Indeksy: 272816, 272822
Kurs: Systemy Operacyjne 2 [P]
Prowadzący: Mgr Inż. Damian Raczkowski

Serwer Czatu

Spis treści:

1.Wstęp

Celem projektu jest napisanie programu, który będzie świadczył usługi wielowątkowego serwera czatu. Serwer głównie odpowiada za sprawdzanie nadchodzących połączeń, odbieranie wiadomości tekstowych i rozsyłaniu ich do klientów oraz przeprowadzanie procesu rozłączenia klienta. Dodatkowo w naszym programie zaimplementowaliśmy kilka unikalnych funkcji tj. historia czatu rozsyłana nowo podłączonym użytkownikom, serwer pamięta również nazwy użytkowników oraz czas przesłania wiadomości i przypisuje te informacje do wiadomości. Dodatkowo system pozwala na wydawanie specjalnych poleceń tj. `Connect, Disconnect` oraz `cls`, które służą do połączenia się z czatem, rozłączenia się z czatem oraz wyczyszczenia historii czatu.

2.Sposób kompilacji

Plik uruchom_skrypt.sh należy umieścić w tym samym katalogu co pliki .py, należy nadać mu prawo do wykonywania za pomocą komendy chmod +x uruchom_skrypt.sh, następnie należy go uruchomić za pomocą komendy bash uruchom_skrypt.sh. Zostanom także utworzone pliki wykonywalne o takich samych nazwach co pliki .py i należy je uruchomić poleceniami ./Server i ./Client.

3.Działanie programu

Server:

HEADER, PORT, FORMAT, DISCONNECT_MESSAGE, ADDR: parametry konfiguracyjne serwera
clients = [] – lista klientów przechowuje aktywne połączenia (sockety)
names = {} – słownik nazw – przechowuje nazwy użytkowników
chat_history = [] – lista zawierająca historię czatu

Zabezpieczone sekcje z lock (dotyczące danych współdzielonych: clients, names, chat_history)
Zabezpieczone sekcje z cout_lock (dotyczące tylko wyjścia na konsolę – żeby uniknąć pomieszanych printów z wielu wątków)

def start() – funkcja odpowiadająca za nasłuchiwanie na nadchodzące połączenia oraz, akceptowanie nowych połączeń i przypisywanie nowych wątków dla klientów wywołując funkcję handle_client()

def broadcast(msg, sender_conn=None) – funkcja wysyłająca przychodzące wiadomości do wszystkich klientów

def handle_client(conn, addr) – funkcja obsługująca nowe połączenie, odbiera nazwę użytkownika, przekazuje klientowi historię czatu oraz uruchamia połączenie z klientem wykrywając przesłane przez niego wiadomości. Wiadomości są również sprawdzane pod kątem wiadomości specjalnych tj. `Disconnect` czy `CLS`.

Client:

`client: socket` `TCP` połączony z serwerem
`HEADER, PORT, FORMAT, DISCONNECT_MESSAGE, ADDR`: parametry konfiguracyjne potrzebne do komunikacji z serwerem

`def send(msg)` - funkcja odpowiedzialna za wysyłanie wiadomości do serwera. Wiadomość kodowana jest do bajtów i poprzedzana nagłówkiem zawierającym jej długość (stałej długości 64 bajty), co pozwala serwerowi poprawnie odczytać dane.

`def receive()` - funkcja działająca w osobnym wątku, ciągle nasłuchująca na wiadomości od serwera. Odbiera dane i wyświetla je w konsoli. Kończy działanie, gdy serwer zakończy połączenie lub wystąpi błąd.

`def start()` - funkcja inicjalizująca klienta:

- *Prosi użytkownika o podanie nazwy i przesyła ją do serwera.*
- *Uruchamia wątek nasłuchujący (`receive()`), aby odbierać wiadomości w tle.*
- *W pętli odczytuje wiadomości od użytkownika i przesyła je do serwera, do momentu wysłania komendy `!DISCONNECT`.*

`def disconnect(self)` - funkcja służąca do **rozłączenia klienta z serwerem**.

- *Wysyła do serwera wiadomość specjalną `!DISCONNECT`, informując go o zakończeniu połączenia.*
- *Po wysłaniu komunikatu zamyka lokalne połączenie socketowe (`self.client.close()`), co skutkuje zakończeniem działania klienta.*

def send_special(self, command) - funkcja odpowiedzialna za **wysyłanie specjalnych komend** (np. `cls` – czyszczenie historii, `!DISCONNECT` – rozłączenie).

- *Przyjmuje jako argument ciąg znaków `command`, który reprezentuje specjalną instrukcję dla serwera.*
- *Wiadomość jest kodowana i wysyłana tak samo jak zwykła wiadomość czatu, ale po stronie serwera trafia do specjalnej obsługi.*

4.Mechanizmy synchronizacji

W projekcie serwera czatu każdy klient działa w osobnym wątku, reprezentując niezależne połączenie z użytkownikiem. W celu zabezpieczenia sekcji krytycznych i uniknięcia problemów wynikających z jednoczesnego dostępu do współdzielonych zasobów, zastosowano następujące mechanizmy synchronizacji:

Zastosowanie blokady lock (mutex) w celu zabezpieczenia sekcji krytycznych operujących na współdzielonych strukturach danych:

- lista klientów clients,
- słownik nazw użytkowników names,
- historia wiadomości chat_history.
  Dzięki temu możliwy jest bezpieczny odczyt i zapis tych danych bez ryzyka kolizji pomiędzy wątkami.

Zastosowanie oddzielnej blokady cout_lock dla operacji print() na konsoli serwera.
Drukowanie logów z wielu wątków mogłoby prowadzić do nakładania się komunikatów, dlatego zastosowano osobny mutex do synchronizacji komunikatów wypisywanych na ekranie.

Wielowątkowość z wykorzystaniem threading.Thread(), gdzie każdy nowy klient otrzymuje własny wątek obsługi.
Serwer tworzy nowe wątki dynamicznie dla każdego połączenia i zapewnia, że obsługa wiadomości i synchronizacja dostępu do danych działa niezależnie od siebie.

Bezpieczne iterowanie po liście clients.copy() w funkcji broadcast(), co zapobiega błędom podczas równoczesnego wysyłania wiadomości i usuwania rozłączonych klientów.
W połączeniu z lock, mechanizm ten chroni serwer przed nieprzewidywalnymi wyjątkami wynikającymi z modyfikacji listy podczas iteracji.

Zabezpieczenie mechanizmu przesyłania historii czatu (chat_history) do nowo podłączonych użytkowników. Historia wiadomości jest odczytywana pod blokadą lock, a jej wysyłka jest próbą jednokierunkową (bez oczekiwania na potwierdzenie), co chroni serwer przed blokadą w przypadku nieprawidłowego klienta.

5. Podsumowanie i wnioski

Napisany program w pełni spełnia założenia projektowe, a dodatkowo oferuje kilka rozszerzeń, które podnoszą funkcjonalność oraz komfort użytkowania. Dzięki odpowiedniemu podejściu do synchronizacji i projektowania wielowątkowego, aplikacja działa stabilnie, efektywnie oraz bezpiecznie, nawet przy wielu jednoczesnych połączeniach.

W projekcie skutecznie zostały zabezpieczone wszystkie sekcje krytyczne, w tym:

- *lista aktywnych klientów,*
- *słownik nazw użytkowników,*
- *historia wiadomości.*

Każda z tych struktur może być modyfikowana przez wiele wątków, dlatego zabezpieczono je przy pomocy mechanizmów blokad (lock), co skutecznie zapobiega konfliktom i błędom wyścigu. Dodatkowo zabezpieczono również wyjścia na konsolę (cout_lock), co eliminuje problem mieszających się komunikatów z różnych wątków i zapewnia czytelny log systemowy po stronie serwera.

Na szczególną uwagę zasługuje także implementacja **prostego, intuicyjnego interfejsu graficznego (GUI)** po stronie klienta. Aplikacja kliencka wyposażona została w przyciski akcji, które pozwalają użytkownikowi w wygodny sposób:

- *połączyć się z serwerem czatu,*
- *wysyłać wiadomości tekstowe,*
- *rozłączyć się z sesji,*
- *a także – co stanowi ciekawy dodatek –* ***wyczyścić historię czatu*** *dla wszystkich użytkowników.*

Dzięki zastosowaniu specjalnych komend (np. !DISCONNECT, cls), komunikacja klient-serwer została wzbogacona o proste protokoły sterujące, które umożliwiają kontrolę nad sesją i jej zawartością.

Wnioski:

Projekt stanowi udane połączenie programowania sieciowego, wielowątkowości i podstaw interfejsu graficznego. Zastosowane mechanizmy synchronizacji gwarantują bezpieczeństwo dostępu do współdzielonych danych, a modularna konstrukcja ułatwia dalszy rozwój aplikacji – np. o nowe komendy, logowanie użytkowników czy prywatne wiadomości.

6.Źródła:

https://www.youtube.com/watch?v=3QiPPX-KeSc
https://docs.python.org/3/library/socket.html
https://docs.python.org/3/library/threading.html
https://docs.python.org/3/library/tkinter.html