Politechnika Wrocławska

Wydział Elektroniki,
Fotoniki i Mikrosystemów

Inżynieria Oprogramowania dla Elektromobilności

Ćwiczenie nr 6. Realizacja komunikacji międzyprocesowej między serwerem a bazą danych

Zagadnienia do opracowania:

- system kontroli wersji na przykładzie Git
- system budowania CMake
- paradygmat programowania obiektowego i jego cechy (hermetyzacja, abstrakcja, dziedziczenie, polimorfizm)
- Test–Driven Development (TDD)
- Google Test framework
- techniki komunikacji międzyprocesowej w systemie Linux
- specyfikacja protokołu MQTT
- programowanie współbieżne w C++
- wyrażenia lambda

Literatura

[1] Williams A. *C++ Concurrency in Action. Practical Multithreading*. Shelter Island, NY, USA: Manning Publications Co., 2006.

[2] *C++ reference*. `https://en.cppreference.com/w/`. [Online; dostęp 21.11.2022].

[3] *CMake. Documentation*. `https://cmake.org/documentation/`. [Online; dostęp 22.10.2022].

[4] W. Gajda. *Git. Rozproszony system kontroli wersji*. Gliwice, Polska: Helion, 2013.

[5] *gMock Cheat Sheet*. `http://google.github.io/googletest/gmock_cheat_sheet.html`. [Online; dostęp 06.12.2022].

[6] *gMock Cookbook*. `http://google.github.io/googletest/gmock_cook_book.html`. [Online; dostęp 06.12.2022].

[7] *gMock for Dummies*. `http://google.github.io/googletest/gmock_for_dummies.html`. [Online; dostęp 06.12.2022].

[8] *Google Test*. `https://github.com/google/googletest`. [Online; dostęp 21.11.2022].

[9] Kornelia Indykiewicz. *Wykład: Inżynieria Oprogramowania dla Elektromobilności*. 2022.

[10] Meyers S. *Effective Modern C++*. Sebastopol, CA, USA: O'Reilly, 2015.

[11] Prata S. *Szkoła Programowania. Język C++*. Gliwice, Polska: Helion, 2006.

[12] S. Chacon; B. Straub. *Pro Git*. `https://git-scm.com/book/pl/v2/`. [Online; dostęp 22.10.2022]. 2014.

Spis treści

1 Cel ćwiczenia **2**

2 Wprowadzenie **2**

2.1 Pobieranie aplikacji serwera wiadomości 2

2.2 Testy aplikacji serwera wiadomości 3

2.3 Komunikacja międzyprocesowa (IPC) 5

2.4 Zintegrowane środowisko programistyczne Qt Creator 9

3 Program ćwiczenia **10**

1. Cel ćwiczenia

Celem ćwiczenia jest opanowanie podstawowych umiejętności z zakresu projektowania i implementacji komunikacji międzyprocesowej w systemie Linux z wykorzystaniem języka C++. Tematyka zajęć obejmuje zagadnienie komunikacji z wykorzystaniem protokołu MQTT.

2. Wprowadzenie

2.1. Pobieranie aplikacji serwera wiadomości

Jeżeli wcześniej pobrano aplikację serwera wiadomości, należy przejść od razu do punktu 2.2. Aby pobrać aplikację serwera wiadomości należy:

1. utworzyć swój katalog roboczy wewnątrz folderu **Documents** (jeżeli katalog jeszcze nie istnieje). Nazwa katalogu powinna być unikalna (np. stanowić numer indeksu);

2. z poziomu utworzonego katalogu uruchomić konsolę systemową (terminal) i pobrać repozytorium projektu wywołując polecenie:

   git clone *adres_repozytorium*

3. przenieść się do katalogu zawierającego pobrane repozytorium:

   cd messageserver

4. za pomocą konsoli systemowej wywołać polecenie:

   git submodule update --init --recursive

   w celu zaimportowania zależności projektowych (podmodułów);

5. jeżeli nie instalowano wcześniej biblioteki paho MQTT (patrz: Ćw. 3), to przenieść się do katalogu:

messageserver\third_parties\libipc\third_parties\paho.mqtt.c

i za pomocą konsoli systemowej wywołać polecenie:
Uwaga: To jest pojedyncze polecenie:

cmake -Bbuild -H. -DPAHO_ENABLE_TESTING=OFF -DPAHO_BUILD_STATIC=ON -DPAHO_WITH_SSL=ON -DPAHO_HIGH_PERFORMANCE=ON

a następnie polecenie:

sudo cmake --build build/ --target install

2.2. Testy aplikacji serwera wiadomości

W celu zbudowania pliku wykonywalnego testów jednostkowych aplikacji serwera wiadomości należy kolejno:

1. przejść do głównego repozytorium projektu (**messageserver**)
2. ukryć lokalne zmiany w plikach repozytorium (jeśli wprowadzano zmiany w kodzie) wywołując za pomocą konsoli systemowej polecenie:

   git stash

3. zsynchronizować stan kopii lokalnej repozytorium ze stanem zdalnego repozytorium wywołując polecenie:

   git pull origin master

4. zsynchronizować stan zależności projektowych (podmodułów) wywołując polecenie:

   git submodule update --init --recursive

5. przywrócić lokalne zmiany w plikach repozytorium (jeśli zmiany zostały ukryte; patrz punkt 2.) wywołując polecenie:

   git stash pop

6. przejść do katalogu **test** i utworzyć w nim katalog roboczy (np. **build**):

   cd test
   mkdir build

7. przenieść się do utworzonego katalogu roboczego i za pomocą konsoli systemowej wywołać polecenie:

   cmake ..

   w celu wykonania skryptu **CMakeLists.txt**;

8. za pomocą konsoli systemowej wywołać polecenie:

   make

   aby zbudować plik wykonywalny testów jednostkowych aplikacji *message_server*.

Zbudowany plik wykonywalny zostanie umieszczony w nowo utworzonym katalogu **deploy** wewnątrz katalogu roboczego (tu: ***build***). Uruchomienie testów (z poziomu katalogu **deploy**) za pomocą konsoli systemowej realizowane jest poleceniem:

./message_server_utest

Kod testów jednostkowych znajduje się w katalogu **utest** położonym wewnątrz katalogu **test**. Poza plikiem **main.cpp** znajdują się w nim dwa pliki z implementacją testów jednostkowych klasy **messageserver::TcpMessage** – **TcpMessageSerializationTest.cpp** oraz **TcpMessageDeserialization-Test.cpp**, plik z testami klasy **messageserver::IpcTranslationStrategy**

– **IpcTranslationStrategyTest.cpp**, a także plik z testami klasy **messageserver::EchoStrategy** – **EchoStrategyTest.cpp**. Deklaracje testowanych klas znajdują się w plikach nagłówkowych, odpowiednio: **TcpMessage.h**, **EchoStrategy.h**, **IpcTranslationStrategy.h** w katalogu **include**; natomiast definicje metod klas umieszczono w plikach: **TcpMessage.cpp**, **EchoStrategy.cpp**, **IpcTranslationStrategy.cpp** w katalogu **src**.

2.3. Komunikacja międzyprocesowa (IPC)

Ogólny model komunikacji międzyprocesowej został opisany w rozdziale 2.5 Komunikacja międzyprocesowa (IPC), Ćw. 3. Na rys. 2.1 przedstawiono przegląd podsystemów wchodzących w skład wdrażanego systemu informatycznego. Aplikacja graficznego interfejsu użytkownika (*Messenger GUI*) stanowi podsystem, który może zostać uruchomiony w ramach tego samego systemu operacyjnego co aplikacja serwera wiadomości (*Message Server*), bądź w ramach oddzielnego systemu operacyjnego, w zależności od potrzeb użytkownika. Wymiana danych między wspomnianymi aplikacjami jest realizowana przy użyciu biblioteki *networking*, udostępniającej funkcje umożliwiające komunikację za pośrednictwem protokołu TCP. Biblioteka wykorzystuje funkcje interfejsu POSIX systemu operacyjnego Linux. Jeżeli obie aplikacje uruchomione są w ramach tego samego systemu operacyjnego, to wymiana danych zachodzi w ramach hosta lokalnego, dlatego adres serwera to adres interfejsu *loopback* – 127.0.0.1. Domyślnie serwer wiadomości nasłuchuje komunikacji TCP na porcie 8080.

Aplikacja procesora danych (*Data Processor*) jest zawsze uruchomiona w ramach tego samego systemu operacyjnego co aplikacja serwera wiadomości. Komunikacja między procesami w ramach systemu operacyjnego Linux realizuje wzorzec architektoniczny brokera komunikatów (*and. message broker*). Warstwą pośredniczącą w wymianie danych między aplikacjami jest broker MQTT (tu: Eclipse Moquitto). Sam protokół MQTT (*ang. Message Queue Telemetry Transport*) z kolei realizuje wzorzec publikacja–subskrypcja. Klienci subskrybują się do brokera na konkretne tematy wiadomości, które

chcą odbierać (*topics*). Wiele aplikacji może jednocześnie być zasubskrybowanych, jak również publikować wiadomości na ten sam temat. Broker odbiera opublikowane wiadomości, filtruje je na podstawie tematu, a następnie przesyła do zasubskrybowanych na dany temat klientów. Serwer wiadomości subskrybuje się na temat *+/message_server* (dowolny nadawca do serwera wiadomości), natomiast procesor danych subskrybuje się na temat *+/data_processor* (dowolny nadawca do procesora danych). Broker MQTT do komunikacji międzyprocesowej wykorzystuje interfejs *loopback* i port TCP o numerze 1883 (127.0.0.1:1883). Komunikacja z wykorzystaniem protokołu MQTT jest realizowana w ramach biblioteki *ipc*, stanowiącej wraper na bibliotekę *Paho MQTT C++*.

Wymiana danych między procesorem danych a bazą danych jest realizowana za pomocą systemu RDBMS – SQLite. Biblioteką języka C++ wykorzystaną w implementacji aplikacji procesora danych jest biblioteka *SQLiteCpp*.

Serwer wiadomości jest aplikacją wielowątkową. Każdy z wątków jest odpowiedzialny za realizację innego zadania:

- wątek główny programu – przeprowadza inicjalizację komponentów aplikacji serwera wiadomości (funkcja **main()**); tworzy pozostałe wątki programu; po przeprowadzonej inicjalizacji jest blokowany wewnątrz funkcji **ipc::ApplicationFacade::execute()** do momentu odebrania przez aplikację jednego z sygnałów systemowych **SIGINT** lub **SIGTERM**; po odblokowaniu wątku głównego następuje zamknięcie aplikacji;

- wątek kolejki TCP – tworzony i zarządzany przez klasę **messageserver::ConnectionScheduler**; odpowiedzialny za obsługę wszystkich aktywnych połączeń z instancjami aplikacji graficznego interfejsu użytkownika za pośrednictwem protokołu TCP (funkcja **messageserver::ConnectionScheduler::handleIncommingTransmission()**); deleguje obsługę połączenia do obiektu realizującego strategię **messageserver::ITransmissionStrategy**;

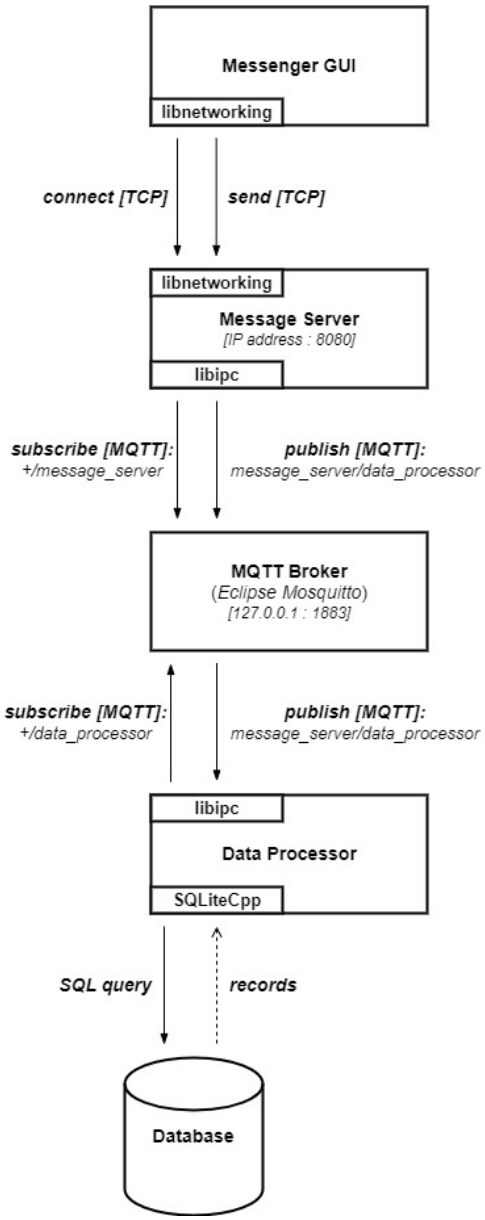
- wątek nasłuchowy TCP – implementowany w ramach klasy **messageserver::ListenerThread**; odpowiedzialny za odbieranie przychodzących połączeń TCP i przekazywanie ich do wątku kolejki TCP;
- wątek kolejki IPC – implementowany w ramach klasy **common::ActiveQueue**, zarządzany przez klasę **ipc::Messenger**; odpowiedzialny za parsowanie przychodzących wiadomości IPC, tłumaczenie ich na wiadomości TCP i przesyłanie do aplikacji graficznego interfejsu użytkownika.

Obsługa transmisji danych w ramach protokołu TCP jest realizowana w wątku kolejki TCP. Funkcja **handleIncommingTransmission()** klasy **messageserver::ConnectionScheduler** cyklicznie przekazuje wszystkie obiekty aktywnych połączeń TCP (realizacja interfejsu **networking::IConnection**) do instancji klasy realizującej interfejs **messageserver::ITransmissionStrategy**. Dzięki wykorzystaniu polimorficznej referencji do klasy bazowej **messageserver::ConnectionScheduler::transmissionStrategy** możliwa jest zmiana trybu pracy aplikacji (tryb echa – diagnostyczny; tryb tłumaczenia wiadomości – standardowy). Interfejs **messageserver::ITransmissionStrategy** udostępnia metodę **handleTcpTransmission()**, która jest implementowana przez klasy **messageserver::IpcTranslationStrategy** oraz **messageserver::EchoStrategy**.

Strategia **messageserver::EchoStrategy** odbiera dane wykorzystując metodę **receive()** interfejsu **networking::IConnection**, a następnie, jeżeli transmisja jest nadal aktywna (status **TRANSMISSION_PENDING**), odsyła z powrotem dane do klienta, bez żadnej modyfikacji.

Strategia **messageserver::IpcTranslationStrategy** ma za zadanie odebrać dane przesłane przez klienta, a następnie, jeżeli transmisja jest nadal aktywna, przeprowadzić translację wiadomości TCP na wiadomość IPC (por.: rozdział 2.5 Komunikacja międzyprocesowa (IPC), Ćw. 3 i rozdział 2.3 Struktura wiadomości TCP, Ćw. 5) i wysłać wiadomość do aplikacji procesora danych za pośrednictwem protokołu MQTT. Tłumaczenie wiadomości TCP (klasa **messageserver::TcpMessage**) do IPC (**ipc::IpcMessage**) jest realizowane przez klasę **messageserver::MessageTranslator** (metoda

translate()). Wiadomość IPC można wysłać wykonując wywołanie zwrotne na funktorze **messageserver::IpcTranslationStrategy::ipcSender** (wywołanie funkcji **sendIpcMessage()** klasy **ipc::ApplicationFacade**).



Rys. 2.1. Podsystemy wdrażanego systemu informatycznego i połączenia między nimi

2.4. Zintegrowane środowisko programistyczne Qt Creator

Proponowanym zintegrowanym środowiskiem programistycznym (*IDE*) do pracy na zajęciach laboratoryjnych jest **Qt Creator**. Program można uruchomić za pomocą konsoli systemowej wywołując polecenie:

qtcreator

Otwarte zostanie okno główne programu. Aby zaimportować projekt *CMake* należy z paska menu wybrać **File** ⟶ **Open File or Project...** W oknie wyboru projektu należy przenieść się do katalogu repozytorium (**messageserver**) i zaznaczyć plik **CMakeLists.txt**, a następnie nacisnąć przycisk **Open** znajdujący się w prawym górnym rogu okna. W oknie konfiguracji projektu należy określić ustawienia narzędzi budowania aplikacji w środowisku **Qt Creator**. Zaznaczając opcję **Select all kits** wybieramy wszystkie dostępne konfiguracje. Kliknięcie przycisku **Configure Project** spowoduje wczytanie źródeł projektu i otwarcie okna edycji plików.

3. Program ćwiczenia

Uwaga: Warunkiem dopuszczenia do realizacji laboratorium jest przedstawienie rozwiązania Zadania 1. na początku zajęć.

Uwaga: Zadania należy realizować w kolejności numerycznej.

Zadanie 1. (Zadanie domowe) Brak zadania domowego.

Zadanie 2. (na ocenę 3.0) Uzupełnij brakującą implementację funkcji **handleTcpTransmission()** klasy **messageserver::IpcTranslationStrategy** aplikacji serwera wiadomości (*message_server*), zgodnie z jej przeznaczeniem (rozdział 2.3). Aby przetestować działanie funkcji:

Opcja 1. Przeprowadź testy manualne. W tym celu uruchom aplikacje: procesora danych (*data_processor*, patrz: Ćw. 3), serwera wiadomości (*message_server*) oraz graficznego interfejsu klienta (*messenger_gui*, patrz: Ćw. 4). Nawiąż połączenie z serwerem, a następnie przetestuj komunikację między aplikacją graficznego interfejsu klienta a bazą danych, przesyłając do serwera poprawne oraz niepoprawne wiadomości zawierające kwerendy SQL w ustandaryzowanym formacie wiadomości TCP (patrz: Ćw. 5). **Uwaga: Skorzystanie z Opcji 1. wymaga wcześniejszego rozwiązania następujących zadań: Zadania 2. z Ćw. 3; Zadania 2. z Ćw. 4; Zadania 2. i Zadania 3. z Ćw. 5.**

Opcja 2. Skorzystaj z zaimplementowanych testów jednostkowych w ramach pliku **IpcTranslationStrategyTest.cpp**. **Uwaga: Poprawna implementacja funkcji poskutkuje bezbłędnym zakończeniem wszystkich testów z pliku IpcTranslationStrategyTest.cpp.**

W protokole z przebiegu ćwiczenia zapisz kolejne kroki algorytmu wdrożonego w ramach implementacji funkcji **handleTcpTransmission()** klasy **messageserver::IpcTranslationStrategy**.

Zadanie 3. (na ocenę 4.0) Przeanalizuj kod aplikacji serwera wiadomości (*message_server*), a następnie wykorzystując diagram sekwencji UML przedstaw przepływ sterowania między komponentami programu w ramach obsługi przychodzącej wiadomości TCP. Za zdarzenie początkowe przyjmij wywołanie funkcji **handleIncommingTransmission()** klasy **messageserver::IpcTranslationStrategy**, a za zdarzenie końcowe wywołanie funkcji **sendIpcMessage()** klasy **ipc::ApplicationFacade**.

Zadanie 4. (na ocenę 5.0) Bazując na implementacji funkcji **handleTcpTransmission()** klasy **messageserver::EchoStrategy** aplikacji serwera wiadomości (*message_server*), napisz testy jednostkowe (białoskrzynkowe) pokrywające następujące przypadki:

- metoda ***networking::IConnection::receive()*** zasygnalizowała zakończenie transmisji, zwracając enumerator **TRANSMISSION_FINISHED** – przypadek testowy ***ShouldHandleFinishedTransmission***;
- metoda ***networking::IConnection::receive()*** zasygnalizowała przekroczenie dozwolonego czasu oczekiwania na przychodzącą wiadomość, zwracając enumerator **TRANSMISSION_TIMEOUT** – przypadek testowy ***ShouldHandleTimedOutTransmission***;
- metoda ***networking::IConnection::receive()*** odebrała wiadomość i zasygnalizowała transmisję w toku, zwracając enumerator **TRANSMISSION_FINISHED** – przypadek testowy ***ShouldHandlePendingTransmission***.

Wszystkie testy umieść w pliku **EchoStrategyTest.cpp**. W protokole z przebiegu ćwiczenia zapisz kolejne kroki algorytmu weryfikacji działania funkcji

handleTcpTransmission(), realizowane w ramach poszczególnych przypadków testowych.

W sprawozdaniu zawrzeć:

- protokół z przebiegu ćwiczenia
- najważniejsze cechy protokołu MQTT;
- kolejne kroki algorytmu wdrożonego w ramach implementacji funkcji **handleTcpTransmission()** klasy **messageserver::IpcTranslationStrategy**; algorytm przedstaw w dowolnej formie (schemat blokowy, opis słowny, pseudokod, kod języka C++, ...)
- diagram sekwencji UML przedstawiający przepływ sterowania między komponentami aplikacji serwera wiadomości w ramach obsługi przychodzącej wiadomości TCP [*jeśli zostało zrealizowane* ***Zadanie 3.***];
- kolejne kroki algorytmu weryfikacji działania funkcji **handleTcpTransmission()**, realizowane w ramach poszczególnych przypadków testowych; algorytm przedstaw w dowolnej formie (schemat blokowy, opis słowny, pseudokod, kod języka C++, ...) [*jeśli zostało zrealizowane* ***Zadanie 4.***].