SerShare

Raport projektowy, wersja z 09.12.2018

Dziennik zmian:

1. 24.10.2018 - utworzenie dokumentu
2. 25.10.2018 - dodanie Danych projektu
3. 26.10.2018 - dodanie Części A
4. 12.11.2018 - dodanie Części B, modyfikacja Części A (zaznaczona na żółto)
5. 09.12.2018 - dodanie Części C, modyfikacja Części B w oparciu o komentarze dr Pałki (zaznaczone na zielono)

1. Dane projektu:

a. Tytuł projektu: SerShare (dawne MilkShare)
b. Numer zespołu: Zespół A
c. Skład z podziałem na role:
   i. **Maciej Wiraszka** - kierownik projektu + specjalista ds. standardów FIPA
   ii. **Julita Ołtusek** - specjalistka ds. algorytmów agentowych
   iii. **Szymon Borodziuk** - specjalista ds. architektury
   iv. **Michał Starosta** - specjalista ds. jakości agentów
   v. **Jędrzej Kalisiak** - specjalista ds. implementacji
   vi. **Cezary Modzelewski** - specjalista ds. implementacji
d. Repozytorium: https://github.com/Zwirek009/SerShare

2. Część A: Identyfikacja problemu

a. Problem

Ludzie marnują jedzenie, kupując produkty spożywcze, które wykorzystują nie do końca. Poza tym, marnują czas na robienie zakupów. Dodatkowo zanieczyszczają środowisko, bo każdy jedzie po zakupy, często własnym samochodem.

b. Propozycja rozwiązania

System zarządzania zasobami żywieniowymi w społeczności lokalnej (np. w budynku lub na osiedlu). System wykorzystuje połączone, inteligentne lodówki, które monitorują własną zawartość i na tej podstawie automatycznie zamawiają jedzenie przez internet. Produkty dostarczane są zbiorczo, okresowo. System umożliwia także wymianę produktów z sąsiadem, w przypadku kiedy nam coś zostało (np. wykorzystaliśmy niecałą zawartość kartonu z mlekiem), a sąsiadowi akurat było to potrzebne.

c. Koncepcja rozwiązania

Rodzaje agentów:

1. **Fridge Agent (agent lodówkowy)** - agent zajmujący się zapewnieniem wszystkich wymaganych towarów w lodówce i obsługą mechanizmów dzielenia się produktami.

   Zadania:
   - Monitoring aktualnego stanu jedzenia w lodówce
   - Tworzenie planu zawartości danej lodówki
   - Obsługa mechanizmu udostępniania jedzenia

   Komunikacja z:
   - Mobile Agent - uzyskiwanie informacji o planowanym zapotrzebowaniu na produkty
   - Storekeeper Agent - wymiana informacji o przewidywanej ilości produktów w lodówce w czasie, informowanie o aktualnych zapasach.
   - Fridge Agent - uzgadnianie wymian produktów.

   Cel:
   - Agent dąży do tego aby lodówka posiadała wymagane produkty

   Umiejscowienie:
   - Agent lodówkowy będzie zainstalowany na każdej lodówce w ilości jeden

2. **Mobile Agent (agent mobilny)** - agent wydzielony tylko do wprowadzania zmian w planie jedzeniowym lub do zgłaszania potrzeb in-time

   Zadania:
   - Pobieranie od użytkownika zmian planu jedzeniowego
   - Pobieranie od użytkownika zgłoszeń in-time

   Komunikacja z:
   - Fridge Agent - wysłanie informacji o zmianach planu jedzeniowego i zgłoszeń in-time

   Cel:
   - Przekazanie agentowi Fridge informacji o planach zakupowych oraz nagłych potrzebach

   Umiejscowienie:
   - Urządzenia mobilne

3. **Storekeeper Agent** - agent zajmujący planowaniem zakupów produktów na podstawie stanu lodówki oraz planów żywieniowych.

   Zadania:
   - Zbieranie planów żywieniowych
   - Planowanie zakupów
   - Zbieranie informacji o stanie lodówki

   Komunikacja z:
   - Fridge Agent - wymiana informacji o aktualnych zasobach
   - Merchant Agent - składanie zamówień na zakupy
   - Mobile Agent - pobieranie informacji o planach żywieniowych

   Cel:
   - Zapewnienie dostępności produktów ustawionych w planach żywieniowych

   Umiejscowienie:
   - Każda lodówka

4. **Merchant Agent** - agent reprezentujący sklep

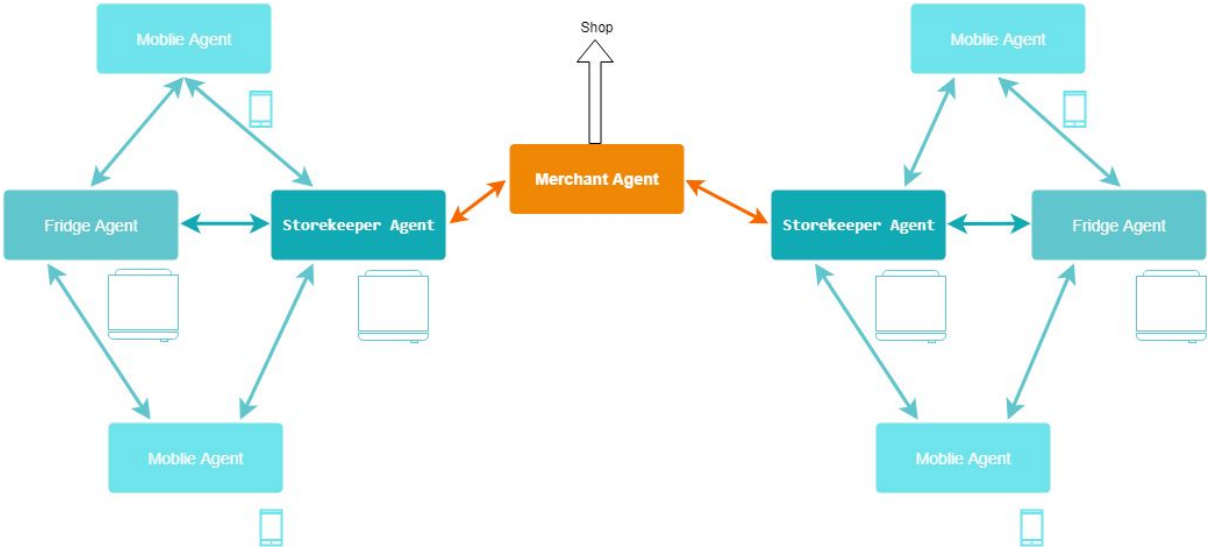
   Zadania:
   - Zbieranie zamówień od wielu Storekeeper Agent
   - Planowanie optymalnych zakupów
   - Zamawianie produktów w sklepach (przy tym wybór sklepów)

   Kontakty:
   - Storekeeper Agent - wymiana informacji o aktualnych potrzebach i planowanych dostawach jedzenia

   Cel:
   - Planowanie jak najrzadszych i najlepiej dopasowanych czasowo zakupów na podstawie zamówień Storekeeper Agentów

   Umiejscowienie:
   - Chmura

Architektura rozwiązania:



3. Część B: Projekt systemu wieloagentowego (w metodyce GAIA)

a. Identyfikacja ról

i. **FridgeStateController** - monitorowanie stanu lodówki, czyli ilości produktów w jej wnętrzu.
ii. **ProductDemandPlanner** - planowanie zapotrzebowania na produkty.
iii. **MinimumSupplyGuard** - dbanie o zapewnienia produktów dla lodówki w krótkim okresie czasu.
iv. **SharingHandler** - zarządzanie wymianą produktów pomiędzy klientami.
v. **OrderManager** - zbieranie zamówień i ich optymalizacja.
vi. **Customer** - układanie planów żywieniowych.

b. Model ról

FridgeStateController

Aktywności:

- CheckFridgeInternals - pobieranie danych z lodówki za pomocą czujników.

Protokoły:

- GetFridgeInternalStateRequest- zapytanie o stan zawartości lodówki.
- SendFridgeInternalStateResponse - informowanie o aktualnym stanie zawartości lodówki.

Role schema: *FridgeStateController*
Description: Zadaniem tej roli jest monitorowanie stanu lodówki, czyli ilości produktów oraz terminu ich ważności, a także informowanie o swoim stanie.
Protocols and Activities: GetFridgeInternalStateRequest, CheckFridgeInternals , SendFridgeInternalStateResponse
Permissions: **generates:** fridge_internal_state - informacja o produktach w lodówce
Responsibilities: Liveness: *FridgeStateController* = (GetFridgeInternalStateRequest . CheckFridgeInternals . SendFridgeInternalStateResponse)$^{\omega}$ Safety: ● true

ProductDemandPlanner

Aktywności:

- EstimateFridgeStatePlan - estymowanie zapasów lodówki w czasie, z uwzględnieniem planów żywieniowych właścicieli.

Protokoły:

- GetEstimatedFrigdeStatePlanRequest - odbieranie żądania przekazania planu lodówkowego.
- SendEstimatedFrigdeStatePlanResponse - przekazanie planu lodówkowego.
- SendFridgeInternalStateRequest - pytanie o aktualny stan produktów w lodówce.
- SendFoodPlanRequest - pytanie o plany żywieniowe właścicieli.

Role schema: *ProductDemandPlanner*
Description: Zadaniem tej roli jest wyznaczanie zapotrzebowania (zakupy) na produkty na podstawie planów żywieniowych i obecnego stanu lodówki. Określa zarówno spodziewane braki produktów jak i potencjalne ich zapasy.
Protocols and Activities: GetEstimatedFrigdeStatePlanRequest, SendFridgeInternalStateRequest , SendFoodPlanRequest , EstimateFutureFridgeStatePlan, SendEstimatedFrigdeStatePlanResponse
Permissions: **reads:** fridge_internal_state - informacja o dostępnych produktach food_plans - plany żywieniowe od wszystkich właścicieli lodówki **generates:** future_fridge_state_plan - informacja o zapasach i brakach produktów w lodówce
Responsibilities: Liveness: *ProductDemandPlanner* = (GetEstimatedFrigdeStatePlanRequest . SendFridgeInternalStateRequest . SendFoodPlanRequest * . EstimateFridgeStatePlan . SendEstimatedFrigdeStatePlanResponse)$^{\omega}$ Safety: ● **true**

MinimumSupplyGuard

Aktywności:

- CalculateShortTermProductDemand - wyznaczanie zapotrzebowania na najbliższy czas (np. dzień)

Protokoły:

- SendEstimatedFrigdeStatePlanRequest - zapytanie do *ProductDemandPlanner* o plan zapotrzebowania
- SendProductShareRequest - zapytanie do sąsiadów o pożyczenie produktów.

Role schema: *MinimumSupplyGuard*
Description: Zadaniem tej roli jest zapewnienie produktów potrzebnych w najbliższym czasie, stosując mechanizm pożyczania.
Protocols and Activities: SendEstimatedFrigdeStatePlanRequest, CalculateShortTermProductDemand, SendProductShareRequest
Permissions: **reads:** future_fridge_state_plan - informacja o zapasach i brakach produktów w lodówce **generates:** short_term_product_demand - aktualne zapotrzebowanie na produkty
Responsibilities: Liveness: *MinimumSupplyGuard* = (SendEstimatedFrigdeStatePlanRequest . CalculateShortTermProductDemand . SendProductShareRequest*)ω Safety: **true**

SharingHandler

Aktywności:

- CheckIfCanShare - sprawdzenie czy można pożyczyć produkt bez naruszenia swojego planu.

Protokoły:

- GetProductShareRequest - oczekiwanie na żądanie produktu.
- SendEstimatedFrigdeStatePlanRequest - zapytanie o plan stanu lodówki.
- SendCustomerShareRequest - prośba o podzielenie się produktem.
- SendProductShareResponse - informacja o produktach, które udało się uzyskać.

Role schema: *SharingHandler*
Description: Zadaniem tej roli jest zadbanie o udostępnienie produktów nadmiarowych, o ile klient wyrazi na to zgodę.
Protocols and Activities: GetProductShareRequest , CheckIfCanShare, GetEstimatedFrigdeStatePlan, SendCustomerShareRequest, SendProductShareResponse
Permissions: **reads:** short_term_product_demand - aktualne zapotrzebowanie na produkty customer_share_response - zgoda/odmowa właściciela future_fridge_state_plan - informacja o zapasach i brakach produktów w lodówce **generates:** fridge_state_balance - informacja o ilości produktów do wymiany, products_shared - informacja o produktach, które zostaną wymienione
Responsibilities: Liveness: *SharingHandler* = (GetProductShareRequest . CheckSharePossible . SendProductShareResponse) CheckSharePossible = (SendEstimatedFrigdeStatePlanRequest . CheckIfCanShare . [SendCustomerShareRequest]) Safety: fridge_state_balance <= 0 ⇒ products_shared = EMPTY customer_share_response = false ⇒ products_shared = EMPTY

OrderManager

Aktywności:

- PrepareMassOrder - przygotowanie zamówienie, wraz z podziałem produktów na odbiorców.
- OrderProducts - zamawianie produktów ze sklepu wraz z podziałem dla kuriera.

Protokoły:

- SendEstimatedFrigdeStatePlanRequest - prośba o przesłanie planów lodówkowych.

Role schema: *OrderManager*
Description: Zadaniem tej roli jest planowanie zamówień produktów ze sklepu oraz ich optymalizacja.
Protocols and Activities: GetEstimatedFrigdeStatePlan, PrepareMassOrder, OrderProducts
Permissions: **reads**: future_fridge_state_plan - informacja o zapasach i brakach produktów w lodówce **generates:** order_list - lista z zamówieniem
Responsibilities: Liveness: *OrderManager* = (SendEstimatedFrigdeStatePlanRequest* . PrepareMassOrder . OrderProducts)$^{\omega}$ *(per-day)* Safety: **true**

Customer

Aktywności:

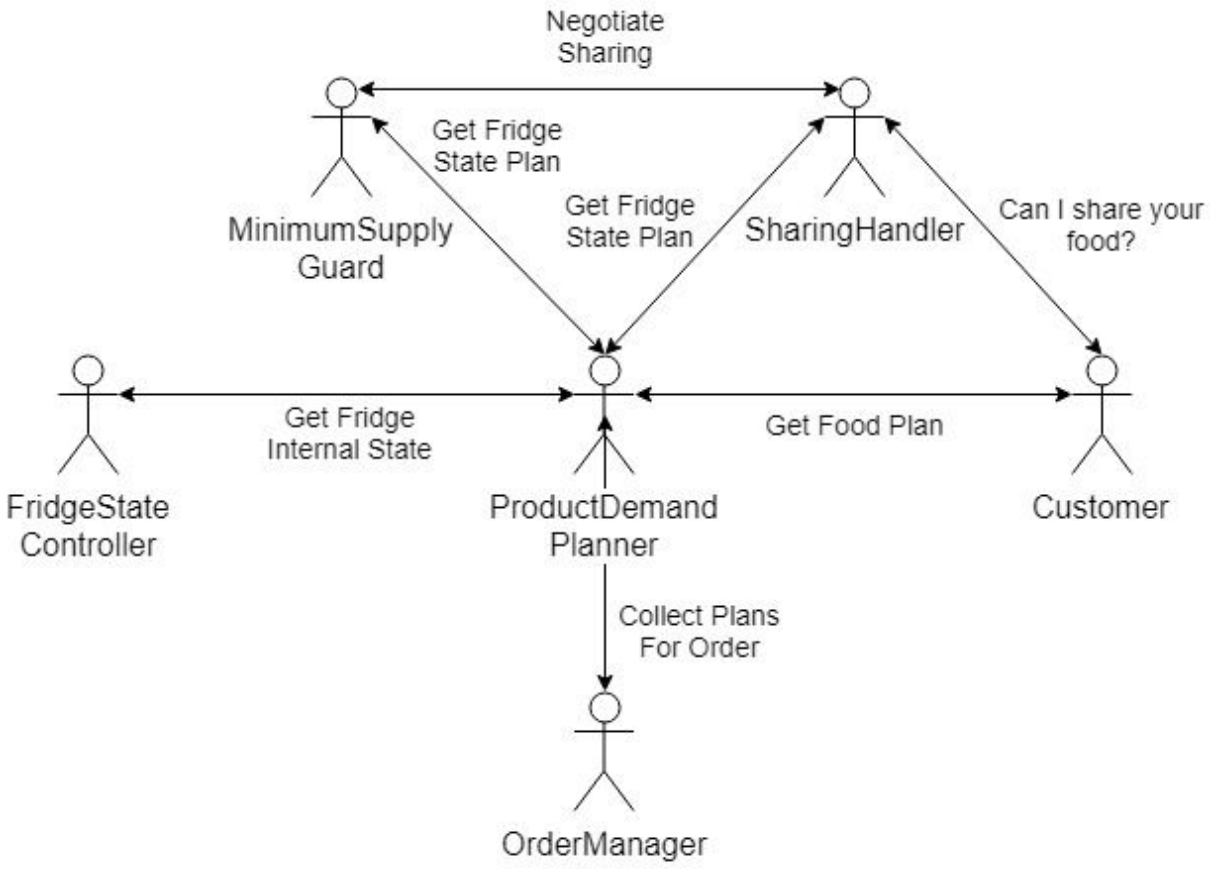
- CheckFoodPlan - pobieranie planu żywieniowego użytkownika
- CheckUserAgreement - pytanie użytkownika o pozwolenie

Protokoły:

- GetShareRequest - czekanie na zapytanie o zgodę na podzielenie się produktami.
- AcceptOrRefuseShare - deklaracja zgody na podzielenie się produktem.
- GetFoodPlanRequest - czekanie na żądanie przesłania planu
- SendFoodPlanResponse - przesłanie planu zapotrzebowania na produkty.

Role schema: *Customer*
Description: Zadaniem tej roli jest odpowiadanie na zapytanie o plan żywieniowy właściciela i odpowiadanie na zapytanie o podzielenie się produktem.
Protocols and Activities: GetFoodPlanRequest, CheckFoodPlan, SendFoodPlanResponse , GetShareRequest, CheckUserAgreement, AcceptOrRefuseShare
Permissions: **generates:** customer_shared_response - zgoda/odmowa użytkownika **reads:** food_plan - plan żywieniowy
Responsibilities: Liveness: *Customer* = (SendFoodPlan || SendAgreementForShare)$^{\omega}$ SendFoodPlan = (GetFoodPlanRequest . CheckFoodPlan . SendFoodPlanResponse) SendAgreementForShare = (GetShareRequest . CheckUserAgreement . AcceptOrRefuseShare) Safety: **true**

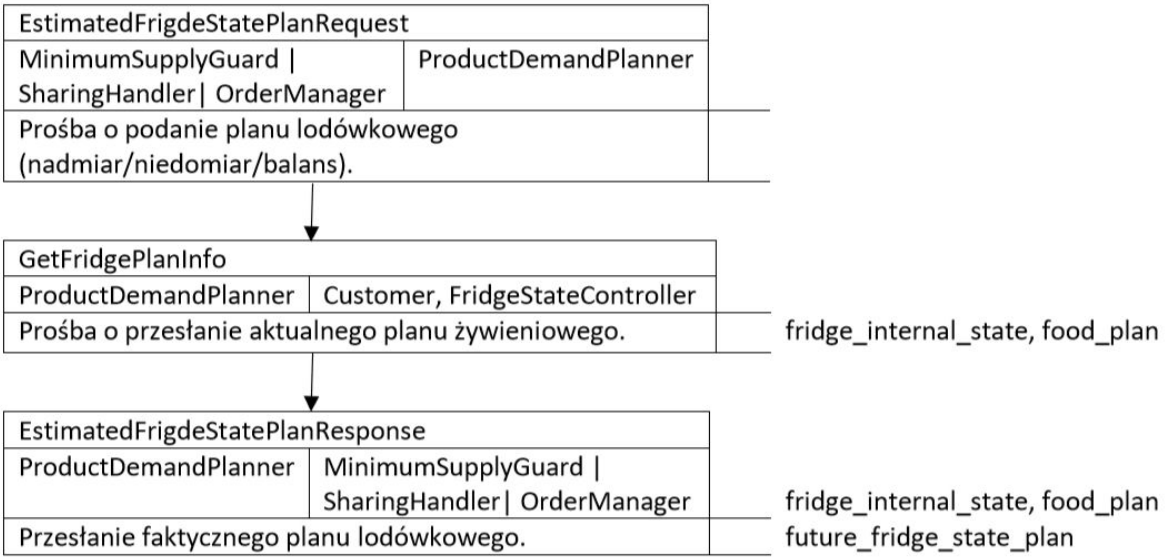
c. Model interakcji



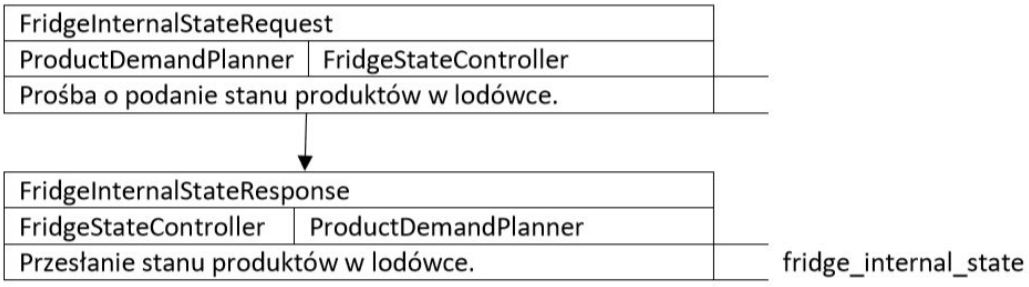
Z powyższego schematu przedstawiającego model interakcji widać, że kluczową rolą w komunikacji i odpowiednim działaniu systemu *SerShare* jest *ProductDemandPlanner*. Wchodzi on w bezpośrednią interakcję z wszystkimi innymi agentami. W związku z tym należy przyłożyć szczególną uwagę do prawidłowej implementacji agenta implementującego rolę *ProductDemandPlanner*.

Definicje protokołów

Protokół: GetFridgeStatePlan.

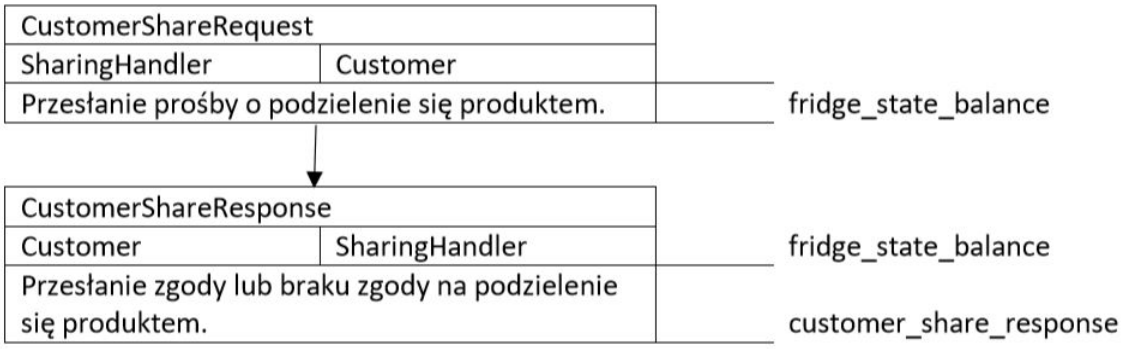


Protokół: GetFridgeInternalState.

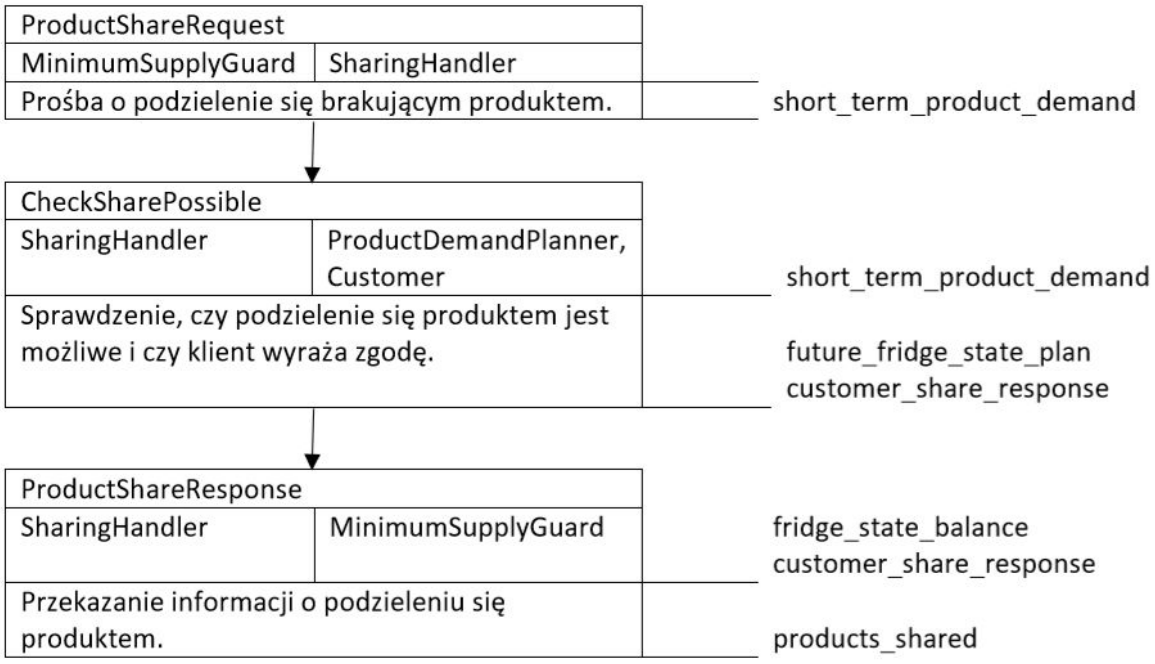


Protokół: GetFoodPlan.

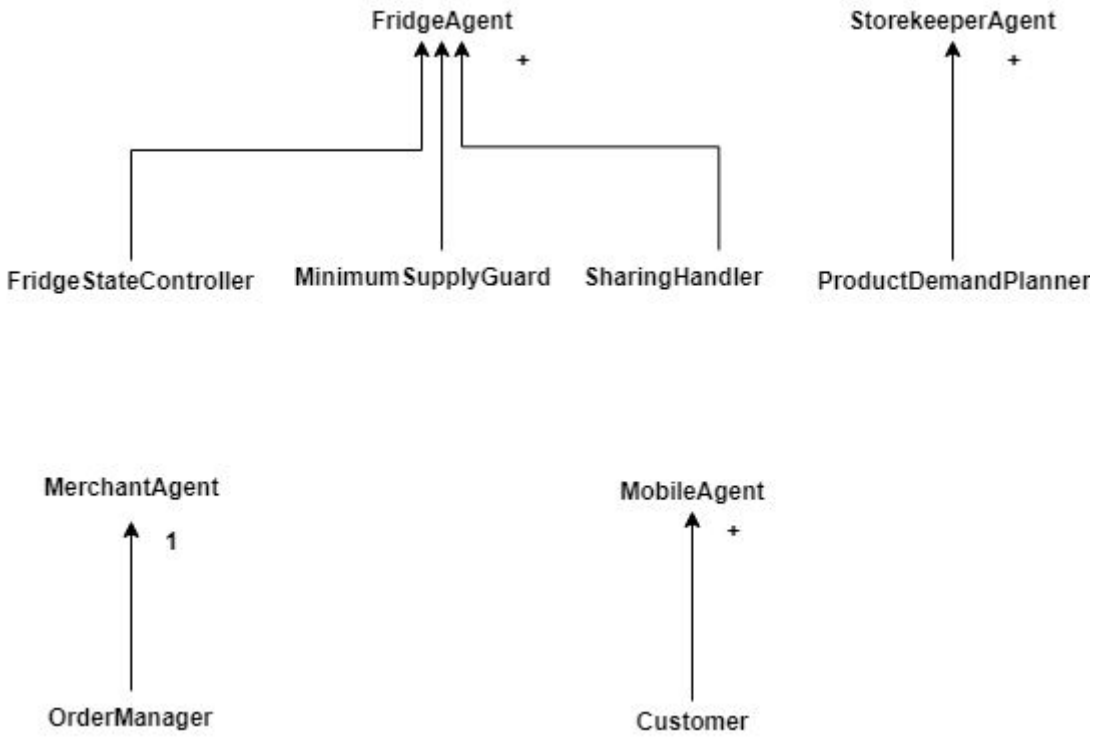
Protokół: GetAgreementForShare.



Protokół: NegotiateShare.



d. Model agentów



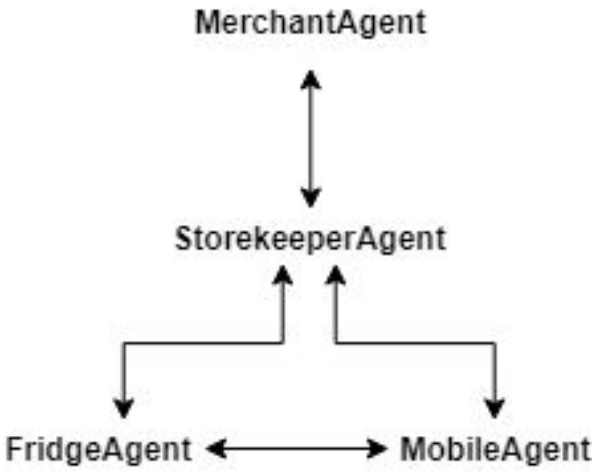
Z powyższych schematów modelu agentów widać, że najbardziej złożonym w kontekście ilości implementowanych ról jest *FridgeAgent*. Wybór taki podyktowany jest faktem, iż role te realizują akcje bezpośrednio powiązane z produktami znajdującymi się w danym momencie w lodówce, którą obsługuje *FridgeAgent*.

Zgodnie z wyszczególnieniem *ProductDemandPlanner* jako szczególnie istotnej roli w komunikacji ze wszystkimi innymi rolami (patrz 2c. Model interakcji), *StorekeeperAgent* implementuje wyłącznie tą rolę, dzięki czemu jego zachowania oraz cele będą nakierowane na realizację wyłącznie odpowiedzialności tej kluczowej roli.

e. Model usług

Usługa	Wejścia	Wyjścia	Warunki wstępne	Warunki końcowe
ustalanie planu zapotrzebowania produktów	fridge_internal_state food_plans	future_fridge_state_plan	food_plans != NULL	future_fridge_state_plan != NULL
sprawdzanie stanu produktów	wyniki z kontrolerów	fridge_internal_state	true	true
ustalanie dziennego zapotrzebowania	future_fridge_state_plan	short_term_product_demand	true	short_term_product_demand != NULL
udostępnianie produktów	short_term_product_demand customer_share_response future_fridge_state_plan	fridge_state_balance products_shared	fridge_state_balance <= 0 ⇒ products_shared = EMPTY customer_share_response = false ⇒ products_shared = EMPTY	fridge_state_balance <= 0 ⇒ products_shared = EMPTY customer_share_response = false ⇒ products_shared = EMPTY
przygotowanie zamówienia	future_fridge_state_plan	order_list	true	order_list != NULL
udostępnienie planu żywieniowego	-	food_plan	true	food_plan != NULL
ustalenie pozwolenia na udostępnianie	-	customer_share_response	true	true

f. Model znajomości



Na powyższym schemacie widać, że *StorekeeperAgent* jest agentem integrującym wszystkie inne agenty. Ponadto, dzięki jego istnieniu, nie ma potrzeby aby *FridgeAgent* oraz *MobileAgent* komunikowały się bezpośrednio z *MerchantAgent*.

Część C: Opis implementacji systemu

a. Framework użyty do implementacji systemu

Do implementacji systemu wybraliśmy framework JADE (Java Agent DEvelopment Framework) - wersja 4.50, język programowania Java.

Głównymi motywacjami tego wyboru były:

- implementacja systemu w JADE odbywa się w języku Java, który wszyscy w zespole znamy
- JADE jest bardzo popularnym, dobrze udokumentowanym frameworkiem agentowym, dzięki czemu w przypadku problemów możemy w łatwy sposób uzyskać wsparcie przeglądając dokumentację lub serwisy społeczności developerów JADE
- JADE jest zgodny ze standardami FIPA, dzięki czemu w łatwy sposób możemy zaimplementować koncepty wieloagentowe w naszym systemie, bez potrzeby tworzenia ich samodzielnie

W celu łatwiejszego procesu konfiguracji środowiska deweloperskiego na różnych komputerach (m.in. instalacja JADE itp.) cały proces ściągania zależności (np. instalacja

JADE), kompilacji oraz tworzenia plików wykonywalnych systemu agentowego został zdefiniowany w Mavenie (plik pom.xml). Dzięki temu możemy implementować system mając wcześniej zainstalowane wyłącznie JDK Javy oraz IDE wspierające Mavena.

b. Sposób implementacji agentów

Utworzyliśmy klasę *SerShareAgent* - standardowego agenta systemu *SerShare*, która dziedziczy po standardowej klasie *Agent* frameworku JADE i zawiera dodatkowo:

- odwołanie na *ContentManager* systemu
- listę id innych znanych agentów
- definicję *SLCodec* dla komunikatów *SerShare (*wybór *FIPA SL* jako języka treści*)*
- metody wspólne dla wszystkich agentów

Wszystkie agenty systemu dziedziczą po klasie *SerShareAgent*, co ułatwia ich implementację w zakresie ich spójności między sobą i pozwala na niepotrzebne powielanie kodu.

Agenty wykorzystują (przez dziedziczenie) zdefiniowane we frameworku JADE zachowania (*behaviours*), które służą do implementacji aktywności (*activities*) oraz protokołów (*protocols*) zdefiniowanych dla ról (*roles*), które implementują (zgodnie z opisem w rozdziale Model ról). Agenty zostały podzielone według wcześniej zaproponowanych typów: fridge, merchant, storekeeper i mobile.

StorekeeperAgent - agent do generowania planów zakupowych dla poszczególnych lodówek. Posiada zapisane informacje o pobranych planach zakupowych oraz stan produktów w lodówce. Ponadto posiada adresy podłączonych mobile agentów oraz fridge agentów. Na podstawie zebranych planów zakupowych oraz stanu lodówki estymuje zapotrzebowanie i planuje następne zakupy. Wykorzystuje do tego funkcje estimate(). Do implementacji zachowań wykorzystuje następujące behaviours:

(abstract) MYCFPBehaviours - specjalne stworzone zachowanie do wysyłania zapytań typu cfp pomiędzy agentami. Składa się z 2 akcji:
wysłanie zapytania do wszystkich zainteresowanych odbiorców i odbieranie ich odpowiedzi. Akcje te występują kolejno po sobie. Ponowne wysłania zapytania może nastąpić poprzez wykorzystanie funkcji reset(). Klasa MYCFPBehavoiurs dostarcza dodatkowo metody do obsługi błędnie skonfigurowanych wiadomości. Aby je obsłużyć można wykorzystać mechanizm podstawowy - wysłanie informacji not understood do nadawcy. Dodatkowo można dodać własną obsługę błędów poprzez nadpisanie metody onErrors(). Następną wykorzystywaną funkcją jest validateMessage(), która służy do walidacji otrzymanych wiadomości. W podstawowym przypadku przyjmuje wiadomości typu inform. Metodę tą można nadpisać własną walidacją

SendFoodPlanRequest - służy do pobierania informacji na temat planów od agentów typu mobile. Nadpisuje klasę MYCFPBehavoiurs. Odpowiedziami są plany produktów poszczególnych mobile aktorów, które są zapisywane przez storekeeper agenta do listy planów. Przyjmuje podstawową walidację i obsługę błędów.

SendFridgeStateRequest - służy do pobierania informacji na temat stanu produktów od agentów typu fridge. Wykorzystuje klase MYCFPBehavoiurs. Odpowiedziami są zawartości poszczególnych aktorów typu fridge, które są zpisywane przez storekeeper agenta do stanu lodówki. Przyjmuje podstawową walidację i obsługę błędów.

GetMobileAid - służy do pobierania wysyłanych przez mobile agent informacji o podłączeniu do systemu. Po odebraniu wiadomości zapisuje id nadawców w liście mobile agentów.

SendEstimatedFridgeStatePlan - służy do wysłania wyliczonego planu do merchant agenta. Jest uruchamiany przez funkcję estimate, wtedy gdy aktor uzna to za słuszne.

EstimateFridgeStatePlan - uruchamia się cyklicznie co określony czas(w modelu symulacyjnym co minutę). Jego zadaniem jest obliczanie nowych wartości planu lodówkowego i gdy minie określony czas(w przypadku symulacyjnym 2 min) i aktor uzna to za stosowne wysłanie informacji o nowym planie lodówkowym do merchant agenta. Główną funkcją agenta wywoływaną przez to zachowanie jest estimate.

MerchantAgent - agent do zamawiania zakupów. Posiada listę planów zakupowych wraz z adresami agentów (typu storekeeper) którzy go wysłali. Jego głównym zadaniem jest określenie jak najbardziej optymalnych zamówień. Wykonuje to dzięki funkcji planOrder która wylicza datę i listę zakupów następnego zamówienia. Ponadto ta metoda po ustaleniu terminu zakupu, dokonuje go za pomocą metody makeOrder(). Oprócz zamawiania zakupów, metoda makeOrder() powiadamia zainteresowanych agentów o zbliżającej się dostawie. Agent ten wykorzystuje następujące zachowania:

GettingFoodPlans - cykliczne nasłuchiwanie na wiadomości dostarczające listy zakupów. Po otrzymaniu wiadomości zapisuje je na liście zakupów wraz z adresem wysyłający zamówienie.

PlanningOrder - zachowanie wykorzystywane do planowania nowych zamówień. Jest wykonywane cyklicznie(w symulacji co 1min). Jego celem jest wywołanie metody planOrder która określa następne zamówienie.

FridgeAgent - agent do pobierania danych o posiadanych produktach z lodówek. Posiada listę produktów, które pobiera z lodówki. Jego głównym zadaniem jest przesyłanie stanu lodówki do agenta StorekeeperAgent. Agent ten wykorzystuje następujące zachowania:

CheckFridgeInternals - cyklicznie (testowo co minute) odpytuje lodówkę o jej stan

GetFridgeInternalStateResposne - odpowiedź na pytanie o liste produktów, wysyła zserializowany obiekt agentowi który zapytał. Przechwytuje wyjątki IO.

MobileAgent - agent zajmuje się pobieraniem planu żywieniowego w postaci mapy Map<LocalDate,List<FoodPosition>> od użytkownika i przesyłaniem go do agenta StorekeeperAgent. Agent ten wykorzystuje następujące zachowania:

GetFoodPlan - to zachowanie jest cykliczne i polega na oczekiwaniu na wiadomość od agenta StorekeeperAgent zawierającą performatywę CFP z prośbą o przesłanie planu żywieniowego. W wyniku otrzymania takiej wiadomości, plan jest pobierany od użytkownika (co zostało zasymulowane poprzez losowanie kilku dat, produktów wraz z ich ilością), a następnie odsyłany do oczekującego agenta.

Ponadto, istnieje dodatkowe zachowanie SendMyAID, które służy do przesyłania wiadomości o id konwersacji “hello-from-<nazwa_typu_agenta>”, dzięki czemu agenty mogą dowiedzieć się o istnieniu innych.

c. Komunikacja

- ### Zastosowane performatywy

1. INFORM
2. CFP
3. NOT_UNDERSTOOD

- ### Zastosowane protokoły komunikacyjne

Nie użyliśmy zdefiniowanych protokołów (np. w standardach FIPA IP), za to zaimplementowaliśmy własne, specyficzne dla naszego przypadku, przy użyciu mechanizmu *behaviours*.

Ponadto, aby uprościć złożoność implementacji, zdecydowaliśmy się nie używać wbudowanych w JADE protokołów, a zamiast tego zdefiniować własne.

- ### Zastosowane języki treści

Komunikacja między agentami odbywa się przy użyciu FIPA Semantic Language (SL).

Główną motywacją wyboru tego języka treści jest to, iż jest on językiem treści zdefiniowanym przez FIPA oraz jest standardowo zaimplementowany we frameworku JADE.

d. Wykorzystane standardy

Z racji wykorzystywania JADE’a, nasz system jest zgodny ze standardami FIPA:

- FIPA ACL (Agent Communication Language)
- FIPA SL (Semantic Language)
- FIPA CAL (Communicative Act Library)

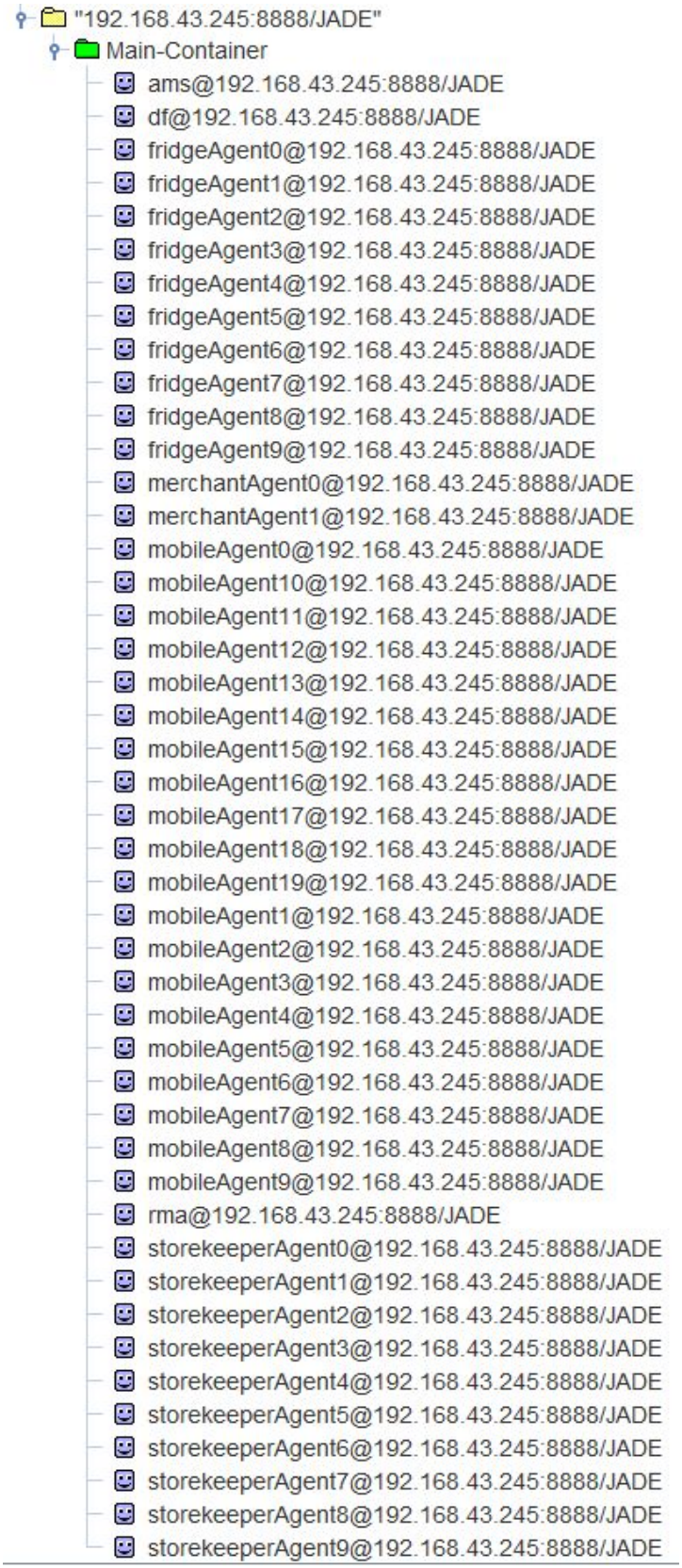
e. Algorytmy

Wszystkie algorytmy używane w systemie *SerShare* zostały zaprojektowane i zaimplementowane specjalnie do ich zastosowań i nie korzystamy ze standardowych algorytmów agentowych. Opis algorytmów został przedstawiony w rozdziale Sposób implementacji agentów.

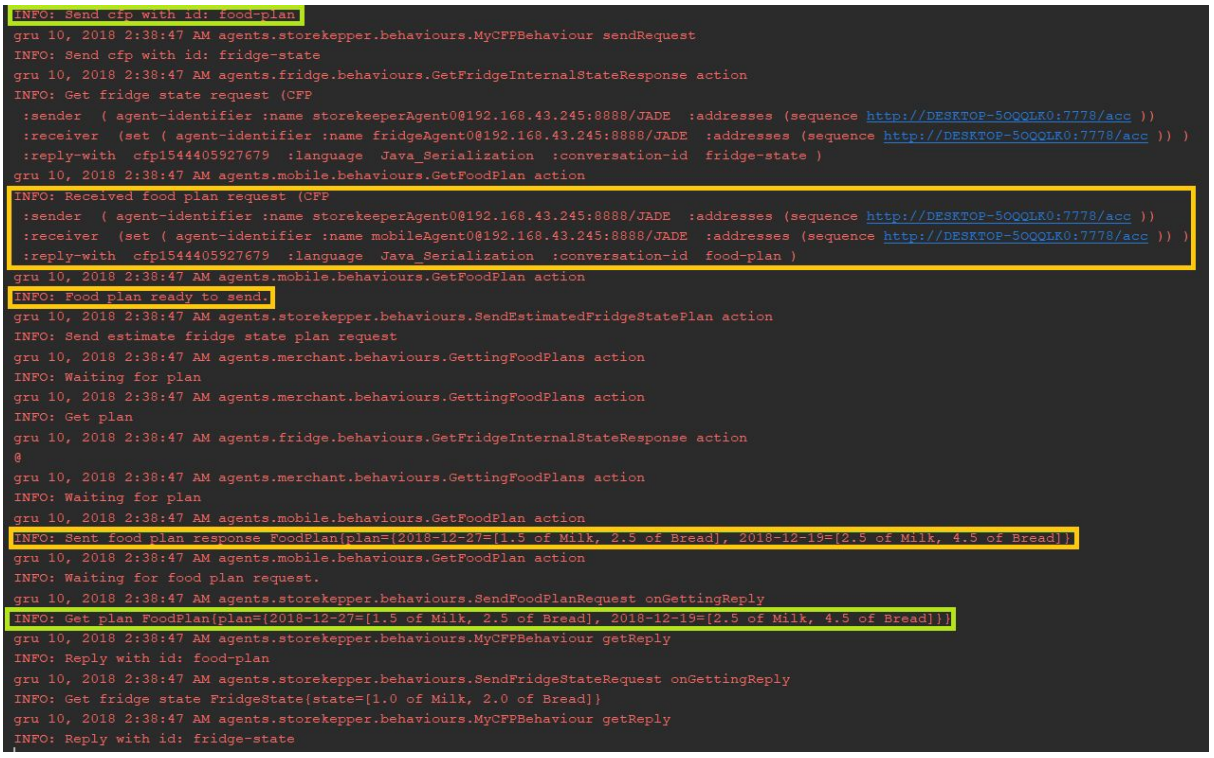
f. Napotkane problemy

- Podczas implementacji równoległych zachowań, w których następowało oczekiwanie na wiadomość, problemem okazało się przechwytywanie wiadomości przez niewłaściwe zachowania. W celu rozwiązania tego problemu użyliśmy szablonów wiadomości, na budowanie których pozwala klasa *jade.lang.acl.MessageTemplate*. Okazuje się, że klasa ta nie filtruje wiadomości w 100%, ponieważ wystąpił jeden przypadek, kiedy pomimo zastosowania filtrowania id konwersacji, została odebrana wiadomość niespełniająca tego warunku. Nie udało się więcej odtworzyć tej sytuacji, dlatego uznaliśmy to jako niezbyt uciążliwą anomalię, spowodowaną prawdopodobnie błędem w bibliotece.
- Początkowo uruchomiliśmy system przy pomocy komend konsolowych (przykładowe utworzenie agenta: “fridge1:agents.fridge.FridgeAgent”). Jednak w przypadku uruchomienia wielu instancji agentów ten sposób nie sprawdza się. Ręczne wpisywanie kilkudziesięciu nazw agentów jest bardzo nieefektywne. Dlatego zdecydowaliśmy się wykorzystać klasę *jade.core.Runtime.* Pozwoliło to na utworzenie kontenera, w którym uruchomienie agentów można było zrealizować w sposób bardziej zautomatyzowany (nazwy tych samych typów agentów wygenerowane w pętli z dodawaniem indeksu iteratora). Tworzone agenty dostają w argumentach nazwy innych agentów, z którymi się komunikują.
- Do tej pory nie został zaimplementowany protokół NegotiateShare. Rozważamy rozwinięcie systemu o ten właśnie protokół w kolejnym etapie projektu.

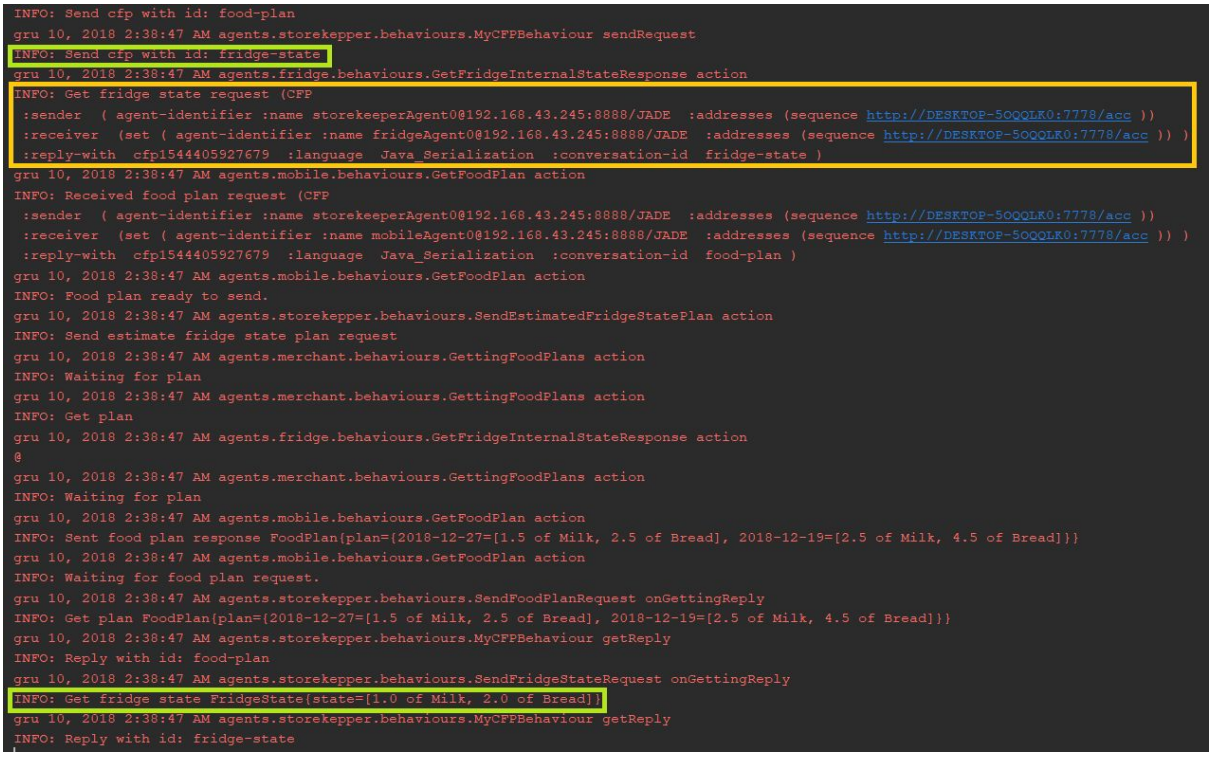
g. Przykładowe zrzuty ekranu z działania



Rysunek 1. Zrzut ekranu z JADE Remote Agent Management GUI.



Rysunek 2. Zrzut ekranu z logami systemu. W ramkach zaznaczone zostały informacje wyświetlane podczas realizacji protokołu GetFoodPlan.



Rysunek 3. Zrzut ekranu z logami systemu. W ramkach zaznaczone zostały informacje wyświetlane podczas realizacji protokołu GetFridgeInternalState.