Agentowe i aktorowe systemy decyzyjne

Raport D: Opis integracji systemu

Zespół: AntyAgenci Szczelnie Decyzyjni

Skład:

- Wiktor Łazarski
- Rafał Kulus
- Michał Szaknis
- Piotr Czernecki

Repozytorium projektu:
https://github.com/pczernec/AASD_AntyAgenciSzczelnieDecyzyjni

Wstęp (dokumentacja z poprzednich etapów)

Opis problemu

Rozwój internetu i narzędzi do natychmiastowej komunikacji przyspieszył wymianę informacji między ludźmi, niezależnie od ich położenia. W dzisiejszych czasach jesteśmy w stanie przesłać znaczącą ilość informacji do osoby znajdującej się po drugiej stronie świata w mgnieniu oka. *Znacząca ilość informacji* jest tutaj rozumiana jako np. nagranie wideo jakiegoś wydarzenia. Dzięki temu informacje rozprzestrzeniają się w bardzo szybkim tempie. Jesteśmy w stanie dowiedzieć się o różnych wydarzeniach (w szczególności w naszej okolicy), tych pozytywnych jak i negatywnych. Wpływa to na nastrój i samopoczucie osób żyjących w pobliżu miejsc wspomnianych zdarzeń. Szczególnie odbijają się na naszym samopoczuciu wydarzenia negatywne, np. bójki, kradzieże, napaści. Bardzo często odwzorowywane to jest w zmianie pewnych parametrów funkcjonowania naszego organizmu, np. przyspieszone tętno, podniesione ciśnieine. Jednocześnie w dzisiejszym świecie znacząca liczba osób posiada inteligetne zegarki, które potrafią odczytywać różne parametry dotyczące funkcjonowania naszego organizmu. Wiele osób nieświadomie znajduje się w okolicach, w których dochodzi do ww. negatywnych zdarzeń, np. turyści, co uważamy, że jest problemem, który chcielibyśmy rozwiązać przez nas system.

Propozycja rozwiązania i koncept systemu

System bezpieczeństwa Twojej lokalizacji

Przedstawione problemy chcielibyśmy rozwiązać poprzez stworzenie systemu aktorowo-agentowego informującego użytkownika o ocenie jego lokalizacji na mapie z zaznaczonymi obszarami. Agentami naszego systemu byłyby urządzenia monitorujące stan zdrowia człowieka, np. smart watche, które komunikowałyby się ze sobą w celu określania potencjalnych lokalizacji, w których może występować zagrożenie dla człowieka. Odpowiednie dane prezentowane byłyby użytkownikowi w postaci powiadomień na telefon bądź na smart watcha. Dzięki naszemu rozwiązaniu użytkownik może uniknąć miejsc, w których występują bójki uliczne lub manifestacje, podczas których dochodzi do konfliktów, np. Marsz Niepodległości. Działanie systemu polegałoby na komunikacji z agentami znajdującymi się w pobliżu i odpytywaniu ich o "samopoczucie", czyli ocenę stanu użytkownika wyliczaną z danych biometrycznych. Na tej podstawie możliwe będzie wyznaczanie parametrów obszaru stosując np. uśrednianie wartości ze znajdujących się w danej części mapy agentów. Pozwoli to uniknąć komunikacji z centralnym serwerem zbierającym dane od wszystkich użytkowników. Całość można realizować poprzez komunikację jedynie między

agentami w danym obszarze. Każdy agent będzie posiadał stan swojego użytkownika do odpowiadania na pytania innych agentów oraz listę z innymi w pobliżu. Celem agenta będzie minimalizowanie czasu spędzanego na obszarach zagrożonych podczas wyznaczania drogi do celu użytkownika. Do rozwiązania określonego zadania można wykorzystać model wielowarstwowy, gdzie poszczególne warstwy mogą być przystosowane do rozwiązywania różnych wariantów problemu. Na przykład przypadek dla małej ilości danych można rozwiązać prostszymi algorytmami, które będą bardziej generalizować dane (np. nie będą niepotrzebnie rozróżniać obszarów o dużej i małej gęstości zebranych informacji).

Wymagania funkcjonalne

- Powiadomienie użytkownika o zagrożeniu na podstawie stanu jego i innych użytkowników w jego pobliżu.

Wymagania niefunkcjonalne

- Działająca sieć urządzeń komunikujących się z innymi urządzeniami w okolicy (Peer-to-Peer).
- Stabilność i działanie systemu niezależne od polityki oszczędzania energii i wydajności urządzenia użytkownika.
- Zapisywanie stanu mapy zagrożeń na wypadek chwilowego braku urządzeń w celu aproksymacji obecnych zagrożeń.

Scenariusz

1. Rola `State collector` okresowo pobiera informacje z czujników analizujących stan zdrowotny oraz lokalizację użytkownika i, jeśli stan się zmienił, zostaje on przesłany do roli `State broadcaster` w ramach komunikatu `USER STATE`.
2. Rola `State broadcaster` otrzymuje komunikat otrzymany od roli `State collector`. Następnie zapisuje otrzymany komunikat w bazie danych. W kolejnym kroku wiadomość jest przesyłana do roli `Danger notifier`, a jej zanonimizowana wersja jest rozgłaszana do innych agentów jako komunikat typu `ANONYMISED USER STATE`.
3. Jednocześnie, agent rozgłaszający wiadomość nasłuchuje w ramach roli `State receiver` na wiadomości rozgłaszane przez inne agenty. Rola ta agreguje odebrane komunikaty i okresowo sprawdza, czy stan agregacji się zmienił, a jeśli tak, przekazuje je do roli `Danger notifier` w ramach komunikatu `ANONYMISED USER STATE BATCH`.
4. Rola `Danger notifier` po otrzymaniu komunikatu `USER STATE` (od roli `State broadcaster`) lub `ANONYMISED USER STATE BATCH` (od roli `State collector`) oblicza na podstawie otrzymanych i historycznych danych nowy poziom zagrożenia użytkownika. W przypadku przekroczenia wartości krytycznej, użytkownik jest powiadamiany o prawdopodobnym zagrożeniu w jego okolicy.

Architektura

Repozytorium zawiera 4 paczki w folderze `packages`:

- `agents`
- `constants`
- `ploter`
- `xmpp_server`

Symulacja agentów jest uruchamiana za pomocą `docker-compose` poleceniem:

```
docker-compose up --force-recreate --build --remove-orphans
```

Skalowanie liczby agentów odbywa się poprzez modyfikację parametru `replicas` dla usługi `agent` w pliku `docker-compose.yml`. Jednocześnie należy również dostosować liczbę dostępnych portów hosta poprzez parametr `ports`, aby zakres był co najmniej tak duży jak liczba agentów.

Zalecana wersja narzędzia `docker-compose` to 1.29.2 ze względu na usunięcie w wersjach wyższych słowa kluczowego `extends`.

Wynik działania systemu agentowego prezentowany jest na bieżaco w konsoli.

`agents`

Paczka `agents` zawiera implementację agenta w języku Python 3.9 z wykorzystaniem biblioteki SPADE v3.2.2.

Niestety SPADE nie jest kompatybilny z Pythonem >= 3.10 ze względu na:

```
TypeError: As of 3.10, the *loop* parameter was removed from Lock() since it is
no longer necessary
```

`constants`

Paczka `constants` zawiera stałe wykorzystywane przez paczki `agents` oraz `ploter`. Powstała ona, aby zdeduplikować kod (w przeciwnym razie mogłoby dojść do desynchronizacji parametrów między paczkami) oraz aby mieć jedno miejsce dla wszystkich parametrów symulacji.

`xmpp_server`

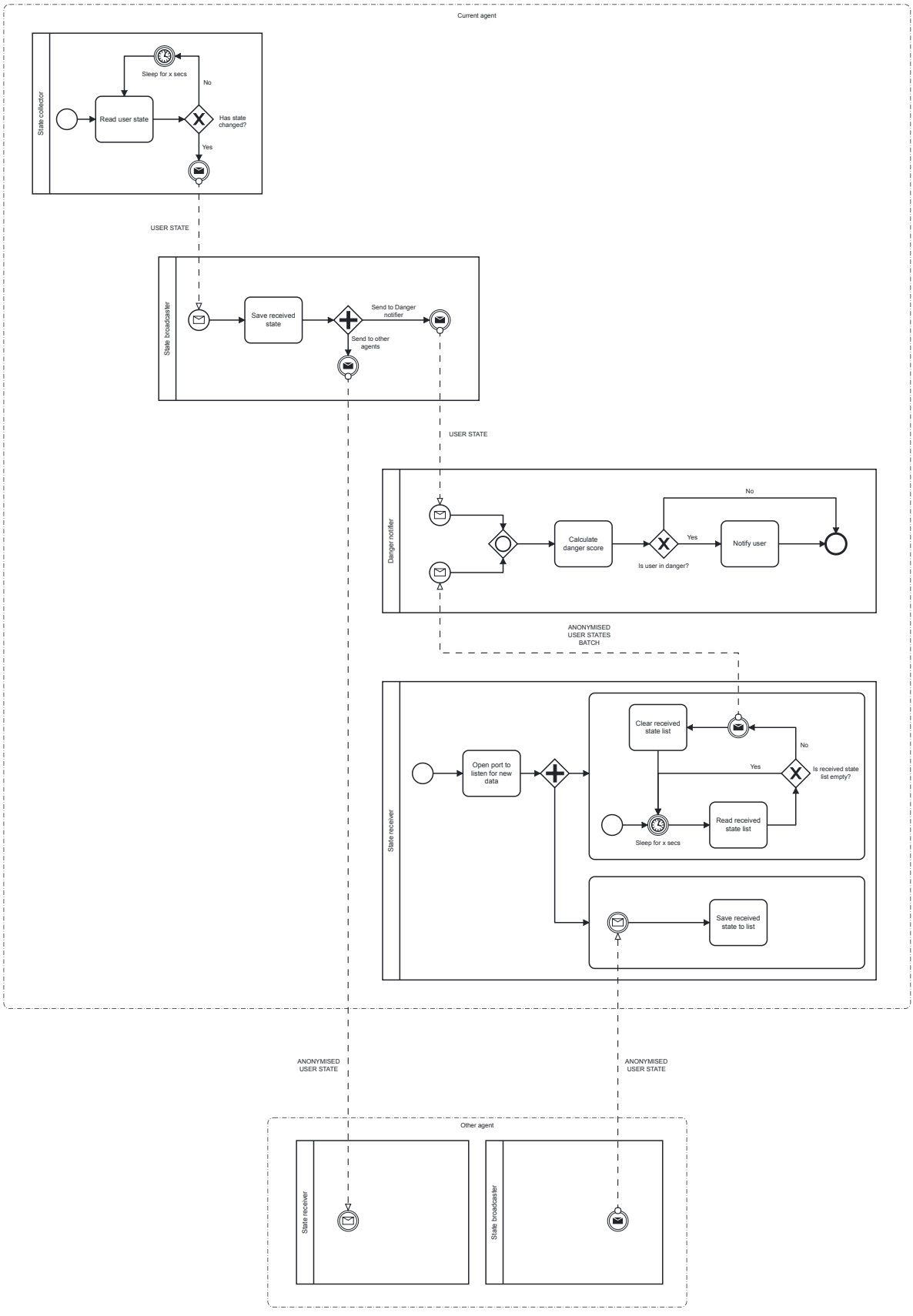
Paczka `xmpp_server` zawiera serwer XMPP zaimplementowany przy pomocy oprogramowania Prosody, które zostało napisane w języku Lua.

`ploter`

Paczka `ploter` zawiera interfejs graficzny wizualizujący stan agentów. Aby uruchomić wizualizację, która jest niezależna od środowiska kontenerowego, należy wywołać:

```
python3 ./packages/ploter/mainwindow.py
```

Schemat agenta



Sposób implementacji agentów

Agent został zaimplementowany w pliku `packages/agents/spade_agents/sm_agent.py` w ramach klasy `SmartWatchAgent`. Zawiera on 4 role / zachowania:

- **StateCollector (PeriodicBehaviour)** --- Odczytywanie (zmiany) stanu użytkownika
- **StateBroadcaster (CyclicBehaviour)** --- Wysyłanie zanonimizowanego stanu użytkownika do urządzeń w okolicy

- **StateReceiver (CyclicBehaviour)** --- Nasłuchiwanie na informacje o stanie od agentów w okolicy
- **DangerNotifier (CyclicBehaviour)** --- Informowanie użytkownika o pobliskim zagrożeniu

Komunikacja między rolami

Każda rola, która spodziewa się wiadomości od innych ról, posiada zmienną będącą kolejką (obiekt klasy `asyncio.queues.Queue`), która wypełniana jest w przypadku odebrania przez rolę wiadomości.

Komunikat --- `USER STATE`

Jest wysyłany przez `StateCollector` a do `StateBroadcaster` a i zawiera stan użytkownika. Stan użytkownika przesyłany jest przy pomocy klasy `UserState`. Klasa `UserState` zawiera informacje na temat lokalizacji agenta (zmienne `x`, `y`) jego zachowanie w środowisku (`angle`, `velocity`) oraz jego stanu zdrowia/psychicznego (zmienna `hp`). Deklaracja klasy zaprezentowana została poniżej.

```
@dataclass
class UserState:
    x: float
    y: float
    hp: float
    angle: float
    velocity: float
```

Komunikat --- `ANONYMISED USER STATE BATCH`

Jest wysyłany przez `StateReceiver` a do `DangerNotifier` a i zawiera listę obiektów `UserState`. Komunika ten służy głownie do zagregowania w systemie stanów użytkowników z okolicy zainteresowania agenta, który taki komunikat otrzymuje.

Komunikacja między agentami

Agenty komunikują się między sobą przy pomocy serwera XMPP. Ponieważ istnieje u nas tylko jeden typ agenta, wiadomości wysyłane przez pewną instancję powinny być rozgłaszane do całej reszty (nie istnieje u nas komunikacja unicastowa).

Komunikat --- `ANONYMISED USER STATE`

Jest wysyłany przez `StateBroadcaster` a do innych agentów (roli StateReceiver) i zawiera zanonimizowany `UserState`, czyli stan użytkownika przeznaczony do wysyłania do innych agentów.

W naszym systemie przyjeliśmy, że językiem treści komunikatów jest format JSON. Powyższy komunikat wykorzystuje performatywę *inform*.

Rozgłaszanie wiadomości

W tym celu został do serwera XMPP dodany mod_broadcast, który dodaje funkcjonalość rozgłaszania wiadomości. Domyślnie wymaga on jednak *whitelisty* zawierającej użytkowników, którzy mają uprawnienia do rozgłaszania wiadomości. Ponieważ początkowo zakładaliśmy, że każdy agent będzie korzystał z własnego konta XMPP, utrzymywanie *whitelisty* byłoby uciążliwe i mogłoby przysporzyć w przyszłości problemów. Postanowiliśmy zatem zmodyfikować wspomniany mod i dostosować go do naszych potrzeb, tj. każda wiadomość jest zawsze rozgłaszana, niezależnie od adresata i nadawcy.

Zmodyfikowany kod znajduje się w `packages/xmpp_server/prosody_modules/mod_broadcast/mod_broadcast.lua`.

Rejestracja użytkowników

Początkowo każdy agent miał korzystać z własnego konta na serwerze XMPP. Planowaliśmy wykorzystać fakt, że po dodaniu do konfiguracji Prosody parametru `allow_registration = true`, Spade powinien bez problemu być w stanie zarejestrować konto dla agenta podczas podłączania do serwera. Niestety wyżej wspomniany parametr nic nie zmienił, nie działał.

Oznaczało to, że konta musiały być zarejestrowane przed startem agentów z wewnątrz kontenera z serwerem. Nie byłoby to problemem, gdyby nie fakt, iż chcemy mieć możliwość skalowania liczby agentów do dowolnej (dużej) wartości. Można by utworzyć za wczasu *x* kont, gdzie *x* byłoby maksymalną liczbą obsługiwanych agentów. Byłoby to jednak rozwiązanie bardzo nieeleganckie i problematyczne, gdyż byłyby tworzone nadmiarowe konta, a ponad to musiałyby być one tworzone poprzez ręczne uruchomienie skryptu lub automatyczne uruchamianie go przy każdym starcie serwera. W dodatku każdy kontener musiałby *jakoś* wiedzieć, z którego konta ma skorzystać.

Ostatecznie podjęliśmy decyzję, że każdy agent korzysta z tego samego konta XMPP, a w metadanych jest przekazywany parametr `agent_id`, który zawiera ID kontenera, w którym uruchomiony jest dany agent. Wspomniane ID jest wykorzystywane do odfiltrowania wiadomości od *samego siebie*.

Wspomniane wyżej rozwiązanie działa ze względu na zastosowanie moda do rozgłaszania, który podmienia wartość pola `from` na własny adres.

Wykorzystane standardy

W naszym projekcie do budowania systemu komunikacji pomiędzy agentami wykorzystaliśmy wspomnianą już wcześniej bibliotekę SPADE - Smart Python multi-Agent Development Environment. Biblioteka ta jest jedną z pierwszych bibliotek napisanych w języku Python do tworzenia systemów agentowych. Udostępnia ona szereg różnych mechanizmów (np. klasa `Message`) pozwalających na implementację systemu komunikacji oraz implementację poszczególnych ról/zachowań agentów pracujących w systemie. Jak możemy wyczytać w dokumentacji

```
The small proof of concept evolved into a full-featured FIPA platform, and the
new SPADE name was coined.
```

Mechanizmy pozwalające na przesyłanie wiadomości między agentami są oparte na standardzie Foundation for Intelligent Physical Agents (FIPA). W związku z powyższym wybór standardu komunikacji jaki zastosowaliśmy w naszym systemie był oczywisty i jest to standard FIPA.

Napotkane problemy

W trakcie prac nad projektem natrafiliśmy na szereg różnych problemów. Poniżej wymieniamy te, które najbardziej wpłynęły na naszą prace:

1) pakiet `spade` wymaga użycia Pythona w wersji <= 3.9
2) znajdowanie listy agentów, którzy znajdują się w obszarze agenta.
3) określanie poziomu zagrożenia obszaru
4) szereg problemów związanych z opracowanie architektury opartej na konteneryzacji agentów występujących w systemie

5) stosowanie programowania asynchronycznego było bardzo trudne w debuggowaniu systemu
6) przekazywanie danych do wizualizacji.

Oprócz wymienionych wyżej problemów, nie zabrakło również problemów z samą koncepcją na realizacje projektu. Długo zastanawialiśmy się jaki będzie najlepszy sposób na implementacje. Rozważaliśmy dwa podejścia:

1) konteneryzacja każdego agenta występującego w systemie
2) opracowanie aplikacji, która zarządzałaby wszystkimi agentami i tworzyła środowisko ich pracy.

Ostatecznie po rozważeniu wszystkich za i przeciw zdecydowaliśmy się na piewsze rozwiązanie, gdyż jest ono bardziej uniwersalne i łatwiej byłoby je wdrożyć w środowisku realnym jeśli mielibyśmy dostęp do odpowiednich urządzeń.

Dużym problemem było dołączenie do implementacji agenta websocketów w celu komunikacji z narzędziem do wizualiacji. Problemem okazał się sposób napisania SPADE, cały framework opiera się o bibliotekę asyncio i udostępnia użytkowikowi interfejs poprzez możliwośc nadpisania funkcji. Powodowało to sytuacje, gdzie losowo po pewnym czasie pętla nasłuchująca na połaczenia była przerywana wyjątkiem. Nie dokońca wiemy dlaczego, gdyż debuggowanie kodu asynchornicznego nie jest proste ani przyjemne. Problem udało się rozwiązać poprzez uruchomienie drugiej pętli przetwarzającej zdarzenia w oddzielnym watku i wykorzystanie kolejki do komunikacji między agentami a websocketami.

GUI

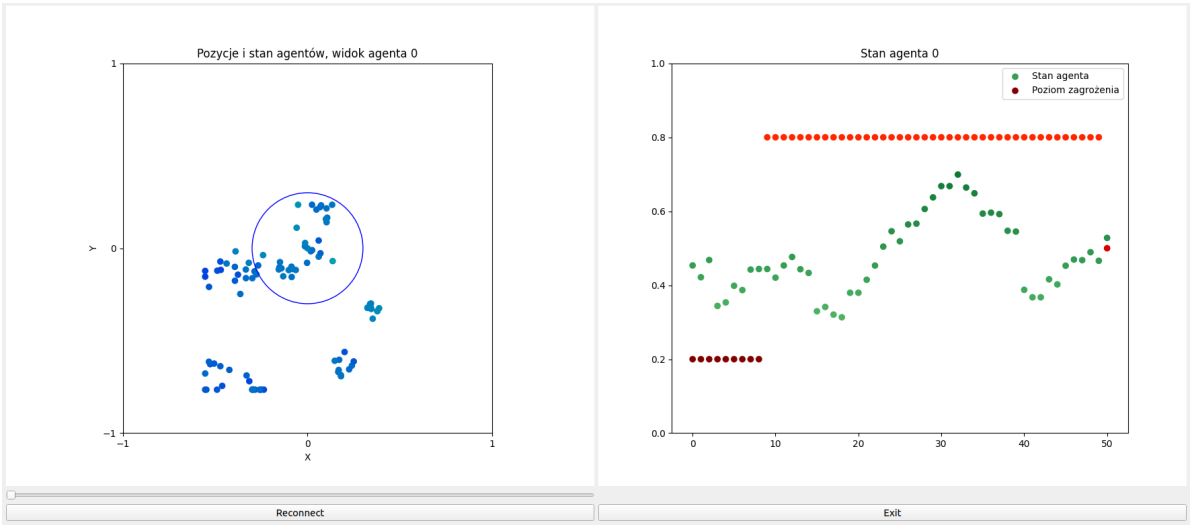
W celu wizualizacji działania symulacji został utworzony interfejs graficzny pozwalający śledzić w czasie rzeczywistym stan agentów.

- Aplikacja okienkowa została napisana w `Qt` z użyciem biblioteki `Pyside2`.
- Wykresy zostały utworzone z użyciem biblioteki `matplotlib`.
- Połączenie z wybranym agentem odbywa się poprzez `WebSocket`.

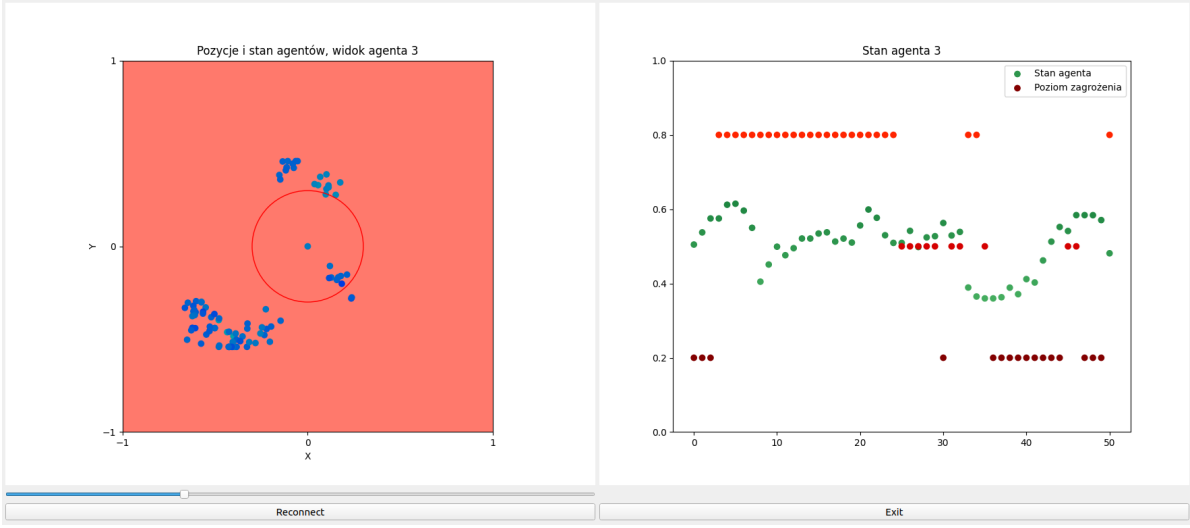
Użytkownik ma możliwość wyboru konkretnego agenta za pomocą suwaka. Okno zawiera dwa wykresy. Pierwszy z wykresów przedstawia centralnie pozycję wybranego agenta wraz z narysowanym konturem otoczenia oraz pozycje pozostałych agentów. W przypadku braku zagrożenia kontur przyjmuje kolor zielony, podczas średniego stanu zagrożenia kolor niebieski, a dla najwyższego stopnia zagrożenia kolor czerwony przy jednoczesnej zmianie koloru tła. Drugi z wykresów przedstawia stan zdrowia oraz poziom zagrożenia agenta w czasie.

Przykładowe zrzuty ekranu z działania

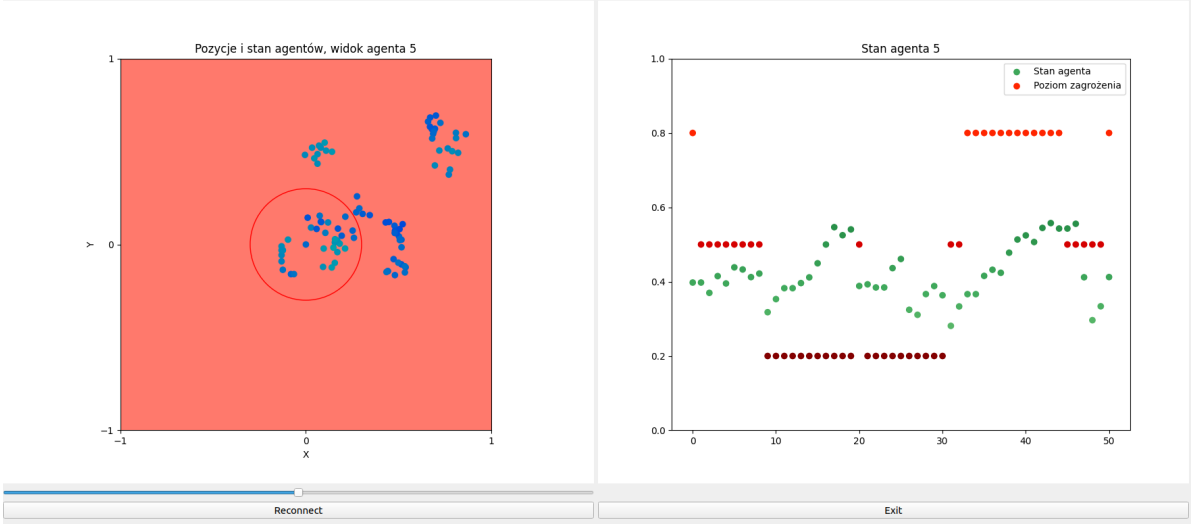
Agent 0:



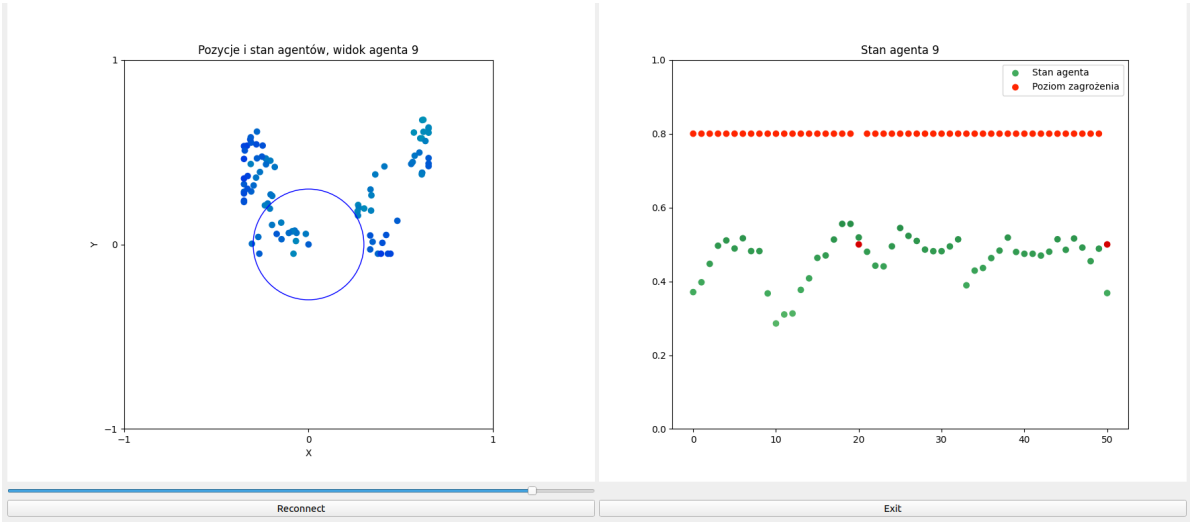
Agent 3:



Agent 5:



Agent 9:



Stosowane algorytmy

Algorytm klasyfikacji obszarów o podwyższonym zagrożeniu

W naszym systemie, oprócz opisanych wcześciej mechanizmów zarządzającymi komunikacją pomiędzy agentami, opracowaliśmy algorytm dokonujący klasyfikacji obszaru w jakim znajduje się nasz agent. Implementacja algorytmu znajduje się w klasie `DangerNotifier`, będącą klasą wewnętrzną klasy `SmartWatchAgent`. Praca algorytmu rozpoczyna się w momencie otrzymania przez rolę odpowiedniego komunikatu - zmienna klasy `DangerNotifier` typu `asyncio.queues.Queue` zostanie wypełniona odpowiednim obiektem. Treścią komunikatu może być zarówno stan użytkownika (agenta) jak i lista stanów użytkowników znajdujących się w obszarze rozgłaszania wiadomości (przez rolę `StateBroadcaster`). Jeśli wiadomość zawiera stan użytkownika, żadna informacji dotyczącej otoczenia nie jest aktualizowana. W przypadku dostarczenia listy zawierającej stany użytkowników algorytm wykonuje szereg czynności mających na celu dokonanie ostatecznej klasyfikacji obszaru pod względem możliwości wystąpienia w nim zagrożenia.

W pierwszym kroku na podstawie lokalizacji przeprowadzana jest filtracja agentów, która znajduje tylko agentów w promieniu naszego obszaru zainteresowania. W naszym systemie, agenci poruszają się na wirtualnej mapie, określonej na płaszczyźnie (x, y), gdzie zarówno x jak i y należą do przedziału $[0, 1]$. Promień, określający okrąg wyznaczający obszar w którym znajduje się nasz agent, jest parametrem algorytmu. Na podstawie A/B testów dokonanych przy użyciu naszej aplikacji do wizualizacji ustaliliśmy, że najlepszą wartością promienia będzie liczba 0.3.

Po przefiltrowaniu listy, na podstawie stanów agentów w określonym przez promień obszarze, algorytm dokonuje analizy tego obszaru w celu określenia poziomu zagrożenia. Analiza polega na zliczeniu ilości agentów, których stan zdrowia/psychiczny wydaje się wykraczać poza określoną normę. Następnie, na podstawie ilości takich agentów klasyfikujemy obszar na **bezpieczny**, **średnio bezpieczny** lub **niebezpieczny**. Parametery zmian klasy obszaru, podobnie jak jego promień, zostały określone przez nas na podstawie A/B testów przeprowadzonych za pomocą naszej wizualizacji. Po przeprowadzonych eksperymentach najlepsze parametry zmiany prezentują się następująco:

- jeśli ilość agentów, których stan zdrowia wykracza poza normę, znajduje się w przedziale $[0, 1]$ to obszar agenta określony zostanie jako **bezpieczny**
- jeśli ilość agentów, których stan zdrowia wykracza poza normę, znajduje się w przedziale $(1, 3]$ to obszar agenta określony zostanie jako **średnio bezpieczny**
- jeśli ilość agentów, których stan zdrowia wykracza poza normę, jest większa niż 3 to obszar agenta określony zostanie jako **niebezpieczny**.

Ze względu na to, że nie mamy dostępu do fizycznych urządzeń to monitorowanie stanu zdrowia agenta zostało zrealizowane w sposób wprowadzenia do systemu dodatkowej zmiennej losowej, z przedziału $[0, 1]$, która określa ten stanu. Zmienna aktualizowana jest w każdym kroku działania systemu, co pozwala na obserwowanie zmian zachowań całego systemu agentowego. W naszym systemie uznajemy, że stan zdrowia agenta jest zagrożony w momencie kiedy współczynnik przekracza wartość 0.4.

W ostatnim kroku algorytmu tworzona jest wiadomość wysyłana do systemu odpowiedzialnego za wyświetlenie rezultatu analiz i ewentualnie wyświetlenie odpowiedniego komunikatu jeśli zachodzi taka potrzeba. Po wysłaniu wiadomości algorytm klasy `DangerNotifier`a, kończy pracę i oczekuję na kolejną wiadomość, dla której przeprowadzi analizę.

Pseudokod algorytmu:

```
wait_for_message()
```

```
if message is UserState:
    send_message_with_updated_user_state()
elif message is list:
    agents_in_zone = get_agents_in_my_zone(radius=0.3)

    unhealthy_state_cnt = 0
    for agent in agents_in_zone:
        if agent.health_state > 0.4:
            unhealthy_state_cnt++

    if unhealthy_state_cnt <= 1:
        zone_class = SAFE
    elif unhealthy_state_cnt <= 3:
        zone_class = MEDIUM_SAFE
    else:
        zone_class = UNSAFE

    send_message_with_updated_user_state_and_updated_env_class()
```

Generowanie nowych położeń agentów

Szybkość zmian położenia agenta w kolejnych krokach określona jest poprzez współczynnik w postaci zmiennej globalnej `SIM_STEP`. Doświadczalnie ustalono jej wartość na 0.1, tak aby zmiany położenia były zauważalne ale nie nazbyt duże.

Wyznaczanie położenia agenta w kolejnych krokach symulacji dobywa się następująco:

1. Generowany jest nowy kierunek ruchu w radianch (oznaczmy jako `new_angle`) oraz aktualizowana jest prędkość zgodnie z następującą zależnością: `velocity = 1 - min( (2 * π) - abs(new_angle - angle), abs(new_angle - angle) ) / π` gdzie `angle` to dotychczasowy kierunek.
2. Obliczenie nowych koordynatów:
   `x += sin(new_angle * π) * velocity * SIM_STEP`
   `y += cos(new_angle * π) * velocity * SIM_STEP`
3. Zastąpienie dotychczasowego kierunku nowym kierunkiem.

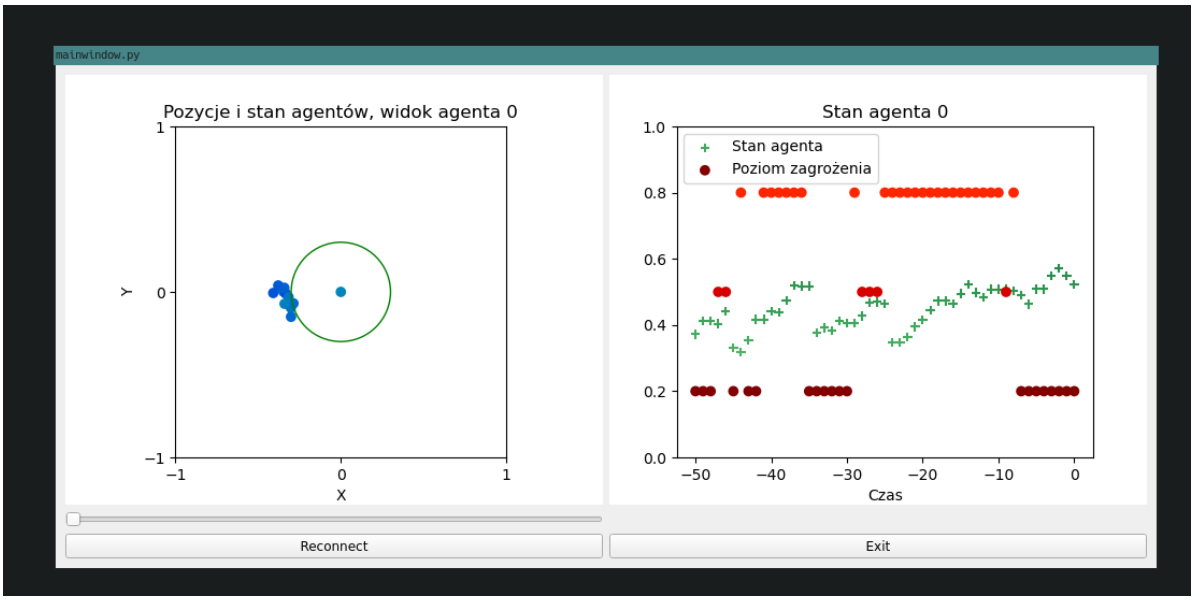
Integracja systemu

Integracja systemu została przeprowadzona przez nas w części dotyczącej implementacji systemu - *Raport C*. W ramach integracji opracowaliśmy aplikację z graficznym interfejsem użytkownika, z którą zintegrowaliśmy nasz system agentowy. Przykładowe działanie aplikacji oraz jej opis zostały przedstawione w *Raporcie C* w rozdziale *GUI*.

Aplikacja z graficznym interfejsem użytkownika służyła nam jako narzędzie do przeprowadzania testów działania naszego systemu agentowego. Ze względu na losowość poruszania się aktorów w trakcie działania aplikacji, zaobserwowanie pewnych sytuacji i reakcji naszego systemu na nie stało się trudne. Dzięki temu nauczyliśmy się, że w przyszłości jednak poza takimi testami warto opracować sobie zestaw scenariuszy, w których może znaleźć się nasz system i obserwować, czy zachodzi pożądana reakcja systemu. Niemniej jednak, zdefiniowaliśmy zestaw sytuacji, które według nas były krytyczne do stwierdzenia, że system działa poprawnie i obserwowaliśmy je w naszej aplikacji z graficznym interfejsem użytkownika. Zdefiniowane przez nas sytuacje to:

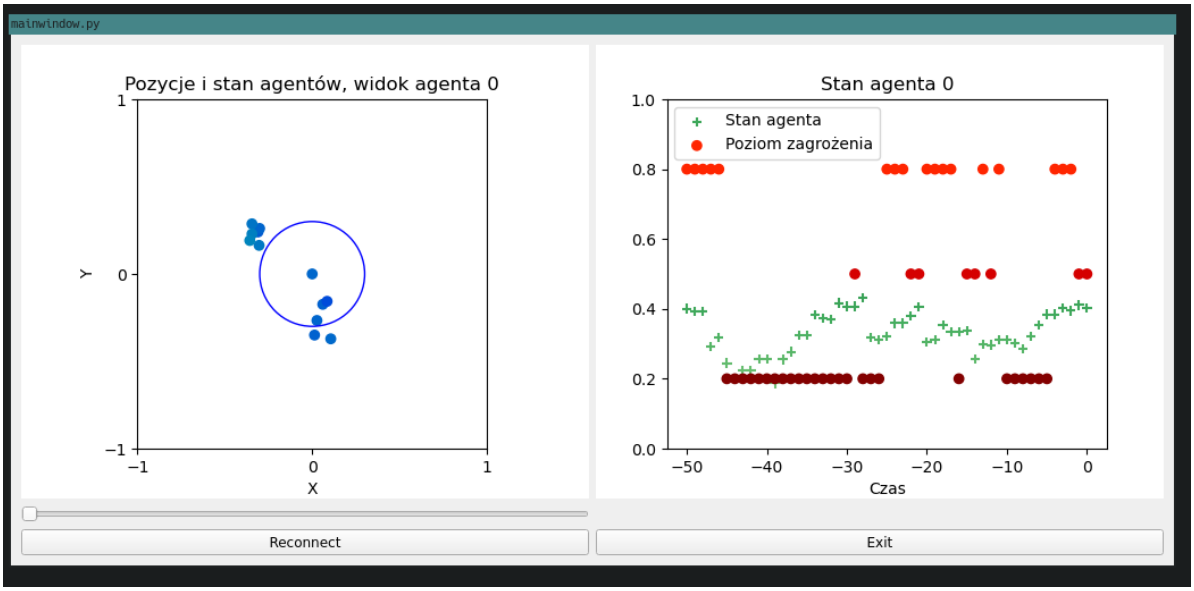
1. Agent znajduje się w sytuacji, w której system nie zgłasza zagrożenia w jego okolicy
2. Agent znajduje się w sytuacji, w której system wykrył średnie zagrożenie w jego okolicy
3. Agent znajduje się w sytuacji, w której system wykrył duże zagrożenie w jego okolicy

Agent znajduje się w sytuacji, w której system nie zgłasza zagrożenia w jego okolicy



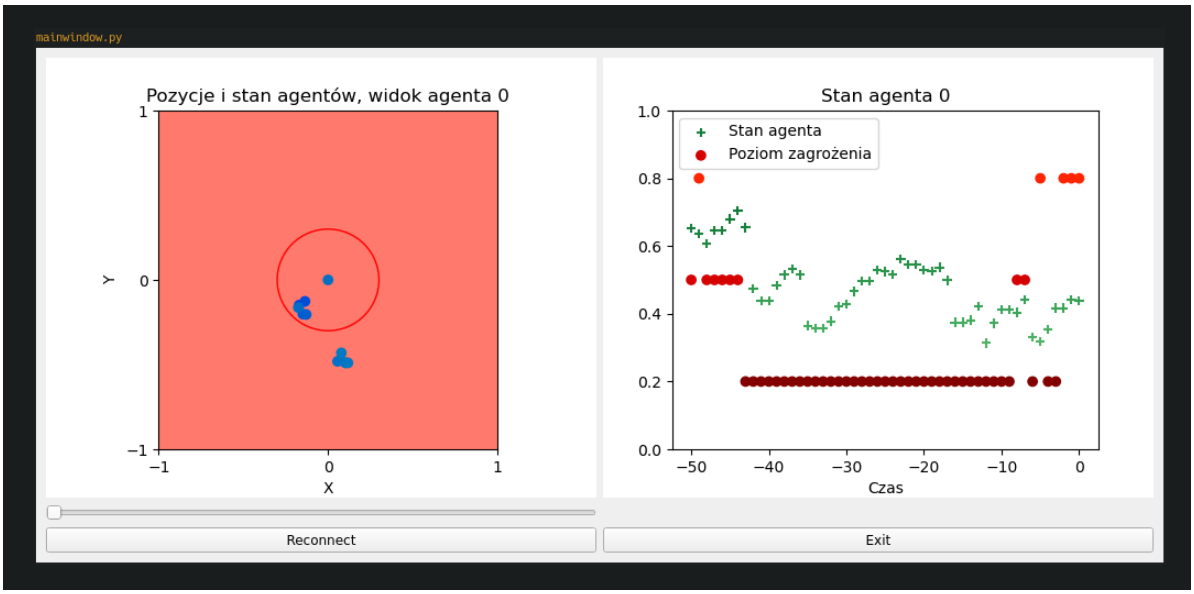
Wizualizacja przedstawiona powyżej pokazuje agenta znajdującego się w zielonym kółko. Oznacza to, że system stwierdza, iż obszar w którym znajduje się agent jest bezpieczny. Na podstawie tej sytuacji możemy stwierdzić, że system działa poprawnie, ponieważ widzimy na niej, że ilość innych agentów w obszarze analizowanego agenta jest zbyt mała, aby system miał szansę określić obszar jako niebezpieczny. Dodatkowo, działanie systemu jest poprawne, gdyż nie zgłasza on żadnego powiadomienia o niebezpieczeństwie. Takie powiadomienie, w naszej aplikacji, wizualizujemy poprzez zmianę koloru tła na czerwony (patrz opis sytuacji *Agent znajduje się w sytuacji w której system wykrył duże zagrożenie w jego okolicy*).

Agent znajduje się w sytuacji, w której system wykrył średnie zagrożenie w jego okolicy



W sytuacji średniego zagrożenia, okrąg reprezentujący obszar, w którym znajduje się agent, zmienia kolor na niebieski. Jest to dodatkowo zaprezentowane na wykresie po prawej stronie, gdzie widać skokową zmianę poziomu zagrożenia do wartości 0.5.

Agent znajduje się w sytuacji w, której system wykrył duże zagrożenie w jego okolicy



Sytuacja, w której agent znajduje się w niebezpieczeństwie, a użytkownik jest o nim powiadamiany, na wizualizacji jest przedstawiona poprzez ustawienie czerwonego tła, a na prawym wykresie poziom zagrożenia ma wartość szczytową (≥0.8).

Podsumowanie

W ramach projektu udało nam się opracować i zaimplementować system agentowy, który chroni użytkownika przed pojawieniem się w obszarze o podwyższonym ryzyku. Praca nad projektem pozwoliła nam na nauczenie się wielu aspektów tworzenia systemów agentowych. Poznaliśmy przede wszystkim metodykę oraz niezbędne narzędzia bez których praktyczne wykonanie systemu byłoby niemożliwe. Opracowany przez nas system, pomimo tego, że nie został wdrożony w rzeczywistym środowisku produkcyjnym (na smartwatchach użytkowników), wydaje nam się skończonym produktem.