UNIWERSYTET GDAŃSKI
Wydział Matematyki, Fizyki i Informatyki

Jakub Ronkiewicz
nr albumu: 238155

Wsparcie dla pracowni 3D

Praca magisterska na kierunku:
INFORMATYKA
Promotor:
dr Piotr Arłukowicz

Gdańsk 2020

Streszczenie

Temat podjęty przeze mnie w pracy magisterskiej został przez mnie wybrany w celu ułatwienia komunikacji w pracowniach 3D. Zauważyłem przestrzeń do poprawy efektywności przeprowadzenia zajęć bądź warsztatów. Patrząc z perspektywy prowadzącego zajęcia, w przypadku większej liczby podopiecznych, można dostrzec kłopot związany ze śledzeniem postępów ich pracy. Ponadto projekt umożliwiłby sprawne przeprowadzanie zajęć w trybie zdalnym. Celem pracy było utworzenie wtyczki działającej w programie Blender. Na początku pracy opisuje wszystkie wymagania dotyczące budowanego projektu. W kolejnym rozdziale opisuję, czym jest Blender oraz historię jego powstania. Następnie określam, czym są wtyczki i jak je budować w prawidłowy sposób. W tym rozdziale znalazł się również krótki opis udostępnianego API Blendera, którego zrozumienie jest niezbędne przy tworzeniu jakichkolwiek skryptów. Prezentuję sposób działania już gotowego projektu, pokazując wszystkie możliwe scenariusze, zarówno ze strony studenta, jak i wykładowcy. Przedstawiam sposób, w jaki testowałem gotowy produkt oraz rezultaty tych testów. W ostatnim rozdziale poruszam problematykę warstwy sieciowej, której poznanie było niezbędne do utworzenia projektu umożliwiającego komunikację pomiędzy maszynami. Znajdują się tam zagadnienia związane z modelem warstwowym TCP/IP, protokołami transmisji, identyfikacji. Prowadzę dyskusję na temat potencjalnych rozwiązań dla warstwy sieciowej wtyczki. Tłumaczę obraną drogę, jaką był wybór socketów ZMQ. Obrazuję przewagę ZMQ nad innymi rozwiązaniami. Opisuję wszystkie napotkane problemy i ich sposoby rozwiązania za pomocą wzorców zawartych w bibliotece ZMQ.

Słowa kluczowe

Blender, programowanie sieciowe, tcp/ip, asynchroniczność, zmq

Spis treści

ROZDZIAŁ 1

Opis projektu

Głównym celem projektu było ułatwienie pracy wykładowcy oraz studentów, w pracowniach komputerowych, w których wykorzystywany jest Blender. Minimalnym założeniem było umożliwienie wykładowcy otrzymania od studenta projektu w postaci pliku `.blend`. Taka wymiana miała przebiegać bezpośrednio w interfejsie blendera bez korzystania z osobnych aplikacji. Projekty studentów powinny być zapisywane na komputerze wykładowcy we wskazanej przez niego lokalizacji. Wykładowca powinien mieć możliwość szybkiego sprawdzenia postępów studentów. To zadanie miało być zrealizowane za pomocą automatycznego przesyłania zrzutów ekranów do wykładowcy. Kolejnym założeniem było umożliwienie wykładowcy sprawnego przeglądania tychże zrzutów, bezpośrednio we wtyczce. Ponadto dodatkową funkcją była możliwość wysyłania wiadomości od studentów bezpośrednio do wykładowcy. Wtyczka miała pracować na protokole TCP/IP.

ROZDZIAŁ 2

Budowa wtyczki

2.1. Blender - opis programu

Blender jest programem służącym do tworzenia wszystkiego co związane z grafiką 3D, jest nieoceniony dla twórców filmów animowanych, efektów specjalnych, interaktywnych aplikacji 3D, a także gier komputerowych. Udostępnia wiele funkcji jak modelowanie 3D, mapowanie UV, edytowanie grafiki rastrowej, teksturowanie, rzeźbienie modeli 3D. Zawiera w sobie silniki pozwalające na symulację zdarzeń fizycznych, takie jak symulacja cieczy i dymu, symulacja cząsteczek. Jest oparty na licencji open-source. Jedną z zalet takiej licencji jest rozbudowana pomocna społeczność. Datą narodzin Blendera jest 2 stycznia 1994, kiedy to holenderskie studio animacyjne *NeoGeo* opublikowało pierwsze kody źródłowe. Stabilna wersja 1.00 została wypuszczona na rynek w styczniu 1995 roku. Za autora programu uznawany jest Ton Roosendaal. W 1998 firma rozpadła się. Ton stworzył nową markę pod nazwą *Not A Number Technologies*, wtedy nastąpił dalszy rozwój programu. W tym czasie aplikacja udostępniana była na licencji shareware. W 2002 roku firma upadła. W celu dalszego rozwoju aplikacji autor utworzył organizację non-profit o nazwie *Blender Foundation*. Ton był pomysłodawcą akcji o nazwie *Free Blender*, której celem było zebranie środków w wysokości 100,000 €. Skutkiem tego, było przejście Blendera na model open-source. Blender jest aplikacją wieloplatformową, dostępną na takich systemach jak Microsoft Windows, MacOs i Linux.

2.2. Blender API

Twórcy Blendera udostępnili interfejs publiczny aplikacji (API), dzięki któremu cała społeczność może tworzyć skrypty oraz addony, przyspieszające pracę grafików 3D. W obecnym momencie API udostępnia klasy i metody pozwalające na:

- Edytowanie danych, które można modyfikować poprzez interfejs użytkownika
- Modyfikacja preferencji użytkownika np. mapowanie klawiszy
- Uruchamianie narzędzi z predefiniowanymi ustawieniami
- Tworzenie własnych menu, nagłówków, czy paneli
- Tworzenie własnych narzędzi
- Tworzenie narzędzi interaktywnych
- Tworzenie własnych silników do renderingu, które integrują się z Blenderem
- Śledzenie zmian w danych i ich właściwościach
- Definiowanie nowych ustawień dla wbudowanych obiektów Blendera
- Tworzenie obiektów 3D

Dane z aktualnie wczytanego pliku `.blend` udostępnianie są przez moduł `bpy.data`. Z jego pomocą możemy otrzymać dane utworzonych obiektów, scen czy materiałów.

2.3. Struktura budowy wtyczki

Wtyczki rozszerzają funkcjonalność Blendera. Ich użycie jest opcjonalne. Niektóre wbudowane w Blendera addony są domyślnie wyłączone, aby niepotrzebnie nie obciążać zasobów maszyny. Zapisywane są w folderze `scripts/addons`.

Przy starcie uruchamiane są tylko te wtyczki, które zostały zaznaczone w ustawieniach użytkownika. Blender rozpoznaje skrypt jako wtyczkę, jeśli znajdzie się w nim zmienna `bl_info`, która zawiera informacje o nazwie, autorze i kategorii. Wtyczka musi mieć określoną strukturę w odróżnieniu od zwykłego skryptu, który wykonuje się w sposób linearny.

2.3.1. Panele

Interfejs Blendera złożony jest w większości z paneli. W celu zdefiniowania panelu należy utworzyć klasę dziedziczącą po `bpy.types.Panel`. W klasie należy nadać wartości następującym zmiennym:

- *bl_idname* - nazwa panelu
- *bl_label* - nagłówek panelu
- *bl_space_type* - określa, w której sekcji programu zostanie umieszczona wtyczka, dostępne możliwości są następujące: `EMPTY`, `VIEW_3D`, `IMAGE_EDITOR`, `NODE_EDITOR`, `SEQUENCE_EDITOR`, `CLIP_EDITOR`, `DOPESHEET_EDITOR`, `GRAPH_EDITOR`, `NLA_EDITOR`, `TEXT_EDITOR`, `CONSOLE`, `INFO`, `TOPBAR`, `STATUSBAR`, `OUTLINER`, `PROPERTIES`, `FILE_BROWSER`, `PREFERENCES`
- *bl_region_type* - wybrany region sekcji, która została podana w *bl_space_type*
- *bl_context* - jeśli chcemy by wtyczka pojawiała się tylko w trybie np. edycji obiektu należy podać `editmode`

W metodzie `draw` implementujemy interfejs panelu. Obiekt `layout` klasy pozwala na rysowanie przycisków, etykiet, formularzy oraz projektowania układu za pomocą metod `column`, `row`. We wtyczce został utworzony tylko jeden panel, który w zupełności wystarcza do wygodnej obsługi. Poniżej znajduję się jego skrócony listing.

```
class OBJECT_PT_CustomPanel(Panel):
    bl_label = "Lecture Connector"
    bl_idname = "OBJECT_PT_custom_panel"
```

```
    bl_category = "Lecture"

    bl_space_type = "VIEW_3D"
    bl_region_type = "UI"

    def draw(self, context):
        layout = self.layout
        mytool = context.window_manager.socket_settings
        scene = bpy.context.scene
        wm = context.window_manager

        try:
            import zmq
            if not mytool.is_connected:
                layout.prop(mytool, "connection_type", text="")
                layout.prop(mytool, "port")

                if mytool.connection_type == 'Client':
                    layout.prop(scene, "networks")
                    layout.prop(mytool, "ip")
                    layout.prop(mytool, "login")
                else:
                    layout.prop(mytool, 'path')
```

2.3.2. Operatory

To właśnie operatory są jednym z najważniejszych elementów w strukturze. Najczęściej ich wykonanie następuje po wciśnięciu przycisku wyświetlanego przez panel. Po rejestracji klasy operatora, możemy go również wykonać w dowolnym miejscu za pomocą `bpy.ops.IDNAME()`. Operator tworzymy poprzez utworzenie klasy dziedziczącej po `bpy.types.Operator`. Zmienne, które należy uzupełnić to: `bl_idname` oraz `bl_label`. Opcjonalnie możemy dodać opcje za pomocą `bl_options`. Klasa `bpy.types.Operator` umożliwia im-

plementację predefiniowanych metod takich jak `poll, invoke, execute, draw, modal` oraz `cancel`. Najważniejszą metodą, którą należy zaimplementować jest `execute(self, context)`. To właśnie w niej znajdują się polecenia, które zostaną wykonane po naciśnięciu przycisku. Metoda `poll` jest niejakim walidatorem, gdy zwróci `False`, metoda `execute` się nie wykona. Poniżej został przedstawiony przykład operatora odpowiedzialnego za wysyłanie wiadomości do wykładowcy, z pozycji studenta.

```
class WM_OT_SendMessage(bpy.types.Operator):
    bl_idname = "wm.send_message"
    bl_label = "Send Msg"

    def execute(self, context):
        data = context.window_manager.socket_settings
        rep_socket = context.window_manager.rep_address.encode('utf-8')
        login =  data.login.encode('ascii')
        uid = data.uid.encode('utf-8')
        msg = data.message.encode('utf-8')
        socket_pub = bpy.types.WindowManager.socket

        socket_pub.send_multipart([b'msg', rep_socket, login, uid, msg])
        data.message = ""
        self.report({'INFO'}, f"Message: {msg.decode('utf-8')} sent")

        return {'FINISHED'}
```

2.3.3. Properties (Właściwości)

Podczas projektowania wtyczki wielce prawdopodobnym jest to, że będziemy musieli w jakiś sposób zarządzać danymi. W celu uporządkowania tychże danych możemy posłużyć się klasą `PropertyGroup`, tworząc klasę dziedziczącą po niej. Dzięki temu grupujemy opcje, które są ze sobą powiązane w jednym obiekcie. Dostępnymi typami właściwości są: `BoolProperty, BoolVectorProperty, CollectionProperty, EnumProperty, FloatProperty,`

`FloatVectorProperty`, `IntProperty`, `IntVectorProperty`, `PointerProperty`, `RemoveProperty`, `StringProperty`. Jako przykład poniżej zaprezentowano listing, który implementuje strukturę przechowującą wszystkie najważniejsze dane niezbędne do działania wtyczki:

```
class ChatProperties(PropertyGroup):
    port : IntProperty(
        name = "Port number",
        description="Port number of client or server",
        default = 5550,
        min = 1024,
        max = 65535
        )

    ip : StringProperty(
        name = "Lecturer IP",
        description = "IP address of lecturer",
        default="127.0.0.1"
    )
    ...
```

2.3.4. Operatory modalne oraz timery

Projekt wymagał tego, aby wtyczka działała w tle, w sposób nieblokujący interfejsu, po to by użytkownik mógł swobodnie korzystać z programu. Zwykłe operatory blokują interfejs, dopóki metoda `execute()` się nie wykona. Z pomocą przychodzą operatory modalne oraz timery. We wtyczce skorzystano z timerów. W momencie utworzenia połączenia przez studenta lub wykładowcę, odpowiednio funkcja `timed_msg_poller_for_student` lub `timed_msg_poller_for_lecturer` jest rejestrowana jako timer za pomocą polecenia `bpy.app.timers.register(self.timed_msg_poller_for_student)`. W przypadku studenta dodatkowo zaimplementowano funkcję, która wysyła zrzut ekranu do wykładowcy co minutę. Wymaga to jej rejestracji jako timer. Poniżej przedstawiono listing tej funkcji.

```
def send_screens_periodically(self):
  socket = bpy.types.WindowManager.socket

  if socket:
      login =  self.socket_settings.login.encode('ascii')
      uid = self.socket_settings.uid.encode('utf-8')
      rep_socket = bpy.types.WindowManager.rep_address.encode('utf-8')

      screenshot("screen.png")

      with open("/tmp/screen.png", 'rb') as file:
          data = file.read()

      socket.send_multipart([b'img', rep_socket, login, uid, data])
  self.report({'INFO'}, 'Screen sent successfully')

  return 60
```

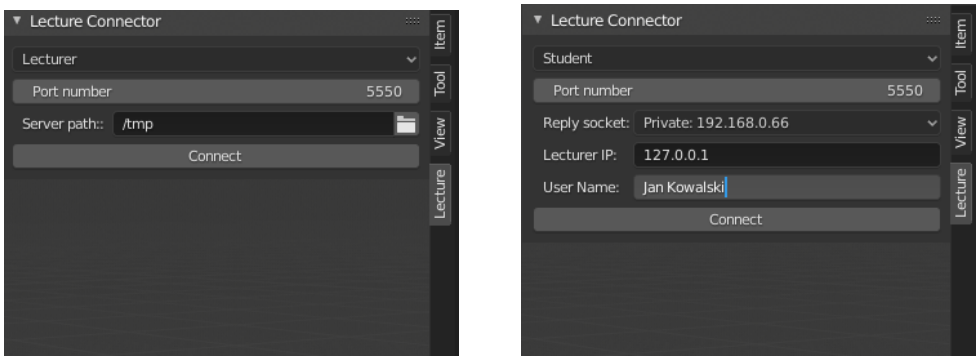
2.3.5. Rejestracja, inicjalizacja komponentów

Zwykłe skrypty nie wymagają rejestracji modułów, komponentów czy klas. W zasadzie w przypadku skryptów nie jest zalecane rejestrowanie jakichkolwiek komponentów. Natomiast przechodząc do tematu budowania wtyczki, etap rejestracji jest niezbędny do poprawnego działania. Wynika to z tego, że wtyczkami możemy w dowolny sposób zarządzać, wykonywać takie czynności jak włączanie/wyłączanie wtyczki, instalacja/deinstalacja. Rejestracja klasy powoduje załadowanie jej implementacji do blendera. Od tego momentu jest ona dostępna tak jak i inne wbudowane funkcjonalności. Po zarejestrowaniu klasy mamy do niej dostęp z `bpy.types.bl_idname`. Do konkretnej klasy odwołujemy się za pomocą pola `bl_idname`, a nie za pomocą nazwy. Listing poniżej przedstawia implementację rejestrowania i wyrejestrowywania klas niezbędnych do prawidłowego działania wtyczki.

```
classes = (
        PIPZMQProperties,
        PIPZMQ_OT_pip_pyzmq,
        WM_OT_EstablishConnection,
        WM_OT_SendMessage,
        WM_OT_CloseConnection,
        WM_OT_CloseClient,
        WM_OT_SendFile,
        WM_OT_SendScreen,
        STUDENT_UL_items,
        STUDENT_OT_actions,
        STUDENT_OT_send,
        StudentObject,
        ChatProperties,
        OBJECT_PT_CustomPanel
    )


def register():
    from bpy.utils import register_class, previews
    for cls in classes:
        register_class(cls)

def unregister():
    from bpy.utils import unregister_class
    for cls in reversed(classes):
        unregister_class(cls)
```

2.4. Opis działania wtyczki

Addon umieszczony jest w *Viewporcie 3D* w panelu *Tools* W pierwszym polu z listy rozwijanej wybieramy typ użytkownika: *lecturer* lub *student*. W zależności od wybranej opcji panel wyświetla inne pola.

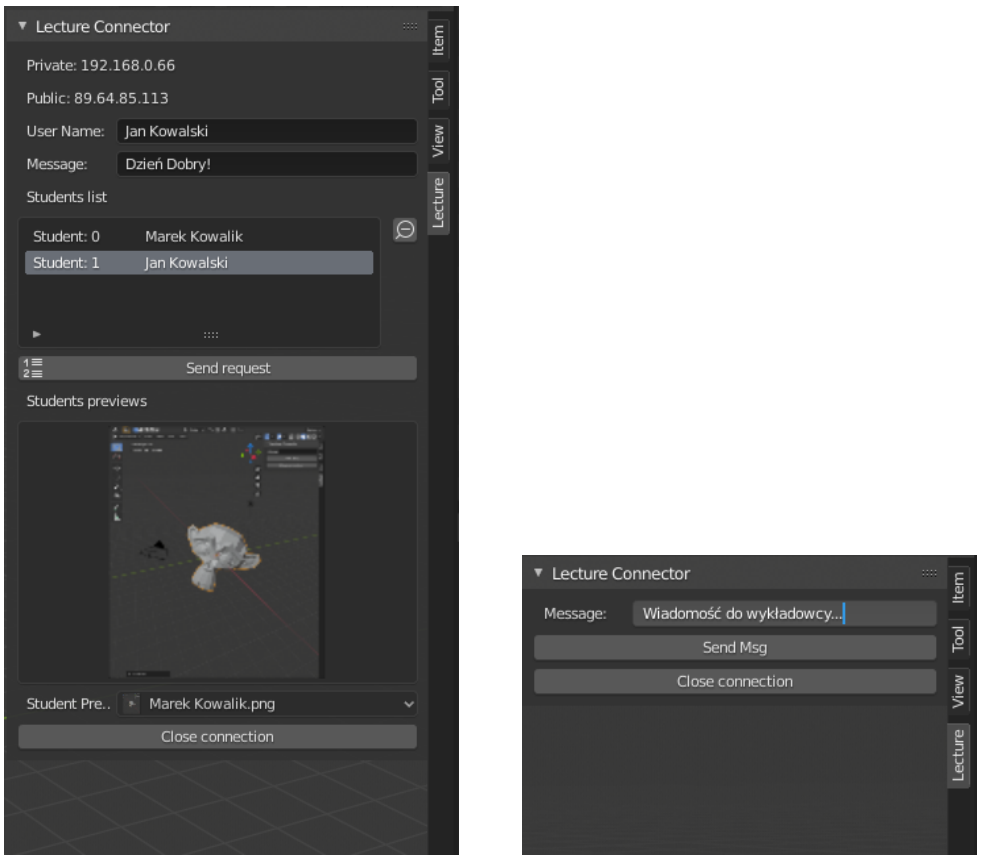


(a) Panel dla wykładowcy (b) Panel dla studenta

Rysunek 2.1: Panel przed utworzeniem połączenia

Źródło: Opracowanie własne

Wykładowca może ustawić port, na którym ma nasłuchiwać socket *SUB*. Nie ma konieczności wprowadzania adresu IP, *ZMQ* wiąże wszystkie dostępne interfejsy sieciowe wykładowcy na wskazanym porcie. Wybiera ścieżkę, w której mają być zapisywane wszystkie screenshoty oraz pliki *.blend*.
Student w momencie połączenia się z wykładowcą musi ustawić odpowiedni port oraz adres IP wykładowcy. Z listy rozwijanej wybiera adres IP dla socketu *REP*, który będzie odpowiadał na żądania wykładowcy o plik *.blend*. Do wyboru ma publiczny adres badź prywatny. Warto wspomnieć, że przy wyborze publicznego adresu IP, router musi mieć ustawione przekierowywanie portów na odpowiednie urządzenie w sieci lokalnej. Port socketu *REP* wybierany jest w sposób losowy. Cały adres wraz z portem przekazywany jest wykładowcy. W celu szybszej identyfikacji studentów przez wykładowcę, student podaję imię i nazwisko.



(a) Panel dla wykładowcy (b) Panel dla studenta

Rysunek 2.2: Panel po nawiązaniu połączenia

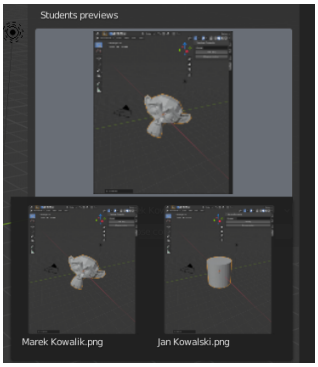
Źródło: Opracowanie własne

Student po połączeniu z wykładowcą ma możliwość wysyłania krótkich wiadomości lub zakończenia połączenia.
Wykładowca posiada znacznie bogatszy interfejs. Na samej górze znajdują się informację o adresach IP, które może przekazać studentom w celu nawiązania połączenia. Niżej umieszczone są pola wyświetlające ostatnią dostarczoną wiadomość. Na środku panelu znajduje się lista studentów. Wykładowca może usuwać wpisy z tej listy za pomocą przycisku znajdującego się po prawej stronie.

Mechanizm pobierania pliku *.blend* od studenta działa następująco:

1. Wykładowca wybiera studenta z listy
2. Kliknięcie przycisku *Send Request* powoduje wysłanie żądania do studenta

Jeśli student odpowie na żądanie wykładowcy, otworzy się nowe okno Blendera z otwartym plikiem *.blend* studenta. Jeśli po określonej liczbie prób student nie odpowie, zostaje usunięty z listy. Poniżej umiejscowiony jest panel do przeglądania postępów studentów za pomocą screenshotów, które studenci dostarczają co minutę.



Rysunek 2.3: Wyświetlanie postępów studentów

Źródło: Opracowanie własne

2.5. Zarządzanie zewnętrznymi bibliotekami w Blenderze

Blender korzysta z własnego wbudowanego interpretera języka Python. Dzieje się tak nawet wtedy, gdy na maszynie mamy zainstalowaną własną wersję. Niezmiennie wykorzystywany będzie interpreter znajdujący się w katalogu instalacyjnym Blendera. Popularnym menadżerem pakietów dla Pythona jest

pip. Od wersji 2.81 *pip* jest dostarczany razem z interpreterem. We wtyczce wykorzystano zewnętrzną bibliotekę *pyzmq*. W bloku try/except została umieszczona instrukcja importująca tą bibliotekę. W momencie, gdy zostanie rzucony wyjątek *ModuleNotFoundError*, w panelu pojawi się przycisk umożliwiający instalację modułu `pyzmq`. Guzik ten jest obsługiwany przez odpowiedni operator.

```
try:
    import pip
except ModuleNotFoundError as e:
    install_props.install_status += f"\nPip import failed. {e}"
    import ensurepip
    ensurepip.bootstrap()
    os.environ.pop("PIP_REQ_TRACKER", None)
```

W przypadku, gdy moduł *pip* nie zostanie znaleziony, moduł *ensurepip* próbuje go zainstalować w odpowiedni sposób.

```
pybin = bpy.app.binary_path_python
if subprocess.call([pybin, '-m', 'pip', 'install', 'pyzmq']) != 0:
    install_props.install_status += "\nCouldn't install pyzmq."
    self.report({'ERROR'}, "Couldn't install pyzmq.")
    return {'CANCELLED'}

self.report({'INFO'}, "pyzmq installed! READY!")
```

Sposób instalacji pakietu pyzmq

Potrzebujemy pełnej ścieżki do interpretera Pythona, z którego korzysta Blender. Zmienna *pybin* przetrzymuje tą wartość. Moduł subprocess pozwala uruchamiać inne procesy z poziomu Pythona. Funkcja call uruchamia proces zgodny z podanymi argumentami, czeka na wykonanie polecenia i zwraca wartość zwróconą przez proces.

2.6. Testy wydajnościowe

W celu przetestowania wtyczki, zaimplementowałem prostego klienta wysyłającego wiadomości do wykładowcy. Uruchamiam 10 klientów równolegle, z czego każdy wysyła 100 wiadomości. Listing poniżej prezentuje implementację klienta.

```
class Client:

    def __init__(self, ip, port, login):
        self.uid = str(uuid4()).encode()
        self.login = login.encode('ascii')
        self.zmq_ctx = zmq.Context().instance()
        self.url = f"tcp://{ip}:{port}"
        self.socket = self.zmq_ctx.socket(zmq.PUB)
        self.socket.connect(self.url)
        print("Connection with lecturer estabilished")
        self.rep_socket = '192.168.0.22'.encode() # mockup
        sleep(1)

    def send_message(self, msg):
        encoded_message = msg.encode()
        print(f"Sending message: {msg} from {self.uid.decode()}")
        self.socket.send_multipart([b'msg', self.rep_socket,
        self.login, self.uid, encoded_message])

    def send_100_messages(self, boilerplate="Test message: #"):
        for x in range(100):
            self.send_message(f"{boilerplate}{x}")
            sleep(0.001)


if __name__ == "__main__":
```

```
    client = Client('127.0.0.1', '5550', 'user')
    client.send_100_messages()
```

Oczekuję otrzymania 1000 wiadomości po stronie wykładowcy. Mierzę również czas w jakim to zadanie zostanie wykonane. Poniższe polecenie powoduje uruchomienie 10 klientów w jednym momencie.

```
for i in {1..10}; do echo -n 'python3 simple_client.py & '; done | bash -
```

Poniżej prezentuję część implementacji odpowiedzialnej za sprawdzenie, czy 1000 wiadomości zostało odebranych.

```
elif mode == 'msg':
    msg = self.socket_settings.message = msg.decode('utf-8')
    messages.append(f"{uid}@{msg}")
    if len(messages) == 1000:
        end = time.time()
        print(f"Enough messages! Elapsed {end - start_time}")
        with open('/tmp/test', 'wb') as f:
            pickle.dump(messages, f)
        messages = []
```

W momencie, w którym wykładowca otrzyma wszystkie wiadomości następuje serializacja listy z wiadomościami, w celu jej późniejszej weryfikacji. Dodatkowo do każdej wiadomości dodaję unikalny identyfikator użytkownika, aby sprawdzić czy ilość klientów i otrzymanych przez nich wiadomości się zgadza. Po każdym wykonaniu uruchamiam test napisany przy pomocy biblioteki `pytest`. Poniżej znajduję się jego listing.

```
import pytest
import pickle

def open_file():
    with open('./test', 'rb') as f:
        return pickle.load(f)
```

```
class TestAddon:
    file = open_file()

    def test_should_get_10_users(self):
        users_set = set(u.split('@')[0] for u in self.file)
        assert len(users_set) == 10

    def test_every_user_should_get_100_messages(self):
        users_messages = {}
        for message in self.file:
            user, mess = message.split('@')
            if user in users_messages:
                users_messages[user].append(mess)
            else:
                users_messages[user] = [mess]
        print(users_messages)
        count_messages = [len(value) for user,value in users_messages.items()]
        assert count_messages.count(100) == 10
```

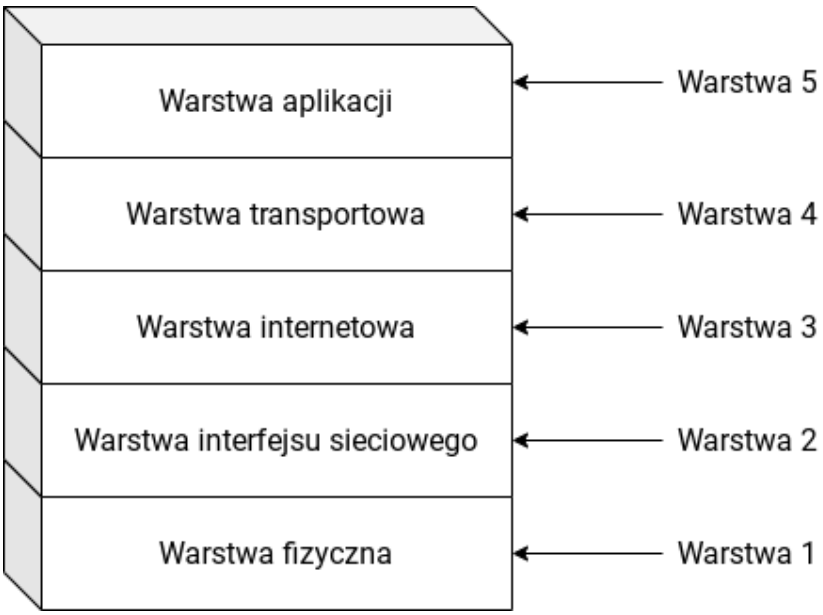
Średni pomiar czasu z 10 testów plasuje się na poziomie 5.5s

ROZDZIAŁ 3

Warstwa sieciowa

3.1. Model warstwowy TCP/IP

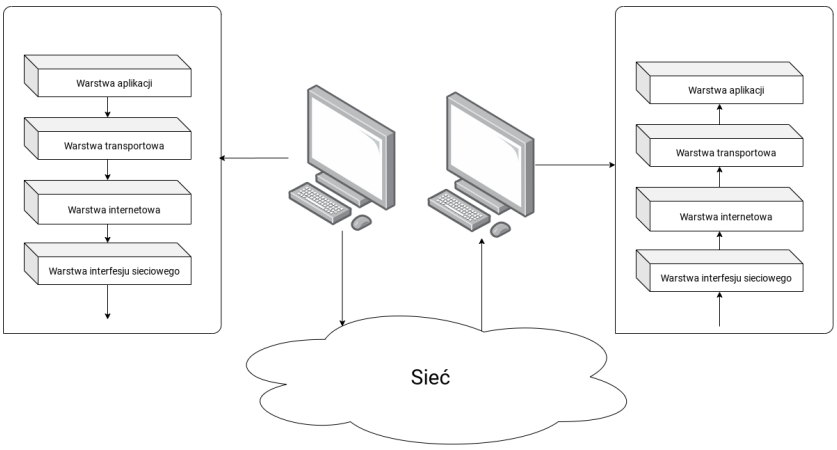
Warstwa fizyczna
Protokoły znajdujące się w warstwie fizycznej, są niskopoziomowe, związane blisko ze sprzętem, opisują wszelkie właściwości elektryczne i radiowe.
Warstwa interfejsu sieciowego
Protokoły warstwy interfejsu sieciowego, są niejako mostem pomiędzy warstwami wyższego rzędu (rozwiązywane programowo), a niższego rzędu (rozwiązywane sprzętowo).
Warstwa internetowa
Warstwa ta odpowiada za połączenie pomiędzy dwoma komputerami i ich komunikację z wykorzystaniem internetu. Protokoły znajdujące się w tej warstwie opisują adresowanie, sposób dzielenia pakietów internetowych na mniejsze bloki, a także obsługę błędów w trakcie połączenia.
Warstwa transportowa
Protokoły w tej warstwie zarządzają/kontrolują połączeniem pomiędzy aplikacjami zainstalowanymi na obu komputerach. Dbają o odpowiednią prędkość transmisji oraz odpowiednią kolejność dostarczania danych.
Warstwa aplikacji
Najwyższa warstwa, która jest najbliższa aplikacji, określa rodzaje wysyłanych wiadomości i sposobu reakcji na nie, poprzez odpowiednie procedury.



Rysunek 3.1: Warstwy sieci

Źródło: Opracowanie własne

Prześledźmy sposób działania protokołów i ich komunikację pomiędzy warstwami. Pakiety wędrują po kolei od jednej warstwy do następnej. Wydajność połączenia jest zapewniona poprzez stosowanie wskaźników na pakiety, zamiast ich kopiowania z jednej warstwy do drugiej. Komputer nr 1 chcąc wysłać dane, tworzy pakiet, który przechodzi przez wszystkie warstwy od aplikacji do interfejsu sieciowego. Mając już dane na poziomie sprzętowym możemy je przetransmitować do Komputera nr 2, który z kolei przetwarza dane przez warstwy ułożone w odwrotnej kolejności. Otrzymuje on dane na najniższym poziomie. Tym samym rozpoczyna od warstwy interfejsu sieciowego, przechodzi sukcesywnie od warstwy do warstwy, aż osiągnie warstwę aplikacji. Pozwoli to zaprezentować dane w sposób czytelny dla użytkownika.



Rysunek 3.2: Sposób działania warstw sieciowych

Źródło: Opracowanie własne

3.2. Protokoły TCP/UDP

W komunikacji internetowej wyróżniamy dwa sposoby wymiany danych. Możliwa jest ona za pomocą strumieni lub komunikatów (datagramów).
TCP - *Transmission Connection Protocol* - opiera się na formacie danych określanej jako strumień. Protokół ten ma charakter połączeniowy. Wiąże się to z koniecznością potwierdzenia nawiązania połączenia przez obie strony. Takie potwierdzenie popularnie bywa nazywanę metodą *handshaking'u*. TCP obsługuje komunikację tylko i wyłącznie jeden-do-jednego. Przekazywane dane mogą być dowolnej długości. Transmitowane są jako sekwencje pojedynczych bajtów lub bloki bajtowe. Natomiast to sieć dostosowuje ilość bajtów przesyłanych w danej chwili. Zatem może dojść zarówno do podziału bloku danych na mniejsze fragmenty lub odwrotnie, do połączenia mniejszych fragmentów w jeden większy blok. Odpowiedzialny jest za to algorytm Nagle'a. Należy podkreślić, że protokół TCP gwarantuje wysłanie danych we właściwej kolejności. Wykorzystywany jest w większości aplikacji.
Cechy protokołu TCP:

- Niezawodny - TCP zarządza potwierdzeniami wiadomości, retransmisją

i opóźnieniami. Dopuszczalne jest kilka prób wysyłania wiadomości. Jeśli w trakcie wysyłania wystąpi jakikolwiek błąd, serwer podejmie ponowną próbę wysłania utraconej części wiadomości

- Gwarancja kolejności - wiadomości są wysyłane w założonej kolejności
- Zasobożerny - TCP wymaga 3 pakietów do utworzenia połączenia z socketem (handshaking), przed wysłaniem jakiekolwiek wiadomości. Tymi pakietami są SYN, SYN+ACK, ACK

UDP - *User Datagram Protocol* - bazuje na transporcie komunikatów. Protokół UDP ma charakter bezpołączeniowy, odmiennie niż w protokole TCP. Zawsze wysyłany jest cały komunikat, nie występuje tu dzielenie na mniejsze fragmenty, ani łączenie w jeden większy. UDP nie ogranicza się do komunikacji jeden-do-jednego. Komunikat może być wysyłany, także do większej grupy odbiorców, czyli jeden-do-wielu. Należy nadmienić, że protokół umożliwia także komunikację wiele-do-jednego. Jedną z większych wad tego protokołu jest, brak gwarancji otrzymania komunikatów w kolejności, w której zostały nadane. Ponadto nie ma gwarancji, że w ogóle takowy komunikat dotrze do adresata. Niestety, może również dojść do duplikacji komunikatów. Protokół UDP wykorzystywany jest głównie w aplikacjach multimedialnych, takich jak transmisje wideo.
Cechy protokołu UDP:

- Zawodny - brak potwierdzenia połączenia, retransmisji czy obsługi opóźnień
- Brak gwarancji kolejności - kolejność wysyłania wiadomość nie musi zostać zachowana
- Lekki - wydajny, szybki, idealny do aplikacji multimedialnych

3.3. Identyfikacja maszyn

W celu zapewnienia jednoznacznego identyfikowania maszyn, utworzony został protokół IP. Adres IP w wersji 4 składa się z 32 bitów. Najczęściej można

spotkać zapis czterech liczb oddzielonych kropkami. W zapisie kropkowym każdy z czterech bajtów prezentuję się jako liczbę dziesiętną (np. 192.168.1.1). Każdy adres składa się z części identyfikującej podsieć i części identyfikującej hosta. Maszyna podłączona do sieci Internet ma dwa rodzaje adresów, publiczny i prywatny. Publiczny zostaje przydzielony przez dostawcę usług. W zależności od dostawcy, bywa dynamiczny bądź statyczny. Natomiast adres prywatny identyfikuje hosta w lokalnej sieci, dzięki temu urządzenia mogą się komunikować nawet bez dostępu do sieci Internet. Ponadto występuje inny sposób komunikacji między procesami. Ograniczając się do jednej maszyny dochodzimy do wniosku, że adres IP nie jest wystarczający by rozróżnić procesy. W tym celu korzystamy z portów. Do wyznaczenia numeru portu mamy określony limit 16 bitów. Liczby od 0 do 1023 są zajęte przez powszechnie znane usługi takie jak FTP, SSH czy HTTP. Natomiast pozostały zakres portów jest dostępny do wykorzystania i nadal pozostaje dość spory.

3.4. Przegląd dostępnych rozwiązań

Istnieje wiele dostępnych rozwiązań, które na pierwszy rzut oka pozwoliłyby zaimplementować warstwę sieciową. Na pierwszy ogień wybrałem podstawowe sockety. Jednak w przypadku zwykłych socketów pojawił się problem z blokowaniem interfejsu użytkownika. Socket służący jako serwer musiał ciągle nasłuchiwać. Wymagało to skorzystania z nieskończonej pętli, która blokowała interfejs. Podjałem próbę przeniesienia tego socketu na osobny wątek. Społeczność Blendera nie zalecała korzystania we wtyczce z osobnych wątków, z powodu niestabilności i potencjalnych wycieków pamięci. Niezbędnym było znalezienie rozwiązania, które działałoby w sposób asynchroniczny. Następną próbą było skorzystanie z biblioteki asyncio. Jednakże w przypadku asyncio pojawiały się konflikty z wewnętrzną pętlą blendera. Potencjalnie dobrym rozwiązaniem był asynchroniczny serwer HTTP, który mógł zostać zaimplementowany za pomocą biblioteki aiohttp. Uznałem, że projekt nie wymaga serwera http i szukałem rozwiązania niskopoziomowego. Ostatecznie wybór padł na ZeroMQ. W następnym rozdziale opisuje jego wady i zalety.

3.5. ZMQ Sockets

ZeroMQ, znane również jako ØMQ, 0MQ, ZMQ.
Geneza nazwy: Zero - oznacza minimalizm tej usługi, natomiast MQ - Message Queue oznacza kolejkę komunikatów. Minimalizm objawia się w tym, że podstawowe API zawiera około 50 funkcji. Dodatkowo biblioteka ta zawiera implementacje dla wielu języków programowania, również dla Pythona, w którym to głównie napisany jest Blender. Teraz przedstawię przewagi 0MQ nad zwykłymi socketami:

- Rozwiązuje problemy z przepełnieniem kolejki komunikatów (high water mark). Jeśli kolejka jest pełna, ZMQ automatycznie blokuje nadawców, albo odrzuca komunikaty, w zależności od wybranego wzorca.
- W bibliotece znajdują się gotowe implementacje wzorców sieciowych, dzieki czemu można w szybki sposób stworzyć własną topologię sieci, dostosowaną do problemu, który rozwiązujemy.
- Dostarcza pełne wiadomości w sposób niezawodny, używając specjalnego ramkowania danych.
- Inteligentnie obsługuje błędy. W przypadkach, gdzie ma to sens, próbuje ponownie wysyłać niedostarczone wiadomości.
- API jest minimalistyczne. Implementacja prostego serwera/klienta zajmuje mniej linii niż w tradycyjnych socketach, przez co kod jest bardziej przejrzysty.
- Wysyłając wiadomość podzieloną na kilka części za pomocą funkcji socket.send_multipart() mamy pewność, że wiadomość dotrze w całości albo w ogóle nie zostanie dostarczona.

Wyobraźmy sobie, że uruchamiamy klienta przed postawieniem serwera. Przy korzystaniu z tradycyjnych socketów, niezwłocznie otrzymamy błąd, który trzeba będzie w konkretny sposób obsłużyć. ZeroMQ pozwala uruchamiać i zatrzymywać konkretne usługi bezwarunkowo. Tak szybko, jak klient wykona

polecenie zmq.connect(), połączenie zostanie nawiązane, nawet jeśli serwer nie został jeszcze uruchomiony. W takiej sytuacji wiadomości kolejkują się po stronie klienta, czekając tylko na moment uruchomienia serwera, jeśli taki nadejdzie wiadomości zostaną dostarczone.

Różnice pomiędzy ZMQ a tradycyjnym protokołem TCP

- Sockety TCP są z góry zaprojektowane do relacji 1 do 1, natomiast sockety ZMQ są zaprojektowane jeden do wielu (w zależności od wybranego typu socketu).
- Możliwość wysyłania wiadomości za pomocą wielu protokołów takich jak tcp, pgm, inproc. Jeśli będziemy mieli ochotę zmienić protokół, zmiana taka będzie bezbolesna, prosta i szybka.
- Jeśli socket podłączy się za pomocą metody socket.connect() automatycznie akceptuje połączenia. Metoda accept() stosowana w tradycyjnym protokole TCP jest zbędna.
- Połączenie sieciowe ma miejsce w tle. Zmq automatycznie próbuje nawiązać ponowne połączenie w przypadku rozłączenia (np. gdy student się rozłączy i połączy ponownie).
- Hermetyzacja, czytelniejszy kod, mniejsze prawdopodobieństwo popełnienia błędów.
- ZMQ ma więcej podobieństw do protokołu UDP. Zarządza komunikatami w podobny sposób, w przeciwieństwie do strumienia bajtów, które ma miejsce w TCP.

Operacje wejścia/wyjścia są wykonywane asynchronicznie, w wątkach w tle. Komunikacja z wątkami aplikacji przebiega za pomocą struktur wolnych od zachowań blokujących wątek. Oznacza to, że wiadomości są ładowane do lokalnej kolejki wejścia i wysyłane z lokalnej kolejki wyjścia, przez co nie blokują aplikacji i nie ma znaczenia, czy aplikacja jest czymś zajęta. W związku z tym, metoda zmq_send() w zasadzie nie jest odpowiedzialna

za wysłanie wiadomości do endpointu, tylko wrzuca wiadomości do kolejki, tak aby później wątek I/O mógł wysłać ją w sposób asynchroniczny. Z reguły zachowanie blokujące nie istnieje, z wyjątkiem pewnych scenariuszy. Podsumowując, wykonanie zmq_send() nie gwarantuje tego, że wiadomość została wysłana. TCP w zeromq można określić jako disconnected TCP, ponieważ zeromq nie wymaga, aby endpoint istniał przed połączeniem się z nim. Zarówno klienci jak i serwery mogę się łączyć i stawiać w każdej chwili.

3.5.1. Komunikaty w ZMQ

Komunikaty w ZMQ mogą mieć dowolną wielkość, dopóki mieszczą się w pamięci. Warto dokonać serializacji danych w wybrany przez siebie sposób jak np. json. Rozsądnym wyjściem jest wybór reprezentacji danych, która jest przenośna.

Ramkowanie danych
Ramka w zmq jest specyfikowana przez długość wysyłanych danych od zera wzwyż. W trakcie powstawania biblioteki komunikat w ZMQ był jedną ramką, jak w UDP. Na dalszych etapach rozwoju dodano rozszerzenie, umożliwiające wysyłanie wieloczęściowych wiadomości. Składają się one z serii ramek z flagą informującą "czy jest więcej części do odczytania" na 1, ostatnia część wiadomości ma tą flagę ustawioną na 0. Przydatne informacje na temat komunikatow w ZMQ:

- Komunikat może być pojedynczy lub wieloczęściowy
- Części komunikatu nazywamy ramkami
- Każda ramka to obiekt typu *zmq_msg_t*
- Wysyłanie i odbieranie ramek przebiega oddzielnie, w niskopoziomowym API

- API wyższego poziomu dostarcza wrapper, który wysyła całą wiadomość wieloczęściową. Gwarantuje to dostarczenie całej wiadomości. Wysłanie jej części jest niemożliwe.

- ZMQ nie wysyła wiadomości (pojedynczej jak i wieloczęśćiowej) od razu, tylko po pewnym nieokreślonym czasie. Z tego względu wiadomość wieloczęściowa musi mieścić się w pamięci. Wysyłanie dużych plików powinno być odpowiednio obsłużone. W wypadku, gdy plik nie mieści się w pamięci, powinien zostać podzielony na części i wysłany za pomocą jednostkowych wiadomości.

3.5.2. Protokół HTTP w ZMQ?

W ZMQ nie jesteśmy w stanie zaimplementować 100% serwera *HTTP*. Wynika to ze sposobu w jaki ramkowane są dane. Tradycyjny protokół *HTTP* używa CR-LF jako delimiter ramek, natomiast ZMQ używa ramek typu *length-specified*. W ZMQ istnieje wzorzec nazywany *REQ-REP*, który działa podobnie do *HTTP*. Jednakże od wersji 3.3 istnieje opcja *ZMQ_ROUTER_RAW*, która pozwala zdefiniować własną metodę ramkowania. Korzystając z niej jesteśmy w stanie napisać pełnoprawny protokół *HTTP*.

3.5.3. Wydajność ZMQ

ZeroMQ wykonuje operacje I/O w wątku pracującym w tle. Jeden wątek I/O (dla wszystkich socketów) jest wystarczający dla normalnych aplikacji, w których obsługa danych na sekundę nie przekracza 1GB. W przypadku, gdy potrzebujemy szybszego przetwarzania danych, możemy ustawić flagę ZMQ_IO_THREADS za pomocą funkcji zmq_ctx_set(). Tradycyjna sieciowa aplikacja używa jednego procesu/wątku dla jednego połączenia (socketu), natomiast ZMQ używa tylko jednego procesu/wątku dla całej infrastruktury, zapewniając jednocześnie skalowalność.

3.5.4. Dostępne wzorce sieciowe w ZMQ

ZMQ kieruje i kolejkuje wiadomości zgodnie z precyzyjnymi przepisami, zwanymi wzorcami. Wzorce zostały zaprojektowane parami, tak aby jeden typ socketu pasował do drugiego. Dostępne wzorce ZMQ:

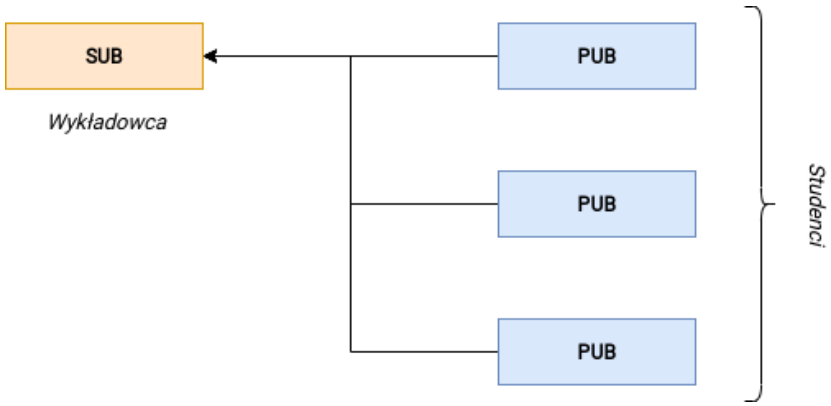
- **Request-Reply**, łączy zbiór klientów ze zbiorem serwisów.
- **Pub-Sub**, łączy zbiór publisherów ze zbiorem subskrybentów.
- **Pipeline**
- **Exclusive Pair**

Typy socketów (każda ze stron może wiązać gniazdo(bind))

- PUB-SUB
- REQ - REP
- REQ - ROUTER
- DEALER - REP
- DEALER - ROUTER - pozwala na asynchroniczne działanie typu request-response. Klient(REQ) rozmawia z ROUTEREM, a Serwer(DEALER) rozmawia z REP
- DEALER - DEALER
- ROUTER - ROUTER
- PUSH - PULL
- PAIR - PAIR

3.6. Architektura sieciowa projektu

Moim celem było utworzenie jak najlżejszej topologii sieciowej, która byłaby niezawodna i spełniała wszystkie postawione warunki. Pierwszym pomysłem, było skorzystanie ze wzorca Subscriber - Publisher. Domyślnie to publikator związuje socket z danym adresem i portem, a subskrybenci podłączają się pod niego. Natomiast nic nie stoi na przeszkodzie, by sytuacja była odwrotna. Subskrybent, który w tej topologii oznacza wykładowcę, związuje socket na wszystkie dostępne interfejsy sieciowe w komputerze wykładowcy. W dalszym etapie wykładowca otrzymuje informacje na temat adresów IP: publicznego, w sytuacji gdy zajęcia odbywają się zdalnie i każdy pracuje w innej sieci, oraz prywatnego, w sytuacji gdy zajęcia odbywają się na uczelni i wszyscy studenci są w tej samej podsieci. Publikatorzy (studenci) podłączają się pod subskrybenta korzystając z adresu, który został udostępniony przez wykładowcę. Można stwierdzić, że niejako wykładowca subskrybuje sockety wszystkich studentów. Komunikacja tego typu jest bardzo szybka, nieblokująca, działa asynchronicznie.



Rysunek 3.3: Wzorzec SUB-PUB

Źródło: Opracowanie własne

Na początku pracy założyłem, że taki typ komunikacji jest wystarczający dla realizacji obranych celów. Dla socketów studentów został utworzony automatyczny timer, który wysyłał zarówno screenshoty ich projektów jak

i całe pliki .blend co minutę. Pliki .blend przy złożonych projektach mogą przybierać dość spore rozmiary. Wysyłanie tych plików w tak krótkich odstępach czasowych, niepotrzebnie obciążałoby sieć. Kolejnym pomysłem było utworzenie osobnego timera, dla wysyłania plików z większymi odstępami czasowymi. Okazało się, że to też nie rozwiązuje problemu. Szczególnie widoczne jest to, w przypadku, kiedy wykładowca chce dokładnie obejrzeć projekt tego studenta w danej chwili. Nie może tego zrobić, gdyż pliki aktualizują się co określony czas.

Optymalne zachowanie wtyczki pozwalałoby w dowolnym momencie na pobranie przez wykładowcę aktualnego projektu studenta, bez niepotrzebnego wysyłania plików wszystkich studentów, kiedy nie są one potrzebne. W tym celu utworzyłem dodatkowe sockety zarówno dla wykładowcy jak i studenta. Wykładowca otrzymał dodatkowy socket typu REQ, którym może odpytywać studentów o plik .blend, a studenci otrzymali socket typu REP, w którym nasłuchują żądań od wykładowcy. Pojawiła się tu pewna wątpliwość, czy socket typu REP nasłuchujący żądań od wykładowcy nie będzie blokujący, ale dzięki mechanizmom pollingu dołączonym do ZMQ, socket ten działa asynchronicznie i nie blokuje interfejsu Blendera studenta.

W tym rozwiązaniu pojawił się kolejny problem. W sytuacji, gdy student się rozłączy, a wykładowca wyśle mu żądanie o plik, komputer wykładowcy będzie oczekiwał na odpowiedź nieskończenie długo, blokując cały interfejs. W celu rozwiązania tego problemu zaimplementowałem dla socketu REP wykładowcy wzorzec Lazy Pirate Pattern.

Czym jest *Lazy Pirate Pattern*?
Jest to niezawodny model typu request-reply z pewnymi zmianami u klienta (request). Algorytm wygląda następująco:

- Skorzystaj z mechanizmu pollingu dla socketu REQ, zwróć odpowiedź tylko gdy jesteś pewien, że poprawna odpowiedź nadeszła

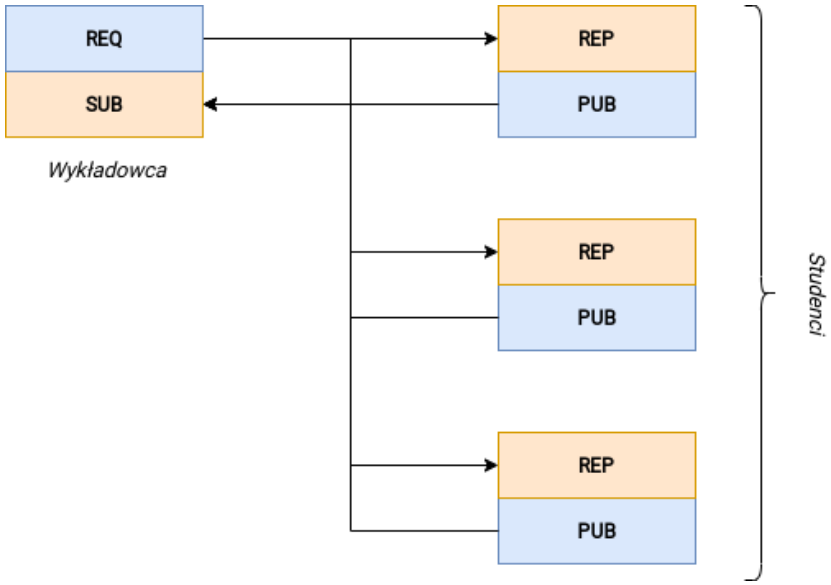
- Wyślij ponownie żądanie, jeśli odpowiedź nie nadeszła w określonym czasie
- Zakończ połączenie, jeśli odpowiedź nie nadeszła po określonej liczbie prób

Poniżej prezentuję implementację tego wzorca w metodzie odpowiedzialnej za wysłanie żądania o plik `.blend` do studenta i oczekiwanie na odpowiedź w postaci pliku.

```
def send_request_for_file(self, context, student):
    '''Lazy Pirate Pattern'''

    req = zmq.Context().instance().socket(zmq.REQ)
    req.connect(student.rep_socket)

    poll = zmq.Poller()
    poll.register(req, zmq.POLLIN)

    retries_left = self.request_retries
    while retries_left:
        self.report({'INFO'}, f"Request for file \
        sending to {student.name} binded to {student.rep_socket} \
        and waiting for reply..")
        req.send_string("SEND FILE")
        expect_reply = True
        while expect_reply:
            socks = dict(poll.poll(self.request_timeout))
            if socks.get(req) == zmq.POLLIN:
                data = req.recv()
                if self.open_file(context, student, data) == {'CANCELLED'}:
                    break
                req.disconnect(student.rep_socket)
                expect_reply = False
```

```
            retries_left = 0
        else:
            self.report({'WARNING'}, \
            'No response from student,retrying...')
            req.setsockopt(zmq.LINGER, 0)
            req.close()
            poll.unregister(req)
            retries_left-=1
            if retries_left == 0:
                self.report({'ERROR'}, \
                'Student seems to be offline, abandoning')
                remove_student(context)
                break
            self.report({'INFO'}, 'Reconnecting and resending')
            req = zmq.Context().instance().socket(zmq.REQ)
            req.connect(student.rep_socket)
            poll.register(req, zmq.POLLIN)
            req.send_string('SEND FILE')
```

Ostateczny diagram sieciowy prezentuje się następująco:



Rysunek 3.4: Architektura sieciowa

Źródło: Opracowanie własne

Zakończenie

Cel pracy został osiągnięty. Zbudowana wtyczka działa prawidłowo, zgodnie z pierwotnymi założeniami. Utworzony plugin może stać się cennym dodatkiem do Blendera i zostać wykorzystany w ośrodkach szkoleniowych na całym świecie. Dodatkowym atutem tego rozwiązania jest możliwość komunikowania się poprzez publiczne adresy IP, dzięki czemu zajęcia mogą być przeprowadzane zdalnie. W obecnej sytuacji panującej na świecie, może okazać się to nieocenione.

Bibliografia

[1] Douglas E. Comer. *Sieci komputerowe i intersieci Wydanie V*. HELION, 2012.

[2] David J. Wetherall Andrew S. Tanenbaum. *Sieci komputerowe. Wydanie V*. HELION, 2012.

[3] Witold Jaworski. *Programownie dodatków do Blendera 2.8*. 2019. https://airplanes3d.net/downloads/pydev2/pydev-blender-pl.pdf.

[4] Borman D. Jacobson V., Braden R. Tcp extensions for high performance. 1992. https://tools.ietf.org/html/rfc1323.

[5] Postel J. Rfc 793 - transmission control protocol. 1981. https://tools.ietf.org/html/rfc793.

[6] John Nagle. Congestion control in ip/tcp internetworks. 1984. https://tools.ietf.org/html/rfc896.

[7] Ton Roosendaal. How blender started, twenty years ago.... 2013. https://code.blender.org/2013/12/how-blender-started-twenty-years-ago/.

[8] Pieter Hintjens. *ØMQ - The Guide*. http://zguide.zeromq.org/page:all.

[9] Blender Foundation. *Blender Python API Documentation*. https://docs.blender.org/api/current/.

Spis rysunków

Oświadczenie

Ja, niżej podpisany(a) oświadczam, iż przedłożona praca dyplomowa została wykonana przeze mnie samodzielnie, nie narusza praw autorskich, interesów prawnych i materialnych innych osób.

.....	.....
data	podpis