

AgenCI - inteligentny system parkingów

Dawid Wysocki Damian Górska Gustaw Daczkowski Robert Żurawski
Wojciech Makos

Październik 2023

1 Wstęp

Jednym z problemów rozwijających się miast jest brak miejsc parkingowych. Nasze rozwiązanie próbuje sprostać temu problemowi poprzez aplikację zarządzającą miejscami na parkingach poprzez dynamiczny cennik.

1.1 Opis problemu

Wraz z rozwojem większych budynków, zwiększa się zageszczenie ludności. Mieszkańcy przemieszczając się po miastach, mogą korzystać z wielu środków transportu publicznego, jednak nadal często korzystają z aut. W samej Warszawie zgodnie z raportem GUS z przełomu 2021 i 2022 roku wynika, że na tysiąc mieszkańców przypada 768 samochodów [1].

Przez to coraz trudniej jest znaleźć miejsce parkingowe. Obowiązuje zasada "kto pierwszy, ten lepszy". Powoduje to niepotrzebne krążenie w poszukiwaniu wolnych miejsc, co zwiększa zatłoczenie ulic i emisję spalin.

1.2 Propozycja rozwiązania problemu

Nasz system stawia sobie za cel rozwiązanie tego problemu poprzez wprowadzenie innowacyjnej aplikacji do zarządzania miejscami parkingowymi. Kluczowym elementem naszego rozwiązania jest dynamiczny cennik, który dostosowuje opłaty za parkowanie do bieżącego zapotrzebowania i dostępności miejsc. Dzięki temu kierowcy będą mieli większą pewność znalezienia miejsca do parkowania, a miasta lub prywatni właściciele parkingów zyskają narzędzie do efektywnego zarządzania swoją infrastrukturą parkingową, będą mogli również maksymalizować swoje zyski.

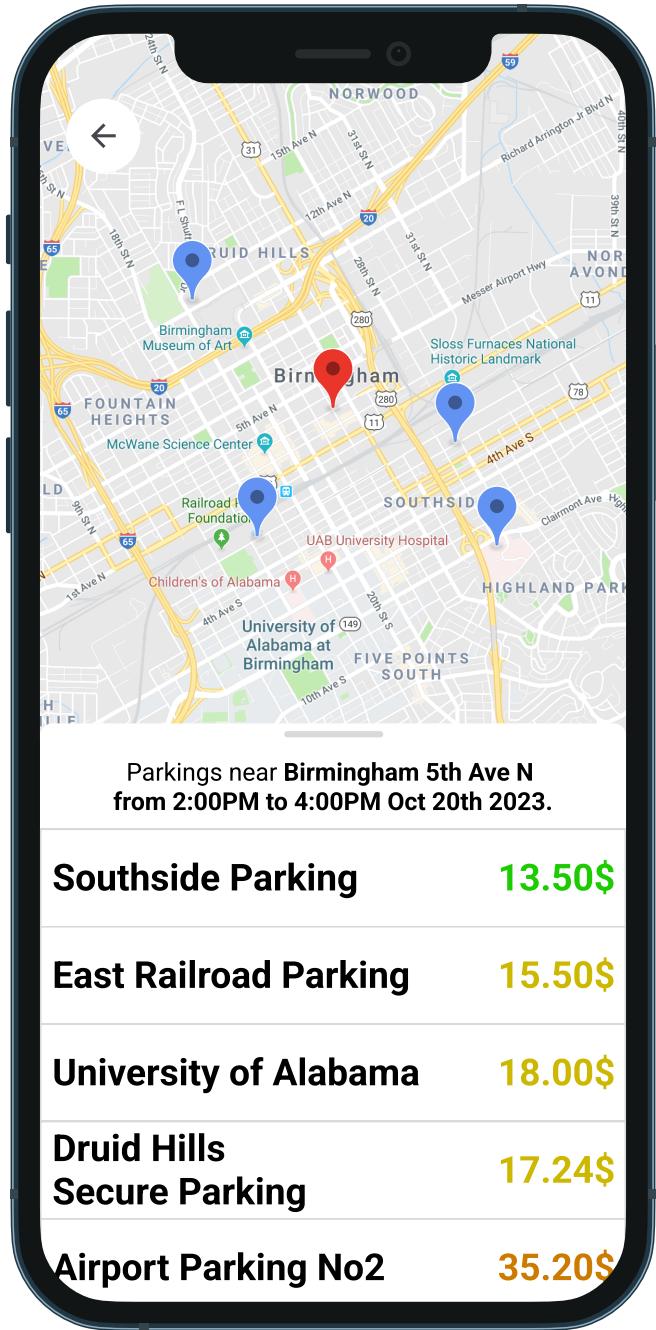
Makieta aplikacji mobilnej pozwalającej na przeglądanie parkingów została przedstawiona na rysunku 1. Repozytorium kodu git, w którym będzie rozwijany projekt, jest zamieszczone na platformie GitHub [2].

1.3 Koncepcja systemu

Rozwiązanie zakłada utworzenie modułów dla dwóch typów agentów: modułu dla kierowców (w formie aplikacji mobilnej lub webowej) oraz modułu dla parkingów. Ponadto w systemie może znaleźć się centralny moduł dbający o poprawność zawieszanych transakcji.

Aplikacja dla klientów parkingów umożliwia kierowcom wprowadzenie zapytania o dostępność miejsc parkingowych w danym miejscu i przedziale czasowym.

System odbierze zapytanie kierowcy o parking przy zadanym adresie i sporządzi listę parkingów wraz z cenami. Odbędzie się to dzięki komunikacji między różnymi agentami wewnątrz systemu. Parkingi tymczasowo zarezerwują przedstawione kierowcy miejsca parkingowe.



Rysunek 1: Makieta interfejsu mobilnego. Ekran przedstawia ceny parkingów w regionie.

Kierowca wybiera jedno z zaproponowanych miejsc parkingowych. Po wybraniu miejsca opłaca je. Po płatności system zwalnia rezerwacje na miejsca parkingowe odrzucone przez kierowcę, a pozostawia rezerwację na miejsce wybrane przez kierowcę.

Kierowca może w aplikacji mobilnej zobaczyć listę z zarezerwowanymi parkingami wraz z godzinami rezerwacji (tj. czas startu i czas trwania) oraz ceną.

2 Wymagania funkcjonalne

Aplikacja oferuje następujące funkcjonalności:

1. Rejestracja i logowanie użytkowników:
 - Umożliwienie użytkownikom rejestracji konta.
 - Zapewnienie możliwości logowania się do istniejącego konta.
2. Przegląd dostępnych miejsc parkingowych:
 - Wyświetlanie dostępnych miejsc parkingowych w określonej lokalizacji.
 - Możliwość filtrowania i sortowania wyników według różnych kryteriów, takich jak cena, odległość od miejsca docelowego itp.
 - Wyświetlanie informacji o dostępności miejsc parkingowych w czasie rzeczywistym.
3. Rezerwacja miejsc parkingowych:
 - Umożliwienie użytkownikom rezerwowania miejsc parkingowych na wyznaczony okres czasu.
 - Wyświetlanie dostępności miejsc w wybranym czasie.
4. Historia rezerwacji:
 - Zapewnienie użytkownikom dostępu do historii swoich rezerwacji, wraz z danymi dotyczącymi cen i czasu trwania rezerwacji.
5. Obsługa zarządzających parkingami:
 - Zapewnienie możliwości właścicielom parkingów zarządzania swoimi ofertami, dostępnością miejsc i dynamicznym cennikiem.
 - Panel administracyjny dla właścicieli parkingów do monitorowania rezerwacji i zarządzania swoimi placami parkingowymi.
6. Obsługa anulowania rezerwacji:
 - Umożliwienie użytkownikom anulowania rezerwacji z określonymi zasadami anulacji i zwrotu kosztów.

3 Projekt systemu

Projekt systemu został wykonany w oparciu o notację GAIA. Przyjęto, że w ramach opisu role pisane są wielką literą, aktywności są podkreślone, a obowiązki, protokoły i aktywności w tekście są pisane w nawiasach.

3.1 Role

- Parkujący - osoba szukająca miejsca parkingowego;
- Zarządzający Parkingiem - właściciel parkingu, ustala liczbę miejsc; przygotowuje miejsca parkingowe, ustala ceny, aktualizuje dane o wolnych miejscach;
- Informator - źródło danych o parkingach, tj. lokalizacja, maksymalna liczba miejsc parkingowych.

3.1.1 Rola: Parkujący

Role Description: Parkujący to osoba poszukująca miejsca parkingowego w systemie zarządzania miejscami parkingowymi.

Protocols and Activities: ParkingsInRangeRequest, ParkingsInRangeResponse, ParkingPriceRequest, ParkingPriceResponse, ChooseParking, ReserveRequest, ReserveResponse

Permissions: Reads supplied parkingsDetails, parkingPrice
Generates parkingChoice

Responsibilities: Liveness:

```
Park =ParkingsInRangeRequest.ParkingsInRangeResponse.  
        .AskForPrices. [AcceptParkingOffer]  
AskForPrices = (ParkingPriceRequest.[ParkingPriceResponse])*  
AcceptParkingOffer =ChooseParking.ReserveRequest.[ReserveResponse]
```

Safety: -

Dokładne objaśnienie roli

Parkujący jest kluczową rolą w systemie. Reprezentuje on klienta z samochodem poszukującego miejsca parkingowego. Jego celem jest zaparkowanie auta w najlepszym możliwym miejscu (Park). W ramach parkowania Parkujący zaprości Informatora o listę parkingów w pobliżu wskazanego adresu (ParkingsInRangeRequest).

Po otrzymaniu odpowiedzi (ParkingsInRangeResponse) parkujący zaprości o ceny od wszystkich Zarządców Parkingów wskazanych przez Informatora (AskForPrices). Zauważmy, że Zarządców Parkingów może być zarówno wielu w danej okolicy, jak i w skrajnym przypadku może nie być żadnego, zatem może się zdarzyć sytuacja braku odpowiedzi (ParkingPriceResponse). Funkcjonalność (AskForPrices) wykonuje się tyle razy, ile zostało wyznaczonych parkingów w danej okolicy przez informatora.

Po otrzymaniu w ofert (ParkingPriceResponse), Parkujący może nie zdecydować się na żadną ofertę lub wybrać jedną z nich (ChooseParking). Gdy wybierze ofertę, przesyła prośbę o rezerwację miejsca do Zarządcy Parkingu (ReserveRequest) i otrzymuje potwierdzenie rezerwacji (ReserveResponse). Jeżeli parkujący nie pojawi się w zarezerwowanym miejscu po określonym czasie to rezerwacja przepada i procedurę rezerwacji należy rozpocząć od początku.

Parkujący zatem powinien mieć pozwolenie na dostęp do danych na temat parkingów (parkingsDetails), uzyskiwać do wglądu ceny miejsc parkingowych dla danego parkingu (parkingPrice) i na tej podstawie podejmować wybór, gdzie będzie chciał zaparkować (parkingChoice).

Zakładamy, że implementacja korzystała by z mechanizmów timeout'u m.in. przy AskForPrices, aby nie czekać zbyt długo na odpowiedzi parkingów. Także po ustaleniu ceny przez Zarządcę Parkingu, Parkujący będzie musiał się zdecydować w określonym czasie, aby utrzymać jak najbardziej sprawiedliwy cennik. W przypadku błędu np. przy rezerwacji odpowiedzialnością Parkującego jest próba wznowienia komunikacji od właściwego momentu.

3.1.2 Rola: Zarządzający Parkingiem

Role Description: Zarządzający Parkingiem to rola odpowiedzialna za ustalanie liczby miejsc na danym parkingu, przygotowywanie miejsc parkingowych, ustalanie cen na podstawie dynamicznego cennika oraz aktualizację danych o dostępnych miejscach.

Protocols and Activities: `ParkingInformationMessage`, `ParkingPriceRequest`, `ParkingPriceCalculation`, `ParkingPriceResponse`, `ReserveRequest`, `ReserveParkingSpot`, `ReserveResponse`, `ReleaseParkingSpot`

Permissions: `Reads supplied` `parkingDecision`

`Changes` `parkingPrice`, `parkingState`

`Generates` `parkingPrice`, `parkingAvailability`, `parkingInformation`

Responsibilities: Liveness:

`ManageParking` =
$$= (UpdateParkingInformation \parallel RespondToParkPriceRequest \parallel RespondToReserveRequest)^\omega$$

`UpdateParkingInformation` = `ParkingInformationMessage`

`RespondToParkPriceRequest` = `ParkingPriceRequest`.`ParkingPriceCalculation`.
[`ParkingPriceResponse`]

`RespondToReserveRequest` = `ReserveRequest`.`ReserveParkingSpot`.
[`ReserveResponse`.`ReleaseParkingSpot`]

Safety: `ParkedCarsCount(timestamp) \leqslant ParkingSpotsCount(timestamp)`

Dokładne objaśnienie roli

Głównym obowiązkiem Zarządcy Parkingu jest obsługa parkingu (`ManageParking`). W ramach obsługi Zarządcy Parkingiem będzie równocześnie:

- Aktualizował informacje statyczne, takie jak ilość wszystkich miejsc czy adres w przypadku ich zmian poprzez wysłanie (`UpdateParkingInformation`) do Informatora.
- Odpowiadał na zapytania o cenę miejsca parkingowego w danych godzinach (`RespondToParkPriceRequest`) wysłanej od Parkującego w ramach wiadomości (`ParkingPriceRequest`) na podstawie własnego algorytmu obliczania cen (`ParkingPriceCalculation`). W przypadku braku miejsc parkingowych w danych godzinach nie odpowie na zapytanie, a jeśli będą miejsca parkingowe wyśle do Parkującego (`ParkingPriceResponse`).
- Odpowiadał na prośby o rezerwację miejsc (`RespondToReserveRequest`). Po otrzymaniu od Parkującego prośby o rezerwację (`ReserveRequest`) Zarządcy Parkingu spróbuje zarezerwować miejsce parkingowe na dane godziny (`ReserveParkingSpot`). W przypadku powodzenia wyśle Parkującemu wiadomość (`ReserveParkingResponse`), a po skończeniu parkowania zwolni miejsce parkującego (`ReleaseParkingSpot`).

W aspekcie pozwoleń zarządzający parkingiem powinien móc dokonywać zmian w cenach za parkowanie (`parkingPrice`) oraz mieć możliwość ich generowania. Dodatkowo powinien móc zmieniać stan miejsc parkingowych na: wolne, zarezerwowane lub zajęte (`parkingState`). Do tego zarządzający parkingiem będzie miał prawo do decydowania o dostępności swojego parkingu (`parkingAvailability`). Dzięki temu zarządzający może ustalić, że parking pomimo wolnych miejsc będzie np. zamknięty. Ponadto zarządzający parkingiem powinien mieć możliwość ustanawiać informacje globalne (`parkingInformation`) o danym parkingu, takie jak: lokalizacja, liczba wolnych miejsc. Informacje (`parkingInformation`) będą przekazywane informatorowi. Odnośnie obowiązków bezpieczeństwa zarządzający parkingiem będzie

musiał zagwarantować, że liczba samochodów na parkingu jest zawsze mniejsza bądź równa liczbie wszystkich miejsc na parkingu. Warunek ten będzie zapobiegał ewentualnym błędom podczas rezerwacji.

3.1.3 Rola: Informator

Role Description: Informator to rola, która dostarcza statyczne dane o parkingach, takie jak lokalizacja i maksymalna liczba miejsc parkingowych. Stanowi źródło informacji kluczowych informacji z punktu widzenia parkującego.

Protocols and Activities: ParkingInformationMessage, ParkingsInRangeRequest, ParkingsInRangeFilter, ParkingsInRangeResponse

Permissions: Reads supplied parkingInformation
Generates parkingsInformationInRange

Responsibilities: Liveness

Handle = (ParkingInformationMessage||RespondToParkingsInRange)^w
RespondToParkingsInRange =ParkingsInRangeRequest.ParkingsInRangeFilter.ParkingsInRangeResponse

Safety: UpToDateParkingInformation = True

Dokładne objaśnienie roli

Głównym obowiązkiem Informatora jest przekazywanie informacji o parkingach do Parkujących na podstawie wiadomości przesłanych przez Zarządców Parkingów.

Aby Informator posiadał informacje o parkingu, obowiązkiem Zarządcy Parkingu jest informowanie o zmianach w adresie i pojemności zarządzającego parkingu poprzez wysłanie do Informatora wiadomości (ParkingInformationMessage). Po odebraniu tej wiadomości Informator zaktualizuje posiadaną wiedzę na temat parkingów.

Gdy Parkujący będzie odpytywał Informatora zapytaniem (ParkingsInRangeRequest) obowiązkiem Informatora jest znalezienie parkingów w okolicy wskazanej przez Parkującego (ParkingsInRangeFilter) i przekazanie adresów potrzebnych do komunikacji z Zarządcami Parkingu oraz informacjami takimi jak dokładny adres parkingu, czy ilość wszystkich miejsc (ParkingsInRangeResponse). W przypadku braku parkingów w zasięgu Informator przekaże pustą listę w odpowiedzi (ParkingsInRangeResponse).

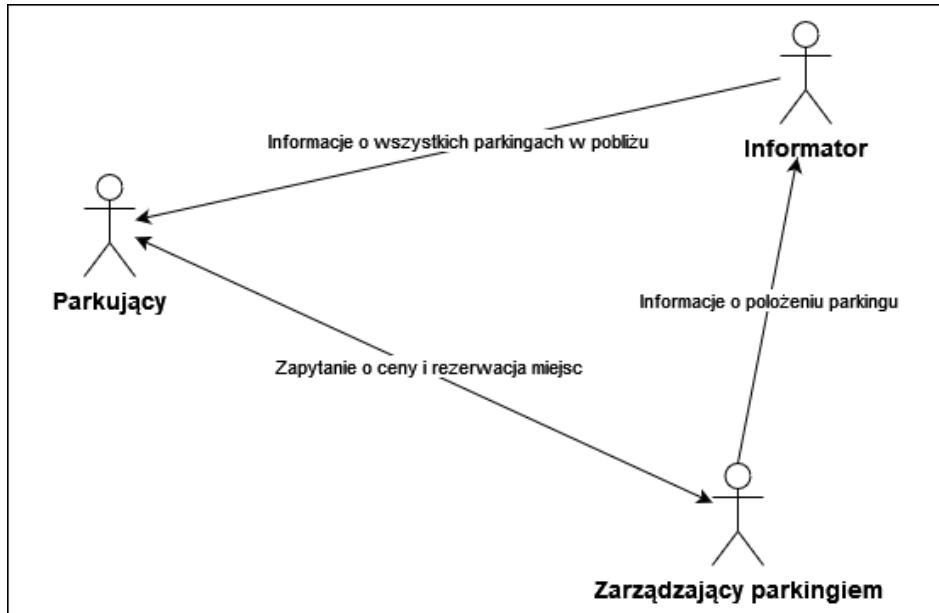
W aspekcie pozwoleń informator na pewno powinien mieć dostęp do danych z pobliskich parkingów(parkingInformation). Na podstawie tych danych oraz zapytania klienta(parkujacego) informator będzie generował listę o parkingach wraz z ich cenami i lokalizacją (parkingsInformationInRange).

3.2 Podział na agentów

Na podstawie ról w opisanych podrozdziale 3.1 zdecydowano się utworzyć 3 agentów, z których każdy odgrywały jedną, osobną rolę tzn. Parkujący, Zarządzający Parkingiem oraz Informator. Podział na trzy odrębne role agentów ułatwia zrozumienie i zarządzanie różnymi aspektami systemu parkingowego. Każdy agent ma swoje specyficzne zadania i odpowiedzialności, co ułatwia utrzymanie klarowności i efektywności w działaniu systemu.

3.3 Interakcja agentów

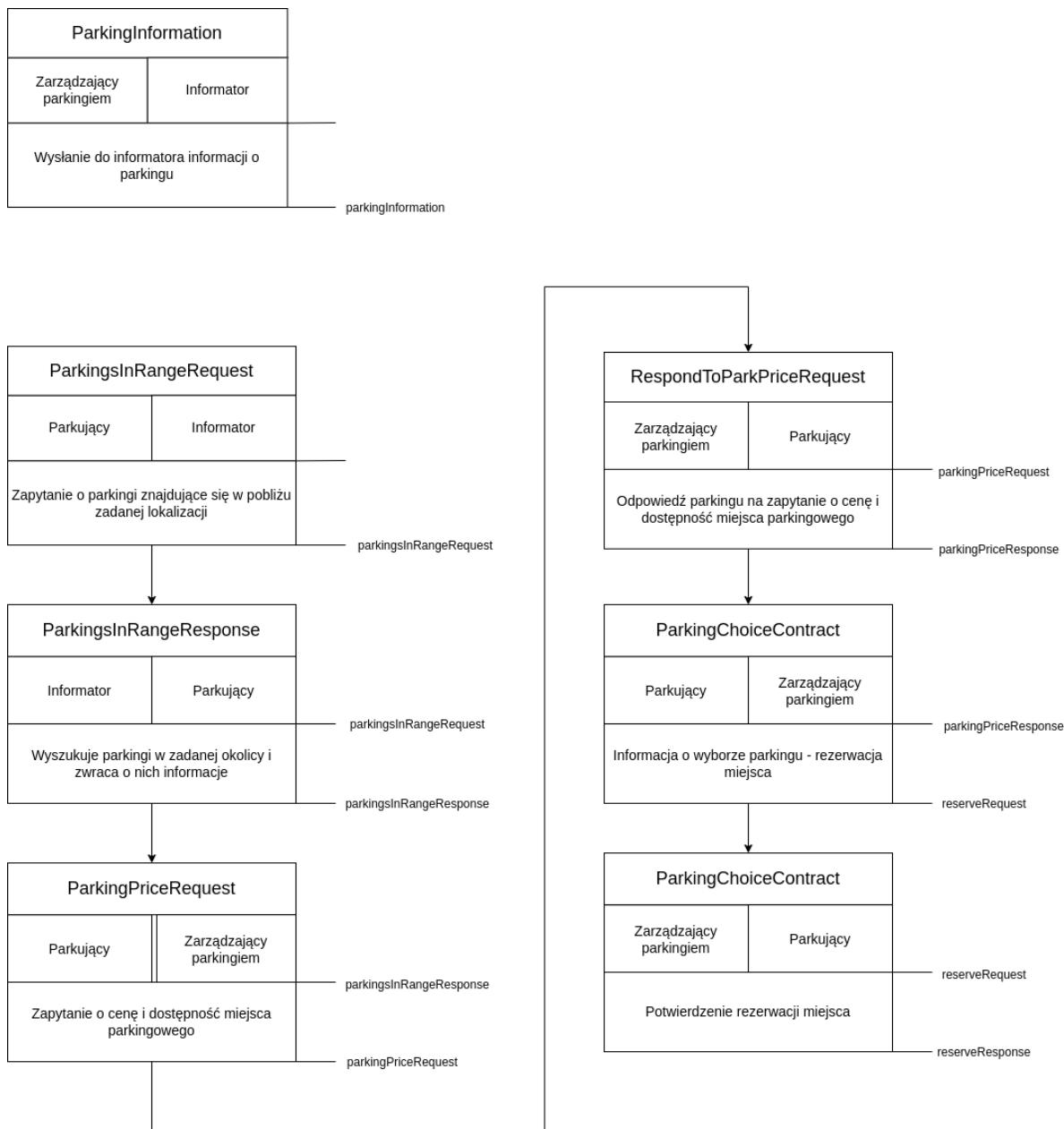
Agenty będą porozumiewać się ze sobą zgodnie z rysunkiem 2. Szczegółowe wiadomości zostały opisane w podrozdziale opisującym role [3.1].



Rysunek 2: Interakcje pomiędzy agentami

3.4 Protokoły w rolach agentów i połączenia w nich

W protokołach ról agentów można wydzielić dwie osobne ścieżki komunikacji, które są niezależne względem siebie. Pierwsza z nich - częściej występująca w systemie - jest komunikacją ról w celu znalezienia odpowiedniego parkingu przez Parkującego oraz zarezerwowania miejsca. Drugą ścieżką jest komunikacja w celu wysłania statycznych informacji o parkingu przez Zarządzającego parkingiem do Informatora, aby dane te mogły zostać zarejestrowane i tym samym dostępne dla Parkujących. Schemat obu ścieżek komunikacyjnych znajduje się na rysunku 3. Przedstawione są na nim protokoły występujące w systemie, powiązania między nimi oraz kolejność ich występowania.



Rysunek 3: Diagram protokołów GAIA

3.5 Zgodność z FIPA

Jesteśmy w stanie znaleźć funkcję przekształcającą pomiędzy Protokołami przedstawionymi w podrozdziale 3.1, a performatywami FIPA:

- **query-if** - ParkingsInRangeRequest, ParkingsPriceRequest
- **inform** - ParkingsInformationMessage, ParkingsInRangeResponse, ParkingPriceResponse,
- **accept-proposal** - ReserveResponse,

- **propose** - ReserveRequest.

Poniżej zaprezentowano kilka przykładowych komunikatów FIPA realizowanych na podstawie opracowanego modelu GAIA:

```
(performatywa :Query-if
    :sender Parkujący
    :receiver Informator
    :content "<ParkingsInRangeRequest>
        <lat>52.13213</lat>
        <lon>20.13123</lon>
    </ParkingsInRangeRequest>""
:reply-with 0965jf034t
:reply-by "2023-11-18 16:55:00"
:language M3XML
:Ontology Inteligentny System Parkingów)

(performatywa :Inform
    :sender Informator
    :receiver Parkującego
    :content "<ParkingsInRangeResponse>
        <ParkingInfo>
            <name>Super parking</name>
            <uri>super-parking@adresserwera.com</uri>
            <parkingSpotsCount>123</parkingSpotsCount>
            <lat>52.13213</lat>
            <lon>20.13123</lon>
        </ParkingInfo>
        ...
        <ParkingInfo>
            <name>Parking zadaszony</name>
            <uri>parking-zadaszony@adresserwera.com</uri>
            <parkingSpotsCount>143</parkingSpotsCount>
            <lat>50.13213</lat>
            <lon>21.13123</lon>
        </ParkingInfo>
    </ParkingsInRangeResponse>""
:reply-with 0965jf034t
:reply-by "2023-11-18 16:55:00"
:language M3XML
:Ontology Inteligentny System Parkingów)

(performatywa :Propose
    :sender Parkujący
    :receiver Zarządzający-parkingiem
    :content "<ReserveRequest/>""
:reply-with 0941jf034t
:reply-by "2023-11-18 16:15:00"
:language M3XML
:Ontology Inteligentny System Parkingów)
```

```
(performatywa :Accept-proposal
:sender Zarządzający-parkingiem
:receiver Parkujący
:content "<ReserveResponse/>"
:reply-with 0965jf034t
:reply-by "2023-11-18 16:55:00"
:language M3XML
:Ontology Inteligentny System Parkingów)
```

4 Opis implementacji

Opracowana implementacja systemu aktorowego (backend) wykorzystuje udostępniane webowe API we frameworku ASP.NET autorstwa firmy Microsoft. Dzięki niemu zarówno parkujący jak i właściciele parkingów są w stanie utworzyć aktorów realizujących powierzone cele na postawionym serwerze oraz obserwować ich komunikację. Aktor odgrywający rolę orkiestratora (nadzorcy) jest inicjowany wewnętrz systemu i użytkownik nie ma do niego dostępu, tzn. nie jest on inicjowany poprzez użytkownika lecz istnieje od początku działania systemu. Z tak zaprojektowanym backendem komunikuje się warstwa prezentacji (frontend), która domyślnie będzie dostępna z urządzeń mobilnych dla zarządców parkingów oraz kierowców. Cała logika systemu (wraz z zachowaniami agentów) jest po stronie backendu.

4.1 Wykorzystany język oraz framework

Opisywany wyżej system wieloaktorowy zaimplementowano w języku C#. Wykorzystano wieloplatformowy framework ASP.NET wraz z biblioteką Microsoft Orleans[3]. Została ona stworzona z myślą o aplikacjach rozproszonych, jednakże wykorzystanie jej właściwości i funkcjonalności posłużyło do właściwej implementacji opracowanego systemu aktorowego. Wybrany framework nie zapewnia automatycznie zgodności komunikatów z FIPA.

4.2 Sposób implementacji aktorów oraz komunikacji między nimi

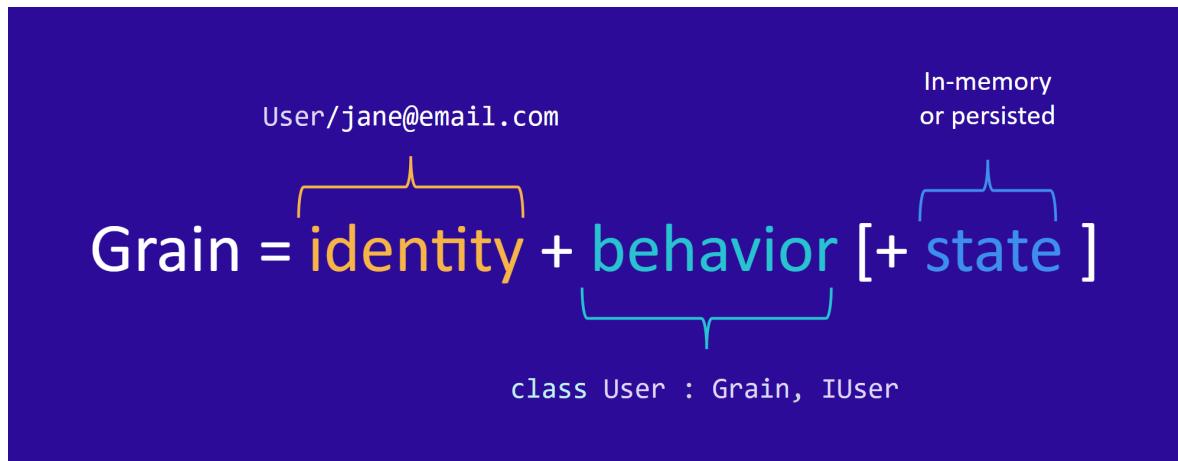
Kod źródłowy został podzielony na 3 projekty:

1. Agenci.Abstractions
2. Agenci.Grains
3. Agenci.WebApi

Pierwszy element to projekt, w którym zostały zaimplementowane struktury protokołów i aktywności, wykorzystywane przez zdefiniowanych aktorów. Drugi z nich to definicja aktorów oraz sposobu realizacji ich zadań. Trzeci z projektów odpowiada za poprawne utworzenie serwera agentowego i komunikację z użytkownikiem na poziomie deweloperskim.

Sposób komunikacji pomiędzy aktorami w aplikacji Orleans działa na zasadzie peer-to-peer; komunikaty wysyłane są asynchronicznie. Model aktora w tym systemie jest określony mianem "ziarna"/"Grain". Grain jest podstawowym elementem aplikacji. Ziarno jest definiowane poprzez 3 składniki pokazane na diagramie 4:

- tożsamość - czyli unikalny identyfikator danego aktora, poprzez wywołanie którego wiadomości mogą być adresowane do właściwego odbiorcy,



Rysunek 4: Definicja ziarna. Źródło: <https://learn.microsoft.com/en-us/dotnet/orleans/media/grain-formulation.svg#lightbox>.

- działanie - czynności, które mogą wykonywać dani aktorzy, zaimplementowane jako klasy danych zachowań. Określają protokoły oraz aktywności dla danych ról.
- stan - przechowywany w pamięci (względy szybszego wznowienia działania aktorów w przypadku awarii systemu)

Serwer aktorowy ma dwudzielny charakter. Pierwsza warstwa odpowiada za komunikację z użytkownikiem. Wykorzystując interfejs użytkownika można wprowadzić dane parkingu lub stworzyć chęć rezerwacji miejsca parkingowego. Po tej operacji wstawione dane trafiają do właściwego klastra wieloagentowego i w tym klastrze realizują się opisane w rozdziale 3 protokoły komunikacyjne. Dzięki tym działaniom wirtualne aktory dogadują się i ustalają miejsce i koszt parkingu. Rezultaty pracy aktorów użytkownik może wyświetlić, a następnie dokonać rezerwacji wybranego przez siebie miejsca w danym parkingu. Informacja o chęci rezerwacji idzie z powrotem do klastra i w tym momencie aktor odpowiadający klientowi dokonuje właściwej rezerwacji.

Poniżej zostaną wylistowane nagłówki funkcji, które realizują protokoły komunikacyjne opisywane w rozdziale 3.4. Dokładną implementację tych funkcji można znaleźć w repozytorium projektu. Przykładowe performatywy takie jak: `query-if`, `inform` czy `propose` są realizowane tutaj na nieco wyższym poziomie, ponieważ zostały one już wpisane w działanie metod poszczególnych klas aktorów. Do realizacji protokołów komunikacyjnych konieczna okazała się implementacja struktur danych:

- `ParkingDetailedInfo` - struktura ta jest nośnikiem informacji o danym parkingu. Zawiera identyfikator parkingu, współrzędne geograficzne tj. długość i szerokość, dane adresowe, maksymalną liczbę miejsc dla samochodów oraz listę rezerwacji. Ta struktura informacyjna będzie wymieniana pomiędzy, gdy do systemu zgłosi się właściciel parkingu. Będzie on musiał dane o swoim parkingu zarejestrować w systemie i przekazać informatorowi.
- `ParkingInfo` - struktura informacyjna wymieniana pomiędzy parkingiem a samym parkującym w przypadku chęci rezerwacji miejsca parkingowego.
- `ParkingInfoWithPrice` - struktura informująca dodatkowo o dostępności danego parkingu, koszcie parkowania, czasie rozpoczęcia i zakończenia parkowania.
- `ParkingsInRange` - zawiera listę parkingów w okolicy
- `Reservation` - nośnik informacji o numerze miejsca, którego będzie dotyczyć rezerwacja wraz z ceną oraz parkującym przypisanym do tego miejsca.

Ostatecznie zaimplementowano następujące protokoły komunikacji pomiędzy ziarnami:

- dotyczące parkującego

```
public async Task<List<ParkingOffer>> GetParkingOffers(double latitude, double longitude, DateTime start, DateTime end)
public async Task<int?> ChooseParking(string parkingId)
public Task<List<ParkingOffer>> GetReservedParkingHistory()
```

- dotyczące informatora

```
void ReceiveParkingInformation(ParkingInfo parkingInfo)
ParkingsInRange GetParkingsInRange(double latitude, double longitude)
void GetDistanceSquared(double latitude1, double longitude1,
                        double latitude2, double longitude2)
```

- dotyczące właściciela parkingu.

```
public async Task<ParkingOffer> GetParkingPriceResponse(DateTime start, DateTime end, string driverKey)
public async Task<int?> ReserveParking(string driverKey)
public async Task CancelReservation(string driverKey)
public Task<ParkingDetailedInfo> GetParkingDetailedInfo()
public async Task UpdateParkingInfo(ParkingInfo parkingInfo)
```

Implementacja przykładowego ziarna jest przedstawiona na listingu poniżej. Jest to implementacja ziarna kierowcy, przedstawia częściową funkcjonalność: realizację pobierania ofert parkingów. Pełna implementacja jest dostępna w repozytorium github.

```
public class DriverUserGrain: Grain, IDriverUserGrain
{
    private readonly IGrainFactory _grainFactory;
    private readonly IPersistentState<List<ParkingOffer>> _parkingOffers;
    private readonly IPersistentState<List<ParkingOffer>> _history;

    public DriverUserGrain(
        IGrainFactory grainFactory,
        [PersistentState("parkingOffers", "driverStore")]
        IPersistentState<List<ParkingOffer>> parkingOffers,
        [PersistentState("history", "driverStore")]
        IPersistentState<List<ParkingOffer>> history)
    {
        _grainFactory = grainFactory;
        _parkingOffers = parkingOffers;
        _history = history;
    }

    public async Task<List<ParkingOffer>> GetParkingOffers(double latitude,
                                                               double longitude,
                                                               DateTime start,
                                                               DateTime end)
    {
        var orchestratorGrain = _grainFactory.GetGrain<IOrchestratorGrain>(Guid.Empty);
        var parkingsInRange = await orchestratorGrain.GetParkingsInRange(latitude, longitude);
        var parkingGrains = parkingsInRange.Parkings
            .Select(parking => _grainFactory.GetGrain<IParkingGrain>(parking.Key));
    }
}
```

```

    var parkingOffers = await Task.WhenAll(
        parkingGrains.Select(parkingGrain
            => parkingGrain.GetParkingPriceResponse(start, end, this.GetPrimaryKeyString())));
    _parkingOffers.State = parkingOffers.ToList();
    await _parkingOffers.WriteStateAsync();

    return parkingOffers.Where(offer => offer.IsAvailable).ToList();
}
}

```

Wykorzystując powyższe metody stworzono deweloperski interfejs serwera przedstawiony na rysunku 5.

The screenshot shows the Swagger UI for the Agenci.WebApi application. At the top, there's a header with the title "Agenci.WebApi" and version "1.0 OAS3". Below the header, a URL "http://localhost:5243/swagger/v1/swagger.json" is displayed. The main content area lists several API endpoints:

- GET /parkings**
- GET /parkings/{id}**
- POST /parkings/{id}**
- GET /drivers/{id}/history**
- GET /drivers/{id}/parkings**
- POST /parkings/{id}/reserve**

Rysunek 5: Interfejs OpenAPI przedstawiający zapytania HTTP dostępne na serwerze.

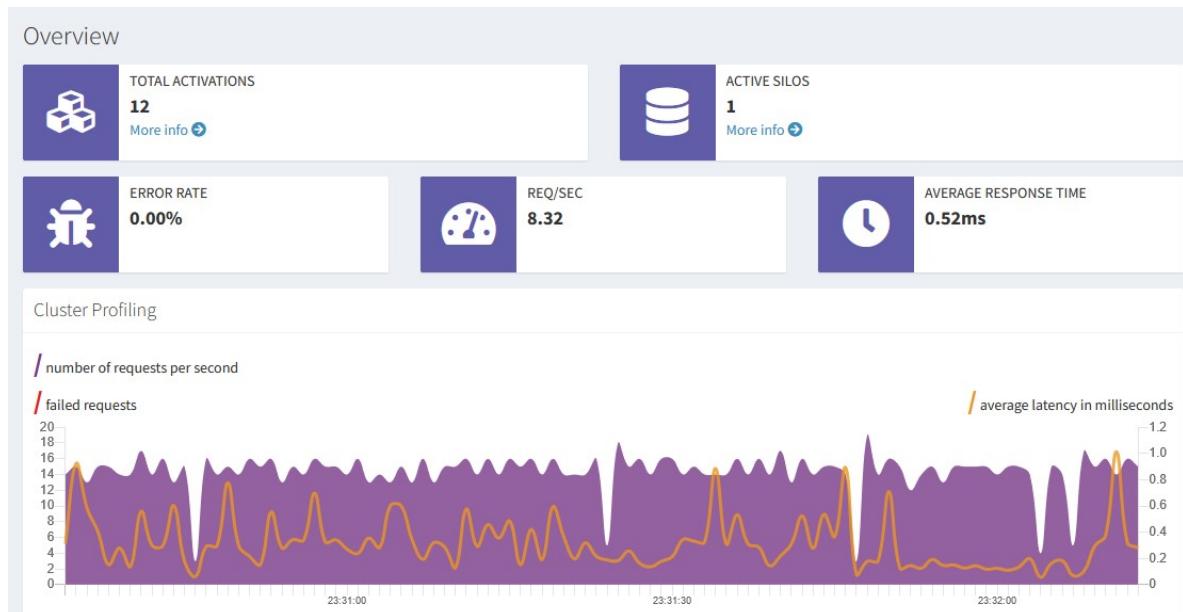
Jest on dostępny w przeglądarce internetowej poprzez protokół HTTP. Udostępnia on takie funkcjonalności jak: pobieranie danych o parkingach, wstawianie danych o parkingach (dla właścicieli parkingu), pobieranie danych o kierowcach, wyrażanie chęci rezerwacji konkretnego parkingu (interfejs dla parkującego).

Przykładowo uruchamiając żądanie GET `/drivers/{id}/parkings`, wywoływana jest metoda `GetParkingOffers` zdefiniowana dla aktora parkującego. W obrębie tej metody zdefiniowana została

komunikacja, w której to parkujący dostaje na początku od informatora listę parkingów `GetParkingsInRange`, następnie dostaje wyceny od parkingów znajdujących się w okolicy `GetParkingPriceResponse`. Następnie parkujący będzie za pomocą aplikacji wybierał parking, który wystawił najkorzystniejszą ofertę. Za to odpowiada ostatnia pozycja ukazana na rysunku 4. Po jej realizacji, zacznie się wykonywać metoda `ChooseParking`, która to sprawdza dostępność wybranego parkingu, oraz wywołuje metodę `ReserveParking`, która to zmienia stan wewnętrzny obiektu danego parkingu.

Niedogodnością rozwiązania może się okazać rozbieżność między nazwami protokołów komunikacyjnych podanych w wersji koncepcji systemu, a nazwami już zaimplementowanymi. Przykładowo jednym z opisywanych obowiązków roli parkującego (rozdział 3.1.1) jest realizacja `AskForPrices`, na którą składa się operacja `ParkingPriceRequest` oraz `ParkingPriceResponse`. Przy implementacji metody te zostały uproszczone już do wywołania przez parkującego samej metody właściciela parkingu `GetParkingPriceResponse`, która bezpośrednio zwraca strukturę danych `ParkingInfoWithPrice`. Delikatna zmiana nazewnictwa nie zmienia założeń projektowych, a ułatwiła i uprościła pod względem realizacji wymyślony system aktorowy.

Ponadto aplikacja udostępnia funkcjonalność administratorską (rysunek 6) całego serwera aktorowego. Wyświetlane są w niej wszelkie aktywności ze strony właścicieli parkingów jak i parkujących.



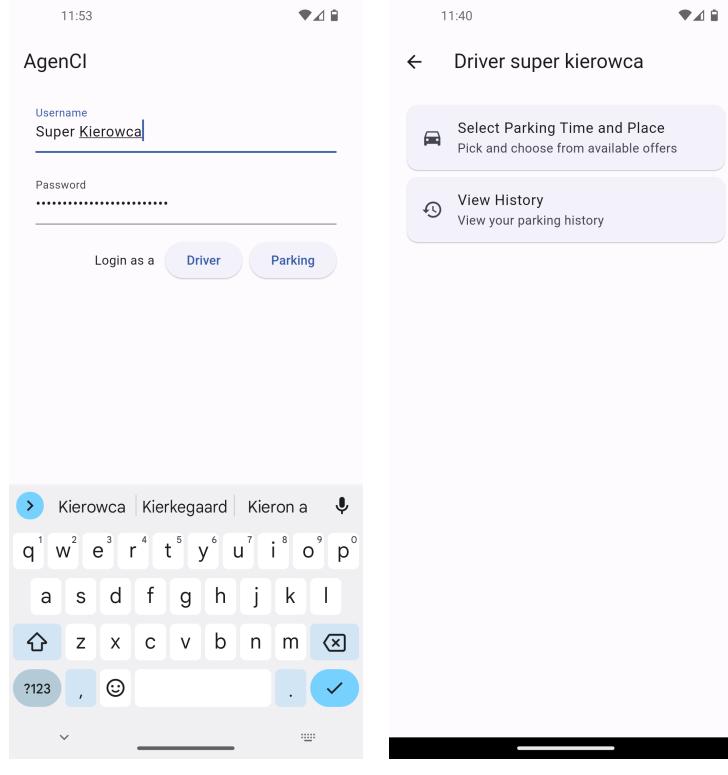
Rysunek 6: Interfejs administratora serwera.

4.3 Napotkane problemy i mankamenty

Przy implementacji nie natknęto na żadne poważne problemy. Jedynym godnym uwagi problemem przy implementacji było przeoczenie dziedziczenia po klasie bazowej `Grain`. Pominięcie tego dziedziczenia zwróciło wyjątek dopiero w czasie wykonania programu, o treści sugerującej problem ze znalezieniem stanu danego `Grain'a`, a nie o braku dziedziczenia po klasie `Grain`. Przez myłętą treść wyjątku znalezienie przyczyny trwało około pół godziny.

4.4 Interfejs użytkownika

Interfejs użytkownika został zaimplementowany przy pomocy biblioteki Flutter. Flutter to otwarto źródłowy framework stworzony przez Google do tworzenia interfejsów użytkownika (UI). Umożliwia



Rysunek 7: Wybrane ekrany z aplikacji na platformie Android.

rozwijanie aplikacji na różne platformy, takie jak iOS, Android, web, desktop itp. przy użyciu jednego kodu źródłowego.

W naszym przypadku ze względu na wykorzystanie biblioteki Google Maps ograniczamy się do platform mobilnych tzn. Android oraz iOS i przeglądarki.

W ramach prototypu nie zaimplementowano w pełni uwierzytelniania i autoryzacji. Działa natomiast w pełni wyszukiwanie i rezerwacja miejsca parkującego przez kierowcę. Ponadto kierowca może wyświetlić historię zarezerwowanych parkingów.

Wybrane fragmenty interfejsu graficznego przedstawiono na rysunkach 7 oraz 8.

4.5 Opis algorytmów

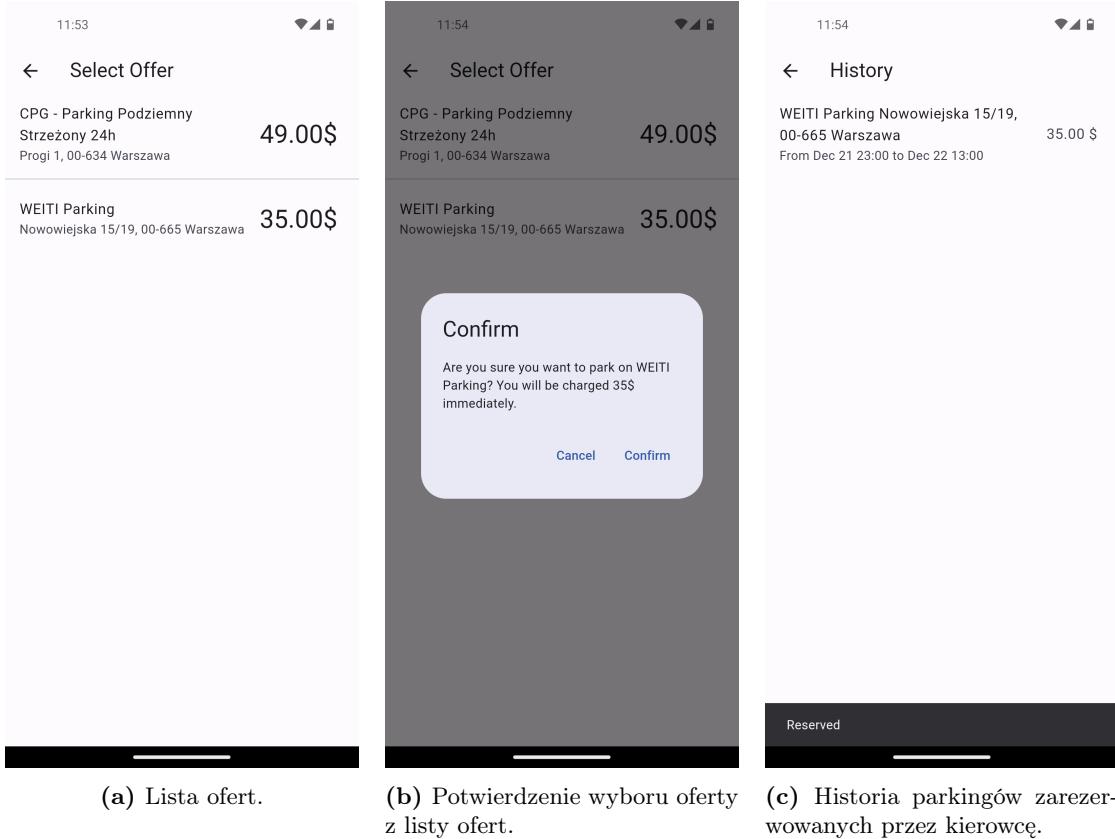
Do odnajdywania najbliższych parkingów i realizacji aktywności ParkingsInRange wykorzystano algorytm wyszukiwania:

4.5.1 Algorytm wyznaczenia wolnych miejsc parkingowych w danym przedziale czasowym

Zadaniem metody GetAvailableSpots jest znalezienie dostępnych miejsc parkingowych w danym przedziale czasowym (start, end).

Lista kroków algorytmu:

- Utworzenie zbioru zablokowanych miejsc



Rysunek 8: Wybrane ekrany z aplikacji na platformie Android.

- Algorytm rozpoczyna się od utworzenia zbioru *blocked*, który zawiera numery miejsc parkingowych, które są zajęte w danym przedziale czasowym
- Dla każdej rezerwacji *reservations.State*, sprawdzane jest, czy istnieje kolizja czasowa z danym przedziałem (*reservation.Start < end || reservation.End > start*). Jeśli tak, numer miejsca parkingowego z tej rezerwacji jest dodawany do zbioru *blocked*.
- Utworzenie listy dostępnych miejsc
 - Następnie algorytm iteruje po wszystkie miejscowości parkingowe od *0* do *parkingInfo.State.MaxCapacity - 1*.
 - Dla każdego miejsca parkingowego sprawdzane jest, czy nie znajduje się w zbiorze *blocked*. Jeśli tak, oznacza to, że miejsce jest zajęte w danym przedziale czasowym, więc zostaje pominięte. Jeśli nie, dodaje numer miejsca parkingowego do listy *availableSpots*.
- Zwracanie listy dostępnych miejsc
 - Metoda zwraca listę *availableSpots*, która zawiera numery miejsc parkingowych, które są dostępne w danym przedziale czasowym.

```
private List<int> GetAvailableSpots(DateTime start, DateTime end)
{
    var blocked = _reservations.State
        .Where(reservation => reservation.Start < end || reservation.End > start)
```

```

        .Select(reservation => reservation.PlaceNumber)
        .ToHashSet();

    var availableSpots = new List<int>();

    for (int spot = 0; spot < _parkingInfo.State.MaxCapacity; ++spot)
    {
        if (!blocked.Contains(spot))
        {
            availableSpots.Add(spot);
        }
    }

    return availableSpots;
}

```

Opisany algorytm opiera się na sprawdzeniu kolizji czasowej z rezerwacjami dla każdego miejsca parkingowego w określonym przedziale czasowym. Miejsca, które są zajęte w tym przedziale, są uznawane za zablokowane, a reszta jest uważana za dostępną. Algorytm ten umożliwia szybkie znalezienie dostępnych miejsc parkingowych, co jest istotne przy powiadamianiu szukających parkingu.



Rysunek 9: Graficzna ilustracja doboru numerów miejsc parkingowych do zbioru blocked.

Użytkownik A zarezerwował miejsce parkingowe jako pierwszy. Między 8:30 a 10:00 doszło do kolizji z użytkownikiem B, który nie mógł zaparkować na zajętym miejscu. Miejsce to zostało dodane do zbioru "blocked". O 10:00 użytkownik A zwolnił miejsce, umożliwiając użytkownikowi C parkowanie między 12:00 a 13:00. O 16:00 użytkownik A chciał ponownie zaparkować, jednak miejsce nie było dodane do "blocked", ponieważ użytkownik C opuścił je przed 16:00.

5 Testy

Przeprowadzono kilka rodzajów testów na stworzonej aplikacji. Dla samej części agentowej stworzono automatyczne testy integracyjne, natomiast testy end-to-end z aplikacją mobilną przeprowadzono manualnie.

5.1 Testy integracyjne

Część agentową możemy przetestować korzystając z API dostarczanego przez Orleans: `TestCluster`. Tworzy on testowy klaster, który możemy dowolnie skonfigurować. W naszym przypadku dodajemy

odpowiednie konfiguracje do przechowywania stanu aktorów i tworzymy fixture z biblioteki testów `xUnit`. Fixture umożliwia tworzenie testowego klastra dla każdej grupy uruchamianych testów, tak aby stan między grupami testów był nie zależny:

```
using Orleans.TestingHost;
using Xunit;

namespace Agenci.UnitTests;

public sealed class ClusterFixture : IDisposable
{
    public TestCluster Cluster { get; } = new TestClusterBuilder()
        .AddSiloBuilderConfigurator<TestSiloConfiguration>()
        .Build();

    public ClusterFixture()
    {
        Cluster.Deploy();
    }

    public void Dispose()
    {
        Cluster.StopAllSilos();
    }
}

file sealed class TestSiloConfiguration : ISiloConfigurator
{
    public void Configure(ISiloBuilder siloBuilder)
    {
        siloBuilder
            .AddMemoryGrainStorage("orchestratorStore")
            .AddMemoryGrainStorage("parkingStore")
            .AddMemoryGrainStorage("driverStore");
    }
}

[CollectionDefinition(Name)]
public sealed class ClusterCollection : ICollectionFixture<ClusterFixture>
{
    public const string Name = nameof(ClusterCollection);
}
```

Po stworzeniu testowego klastra możemy testować logikę naszej aplikacji za pomocą testów integracyjnych. Poniższy przykład przedstawia test głównego scenariusza w naszej aplikacji - rezerwacji parkingów. Na początku pobieramy referencję dla orkiestratora (informatora), przykładowego kierowcy i przykładowego zarządcy parkingu. Aktualizujemy informacje o parkingu i weryfikujemy, czy orkiestrator został o nich poinformowany. Próbujemy uzyskać oferty parkingu dla kierowcy, a następnie wybieramy pierwszą i jedyną ofertę od testowego parkingu. Na koniec weryfikujemy, czy zarówno zarządcy parkingu jak i kierowca posiada informacje dotyczącą właśnie utworzonej rezerwacji parkingu:

```
using Agenci.Abstractions;
using Orleans.TestingHost;
```

```

using Xunit;

namespace Agenci.UnitTests;

[Collection(ClusterCollection.Name)]
public class ReserveParkingTests(ClusterFixture fixture)
{
    private readonly TestCluster _cluster = fixture.Cluster;

    [Fact]
    public async Task ReserveParkingTest()
    {
        var orchestrator = _cluster.GrainFactory.GetGrain<IOrchestratorGrain>(Guid.Empty);
        var parking = _cluster.GrainFactory.GetGrain<IParkingUserGrain>("Test parking");
        var driver = _cluster.GrainFactory.GetGrain<IDriverUserGrain>("Test driver");

        // Create parking
        await parking.UpdateParkingInfo(new ParkingInfo
        {
            Key = "Test parking",
            Address = "Test address",
            Latitude = 0,
            Longitude = 0,
            MaxCapacity = 1
        });

        // Check if parking has been added to orchestrator
        var parkingsInRange = await orchestrator.GetParkingsInRange(0, 0);
        Assert.Single(parkingsInRange.Parkings);

        // Get offers for driver
        var offers = await driver.GetParkingOffers(0, 0,
            DateTime.Now, DateTime.Now.AddHours(2));

        // Check if there is an offer
        Assert.Single(offers);

        // Choose parking
        var parkingNumber = await driver.ChooseParking("Test parking");

        // Check if parking has been reserved for driver
        Assert.NotNull(parkingNumber);
        var history = await driver.GetReservedParkingHistory();
        Assert.Single(history);
        Assert.Equal(history[0].Key, parking.GetPrimaryKeyString());

        // Check if parking has been reserved for parking
        var parkingInfo = await parking.GetParkingDetailedInfo();
        Assert.Single(parkingInfo.Reservations);
        Assert.Equal(parkingInfo.Reservations[0].PlaceNumber, parkingNumber);
        Assert.Equal(parkingInfo.Reservations[0].DriverKey, driver.GetPrimaryKeyString());
    }
}

```

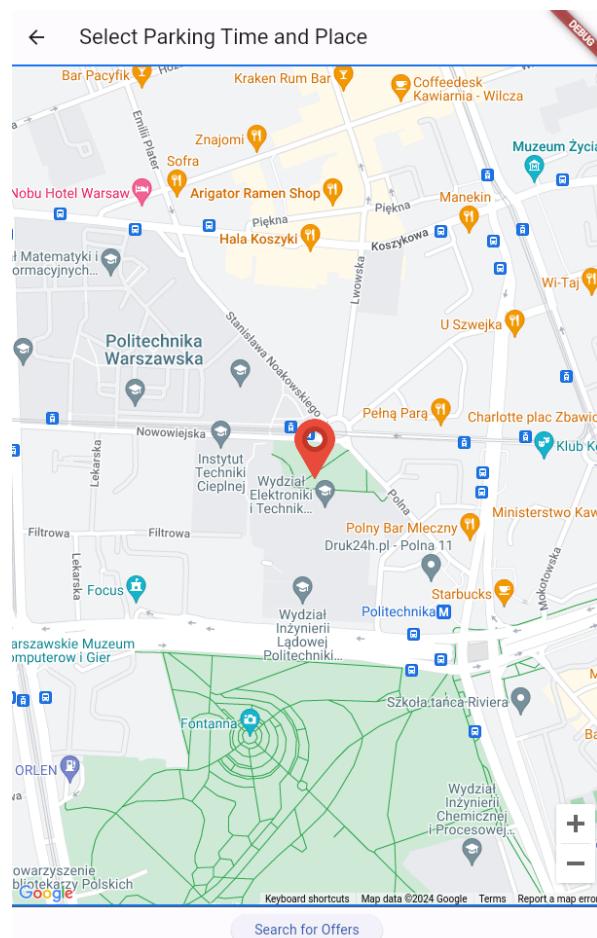
}

5.2 Testy manualne end-to-end

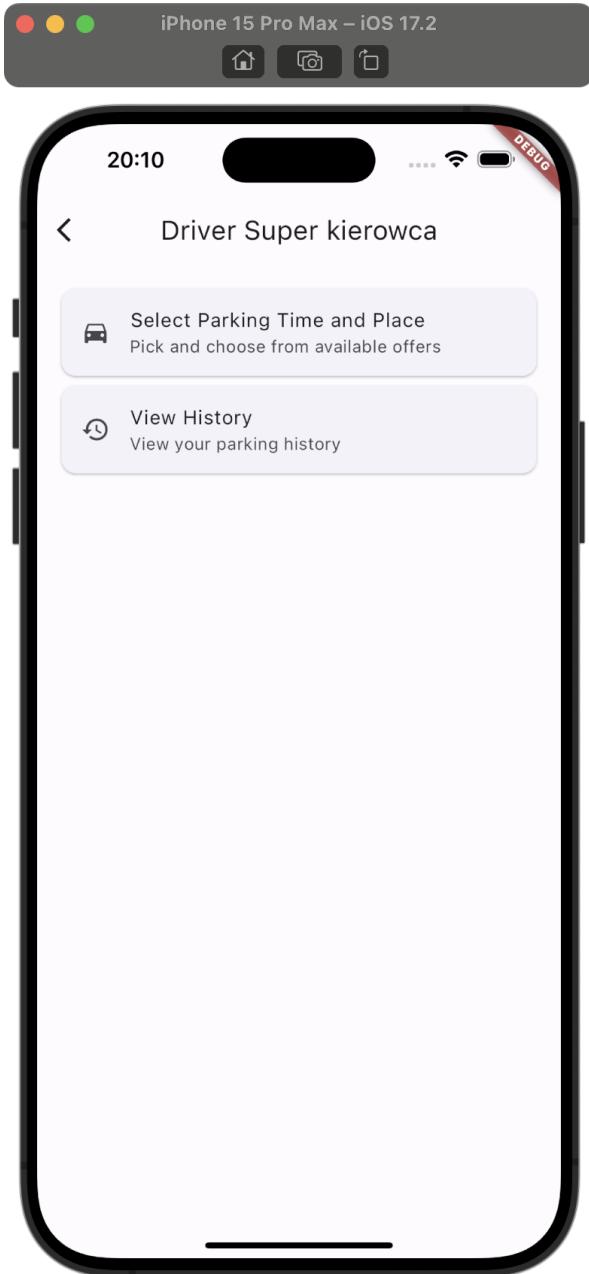
Testy manualne odbywały się głównie za pomocą aplikacji napisanej we frameworku Flutter. W aplikacji testowane były wszystkie możliwe scenariusze użycia zarówno dla użytkownika będącego parkującym jak i zarządzającym parkingiem. Podczas testów dodatkowo na bieżąco wykonywane były wywołania WebApi z poziomu interfejsu Swagger-owego. Dzięki nim sprawdzane były uzyskiwane wyniki zwrocone przez endpoint czy są zgodne z oczekiwanyymi rezultatami. Testy aplikacji odbywały się na kilku platformach. Były to:

- Android
- IOS
- aplikacja desktop-owa linux
- aplikacja desktop-owa MacOS
- aplikacja webowa (testy na przeglądarce Google Chrome)

Przykładowe ekranie z testów aplikacji znajdują się na rysunkach 10 oraz 11.



Rysunek 10: Przykładowy ekran z mapą do określania lokalizacji do wyszukiwania dostępnych parkingów.

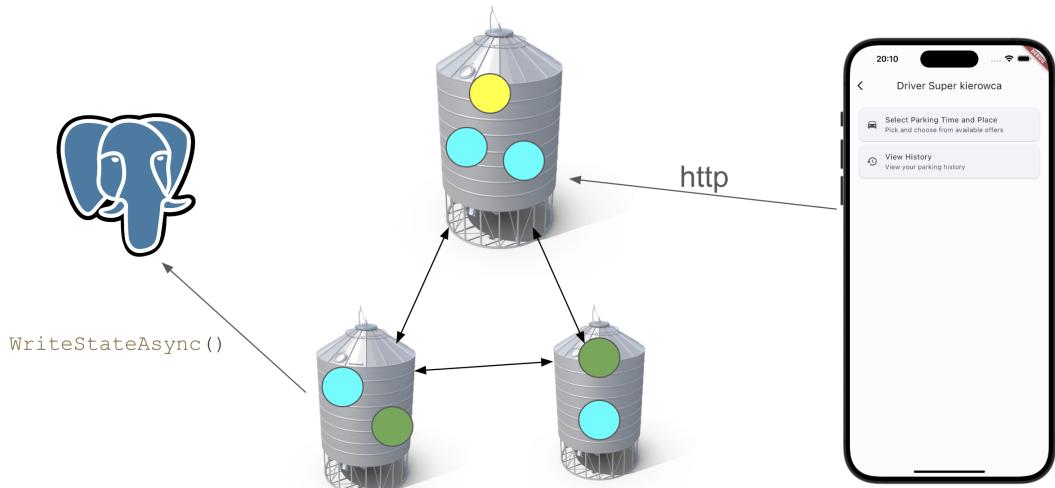


Rysunek 11: Przykładowy ekran podczas testów na platformie IOS.

6 Opis integracji systemu

Główny komponent - system agentowy zaimplementowany jest po stronie backendu. Framework Orleans odpowiada za komunikację pomiędzy agentami. Używa do tego wewnętrznych protokołów opartych o stos TCP/IP. Poszczególne instancje ziaren (Grains) są trzymane w silosach (Silos), które to odpowiadają za komunikację pomiędzy fizycznymi serwerami wewnętrz klastra.

Interakcja użytkownika z klastrem Orleans jest zapewniona poprzez aplikację frontendową, która poprzez Web API kontaktuje się z serwerem, w którym to działa Orleans. Bazą danych odpowiedzialną



Rysunek 12: Diagram integracji systemu

za przechowywanie stanu (stan ten potrzebny jest dla Orleans) jest PostgreSQL. Diagram komunikacji pomiędzy komponentami został przedstawiony na rysunku 12. Sama implementacja po stronie backendu jest łatwa - ze względu na natywne wsparcie ASP.NET. Web API zostało zaimplementowane zgodnie z metodyką REST, przykładowe metody API przedstawione są na rysunku 5. Zaletami takiego podejścia są: modularność, bezstanowość, prostota implementacji, duża przepustowość, ustandaryzowana integracja kolejnych platform czy typów klientów oraz natywne dostępna możliwość uwierzytelniania (z poziomu ASP.NET). Wady to z kolei: konieczność utworzenia frontendu celem prezentacji danych, brak możliwości komunikacji od serwera do klienta (należy użyć protokołu np. websockets, który będzie obok REST). Kolejna wada to duplikacja kodu - obiekty służące do transportu danych muszą zostać zaimplementowane po obu stronach (backend oraz wszystkie technologie frontendowe) - dzięki narzędziu Swagger oraz standardowi Open API możliwe jest automatyczne generowanie kodu klientów.

7 Case studies

Przykładowymi klientami oferowanej aplikacji są właściciele pojazdów, poszukujący najbliższych i najtańszych, wolnych miejsc parkingowych oraz właściciele parkingów w miastach.

W celu zobrazowania wykorzystania naszego systemu można wyobrazić sobie następującą sytuację. Klient A przyjeżdżający w godzinach porannych do dużego miasta, w którym wcześniej nie był, wybierając się do urzędu po przybyciu na miejsce przekonuje się, że przed urzędem nie ma wytyczonego parkingu. Powstaje problem, którego rozwiążanie można odnaleźć w naszej aplikacji. Klient A po zalogowaniu się do systemu parkingowego zgłasza chęć zaparkowania wokół jego lokalizacji. Klienci X,Y,Z będący właścicielami parkingów znajdujących się nieopodal urzędu po uprzedniej weryfikacji zajętości parkingów wysyłają oferty za miejsce parkingowe na ich parkingu. Klient A (kierujący pojazdem) wybiera parking X, Y lub Z, na którym zostawi swój pojazd. Czynnikiem decydującym o wyborze nie musi być tutaj tylko cena danego parkingu, ale również odległość parkingu od urzędu i czas, którym dysponuje nasz klient.

Beneficjentami całego rozwiązania są oczywiście właściciele samochodów poszukujący miejsc parkingowych oraz właściciele parkingów. Dzięki aplikacji parkujący mogą w krótkim czasie znaleźć wolne

miejsce parkingowe i nie martwić się gdzie zostawić samochód. Ponadto parkujący oszczędzają czas na niepotrzebnym poszukiwaniu miejsca parkingowego, ograniczając tym samym emisję spalin. Właściele parkingów z kolei zyskują na tym, że delegują kwestie rezerwacji i obsługi parkingu oddzielnemu systemowi, przy czym maksymalizują swoje dochody.

8 Braki i sposoby ulepszenia

Aktualnie implementacja korzysta z jednego informatora globalnie. Możemy zauważyć, że w przypadku dużego obciążenia aplikacji informator byłby szczególnie obciążony, gdyż uczestniczy w przekazaniu informacji o parkingach w danej okolicy. Jednak jesteśmy w stanie zapobiec temu problemowi poprzez stworzenie informatorów zbierających wiedzę o danym obszarze. Moglibyśmy takich informatorów identyfikować i komunikować się poprzez unikatowe klucze dla danego obszaru. Istnieje kilka algorytmów umożliwiających tworzenie kluczów dla regionów. Jednymi z popularniejszych są algorytmy Geohash [4] oraz Uber H3 [5], które tworzą kubelki o znanych kluczach na podstawie szerokości i długości geograficznej. Geohash tworzy siatkę prostokątów, natomiast Uber H3 heksagonów.



Rysunek 13: Podział San Francisco na heksagonalne obszary algorytmem H3.

Przy porozumiewaniu się z informatorem kierowca musiałby odpytać informatora odpowiedzialnego za ten sam sześcian w którym szuka parkingu oraz 6 sześcianów sąsiadujących z nim (rysunek 13). Informacje od sąsiadów przydają się, gdy znajdujemy się na krawędzi naszego sześcianu i potencjalnie parkingi z sąsiada mogą być bliżej naszego celu. Zauważmy, że zawsze będziemy odpytywać tą samą ilość informatorów, i koszt znajdowania sąsiadów jest stały, zatem odpytywanie odbywa się w czasie stałym.

9 Wnioski

Wyzwaniem, które stanowiło punkt wyjścia dla projektu naszego systemu, było narastające ograniczenie dostępu do miejsc parkingowych w miastach, co konsekwentnie przyczyniało się do wzrostu zatłoczenia ulic oraz emisji spalin.

Zaproponowane przez nas rozwiązanie opiera się na aplikacji zarządzającej miejscami parkingowymi poprzez dynamiczny cennik. Ma to zwiększyć dostępność miejsc parkingowych dla kierowców oraz umożliwić efektywne zarządzanie infrastrukturą parkingową przez miasta lub prywatnych właścicieli parkingów.

Aplikacja została zaprojektowana w myśl paradygmatu programowania aktorowego, odzwierciedlając trzy główne role systemu: Parkujący, Zarządzający Parkingiem i Informator. Aktorzy ci zostali utworzeni w formie Grain'ów z biblioteki Microsoft Orleans, z której pomocy skorzystaliśmy w projekcie. Innym ważnym wykorzystywanym zasobem był również framework ASP.NET. Całość funkcjonalności, czyli między innymi rejestracja i logowanie użytkowników, przegląd dostępnych miejsc parkingowych, rezerwacja miejsc, historia rezerwacji, obsługa zarządzających parkingami czy obsługa anulowania rezerwacji został wystawiony a webowym API.

Testy integracyjne oraz manualne end-to-end przeprowadzone na systemie potwierdziły poprawność jego działania, oraz pokazały, że jest on gotowy do użytku. Testy integracyjne pozwoliły zweryfikować poprawność interakcji dotyczących podstawowych funkcjonalności systemu takich jak, znalezienia odpowiedniego parkingu przez Parkującego czy przekazywanie statycznych informacji o parkingu przez Zarządzającego parkingiem do Informatora. Natomiast testy manualne pozwoliły zweryfikować interfejs użytkownika oraz funkcjonalności systemu w warunkach symulujących rzeczywiste jego wykorzystanie.

Zaprojektowany oraz zaimplementowany przez nas system spełnia postawione na początku projektu wymagania, a jego skuteczne działanie potwierdzają wykonane na nim testy. W rezultacie uzyskaliśmy kompleksowe, działające rozwiązanie, rozwiązujące problem narastającego ograniczenia dostępu do miejsc parkingowych w miastach.

Bibliografia

- [1] Piotr Wróblewski. „Ile jest samochodów w Warszawie? Ze spisu wynika aż 1,7 miliona. Od lat te liczby rosną”. W: *Warszawa Nasze Miasto* (4 lut. 2023). URL: <https://warszawa.naszemiasto.pl/ile-jest-samochodow-w-warszawie-ze-spisu-wynika-az-1-7/ar/c1-9194555> (term. wiz. 18. 10. 2023).
- [2] Zespół agenCI. *Repozytorium GitHub*. 2023. URL: <https://github.com/TortillaZHawaii/agenCI>.
- [3] Piotr Wróblewski. „Microsoft Orleans”. W: (12 czer. 2022). URL: <https://learn.microsoft.com/en-us/dotnet/orleans/overview>.
- [4] „Geohash”. W: (3 sty. 2024). URL: <https://en.wikipedia.org/wiki/Geohash>.
- [5] Isaac Brodsky. „H3: Uber’s Hexagonal Hierarchical Spatial Index”. W: *Uber Blog* (27 czer. 2018). URL: <https://www.uber.com/en-PL/blog/h3/>.