Politechnika Wrocławska

Wydział Elektroniki,
Fotoniki i Mikrosystemów

Inżynieria Oprogramowania dla Elektromobilności

Ćwiczenie nr 5. Serializacja i deserializacja danych

Zagadnienia do opracowania:

- system kontroli wersji na przykładzie Git
- system budowania CMake
- paradygmat programowania obiektowego i jego cechy (hermetyzacja, abstrakcja, dziedziczenie, polimorfizm)
- klasa std::string
- kontener std::vector
- format JSON i jego specyfikacja
- pojęcia serializacji i deserializacji danych
- Test–Driven Development (TDD)
- Google Test framework

Literatura

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

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

[5] *Introducing JSON*. `https://www.json.org/json-en.html`. [Online; dostęp 21.11.2022].

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

[7] 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 serializacji i deserializacji danych z wykorzystaniem formatu JSON. Tematyka zajęć obejmuje ćwiczenia programowania w języku C++ w metodyce Test–Driven Development.

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**. Deklaracja testowanej klasy znajduje się w pliku nagłówkowym **TcpMessage.h** w katalogu **include**, natomiast definicje metod klasy umieszczono w pliku **TcpMessage.cpp** w katalogu **src**.

2.3. Struktura wiadomości TCP

Komunikacja między aplikacją serwera wiadomości a graficznym interfejsem użytkownika jest realizowana z wykorzystaniem biblioteki **networking**. Odbierane i wysyłane wiadomości są zapisane w postaci literałów znakowych przechowywanych w instancji klasy **std::string** (por. Ćw. 4):

- ***void networking::IConnection::send(const std::string & message)***
- ***networking::IConnection::Status networking::IConnection::receive(std::string & outputBuffer)***

Zanim wiadomość od klienta TCP zostanie przesłana do aplikacji procesora danych, musi zostać odpowiednio przetworzona. Pierwszym krokiem jest deserializacja transmitowanych danych do instacji klasy **messageserver::TcpMessage**. W tym celu format poprawnej wiadomości obsługiwanej przez aplikację serwera wiadomości został ustandaryzowany z wykorzystaniem notacji JSON:

```
{
   "command":"",
   "table":"",
   "record":[{"":""},{"":""},{"":""}]
}
```

Żadna z par klucz-wartość nie jest obowiązkowo wymagana w odebranej wiadomości TCP. Oznacza to, że najprostsza poprawna wiadomość będzie miała postać:

```
{}
```

Przykład poprawnej wiadomości zawierającej żądanie wykonania kwerendy

SELECT na tabeli *users* dla kolumny *surname* o wartości *Kowalski* wygląda następująco:

```
{
   "command":"select",
   "table":"users",
   "record":[
      {"surname":"Kowalski"}
   ]
}
```

Klasa **messageserver::TcpMessage** posiada trzy prywatne atrybuty odpowiadające poszczególnym elementom poprawnej wiadomości TCP (patrz plik **TcpMessage.h**):

- **sqlCommand** – przechowuje literał łańcuchowy przypisany do klucza **command** (jeśli przesłana wiadomość zawiera klucz **command**; np. *"select"*);
- **sqlTable** – przechowuje literał łańcuchowy przypisany do klucza **table** (jeśli przesłana wiadomość zawiera klucz **table**; np. *"users"*);
- **sqlRecord** – przechowuje listę par literałów łańcuchowych reprezentujących obiekty znajdujące się w tablicy przypisanej do klucza **record** (jeśli przesłana wiadomość zawiera klucz **record**; np. *"surname"–"Kowalski"*).

W związku z tym, **każda wartość zawarta w wiadomości TCP musi być literałem łańcuchowym** (napisem). Każdy z atrybutów posiada swój *getter* oraz *setter*:

```
std::string messageserver::TcpMessage::getSqlCommand() const;
std::string messageserver::TcpMessage::getSqlTable() const;
```

std::vector<messageserver::TcpMessage::SqlColumnEntry> messageserver::TcpMessage::getSqlRecord() const;

void messageserver::TcpMessage::setSqlCommand(const std::string & command);
void messageserver::TcpMessage::setSqlTable(const std::string & table);
void messageserver::TcpMessage::setSqlRecord(const std::vector< messageserver::TcpMessage::SqlColumnEntry> & record);

W celu przesłania odpowiedzi od aplikacji procesora danych do graficznego interfejsu użytkownika należy przeprowadzić serializację instancji klasy **messageserver::TcpMessage** do postaci literału łańcuchowego. Operacje serializacji i deserializacji realizowane są za pomocą odpowiednich metod klasy **messageserver::TcpMessage**:

- **std::string messageserver::TcpMessage::serialize() const;**
- **void messageserver::TcpMessage::deserialize(const std::string & content);**

2.4. Biblioteka rapidjson

Biblioteką języka C++, która w ramach zajęć zostanie wykorzystana do przeprowadzenia serializacji i deserializacji danych będzie **rapidjson**. Pełną dokumentację biblioteki można znaleźć na stronie: `https://rapidjson.org/index.html`. Biblioteka jest dołączona do aplikacji serwera wiadomości jako podmoduł systemu kontroli wersji Git (*git submodule*).

Utworzenie pustego obiektu JSON jest realizowane przez instancję klasy **rapidjson::Document**:

```
// Utworz pusty dokument
rapidjson::Document json;
// Ustaw dokument jako obiekt JSON
json.SetObject();
```

Parsowanie literału łańcuchowego zawierającego dane w formacie JSON można przeprowadzić wywołując na utworzonym obiekcie funkcję **rapidjson::Document::Parse()**:

```
std::string data = "{\"command\":\"backup\"}"
// Utworz pusty dokument
rapidjson::Document json;
// Przeparsuj literal lancuchowy
json.Parse(data.c_str());
```

Zapis danych z instancji klasy **rapidjson::Document** do zmiennej typu **std::string** przeprowadza się przy pomocy pomocniczej klasy szablonowej **rapidjson::Writer**:

```
// Zmienne pomocnicze
rapidjson::StringBuffer buffer;
rapidjson::Writer<rapidjson::StringBuffer> writer(
   buffer);
// Utworz pusty dokument
rapidjson::Document json;
// Wypelnij obiekt JSON
{...}
// Przyjmij obiekt klasy Writer
json.Accept(writer);
```

```
// Zapisz stan obiektu JSON do postaci lancucha
   znakowego
std::string data = buffer.GetString();
```

Dodawanie par klucz–wartość do obiektu JSON jest realizowane za pomocą metody **rapidjson::Document::AddMember()**. Zarówno klucz, jak i wartość przekazywane są w postaci instancji klasy **rapidjson::Value**. Aby ustawić zawartość literałów łańcuchowych należy posłużyć się metodą **rapidjson::Value::SetString()**:

```
// Utworz pusty dokument
rapidjson::Document json;
// Ustaw dokument jako obiekt JSON
json.SetObject();

// Zmienne pomocnicze
rapidjson::Value key;
rapidjson::Value value;
auto & memoryAllocator = json.GetAllocator();

// Dodaj pare klucz-wartosc ("command":"select")
key.SetString("command", memoryAllocator);
value.SetString("select", memoryAllocator);
json.AddMember(key, value, memoryAllocator);
```

W celu dodania tablicy do obiektu JSON należy skorzystać z funkcji **rapidjson::Value::SetArray()**. Dodawanie kolejnych elementów do tablicy jest realizowane z wykorzystaniem metody **rapidjson::Value::PushBack()**:

```
// Utworz pusty dokument
rapidjson::Document json;
// Ustaw dokument jako obiekt JSON
```

```
4  json.SetObject();
5
6  // Zmienne pomocnicze
7  rapidjson::Value key;
8  rapidjson::Value value;
9  auto & memoryAllocator = json.GetAllocator();
10
11 // Zdefiniuj pare klucz-wartosc
12 // Ustaw klucz jako "request"
13 key.SetString("request", memoryAllocator);
14 // Ustaw wartosc jako tablice []
15 value.SetArray();
16
17 // Dodaj obiekt {"surname":"Kowalski"} do tablicy
18 // Zmienne pomocnicze
19 rapidjson::Value entry;
20 rapidjson::Value keyEntry;
21 rapidjson::Value valueEntry;
22
23 // Ustaw pusty obiekt JSON
24 entry.SetObject();
25
26 // Zdefiniuj pare klucz-wartosc
27 keyEntry.SetString("surname", memoryAllocator);
28 valueEntry.SetString("Kowalski", memoryAllocator);
29
30 // Dodaj pare klucz-wartosc ("surname":"Kowalski")
      do pustego obiektu JSON
31 entry.AddMember(keyEntry, valueEntry,
      memoryAllocator);
32 // Dodaj obiekt JSON do tablicy
```

```
33 value.PushBack(entry, memoryAllocator);
34
35 // Dodaj pare klucz-wartosc ("request":[{"surname":"
      Kowalski"}])
36 json.AddMember(key, value, memoryAllocator);
```

Aby pobrać parę klucz–wartość z obiektu JSON za pomocą określonego klucza można posłużyć się metodą **rapidjson::Document::FindMember()**. Jeżeli zadany klucz został znaleziony, to funkcja zwróci iterator do odpowiedniej pozycji w obiekcie JSON, albo iterator do pozycji poza obiektem JSON w przeciwnym wypadku. Pobieranie klucza jest realizowane za pomocą atrybutu **name**, natomiast pobieranie wartości za pomocą atrybutu **value** iteratora:

```
1  std::string data = "{\"command\":\"select\",\"table
      \":\"users\",\"request\":[{\"surname\":\"Kowalski
      \"}]}"
2  // Utworz pusty dokument
3  rapidjson::Document json;
4  // Przeparsuj literal lancuchowy
5  json.Parse(data.c_str());
6
7  // Zmienne pomocnicze
8  std::string key;
9  std::string value;
10
11 // Znajdz pare klucz-wartosc za pomoca klucza "table
      "
12 rapidjson::Value::ConstMemberIterator iterator =
      json.FindMember("table");
13
```

```
// Sprawdz czy zwrocony iterator wskazuje na poprawna pozycje
if (iterator != json.MemberEnd()) {
    // Pobierz klucz "table"
    key = iterator->name.GetString();
    // Pobierz wartosc przypisana do klucza "table"
    value = iterator->value.GetString();
}
```

Należy pamiętać, że żądanie niewłaściwego typu danych grozi niezdefiniowanym zachowaniem. Warto przeprowadzić weryfikację typu przed pobraniem danych:

```
// Znajdz pare klucz-wartosc za pomoca klucza "request"
rapidjson::Value::ConstMemberIterator iterator = json.FindMember("request");

// Sprawdz czy zwrocony iterator wskazuje na poprawna pozycje
if (iterator != json.MemberEnd()) {
    if (iterator->value.IsString()) {
        auto value = iterator->value.GetString();
        // Przetworz pobrana wartosc
        {...}
    } else if (iterator->value.IsArray()) {
        auto value = iterator->value.GetArray();
        // Przetworz pobrana wartosc
        {...}
    } else if (iterator->value.IsObject()) {
        auto value = iterator->value.GetObject();
        // Przetworz pobrana wartosc
```

```
        {...}
    } else {
        // Obsluga pozostalych przypadkow
        {...}
    }
}
```

Po pobranej w ten sposób tablicy można iterować, np. za pomocą zakresowej pętli *for* (*range–based for loop*). Iterator można wykorzystać do pobrania pary klucz–wartość również wtedy, gdy nie znamy zawartości klucza:

```
// Znajdz pare klucz-wartosc za pomoca klucza "request"
rapidjson::Value::ConstMemberIterator iterator = json.FindMember("request");

// Sprawdz czy zwrocony iterator wskazuje na poprawna pozycje
if (iterator != json.MemberEnd()) {
    // Sprawdz czy pobierana wartosc jest tablica
    if (iterator->value.IsArray()) {
        // Pobierz tablice
        auto array = iterator->value.GetArray();
        // Iteruj po kolejnych elementach tablicy
        for (const auto & entry : array) {
            // Jezeli element tablicy jest obiektem JSON
            if (entry.IsObject()) {
                // Pobierz wszystkie pary klucz-wartosc
                for (auto iter = entry.MemberBegin(); iter != entry.MemberEnd(); ++iter) {
```

```
16                      // Jezeli klucz i wartosc sa
    literalami lancuchowymi
17                      if (iter->name.IsString() and
    iter->value.IsString()) {
18                          // Pobierz klucz
19                          auto name = iter->name.
    GetString();
20                          // Pobierz wartosc
21                          auto value = iter->value.
    GetString();
22                          // Przetworz pobrana pare
    klucz-wartosc
23                          {...}
24                      }
25                  }
26              }
27          }
28      }
29  }
```

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) Zapoznać się z formatem poprawnej wiadomości TCP obsługiwanej przez aplikację serwera wiadomości (rozdział 2.3), a następnie zapisać kolejne kroki algorytmu sprawdzającego czy odebrany literał łańcuchowy zawiera dane w postaci poprawnie sformatowanego obiektu JSON.

Zadanie 2. (na ocenę 3.0) Uzupełnij brakującą implementację funkcji **serialize()** klasy **messageserver::TcpMessage** aplikacji serwera wiadomości (*message_server*), zgodnie z jej przeznaczeniem (rozdział 2.3). Skorzystaj z zaimplementowanych testów jednostkowych, aby przetestować działanie funkcji (**TcpMessageSerializationTest.cpp**). W protokole z przebiegu ćwiczenia zapisz kolejne kroki algorytmu serializacji instancji klasy **messageserver::TcpMessage** do postaci łańcucha znaków.
Uwaga: Poprawna implementacja funkcji poskutkuje bezbłędnym zakończeniem wszystkich testów z pliku TcpMessageSerializationTest.cpp.

Zadanie 3. (na ocenę 4.0) Uzupełnij brakującą implementację funkcji **deserialize()** klasy **messageserver::TcpMessage** aplikacji serwera wiadomości (*message_server*), zgodnie z jej przeznaczeniem (rozdział 2.3). Skorzystaj z zaimplementowanych testów jednostkowych, aby przetestować działanie funkcji (**TcpMessageDeserializationTest.cpp**). W protokole z przebiegu ćwiczenia zapisz kolejne kroki algorytmu deserializacji łańcucha znaków do instancji klasy **messageserver::TcpMessage**.

Uwaga: Poprawna implementacja funkcji poskutkuje bezbłędnym zakończeniem wszystkich testów z pliku TcpMessageDeserializationTest.cpp.

Zadanie 4. (na ocenę 5.0) Przeanalizuj publiczny interfejs klasy **messageserver::TcpMessage** (plik **TcpMessage.h**), a następnie odpowiedz na pytanie:
Dlaczego dane przesłane za pomocą protokołu TCP po deserializacji do instancji klasy messageserver::TcpMessage nie są bezpieczne przed niepowołaną/przypadkową modyfikacją?
Zaproponuj modyfikacje publicznego interfejsu klasy, które zapewniłyby integralność danych po deserializacji, przy jednoczesnym zachowaniu funkcjonalności klasy.

W sprawozdaniu zawrzeć:

- protokół z przebiegu ćwiczenia;
- wnioski i własne przemyślenia na temat pracy w metodyce Test–Driven Development;
- kolejne kroki algorytmu serializacji instancji klasy **messageserver::TcpMessage** do postaci łańcucha znaków
- kolejne kroki algorytmu deserializacji łańcucha znaków do instancji klasy **messageserver::TcpMessage** [*jeśli zostało zrealizowane* ***Zadanie 3.***];
- propozycje modyfikacji publicznego interfejsu klasy **messageserver::TcpMessage** w celu zapewnienia integralności danych przesłanych przy pomocy protokołu TCP po deserializacji [*jeśli zostało zrealizowane* ***Zadanie 4.***].