w folderze simulation_visualisation mapa sektorów w pliku simulation_map.png.

Aby rozpocząć działanie programu należy uruchomić plik main.py. Rozpocznie się wtedy symulacja, której kolejne etapy wypisywane są w konsoli (rys. 6). Wyniki wizualizacji zapisywane są w folderze simulation_visualisation w plikach opatrzonych odpowiednim numerem kroku (rys. 7).

```
Flight controller system is now running and waiting for connections...
==========================================
UPDATE STEP 0
Connected to the flight controller system.Connected to: ('127.0.0.1', 58558)

Connected to the flight controller system.
Connected to: ('127.0.0.1', 58559)
Connected to the flight controller system.Connected to: ('127.0.0.1', 58560)

Connected to the flight controller system.Connected to: ('127.0.0.1', 58561)

Connected to the flight controller system.Connected to: ('127.0.0.1', 58562)

Connected to the flight controller system.
Connected to: ('127.0.0.1', 58563)
Connected to the flight controller system.Connected to: ('127.0.0.1', 58564)

<socket.socket fd=1964, family=2, type=1, proto=0, laddr=('127.0.0.1', 8000), raddr=('127.0.0.1', 58558)>
<socket.socket fd=2204, family=2, type=1, proto=0, laddr=('127.0.0.1', 8000), raddr=('127.0.0.1', 58559)>
<socket.socket fd=2240, family=2, type=1, proto=0, laddr=('127.0.0.1', 8000), raddr=('127.0.0.1', 58560)>
<socket.socket fd=2256, family=2, type=1, proto=0, laddr=('127.0.0.1', 8000), raddr=('127.0.0.1', 58561)>
<socket.socket fd=2236, family=2, type=1, proto=0, laddr=('127.0.0.1', 8000), raddr=('127.0.0.1', 58562)>
<socket.socket fd=2332, family=2, type=1, proto=0, laddr=('127.0.0.1', 8000), raddr=('127.0.0.1', 58563)>
<socket.socket fd=2372, family=2, type=1, proto=0, laddr=('127.0.0.1', 8000), raddr=('127.0.0.1', 58564)>
```

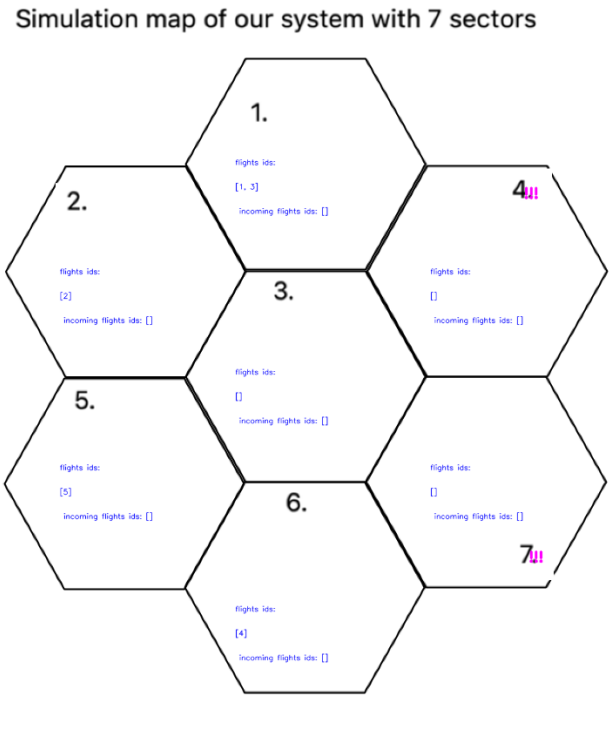
Rysunek 6. Wyniki uruchomienia programu



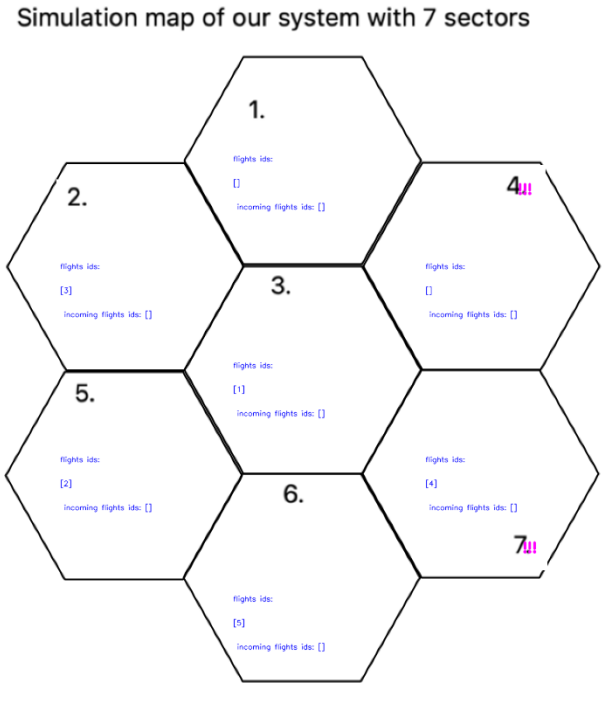
Rysunek 7. Wizualizacja po uruchomieniu programu

5. Wyniki symulacji

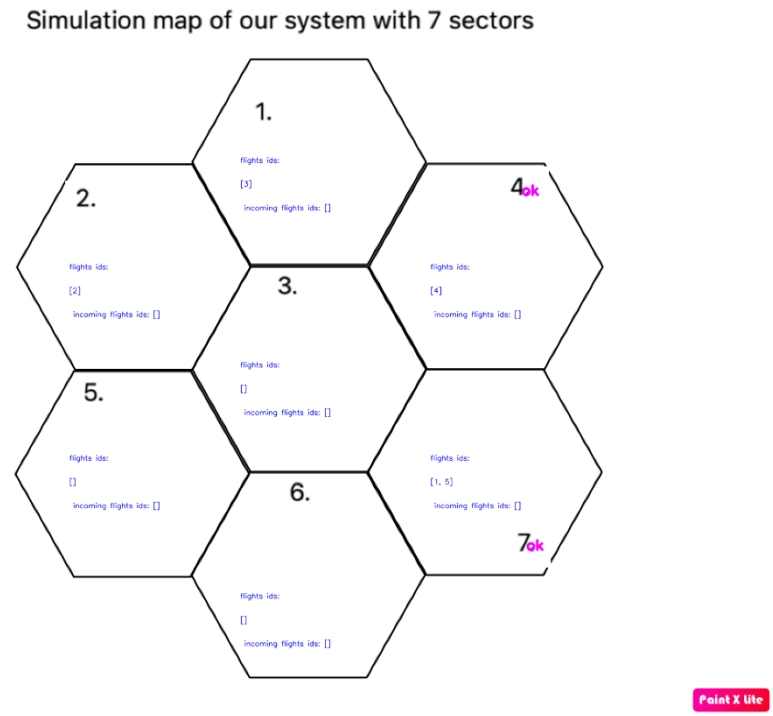
Na rysunkach 7 – 9 przedstawiono wyniki wizualizacji symulacji projektu. Alerty zostały wygenerowane i naprawione sztucznie w celu pokazania działania ich wizualizacji, ponieważ oryginalnie nie występowały w naszym scenariuszu testowym.



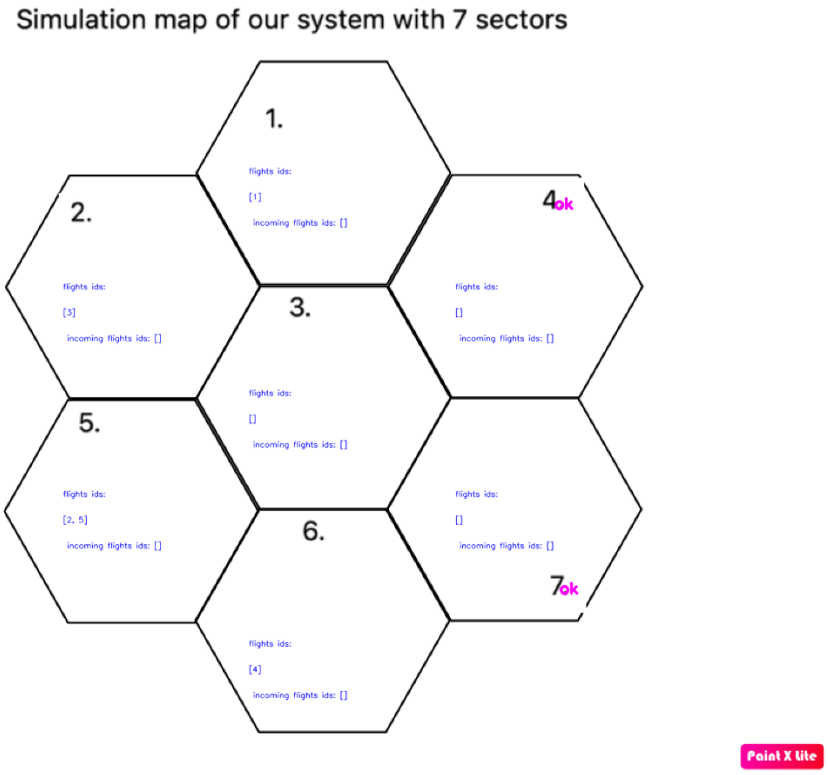
Rysunek 8. Zerowy krok projektu



Rysunek 9. Pierwszy krok projektu



Rysunek 10. Drugi krok projektu



Rysunek 11. Trzeci krok projektu

Na podstawie wizualizacji możemy zauważyć kolejne etapy tras samolotów. Na rysunkach 8 i 9 znajdują się czerwone wykrzykniki w dwóch sektorach – obrazują one sztucznie wygenerowane alerty. Na rysunkach 10 i 11 zamiast wykrzykników pojawił się napis „ok” – oznacza on, że alerty zostały już rozpatrzone i naprawione.

6. Podsumowanie

Udało nam się zaimplementować logikę projektu, jednak musieliśmy dokonać w niej zmian i uproszczeń.

Problemem, który napotkaliśmy podczas implementacji było to, że podczas threadingu za każdym razem tworzyły nam się nowe sockety. Nie mogliśmy więc poprawnie przesłać danych między agentami. Problem ten udało się jednak rozwiązać.

Inną trudnością jest problem z dostępem do danych zawartych w socketach przez inne elementy projektu, co przeszkadza m.in. w otrzymaniu danych do poprawnej wizualizacji.

Ostatecznie z uwagi na powyższe problemu musieliśmy zmienić sposób komunikacji między kontrolerami z peer-to-peer na klient-serwer, gdzie serwerem jest system, a klientami są kontrolerzy. Niestety w ten sposób system stracił jedną z zalet rozproszoności, jaką jest możliwość poprawnego działania nawet w przypadku awarii jego części. W naszym projekcie w przypadku awarii serwera kontrolerzy nie będą mogli się ze sobą komunikować i system przestanie działać.

Projekt okazał się być bardzo złożony i trudny w implementacji. Działanie systemu rozproszonego jest dość skomplikowane, a u nas dodatkowo nie było jasnego podziału na klientów i serwer, co znacząco utrudniło komunikację za pomocą socketów. Dodatkową trudnością był fakt, że każdy z nas korzystał z biblioteki socket w pythonie po raz pierwszy w życiu, więc wszystkiego musieliśmy się uczyć podczas pracy na projektem.

Jednak mimo braku ostatecznego sukcesu projekt pozwolił nam na zapoznanie się z ideą i działaniem systemu rozproszonego. Nauczyliśmy się także, na co zwracać uwagę podczas jego projektowania. Poznaliśmy także kolejne kroki procesu planowania i implementacji systemu. Zapoznaliśmy się także z jednym z narzędzi do implementacji systemu rozproszonego i komunikacji między jego elementami – pythonową biblioteką socket.

7. Podział pracy:

Zadanie	Osoba odpowiedzialna
Opis założeń projektu	Joanna Nużka, Michał Motyl
Scenariusze biznesowe	Joanna Nużka, Michał Motyl, Kamil Pieprzycki
Diagram przepływu danych	Joanna Nużka
Diagram przypadków użycia	Kamil Pieprzycki, Michał Motyl, Joanna Nużka
Diagram klas	Kamil Pieprzycki, Michał Motyl, Joanna Nużka
Model danych	Kamil Pieprzycki, Michał Motyl, Joanna Nużka
Generacja systemu na podstawie danych z pliku	Joanna Nużka
Implementacja logiki projektu	Michał Motyl
Implementacja komunikacji między kontrolerami i lotami	Kamil Pieprzycki
Dokumentacja	Joanna Nużka, Kamil Pieprzycki