Aplikacja Sieciowa – wątki po stronie klienta

Na ostatnich zajęciach zajmowaliśmy się komunikacją pomiędzy klientem a serwerem. Wynikiem naszej pracy był program klienta, który za pomocą serwera mógł się połączyć z drugim klientem. Cała komunikacja, pomimo tego, że przebiegała bez problemów, miała jedną wadę: za każdym razem kiedy wysyłano komunikat do drugiego klienta należało poprzedzić go bajtem '1' natomiast chcąc odczytać komunikat należało wybrać '2'. Problem ten wynikał z synchronizmu przesyłania i odbierania danych. W programie klienta występował tylko jeden wątek główny (*main*) , który w tym samym czasie może obsługiwać tylko jedne typ operacji: zapis lub odczyt z serwera.

W dzisiejszym programie rozwiążemy ten problem poprzez dodanie wątku roboczego, który przez cały czas będzie odczytywał dane z serwera.

Najpierw jednak musimy zmodyfikować program serwera który będzie teraz wykonywał operacje wymiany danych bez „zwracania uwagi" na pierwszy bajt komunikatów pochodzących od klienta.

W miejsce kodu zajmującego się transferem danych:

```
if(server.recvbuf[0]=='1') ///komunikat jest przeznaczony dla klienta drugiego wówczas:
{
    for(int j=0; j<31; j++)
    {
        if(client2==current_client)
        server.msg2client1[j]=server.recvbuf[j+1];
        if(client1==current_client)
        server.msg2client2[j]=server.recvbuf[j+1];
    }
}

if(server.recvbuf[0]=='2') ///pobieramy komunikat:
{
    if(client2==current_client){
        for(int j=0; j<32; j++)
        server.sendbuf[j]=server.msg2client2[j];
        server.sendData(client2);
    }
    if(client1==current_client){
        for(int j=0; j<32; j++)
        server.sendbuf[j]=server.msg2client1[j];
        server.sendData(client1);
    }
}
```

Musimy dodać kod który zrealizuje operacje:

```
Jeśli current_client==client1 to:

      Przypisz recvbuf do msg2client2

      Przypisz msg2client2 do sendbuf

      wyślij dane do client2

Jeśli current_client==client2 to:

      Przypisz recvbuf do msg2client1

      Przypisz msg2client1 do sendbuf

      wyślij dane do client1
```

Wykonanie tych operacji kończy zmiany w aplikacji serwera.Należy skompilować i uruchomić serwer. Teraz musmimy zmodyfikować program klienta. Tutaj podobnie jak ma to miejsce w aplikacji serwera musimy dołączyć nowy wątek (wątek dołączony zostanie do procesu programu). W *main* dodajemy wątek oraz funkcję CreateThread:

```
DWORD thread; // watek – numer

// Initialize Winsock.
klient.initializeWSA();
// Create a socket.
klient.socketCreate();
// bind socket.
//Prompt the user to enter anIP
cout<<"Enter an IP address (127.0.0.1 is the loopback address)"<<endl;
//Wait for the user to input an IP
 cin >> ipAddress;
//connect to server
klient.connectToServer(ipAddress.c_str(), port);
//Create thread
CreateThread(NULL, 0,receive_cmds,0, 0, &thread);

...
```

Zaraz za staworzonym wątkiem rozpoczyna się pętla, która przez cały czas czeka na komunikat wpisany przez użytkownika. Dotychczas w pętli znajdowała się też część kodu obsługujacego odbieranie danych. Tym razem, kod ten wykonany zostanie we funkcji wątku oznaczonej nazwą: *receive_cmds*.

Wunkcja ta musi być zdefiniowana w następujący sposób:

```
DWORD WINAPI receive_cmds(LPVOID lpParam)
{

                    Ciało funkcji

return 0;

}
```

Ciało funkcji stanowi cześć kodu, odpowiadająca za odbiór danych:

```
klient.bytesRecv=-1;

while(klient.bytesRecv<0 )
{
    //  printf("receiving");
        klient.receiveData();
}

//printf("Klient otrzymal komunikat: %s\n",klient.recvbuf);
printf("<<: ");
for(int k=0; k<32; k++){
printf("%c", klient.recvbuf[k]);
}
printf("\n");
printf(">>: ");

klient.clearBuff();
```

w całości zamknięta w **pętli nieskończonej**.

Program klienta jest zakończony. Należy teraz skompilować i uruchomić aplikację. Podczas testów dane powinny być asynchronicznie odbierane oraz wysyłane.

Pierwszy problem naszego „komunikatora” jest już rozwiązany. Teraz pozostaje kwestia przesyłania danych w innej formie niż tablica znaków. Otóż bardzo często chcemy przesłać dane różnego typu określające np. aktualny stan gry. Najlepszym rozwiązaniem jest przesłanie struktury tzw. ramki. Najpierw musimy dodać strukturę do programu. Zaczynamy od serwera gdzie ramka danych już występuje i jest ona zdeklarowana w pliku nagłówkowym:

```
#endif
#include "winsock2.h"

struct SFrame {
    char data[32];
    int flag;
};

class CS[illegible]WSA
{
public:
    CServerWSA(void);
public:
```

W klasie *CServerWSA* można też znaleźć pola typu naszej struktury o nazwach:

```
SFrame inbuff;
SFrame outbuff;
```

Teraz musimy użyć naszych ramek. W serwerze zaczynamy od powiększenia buforów wysyłającego i odbierającego. Wysyłać będziemy typ całkowity (4bajty) oraz 32 znaki (32 bajty) nasze bufory muszą mieć co najmniej 36 bajtów pojemności. Rozszerzamy je zatem np. do 40 znaków:

```
public:
    SOCKET m_socket;//socket
    sockaddr_in server_service; // struktura zwiazana z serwerem - patrz bindSocket
    char sendbuf[40]; //bufor wyjściowy
    char recvbuf[40]; // bufor wejściowy
    char msg2client1[32];
    char msg2client2[32];
    int bytesRecv; // liczba odebranych bajtow
    SFrame inbuff;
    SFrame outbuff;
```

Teraz przechodzimy do pliku *CServerWSA.cpp* gdzie zaczniemy od funkcji czyszczącej bufory. Musi ona czyścić całe 40 bajtów:

```
for(int i = 1;i<40;i++){
        sendbuf[i]=0;
        recvbuf[i]=0;
}
```

Następnie zmieniamy metodę wysyłającą dane, zmieniając jej kod na:

```
void * wsk_void;
wsk_void = &outbuff;
char * wsk_char=(char *)wsk_void;
int k=sizeof(outbuff);
for (int i=0;i<sizeof(outbuff);i++) {
        sendbuf[i] = wsk_char[i];
}

int bytesSent = send(client, sendbuf, sizeof(outbuff), 0 );
printf( "Bytes Sent: %ld\n", bytesSent );
```

Można zauważyć, że przez referencję przekazujemy dane ze struktury do wskaźnika, który następnie jest rzutowany na typ char.
Podobną operację musimy wykonać w metodzie odbierającej dane:

```
void CServerWSA::receiveData(SOCKET client){
    //Odebrane dane są zapisywane do bufora recbuf, którego maksymalny rozmiar jet określony
    //w wywołaniu funkcji recv. Ilość danych odczytanych jest zwracana do zmiennej bytesRecv
    bytesRecv = SOCKET_ERROR;
    while( bytesRecv == SOCKET_ERROR ) {
        bytesRecv = recv( client, recvbuf, 36, 0 );
        if ( bytesRecv == 0 || bytesRecv == WSAECONNRESET ) {
            printf( "Connection Closed.\n");
            break;
        }
        if (bytesRecv < 0)
            return;

        void * wsk_void;
        wsk_void = &inbuff;
        char * wsk_char = (char *) wsk_void;
        int k = sizeof(inbuff);
        for (int i=0;i<sizeof(inbuff);i++)
        wsk_char[i] = recvbuf[i];



    printf( "Bytes Recv: %ld\n", bytesRecv );
    }

}
```

Tutaj jednak celem jest przepisanie odebranych danych do postaci struktury. Podsumowując: za każdym razem dane są wysyłane z ramki *SFrame outbuff*, jednocześnie dane przychodzące od klientów formatowane są do postaci ramki *SFrame inbuff*. Na rysunku powyżej podkreślono funkcję odbierającą, która jak widać czeka na 36 bajtów. Jest to bardzo ważne, bowiem oczekiwanie na inną ilość danych spowoduje podzielenie paczek danych oraz późniejsze błędne formatowanie.

Na koniec, w serwerze zmieniamy postać funkcji wykonywanej we wątku. Dla przykładu podano postać obsługi klienta nr. 2:

```
if(client2==current_client){
                    for(int j=0; j<32; j++)
                    {
                        server.msg2client1[j]=server.inbuff.data[j];
                        server.outbuff.data[j]=server.msg2client1[j];
                    }
                    server.outbuff.flag=2; ///flaga mówiąca o nadawcy
                    server.sendData(client1);
                }
```

Analogicznie należy zmodyfikować obsługę klienta 1.

Skończyliśmy zmieniać aplikację serwera, nadszedł czas na aplikację klienta. Tutaj modyfikację zaczynamy od dodania ramki danych (struktury) oraz dodania odpowiednich pól: *inbuff* oraz *outbuff*:

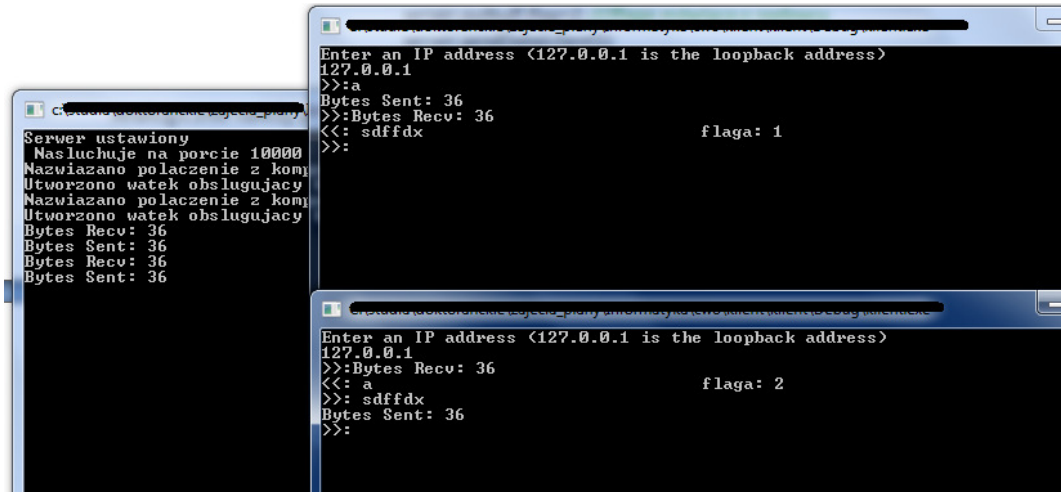
```
struct SFrame {
    char data[32];
    int flag;
};

class CClientWSA
{
    public:
        CClientWSA(void);
    public:
        ~CClientWSA(void);
        void initializeWSA(); // Initialize Winsock.
        void socketCreate(); // Create a socket.
        void connectToServer(const char * addr, unsigned short port); //po³aczenie z serwerem
        void sendData(); //send sendbuff
        void receiveData(); //receive data
        void clearBuff(void);//czysci bufor wejsciowy i wyjsciowy
    public:
        SOCKET m_socket;//socket
        sockaddr_in clientService; // struktura zwiazana z serwerem - patrz connectToServer
        char sendbuf[40]; //bufor wyjoeciowy
        char recvbuf[40]; // bufor wejoeciowy
        int bytesRecv;//liczba otrzymanych bajtow;
        SFrame inbuff;
        SFrame outbuff;
};
```

Działając tak jak podczas aplikacji serwera zmieniamy metody: wysyłającą dane, odbierającą dane oraz czyszczącą bufory. Ostatecznie w wątku modyfikujemy wyświetlanie danych:

```
for(int k=0; k<32; k++){
        printf("%c", klient.inbuff.data[k]);
}
printf(" flaga: %d", klient.inbuff.flag);
```

oraz modyfikujemy wysyłanie danych w main, tak żeby wysyłane string wpisywany był bezpośrednio do struktury. Poprawnie działające aplikacje serwera i klienta, podczas asynchronicznej komunikacji wymieniają dane w postaci znaków oraz flagi:



Podsumujemy nasze aplikacje:

1. Serwer jak i klient działają w oparciu o wątek główny i roboczy
2. Serwer oraz klient przesyłają pomiędzy sobą struktury danych
3. Struktura danych (ramka) może składać się z różnej ilości danych, podczas wysłania danych wszystko traktowane jest jako znaki.

Stan wiedzy po zajęciach:

1. **Struktura jako pole klasy**
2. **Mechanizm przesyłania struktury**
3. **Wątek – mechanizm, cel, realizacja.**

Sprawozdanie,

1. Nasz ramka ma składać się z tablicy znaków (32 znaki) oraz dwóch zmiennych typu całkowitego. Zmodyfikować kod tak aby możliwe było przesyłanie ramki. Załączyć kod, który uległ modyfikacji.
2. Nasz serwer obsługuje *n* (gdzie n=3) klientów. Zaproponuj mechanizm przesyłania danych, tak, aby możliwe było przesłanie danych pomiędzy dowolnymi klientami. Dokonaj zmian w kodzie klienta i serwera (wykonanych na zajęciach) tak by spełnić wymogi zadania.
   Uwaga: każdy klient wie od kogo otrzymał wiadomość, natomiast podczas wysyłania wiadomości wysyłana jest do konkretnego klienta. Załączyć kod, który uległ modyfikacji oraz dodać stosowny komentarz.