Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Wieloagentowe Systemy Decyzyjne

SmartPark

Jan Dubiński | 271165
Joanna Kaleta | 271181
Aleksander Ogonowski | 267381
Maria Oniszuczuk | 271199
Kornel Szymczyk | 267778

WARSZAWA 2019

repozytorium git: `https://github.com/Jot-De/WSD/`

Spis treści

1. Opis problemu

Coraz większym problemem rozwijających się aglomeracji staje się nadmierne zagęszczenie ruchu samochodowego, a co się z tym wiąże - trudność w znalezieniu miejsc parkingowych. Brak łatwo dostępnej informacji o wolnych miejscach do parkowania zarówno na pobliskich parkingach, jak i w przestrzeni publicznej prowadzi do irytacji kierowców oraz marnowania ich czasu. Jedynym sposobem aby przekonać się, że wybrany parking jest w pełni obłożony, jest pojawienie się tam osobiście. Kierowcy w poszukiwaniu miejsc parkingowych krążą po ulicach dodatkowo potęgując korki uliczne. Stając w miejscach publicznych, w których parkowanie jest niedozwolone, dodatkowo utrudniają ruch pieszym i pozostałym kierowcom, jednocześnie narażając się na nieprzyjemne konsekwencje finansowe. Naszą propozycją mającą ułatwić parkowanie i zredukować ruch uliczny jest system wieloagentowy.

2. Słowny opis koncepcji systemu

Proponowanym przez nas rozwiązaniem problemu omówionego w punkcie pierwszym jest system wieloagentowy ułatwiający znalezienie miejsca parkingowego, który pozwoli na znaczną redukcję liczby krążących w jego poszukiwaniu samochodów.

System będzie składać się z sieci parkingów oraz aplikacji mobilnej dla kierowców pojazdów. Przy wykorzystaniu aplikacji, użytkownik będzie mógł zrealizować wyszukanie oraz nawigację do wybranego pobliskiego parkingu po wcześniejszym zgłoszeniu swojej lokalizacji.

System po zgłoszeniu chęci zaparkowania proponuje pięć pobliskich parkingów z wolnymi miejscami, spośród których kierowca może wybrać ten do którego chce się udać. Kluczowe w systemie jest pozwolenie kierowcy na znalezienie jak najbliższego odpowiadającego mu parkingu z przynajmniej jednym wolnym miejscem. Celem parkingu jest natomiast pozbycie się pustych miejsc, czyli jego zapełnienie.

Każde miejsce parkingowe powinno zostać wyposażone w czujniki ultradźwiękowe wykrywające zajętość miejsca oraz kamerę identyfikującą konkretny pojazd. W przypadku zwolnienia miejsca powinna podnosić się blokada uniemożliwiająca wjazd niezgłoszonemu na dany parking wsamochodowi. Każdy parking agreguje konkretną liczbę miejsc postojowych na przykład na odcinku konkretnej ulicy i posiada informacje o ich zajętości. Każde miejsce parkingowe po zwolnieniu wysyła adekwatną informację do parkingu, do którego przynależy, dzięki czemu może on zgłosić chęć przyjęcia następnego kierowcy. Gdy kierowca zadeklaruje chęć zaparkowania, po czym jest nawigowany do określonego parkingu, powoduje to zwiększenie się jego zajętości. Parking oczekuje na kierowcę przez pewien ustalony czas, nie przyjmując kolejnych deklarujących chęć pojazdów, jeżeli nie ma w swoim obrębie wolnych miejsc. Dzięki takiemu rozwiązaniu mamy pewność, że po dojechaniu na parking znajdziemy puste miejsce. Gdy kierowca podjedzie do miejsca na przydzielonym parkingu i zostanie poprawnie zidentyfikowany przez kamerę poprzez numer rejestracyjny, blokada opuszcza się, kierowca parkuje na miejscu postojowym, a miejsce informuje parking o jego prawidłowym zajęciu, zwiększając zajętość jego aż do chwili opuszczenia miejsca przez pojazd. Przydzielony parking docelowo powinien być oznaczony np. przy wykorzystaniu Open Street Map jako region do którego prowadzony jest kierowca.

Podczas korzystania z aplikacji można wydzielić kilka typowych scenariuszy:
Scenariusz 1 - główny - wyszukanie miejsca parkingowego:

1. Użytkownik deklaruje w aplikacji chęć znalezienia miejsca parkingowego.
2. System wyszukuje parkingi z przynajmniej jednym wolnym miejscem w okolicy użytkownika komunikując się z pobliskimi parkingami.
3. System proponuje parkingi, które użytkownik może zaakceptować.
4. Użytkownik wybiera w aplikacji jeden proponowanych przez system parking.
5. System nawiguje kierowcę do parkingu.
6. Kierowca stawia się na dowolnym miejscu parkingowym w obrębie parkingu

Scenariusz 2 - użytkownik oddala się od parkingu

1-5. Jak w scenariuszu głównym
6. Kierowca nie kieruje się na parking (jego pozycja oddala się).
7.Parking sprawdza czy proces postępuje
8.Kierowca nadal się oddala
9. Parking dokonuje zwolnienia miejsca parkingowego

Scenariusz 3 - alternatywny do scenariusza 1 - użytkownik odrzuca proponowane parking

1-4. Jak w scenariuszu 2
5. Użytkownik nie wybiera zasugerowanego przez system parkingu
6. Kierowca oddala się o pewną odległość
7. Powrót do punktu 1 scenariusza głównego

Scenariusz 4 - użytkownik rezygnuje z rezerwacji

1-5. Jak w scenariuszu głównym
6. Kierowca rezygnuje z miejsca parkingowego
7. Parking dokonuje zwolnienia miejsca parkingowego

Scenariusz 5 - brak wolnych miejsc parkingowych

1-3. Jak w scenariuszu głównym
4. Aplikacja wyświetla komunikat o braku wolnych miejsc parkingowych na każdym typie parkingu.
5. Kierowca oddala się o pewną odległość
6. Powrót do punktu 1 scenariusza głównego.

3. Architektura systemu

Celem nadrzędnym systemu jest maksymalne zapełnienie parkingów.

System opiera się na dwóch rodzajach agentów

- kierowców samochodów z zainstalowaną aplikacją
- parkingów agregujących miejsca parkingowe

Kierowcy dążą do znalezienie parkingu jak najbliżej aktualnego miejsca. Parkingi dążą do maksymalnego zapełnienie się. System wysyła kierowcy propozycję miejsca parkingowego na parkingu, który znajduje się najbliżej jego aktualnego położenia i który posiada wolne miejsca. Po zaakceptowaniu przesłanej propozycji następuje zarezerwowanie miejsca postojowego dla klienta na podstawie jego unikalnego identyfikatora.

System wymaga, aby samochód cyklicznie wysyłał zapytania do parkingów otrzymując w odpowiedzi ich położenie.

System uzgadniania miejsc nie może dopuścić do sytuacji gdy przy jednoczesnym zgłoszeniu chęci parkowania przez dwa lub więcej samochody zostaną przydzielone dwa samochody na jedno wolne miejsce parkingowe.

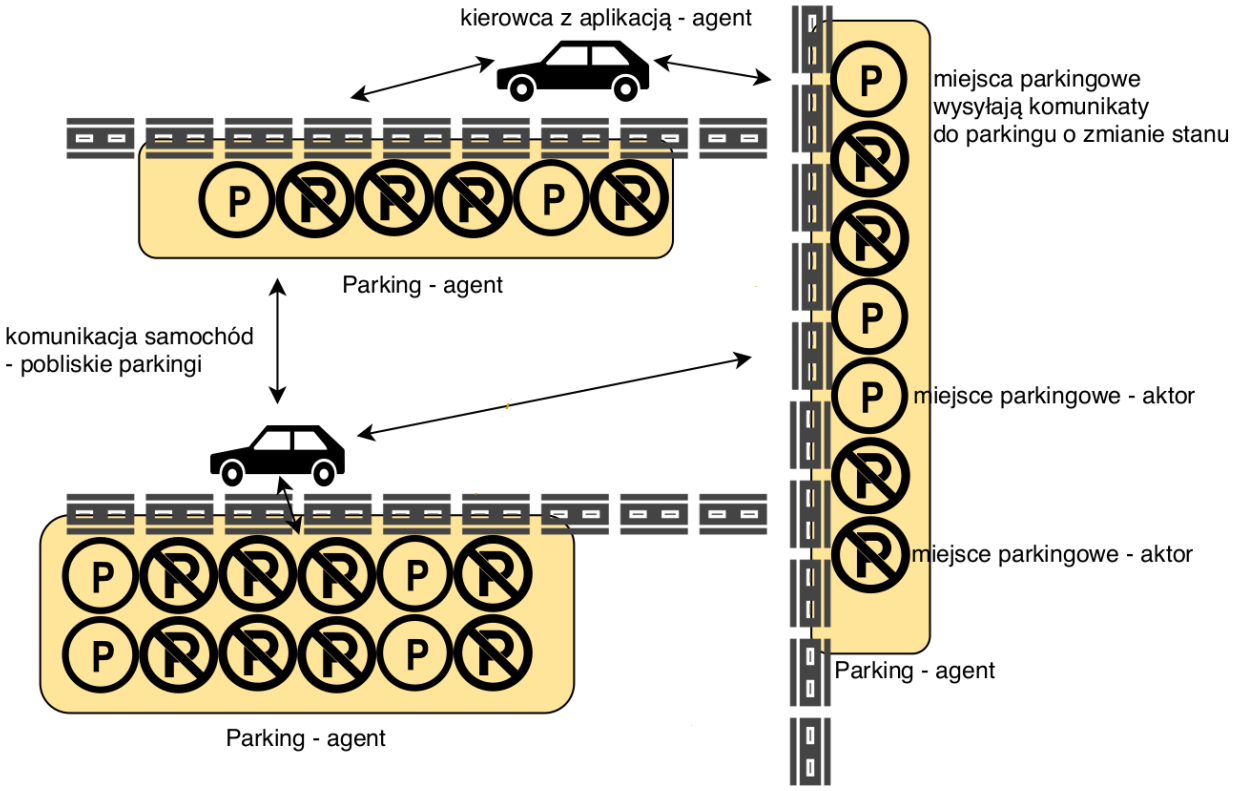
Parking nadzoruje, czy kierowca nie oddala się od niego po zgłoszeniu chęci zaparkowania przez dłuższy czas lub nie wysłał zgłoszenia dotyczącego anulowania rezerwacji. W przypadku wystąpienia jednego z tych zdarzeń parking odwołuje rezerwację zwalniając przydzielone dla samochodu miejsce.

Miejsca parkingowe są natomiast aktorami będącymi nieaktywnymi przez większość czasu. Ich zadanie polega jedynie na wysyłaniu komunikatów do parkingu, do którego przynależą. Jest to komunikat o zajęciu i zwolnieniu miejsca przez kierowcę. Na tej podstawie parking wie ile samochodów znajduje się na nim, a ile może przyjąć. Od liczby samochodów, które parking może przyjąć należy odjąć również samochody, które zgłosiły chęć parkowania, ale jeszcze nie dojechały. Nigdy jednak nie zostaje przyjęte więcej samochodów niż liczba wolnych miejsc.

Kierowcy komunikują się z parkingami zgłaszając chęć parkowania lub odrzucając sugerowany parking. Generalnie nie ma potrzeby, aby samochody prowadziły komunikację między sobą.

Naszymi propozycjami rozwoju systemu w kolejnych etapach jest:

- dodanie nadzoru nad czujnikami - sygnalizowanie awarii i prowadzenie napraw
- udoskonalenie algorytmu monitorującego zachowanie kierowcy tak, aby uwzględniał on korki, objazdy, wypadki itd.



Rysunek 1. Architektura systemu.

4. Projekt systemu wieloagentowego

System SmartPark został zaprojektowany zgodnie z wytycznymi metodologii GAIA.

4.1. Identyfikacja ról

Role agenta CarAgent:

1. Client
   - zgłasza chęć zaparkowania w pobliżu udostępnionej lokalizacji
   - akceptuje lub odrzuca propozycje wskazanego miejsca
2. Parking mapper
   - mapuje istniejące parkingi
3. Approacher
   - zbliża się do parkingu i wysyła komunikat z aktualną lokalizacją
   - może zrezygnować z zarezerwowanego miejsca

Role agenta ParkingAgent:

4. Parking Manager
   - kontrola dostępności miejsc na parkingu (stanu wewnętrznego),
   - przydziela miejsce parkingowe na parkingu, jeśli takie jest dostępne
5. Car Tracker
   - monitoruje samochód który zarezerwował miejsce (żeby zamknąć rezerwacje jeśli nie będzie się zbliżał lub nie przyjedzie przez 10 min)

4.2. Model ról

4.2.1. Parking Manager

Aktywności:

- CheckPlacesAvaliablility - sprawdzanie liczby wolnych miejsc na parkingu
- MakeReservation - wykonanie rezerwacji
- CancelReservation - zwolnienie rezerwację

Protokoły:

- MapParkings
- PlaceReservations
- FreePlace

Tabela 1. Rola Parking Manager.

Role schema: Parking Manager
Description: • Zwraca informacje o lokalizacji parkingu • Kontroluje dostępność miejsc na parkingu • Przydziela miejsce parkingowe na danym parkingu, jeśli takie jest dostępne
Protocols and Activities: CheckPlacesAvailability, MakeReservation, CancelReservation, MapParkings(SendCoordinates), PlaceReservations(SendAvailablePlaceInfo), FreePlace(ConfirmFreedPlace)
Permissions: reads: sensors_data generates: parking_location, free_place_count
Responsibilities: Liveness: PARKING MANAGER = [MapParkings]. (CheckPlacesAvailability. [SendAvailablePlaceInfo]. [MakeReservation]. [CancelReservation]. ConfirmFreedPlace)* MAPPARKINGS = SendCoordinates Safety: true

4.2.2. Car Tracker

Aktywności:

- ShouldMaintainReservation - podejmowanie decyzji o utrzymaniu rezerwacji (jeśli Approacher się oddala/nie pojawia, podniesienie flagi ApproacherIsNotComing)

Protokoły:

- TrackCar
- TrackReservation
- ReservationCancellation
- FreePlace

Tabela 2. Rola Car Tracker.

Role schema: Car Tracker
Description: • monitoruje samochód, który zarezerwował miejsce i podejmuje decyzję, czy należy rezerwacje zamknąć (jeśli np. samochód nie przybywa w ciągu kilku minut lub oddala się przez dłuższy czas)
Protocols and Activities: ShouldMaintainReservation, TrackReservation(ConfirmReservationInfo) TrackCar(SubscribeForClientLocation), ReservationCancellation(ConfirmCancellation), FreePlace(RequestSetPlaceFree)
Permissions: reads: approacher_info, car_location, reservation_cancellation, IsApproacher generates: ApproacherIsNotComing, ApproacherCancelledReservation
Responsibilities: Liveness: CAR TRACKER = TrackReservation. TrackCar. [ReservationCancellation]. FreePlace TRACKRESERVATION = ConfirmReservationInfo TRACKCAR = SubscribeForClientLocation. ShouldMaintainReservation RESERVATIONCANCELLATION = ConfirmCancellation FREEPLACE = RequestSetPlaceFree Safety: IsApproacher = True

4.2.3. Parking Mapper

Aktywności:

- CreateParkingDict - zapamiętuje lokalizacje opowiadające parkingom
- RunUpdate - włącza skanowanie parkingów w celu pobrania lokalizacji

Protokoły:

- MapParking

Tabela 3. Rola Parking Mapper.

Role schema: ParkingMapper
Description: • mapuje istniejące parkingi
Protocols and Activities: RunUpdate, MapParking(UpdateListOfParkings)
Permissions: reads: parking_location generates: parking_list
Responsibilities: Liveness: PARKINGMAPPER = (RunUpdate.MapParking.CreateParkingDict)ω MAPPARKING = UpdateListOfParkings Safety: true

4.2.4. Client

Aktywności:

- GetMyLocation - lokalizuje się
- ComputeParkingList - wybiera kilka parkingów w najbliższej odległości z wolnymi miejscami postojowymi, z którymi będzie się komunikować

Protokoły:

- PlaceReservation
- TrackReservation

Tabela 4. Rola Client.

Role schema: Client
Description: • zgłasza chęć zaparkowania w pobliżu udostępnionej lokalizacji • wybiera jeden z zaproponowanych parkingów lub odrzuca propozycje
Protocols and Activities: GetMyLocation, PlaceReservation(CallForParkingOffers, AcceptParkingOffer), TrackReservation(SendReservationInfo), ComputeParkingList
Permissions: reads: parking_list generates: IsApproacher, car_location, selected_parking_list
Responsibilities: Liveness: CLIENT = (GetMyLocation. ComputeParkingList. CallForParkingOffers. [AcceptParkingOffer]. TrackReservation)* TRACKRESERVATION = SendReservationInfo Safety: isApproacher = false

4.2.5. Approacher

Aktywności:

- ShouldParkingApproaching - podejmowanie decyzji czy chce zmierzać do parkingu i ewentualna jazda w jego stronę

Protokoły:

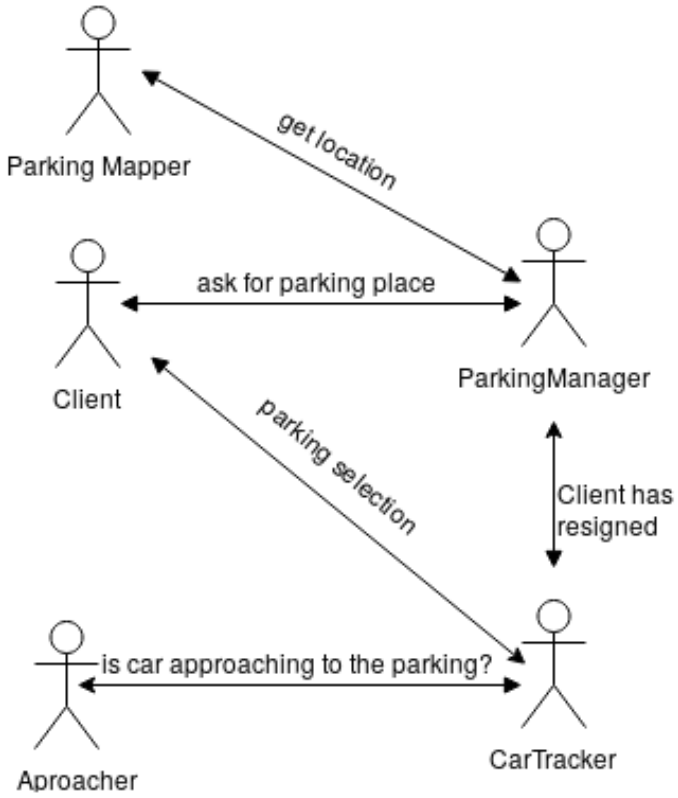
- TrackCar
- Reservation Cancellation
- FreePlace

Tabela 5. Rola Approacher.

Role schema: Approacher
Description: • ma zarezerwowane miejsce na parkingu i wysyła informację o swoim aktualnym położeniu (ciągle) • może zrezygnować z zarezerwowanego miejsca
Protocols and Activities: TrackCar(SendApproacherLocation), Reservation Cancellation(CancelClientReservation)
Permissions: reads: IsApproacher generates: reservation_cancellation, car_location
Responsibilities: Liveness: APPROACHER = TrackCar TRACKCAR = SendReservationInfo. (SendApproacherLocation | ReservationCancellation)ω RESERVATIONCANCELLATION = CancelClientReservation Safety: isApproacher = true

4.3. Model interakcji

Na rysunku 2 przedstawiającym model interakcji widać, że nie posiada on żadnej roli centralnej komunikującej się bezpośrednio ze wszystkimi innymi rolami. System posiada znaczne rozproszenie ról, biorąc pod uwagę występowanie tylko dwóch typów agentów wcielających się w różne role. Wszystkie agenty wchodzą między sobą w interakcje. Każdy z agentów powinien zostać więc poprawnie zaimplementowany w celu utrzymania systemu.



Rysunek 2. Model interakcji.

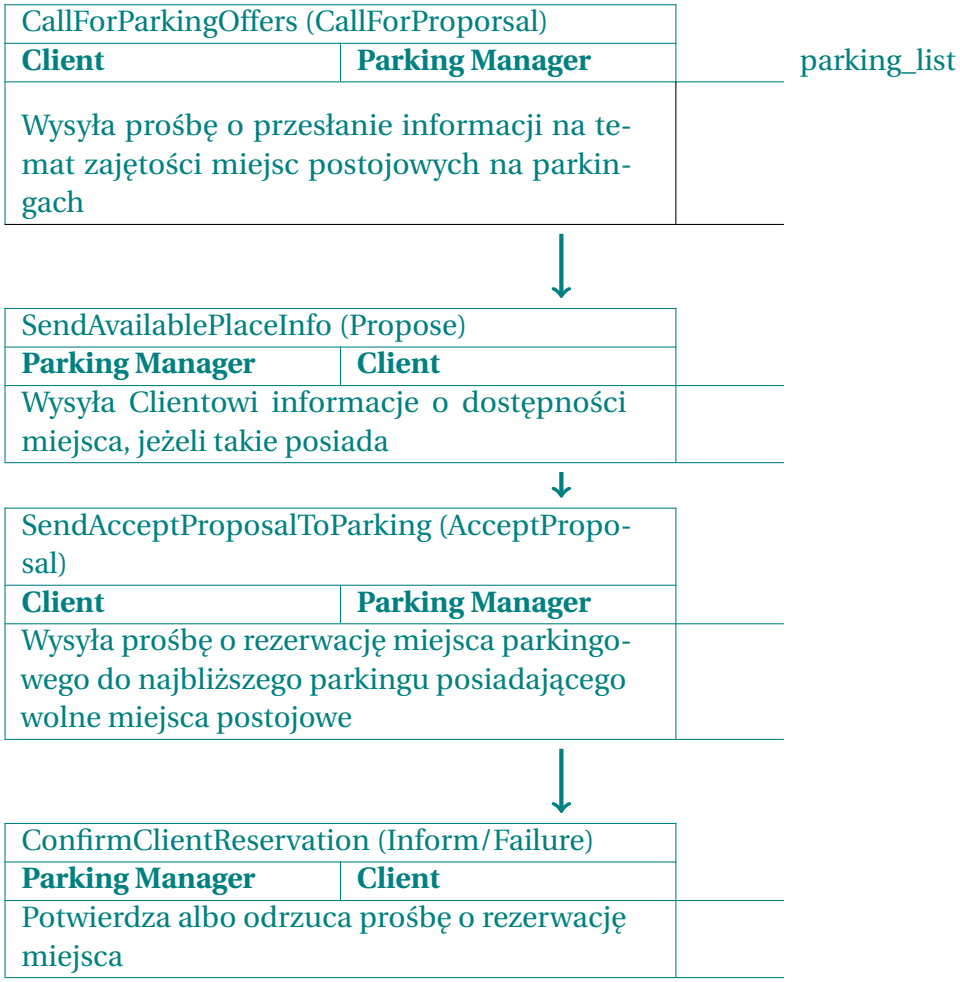
4.4. Definicje protokołów

4.4.1. Map Parkings

UpdateListOfParkings (INFORM REF)	
ParkingMapper	**ParkingManager**
Wysyła zapytania do wszystkich parkingów z prośbą o informacje o ich lokalizacji i uaktualnia ich położenie	

SendCoordinates (INFORM)		
ParkingManager	**ParkingMapper**	
wysyła koordynaty parkingu		parking_location

4.4.2. PlaceReservation



4.4.3. TrackReservation

SendReservationInfo (INFORM)	
Client	**CarTracker**
Przekazuje dane rezerwacji do Car Trackera po akceptowaniu wskazanego miejsce	

ConfirmReservationInfo (Confirm)	
Car Tracker	**Client**
Potwierdza otrzymanie informacji o rezerwacji	

4.4.4. TrackCar

SubscribeForClientLocation (Subscribe)	
CarTracker	**Approacher**
Prosi Approachera o cykliczne wysyłanie swojej lokalizacji	

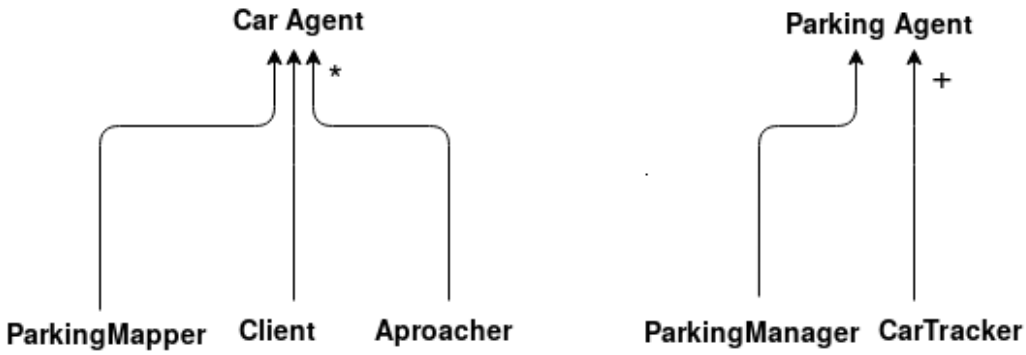
SendApproacherLocation (inform)	
Approacher	**CarTracker**
Wysyła cyklicznie dane o swojej lokalizacji do Car Trackera	

4.4.5. ReservationCancellation

CancelClientReservation (CANCEL)		
Approacher	**CarTracker**	
Wysyła informacje o rezygnacji z miejsca parkingowego		AproacherCancelledReservation

ConfirmCancellation (Confirm)		
CarTracker	**Approacher**	
Potwierdza otrzymanie informacji o rezygnacji		AproacherCancelledReservation

4.5. Model agentów



Rysunek 3. Model agentów.

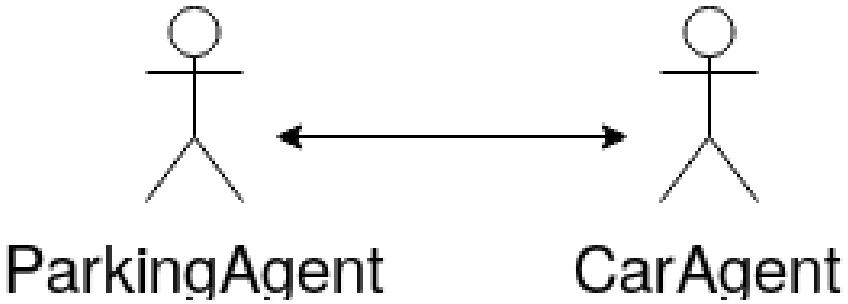
Na podstawie powyższych schematów (Rys. 3) modeli agentów można stwierdzić, że oba rodzaje agentów występujących w systemie są złożone w kontekście zaimplementowanych w nich ról. ParkingAgent posiada nieco większe skomplikowanie z uwagi na bardziej odpowiedzialną rolę jaką pełni w systemie, ponieważ to agenty-parkingi zarządzają agentami-pojazdami jednocześnie spełniając ich prośby. System nie może istnieć bez żadnej instancji ParkingAgent, ponieważ CarAgent w roli klienta nie mógłby wchodzić w interakcje i w rezultacie zaparkować. Natomiast można wyobrazić sobie istnienie systemu z instancjami ParkingAgent'a bez drugiego rodzaju agentów występujących w systemie.

4.6. Model usług

Tabela 6. Model usług.

Usługa	Wejścia	Wyjścia	Warunki wstępne	Warunki końcowe
Tworzenie listy lokalizacji istniejących parkingów	-	parking_list	true	parking_list =/= NULL
Tworzenie listy proponowanych parkingów z wolnymi miejscami w okolicy	car_location	selected_-parking_list	parking_list =/= NULL, car_location =/= NULL	length(sort(selected_-parking_list)) == length(distance(car_-location, parking_location) > set_distance) and length(sort(selected_-parking_list)) <=5 (wybór do 5 najbliższych parkingów z okolicy lub mniej jeśli ich liczba< 5)
Znalezienie wolnego miejsca	selected_-parking_list	reservation_info	parking_list =/= NULL	reservation_info =/= NULL
Śledzenie położenia samochodu	car_location reservation_info	ApproacherIsNot-Coming	reservation_info =/= NULL	ApproacherIsNot-Coming = True OR IsApproacher = False
Odwoływanie zarezerwowanego miejsca	reservation_info	Approacher-Cancelled-Reservation	reservation_info =/= NULL	ApproacherCancelled-Reservetation = True

4.7. Model znajomości



Rysunek 4. Model znajomości.

Model znajomości agentów w systemie (Rys. 4) jest nieskomplikowany z uwagi na małą liczbę agentów w nim występujących. Wszystkie rodzaje agentów komunikują się między sobą. Nadmienić należy że architektura systemu została zaprojektowana w ten sposób, że nie wymaga komunikacji pomiędzy różnymi instancjami tego samego rodzaju agenta. W przypadku agenta ParkingAgent następuje komunikacja pomiędzy rolami, w które wciela się ta sama instancja tego rodzaju agenta.

5. Opis implementacji systemu

5.1. Framework

Framework Narzędziem użytym w realizacji projektu jest JADE 4.5.0 (Java Agent Development Framework). Jest to popularny framework przeznaczony do implementowania systemów wieloagentowych w pełni zaimplementowany w języku Java. Wybraliśmy JADE, ponieważ jak można przeczytać w dokumentacji:

- w JADE wdrożony został pełny model komunikacji FIPA, a jego komponenty zostały wyraźnie wyróżnione i w pełni zintegrowane: protokoły interakcji, koperta, ACL, języki treści, schematy kodowania, ontologie i protokoły transportowe.
- architektura komunikacji oferuje elastyczne i wydajne przesyłanie wiadomości, gdzie JADE tworzy i zarządza kolejką przychodzących wiadomości ACL. Agenci mogą uzyskać dostęp do swojej kolejki za pomocą kombinacji kilku trybów: blokowania, odpytywania, limitu czasu i dopasowywania wzorców.
- Dodatkowo przydatną funkcjonalnością jest możliwość sterowania konfiguracją za pomocą interfejsu GUI.

5.2. Sposób implementacji agentów

Utworzyliśmy dwie klasy agentów CarAgent oraz ParkingAgent. Każdy agent dziedziczy funkcjonalność po klasie jade.core.Agent. Agenty wykorzystują klasę jade.core.behaviours. Behaviour do implementacji aktywności oraz protokołów zdefiniowanych dla ról.

CarAgent - Agent samochód, który może się przemieszczać. Dla celów testowych przyjęliśmy, iż na początku swojego istnienia agent ten losuje swoją pozycję początkową na mapie i dodaje zachowania UpdateListOfParkings, CallForParkingOffers, SendReservationInfo oraz ReservationCancellation, ListenForLocationCancelSubscriptionFromCarTracker.

UpdateListOfParkings - służy do wysyłania zapytania do wszystkich parkingów z prośbą o informacje o ich lokalizacji i uaktualnienia ich położenia. Protokół ten jest aktywowany na początku istnienia agenta samochodu. Agent zapisuje położenia wszystkich parkingów w HashMapie, gdzie kluczem jest AID parkingu, a wartością jest lokalizacja parkingu. Lokalizacje wszystkich parkingów są niezbędne dla CarAgent, ponieważ chce on wiedzieć, który parking dysponujący wolnym miejscem znajduje się najbliżej.

CallForParkingOffers - służy do zapoczątkowania konwersacji z najbliższymi parkingami w celu zarezerwowania miejsca. Agent samochód wysyła wiadomość typu CFP i otrzymuje od parkingów posiadających wolne miejsce odpowiedź typu PROPOSE lub REFUSE. Następnie typowany jest najbliższy parking z wolnym miejscem i podejmowana jest próba dokonania rezerwacji miejsca (ACCEPT PROPOSAL). Jeżeli CarAgent otrzyma odpowiedź pozytywną (INFORM) od ParkingAgent, to dany parking jest zapisany w pa-

mięci jako cel podróży, jeżeli jednak otrzymamy odpowiedź negatywną (FAILURE) to ponownie rozpoczynamy protokół CallForParkingOffers.

SendReservationInfo - służy do wysłania wiadomości od CarAgent (INFORM) do ParkingAgent o rezerwacji zrobionej dla konkretnego agenta. Po otrzymaniu informacji ParkingAgent wysyła potwierdzenie CONFIRM.

CancelClientReservation - służy do wysłania wiadomości o rezygnacji z zarezerwowanego miejsca (CANCEL) do ParkingAgent, będącego do tej pory celem podróży agenta-samochodu (CarAgent). Po otrzymaniu takiego żądania ParkingAgent zwalnia miejsce postojowe, a następnie wysyła potwierdzenie wykonania akcji (CONFIRM).

ListenForLocationCancelSubscriptionFromCarTracker - służy do nasłuchiwania czy CarTracker chce przestać subskrybować wiadomości, które są wysyłane w protokole SendReservationInfo.

ParkingAgent - agent parking posiadający miejsca parkingowe udostępniane pojazdom. Głównym zadaniem agenta jest oferowanie wolnych miejsc parkingowych oraz śledzenie aktualnej pozycji samochodów posiadających rezerwację. Agent ten wykorzystuje zachowania:

SendCoordinates - służy do wysłania informacji o swojej pozycji. Zachowanie to jest cykliczne i wykonywane po otrzymaniu wiadomości od agenta samochodu typu INFORM_REF.

SendAvailablePlaceInfo - zachowanie to jest cykliczne, wykonywane po przyjściu wiadomości zawierającej performatywę CFP od CarAgent. Jeżeli agent posiada wolne miejsce parkingowe to wysyła wiadomość typu PROPOSE w odpowiedzi. W przypadku kiedy agent nie posiada wolnego miejsca następuje wysłanie odpowiedzi zawierającej performatywę REFUSE.

ConfirmReservation - zachowanie cykliczne; po otrzymaniu od CarAgent wiadomości zawierającej performatywę ACCEPT_PROPOSAL, wyrażającej chęć zarezerwowania miejsca na danym parkingu, ParkingAgent zmienia stan miejsca z “wolny” na “zajęty” i odsyła wiadomość o pozytywnym przebiegu procesu (INFORM). Jeśli dane miejsce zostanie przed otrzymaniem wiadomości zawierającej ACCEPT_PROPOSAL zajęte przez innego CarAgent, zostaje wysłana wiadomość informująca o niepowodzeniu rezerwacji (FAILURE).

ConfirmCancellation - zachowanie cykliczne, po otrzymaniu wiadomości od CarAgent wyrażającej żądanie usunięcia rezerwacji miejsca parkingowego (CANCEL) zmienia stan miejsca z “zajęty” na “wolny”, a następnie wysyła odpowiedź o powodzeniu akcji (CONFIRM).

getReservationInfo - zachowanie cykliczne; służy do odbierania wiadomości od CarAgent o dokonaniu rezerwacji dla tego CarAgenta na parkingu, którym jest adresat wiadomości - ParkingAgent. Po otrzymaniu informacji ParkingAgent przesyła potwierdzenie CONFIRM.

5.3. Sposób implementacji komunikatów

5.3.1. Zastosowane performatywy

1. INFORM_REF
2. INFORM
3. FAILURE
4. CFP
5. PROPOSE
6. ACCEPT_PROPOSAL
7. CANCEL
8. CONFIRM
9. REFUSE

5.3.2. Protokoły komunikacyjne

W trakcie dotychczasowej implementacji został użyty tylko jeden typowy protokół komunikacyjny zdefiniowany przez FIPA i jest nim Subscribe Interaction Protocol, który został zastosowany do śledzenia Car Agents przez rolę CarTracker realizowaną przez Parking Agent. Pozostała wymiana komunikatów pomiędzy agentami została zebrana w indywidualnie zdefiniowanych dla naszego systemu protokołach stworzonych poprzez rozszerzenie klasy Behaviours, CyclicBehaviours oraz TickerBehaviours.

5.3.3. Zastosowane języki treści

Zastosowanym językiem treści jest FIPA Semantic Language (SL), który jest standardowo zaimplementowany we frameworku JADE.

5.4. Wykorzystane standardy

Implementowany system jest zgodny ze standardami FIPA, ze względu na wykorzystanie frameworka JADE.

- FIPA ACL (Agent Communication Language)
- FIPA SL (Semantic Language) - język semantyczny dla komunikacji ACL

5.5. Algorytmy

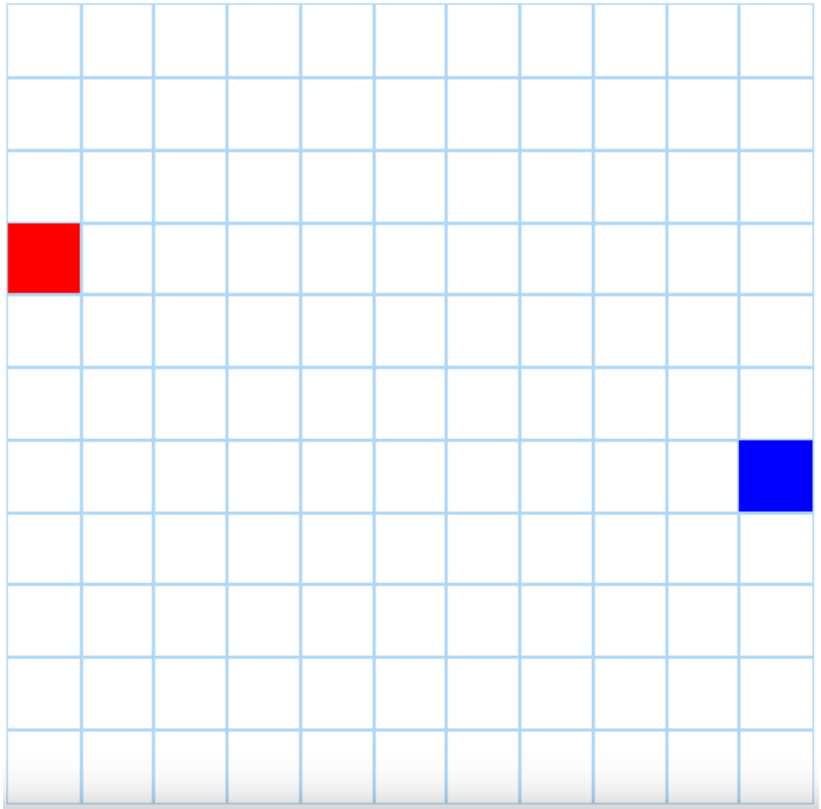
Algorytmy wykorzystane w systemie zostały pokrótce opisane w sposobie implementacji agentów. Na uwagę zasługuje sposób implementacji środowiska oraz sposób poruszania się po nim agentów. Stan naszego systemu jest zdefiniowany w postaci mapy 2D, gdzie każda z kratek odpowiada współrzędnej w środowisku agentów. Do każdej współrzędnej mogą być przypisane trzy wartości: czy współrzędna jest ulicą czy parkingiem. Samochody są losowo inicjalizowane na współrzędnych odpowiadających ulicom. Parking docelowy jest znajdowany poprzez obliczanie ścieżek do wszystkich parkingów, a następnie wybranie najkrótszej trasy. Samochód porusza się w jego kierunku po drodze obliczonej

w algorytmie wybierania ścieżki (opisanym poniżej). Końcowo samochód wjeżdża na współrzędną z parkingiem. Dopuszcza się możliwość, że dwa samochody znajdują się na tej samej współrzędnej. W aktualnej wersji systemu wszystkie pola mapy są ulicami.

Algorytm wyznaczania ścieżki został zaimplementowany przy pomocy algorytmu przeszukiwania wszerz (BFS). Algorytm ten w formie iteracyjnej wykorzystuje kolejkę do określenia, które kolejne punkty powinniśmy przebadać. Rozpoczynając od punktu startowego dodajemy wszystkie sąsiednie, nieodwiedzone wcześniej, punkty do kolejki i powtarzamy ten proces dla wszystkich punktów w kolejce. Jeżeli dotrzemy do punktu docelowego to zwracamy listę, która jest ścieżką od miejsca początkowego do docelowego. Nie jest to najoptymalniejszy algorytm (u nas, $O(i \cdot j)$ time | $O(i \cdot j)$ space, gdzie i - szerokość mapy, j - wysokość mapy), jednak został wybrany ze względu na łatwość implementacji w realizowanym systemie.

5.6. Wizualizacja

Postanowiliśmy zwizualizować działanie naszego projektu poprzez wyświetlanie informacji o aktualnym stanie systemu w konsoli oraz pokazywaniu położenia agentów na mapie 2D. Agent aktualizując swoje położenie wysyła informację do serwera, który przechowuje dane o bieżącym położeniu każdego agenta. Następnie serwer przekazuje uaktualnione informacje do aplikacji React, gdzie wyświetlana jest mapa. Backend i Frontend połączone są ze sobą poprzez websocket, więc wszystkie aktualizacje są wykonywane w czasie rzeczywistym. Ze względu na taki stos technologiczny możliwe są opóźnienia pomiędzy prezentowanym stanem systemu na mapie, a w konsoli. Przykładowa mapa jest widoczna na rysunku 5, gdzie kratka: biała oznacza drogę, czerwona parking, a niebieska samochód. Przykładowe wizualizacje naszego systemu są dostępne w postaci gifów na repozytorium naszego projektu w README.



Rysunek 5. Przykład mapy.

5.7. Napotkane problemy i zmiany w kolejnej iteracji

- Żaden członek zespołu nie jest biegły w Javie, więc dodatkową trudnością było szybkie zapoznanie się z samym językiem, a nie tylko nowym frameworkiem. Z tego powodu, nie mogliśmy się wyłącznie skupić na najlepszym rozwiązaniu i implementacji problemu.
- Podczas implementacji równoległego oczekiwania na wiadomości mieliśmy problem z przechwytywaniem wiadomości przez niepowołane do tego zachowania. Problem ten rozwiązaliśmy wykorzystując zaimplementowane w JADE szablony wiadomości (jade.lang.acl.MessageTemplate). Za pomocą metod MatchPerformative, MatchConversationID i wyfiltrowaliśmy przychodzące wiadomości skierowane do poszczególnych zachowań.

Spis rysunków

Spis tabel