*BARTOSZ CYWIŃSKI (304025) & ŁUKASZ STANISZEWSKI (304098)*

# REAL-TIME CLIENT-SERVER FACE MASK DETECTOR

*PROJEKT W RAMACH PRZEDMIOTU SKPS – ZESPÓŁ 42*

# Spis treści

# 1. OPIS PROJEKTU

## 1.1. WSTĘPNY OPIS PROJEKTU

W ramach projektu powstał system, rozpoznający w czasie rzeczywistym czy człowiek posiada założoną maskę chroniącą przed zakażeniem SARS-CoV-2. W ramach projektu powstało połączenie sieciowe między klientem, a serwerem, działającymi jako dwa oddzielne programy na dwóch osobnych systemach operacyjnych.

Serwer jest napisanym w języku Python skryptem, który pobiera obraz z kamery, wykrywa czy na obrazie jest jedna twarz i czy posiada ona założoną maskę. Następnie serwer przekazuje informację o tym co wykrył podłączonemu do niego klientowi.

Klient jest napisanym w języku Python skryptem, który bogaty jest w interfejs graficzny, za pomocą którego klient podłącza się do serwera. Po podłączeniu się do serwera przeprowadzany jest proces wykrywania maski na twarzy.

## 1.2. WYMAGANIA PROJEKTU

Przed projektowanym systemem (w całości) zostały postawione następujące wymagania:

- Powstanie połączenia klient-serwer z kanałem komunikacji zapewnianym przez sieć z użyciem gniazd socket. Połączenie zostało konkretnie opisane w punkcie 1.3.
- Klient i serwer działające na dwóch różnych systemach operacyjnych: odpowiednio Windowsie i Linuxie, przy czym klient działający na systemie Windows jest systemem operacyjnym gospodarza, a serwer działający na Linuxie jest wirtualizowanym systemem operacyjnym – gościem. Dokładny opis środowisk został opisany w punkcie 2.

Przed projektowanym systemem po stronie serwera zostały postawione następujące wymagania:

- Serwer jest napisanym w języku Python 3.9 skryptem, który uruchamia się automatycznie przy starcie systemu.
- Serwer nasłuchuje na wybranym porcie i umożliwia przyłączenie się do niego jednego klienta. Próba przyłączenia się drugiego klienta w trakcie wymiany informacji z pierwszym kończy się odrzuceniem jego połączenia.
- Poprawne przyłączenie się klienta do serwera gwarantuje zestawienie połączenia i stworzenie nowego wątku w ramach procesu serwera, w którym to w pętli (do momentu przerwania połączenia przez klienta) wysyłana jest do klienta informacja o wykrytej/niewykrytej masce.
- W przypadku zerwania połączenia przez klienta, serwer dzięki odpowiedniej obsłudze wyjątków bezpiecznie kończy połączenie i umożliwia przyłączenie się do niego nowemu klientowi.
- Obraz z kamery jest pobierany w czasie rzeczywistym, nawet przy dużym obciążeniu środowiska gościa, tzn. największa możliwa różnica czasu między pobraniem dwóch kolejnych klatek przez kamerę wynosi 300ms.

- Informacja o wykryciu zwracana przez serwer ograniczona jest do czterech możliwości: nie wykryto twarzy, brak maski na twarzy, maska na twarzy lub wykrycie więcej niż jednej twarzy. Razem z decyzją, dostarczana jest również informacja o pewności klasyfikatora co do decyzji.

Przed projektowanym systemem po stronie klienta zostały postawione następujące wymagania:

- Klient jest napisanym w języku Python 3.9 skryptem.
- Program posiada przyjazny dla użytkownika interfejs graficzny, przy użyciu którego może on sprawdzić czy przed kamerą znajduje się osoba i jeśli tak to czy ma założoną maskę.
- Główna procedura przypomina znaną z programowania połączenia z bazą danych technikę try-with-resource, gdzie klient, aby zdobyć informację, otwiera połączenie z serwerem, pobiera przez 5 sekund potrzebne informacje i zamyka bezpiecznie połączenie.
- Po przeprowadzeniu procesu wykrywania maski wyświetlany jest wynik otrzymany na wyjściu klasyfikatora. Oprócz samego wyniku na ekranie wyświetlane są też odnośniki do stron internetowych, gdzie kupić można maseczki COVID'owe oraz gdzie przeczytać można obecne obostrzenia dotyczące pandemii obowiązujące w Polsce. Ponadto w oknie, gdzie wyświetlany jest wynik, uruchamiane jest krótkie wideo dotyczące bezpieczeństwa w trakcie pandemii.

Przed samym projektem zostały postawione następujące wymagania:

- Kodem źródłowym napisanym w ramach projektu zarządzaliśmy za pomocą systemu kontroli wersji Git. Jako serwis dla projektu wybrany został GitHub.
- Powstałe repozytorium znajduje się pod adresem: https://github.com/bartooo/mask-detector.
- W ramach projektu zestawiony został potok CI/CD z wykorzystaniem GitHub Actions, który umożliwił automatyczne sprawdzenie przechodzenia testów jednostkowych, działanie zgodne z wersją Pythona 3.8 oraz 3.9, a także kontrolę stylu pisania kodu za każdym razem, gdy nastąpi zmiana w repozytorium.
- W ramach projektu przeprowadzony został szereg testów, które opisane są w punkcie 5.

## 1.3. OPIS POŁĄCZENIA

W ramach projektu, w celu komunikacji między procesami uruchomionymi w dwóch różnych systemach zestawione jest połączenie oparte na architekturze klient-serwer przez sieć z użyciem gniazd socket – punktów końcowych w dwukierunkowej komunikacji między programami działającymi w sieci. Klient w tym przypadku łączy się do serwera i odbiera wysyłane przez niego informacje, po czym kończy połączenie.

Działanie gniazda serwera opiera się na oczekiwaniu na request klienta. Aby było to możliwe, serwer na początku wykonuje z ang. bind – ustawia adres IP (w naszym przypadku IPv4) oraz port, czyli łącznie adres, pod jakim może on zostać znaleziony przez klienta.

Kiedy adres ten zostaje ustawiony, serwer oczekuje w uśpieniu na request ze strony klienta i gdy go otrzymuje, jest budzony. Jeśli chodzi o rodzinę adresów (format struktury adresu), została użyta rodzina AF_INET, która umożliwia komunikację między procesami działającymi w różnych systemach i wykorzystuje adresy IPv4, natomiast obsługiwane są strumieniowe typy gniazd (SOCK_STREAM), które są zorientowane na połączenie i wysyłają dane bez błędów czy powtórzeń przy założeniu, że są strumieniem bajtów – używany jest więc protokół kontroli transmisji TCP. Po stronie serwera, adresem IPv4 gniazda jest 0.0.0.0, co oznacza, że akceptowane jest każde połączenie wchodzące na adres IPv4 maszyny wirtualnej, natomiast serwer nasłuchuje na porcie 8006. Dodatkowo dla gniazda serwera ustalono opcję ponownego używania adresów (aby nie wystąpił błąd z zajętym portem). Po stworzeniu gniazda, w wyniku wykonania komendy netstat -lt na liście serwerów jest widoczny serwer:

```
luki@ubuntu:~$ netstat -lt
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 localhost:46825         0.0.0.0:*               LISTEN
tcp        0      0 localhost:domain        0.0.0.0:*               LISTEN
tcp        0      0 localhost:ipp           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:2115            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:8006            0.0.0.0:*               LISTEN
tcp6       0      0 ip6-localhost:ipp       [::]:*                  LISTEN
```

**RYS 1.1 – NASZ SERWER NA LIŚCIE SERWERÓW PO UŻYCIU NETSTAT -LT**

Po stronie gniazda klienta, na początku następuje jego stworzenie – korzysta on z dokładnie tej samej rodziny adresów (AF_INET), a także zorientowany jest na strumieniowe typy gniazd (SOCK_STREAM). Po jego stworzeniu, koniecznym jest połączenie go z gniazdem serwera poprzez podanie jego adresu IPv4 oraz portu. Adresem portu niezmiennie jest 8006, natomiast adres serwera wyczytać można poprzez zastosowanie polecenia ifconfig na maszynie wirtualnej – jest to 192.168.204.129. Ze względu jednak na fakt, że jest to połączenie w ramach jednej sieci lokalnej i jednego komputera, a między dwoma różnymi systemami, możliwe jest podanie zamiast adresu IPv4 nazwy hosta (serwera) – w tym przypadku będzie to ubuntu.

```
luki@ubuntu: ~
luki@ubuntu:~$ ifconfig
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.204.129  netmask 255.255.255.0  broadcast 192.168.204.255
```

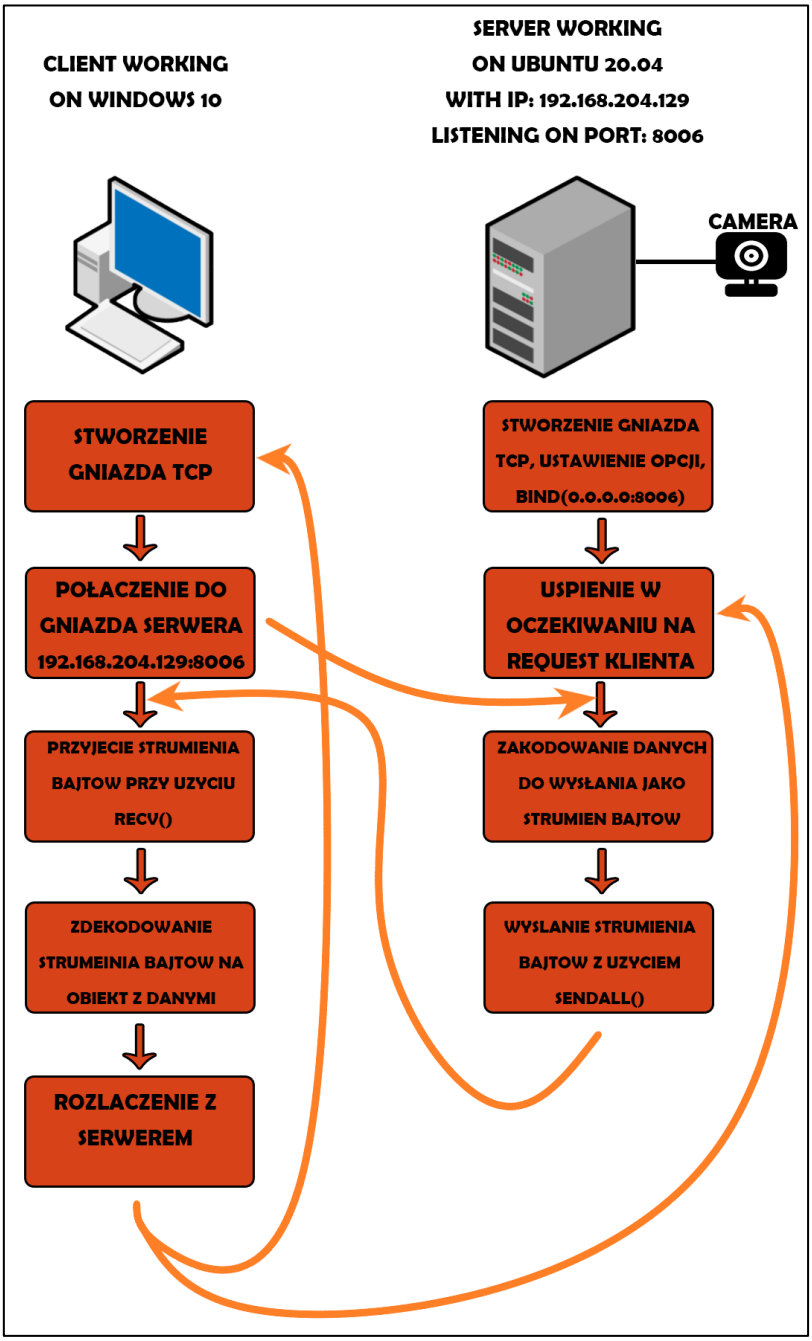
***RYS. 1.2 – IPv4 MASZYNY WIRTUALNEJ PO ZASTOSOWANIU POLECENIA IFCONFIG***

Jeśli chodzi o samą wymianę danych, po wykonaniu connect() po stronie klienta i accept() po stronie serwera, zestawiony jest dwukierunkowy kanał komunikacji, jednak w przypadku tego projektu dane są przesyłane tylko od serwera do klienta. Dlatego też serwer po odpowiednim przygotowaniu danych rozsyła je z użyciem polecenia sendall(), natomiast klient otrzymuje je z użyciem metody recv().

Warto również wspomnieć o tym jak dane są wysyłane między gniazdami. Jak wcześniej zostało wspomniane – transmisja jest zorientowana na przesyłanie danych w postaci strumieni bajtów, tak więc konieczne jest odpowiednie przerobienie danych wysyłanych po stronie serwera na strumienie bajtów (ich serializacja), a następnie deserializacja ich, aby uzyskać odpowiednią strukturę.

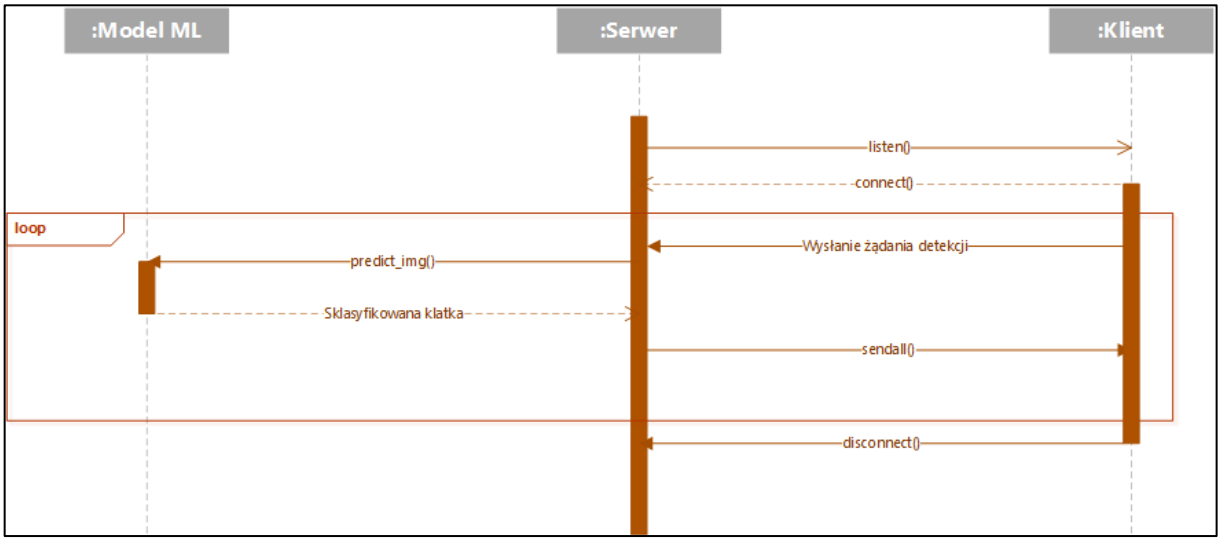
Tak więc serwer po zdobyciu odpowiednich danych z detektora tworzy obiekt zawierający te dane, konwertuje go do strumienia bajtów, a następnie tworzy z niego ramkę gotową do wysłania, która poza samymi bajtami z danymi zawiera na początku 8 bajtową (również skonwertowaną do strumienia bajtów) liczbę definiującą rozmiar przesyłanych danych. Tak więc, klient w trakcie odbierania danych od serwera, na początku dekoduje pierwsze 8 bajtów jako rozmiar wysyłanych danych, a następnie odbiera strumień bajtów w odpowiedniej ilości jako dane i je dekoduje na obiekt posiadający dane już w odpowiedniej formie.

Niżej zamieszczone zostały powyższe informacje w postaci odpowiedniego schematu:




*RYS 1.3 – SCHEMAT POŁĄCZENIA*

## 1.4. DIAGRAM UML SEKWENCJI




*RYS 1.4 – DIAGRAM SEKWENCJI DLA POJEDYNCZEGO KLIENTA*

# 2. ŚRODOWISKO PROJEKTU

## 2.1. OPIS ŚRODOWISKA

Ze względu na fakt, że procesy klienta i serwera działają na dwóch różnych systemach operacyjnych, konieczne jest odpowiednie podzielenie i dobór środowiska. Dlatego też klient działa na Windowsie 10 – systemie operacyjnym gospodarza, natomiast serwer jest procesem działającym na Linuxie w dystrybucji Ubuntu 20.04 LTS – wirtualizowanym systemem operacyjnym – tzw. gościem, który jest uruchamiany z użyciem oprogramowania wirtualizującego VMWare Workstation 16 Player.

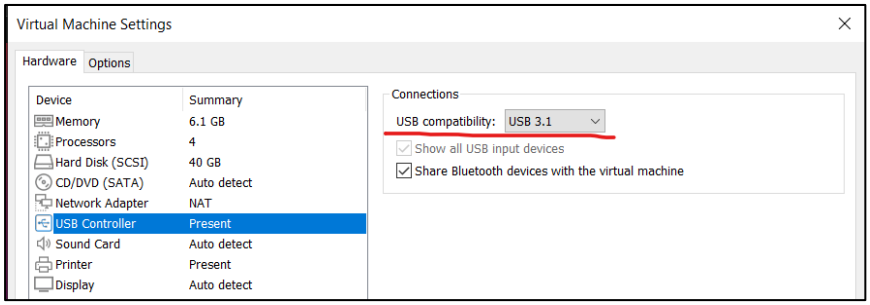
Podział ten został zastosowany w ten sposób, ponieważ proces serwera działa w czasie rzeczywistym, tak więc konieczne były modyfikacje jego systemu, a je wygodniej jest wykonywać na Linuxie.

Dystrybucja Ubuntu 20.04 LTS została wybrana ze względu na długi czas jej wsparcia, wygodność, a także dobrą współpracę z biblioteką TensorFlow wykorzystywaną w algorytmach machine learningowych w ramach tego projektu (opisanych w punkcie 4.2).

Jeśli chodzi o oprogramowanie wirtualizujące, dokonywany był wybór między VMWare, a VirtualBoxem, jednak ze względu na fakt, że VirtualBox bez wyłączenia w komputerze oprogramowania Hyper-V nie udostępnia gościowi wielu list rozkazów procesora, które zawiera procesor, a biblioteka TensorFlow wymaga listy instrukcji wzbogaconej o Advanced Vector Extensions (AVX), postawiliśmy na niezawodnego VMWare, przy którym ten problem nie występuje.

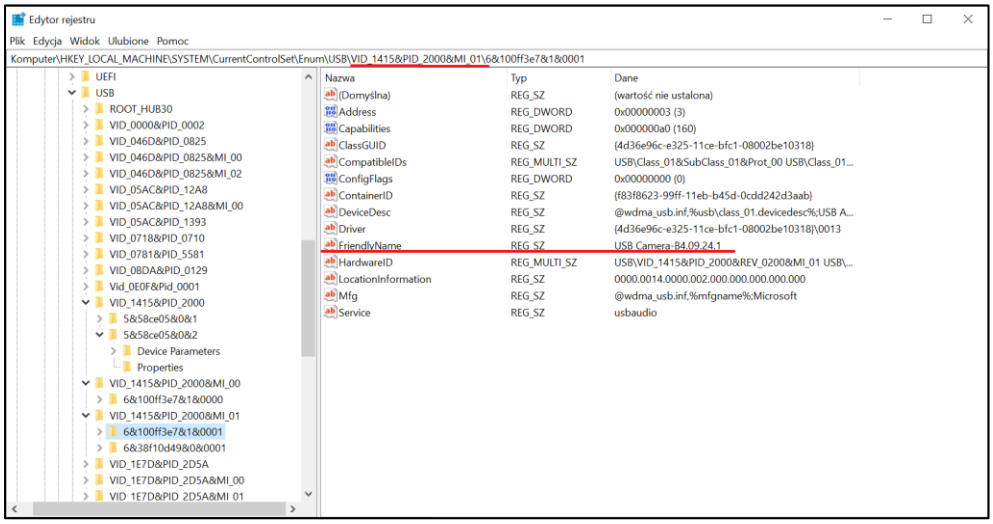
## 2.2. MODYFIKACJE ŚRODOWISKA

Pierwsza z modyfikacji wirtualnej maszyny wynika z konieczności udostępnienia kamery gospodarza w postaci urządzenia USB gościowi. Aby tego dokonać niezbędnym jest ustawienie kompatybilności USB z USB3.1 (inaczej zamiast obrazu rzeczywistego, pobierany będzie czarny obraz).




*RYS 2.1 – USTAWIENIE KOMPATYBILNOŚCI Z USB 3.1 W VMWARE*

Ze względu na fakt, że program koniecznie uruchamiać się musi przy starcie systemu, wymaganym jest ustawienie urządzenia USB tak, aby automatycznie przyłączał się do gościa wraz ze startem systemu. VMWare nie udostępnia takiej możliwości poprzez graficzny interfejs użytkownika, ale jest to możliwe przy użyciu pliku konfiguracyjnego wirtualnej maszyny (.vmx). Należy najpierw znaleźć przy użyciu edytora rejestru Windows tzw. vendor ID oraz product ID urządzenia USB, które chce się podłączyć.




*RYS 2.2 – VENDOR ID ORAZ PRODUCT ID W REJESTRZE WINDOWS*

Następnie wiedząc, że dla przykładowego urządzenia USB vendor_ID wynosi 0x1415, a product_id 0x2000, koniecznym jest umieszczenie odpowiedniej linii na końcu pliku konfiguracyjnego:

```
usb.autoConnect.device0 = "0x1415:0x2000"
```

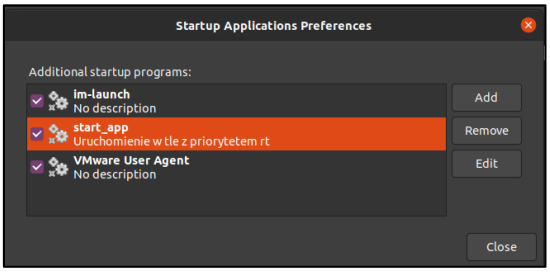
*RYS 2.3 – OSTATNIA LINIA PLIKU KONFIGURACYJNEGO .VMX*

Aby zapewnić, że aplikacja uruchomi się automatycznie ze startem systemu koniecznym jest stworzenie skryptu powłoki BASH, który zmienia katalog bieżący na katalog projektu, aktywuje wirtualne środkowisko Pythona, stworzone specjalnie dla procesu serwera, a następnie uruchamia skrypt w języku Python, będący skryptem serwera.

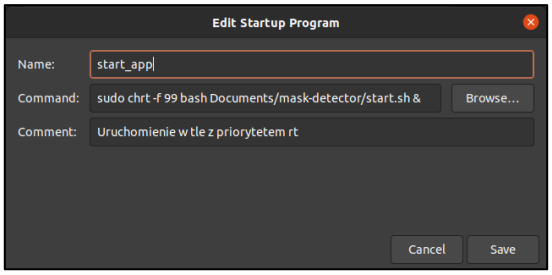
```
#!/bin/bash
cd /home/luki/Documents/mask-detector/
source /home/luki/Documents/mask-detector/venv/bin/activate
python3 /home/luki/Documents/mask-detector/mainServer.py
```

*RYS 2.4 – SKRYPT POWŁOKI BASH*

Następnie koniecznym jest użycie stworzonego skryptu powłoki BASH tak, aby był wykonywany przy starcie systemu w tle. Do tego została użyta wbudowana w Ubuntu, aplikacja Startup Applications Preferences, do której dodana została nasza aplikacja. Komenda programu uruchamianego jest wykonaniem (bash) skryptu (start.sh) w tle (&), ponieważ serwer działa na zasadzie pętli nieskończonej.




*RYS. 2.5 – WIDOK APLIKACJI W PROGRAMIE STARUP APPLICATIONS PREFERENCES*




RYS. 2.6 – PEŁNA APLIKACJA URUCHAMIANA WRAZ Z KOMENDĄ JĄ WYKONUJĄCĄ

Istotne jest jednak zwrócenie uwagi na początek komendy. Uruchomienie sudo chrt -f 99 zapewnia możliwość uruchomienia procesu z najwyższym możliwym priorytetem dla procesów czasu rzeczywistego, jednak opisem działania systemu w czasie rzeczywistym zajmiemy się w punkcie 3.

Za pierwszym uruchomieniem klienta, bezpośrednio po uruchomieniu maszyny gospodarza, a potem gościa, może wystąpić problem niemożliwości połączenia się z serwerem, co wynika z faktu, że pomimo tego, że sam gość widzi gospodarza w sieci, gospodarz nie widzi gościa (najłatwiej to sprawdzić przez obustronny ping na adresach IPv4). Wynika to z faktu niepoprawnego uruchomienia adaptera ethernetowego VMNet8 (dla techniki NAT w VMWare) po stronie gospodarza. Jednym z rozwiązań jest ponowne uruchomienie systemu gospodarza (co wiąże się również z ponownym uruchomieniem systemu gościa) lub wygodniej – zrestartowanie adaptera poprzez jego wyłączenie i włączenie z użyciem odpowiednich komend w terminalu gospodarza uruchomionym z prawami administratora.

```
Disable-NetAdapter -Name "VMware Network Adapter VMnet8" -Confirm:$false
Enable-NetAdapter -Name "VMware Network Adapter VMnet8" -Confirm:$false
```

*RYS 2.7 – KOMENDY PONOWNIE URUCHAMIAJĄCE ADAPTER VMNET8*

# 3. DZIAŁANIE W CZASIE RZECZYWISTYM

Przed stworzonym systemem (po stronie serwera) postawione zostało wymaganie działania w czasie rzeczywistym. Aby tego dokonać, najpierw podjęliśmy decyzję, wobec którego fragmentu całego systemu powinny być realizowane ograniczenia czasowe. Uznaliśmy więc, że, aby spełniać płynność pobieranego obrazu – istotnym będzie zapewnienie jak najmniejszej różnicy w czasie między kolejnymi pobraniami obrazu z kamery po stronie serwera. Wobec tego ustalono, że obraz ten powinien być pobierany co maksymalnie 300ms (ograniczenie te odnosi się również sytuacji, kiedy serwer jest bardzo obciążony przez sztucznie wygenerowane obciążenie).

Aby rozwiązać ten problem – zdecydowaliśmy się na odpowiednie nadanie naszemu procesowi (serwera) atrybutów planowania czasu-rzeczywistego. Po wykonaniu szeregu testów (opisanych w punkcie 5.3), podjęliśmy decyzję, że zastosujemy wobec naszego procesu politykę planowania FIFO (Round-Robin nie pasuje do zadania ze względu na to, że proces nie ma ograniczenia od góry na czas) z najwyższym możliwym priorytetem dla procesów czasu rzeczywistego, co można sprawdzić przy użyciu odpowiednich komend.

```
luki@ubuntu:~$ chrt --max
SCHED_OTHER min/max priority    : 0/0
SCHED_FIFO min/max priority     : 1/99
SCHED_RR min/max priority       : 1/99
SCHED_BATCH min/max priority    : 0/0
SCHED_IDLE min/max priority     : 0/0
```

*RYS 3.1 – SPRAWDZENIE MAKSYMALNYCH MOŻLIWYCH PRIORYTETÓW DLA RÓŻNYCH POLITYK SZEREGOWANIA*

Tak więc, w celu usprawnienia działania naszego systemu, proces uruchamiany jest z początkiem komendy *sudo chrt -f 99*.

Aby móc zmieniać politykę szeregowania i priorytet procesowi, gdy jest uruchamiane automatycznie przy starcie systemu, koniecznym było zadbanie o to, aby użytkownik mógł wykonać komendę tą bez konieczności podania hasła. W tym celu posłużyć się trzeba komendą sudo visudo i na końcu pliku sudoers.tmp dodać linię umożliwiającą użytkownikowi wykonywanie sudo chrt bez konieczności podawania hasła.

```
#includedir /etc/sudoers.d
%sudo ALL = (ALL) NOPASSWD: /usr/bin/nice, /usr/bin/renice
%sudo ALL = (ALL) NOPASSWD: /usr/bin/chrt
```

**RYS. 3.2 – KONIEC PLIKU SUDOERS.TMP Z LINIĄ ZAPEWNIAJĄCĄ WYKONANIE SUDO CHRT BEZ KONIECZNOŚCI PODANIA HASŁA**

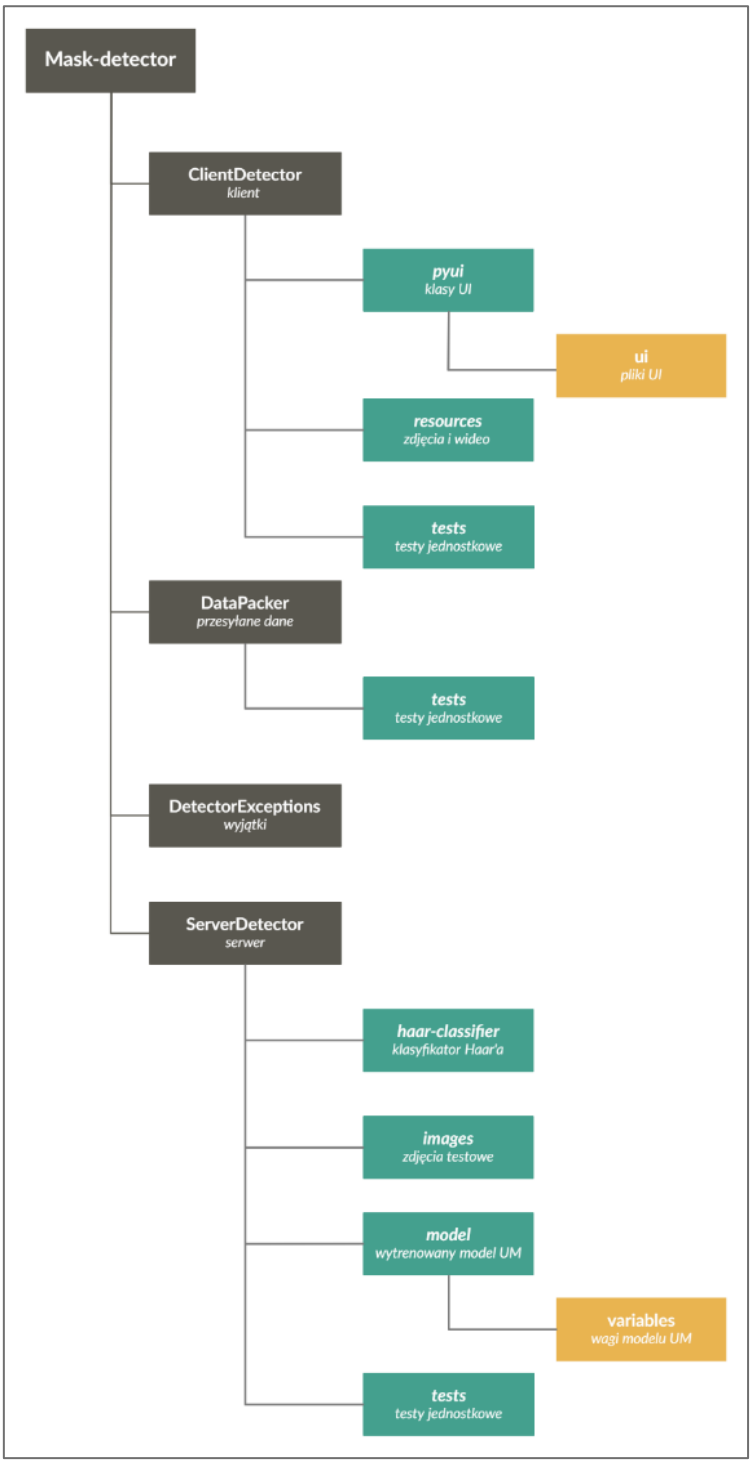
Po wykonaniu wszystkich zmian i ponownym uruchomieniu systemu, przy użyciu komendy top można sprawdzić zarówno priorytet procesu serwera jak i to, czy zajmuje czas procesora.

Dodatkowo, przy opisie pracy w czasie rzeczywistym, niezbędnym jest wspomnienie używanej przez nas biblioteki OpenCV, która, przede wszystkim sprzętowo (a nie systemowo), gwarantuje duże wsparcie dla działania w czasie rzeczywistym poprzez zastosowanie przyspieszenia GPU dla operacji w czasie rzeczywistym, a także korzystanie ze zbioru instrukcji procesora MMX oraz SSE.

# 4. IMPLEMENTACJA

W niniejszym rozdziale opisane zostaną moduły języka jakie zostały użyte przy tworzeniu najważniejszych fragmentów systemu oraz najważniejsze funkcje jakie zostały stworzone. Dodatkowo w punkcie 4.2. przedstawiony zostanie pełny opis części systemu związanej z uczeniem maszynowym.

## 4.1. STRUKTURA KATALOGÓW W PROJEKCIE



## 4.2. TRENOWANIE MODELU, DETEKCJA TWARZY I MASKI

### 4.2.1. ZBIÓR DANYCH

Wykorzystany został zbiór danych zawierający łącznie 1376 zdjęć (690 zdjęć osób z założonymi maseczkami oraz 686 zdjęć osób bez założonych maseczek).

*RYS 4.1 – PRZYKŁADOWE ZDJĘCIE Z KATALOGU WITH_MASK*

*RYS 4.2 – PRZYKŁADOWE ZDJĘCIE Z KATALOGU WITHOUT_MASK*

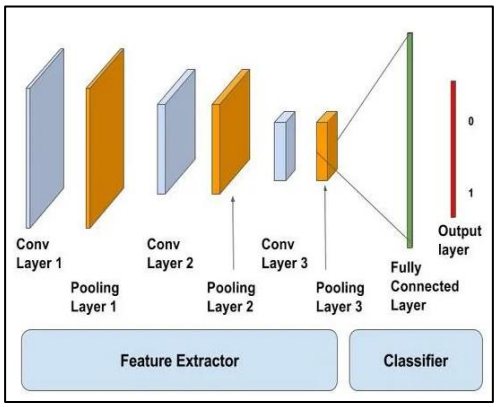
Zdjęcia znajdą się w katalogu *data* w podkatalogach *with_mask* oraz *without_mask*. Nazwy podkatalogów są jednocześnie etykietami klas detekowanych.

Klasa Dataset reprezentująca zbiór danych znajduje się w pliku dataset.py. Zdjęcia wczytywane są w metodzie create_dataset() za pomocą keras.preprocessing. Tworzony jest zarówno zbiór treningowy jak i zbiór walidacyjny. Zbiór treningowy składa się z 80% zdjęć w całym zbiorze danych – 1101 zdjęć, a zbiór walidacyjny składa się z 20% - 275 zdjęć. Zdjęciom przy wczytywaniu ujednolicany jest rozmiar (400 x 400 pikseli), by trenowanie odbywało się na macierzach o takich samych wymiarach. Ponadto zbiór dzielony jest na 32 batch'e.

### 4.2.2. MODEL UCZENIA MASZYNOWEGO

Klasa *FaceMaskDetector* reprezentuje klasyfikator rozpoznający czy człowiek ma założoną maskę czy nie i znajduje się w pliku *face_mask_detector.py*. Model tworzony jest w metodzie *create()* i składa się z:

- Trzech warstw konwolucyjnych, których zadaniem jest wyodrębnienie charakterystycznych cech, które produkują duże wartości na wyjściu po przejściu przez funkcję aktywacji neuronów.
- Trzech warstw max-pooling, które zmniejszają wymiary warstw (wysokość i szerokość) zmniejszając tym samym liczbę parametrów, co zmniejsza liczbę obliczeń i zapobiega przeuczeniu.
- Dwóch warstw w pełni połączonych, przy czym ostatnia z nich daje wynik klasyfikacji.
- Dodatkowo model przed warstwami w pełni połączonymi zawiera warstwę zamieniającą macierz o wymiarach 50x50x64 w jeden wektor o długości 160000 elementów.




*RYS 4.3 – MODEL UCZENIA MASZYNOWEGO*

### 4.2.3. ZAPOBIEGANIE PRZEUCZENIU

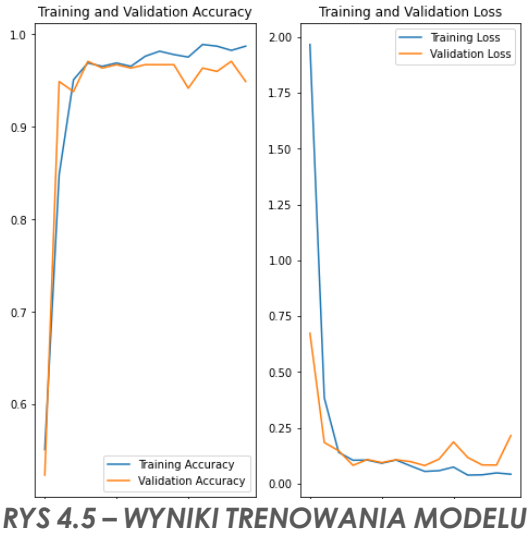
Pierwszym sposobem na zapobiegnięcie przeuczeniu jest powiększenie zbioru danych, które zapewnia większą ilość danych trenujących. W tym podejściu generowane są dodatkowe próbki w zbiorze trenującym z istniejących już danych – próbki są losowo transformowane np.: przez zmienianie orientacji lub przez przybliżenie obrazów. Warstwa, która powiększa zbiór danych jest pierwszą warstwą w splotowej sieci neuronowej.

**RYS 4.4 – PRZYKŁAD WYGENEROWANYCH DODATKOWYCH PRÓBEK DANYCH**

Inna techniką zapobiegania przeuczeniu, która została zastosowana w modelu, jest formą regularyzacji– dropout. W tym podejściu losowe neurony w warstwie są odrzucane, przez ustawienie ich funkcji aktywacji na zero. W modelu losowo odrzucane jest 20% neuronów w warstwie.

### 4.2.4. TRENOWANIE MODELU

Trening modelu uruchamiany jest w metodzie *train_model()*. Model został wytrenowany przez 15 epok. Po jego wytrenowaniu, skuteczności modelu zostały przedstawione na wykresach poniżej.




***RYS 4.5 – WYNIKI TRENOWANIA MODELU***

Model ten przy osiągnięciu ok. 96% skuteczności i braku znaczących oznak przeuczenia (zarówno skuteczność jak i błąd zbiorów treningowego i walidacyjnego są podobne) uznany może zostać za odpowiedni dla zadanego problemu, gdyż osiąga bardzo dużą skuteczność.

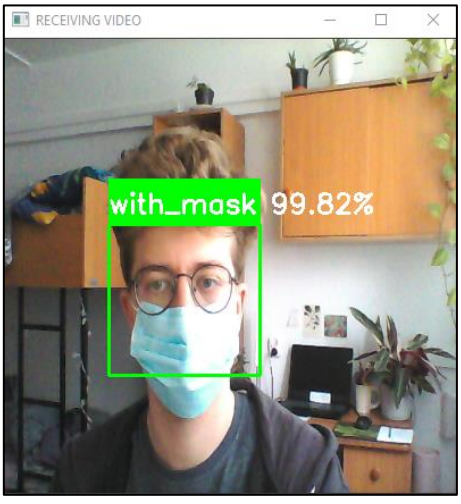
Model można zapisać za pomocą metody *save_model()*, co umożliwia przeprowadzanie klasyfikacji po wczytaniu wytrenowanego wcześniej modelu, co zostało wykorzystane w naszym systemie.

### 4.2.5. PREDYKCJA

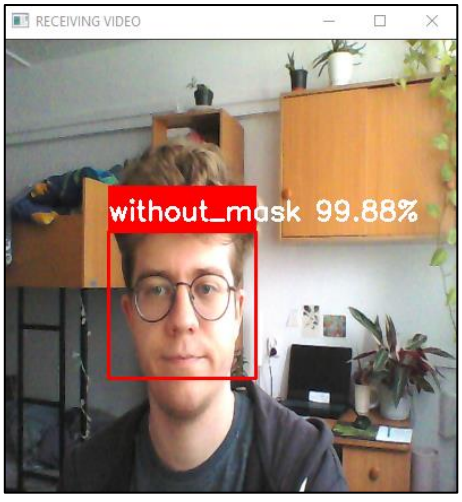
Predykcja z użyciem wytrenowanego modelu jest możliwa za pomocą metody predict_img(). Po wywołaniu metody predict_img() wykrywane są wszystkie twarze na zdjęciu, za pomocą metody detect_faces(). Twarze wykrywane są za pomocą algorytmu kaskady Haara. Gotowy model znajduje się w pliku /haar-classifier/haarcascade_frontalface_default.xml i jest on wczytywany przez obiekt klasy FaceMaskDetector z tego pliku.

Algorytm kaskady Haara charakteryzuje się tym, że działa on bardzo szybko, a jego model jest bardzo lekki. Ceną, jaką płaci się za szybkość działania algorytmu jest jego skuteczność – nie radzi on sobie z twarzami widocznymi z profilu lub z twarzami, które są rozmazane. W tym projekcie problem ten nie wydaje się istotny – z założenia użytkownik powinien patrzeć prosto w kamerę. Istotna jest natomiast szybkość działania.

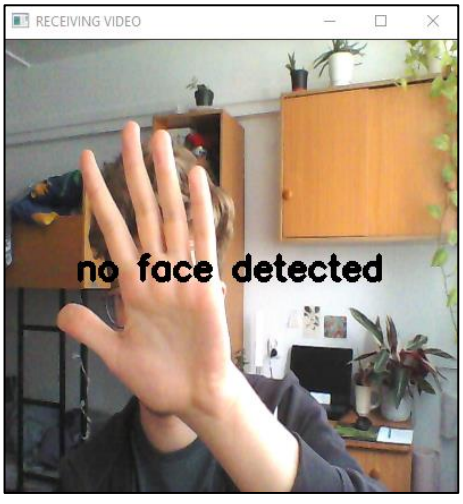
Po znalezieniu twarzy na obrazie, następuje klasyfikacja twarzy przez model. Model na wejściu otrzymuje tylko obszar obrazu zawierający twarz, żeby zapewnić większą skuteczność klasyfikacji. Z wyjść modelu odczytywana jest klasa oraz pewność z jaką twarz należy do danej klasy, wyrażana w procentach. Następnie na obrazie umieszczana jest informacja odczytywana z wyjść modelu. Jeśli żadna twarz nie zostanie wykryta na obrazie, również zostanie umieszczony na obrazie stosowny komunikat o tym informujący.




RYS 4.6 – PRZYKŁADOWY OBRAZ, GDY MASKA ZOSTAŁA WYKRYTA




RYS 4.7 – PRZYKŁADOWY OBRAZ, GDY MASKA NIE ZOSTAŁA WYKRYTA




RYS 4.8 – PRZYKŁADOWY OBRAZ, GDY ŻADNEJ TWARZY NIE WYKRYTO

## 4.3. PROCES SERWERA

Moduły języka Python użyte w ramach procesu serwera:

- **socket** – do implementacji połączenia sieciowego po stronie serwera,
- **threading** – zapewnia implementację wielowątkowości w ramach procesu, niezbędny w momencie, gdy przyłączy się klient – tworzony jest wtedy nowy wątek, w którym to w pętli przesyłane są dane od serwera do klienta,
- **cv2** – biblioteka zapewniająca możliwość pobrania obrazu z kamery, wspiera działanie w czasie rzeczywistym,
- **pickle** oraz **struct** – biblioteki umożliwiające odpowiednio konwersję obiektów do strumienia bajtów oraz dodawanie ramki do wysyłanego strumienia,
- **DataPacker** – klasa stworzona do przechowywania wszystkich danych, które będą przesyłane między serwerem a klientem, obiekty tej klasy zawierają obraz przesyłany, informację o klasie wykrywanego obrazu, procent pewności oraz aktualne opóźnienie na serwerze i są bezpośrednio przekształcane w strumień bajtów,
- **FaceMaskDetector** – zaimplementowany w poprzednim podpunkcie detektor twarzy i maski na podawanym obrazie.

Serwer posiada odpowiednią obsługę sytuacji wyjątkowych związanych m.in. z nagłym zerwaniem połączenia ze strony klienta czy próbie podłączenia drugiego klienta do serwera w momencie, w którym obsługuje on już pierwszego (niezaakceptowanie go).

## 4.4. PROCES KLIENTA

Moduły języka Python użyte w ramach procesu klienta:

- **socket –** do implementacji połączenia sieciowego po stronie klienta,
- **cv2 –** do wyświetlania w oddzielnym oknie obrazu przychodzącego od serwera,
- **pickle** oraz **struct** – biblioteki umożliwiające odpowiednio konwersję strumienia bajtów do obiektu oraz wyczytanie długości danych z otrzymywanego strumienia,
- **ClientGui –** wskazanie na obiekt uruchamiający GUI związany z procesem klienta,
- **PyQT5 -** nakładka na bibliotekę Qt umożliwiająca tworzenie interfejsu graficznego dla programów komputerowych pisanych w języku Python.

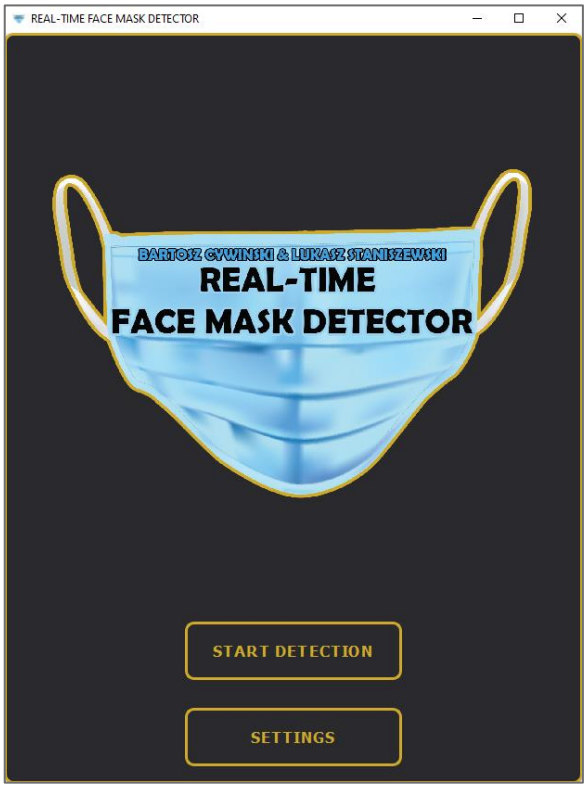
## 4.5. GRAFICZNY INTERFEJS UŻYTKOWNIKA

### 4.5.1. TECHNOLOGIE

Do stworzenia Graficznego Interfejsu Użytkownika wykorzystaliśmy bibliotekę PyQt5. Ponadto do łatwiejszego i wydajniejszego projektowania interfejsu wykorzystaliśmy QtDesigner'a.

### 4.5.2. OKNO STARTOWE

W oknie startowym użytkownik może zacząć proces wykrywania maski przez system za pomocą przycisku *START DETECTION*. Klient musi być jednak podłączony do właściwego serwera, inaczej aplikacja nie pozwoli rozpocząć procesu detekcji, a na ekranie pojawi się komunikat przypominający o ustawieniu połączenia. Ustawić i przetestować podłączenie z serwerem można naciskając przycisk *SETTINGS*.

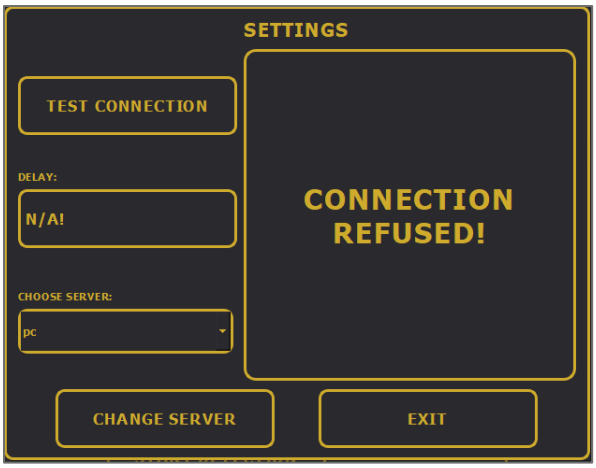



RYS 4.9 OKNO STARTOWE

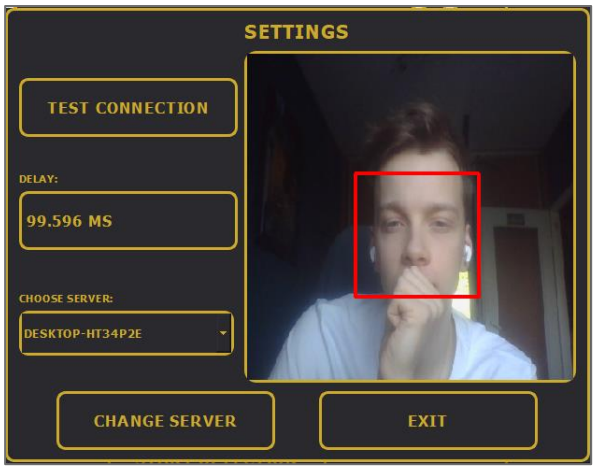

RYS 4.10 OKNO STARTOWE BEZ WCZEŚNIEJSZEGO USTAWIENIA POŁĄCZENIA Z SERWEREM

### 4.5.3. OKNO USTAWIEŃ

W polu *CHOOSE SERVER* klient może wybrać nazwę serwera z już istniejących, do jakiego chce się połączyć lub wprowadzić nową nazwę. Po wybraniu nazwy serwera, klient może przetestować połączenie z serwerem za pomocą przycisku *TEST CONNECTION*. Jeśli połączenie do wybranego serwera nie będzie możliwe, pojawi się stosowny komunikat. Jeśli jednak klientowi uda się połączyć z wybranym serwerem, pojawi się obraz z kamery użytkownika, a w polu *DELAY* pojawi się opóźnienie między przesłaniem obrazu do serwera, a odebraniem obrazu przez serwer, należy jednak pamiętać o zatwierdzeniu zmiany serwera klikając przycisk *CHANGE SERVER*.



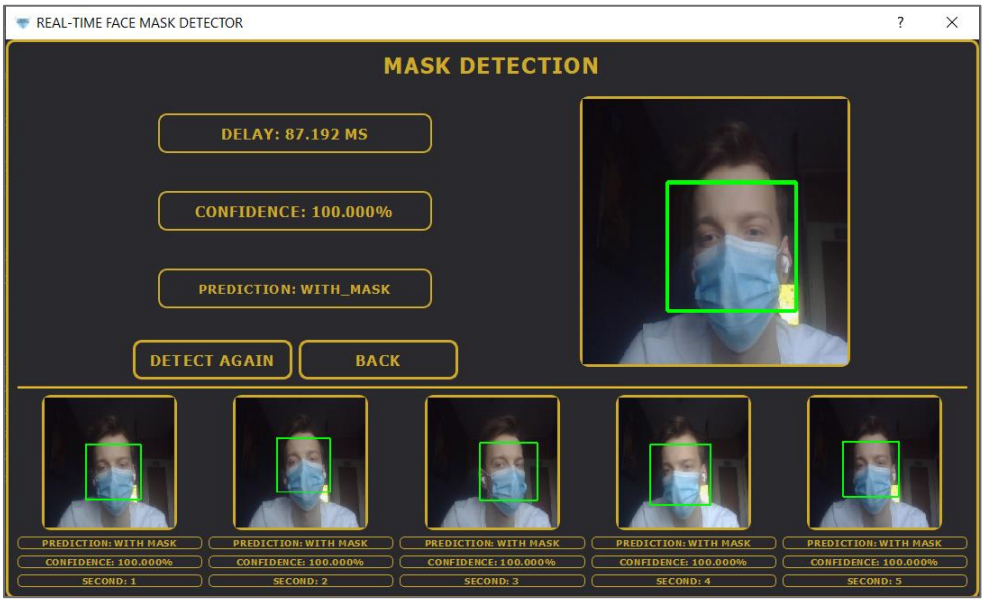

RYS 4.11 OKNO USTAWIEŃ PO NIEUDANYM NAWIĄZANIU POŁĄCZENIA Z SERWEREM



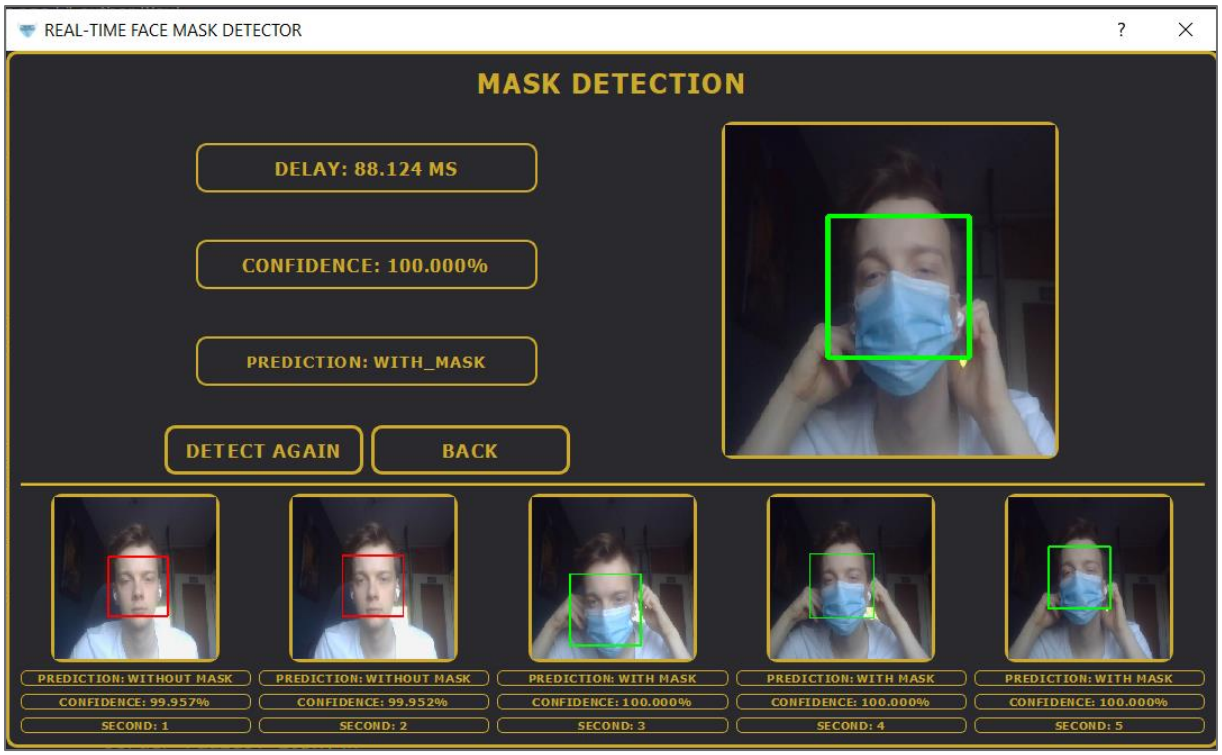

RYS 4.12 OKNO USTAWIEŃ PO UDANYM NAWIĄZANIU POŁĄCZENIA Z SERWEREM

### 4.5.4. OKNO WYKRYWANIA TWARZY

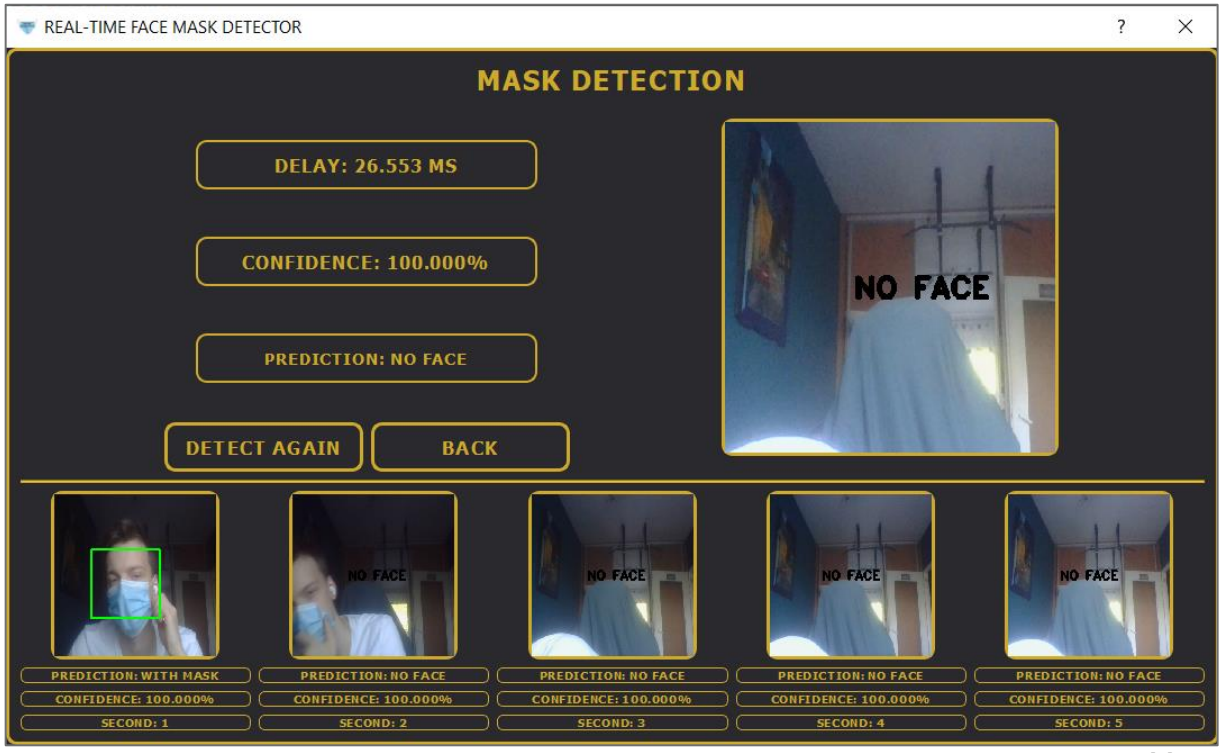
Po ustawieniu poprawnego połączenia z serwerem, naciskając przycisk *START DETECTION* w oknie startowym, rozpocznie się proces wykrywania twarzy.



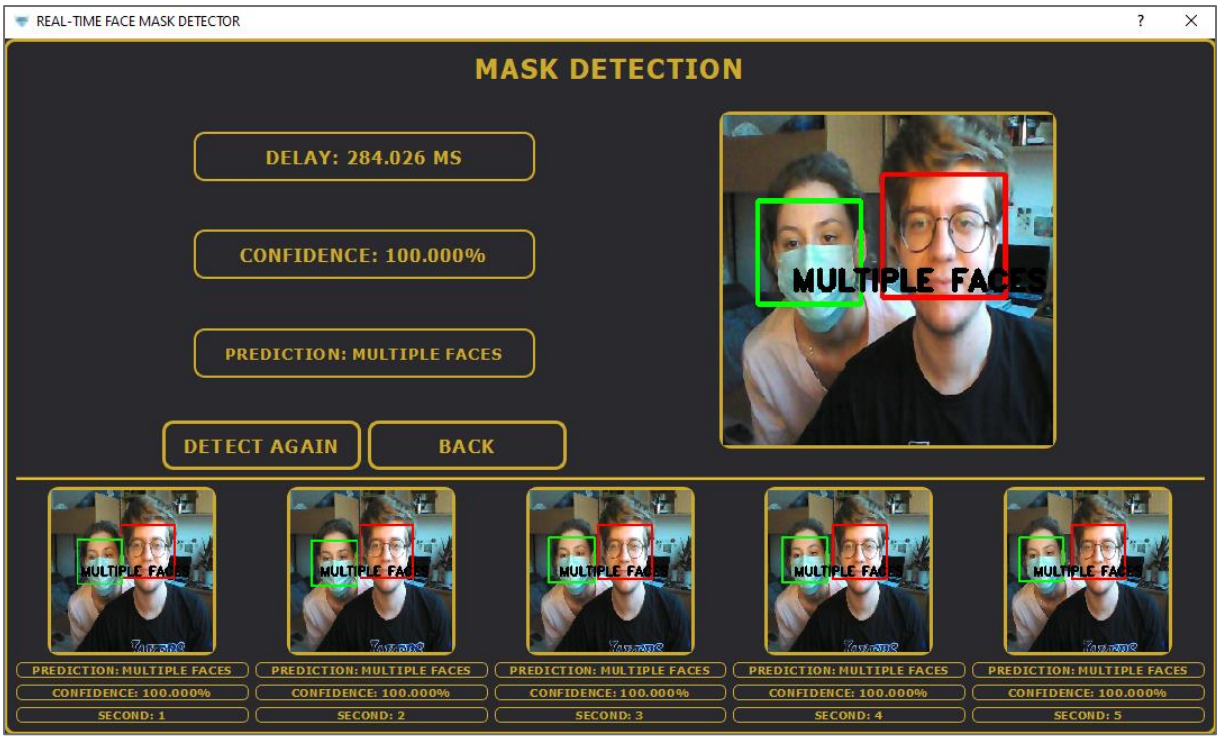

RYS. 4.13 OKNO WYKRYWANIA MASKI, GDZIE PRZEZ CAŁY CZAS TRWANIA PROCESU WYKRYWANIA SYSTEM STWIERDZAŁ OBECNOŚĆ MASKI NA TWARZY




**RYS. 4.14 OKNO WYKRYWANIA MASKI, GDZIE W PIERWSZYCH DWÓCH SEKUNDACH SYSTEM STWIERDZIŁ BRAK MASKI NA TWARZY, A W KOLEJNYCH TRZECH SEKUNDACH STWIERDZIŁ OBECNOŚĆ MASKI NA TWARZY**




**RYS. 4.15 OKNO WYKRYWANIA MASKI, GDZIE W PIERWSZEJ SEKUNDZIE SYSTEM STWIERDZIŁ OBECNOŚĆ MASKI NA TWARZY, A W KOLEJNYCH SEKUNDACH NIE WYKRYŁ TWARZY NA OBRAZIE**




**RYS. 4.15 OKNO WYKRYWANIA MASKI, GDZIE PRZEZ CAŁY CZAS TRWANIA PROCESU WYKRYWANIA SYSTEM STWIERDZAŁ OBECNOŚĆ WIĘCEJ NIŻ JEDNEJ TWARZY NA OBRAZIE**

W prawej górnej części ekranu można obserwować obraz z kamery klienta. Na obrazie, jeśli twarz zostanie wykryta, zaobserwować można twarz w ramkach. Czerwona ramka oznacza, że na obrazie została wykryta twarz bez maseczki. Zielona ramka natomiast oznacza, że została wykryta twarz z założoną maseczką. Są też możliwe przypadki, kiedy nie zostanie wykrywa żadna twarz na przesyłanym obrazie z kamery – wtedy na obrazie można zaobserwować komunikat *NO FACE*. Ostatnią możliwością jest wykrycie więcej niż jednej twarzy na obrazie – w takiej sytuacji klasyfikator nie byłby w stanie przedstawić na wyjściu jednoznacznego wyniku, czy maseczka została wykryta czy nie. Dlatego w przypadku wykrycia więcej niż jednej twarzy, na obrazie można zaobserwować komunikat *MULTIPLE FACES*. W polu *DELAY* na bieżąco można zobaczyć jakie jest obecnie opóźnienie na serwerze, wyrażane w milisekundach, między dwiema ostatnio wykonanymi klatkami obrazu z kamery. Pole *CONFIDENCE* zawiera informację o pewności klasyfikatora co do decyzji, wyrażaną w procentach. Pole *PREDICTION* natomiast przedstawia predykcję – wyjście klasyfikatora.

Proces wykrywania maski trwa 5 sekund. W każdej sekundzie zapisywana jest jedna klatka z obrazu z kamery, która następnie pojawia się na dole ekranu. Każda klatka przechowywana jest razem z predykcją klasyfikatora, pewnością klasyfikatora i sekundą, w której dana klatka została pobrana. Po upłynięciu czasu 5 sekund proces wykrywania maski się kończy, a na ekranie pojawia się okno z ostateczną decyzją.

Po ukończeniu procesu wykrywania twarzy można również powtórzyć proces wykrywania za pomocą przycisku *DETECT AGAIN*, który zaczyna proces wykrywania twarzy od nowa. Ponadto, również po ukończeniu wykrywania twarzy, można za pomocą przycisku *BACK* wrócić do okna startowego. Nie jest możliwe ani wrócenie do okna startowego, ani wystartowania procesu detekcji od nowa w trakcie trwającej już detekcji.

### 4.5.5. OKNO WYNIKU WYKRYWANIA TWARZY

Po zakończonym wykrywaniu twarzy pojawia się okno, na którym w górnej części ekranu przedstawiony jest wynik wykrywania twarzy. Poza samym wynikiem, w oknie uruchamiane jest również wideo o tematyce zachowania bezpieczeństwa w czasie pandemii. Ponadto, za pomocą przycisku *BUY MASK* użytkownik zostanie przekierowany na stronę, na której można kupić maseczki COVID'owe, a za pomocą przycisku *READ RESTRICTIONS* użytkownik zostaje przekierowany na stronę z obecnymi obostrzeniami związanymi z pandemią w Polsce.


RYS. 4.16 OKNO WYNIKU DETEKCJI ZE STWIERDZENIEM WYKRYCIA MASKI NA TWARZY

# 5. TESTOWANIE

W ramach tego rozdziału zostaną opisane testy stworzonego systemu.

## 5.1. TESTY JEDNOSTKOWE

- Dla klasy *FaceMaskDetector* przeprowadzono serię testów jednostkowych sprawdzających poprawność klasyfikacji zaimplementowanego modelu uczenia maszynowego. Testy polegały na sprawdzaniu poprawności wykrycia masek na twarzach osób na zdjęciach.
- Dla klasy *ServerDetector* przeprowadzono testy sprawdzające poprawność tworzenia serwera na zadanym adresie i porcie.
- Dla klasy *DataPacker* przeprowadzono testy sprawdzające poprawność przesyłania danych. Sprawdzono również poprawność rzucania wyjątków w przypadku próby przesłania niepoprawnych danych, takich jak ujemna pewność predykcji modelu uczenia maszynowego.
- Dla klasy *HostClient* przeprowadzono testy sprawdzające poprawność podłączenia się klienta o zadanym adresie i porcie.

```
============================================= test session starts =============================================
platform win32 -- Python 3.9.5, pytest-6.2.3, py-1.10.0, pluggy-0.13.1
rootdir: C:\Users\01149762\Documents\Python_projects\VSCProjects\mask-detector
collected 15 items

ClientDetector\tests\test_HostClient.py .                                                          [  6%]
DataPacker\tests\test_DataPacker.py ..                                                             [ 20%]
ServerDetector\tests\test_ServerDetector.py .                                                      [ 26%]
ServerDetector\tests\test_mask_detector.py ...........                                             [100%]

============================================== 15 passed in 6.95s ==============================================
```

RYS. 5.1. – OSTATECZNY WYNIK PRZEPROWADZONYCH TESTÓW JEDNOSTKOWYCH

## 5.2. TESTY FUNKCJONALNE

Przeprowadziliśmy szereg testów sprawdzając poprawność działania zaimplementowanego przez nas systemu. Testy dotyczyły m.in:

- Poprawności wykrywania maski na twarzy.
- Poprawności zachowania systemu przy nieskonfigurowanym połączeniu z serwerem.
- Poprawności zachowania systemu przy próbie ponownego uruchomienia wykrywania maski podczas działającego już innego procesu wykrywania.
- Poprawności zachowania systemu przy próbie powrócenia do okna startowego podczas działającego już procesu wykrywania.
- Poprawności zachowania systemu przy próbie ponownego uruchomienia wykrywania maski podczas wyświetlania wyniku poprzedniego wykrywania.
- Poprawności zachowania systemu przy próbie powrócenia do okna startowego podczas wyświetlania wyniku wykrywania.
- Poprawności połączenia klienta z serwerem na różnych komputerach.

## 5.3. TESTY DLA SYSTEMU CZASU RZECZYWISTEGO

Aby dobrze sprawdzić, jak działa nasz system w warunkach czasu rzeczywistego, zaimplementowano w ramach klienta możliwość logowania pracy systemu czasu rzeczywistego, a także w przesyłanych ramkach między klientem i serwerem, serwer przesyła (mierzone przez samego siebie) opóźnienie między pobraniem dwóch ostatnich ramek z kamery.

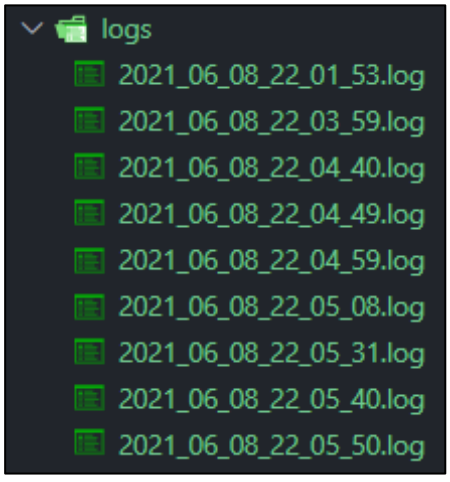
W przypadku ustawienia, przy uruchamianiu procesu klienta, opcji –logging, klient dba o to, aby szereg odebranych informacji z ramek umieszczać w folderze logs/ w odpowiednim pliku i na podstawie odebranych ramek liczyć średnie opóźnienie po stronie serwera w milisekundach.

```
python .\mainClient.py --logging
```

RYS. 5.2. – KOMENDA PO STRONIE KLIENTA POZWALAJĄCA NA LOGOWANIE PRACY SERWERA

```
38  2021/06/08||22:05:08||DECISION: NO FACE||DELAY: 96.463MS
39  2021/06/08||22:05:08||DECISION: NO FACE||DELAY: 42.236MS
40  2021/06/08||22:05:08||DECISION: without_mask||DELAY: 44.646MS
41  2021/06/08||22:05:08||DECISION: without_mask||DELAY: 99.910MS
42  2021/06/08||22:05:08||DECISION: without_mask||DELAY: 90.980MS
43  2021/06/08||22:05:08||DECISION: without_mask||DELAY: 94.505MS
44  2021/06/08||22:05:08||DECISION: without_mask||DELAY: 94.026MS
45  ~~~~~~~~~~~~~~~~~~~
46  SUMMARY DELAY: 91.24022727272727MS
```

RYS. 5.4. – FRAGMENT PLIKU .LOG




RYS. 5.3. – FOLDER Z LOGAMI

W ten sposób dokonywaliśmy pomiarów działania systemu. Nasze pomiary przeprowadziliśmy na 4 przypadkach działania programu serwera:

- przed kamerą nie występuje osoba (przez co nie jest uruchamiany zaawansowany algorytm sztucznej inteligencji, zużywający najwięcej czasu procesora), natomiast sam system, na którym znajduje się program, nie posiada sztucznie wygenerowanego obciążenia,
- przed kamerą występuje osoba, natomiast sam system nie jest dodatkowo obciążony,
- przed kamerą nie występuje osoba, a serwer posiada sztucznie wygenerowane obciążenie z użyciem metody `stress –cpu 10 –io 10 –vm 10`, powodującą sumaryczne obciążenie 30 pracowników (workers) na system,
- przed kamerą występuje osoba, sam system jest dodatkowo obciążony jak w poprzednim punkcie.

Pierwszą opcją było uruchomienie procesu serwera w sposób zwykły, bez nadawania priorytetów i planu szeregowania, otrzymano następujące wyniki:

| DZIAŁANIE PROGRAMU PO STRONIE SERWERA | ŚREDNIA RÓŻNICA CZASU MIĘDZY DWIEMA KLATKAMI PO STRONIE SERWERA [MS] |
|---|---|
| BRAK OSOBY, BRAK OBCIĄŻENIA | 48.1 |
| WYSTĘPUJE OSOBA, BRAK OBCIĄZENIA | 101.24 |
| BRAK OSOBY, SZTUCZNE OBCIĄŻENIE | 173.92 |
| WYSTĘPUJE OSOBA, SZTUCZNE OBCIĄŻENIE | 781.77 |

**RYS. 5.4. – POMIARY CZASÓW PRZY ZWYKŁYM URUCHOMIENIU PROCESU SERWERA**

Jak widać, serwer w większości spełnia ograniczenie czasowe przez nas wcześniej wyspecyfikowane, jednak dla dużego obciążenia radzi on sobie fatalnie, dlatego też zwykłe uruchomienie programu nie wystarcza.

Następnie zdecydowano się uruchomić program serwera w przestrzeni procesów użytkownika, ale z parametrem **nice** równym **-20**, czyli z największym możliwym priorytetem procesu w przestrzeni procesów użytkownika, otrzymano następujące wyniki:

| DZIAŁANIE PROGRAMU PO STRONIE SERWERA | ŚREDNIA RÓŻNICA CZASU MIĘDZY DWIEMA KLATKAMI PO STRONIE SERWERA [MS] |
|---|---|
| BRAK OSOBY, BRAK OBCIĄŻENIA | 13.58 |
| WYSTĘPUJE OSOBA, BRAK OBCIĄZENIA | 102.2 |
| BRAK OSOBY, SZTUCZNE OBCIĄŻENIE | 33.99 |
| WYSTĘPUJE OSOBA, SZTUCZNE OBCIĄŻENIE | 129.55 |

**RYS. 5.5. – POMIARY CZASÓW PRZY URUCHOMIENIU PROCESU SERWERA Z PARAMETREM NICE**

Jak widać, sytuacja poprawiła się diametralnie i serwer za każdym razem spełnia nasze ograniczenia czasowe. Problem może jednak nastąpić, jeśli obciążenie na systemie pochodziłoby od procesów o priorytecie czasu rzeczywistego – przez wyższość tego priorytetu nad priorytetem procesów użytkownika, sytuacja byłaby analogiczna jak w poprzednim teście.

Ostatecznym testem, dzięki któremu udało nam się podjąć decyzję o szeregowaniu, było uruchomienie programu z użyciem komendy *sudo chrt -f 99*, czyli z największym możliwym priorytetem procesu w przestrzeni procesów czasu rzeczywistego i z użyciem polityki szeregowania FIFO, otrzymano następujące wyniki:

| DZIAŁANIE PROGRAMU PO STRONIE SERWERA | ŚREDNIA RÓŻNICA CZASU MIĘDZY DWIEMA KLATKAMI PO STRONIE SERWERA [MS] |
|---|---|
| BRAK OSOBY, BRAK OBCIĄŻENIA | 27.46 |
| WYSTĘPUJE OSOBA, BRAK OBCIĄZENIA | 103.73 |
| BRAK OSOBY, SZTUCZNE OBCIĄŻENIE | 30.66 |
| WYSTĘPUJE OSOBA, SZTUCZNE OBCIĄŻENIE | 113.58 |

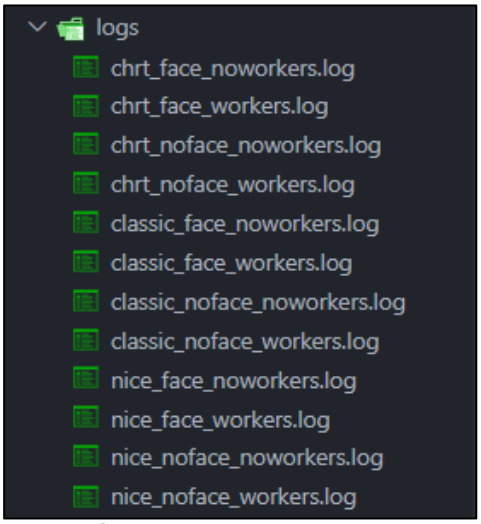
**RYS. 5.6. – POMIARY CZASÓW PRZY URUCHOMIENIU PROCESU SERWERA Z NAJWYŻSZYM MOŻLIWYM PRIORYTETEM Z UŻYCIEM POLITYKI SZEREGOWANIA FIFO**

Jak możemy zauważyć, nie dość że tak uruchomiony program radzi sobie lepiej w przypadku większego obciążenia, to nie wystąpi tu problem z obciążeniem przez procesy czasu rzeczywistego, ponieważ nasz proces działa na najwyższym priorytecie:

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
17735 root      rt   0 3088356 631336 183016 S 103,3  11,1   2:20.33 python3
```

**RYS. 5.7. – PRIORYTET PROCESU SERWERA PRZY URUCHOMIENIU Z PRIORYTETEM CZASU RZECZYWISTEGO**

Wyniki tych testów można również znaleźć w folderze logs:




**RYS. 5.8. – WYNIKI TESTÓW W CZASIE RZECZYWISTYM W FOLDERZE LOGS**