1.Imię, nazwisko, symbol grupy dziekańskiej, dzień i godzina odbywania zajęć laboratoryjnych
Jakub Riegel, I-41, środa 11.45

2. Sformułowanie tematu zadania
AUTO –system zarządzający flotą autonomicznych samochodów, jeżdżących na terenie Poznania, połączony z symulatorem pojazdów oraz przykładową aplikacją przeglądarkową, umożliwiającą zamawianie przejazdów.

Aplikacja jest dostępna pod adresem: auto.jrie.eu
Dodatkowa dokumentacja znajduje się na repozytorium projektu na platformie GitHub: github.com/jakubriegel/AUTO

3. Charakterystyka dziedziny problemowej
- utworzenie programu przyjmującego od klientów zlecenia na kursy autonomicznych samochodów i przydzielającego je do konkretnych pojazdów oraz monitorującego status tych zleceń; aplikacja będzie działać na serwerze z systemem Debian [technologie: C++, cppcms, cURLpp, Google Maps API]
- opracowanie i zaimplementowanie algorytmu przydzielania tras w taki sposób, żeby zmaksymalizować ilość wykonanych kursów przy danej flocie pojazdów
- opracowanie i zaimplementowanie algorytmu optymalizującego ilość pojazdów w całej strefie działania usługi, celem osiągnięcia minimalnego czasu oczekiwania na kurs przez klienta
- utworzenie symulatora pojazdów, który będzie się komunikował z głównym systemem i realnie odwzorowywał zachowanie floty [technologie: C++, cURLpp, Google Maps API]
- utworzenie symulatora zleceń od klientów, który za pomocą żądań HTTP będzie je wysyłał do głównego systemu [technologie: C++, cURLpp, Google Maps API]
- utworzenie niewielkiej aplikacji przeglądarkowej, pozwalającej na podgląd mapy z zaznaczonymi pojazdami, ich statusu oraz wprowadzania zleceń [technologie: TypeScript, Google Maps JS API]
- implementowanie wszystkich elementów pozwalające na przyszłą rozbudowę projektu, m. in. poprzez stworzenie modułowego silnika obiektowego oraz komunikowanie się z warstwą użytkownika za pomocą JSON API

4. Dyskusja algorytmów rozwiązania, wskazanie algorytmu wybranego do implementacji, uzasadnienie wyboru algorytmu
Pierwszym problemem do rozwiązania było wprawienie samochodów w ruch. Od samego początku założyłem, że będą się one poruszały po prawdziwej mapie, a nie po symulowanym układzie współrzędnych. Potrzebowałem zatem silnika mapowego, który jest w stanie zarówno wyświetlać punkty na mapie, jak i generować trasy przejazdu. Początkowy wybór padł na OpenStreetMap, ze względu na otwartą licencję i duże możliwości personalizacji. Jednak okazało się, że do „uruchomienia" takiego systemu w OSM potrzebne jest dużo wkładu w jego implementację, nawet przy korzystaniu z gotowych bibliotek. Jako że nie było to tematem projektu, postanowiłem zrezygnować z OSM i korzystać z Google Maps API. Dzięki temu dostałem dostęp do wszystkich usług Google Maps z poziomu żądań http, a cały system zyskał na realności, ponieważ Google Maps biorą pod uwagę aktualną sytuację na drogach. Dodatkowo w prosty sposób mogłem zaimplementować zamawianie aut z poziomu front-endu, używając do tego przygotowanego przez Google SeachBoxa, który wyświetla listę propozycji na bazie wpisanego tekstu i automatycznie konwertuje je do gotowych do wysłania dalej współrzędnych geograficznych. Przemieszczanie się samochodu oparłem na strukturze, jaką ma trasa przejazdu, wysłana przez Google Maps. Jest ona podzielona na etapy(±odległość między skrzyżowaniami), dla których posiadam informacje o ich punkcie startowym, końcowym i czasie potrzebnym na jego przejechanie. Symulacja ruchu polega na ustawieniu pojazdu na początku danego etapu, a po upłynięciu czasu potrzebnego na jego przejechanie, przesunięcie pojazdu na początek kolejnego. W trakcie nanoszeniu pojazdów na mapę

w czasie rzeczywistym, daje to wiarygodne wrażenie tego, że poruszają się one po mieście i co jakiś czas przesyłają serwerowi swoja lokalizację. Rozwiązanie to jest proste w zaimplementowaniu i nieskomplikowane obliczeniowo, dlatego przyjąłem je jako ostateczne.

Po stworzeniu logiki pozwalającej na zamawianie kursów, stanąłem przed zadaniem określenia obszaru, na którym usługa będzie dostępna. Najprostszym rozwiązaniem byłoby zaznaczenie punktu w środku Poznania i określenie wokół niego promienia działania systemu. Jednak dla produkcyjnego zastosowania potrzebne by było dokładniejsze rozwiązanie, które pozwalałoby np. na wykreślenie pewnych obszarów lub dodanie stref „wysp" dla podmiejskich miejscowości. Usługi takie jak Uber czy wypożyczalnie aut na minuty zwykle przedstawiają strefę swojej dostępności jako nieregularny wielokąt. Chcąc osiągnąć takie rozwiązanie, musiałem znaleźć sprawny algorytm, sprawdzający, czy punkt znajduje się wewnątrz dowolnego wielokąta. Zdecydowałem się na użycie algorytmu, polegającego na zliczaniu punktów przecięcia odcinka między sprawdzanym punktem i punktem, który na pewno jest poza obszarem, a krawędziami wielokąta. Moją własną implementację tej metody można znaleźć w klasie *Area*. Ponadto przechowywanie obszaru jako tablicy tablic punktów, pozwoliło mi na dodawanie stref-wysp poza Poznaniem.

Ostatnim dużym wyzwaniem algorytmicznym było zapewnienie szybkiego dostępu do samochodów na całym obszarze działania usługi. Ze względu na to że generator zamówień wybiera całkowicie losowe lokalizacje, w projekcie tej skali nie było sensu w stosowaniu algorytmu analizującego w czasie rzeczywistym zagęszczenie zamówień na danych obszarze. Zamiast tego postanowiłem rozmieścić w mieście bazy dla pojazdów. Każda baza posiada określoną ilość slotów, do których mogą podłączyć się samochody. Bazy dbają o to, żeby w każdej z nich zawsze przynajmniej połowa stanowisk była obsadzona. Oprócz tego bazy mają przypisany promień wokół siebie. Jeżeli jakieś auto skończy kurs wewnątrz niego, jest ono wysyłane do danej bazy. W przypadku gdy w bazie brakuje miejsc, pojazd jest wysyłany do portu(głównej bazy). Dzięki temu pojedyncze sektory nie są przeładowane samochodami, a inne sektory zawsze mają dostęp do wolnych pojazdów. Ponadto samochody nie zajmują wolnych miejsc parkingowych na ulicy, gdy w pobliżu znajduje się ich dedykowana baza.

5. O implementacji, czyli charakterystyka wybranych struktur danych użytych w programie

AUTO posiada modułową strukturę. Jej głównym punktem jest obiekt typu ***AUTO***, w którym odbywa się zarządzenie całym systemem. Kolejne moduły to **serwer**, **symulator pojazdów** i **generator zamówień**. Technicznie są one wątkami, które uruchamiają dedykowane metody z *AUTO*, dzięki czemu komunikacja między nimi jest bardzo prosta. Przetwarzają one informacje w swoim zakresie, a następnie wysyłają gotowe dane do innych modułów, przy zachowaniu zasady, że główna logika znajduje się w *AUTO*.

Serwer jest napisany w Crow i odpowiada na żądania http wysyłane do systemu, generując odpowiedzi w formacie JSON. Po otrzymaniu dane są wstępnie przetwarzane, a następnie wysyłane do odpowiedniego modułu. Odpowiedź http jest generowana na podstawie danych zwrotnych z modułu. **Symulator** uruchamiany jest cyklicznie, zgodnie z interwałem zdefiniowanym w *config.json*. Jego działanie polega na wywołaniu metody *update()* na każdym pojeździe oraz bazie. Metoda przyjmuje aktualny czas jako argument i na jego podstawie uaktualniane są dane obiektów. **Generator zamówień** uruchamiany jest na takiej samej zasadzie jak symulator. Jego działanie polega na wylosowaniu punktu startowego i końcowego, a następnie zamówieniu kursu. Warto zaznaczyć że losowanie jest dwu etapowe. W pierwszej kolejności losowana jest wyspa, na której będzie się znajdował punkt. Dla poprawienia realistyczności prawdopodobieństwo wylosowania Poznania(głównej wyspy) jest zwiększone. Następnie losowany jest właściwe położenie. Jako że wyspy mogą być dowolnym wielokątem, dla każdej z nich wyznaczane są najdalej wysunięte punkty w każdym kierunku świata. Są to granice przedziałów z których losowane są współrzędne punktu. Losowanie jest powtarzane do momentu, w którym wylosowany punkt znajdzie się na wybranej wyspie. Jako że Google Maps automatycznie rozpocznie trasę z drogi umieszczonej najbliżej punku, nie potrzebne jest sprawdzanie, czy wylosowane miejsce znajduje się przy ulicy.

Trzonem spójności systemu jest nie tylko sprawna struktura ale także mechanizm stałych zdefiniowanych w abstrakcyjnej klasie *Const*. Standaryzuje on komunikaty zwracane przez poszczególne funkcje oraz porządkuje wartości zależne od pliku *config.json*. Komunikaty zdefiniowane są jako statusy i są stałe niezależnie od konfiguracji aplikacji. Te statusy przede

wszystkim są przypisywane samochodom, dzięki czemu w łatwy można określić stan w jakim znajduje się samochód. Oprócz tego zdefiniowane są między uniwersalny status OK i ERROR, które są zwracane przez procedury, co pozwala na weryfikację ich powodzenia. Wartości wczytywane z pliku konfiguracyjnego to przede wszystkim dane systemu, takie jak: ilość samochodów, parametry baterii pojazdów, ilość i położenie baz, strefa działania usługi, oraz dane potrzebne do jego symulacji, m. in. interwały symulatora i generatora czy częstotliwość zamówień. Takie podejście zapewnia łatwą i szybką konfigurację aplikacji.

Na koniec dodam jeszcze słowo o użyciu języka C++. Projekt wykorzystuje standard C++11, spowodowane to jest wymaganiami dołączonych bibliotek. Starałem się maksymalnie wykorzystać możliwości STL oraz korzystać z jak najinteligentniejszych rozwiązań języka. Przykładem tego może być użycie *std::lock_guard* czy *std::pair*. Dla wygody pisania i usprawnienia czytelności kodu często wykorzystywałem również słowo kluczowe *auto*. Dbałem też o optymalizację pamięciową. Argumenty są przekazywane jako referencje lub wskaźniki. Dotyczy to także statycznych pól klasy *Const*, którym pamięć jest przydzielana w pliku *Const.cpp* w momencie uruchomienia programu. Dodatkowo postanowiłem uprościć importowanie zależności do projektu. Wszystkie zależności STL, zewnętrzne biblioteki, wewnętrzne zależności oraz aliasy zdefiniowane są w pliku *project.hpp*. Dzięki takiemu rozwiązaniu, każdy plik w projekcie zawiera tylko odniesienie do *project.hpp*. Było to możliwe przy zastosowaniu odpowiedniej kolejności załączania plików oraz użyciu predefinicji niektórych klas.

6. Interfejs użytkownika, czyli o sposobie korzystania z programu

System jest uruchamiany za pośrednictwem terminalu systemu Debian. Do działania usługi w sposób ciągły wykorzystuję wirtualne konsole *screen*. Komendy potrzebne do skompilowania i uruchomienia systemu znajdują się w dokumentacji na platformie GitHub.

Użytkownicy mogą korzystać z usługi na pomocą aplikacji przeglądarkowej, dostępnej pod adresem *auto.jrie.eu*. Z jej poziomu można sprawdzić na mapie aktualne położenie pojazdów oraz ich stan, zamówić przejazd i poznać statystyki systemu.

7. Testowanie działania programu

Jako że efekt działania systemu jest widoczny na mapie, wraz z logami wyświetlanymi przez aplikację po stronie serwera były to główne metody testowania jego działania.

Źródłem informacji o zdolności systemu do samo balansowania się jest prowadzona statystyka oczekiwania przez klientów na kurs. Po kilku dniach działania systemu wyniósł on ok. 3-4min, co mnie zaskoczyło, ponieważ planowałem czas między 5 a 7min.

8. Literatura i inne uwagi (opcjonalnie)

-