POLITECHNIKA WARSZAWSKA

AGENTOWE I AKTOROWE SYSTEMY DECYZYJNE

Projekt

SMART CITY

DYNAMICZNY ROZKŁAD JAZDY

Prowadzący:

dr inż. Dominik Ryżko

Zespół Manhattan:

Zuzanna Górecka

Kamil Szydłowski

Michał Oracki

Warszawa 2023/2024

Identyfikacja i opis problemu

W wielu miastach często napotykamy na problem niewystarczającej efektywności komunikacji miejskiej. Konieczność wielokrotnych przesiadek w trakcie podróży z jednego punktu do drugiego jest powszechnym utrudnieniem. Każda przesiadka wydłuża czas podróży, ponieważ po zakończeniu etapu jazdy jednym środkiem transportu konieczne jest oczekiwanie na kolejny. Dodatkowo, czasami zdarza się, że w wyniku opóźnienia lub przedwczesnego przybycia pojazdów komunikacji publicznej nie zdążymy na planowaną przesiadkę.

Ponadto, komunikacja miejska zatrzymuje się na z góry ustalonych przystankach. Część mieszkańców ma daleko do najbliższego przystanku. Istnieją również takie przystanki, na których mało ludzi wsiada lub wysiada.

Należy również zauważyć, że autobusy mogą mieć różne obłożenie o tych samych godzinach, ale w różnych dniach lub tygodniach. Niekiedy są zatłoczone, a czasem praktycznie puste. Różnice w obłożeniu pojazdów mogą być znaczące, co prowadzi do nieefektywnego wykorzystania zasobów i nieoptymalnego zarządzania zużyciem paliwa.

Opis rozwiązania

Celem naszego projektu jest stworzenie systemu, który zapewni wygodny, bezprzesiadkowy transport publiczny. Nasz system będzie zarządzał autobusami poprzez dostosowywanie na bieżąco liczby oraz tras autobusów w zależności od zapotrzebowania. Nie będą funkcjonować z góry przyjęte rozkłady jazdy – konkretne godziny przyjazdu i odjazdu oraz konkretne lokalizacje przystanków. Autobusy będą odbierać i odwozić pasażerów do przez nich wybranych miejsc.

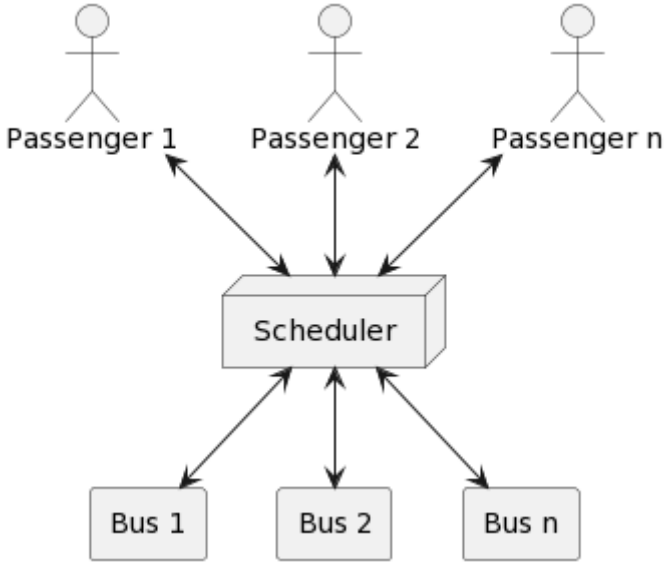
Koncepcja systemu

Pasażerowie będą zgłaszać chęć przejazdu z jednego punktu do drugiego poprzez aplikacje mobilne. Zgłoszenia będą trafiać do centralnego systemu, który roześle informacje do autobusów. Następnie każdy autobus na podstawie swojego stanu – czy stoi czy jedzie (czy jest aktywny), lokalizacji, liście przystanków, wylicza koszt podjęcia nowego pasażera. Przesyła informacje do centralnego systemu, który wybierze według niego najodpowiedniejszy środek transportu i przekaże informacje pasażerowi oraz wybranemu autobusowi.

W przypadku małego zapotrzebowania część autobusów może zostać w miejscu.

Przyjmujemy, że autobusy poruszają się po ulicach o planie Manhattanu (siatka). Odległość jednego punktu do drugiego jest odległością miejską.

Proponowana architektura rozwiązania przedstawiona jest na rys. 1



Rys. 1 Architektura wieloagentowa na poziomie logicznym

Dla kogo

Pierwszoplanowi interesariusze

- Zarząd transportu miejskiego ZTM
  Dzięki systemowi efektywniej wykorzysta pojazdy, co poskutkuje redukcją paliwa. Może zyskać więcej klientów, dzięki lepszemu zarządzaniu komunikacją.
- Kierowcy autobusów
  Zamiast realizować puste kursy mogą odpocząć.
- Pasażerowie
  Będą mieli zapewniony bez przesiadkowy transport.

Drugoplanowi interesariusze

- Taksówkarze
  Mogą stracić część klientów.

Repozytorium

https://gitlab-stud.elka.pw.edu.pl/manhattan/dynamic-timetable

---

Projektowanie aplikacji

Wymagania funkcjonalne

- System zapewnia, że autobusy są dostępne 24/7
- Pasażer ma możliwość zgłoszenia chęci przejazdu z dowolnego punktu A do punktu B
- Zgłoszenie powinno trafić do centralnego systemu
- System akceptuje zgłoszenia każdego klienta i zwraca pasażerowi informację w jaki autobus powinien wsiąść

Role

- Scheduler – Centralny system
- Driving bus – jeżdżący bus
- Routing bus – bus komunikujący się z systemem centralnym i planujący trasę
- Passenger – pasażer

Schemat ról

Role schema: **PASSENGER**
Description: Passenger selects destination of the trip, then waits for the bus to arrive and gets taken by that bus to previously selected destination
Protocols and Activities SelectDestination, RequestForTravel, AwaitTravelPlan, WaitForBus, GetIntoBus, DriveInBus, ExitBus, ReceiveBusFailureMsg, HandleBusFailure
Permissions **read** **supplied** tripPlan **generates** passengerInfo
Responsibilities Liveness: **PASSENGER** = (SelectDestination . RequestForTravel . (AwaitTravelPlan)$^{\omega}$. (WaitForBus)$^{\omega}$. GetIntoBus . DriveInBus . ExitBus) || (ReceiveBusFailureMsg$^{\omega}$. [HandleBusFailure]) Safety: schedulerRunning = true

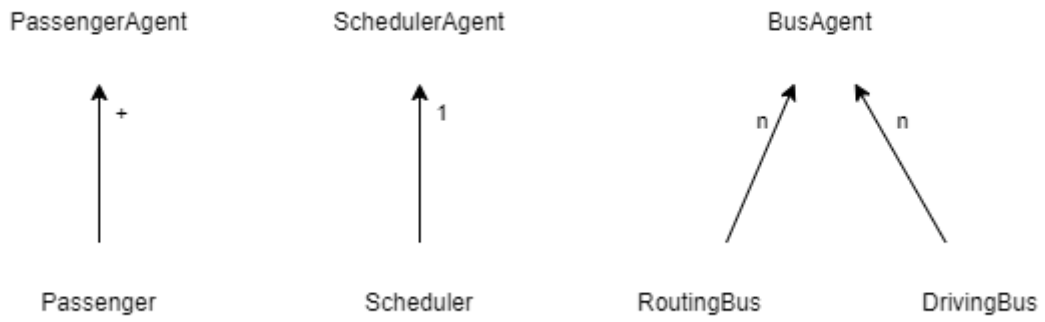
Role schema: **SCHEDULER**
Description: Handles user request for new trips and selects the most optimal bus
Protocols and Activities ReceiveTravelRequest, SavePassagerInfo, Cfp, ReceiveBusPropose , SelectBus, ReplyBus, SendTravelPlan
Permissions **read** **supplied** passengerInfo **supplied** busInfo **generates** tripPlan replyBusProposal
Responsibilities Liveness: **SCHEDULER**=(ReceiveTravelRequest. SavePassagerInfo . Cfp . ReceiveBusPropose . SelectBus . ReplyBusProposal . SendTravelPlan)$^{\omega}$ Safety: true

Role schema: **ROUTINGBUS**
Description: Handles scheduler requests for new passengers
Protocols and Activities ReceiveCfp, GetBusInformation, CalculatePotentialCost, SendBusInfo, WaitForDecision, CalculateRoute, SendNewRoute, ReceivePassengerMsg
Permissions **read** **supplied** replyBusProposal **generates** busInfo

route
Responsibilities Liveness: **ROUTINGBUS** =(ReceiveCfp. GetBusInformation . CalculatePotentialCost . SendBusInfo . WaitForDecision . [CalculateRoute . SendNewRoute])$^{\omega}$ || ReceivePassengerMsg Safety: schedulerRunning = true

Role schema: **DRIVINGBUS**
Description: Drives passengers to requested destinations
Protocols and Activities GetNewRoute, SaveRoute, ChangeStatus, Move, HandlePassenger, HandleFailure, InformPassenger
Permissions **read** **supplied** route **changes** busState
Responsibilities Liveness: **DRIVINGBUS** =(([GetNewRoute . SaveRoute . [ChangeStatus]]) | ([ChangeStatus]) . [Move] . [HandlePassenger])$^{\omega}$ || [HandleFailure . InformPassenger] Safety: routingBusRunning = true

Model agentów



Klasa **PassengerAgent** będzie odgrywać rolę **Passenger**. Instancji klasy **PassengerAgent** może być wiele. Aby system miał sens, powinna być minimum 1 instancja.

Klasa **SchedulerAgent** będzie odgrywać rolę **Scheduler**. Będzie dokładnie 1 instancja tej klasy.

Złożeniem ról **RoutingBus** i **DrivingBus** będzie klasa **BusAgent**. Jej instancji będzie pewna określona liczba.

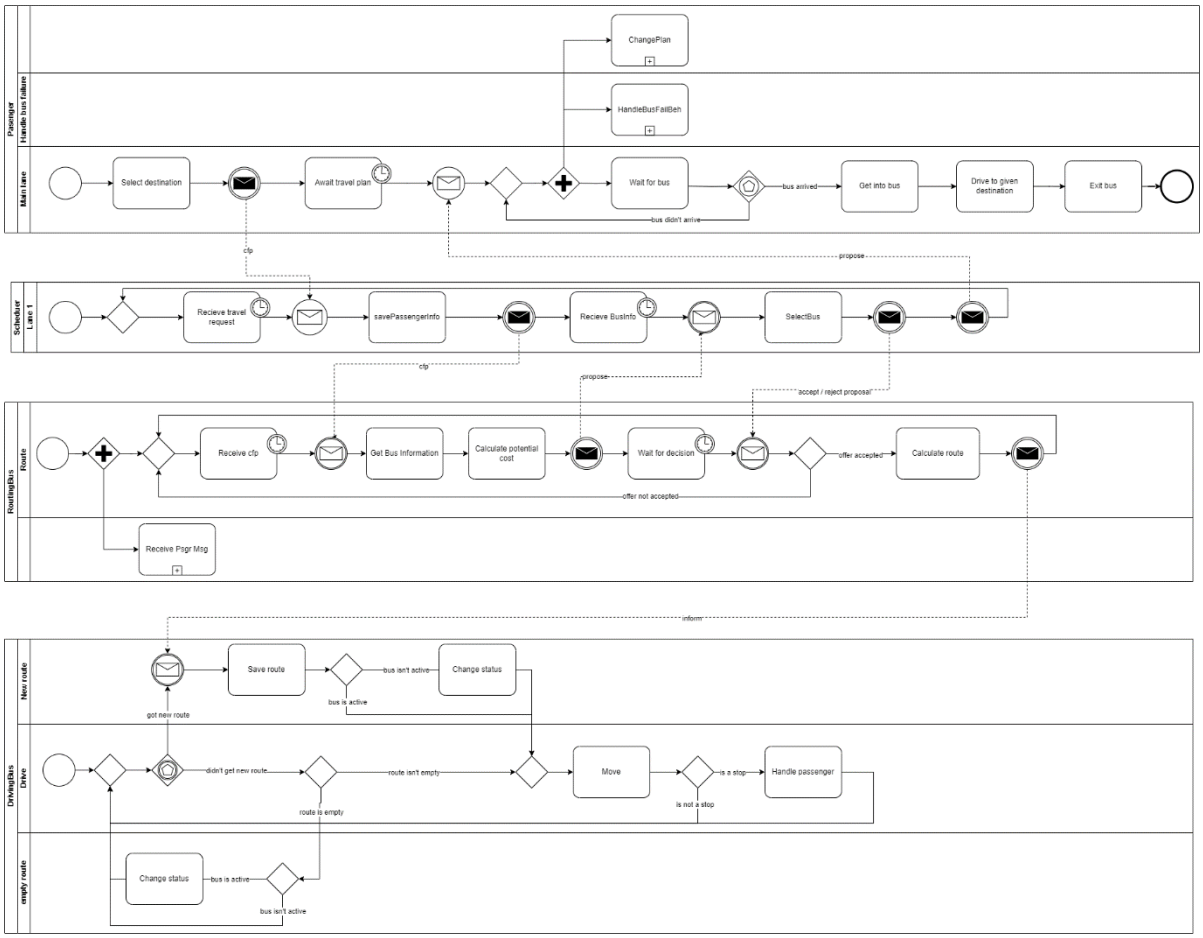
Metoda projektowania

BPMN 2.0

Diagramy

Diagramy kolaboracji

Główny scenariusz – realizacja przejazdów



Passenger (Pasażer)

Proces rozpoczyna **Pasażer** wysyłający prośbę o przejazd do **Centralnego Systemu**. Przesyła on informacje o lokalizacji punktu początkowego i końcowego. Następnie otrzymuje on informację od **Centralnego Systemu** o numerze autobusu do którego ma wsiąść. **Pasażer** czeka na autobus w lokalizacji początkowej. Gdy autobus podjedzie, wsiada do pojazdu. Autobus zawozi go do wybranej lokalizacji (po drodze mogą być inne przystanki). Gdy **Pasażer** znajdzie się w docelowym miejscu, wysiada z autobusu.

Scheduler (Centralny System)

Na początku **Centralny System** dostaje prośbę o przejazd od **Pasażera**. Zapisuje informacje o lokalizacji jego przystanku początkowego i końcowego. Następnie wysyła żądanie do wszystkich **Autobusów Planujących Trasy** o przesłanie informacji o hipotetycznym koszcie przyjęcia nowego pasażera. Na

podstawie tych informacji wybiera według niego najlepszy Autobus. Wysyła informację zwrotną do wszystkich **Autobusów Planujących Trasy** – jeden dostanie akceptację przyjęcia nowego pasażera, reszta odrzucenie oferty. Wysyła również informację do **Pasażera** o numerze busa, do którego ma wsiąść. Po wykonaniu tych wszystkich kroków **Centralny System** jest gotowy do przyjęcia nowych zgłoszeń od kolejnych **Pasażerów**.

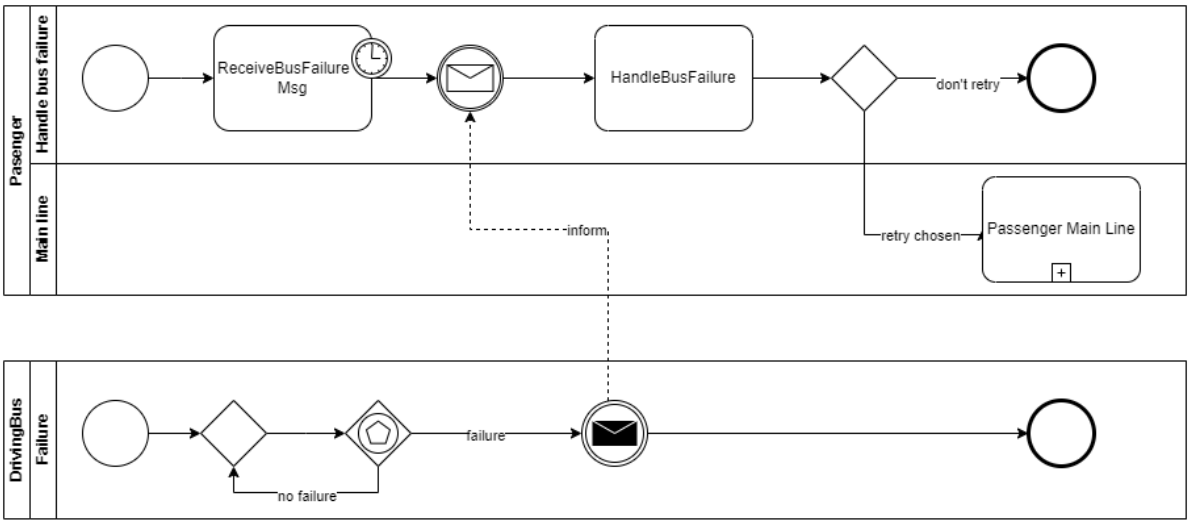
RoutingBus (Autobus Planujący Trasę)

Inicjując proces, **Autobus Planujący Trasę** oczekuje na przyjęcie od **Centralnego Systemu** propozycji przewozu nowego pasażera (oferty przystąpienia do aukcji). Następnie ustala swój stan i oblicza, jak bardzo jego trasa musiałaby się w efekcie zmienić. Ten wynik przesyła do **Centralnego Systemu**, czeka na decyzję, a w konsekwencji odbiera je. Jeśli rozstrzygnięcie jest pozytywne, wyznacza nową trasę (zmienia kolejność przystanków z uwzględnieniem nowych) i niezwłocznie przesyła ją przyporządkowanemu mu **Autobusowi Jeżdżącemu**. Na sam koniec iteracja się powtarza - **Autobus Planujący Trasę** rozpoczyna oczekiwanie na otrzymanie propozycji przystąpienia do nowej aukcji. Tak samo się dzieje, jeśli wymienione wyżej rozstrzygnięcie jest negatywne.

DrivingBus (Autobus Jeżdżący)

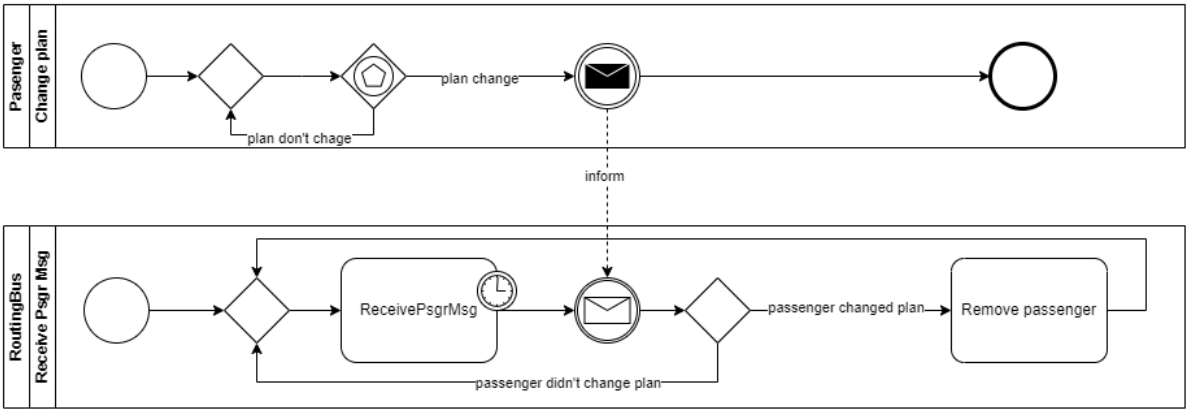
Na samym początku procesu, **Autobus Jeżdżący** nie posiada ustalonej trasy (na poziomie koncepcyjnym stoi w zajezdni). Po jej otrzymaniu od **Autobusu Planującego Trasę**, zapisuje ją sobie i zmienia status na aktywny. Rozpoczyna poruszanie się (dla przypomnienia: używamy metryki Manhattan, a zatem ruch będzie się odbywać po szachownicy). Jeśli w danym *kwadracie* oczekuje **Pasażer**, to jest on obsługiwany. Po tej czynności, **Autobus Jeżdżący** powraca do sprawdzenia, czy gotowa jest nowa trasa. Jeśli tak – postępuje analogicznie jak przy starcie procesu. W przeciwnym przypadku – sprawdza, czy pozostała trasa jest pusta. Obecność kolejnych przystanków jest równoznaczna z koniecznością dotarcia do nich według przedstawionego algorytmu. Dotarcie do wszystkich przystanków pozwala na postój, a tym samym na powrót do oczekiwania na przyjęcie nowej trasy.

Awaria autobusu



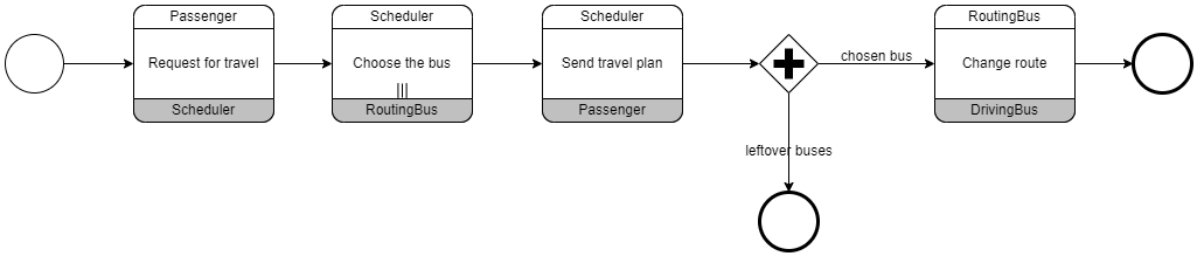
W przypadku wystąpienia awarii **DrivingBus** przesyła informację o awarii do wszystkich przyporządkowanych **Pasażerów** – zarówno do tych, którzy czekają, aż autobus przyjedzie jak i tych, którzy już jadą autobusem. Następnie pasażerowie mogą zdecydować się ponownie skorzystać z usług systemu, wysyłając ponownie prośbę o przejazd do **Schedulera** i rozpoczynając ponownie proces przejazdu.

Zmiana planów pasażera



Istnieje również możliwość, że **Pasażer** zmieni swoje plany – zrezygnuje z przejazdu jeszcze zanim autobus przyjedzie (np. z powodu zbyt długiego czasu oczekiwania) lub podczas podróży (np. wysiądzie na innym przystanku po drodze). Dlatego równolegle **RoutingBus** nasłuchuje wiadomości od **Pasażerów**. Jeżeli dostanie informację o rezygnacji **Pasażera** z przejazdu usuwa przystanek początkowy (jeżeli jeszcze nie odebrał **Pasażera**) i przystanek końcowy **Pasażera**.

Diagram choreografii



W pierwszym kroku **Pasażer** przesyła do **Centralnego Systemu** żądanie realizacji podróży. W oparciu o to, **Centralny System** przeprowadza aukcję – rozsyła **Autobusom Planującym Trasę** propozycje przyjęcia nowego pasażera. Każdy **Autobus Planujący Trasę** zwraca **Centralnemu Systemowi** informację o tym, jak bardzo jego trasa musiałaby się w efekcie zmienić. Po wyborze najkorzystniejszej oferty **Pasażer** zostaje poinformowany o ustalonym planie podróży. **Centralny System** przesyła również **Autobusom Planującym Trasę** swoje decyzje (jeden agent otrzymuje rozstrzygnięcie pozytywne, reszta – negatywne). Na koniec **Autobus Planujący Trasę**, który otrzymał decyzję pozytywną, przesyła przyporządkowanemu mu **Autobusowi Jeżdżącemu** polecenie zmiany trasy.

Implementacja systemu

Język i framework

Niniejszy projekt realizujemy w języku **Python 3** we frameworku **Spade**.

Implementacja agentów

W celu przedstawienia **najważniejszej funkcjonalności systemu** zdecydowaliśmy się na implementację agentów: **Passenger**, **Scheduler** oraz **RoutingBus**. Ich integracja pozwala zademonstrować cykl działania począwszy od **zgłoszenia przejazdu**, przez **wybór autobusu do realizacji trasy**, aż po **powiadomienie pasażera o wyznaczeniu pojazdu**.

Nie zdecydowaliśmy się natomiast na implementację **DrivingBus**, przez co autobusy nie przemieszczają się. W zamian tego uwzględniliśmy jednak sytuacje awaryjne takie jak: (1) **rezygnacja pasażera z podróży** i (2) **awaria autobusu**.

Passenger

Atrybuty (pola)

Pole	**Opis**
main_beh	Zawiera informacje na temat głównego zachowania agenta
change_plan_beh	Zawiera informacje na temat zachowania podczas zmiany planów
starting_point	Koordynaty początkowe dla pasażera w formacie *tuple[float, float]*, np. (1.21, 99.8)
destination	Koordynaty końcowe podróży w formacie *tuple[float, float]*, np. (1.21, 99.8)
bus_id	ID przydzielonego busa
passenger_id	ID pasażera
travel_counter	Licznik iteracji podróży

Zachowania

Zachowanie	**Typ**	**Opis**
PassengerBehaviour	FSMBehaviour	Zachowanie złożone z następujących po sobie pojedynczych stanów wyznaczających główny cykl życia agenta.
HandleBusFailBeh	FSMBehaviour	Zachowanie złożone z następujących po sobie pojedynczych stanów wyznaczających kroki podejmowane w celu poradzenia sobie z awarią busa
ChangePlan	CyclicBehaviour	Zachowanie cykliczne umożliwiające zmianę planu przez pasażera

Cykl życia

Stan	**Opis**	**Możliwe przejścia**

SELECT_DESTINATION	Wybranie koordynatów końcowych podróży	REQUEST_TRAVEL
REQUEST_TRAVEL	Prośba o wyznaczenie busa	AWAIT_TRAVEL_PLAN
AWAIT_TRAVEL_PLAN	Oczekiwanie na plan podróży	SELECT_DESTINATION
		WAIT_FOR_BUS
WAIT_FOR_BUS	Oczekiwanie na busa	TRAVEL
TRAVEL	Podróż	EXIT_BUS_SUCCESSFULL
EXIT_BUS_SUCCESSFULL	Zakończenie powodzeniem podróży	
BUS_FAILURE_MSG	Otrzymanie informacji o awarii busa	BUS_FAILURE_MSG
		HANDLE_BUS_FAILURE
HANDLE_BUS_FAILURE	Poradzenie sobie z awarią busa	EXIT_BUS_FAILURE
EXIT_BUS_FAILURE	Zakończenie podróży z niepowodzeniem	

Wysyłane komunikaty

Komunikat	**Odbiorca**	**Performatyw**	**Ontologia**	**Język treści**
Prośba nowej podróży od pasażera	Scheduler	cfp	travel_request	JSON
Rezygnacja z podróży	RoutingBus	inform	resignation	JSON

Odbierane komunikaty

Komunikat	**Nadawca**	**Performatyw**	**Ontologia**	**Język treści**
Oferta planu podróży	Scheduler	propose	travel_request	JSON

Scheduler

Atrybuty (pola)

Pole	**Opis**
msg	Zawiera aktualną wiadomość
passenger_info	Informacje dotyczące pasażera
selected_bus	Informacje na temat aktualnie wybranego busa
buses	Lista dostępnych busów
costs	Mapa kosztów

Zachowania

Zachowanie	**Typ**	**Opis**
SchedulerBehaviour	FSMBehaviour	Zachowanie złożone z następujących po sobie pojedynczych stanów wyznaczających główny cykl życia agenta.

Cykl życia

Stan	Opis	Możliwe przejścia
RECEIVE_PASSENGER	Otrzymanie informacji o potencjalnym pasażerze	RECEIVE_PASSENGER SAVE_PASSENGER_INFO
SAVE_PASSENGER_INFO	Zapisanie informacji o pasażerze	CFP
CFP	Wysłanie informacji o pasażerze do busa	RECEIVE_BUS_PROPOSE
RECEIVE_BUS_PROPOSE	Otrzymanie informacji o propozycji busa	CFP SELECT_BUS
SELECT_BUS	Wybranie busa	REPLY_BUS
REPLY_BUS	Akceptacja busa	SEND_TRAVELPLAN
SEND_TRAVELPLAN	Wysłanie planu podróży do pasażera	RECEIVE_PASSENGER

Wysyłane komunikaty

Komunikat	Odbiorca	Performatyw	Ontologia	Język treści
Oferta przewozu nowej osoby	RoutingBus	cfp	select_bus	JSON
Akceptacja przewozu nowej osoby	RoutingBus	accept	select_bus	JSON
Oferta planu podróży	Passenger	propose	travel_request	JSON

Odbierane komunikaty

Komunikat	Nadawca	Performatyw	Ontologia	Język treści
Prośba nowej podróży od pasażera	Passenger	cfp	travel_request	JSON
Koszt zmiany trasy	RoutingBus	propose	select_bus	JSON

RoutingBus

Atrybuty (pola)

Pole	Opis
id	Identyfikator
active	Zawiera informację, czy autobus jest w ruchu
path	Lista punktów wyznaczająca w podanej kolejności drogę
potential_path	Potencjalna nowa trasa wyznaczona podczas estymacji kosztu zmiany trasy

msg	Odebrana od schedulera wiadomość
position	Aktualna pozycja

Zachowania

Zachowanie	**Typ**	**Opis**
RoutingBusBehaviour	FSMBehaviour	Zachowanie złożone z następujących po sobie pojedynczych stanów wyznaczających główny cykl życia agenta.
ReceivePassengerMsg	CyclicBehaviour	Cyklicznie wykonywane zachowanie do sprawdzania, czy pasażer odwołał przejazd.

Cykl życia

Stan	**Opis**	**Możliwe przejścia**
RECEIVE_CFP	Oczekiwanie na przyjęcie oferty przewozu nowej osoby	RECEIVE_CFP
		GET_BUS_INFORMATION
GET_BUS_INFORMATION	Ustalenie swojej pozycji (x, y)	CALCULATE_POTENTIAL_COST
CALCULATE_POTENTIAL_COST	Obliczenie kosztu zmiany trasy	WAIT_FOR_DECISION
WAIT_FOR_DECISION	Oczekiwanie na decyzję o zmianie trasy	CALCULATE_ROUTE
		RECEIVE_CFP
CALCULATE_ROUTE	Wyznaczenie nowej trasy	RECEIVE_CFP

Wysyłane komunikaty

Komunikat	**Odbiorca**	**Performatyw**	**Ontologia**	**Język treści**
Koszt zmiany trasy	Scheduler	propose	select_bus	JSON

Odbierane komunikaty

Komunikat	**Nadawca**	**Performatyw**	**Ontologia**	**Język treści**
Oferta przewozu nowej osoby	Scheduler	cfp	select_bus	JSON
Akceptacja przewozu nowej osoby	Scheduler	accept	select_bus	JSON
Rezygnacja z przejazdu	Passenger	inform	resignation	JSON

Standardy i protokoły

Wybrany przez nas framework SPADE wspiera **FIPA**. Wiadomości wysyłane przez agenty mają strukturę wiadomości FIPA (performatywę, nadawcę, odbiorcę, treść wiadomości, język, ontologię).

Napotkane problemy

Rejestracja kilku użytkowników na serwerze XMPP (*Prosody*).

Problem ten pojawił się w kontekście tego, że używany serwer uruchamiamy za pomocą *docker compose* w kontenerze. Ustawienie odpowiednich zmiennych środowiskowych dozwala bowiem wyłącznie na rejestrację jednego użytkownika. Jednocześnie modyfikacja *entrypointu* kontenera uniemożliwiała jego prawidłowe działanie. W celu rozwiązania problemu napisaliśmy plik wsadowy, który pobiera id kontenera i tam *ręcznie* rejestruje wielu użytkowników za pomocą *docker exec*.

Działanie systemu

Założenia

1. Pasażer został zasymulowany
2. Położenie agentów RoutingBus i Passenger oraz cel podróży agenta Passenger losowane są z rozkładu ciągłego z przedziału [0, 100]
3. Aby zasymulować awarię autobusu i rezygnację pasażera program cyklicznie losuje z określonym prawdopodobieństwem wystąpienie tych scenariuszy alternatywnych.
4. W celach demonstracyjnych istnieje wyłącznie 1 agent Passenger oraz 2 obiekty RoutingBus, pomimo że z założenia system może obsługiwać dowolną ich liczbę.
5. Domyślne (początkowe) trasy obu instancji RoutingBus są stałe i wynoszą odpowiednio: [[0, 0], [20, 20], [40, 40], [60, 60], [80, 80]] oraz [[100, 0], [80, 20], [60, 40], [40, 60], [20, 80]].

Uruchomienie

Windows

```
docker compose up -d

./start.bat

python .\dynamic-timetable\main.py
```

Linux

```
docker compose up -d

container_id=$(docker compose ps -q)

docker exec -it $container_id prosodyctl register scheduler localhost scheduler

docker exec -it $container_id prosodyctl register passenger
 localhost passenger

docker exec -it $container_id prosodyctl register routing_bus1
 localhost routing_bus1

docker exec -it $container_id prosodyctl register routing_bus2
 localhost routing_bus2
```

Działanie

Cały proces zaczyna się od wybrania przez pasażera koordynatów, do których chce się udać. Następnie główny system (Scheduler) przypisuje dla pasażera odpowiedni bus, którym może udać się w wyznaczone miejsce. Oczywiście w celach prezentacyjnych jesteśmy ograniczeni do założeń opisanych powyżej. Istnieje również możliwość awarii busa oraz rezygnacji pasażera z przejazdu. Poniżej przedstawiony został zrzut ekranu przedstawiający logi z działania aplikacji.

```
2023-12-29 17:13:07,374 - AASD - [DEBUG] - Scheduler: Message sent
2023-12-29 17:13:07,374 - AASD - [DEBUG] - Scheduler: ReceiveBusPropose running
2023-12-29 17:13:07,374 - AASD - [INFO] - RoutingBus routing_bus1@localhost: Message received with content: {"passenger_info":
 {"passenger_jid": ["passenger", "localhost", null], "start_point": [89, 63], "destination": [22, 31]}}
2023-12-29 17:13:07,374 - AASD - [INFO] - RoutingBus routing_bus2@localhost: Message received with content: {"passenger_info":
 {"passenger_jid": ["passenger", "localhost", null], "start_point": [89, 63], "destination": [22, 31]}}
2023-12-29 17:13:07,374 - AASD - [DEBUG] - RoutingBus routing_bus1@localhost: GetBusInformation running
2023-12-29 17:13:07,374 - AASD - [INFO] - RoutingBus routing_bus1@localhost: Position: [10, 24]
2023-12-29 17:13:07,374 - AASD - [DEBUG] - RoutingBus routing_bus1@localhost: ReceivePassengerMsg running
2023-12-29 17:13:07,374 - AASD - [DEBUG] - RoutingBus routing_bus2@localhost: GetBusInformation running
2023-12-29 17:13:07,374 - AASD - [INFO] - RoutingBus routing_bus2@localhost: Position: [44, 83]
2023-12-29 17:13:07,374 - AASD - [DEBUG] - RoutingBus routing_bus2@localhost: ReceivePassengerMsg running
2023-12-29 17:13:07,374 - AASD - [DEBUG] - RoutingBus routing_bus1@localhost: CalculatePotentialCost running
2023-12-29 17:13:07,374 - AASD - [INFO] - RoutingBus routing_bus1@localhost: increase_in_length = 125
2023-12-29 17:13:07,374 - AASD - [DEBUG] - RoutingBus routing_bus1@localhost: potential_cost sent!
2023-12-29 17:13:07,374 - AASD - [DEBUG] - RoutingBus routing_bus2@localhost: CalculatePotentialCost running
2023-12-29 17:13:07,374 - AASD - [INFO] - RoutingBus routing_bus2@localhost: increase_in_length = 115
2023-12-29 17:13:07,374 - AASD - [DEBUG] - RoutingBus routing_bus2@localhost: potential_cost sent!
2023-12-29 17:13:07,374 - AASD - [DEBUG] - RoutingBus routing_bus1@localhost: WaitForDecision running
2023-12-29 17:13:07,374 - AASD - [DEBUG] - RoutingBus routing_bus2@localhost: WaitForDecision running
2023-12-29 17:13:07,374 - AASD - [INFO] - Scheduler: ReceiveBusPropose - Message received with content: {"id": "routing_bus1@l
ocalhost", "potential_cost": 125}
2023-12-29 17:13:07,374 - AASD - [INFO] - Scheduler: ReceiveBusPropose - Message received with content: {"id": "routing_bus2@l
ocalhost", "potential_cost": 115}
2023-12-29 17:13:07,389 - AASD - [DEBUG] - Scheduler: SelectBus running
2023-12-29 17:13:07,389 - AASD - [DEBUG] - Scheduler: costs = {'routing_bus1@localhost': 125, 'routing_bus2@localhost': 115}
2023-12-29 17:13:07,390 - AASD - [INFO] - Scheduler: selected routing_bus2@localhost with cost 115
2023-12-29 17:13:07,392 - AASD - [DEBUG] - Scheduler: ReplyBus running
2023-12-29 17:13:07,392 - AASD - [DEBUG] - Scheduler: Message sent!
2023-12-29 17:13:07,393 - AASD - [DEBUG] - Scheduler: SendTravelPlan running
2023-12-29 17:13:07,394 - AASD - [INFO] - Scheduler: SEND_TRAVELPLAN - Message sent!
2023-12-29 17:13:07,395 - AASD - [DEBUG] - Scheduler: ReceiveTravelRequest running
2023-12-29 17:13:07,395 - AASD - [INFO] - RoutingBus routing_bus2@localhost: New route accepted
2023-12-29 17:13:07,396 - AASD - [DEBUG] - RoutingBus routing_bus2@localhost: CalculateRoute running
2023-12-29 17:13:07,396 - AASD - [INFO] - RoutingBus routing_bus2@localhost: new_path = [[100, 0], [80, 20], [89, 63], [60, 40
], [40, 60], [20, 80], [22, 31]]
```

Jak widać posiadamy informacje o czasie wysyłanych komunikatów oraz ich treści. Jesteśmy w stanie zobaczyć również moment wyboru lepszego busa (bus nr 2), który posiada mniejszy koszt. Opis wyboru busa jest opisany poniżej.

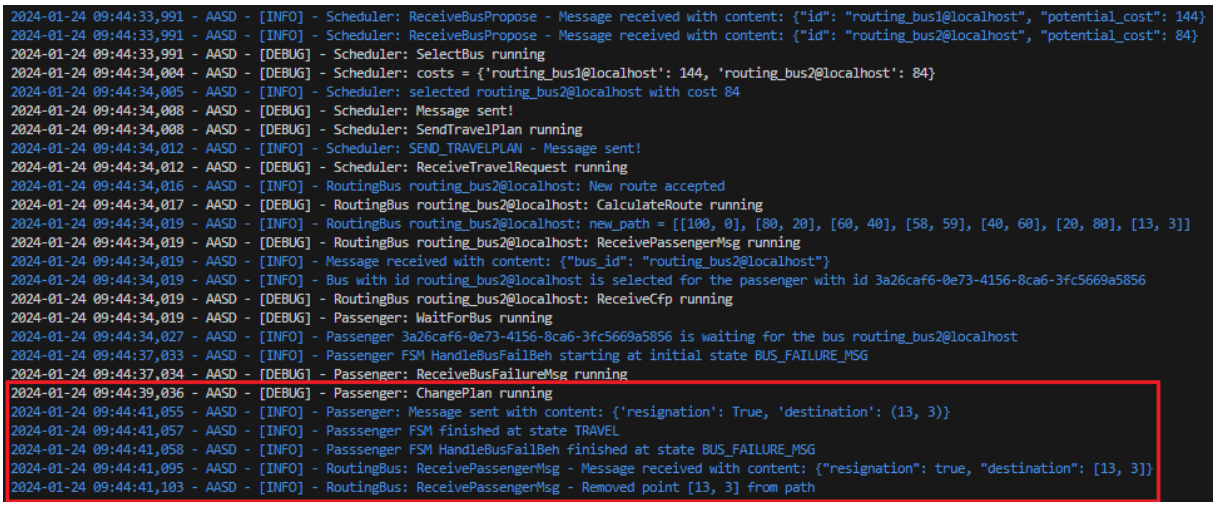
Zaimplementowaliśmy również opisane scenariusze alternatywne

Awaria autobusu

```
2024-01-24 09:36:13,460 - AASD - [DEBUG] - RoutingBus routing_bus2@localhost: ReceivePassengerMsg running
2024-01-24 09:36:13,462 - AASD - [INFO] - Message received with content: {"bus_id": "routing_bus2@localhost"}
2024-01-24 09:36:13,464 - AASD - [INFO] - Bus with id routing_bus2@localhost is selected for the passenger with id f06e4163-bb8b-453d-9f98-607e064a
b480
2024-01-24 09:36:13,466 - AASD - [DEBUG] - RoutingBus routing_bus2@localhost: ReceiveCfp running
2024-01-24 09:36:13,468 - AASD - [DEBUG] - Passenger: WaitForBus running
2024-01-24 09:36:13,471 - AASD - [INFO] - Passenger f06e4163-bb8b-453d-9f98-607e064ab480 is waiting for the bus routing_bus2@localhost
2024-01-24 09:36:16,488 - AASD - [INFO] - Passenger FSM HandleBusFailBeh starting at initial state BUS_FAILURE_MSG
2024-01-24 09:36:16,490 - AASD - [DEBUG] - Passenger: ReceiveBusFailureMsg running
2024-01-24 09:36:18,503 - AASD - [INFO] - Passenger: ReceiveBusFailureMsg - Bus broke down during the trip.
2024-01-24 09:36:18,505 - AASD - [DEBUG] - Passenger: ChangePlan running
2024-01-24 09:36:20,528 - AASD - [INFO] - Passsenger FSM finished at state TRAVEL
2024-01-24 09:36:20,530 - AASD - [DEBUG] - Passenger: HandleBusFailure running
2024-01-24 09:36:20,532 - AASD - [INFO] - Passenger f06e4163-bb8b-453d-9f98-607e064ab480 is starting looking for a next bus from new startng locati
on: (75, 74)
2024-01-24 09:36:20,538 - AASD - [INFO] - Passenger FSM starting at initial state SELECT_DESTINATION
2024-01-24 09:36:20,539 - AASD - [DEBUG] - Passenger: SelectDestination running
2024-01-24 09:36:20,543 - AASD - [INFO] - Passenger f06e4163-bb8b-453d-9f98-607e064ab480 is selecting destination
2024-01-24 09:36:20,546 - AASD - [INFO] - Passsenger FSM HandleBusFailBeh finished at state HANDLE_BUS_FAILURE
2024-01-24 09:36:20,550 - AASD - [DEBUG] - Passenger: RequestForTravel running
2024-01-24 09:36:20,556 - AASD - [INFO] - Passenger: Message sent with content: {"start_point": [75, 74], "destination": [89, 30]}
2024-01-24 09:36:20,557 - AASD - [DEBUG] - Passenger: AwaitTravelPlan running
2024-01-24 09:36:20,559 - AASD - [INFO] - RoutingBus routing_bus1@localhost: New route not accepted
2024-01-24 09:36:20,563 - AASD - [DEBUG] - RoutingBus routing_bus1@localhost: ReceiveCfp running
2024-01-24 09:36:20,565 - AASD - [INFO] - Scheduler: ReceiveTravelRequest - Message received with content: {"start_point": [75, 74], "destination":
 [89, 30]}
2024-01-24 09:36:20,571 - AASD - [DEBUG] - Scheduler: SavePassengerInfo running
2024-01-24 09:36:20,578 - AASD - [DEBUG] - Scheduler: Cfp running
2024-01-24 09:36:20,589 - AASD - [DEBUG] - Scheduler: Message sent
2024-01-24 09:36:20,594 - AASD - [DEBUG] - Scheduler: Message sent
```

Logi związane z awarią autobusu znajdują się w czerwonym prostokącie. Pasażer odbiera wiadomość o awarii od autobusu. Następnie decyduje się ponownie skorzystać z usług systemu i ponownie wysyła zgłoszenie chęci przejazdu do Schedulera i cały proces rozpoczyna się na nowo.

Zmiana planów pasażera



Logi związane ze zmianą planów pasażera znajdują się w czerwonym prostokącie. Widać, że pasażer wysyła wiadomość o rezygnacji do odpowiedniego Routing Busa, który to następnie usuwa ze swojej trasy punkt startowy i początkowy pasażera. Należy również zaznaczyć, że Routing Bus w swojej trasie przechowuje punkty startu i docelowe podróży każdego pasażera. Dzięki czemu, w przypadku, jeżeli Routing Bus posiadał kilku pasażerów, którzy chcieli wsiąść lub wysiąść w tym samym miejscu, w którym rezygnujący pasażer, to zostanie usunięty tylko punkt pasażera rezygnującego. Pozostali pasażerowie będą mogli wsiąść lub wysiąść w swoich miejscach.

Algorytmy

Najbardziej zaawansowaną algorytmicznie częścią naszego projektu jest **oszacowanie kosztu zmiany trasy autobusu** w oparciu o dwa nowe punkty i jej równoległe **wyznaczenie**.

Aby rozwiązać klasycznie ten problem, każdy RoutingBus powinien przeszukać wszystkie możliwe kombinacje trasy z nowymi punktami, a następnie wybrać wariant z najmniejszą zmianą długości trasy. Należy, jednakże zauważyć, że złożoność obliczeniowa tego algorytmu byłaby dosyć wysoka, szczególnie dla dłuższych ścieżek.

Tak więc, zdecydowaliśmy się użyć **heurystyki najbliższych sąsiadów**. Mianowicie dodajemy każdy nowy punkt między dwa istniejące punkty, które są do niego najbliższe (zgodnie z metryką Manhattan). Osobno rozważamy przypadki, gdy nowy punkt powinien się znaleźć na początku albo na końcu trasy. W oparciu o tak zawężoną przestrzeń poszukiwań obliczamy koszt zmiany trasy, a ewentualnie także wybieramy tak właśnie wyznaczoną nową trasę. W tym kontekście należy zauważyć, że dotychczasowa trasa powinna być optymalna lub też quasi-optymalna, co biorąc pod uwagę wyznaczanie w ten sposób trasy począwszy od pustego zbioru punktów, jest założeniem poprawnym.

Lista kroków tego heurystycznego algorytmu przestawia się następująco:

1. **Sprawdzenie Pustej Ścieżki**
    a. Jeśli początkowa ścieżka jest pusta, algorytm kończy działanie, zwracając nowe punkty jako ścieżkę i 0 jako wzrost długości.
2. **Wstawienie Pierwszego Nowego Punktu**
    a. Dla pierwszego nowego punktu, algorytm szuka optymalnego miejsca wstawienia na ścieżce.
    b. Sprawdza możliwość wstawienia punktu na początek ścieżki.

c. Iteruje przez wszystkie istniejące pary punktów na ścieżce, rozważając wstawienie nowego punktu między nimi.
d. Sprawdza możliwość wstawienia punktu na koniec ścieżki.
e. Wybiera miejsce, które minimalizuje wzrost długości ścieżki po dodaniu punktu.
f. Wstawia punkt w wybrane miejsce na ścieżce.

3. **Wstawienie Drugiego Nowego Punktu**
   a. Dla drugiego nowego punktu, algorytm ponownie szuka optymalnego miejsca wstawienia, zaczynając od miejsca tuż za wstawionym pierwszym punktem.
   b. Powtarza proces sprawdzania możliwości wstawienia punktu między istniejącymi punktami na ścieżce oraz na jej końcu, podobnie jak dla pierwszego punktu.
   c. Wybiera miejsce dla drugiego punktu, które minimalizuje wzrost długości ścieżki po jego dodaniu, z zastrzeżeniem, że musi być on wstawiony po pierwszym punkcie.
   d. Wstawia drugi punkt w wybrane miejsce na ścieżce.
4. **Obliczenie Całkowitego Wzrostu Długości Ścieżki**
   a. Algorytm sumuje wzrosty długości ścieżki spowodowane dodaniem obu nowych punktów.
5. **Zwrócenie Wyników**
   a. Na koniec algorytm zwraca nową ścieżkę, uwzględniającą oba dodane punkty, oraz całkowity wzrost długości ścieżki.

Integracja systemu

Elementy, których zabrakło w poprzednich częściach

Do systemu względem poprzednich części dodaliśmy testy jednostkowe i integracyjne. Poza tym, zgodnie z ustaleniami, system nie zawiera nowych elementów takich jak nowe agenty, czy komunikaty.

Opis integracji systemu

Serwer XMPP (*Prosody*) jest uruchamiany za pomocą narzędzia *docker compose*. Rejestracja użytkowników (tu: agentów) może być przeprowadzona automatycznie poprzez wywołanie pliku wsadowego *start.bat*. Tak uruchomiony i skunfigurowany serwer XMPP jest dostępny w domenie *localhost*. Rozpoczęcie symulacji odbywa się za pomocą wywołania *python .\dynamic-timetable\main.py*. Dostarczone skrypty zapewniają integrację cyklu życia agentów zgodnie z poniższą tabelą:

Agent	Czynność
Routing Bus 1	Start
Routing Bus 2	Start
Scheduler	Start
Passenger	Start
Scheduler	Oczekiwanie na zakończenie
Scheduler	Stop
Passenger	Stop
Routing Bus 1	Stop
Routing Bus 2	Stop

W ramach powyższego cyklu każdy agent asynchronicznie wysyła i odbiera komunikaty oraz zmienia swoje stany, co zostało szczegółowo omówione w rozdziale *Implementacja systemu: Implementacja agentów*.

Braki i napotkane problemy

Tak jak w poprzedniej części w celu przedstawienia **najważniejszej funkcjonalności systemu** implementujemy wyłącznie agentów: **Passenger**, **Scheduler** oraz **RoutingBus**. Ich integracja pozwala zademonstrować cykl działania począwszy od **zgłoszenia przejazdu**, przez **wybór autobusu do realizacji trasy**, aż po **powiadomienie pasażera o wyznaczeniu pojazdu**. Tym samym agent **DrivingBus** nie został zaimplementowany, przez co autobusy nie przemieszczają się. Niemniej jest to zgodne z ustaleniami.

Testy

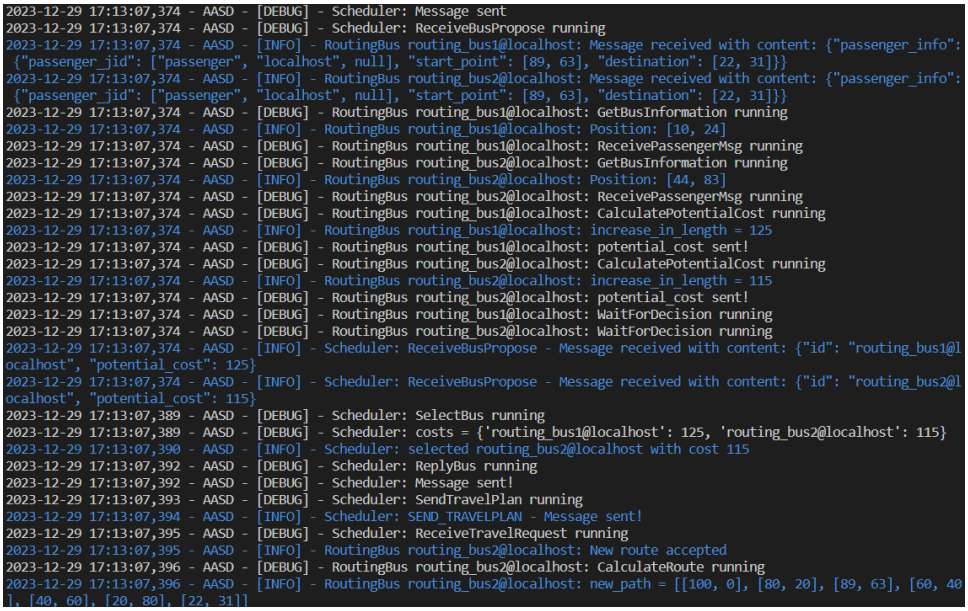
Opis i wyniki przeprowadzonych testów przedstawiają się zgodnie z poniższą tabelką:

Nazwa	Liczba zaliczonych (*passed*)	Liczba wszystkich	Opis
Jednostkowe	23	23	Testują działanie pojedynczych stanów w cyklach życia agentów.
Integracyjne	1	1	Wykrywa błędy w agentach w całym ich cyklu życia podczas komunikacji ze sobą.

Case studies

Scenariusz główny

Cały proces zaczyna się od wybrania przez pasażera koordynatów, do których chce się udać. Następnie główny system (Scheduler) przypisuje dla pasażera odpowiedni bus, którym może udać się w wyznaczone miejsce. Oczywiście w celach prezentacyjnych jesteśmy ograniczeni do założeń opisanych powyżej. Istnieje również możliwość awarii busa oraz rezygnacji pasażera z przejazdu. Poniżej przedstawiony został zrzut ekranu przedstawiający logi z działania aplikacji.

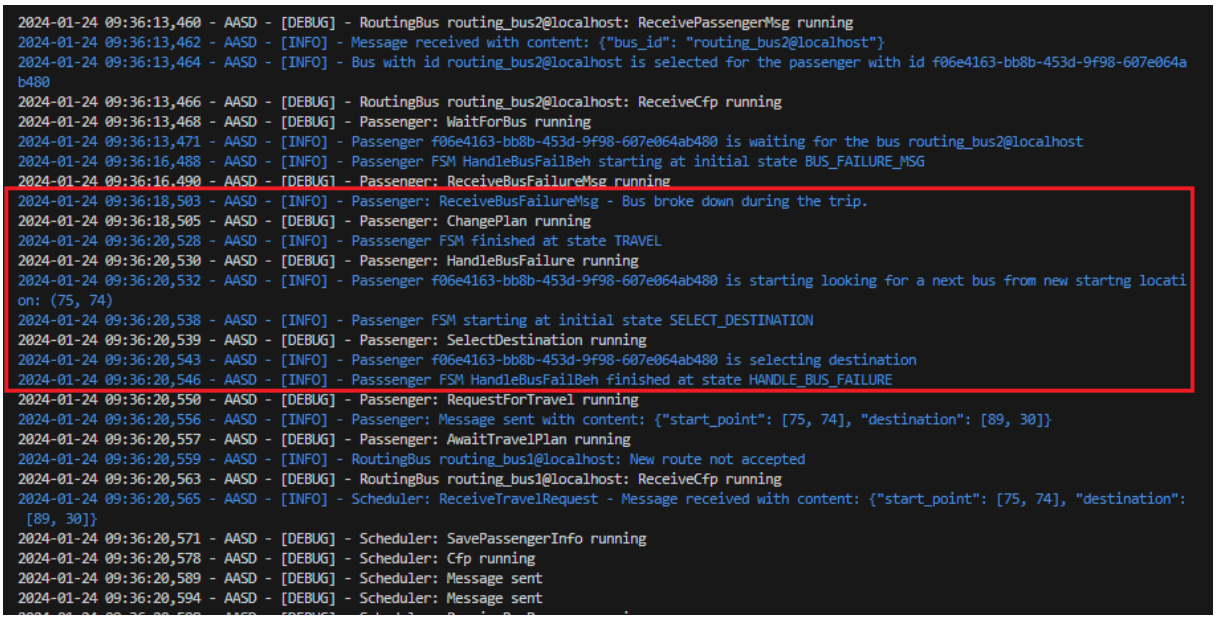


Jak widać posiadamy informacje o czasie wysyłanych komunikatów oraz ich treści. Jesteśmy w stanie zobaczyć również moment wyboru lepszego busa (bus nr 2), który posiada mniejszy koszt. Opis wyboru busa jest opisany poniżej.

Scenariusze alternatywne

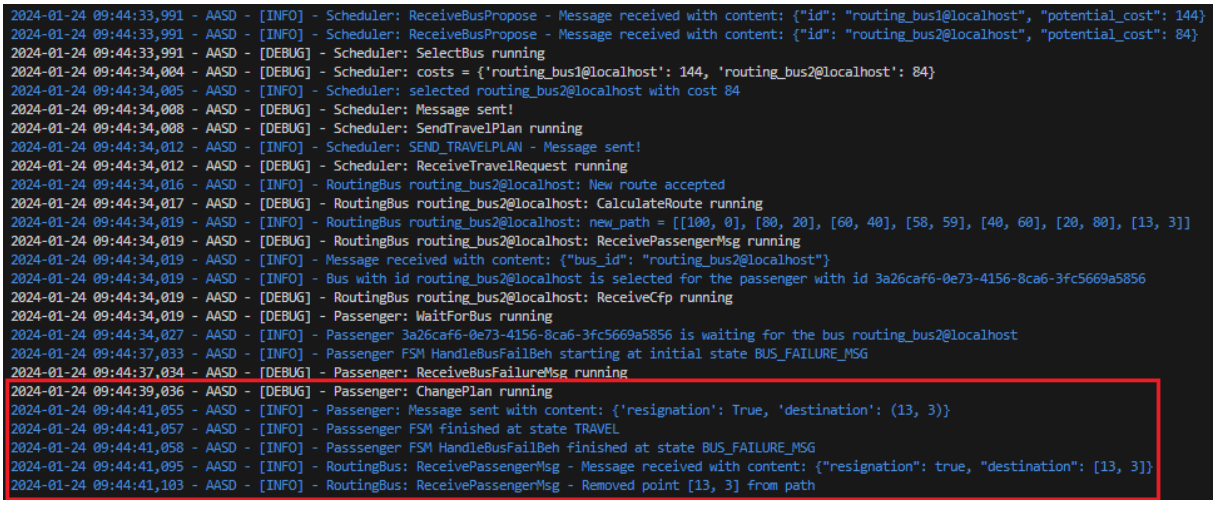
Zaimplementowaliśmy również opisane scenariusze alternatywne

Awaria autobusu



Logi związane z awarią autobusu znajdują się w czerwonym prostokącie. Pasażer odbiera wiadomość o awarii od autobusu. Następnie decyduje się ponownie skorzystać z usług systemu i ponownie wysyła zgłoszenie chęci przejazdu do Schedulera i cały proces rozpoczyna się na nowo.

Zmiana planów pasażera



Logi związane ze zmianą planów pasażera znajdują się w czerwonym prostokącie. Widać, że pasażer wysyła wiadomość o rezygnacji do odpowiedniego Routing Busa, który to następnie usuwa ze swojej trasy punkt startowy i początkowy pasażera. Należy również zaznaczyć, że Routing Bus w swojej trasie przechowuje punkty startu i docelowe podróży każdego pasażera. Dzięki czemu, w przypadku, jeżeli Routing Bus posiadał kilku pasażerów, którzy chcieli wsiąść lub wysiąść w tym samym miejscu, w

którym rezygnujący pasażer, to zostanie usunięty tylko punkt pasażera rezygnującego. Pozostali pasażerowie będą mogli wsiąść lub wysiąść w swoich miejscach.

Czas oczekiwania na przydzielenie busa

Liczba pasażerów	Liczba autobusów	Czas oczekiwania pasażera na przydzielenie busa [s]	Max czas oczekiwania [s]	Średni czas oczekiwania [s]
4	2	[[0.047], [0.039], [0.038], [0.048]]	0.048	0.043
8	2	[[0.089], [0.042], [0.04], [0.036], [0.035], [0.041], [0.04], [0.045]]	0.089	0.046
12	2	[[0.076], [0.073], [0.093], [0.099], [0.096], [0.131], [0.075], [0.128], [0.144], [0.134], [0.145], [0.473]]	0.473	0.139
20	2	[[0.124], [0.073], [0.304], [0.124], [0.11], [0.116], [0.141], [0.074], [0.093], [0.12], [0.116], [0.054], [0.133], [0.071], [0.131], [0.134], [0.131], [0.142], [0.142], [0.058]]	0.304	0.120
8	5	[[0.119], [0.128], [0.224], [0.24], [0.132], [0.127], [0.226], [0.219]]	0.240	0.177
12	5	[[0.193], [0.236], [0.138], [0.199], [0.298], [0.144], [0.195], [0.217], [0.157], [0.23], [0.151], [0.183]]	0.298	0.195

Przeprowadzono kilka scenariuszy z różną liczbą pasażerów i autobusów. Jak widać czas oczekiwania pasażera na przydzielenie autobusu jest krótki – mniejszy niż 1 sekunda. Widać, że system spełnia wymagania dotyczące szybkości odpowiedzi i efektywności, zapewniając sprawnie działający przydział autobusów, co przekłada się na zadowolenie użytkowników i optymalne wykorzystanie zasobów.

Należy również wziąć pod uwagę, że w przedstawionych scenariuszach użyliśmy symulacji działającej na 1 urządzeniu. W prawdziwych warunkach obliczenie kosztu zmiany trasy odbywałoby się na różnych urządzeniach dzięki czemu byłoby naprawdę zrównoleglone.

Wnioski

1. Wykorzystując paradygmat agentowy, z powodzeniem udało się zrealizować projekt systemu komunikacji miejskiej korzystającej z dynamicznego rozkładu jazdy.
2. Pozytywnie oceniamy projektowanie systemu polegające najpierw na wykorzystaniu metody *Gaia*, po to aby w spójny przejść do modelu *BPMN*.
3. Framework *Spade* w łatwy i przyjazny sposób pozwala zamodelować system wieloagentowy, dając przy tym jednocześnie duże możliwości jego specjalizacji.
4. Paradygmat agentowy z powodzeniem nadaje się do modelowania przynajmniej części problemów, które wcześniej rozwiązywaliśmy wieloprocesowo (względnie wielowątkowo).