Politechnika Wrocławska

Wydział Elektroniki,
Fotoniki i Mikrosystemów

Inżynieria Oprogramowania dla Elektromobilności

Ćwiczenie nr 4. Implementacja wielowątkowego komunikatora w architekturze klient-serwer

Zagadnienia do opracowania:

- system kontroli wersji na przykładzie Git
- system budowania CMake
- diagramy UML (klas, sekwencji)
- paradygmat programowania obiektowego i jego cechy (hermetyzacja, abstrakcja, dziedziczenie, polimorfizm)
- inteligentne wskaźniki (*ang. smart pointers*) języka C++; zastosowania klasy *std::unique_ptr*
- protokół komunikacyjny TCP
- wzorzec architektoniczny klient–serwer
- kreacyjne wzorce projektowe
- programowanie współbieżne w C++

Literatura

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

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

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

[4] N. M. Josuttis. *C++ Biblioteka Standardowa*. Gliwice, Polska: Helion, 2012.

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

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

[7] J. Schmuller. *UML dla każdego*. Gliwice, Polska: Helion, 2003.

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

Spis treści

1. Cel ćwiczenia

Celem ćwiczenia jest opanowanie podstawowych umiejętności z zakresu projektowania i implementacji aplikacji w architekturze klient–serwer. Tematyka zajęć obejmuje zagadnienie wielowątkowej komunikacji bezprzewodowej z wykorzystaniem protokołu TCP.

2. Wprowadzenie

2.1. Aplikacja serwera wiadomości

Aplikacja serwera wiadomości (*message_server*) to podsystem stanowiący pośredni punkt komunikacji między aplikacją procesora danych (*data_processor*) a interfejsem użytkownika. Jej zadaniem jest tłumaczenie wiadomości przesyłanych bezprzewodowo za pośrednictwem protokołu TCP przez aplikacje uruchomione na innych urządzeniach na wiadomości w komunikacji międzyprocesowej (IPC). Aplikacja może obsługiwać wielu klientów TCP jednocześnie dzięki wielowątkowemu przetwarzaniu odbieranych wiadomości. Każde kolejne oczekujące połączenie z klientem TCP obsługiwane jest na osobnym wątku systemowym. Serwer może również pracować w trybie echa, odsyłając odebrane wiadomości z powrotem do nadawcy; z pominięciem komunikacji z procesorem danych (funkcja diagnostyczna).

2.2. Budowanie aplikacji serwera wiadomości

W celu zbudowania pliku wykonywalnego aplikacji serwera wiadomości należy kolejno:

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. w głównym katalogu repozytorium (**messageserver**) utworzyć katalog roboczy (np. **build**):

   mkdir build

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

   cmake ..

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

7. za pomocą konsoli systemowej wywołać polecenie:

   make

   aby zbudować plik wykonywalny aplikacji *message_server*.

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

./message_server

Pomyślne uruchomienie aplikacji zostanie podsumowane na konsoli w postaci informacji o połączeniu z brokerem MQTT:

Application connected to MQTT broker

Aby zakończyć działanie aplikacji można posłużyć się sygnałem systemowym przerwania (SIGINT) – skrót klawiszowy z poziomu konsoli systemowej **CTRL + C**.

2.3. Konfiguracja aplikacji serwera wiadomości

Katalog **data** znajdujący się w głównym katalogu repozytorium projektu (**messageserver**) zawiera plik konfiguracyjny w formacie JSON **app_config.json**, składający się z czterech atrybutów:

- **port_number** – numer portu protokołu TCP (domyślnie 8080);
- **awaiting_connections_queue_depth** – maksymalna głębokość kolejki oczekujących na nawiązanie połączenia klientów TCP;
- **enable_echo** – dopuszczalne wartości **true**/**false**; włącz albo wyłącz pracę w trybie echa (z pominięciem komunikacji z procesorem danych); **Uwaga: W celu realizacji Ćw. nr 4 należy ustawić wartość na true**;
- **enable_ssl** – dopuszczalne wartości **true**/**false**; włącz albo wyłącz komunikację z wykorzystaniem protokołu bezpieczeństwa SSL; **Uwaga: W celu realizacji Ćw. nr 4 należy ustawić wartość na false**.

Plik konfiguracyjny jest parsowany podczas uruchamiania aplikacji. W celu wczytania zmian w pliku należy ponownie uruchomić aplikację serwera wiadomości.

2.4. Aplikacja graficznego interfejsu użytkownika

Aplikacja graficznego interfejsu użytkownika (*messenger_gui*) to podsystem stanowiący interfejs komunikacyjny użytkownika z implementowanym systemem informatycznym. Jest to również jedyna aplikacja stanowiąca warstwę prezentacji systemu. Jej zadaniem jest pobieranie danych wejściowych od użytkownika i przesyłanie ich za pośrednictwem bezprzewodowego protokołu TCP do serwera wiadomości (*message_server*). Interfejs graficzny umożliwia również wyświetlanie odebranych odpowiedzi z serwera.

2.5. Budowanie aplikacji graficznego interfejsu użytkownika

W celu zbudowania pliku wykonywalnego aplikacji graficznego interfejsu użytkownika należy kolejno:

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 messengergui

4. za pomocą konsoli systemowej wywołać polecenie:

   git submodule update --init --recursive

   w celu zaimportowania zależności projektowych (podmodułów);

5. w głównym katalogu repozytorium (**messengergui**) utworzyć katalog roboczy (np. **build**):

   mkdir build

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

   cmake ..

   w celu wykonania skryptu **CMakeLists.txt**. Jeżeli wystąpi błąd:

   Could not find a package configuration file provided by "QT",

   oznacza to, że pakiety biblioteki **Qt** nie są zainstalowane w systemie operacyjnym. Instalację można przeprowadzić za pomocą konsoli systemowej, wywołując polecenie:

   sudo apt-get install qtbase5-dev qtchooser qt5-qmake qtbase5-dev-tools

7. za pomocą konsoli systemowej wywołać polecenie:

   make

   aby zbudować plik wykonywalny aplikacji *messenger_gui*.

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

./messenger_gui

Pomyślne uruchomienie aplikacji okienkowej poskutkuje uruchomieniem graficznego interfejsu użytkownika (GUI).

2.6. Komunikacja bezprzewodowa

Komunikacja bezprzewodowa między aplikacją graficznego interfejsu użytkownika a serwerem wiadomości odbywa się za pośrednictwem protokołu TCP (*ang. Transmission Control Protocol*), realizując wzorzec klient–serwer. Wiele aplikacji klienckich może jednocześnie być połączonych z tym samym serwerem. Serwer wiadomości w sposób pasywny nasłuchuje komunikacji na wybranym porcie gniazda TCP, podczas gdy aplikacja graficznego interfejsu użytkownika stanowi stronę aktywną, inicjując komunikację bezprzewodową. Aby rozpocząć nasłuch na gnieździe TCP w systemie operacyjnym Linux (serwer TCP) należy wywołać kolejno:

1. utworzyć gniazdo wywołując funkcję ***int socket(int domain, int type, int protocol)*** interfejsu POSIX – argument **domain** specyfikuje przestrzeń adresową (IPv4 / IPv6); argument **type** określa rodzaj komunikacji (połączeniowa / bezpołączeniowa); argument **protocol** definiuje rodzaj protokołu komunikacyjnego; funkcja zwraca deskryptor pliku (*ang. file descriptor*)[1] powiązany z utworzonym gniazdem;

2. powiązać otwarte gniazdo z adresem IP i numerem portu węzła sieciowego wywołując funkcję ***int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)*** interfejsu POSIX – argument **sockfd** to deskryptor pliku gniazda; argument **addr** stanowi strukturę parametrów zawierającą adres IP oraz numer portu; argument **addrlen** stanowi rozmiar struktury **addr** w bajtach; funkcja zwraca status wykonania operacji (0 w przypadku powodzenia);

3. wprowadzić gniazdo w tryb pasywnego nasłuchu wywołując funkcję ***int listen(int sockfd, int backlog)*** interfejsu POSIX – argument **sockfd** to deskryptor pliku gniazda; argument **backlog** stanowi maksymalną głębokość kolejki oczekujących na nawiązanie połączenia klien-

[1] unikalny numer umożliwiający identyfikację pliku otwartego w systemie operacyjnym

tów TCP; funkcja zwraca status wykonania operacji (0 w przypadku powodzenia);

4. ustanowić połączenie z pierwszym z oczekujących klientów TCP z kolejki wywołując funkcję ***int accept(int sockfd, struct sockaddr *restrict addr, socklen_t *restrict addrlen)*** interfejsu POSIX – argument **sockfd** to deskryptor pliku gniazda; argumenty **addr** i **addrlen** to bufory na dane definiujące węzeł klienta (adres IP, numer portu); funkcja blokuje wątek systemowy do momentu akceptacji żądania połączenia z wykorzystaniem protokołu TCP; zwracany jest deskryptor pliku powiązany z gniazdem klienta.

Aby wysłać żądanie nawiązania połączenia TCP w systemie operacyjnym Linux (klient TCP) należy wywołać kolejno:

1. utworzyć gniazdo wywołując funkcję ***int socket(int domain, int type, int protocol)*** interfejsu POSIX;

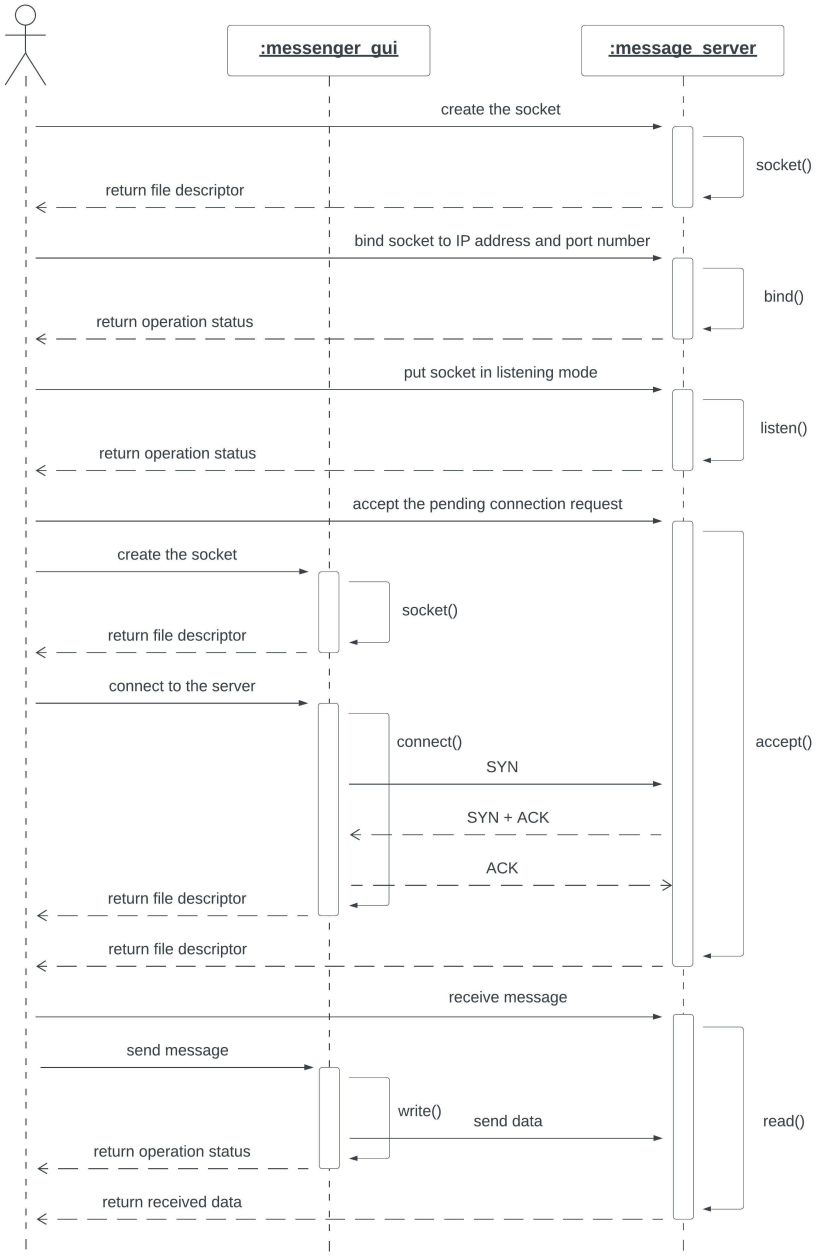
2. połączyć utworzone gniazdo z gniazdem nasłuchowym TCP wywołując funkcję ***int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen)*** interfejsu POSIX – argument **sockfd** to deskryptor pliku gniazda klienta zwrócony przez funkcję ***socket()***; argument **addr** stanowi strukturę parametrów zawierającą adres IP oraz numer portu gniazda nasłuchowego; argument **addrlen** stanowi rozmiar struktury **addr** w bajtach; funkcja zwraca status wykonania operacji (0 w przypadku powodzenia).

Wysyłanie i odbiór danych w systemie operacyjnym Linux mogą zostać zrealizowane z wykorzystaniem deskryptorów plików gniazd serwera i klienta za pomocą funkcji interfejsu POSIX:

- ***ssize_t write(int fd, const void *buf, size_t count)***
- ***ssize_t read(int fd, void *buf, size_t count)***

Wywołanie funkcji ***int close(int fd)*** na deskryptorze pliku spowoduje zamknięcie powiązanego z nim gniazda.

Schemat komunikacji bezprzewodowej przedstawiono na rys. 2.1.



Rys. 2.1. Komunikacja bezprzewodowa za pośrednictwem protokołu TCP

Pakiety wymieniane między aplikacjami w ramach komunikacji bezprzewodowej można obserwować korzystając z darmowego oprogramowania **Wireshark**. Jest to tzw. *sniffer*, czyli aplikacja przechwytująca i analizująca komunikację sieciową. Instalację programu można przeprowadzić z poziomu konsoli systemowej wywołując polecenie:

sudo apt install wireshark-qt

Uruchomienie aplikacji jest realizowane poleceniem:

sudo wireshark

Aby rozpocząć przechwytywanie komunikacji sieciowej należy nacisnąć przycisk **Start capturing packets**. W celu ograniczenia zakresu wyświetlania komunikacji sieciowej można posłużyć się filtrami (**Apply display filter**), np.:

- **tcp** – wyświetla tylko pakiety protokołu TCP;
- **ip.addr == 192.168.0.21** – wyświetla tylko pakiety wymienione z węzłem o adresie IP 192.168.0.21.

Komunikacja między serwerem wiadomości a aplikacją procesora danych odbywa się za pośrednictwem protokołu MQTT (patrz: Ćw. 3). Aplikacja serwera wiadomości subskrybuje się do brokera MQTT na tematy wiadomości **+/message_server** (dowolny nadawca do aplikacji o nazwie message_server).

2.7. Struktura aplikacji

Katalog **src**, znajdujący się w głównym katalogu repozytorium projektów zawiera wszystkie pliki źródłowe aplikacji. W pliku **main.cpp** znajduje się główna funkcja programu realizująca sekwencję inicjalizacji obiektów składających się na aplikację. Aplikacje korzystają z zewnętrznych bibliotek programistycznych dołączanych jako podmoduły systemu kontroli wersji Git (*git submodules*):

- aplikacja serwera wiadomości – biblioteki **networking**, **common**, **ipc**;
- aplikacja graficznego interfeksu użytkownika – biblioteki **networking**, **common**, **Qt**.

Klasa **networking::TcpSocket** biblioteki **networking** implementuje funkcje potrzebne do nawiązania połączenia bezprzewodowego przy użyciu protokołu TCP. Stanowi realizację klasy interfejsu **networking::ISocket**:

1. konstruktor ***TcpSocket()*** – tworzy gniazdo połączenia przez wywołanie funkcji ***socket()*** interfejsu POSIX; w przypadku niepowodzenia rzucany jest wyjątek **std::runtime_error**;
2. funkcja ***void bind(uint16_t portNumber)*** – wiąże gniazdo połączenia z adresem IP komputera i przekazanym numerem portu (argument **portNumber**) przez wywołanie funkcji ***bind()*** interfejsu POSIX;
3. funkcja ***void listen(uint64_t awaitingConnectionsQueueDepth, const ConnectionCallback & callback)*** – ustawia gniazdo połączenia w tryb nasłuchowy przez wywołanie funkcji ***listen()*** interfejsu POSIX i rozpoczyna cykliczną akceptację nadchodzących żądań połączenia przez wywoływanie funkcji ***accept()*** interfejsu POSIX na osobnym wątku systemowym; argument **awaitingConnectionsQueueDepth** stanowi maksymalną głębokość kolejki oczekujących na nawiązanie połączenia klientów TCP, natomiast argument **callback** to obiekt funkcyjny, na którym zostanie wykonane wywołanie zwrotne z obiektem nowo utworzonego połączenia TCP (realizacja interfejsu **networking::IConnection**);
4. funkcja ***std::unique_ptr<IConnection> connect(const std::string & ipAddress, uint16_t portNumber)*** – łączy gniazdo połączenia z gniazdem nasłuchowym otwartym na węźle o zadanym adresie IP (argument **ipAddress**) i numerze portu (argument **portNumber**) przez wywołanie funkcji ***connect()*** interfejsu POSIX. Funkcja zwraca

obiekt nowo utworzonego połączenia TCP realizujący interfejs **networking::IConnection**

5. destruktor ***∼TcpSocket()*** – zamyka gniazdo połączenia przez wywołanie funkcji ***close()*** interfejsu POSIX.

Klasa interfejsu **networking::IConnection** biblioteki **networking** dostarcza funkcje potrzebne do nadawania i odbierania wiadomości przy użyciu deskryptorów plików gniazd połączenia TCP:

1. funkcja ***void send(const std::string & message)*** – wysyła wiadomość (argument **message**) do stowarzyszonego gniazda połączenia TCP;

2. funkcja ***Status receive(std::string & outputBuffer)*** – odbiera pakiet danych o maksymalnym rozmiarze 1024 B i zapisuje go do przekazanego bufora (argument **outputBuffer**); funkcja zwraca enumerator TRANSMISSION_FINISHED, jeśli połączenie TCP zostało zakończone, albo enumerator TRANSMISSION_PENDING w przeciwnym razie.

Klasa interfejsu **networking::ISocketFactory** biblioteki **networking** udostępnia funkcję umożliwiającą utworzenie obiektu realizującego interfejs **networking::ISocket**:

std::unique_ptr<ISocket> createSocket().

Klasa **networking::TcpFactory**, stanowiąca realizację interfejsu **networking::ISocketFactory**, tworzy instancje klasy **networking::TcpSocket**, natomiast klasa **networking::SslFactory** tworzy instancje klasy **networking::SslSocket**. Tworzenie odpowiedniego obiektu gniazda połączenia w przypadku obu aplikacji zostało zrealizowane z wykorzystaniem polimorficznego wskaźnika na klasę bazową:

std::unique_ptr<ISocketFactory>.

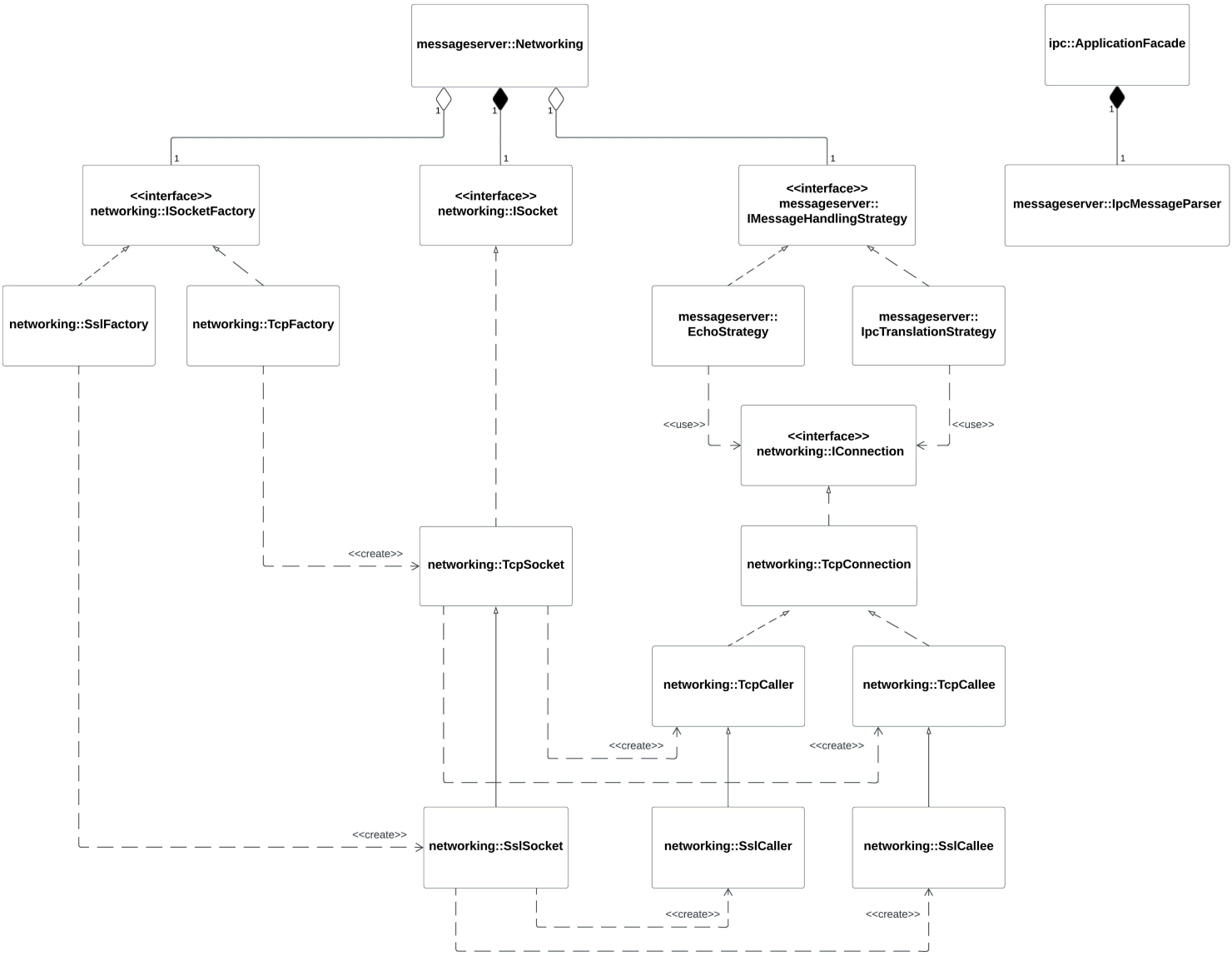
Aplikacja graficznego interfejsu użytkownika składa się z dwóch głównych klas: **messengergui::Networking** oraz **MessengerView**. Klasa **messengergui::Networking** implementuje funkcje realizujące komunikację z serwerem TCP:

- ***void connect(const std::string & ipAddress, uint16_t portNumber)*** – funkcja tworzy gniazdo TCP, a następnie nawiązuje połączenie z serwerem TCP zlokalizowanym pod zadanym adresem IP (argument **ipAddress**) i numerem portu (argument **portNumber**). Funkcja wspiera wyłącznie adresy protokołu IPv4;
- ***void disconnect()*** – funkcja zamyka połączenie z serwerem TCP, wraz z istniejącym gniazdem połączenia;
- ***void sendMessage(const std::string & message)*** – funkcja wysyła wiadomość (argument **message**) do serwera TCP, jeśli zostało nawiązane połączenie;
- ***std::string receiveMessage()*** – funkcja odbiera wiadomość od serwera TCP (wartość zwracana z funkcji), jeśli zostało nawiązane połączenie;
- ***void setSocketFactory(std::unique_ptr<networking::ISocketFactory> factory)*** – setter umożliwiający ustawienie polimorficznego wskaźnika na obiekt interfejsu **networking::ISocketFactory**.

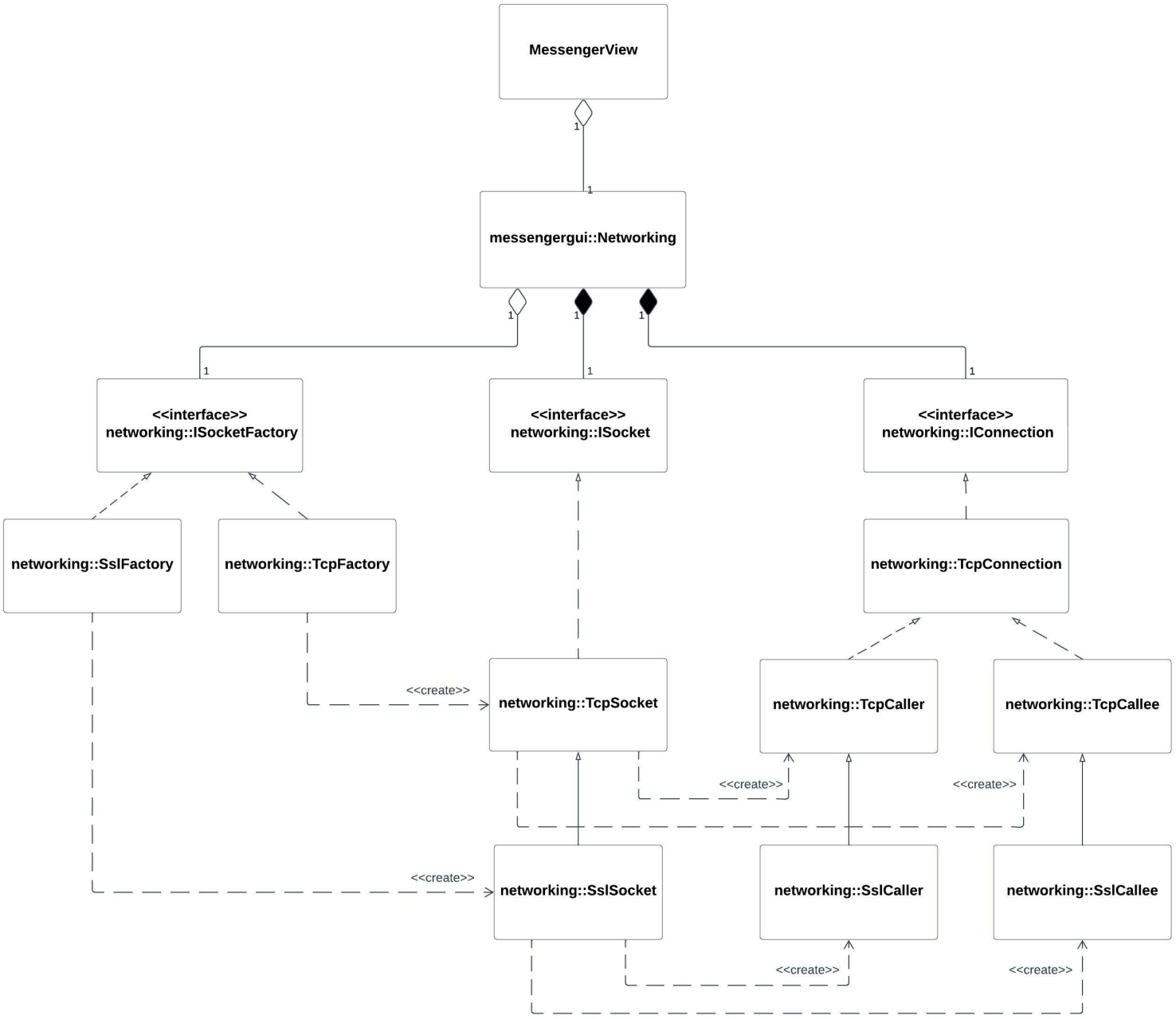
Klasa **MessengerView** implementuje interfejs użytkownika aplikacji *messenger_gui*. Obsługa poszczególnych elementów interfejsu realizowana jest przez wywołanie odpowiednich metod klasy **messengergui::Networking**.

Serwer wiadomości niszczy obiekty połączenia TCP w przypadku odebrania wiadomości **"stop"**.

Strukturę aplikacji serwera wiadomości w postaci diagramu klas UML przedstawiono na rys. 2.2, natomiast strukturę aplikacji graficznego interfejsu użytkownika na rys. 2.3.



Rys. 2.2. Diagram klas UML aplikacji *message_server*



Rys. 2.3. Diagram klas UML aplikacji *messenger_gui*

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) Przeanalizuj diagramy klas UML aplikacji serwera wiadomości i graficznego interfejsu użytkownika (rys. 2.2 i rys. 2.3) oraz kod biblioteki **networking**, a w szczególności klasy:

- **networking::ISocketFactory**;
- **networking::TcpFactory**;
- **networking::ISocket**;
- **networking::TcpSocket**;
- **networking::IConnection**;
- **networking::TcpConnection**);

a następnie odpowiedz na pytania:

1. Jaką metodę należy wywołać, aby utworzyć gniazdo połączenia TCP?
2. W jaki sposób można zamknąć gniazdo połączenia TCP?
3. Jakie metody i w jakiej kolejności należy wywołać, aby skonfigurować utworzone gniazdo TCP do pracy w trybie nasłuchowym (konfiguracja serwera TCP)?
4. Jakie metody i w jakiej kolejności należy wywołać, aby połączyć utworzone gniazdo TCP z gniazdem nasłuchowym serwera TCP (konfiguracja klienta TCP)?

5. Jaką metodę należy wywołać, aby wysłać wiadomość za pośrednictwem protokołu TCP?

6. Jaką metodę należy wywołać, aby odebrać wiadomość za pośrednictwem protokołu TCP?

Zadanie 2. (na ocenę 3.0) Uzupełnij brakującą implementację klasy **messengergui::Networking** aplikacji graficznego interfejsu użytkownika (*messenger_gui*), zgodnie z przeznaczeniem poszczególnych funkcji (rozdział 2.7):

- ***void connect(const std::string & ipAddress, uint16_t portNumber)***;
- ***void sendMessage(const std::string & message)***;
- ***std::string receiveMessage()***.

Następnie przetestuj działanie aplikacji łącząc się z serwerem wiadomości uruchomionym na osobnym komputerze lub lokalnie (***localhost*** − **adres IP 127.0.0.1**; decyduje prowadzący zajęcia). Zweryfikuj pracę serwera wiadomości w trybie **ECHO**. Adres IP komputera można sprawdzić wywołując z poziomu konsoli systemowej polecenie: **hostname -I**. W protokole z przebiegu ćwiczenia zapisz metody biblioteki ***networking*** zastosowane do implementacji poszczególnych funkcji klasy **messengergui::Networking**.

Zadanie 3. (na ocenę 4.0) Przeanalizuj kod aplikacji serwera wiadomości oraz graficznego interfejsu użytkownika, a następnie korzystając z diagramów klas UML obu aplikacji (rys. 2.2 i rys. 2.3) zidentyfikuj co najmniej 3 różne zastosowane wzorce projektowe. W protokole z przebiegu ćwiczenia zapisz nazwy rozpoznanych wzorców projektowych; wymień klasy, które je współtworzą oraz określ relacje między tymi klasami (dziedziczenie / realizacja / asocjacja / agregacja / kompozycja).

Zadanie 4. (na ocenę 5.0) Uruchom aplikację serwera wiadomości, a następnie wykorzystując program **wireshark** zaobserwuj wymieniane pakiety

protokołu TCP między serwerem a aplikacją graficznego interfejsu użytkownika:

- przy nawiązywaniu połączenia między aplikacjami;
- podczas przesyłania wiadomości tekstowych między aplikacjami;
- podczas zamykania połączenia (za pomocą wiadomości **"stop"**, za pomocą przycisku **disconnect**, przez nagłe zamknięcie aplikacji klienta TCP).

W protokole z przebiegu ćwiczenia zapisz zaobserwowane pakiety TCP wraz z informacją o kierunku komunikacji (klient → serwer / serwer → klient). **Uwaga: program wireshark nie umożliwia obserwacji pakietów wymienianych w ramach hosta lokalnego (adres IP 127.0.0.1).**

W sprawozdaniu zawrzeć:

- protokół z przebiegu ćwiczenia;
- najważniejsze cechy protokołu komunikacyjnego TCP;
- metody biblioteki ***networking*** zastosowane do implementacji poszczególnych funkcji klasy **messengergui::Networking**;
- rozpoznane wzorce projektowe (nazwa wzorca; klasy współtworzące dany wzorzec, relacje między klasami) [*jeśli zostało zrealizowane* ***Zadanie 3.***];
- schemat komunikacji między aplikacją graficznego interfejsu użytkownika a serwerem wiadomości z uwzględnieniem wymienianych pakietów protokołu TCP [*jeśli zostało zrealizowane* ***Zadanie 4.***].