Września dnia 20.01.2025

Jakub Frąckowiak 144274

Sieci Komputerowe 2

Prowadzący: Norbert Langner

Projekt komunikatora w oparciu o architekturę klient – serwer

K**omunikator tekstowy** w architekturze klient-serwer, pozwalający na wymianę wiadomości w czasie rzeczywistym. Serwer, napisany w języku C++, działa na systemie Linux, obsługuje wiele klientów jednocześnie dzięki wieloprocesowości.
Klient z graficznym interfejsem użytkownika (GUI), stworzony w Pythonie przy użyciu biblioteki **Tkinter**, może działać na dowolnym systemie operacyjnym (np. Windows, Linux).

Główne funkcjonalności:

1. **Serwer**:
    - Obsługuje jednoczesne połączenia od wielu klientów.
    - Odbiera wiadomości od klientów i odsyła odpowiedź.
    - Rejestruje każde nowe połączenie oraz wiadomości w konsoli.
2. **Klient**:
    - Posiada GUI z polami:
        - **Lista wiadomości**: Wyświetla historię rozmowy, z wiadomościami oddzielonymi kreską.
        - **Pole do wpisywania wiadomości**: Zwiększone w stosunku do standardowego pola tekstowego.
        - **Pole konfiguracji**: Pozwala użytkownikowi ustawić adres IP, port serwera oraz nazwę użytkownika.
    - Każda wiadomość wysyłana przez klienta jest opatrzona nazwą użytkownika.
3. **Komunikacja**:
    - Wymiana danych odbywa się poprzez połączenie TCP.
    - Wiadomości przesyłane są w formacie tekstowym kodowanym w UTF-8.

Cel projektu:

Projekt demonstruje podstawową komunikację sieciową między klientem a serwerem w środowisku heterogenicznym (Linux i Windows), co może być wykorzystane jako baza dla bardziej złożonych systemów komunikacji, takich jak chat grupowy lub systemy do współpracy w czasie rzeczywistym.

1. Uruchomienie

1) Server:

g++ -o server server.cpp

./server

2) klient:

python client.py

2. Ważne funkcje

```
// Funkcja obsługująca sygnał zakończenia procesów dziecka (SIGCHLD)
void handle_sigchld(int signo) {
    while (waitpid(-1, nullptr, WNOHANG) > 0) {} // Oczekuje na zakończenie procesów
```

Powyższa funkcja Zapewnia obsługę sygnału **SIGCHLD** wysyłanego przez system operacyjny, gdy proces potomny (dziecko) zakończy działanie. Dzięki tej funkcji procesy zombie (zakończone procesy potomne, które nie zostały zwolnione) są eliminowane.

Funkcja używa waitpid z flagą WHOHANG, aby nie blokować serwera podczas oczekiwania na zakończenie procesów dziecka. Jest wywoływana automatycznie po zakończeniu działania dowolnego procesu dziecka.

Tworzenie gniazda serwera

```
// Tworzenie gniazda serwera
if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    perror("Socket creation failed");
    exit(EXIT_FAILURE);
}
```

Funkcja tworzy gniazdo sieciowe, które będzie używane do obsługi połączeń przychodzących od klientów.

Funkcja socket tworzy gniazdo TCP (SOCK_STREAM) w rodzinie adresów IPv4 (AF_INET).
W przypadku błędu tworzenia wyświetlany jest komunikat, a program kończy działanie.

Bindowanie gniazda d adresu i portu

```
// Konfiguracja adresu serwera
server_addr.sin_family = AF_INET;         // IPv4
server_addr.sin_port = htons(PORT);       // Port w porządku sieciowym
inet_aton(SERVER_IP, &server_addr.sin_addr); // Konwersja adresu IP do struktury

// Bindowanie gniazda serwera do adresu i portu
if (bind(server_fd, (sockaddr*)&server_addr, sizeof(server_addr)) == -1) {
    perror("Bind failed");
    close(server_fd);
    exit(EXIT_FAILURE);
}
```

Funkcja przypisuje gniazdo do konkretnego adresu IP i portu, aby umożliwić serwerowi nasłuchiwanie na połączenia.

sockaddr_in - przechowuje dane dotyczące adresu IP i portu.
bind – łączy gniazdo z adresem IP i portem
W przypadku błędu wyświetlany jest komunikat, a program kończy działanie.

Nasłuchiwanie na połączenia klientów

```
// Rozpoczęcie nasłuchiwania na gnieździe serwera
if (listen(server_fd, 10) == -1) {
    perror("Listen failed");
    close(server_fd);
    exit(EXIT_FAILURE);
}
```

Konfiguruje serwer do nasłuchiwania na maksymalnie 10 jednoczesnych połączeń od klientów.
Funkcja ***listen*** informuje system, e serwer jest gotowy do przyjmowania połączeń.

Akceptowanie połączeń klientów

```
// Akceptowanie nowego połączenia
client_fd = accept(server_fd, (sockaddr*)&client_addr, &client_len);
if (client_fd == -1) {
    perror("Accept failed");
    continue;
}

std::cout << "Nowe połączenie od " << inet_ntoa(client_addr.sin_addr) << std::endl;
```

Obsługa połączeń przychodzących od klientów, tworząc nowe gniazda dla każdego klienta.
Funkcja ***accept*** tworzy nowe gniazdo ***client_fd***, które obsługuje połączenie z klientem.
Adres klienta jest logowany w konsoli za pomocą ***inet_ntoa***

Obsługa klienta w procesie potomnym

Proces rodzica zamyka deskryptor klienta i powraca do nasłuchiwania kolejnych połączeń.

Proces dzicka obsługuje dane od klienta w pętli:

1. Odczytuje wiadomość z gniazda client_fd
2. Wyświetla je w konsoli serwera
3. Odpowiada klientowi, dodając odpowiedni komunikat serwera.

Gdy Klient kończy połączenie, proces dziecka również kończy działanie

```
if (fork() == 0) { // Proces dziecka
    close(server_fd); // Proces dziecka nie potrzebuje deskryptora serwera

    char buffer[1024] = {0}; // Bufor na dane od klienta
    while (true) {
        // Odbieranie danych od klienta
        ssize_t bytes_read = read(client_fd, buffer, sizeof(buffer) - 1);
        if (bytes_read <= 0) {
            std::cout << "Klient zakończył połączenie." << std::endl;
            break; // Zakończenie połączenia
        }

        buffer[bytes_read] = '\0'; // Dodanie znaku końca stringa
        std::cout << "Otrzymano: " << buffer << std::endl;

        // Odesłanie potwierdzenia do klienta
        std::string response = "Serwer odpowiada: " + std::string(buffer);
        write(client_fd, response.c_str(), response.size());
    }

    close(client_fd); // Zamknięcie połączenia z klientem
    exit(0);          // Zakończenie procesu dziecka
} else {
    close(client_fd); // Proces rodzica zamyka deskryptor klienta
}
```

Zamknięcie gniazd

```
close(server_fd); // Zamknięcie gniazda serwera
close(client_fd); // Zamknięcie połączenia z klientem
```

Upewnia się, że wszystkie gniazda są zamykane po zakończeniu pracy.
Każde gniazdo, które nie jest już potrzebne, zostaje zamknięte w celu zwolnienia zasobów

Inicjalizacja aplikacji GUI

```python
def __init__(self, root):
    self.root = root
    self.root.title("Komunikator Klient")
    self.client_socket = None  # Gniazdo klienta
    self.username = None  # Nazwa użytkownika
```

W aplikacji GUI mamy sekcje konfiguracji połączenia, podanie nazwy użytkownika, okno wiadomości i informacje o statusie połączenia.

Połączenie z serwerem

```python
def connect_to_server(self):
    # Pobieranie IP, portu serwera i nazwy użytkownika
    server_ip = self.server_ip_entry.get()
    server_port = int(self.server_port_entry.get())
    self.username = self.username_entry.get().strip()

    if not self.username:
        messagebox.showerror("Błąd", "Nazwa użytkownika nie może być pusta.")
        return

    try:
        # Nawiązywanie połączenia z serwerem
        self.client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.client_socket.connect((server_ip, server_port))
        self.status_label.config(text="Połączony z serwerem", fg="green")
        self.send_button.config(state=tk.NORMAL)
        self.connect_button.config(state=tk.DISABLED)
        self.username_entry.config(state=tk.DISABLED)

        # Wątek odbierający wiadomości od serwera
        threading.Thread(target=self.receive_messages, daemon=True).start()
    except Exception as e:
        messagebox.showerror("Błąd", f"Nie można połączyć z serwerem: {e}")
```

Pobiera konfiguracje połączenie IP, port oraz nazwę użytkownika. Tworzy gniazdo TCP i nawiązuje połączenie z serwerem. Po nawiązaniu połączenia:

1) Wyświetla status połączono
2) Aktywuje przycisk do wysłania wiadomości
3) Włącza możliwość edycji komunikacji

Uruchamia wątek do odbierania wiadomości ***receive_messages***

Odbieranie wiadomości

```
def receive_messages(self):
    # Odbieranie wiadomości od serwera
    while True:
        try:
            msg = self.client_socket.recv(1024).decode("utf-8")
            if msg:
                self.msg_list.insert(tk.END, f"Serwer: {msg}")
                self.msg_list.insert(tk.END, "-" * 50)  # Kreska oddzielająca
            else:
                break
        except:
            break
```

Odbiera wiadomości od serwera w sposób asynchroniczny (w osobnym wątku). Wyświetla odebrane wiadomości w oknie rozmowy.

Działa w osobnym wątku, aby nie blokować głównego interfejsu użytkownika. W pętli odbiera wiadomości z serwera za pomocą recv:
Dekoduje wiadomość w formacie UTF-8.
Wyświetla ją w polu listy wiadomości z kreską oddzielającą.

Kończy działanie, gdy serwer zamknie połączenie.

Wysyłanie wiadomości

```
def send_message(self):
    # Wysyłanie wiadomości do serwera
    msg = self.entry_field.get("1.0", tk.END).strip()
    if msg:
        try:
            # Dodanie nazwy użytkownika do wiadomości
            full_message = f"{self.username}: {msg}"
            self.client_socket.sendall(full_message.encode("utf-8"))

            # Wyświetlenie wiadomości w oknie klienta
            self.msg_list.insert(tk.END, f"Ty: {msg}")
            self.msg_list.insert(tk.END, "-" * 50)  # Kreska oddzielająca
            self.entry_field.delete("1.0", tk.END)
        except Exception as e:
            messagebox.showerror("Błąd", f"Nie można wysłać wiadomości: {e}")
```

Wysyła wiadomość do serwera poprzez ***sendall.***

Zamknięcie połączenia z serwerem

```
def on_close(self):
    # Zamknięcie połączenia przy zamykaniu aplikacji
    if self.client_socket:
        self.client_socket.close()
    self.root.destroy()
```

Zamyka połącznie z serwerem i kończy działanie aplikacji po zamknięciu okna przez użytkownika.
Zamyka aktywne połączenie z serwerem.

4. Podsumowanie

Projekt nauczył mnie podstaw komunikacji sieciowej opartej na protokole TCP, w tym tworzenia, bindowania i nasłuchiwania gniazd na serwerze oraz nawiązywania połączenia przez klienta. Serwer może obsługiwać wielu klientów jednocześnie, wykorzystując wieloprocesowość, oraz jak zarządzać procesami potomnymi, unikając problemu procesów zombie dzięki obsłudze sygnału **SIGCHLD**. Wykorzystałem asynchroniczną komunikację przy użyciu wątków, aby umożliwić odbieranie wiadomości od serwera bez blokowania interfejsu użytkownika. Wykorzystałem praktyczne aspekty protokołu TCP, takie jak kodowanie i dekodowanie przesyłanych wiadomości w UTF-8, oraz tworzenie prostego protokołu komunikacji.