


Wieloagentowy Symulator Dystrybucji

Dokumentacja końcowa

Wojciech Rokicki
Krzysztof Pierczyk



WSTĘP

Celem projektu było stworzenie aplikacji umożliwiającej przeprowadzanie symulacji transportu materiałów na terenie kraju. Symulacja miała za zadanie określić możliwość (lub jej brak) wykonania zaplanowanych przez użytkownika transportów i zaproponować ich realizację. Jedną z głównych funkcji aplikacji miała być możliwość ingerencji użytkownika w przebieg symulacji poprzez wprowadzanie pewnych zdarzeń utrudniających realizację transportów. Symulacja miała zawierać także element losowe imitujące rzeczywiste zdarzenia, do których może dojść w trakcie transportu.

Interfejs udostępniony użytkownikowi miał być obsługiwany przez przeglądarkę internetową. Aplikacja miała być oparta w głównej mierze o język C++, a jej implementacja miała wykorzystywać paradygmat obiektowy.

Pierwotna koncepcja

Początkowe plany zakładały wykorzystanie języka Python oraz struktury projektowej (*ang. Framework*) **Django**, która stanowiłaby interfejs pomiędzy częścią kliencką, a właściwą symulacją. Główny rdzeń programu, napisany w C++, miał działać równolegle z częścią serwerową, jako odrębny proces, i komunikować się z nią poprzez mechanizmy komunikacji międzyprocesowej takie jak **kolejki komunikatów** czy **pamięć współdzielona**. Pomniejsze moduły miały być uruchamiane okresowo przy zająciu określonych warunków, jako moduły języka Python. Wykorzystana miała zostać do tego biblioteka **boost.Python**.

Pierwotna koncepcja, pomimo swoich licznych zalet, sprowadzała naszą ingerencję w strukturę sieciową do minimum, oferując gotowy serwer HTTP oraz mechanizmy analizy gramatycznej przychodzących zapytań. Chcąc poszerzyć swoją wiedzę z zakresu niskopoziomowego programowania sieciowego zdecydowaliśmy się na stworzenie własnego serwera z użyciem bibliotek **boost.asio** oraz **boost.beast**. Pierwsza z nich zaimplementowana została z wykorzystaniem wzorca projektowego **Proactor**, który pozwala na efektywne implementowanie asynchronicznych operacji sieciowych.

STRUKTURA PROJEKTU

Ze względu na specyfikę zadania oczywistym posunięciem było zastosowanie wzorca architektonicznego **klient-serwer**. Część kliencka, działająca na maszynie użytkownika, funkcjonuje w środowisku przeglądarki internetowej wykorzystując udostępniane przez nią mechanizmy komunikacji sieciowej. Część serwerowa, napisana w pełni w języku C++, może zostać uruchomiona zarówno na maszynie użytkownika jak i na odrębnej stacji znajdującej się w sieci, do której dostęp ma użytkownik.

Część kliencka

Część kliencka została zrealizowana całkowicie z wykorzystaniem języków HTML, CSS, oraz JavaScript. Wykorzystane zostały również mechanizmy udostępniane przez przeglądarki internetowe jak np. automatyczne generowanie zapytań HTTP pobierających z serwera zasoby. Zadania części klienckiej można podzielić na trzy kategorie:

- Prezentacja użytkownikowi aktualnego stanu aplikacji w postaci graficznej
- Przechwytywanie akcji wykonywanych przez użytkownika, walidacja wprowadzanych danych i przesyłanie informacji o zdarzeniach do serwera
- Komunikacja z sieciowym interfejsem programistycznym (*ang. Application Programming Interface - API*) Google Maps w celu umiejscowienia symulacji na rzeczywistej mapie

Interfejs graficzny

Interfejs graficzny, jak wyżej wspomniano, został zaimplementowany przy użyciu standardowych narzędzi stosowanych w aplikacjach sieciowych. Treść stron składających się na aplikację została odseparowana od opisu ich wyglądu (Kaskadowych Arkuszy Stylów – *ang. Cascadeing Style Sheets*), co pozwala na łatwą zmianę motywów graficznych aplikacji poprzez podmianę na serwerze plików z rozszerzeniem .css.

W aplikacji zastosowano wiele skryptów w języku JavaScript, które wprowadzają do interfejsu graficznego elementy interakcyjne, co wpisuje się we współczesne trendy tworzenia aplikacji internetowych.

Interakcje użytkownik-serwer

Drugim z zadań strony klienckiej jest przechwytywanie zdarzeń inicjowanych przez użytkownika (np. zatwierdzania wypełnionych formularzy, wciśnięcia przycisków przekierowujących do innej sekcji aplikacji) oraz wstępna walidacja danych. Walidacja ta sprowadza się w głównej mierze to tego, aby użytkownik nie mógł przejść do pewnych segmentów aplikacji bez pełnego wypełnienia wymaganych formularzy. Ponadto skrypty JavaScript pilnują, aby wartości wpisywane w przeznaczone do tego pola były odpowiedniego typu (np. typ całkowity dla ilości agentów, albo liczba dodatnia dla wartości prawdopodobieństwa).

Zdarzenia rejestrowane po stronie klienckiej są przesyłane za pośrednictwem protokołu HTTP (w wersji 1.1) do serwera. Istnieje istotny podział na zapytania, jakie są do niego wysyłane:

- Zapytania o przesłanie zasobów, takich jak pliki stron, skryptów, czy obrazów **są wysyłane z użyciem metody GET**. Zapytania te są generowane automatycznie przez przeglądarkę internetową w odpowiedzi na wciśnięcie przez użytkownika odpowiedniego linku, przekierowującego do innej sekcji aplikacji
- Zapytania dotyczące akcji wykonywanych przez użytkownika, w szczególności wprowadzanie przez niego danych, **są wysyłane z użyciem metody POST** przy użyciu asynchronicznej komunikacji skryptów JavaScript z serwerem.

Rozróżnienie to będzie bardzo istotne w momencie omawiania mechaniki działania strony serwerowej.

Interfejsy Google Maps

Kluczowym punktem aplikacji jest wykorzystanie rzeczywistych map dostarczanych przez interfejsy programistyczne Google Maps. Mapy te pozwalają nie tylko na poprawienie aspektów estetycznych aplikacji ale też pozwalają **zwiększyć wiarygodność przeprowadzanych symulacji** poprzez wykorzystanie danych z rzeczywistych dróg.

Możliwości jakie wprowadzają w tym zakresie interfejsy programistyczny oferujące dostęp do rzeczywistych map i związanych z nimi danych to m.in.:

- Określenie maksymalnych prędkości dopuszczalnych na poszczególnych drogach
- Możliwość określenia przepustowości tras i prawdopodobieństwa wystąpienia zatorów w określonych porach dnia
- Lepsza estymacja możliwości dojścia do wypadków na wskazanych odcinkach trasy bazująca na danych historycznych
- Możliwość wykluczenia pewnych tras ze względu na ograniczenia nałożone przez przepisy ruchu drogowego na wymiary i ciężar pojazdów

Poprawa jakości symulacji pod względem jej przełożenia na potencjalnie przeprowadzane, rzeczywiste transporty **stanowi kluczowe zagadnienie**, ponieważ przekłada się to na faktyczne korzyści płynące z jej stosowania.

W projekcie znalazły zastosowanie cztery interfejsy programistyczne z portfolio firmy Google:

- Maps Embed API
- Maps JavaScript API
- Directions API
- Roads API

Maps Embed API zostało wykorzystane do zagnieżdżenia apletu mapy w sekcji głównej symulacji. Directions API służy zarówno do planowania tras dla zadanych transportów jak i do wyświetlenia tych tras na mapie. Roads API dostarcza szczegółowych informacji na temat dróg, jak np. precyzyjny kształt jezdni lub ograniczenia obowiązujące na drogach.

Uwaga: Ze względu na ograniczenie dużej części funkcji obecnych w Roads API przez system abonamentowy obecnie szukana jest alternatywa wśród darmowych interfejsów.

Część serwerowa

Część serwerowa stanowi logiczny trzon całego projektu. Jej przemyślana implementacja była dla nas szczególnie ważna ze względu na chęć dalszej rozbudowy aplikacji po zakończeniu semestru. Przy tworzeniu serwera przyświecały nam dwa cele:

- Odseparowanie elementów komunikacji sieciowej od właściwej logiki biznesowej
- Położenie nacisku na modularność programu, która umożliwiłaby łatwą implementację nowych funkcji oraz **możliwość zaimplementowania innych interfejsów użytkownika bez konieczności przeprojektowywania logiki aplikacji**

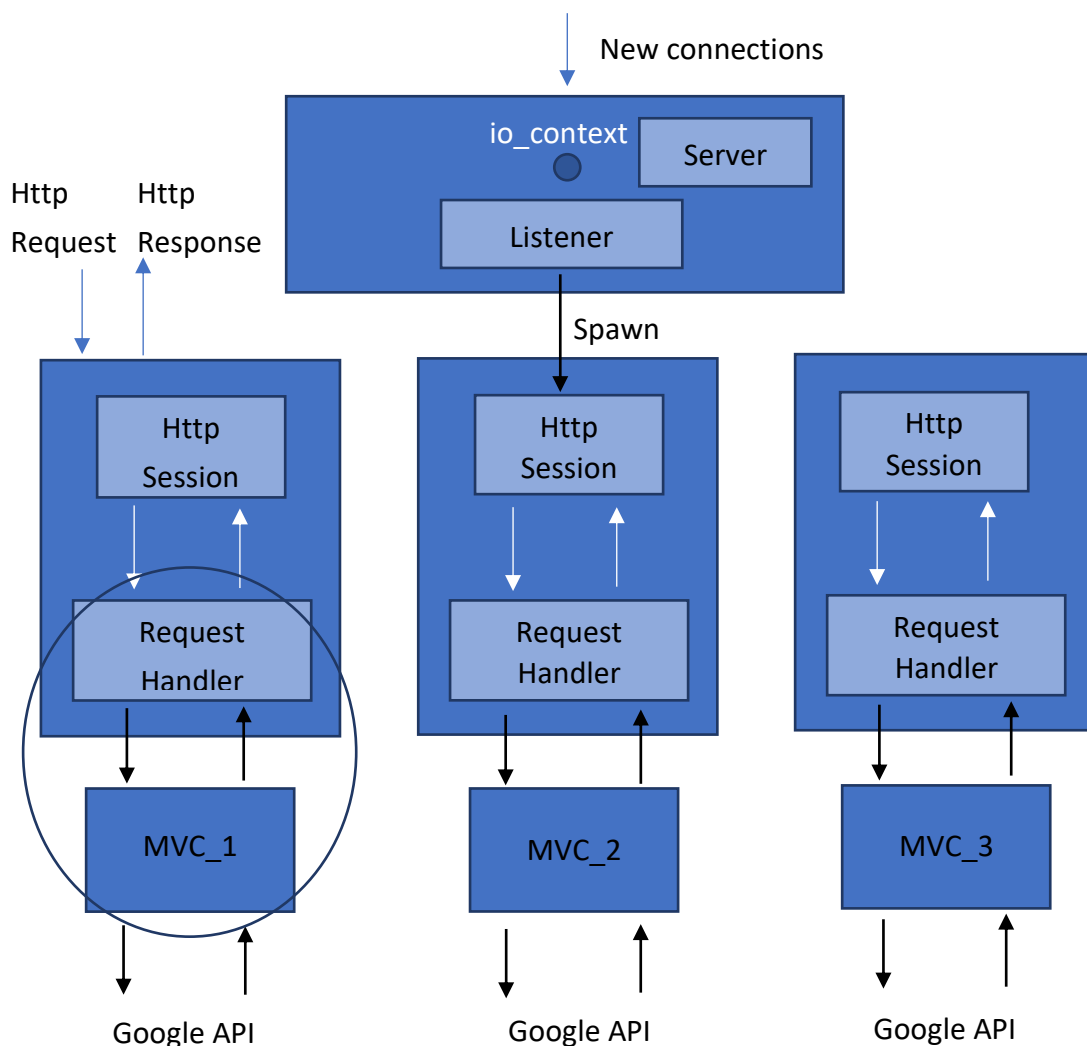
Naszym zdaniem oba te cele udało się w dużym stopniu spełnić sprawiając, że projekt ma duży potencjał do dalszego rozwoju. Ponadto, jego logiczna konstrukcja pozwala na szybkie zrozumienie przepływu sterowania w aplikacji oraz mechanizmów zarządzania obiektami co ułatwia potencjalne wdrożenie się w projekt przez osoby trzecie.

Separacja mechanizmów sieciowych od logiki biznesowej

Wykorzystanie wzorca architektonicznego **klient-serwer** pozwoliło nam na ściśle wyizolowanie zadań należących do sekcji zajmującej się mechanizmami komunikacji sieciowej. Dzięki temu nie dopuściliśmy do nadmiernego rozrośnięcia się kodu odpowiedzialnego za te mechanizmy i zapobiegliśmy przemieszaniu logiki biznesowej z mechanizmami komunikacji.

Poniższy diagram przedstawia ideologiczny schemat działa struktury sieciowej. Obiektem inicjującym pracę całej aplikacji jest instancja klasy Server. Pomimo, że w aplikacji istnieje tylko jeden taki obiekt klasa ta świadomie **nie została zaimplementowana jako Singleton**. Jest to spowodowane faktem, że istnieje możliwość stworzenia kilku instancji tej klasy, po jednej na wątek, które działałyby na odrębnych portach.

Jako, że struktura serwera HTTP (klasy Server) nie jest ściśle związana ze specyfiką implementacji instancji aplikacji możliwe jest **wykorzystanie tej samej klasy serwera do obsługi wielu różnych aplikacji sieciowych opartych na strukturze model-widok-kontroler**. Jedynym wymogiem jaki musi spełniać aplikacja jest zaimplementowanie odpowiednich interfejsów klas Widoku i Kontrolera



Rysunek 1. Struktura modułu sieciowego części serwerowej.

Po utworzeniu obiektu serwera i jego uruchomieniu (wywołaniu blokującej metody `run()`) zostaje stworzony obiekt klasy `Listener`. Jest on odpowiedzialny za nasłuchiwanie na wskazanym porcie TCP i akceptowanie przychodzących połączeń HTTP.

Każde nowe połączenia (sesja) otrzymuje swój obiekt klasy `HttpSession`, który dokonuje:

- Odczytania otrzymanego zapytania http
- Przekazania zapytania do obiektu klasy `RequestHandler`, która je interpretuje i przetwarza
- Odesłania odpowiedzi w postaci wiadomości HTTP

Rozdział obsługi sesji powiązanych z różnymi klientami (w obecnej wersji programu kliencie są rozróżniani na bazie adresu IP) pozwala na równoległe podtrzymanie wielu kanałów komunikacji, a co za tym idzie obsłużenie wielu użytkowników.

Strategia zarządzania czasem życia obiektów dynamicznych

*Budowa aplikacji sieciowych opartych o bibliotekę **boost.asio** charakteryzuje się potrzebą asynchronicznego przetwarzania różnych obiektów. Aby uniknąć niepożądanych wycieków pamięci zastosowano specyficzny model zarządzania czasem życia obiektów.*

Instancje klas opierających swoje działanie na operacjach asynchronicznych (Listener i HttpSession) pracują w cyklach. Każdy cykl składa się z szeregu operacji kończących się Wywołaniem metody asynchronicznej. W takiej sytuacji najlepszym wyjściem jest przerzucenie zadania zwolnienia zasobów obiektu na sam rzeczony obiekt.

*Każda instancja powyższych klas tworzona jest poprzez wywołanie **std::make_shared** i wywołanie na zwróconym wskaźniku metody **run()**. Po jej powrocie zwrócony wskaźnik jest niszczone. Zaalokowany obiekt istnieje tak długo jak długo w swojej wewnętrznej strukturze kultywuje tworzenie kopii wskaźnika **std::shared_ptr** na samego siebie (obie klasy dziedziczą po **std::shared_from_this**). Gdy obiekt przestaje być potrzebny (np. zostaje zamknięta sesja HTTP) jeden z segmentów cyklu powraca bez wcześniejszego utworzenia kopii wskaźnika. W ten sposób obiekt jest niszczone, a jego zasoby zwracane do programu.*

Klasa RequestHandler stanowi bezpośredni interfejs pomiędzy częścią sieciową, a logiką biznesową. Jest ona odpowiedzialna za:

- Wyodrębnienie z zapytania HTTP jego typu (oraz ewentualnej treści – *ang. body*)
- Wywołanie metody Widoku, która zwraca ścieżkę do zasobu, który należy odesłać do klienta (w przypadku zapytań GET)
- Wywołanie metody Kontrolera i na bazie zwróconej przez nią wartości, odpowiedniej metody Widoku, która zwróci dane przeznaczone do odesłania do klienta

Ze względu na fakt, że interfejs sieciowy jest odizolowany od modelu poprzez instancje Widoku i Kontrolera, mogliśmy w pełni skupić się na niezależnej implementacji obu części. Takie podejście przyczyniło się do **znacznego ułatwienia odpluskwiania programu**.

Jak zostało wyżej powiedziane, część sieciowa oparta została na dwóch filarach: bibliotece **boost.asio** implementującej mechanizmy obsługi strumieni wejścia/wyjścia z wykorzystaniem wzorca projektowego Proactora, oraz **boost.beast** dostarczającej zestaw klas do obsługi protokołu HTTP, opartej na bibliotece boost.asio.

Napisanie generycznego serwera w języku C++ pozwoliło na pełne dopasowanie go do naszych potrzeb i zminimalizowanie marnotrawstwa mocy obliczeniowej.

O wielowątkowości serwera słów kilka

Ze względu na brak potrzeby oraz dodatkowe trudności związane z implementacją wielowątkowej obsługi serwera, zdecydowaliśmy się na pozostanie przy strukturze jednowątkowej.

Struktura klasy Serwer (i łańcucha wykorzystywanych przezeń klas) została jednak przemyślana tak, aby w razie potrzeby implementacja wielu wątków sprowadzała się do wprowadzenia minimalnej ilości zmian w istniejącym już kodzie.

Logika biznesowa

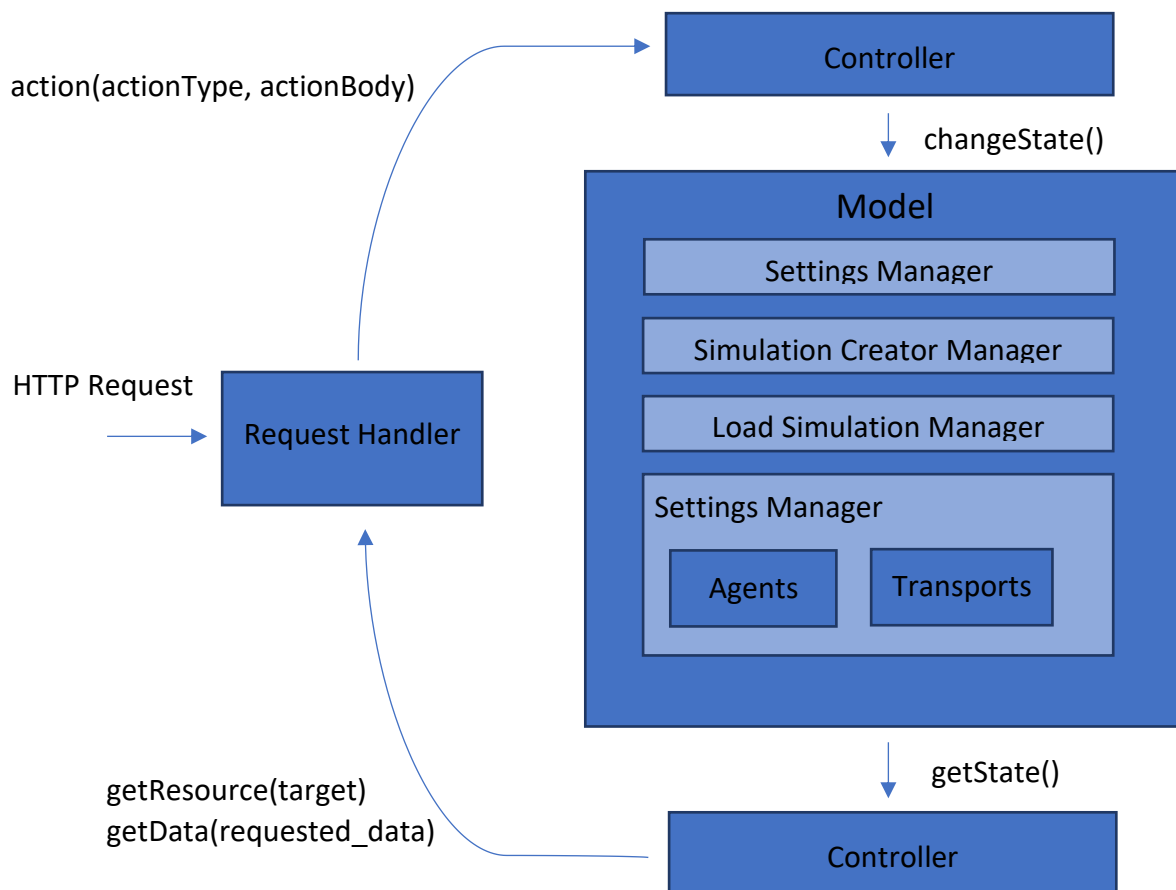
Jak już podkreślaliśmy, w trakcie tworzenia projektu położyliśmy duży nacisk na modularność całej struktury. Idealnym rozwiązaniem okazał się w naszym przypadku wzorzec architektoniczny Model-Widok-Kontroler (*ang. Model-View-Controller, MVC*).

Model stanowi kompletną reprezentację stanu aplikacji. Zawiera on zarówno dane dotyczące ogólnego działania aplikacji jak i szczegółową reprezentację symulacji. Model implementuje wszystkie potrzebne metody reprezentujące rzeczywiste interakcji w jakie użytkownik może wejść z aplikacją

Kontroler przetwarza rodzaj i treść zapytań (tu: zapytań metodą POST) na wywołania odpowiednich metod modelu. Na skutek zmiany w modelu powinno nastąpić odświeżenie lub wczytanie pewnych danych w interfejsie użytkownika, metoda Kontrolera zwraca typ danych, o których powinien dostarczyć Widok.

Widok w odpowiedzi na wywołanie jednej z dwóch funkcji jego interfejsu odczytuje aktualny stan modelu i zwraca:

- Ścieżkę do zasobu, który należy odesłać do klienta, albo
- Dane, w formie tekstowej, które należy odesłać do klienta



Rysunek 2. Struktura komunikacji pomiędzy częścią serwerową, a instancją aplikacji.

WYKORZYSTANE BIBLIOTEKI

Poniżej spisano wszystkie (niestandardowe) biblioteki wykorzystane przy pisaniu części serwerowej wraz z krótkim opisem ich wykorzystania.

Nazwa biblioteki	Opis wykorzystania
Boost.asio	Komunikacja http klient-serwer. Wykorzystanie mechanizmu asynchronicznych czasomierzy do realizacji limitów czasowych dla mechanizmów sieciowych. Komunikacja HTTP z interfejsami Google Directions API i Google Roads API w celu wyznaczania tras dla agentów.
Boost.beast	Praktyczne zastosowanie protokołu http w zadaniach wymienionych w podpunkcie.
Boost.ublas	Reprezentacja położenia agentów i tras na mapie z wykorzystaniem dwuwymiarowych wektorów.
Boost.property_tree	Reprezentacja danych dotyczących symulacji, w szczególności danych z formularzy. Serializacja i deserializacja obiektów opisujących stan aplikacji do postaci tekstu w formacie JSON.
Boost.program_options	Konfigurowanie działania programu za pomocą plików konfiguracyjnych i argumentów wywołania w linii poleceń
Boost.test	Testy jednostkowe
Sqlite3	Integracja baz danych w celu przechowywania zapisów opcji, danych o uzupełnionych formularzach i stanów symulacji.
SSL, Crypto	Komunikacja z wykorzystaniem szyfrowanego protokołu HTTPS (wymóg ze strony Google Maps API).

PODSUMOWANIE PROJEKTU

Pomimo ponad dwukrotnie większej liczbie godzin spędzonych nad projekt niż planowano, nie udało się nam go zakończyć w terminie. Jako powody takiego stanu rzeczy można by wymienić:

- Niedoszacowanie czasochłonności części planowanych elementów oprogramowania
- Bagatelizowanie znaczenia narzędzi wykorzystywanych przy codziennej pracy programisty (systemy automatycznego budowania, systemy ciągłej integracji)
- Skupienie się na wykorzystaniu nowych, nieznanych nam wcześniej technologii (tu: boost.asio, boost.beast) zamiast wykorzystanie prostych, sprawdzonych metod (gotowe serwery http dostarczane wraz z interpreterem języka Python)

Lista niezrealizowanych funkcjonalności programu została przedstawiona poniżej. Ze względu na zdobyte w czasie realizacji projektu doświadczenie pokusiliśmy się o podanie w nawiasach estymowanego czasu potrzebnego na dokończenie poszczególnych mechanizmów. Proszę zwrócić uwagę na różnice pomiędzy poniższymi przewidywaniami, a wykazem czasochłonności podobnych zadań w tabeli podanej na końcu dokumentu. Różnice te wynikają z faktu, iż opanowaliśmy już w dobrym stopniu narzędzia (np. biblioteki dostępne w projekcie boost) pozwalające na realizację poniższych zadań.

1. Wysyłanie z serwera sformatowanej wiadomości na temat stanu symulacji (1h)
2. Cykliczna aktualizacja położenia pojazdów na bazie ich aktualnej prędkości oraz aktualizowanie szczegółowych informacji o drodze na bazie zapytań wysyłanych do Google Maps Roads API (3h)
3. Optymalizacja transportów uwzględniająca wymianę samochodów pomiędzy kierowcami (3h)
4. Opracowanie mechanizmu zapisywania i ładowania symulacji z pliku (3h)
5. Realizacja przez model zapytań o modyfikację stanu kierowców na podstawie otrzymywanych zapytań http (2h)
6. Dokończenie sporządzania pełnych testów jednostkowych dla klas MVC (5h)

Łącznie otrzymujemy pulę 17h (po 8.5h na członka zespołu).

Zyski z projektu

Realizacja projektu przyczyniła się do głębszego zrozumienia zaawansowanych mechanizmów języka C++ oraz praktycznego zapoznania się ze standardowymi bibliotekami projektu boost. Ponadto nabyliśmy dużego doświadczenia w kwestii planowania struktury większych projektów i poznaliśmy newralgiczne punkty w procesie ich tworzenia.

Projekt ten był także dla nas pierwszą okazją do grupowej pracy nad złożonym oprogramowaniem. Dzięki temu nauczyliśmy się sprawnej organizacji celów i zadań prowadzących do ich realizacji, a także efektywnego podziału obowiązków.

Ze względu na szczególne cechy projektu mieliśmy też okazję do zapoznania się ze szczegółami działania popularnych protokołów komunikacyjnych warstwy aplikacji modelu TCP/IP, w szczególności **HTTP** oraz **WebSocket**.

Dalszy rozwój

Ze względu na niewielką ilość pracy wymaganej do finalizacji programu zdecydowaliśmy się na jego dokończenie pomimo upływającego terminu oddania. Mając do dyspozycji dobrze ustrukturyzowany projekt chcielibyśmy go domknąć implementując jednocześnie kilka funkcjonalności, które przeszły nam przez myśl w momencie projektowania, a które nie były dla nas krytyczne z punktu widzenia wymagań postawionych przez zadanie.

Ponadto, struktura serwerowa odseparowana od logiki aplikacji zachęca do próby realizacji innych aplikacji z wykorzystaniem tej samej struktury serwerowej (jedynym wymogiem jest dopasowanie – trywialnego – interfejsu Widoku i Kontroler). Z drugiej strony ciekawy wydaje się także pomysł przeniesienia symulatora na grunt inny niż sieciowy. Modularność struktury MVC sprawia, że sprowadza się to jedynie do wymiany klas Widoku i Kontrolera.

Dalsze prace, choć nie z taką intensywnością jak w ciągu ostatniego miesiąca, będą więc trwały, a sam projekt możliwe, iż stanie się podstawą do realizacji innych aplikacji bazujących na tej samej strukturze.

Poniższa tabela prezentuje nakład pracy w godzinach dla poszczególnych zadań:

Opis zadania	Planowany nakład pracy	Rzeczywisty nakład pracy
Interfejs graficzny		
Stworzenie plików statycznych menu aplikacji (HTML, CSS)	4	6
Stworzenie plików statycznych interfejsu symulacji (HTML, CSS)	3	6
Obsługa Google Maps API w aplikacji klienckiej (HTML , JavaScript)	5	12
Interaktywne elementy interfejsu graficznego w aplikacji klienckiej (JavaScript)	3	25
Mechanika części klienckiej		
Komunikacja klient-serwer poprzez protokół HTTP (JavaScript AJAX)	4	25
Format zapytań i parametrów przesyłanych do serwera	2	5
Globalne opcje aplikacji	5	1
Mechanizm wprowadzania parametrów symulacji	4	8
Wizualizacja symulacji na podstawie danych odesłanych przez serwer	9	15
Mechanizm modyfikowania symulacji w czasie rzeczywistym	8	10
Możliwość zapisywania i wczytywania przeprowadzonych symulacji	2	-
Możliwość zapisywania i wczytywania zestawów parametrów symulacji z pliku	2	-
Możliwość zapisywania i wczytywania symulacji/parametrów z serwera	2	4
Dodatkowe motywy graficzne dla aplikacji*	4	-
Oprawa dźwiękowa*	4	-

* - funkcjonalność dodatkowa, której zaimplementowanie zależne będzie od pracochłonności projektu

Opis zadania	Planowany nakład pracy	Rzeczywisty nakład pracy
Część serwerowa		
Opracowanie koncepcji ogólnej struktury aplikacji umożliwiającej odseparowanie mechanizmów sieciowych i logi aplikacji oraz podatnej na rozwój poprzez prostą integrację nowych modułów	-	12
Stworzenie własnego serwera HTTP obsługującego podstawowe zapytania o zasoby (GET i HEAD)	-	50
Format przesyłania symulacji i parametrów do klienta	2	5
Stworzenie interfejsu pomiędzy serwerem HTTP, a instancjami symulatora	-	10
Stworzenie mechanizmów pozwalających na równoległe przetwarzanie rozróżnialnych instancji aplikacji dla wielu klientów	-	15
Stworzenie abstrakcyjnego modelu aplikacji udostępniającego uniwersalne metody pozwalające na włączenie go do pracy z różnymi modułami peryferyjnymi (zapewniającymi interakcję z użytkownikiem)	-	8
Stworzenie abstrakcyjnego modelu symulacji umożliwiającego uniwersalną komunikację z różnymi modułami peryferyjnymi		18
Zamodelowanie klasy agenta i sposobu ich interakcji z modelem symulacji	3	6
Stworzenie Kontrolera przetwarzającego akcje wykonane przez użytkownika na odpowiednie wywołania metod modelu	-	15
Stworzenie Widoku pozwalającego na wydobycie z modelu odpowiednich danych na żądanie klienta	-	10
Algorytmy wyznaczania tras przez agentów	9	-
Komunikacja agent-serwer	3	-
Zbiorowa optymalizacja tras wyznaczonych przez agentów	6	-

Opis zadania	Planowany nakład pracy	Rzeczywisty nakład pracy
Część serwerowa		
Obsługa zapytań o modyfikację parametrów symulacji w trakcie jej trwania	8	-
Obsługa warunkowego trasowania przez agentów	3	-
Implementacja baz danych przechowywanych na serwerze	4	7
Prace nad zrównolegleniem wykonywanych obliczeń poprzez wprowadzenie wielowątkowości	4	5
Testy jednostkowe	-	8
Inne		
Napisanie skryptów budujących (SCons)	2	8
Przetestowanie aplikacji na systemie Linux	1	3
Napisanie dokumentacji dla użytkownika	2	1
Napisanie dokumentacji końcowej	3	3
Łącznie	108	301