Politechnika Warszawska
Wydział Mechatroniki

Programowanie niskopoziomowe

Projekt:
Program w asemblerze – klient oraz serwer bazujący na protokole TCP/IP odbierający i interpretujący przesyłany przez aplikacje klienckie tekst. Wielowątkowość.

Filip Szatkowski, gr. 33ip
Prowadzący: dr inż. Sławomir Paśko

Warszawa 2017/18

1. Cel projektu

Celem projektu było napisanie w asemblerze programów serwera oraz klienta komunikujących się za pomocą protokołu TCP/IP. Serwer miał interpretować przesyłany przez klienta tekst i wysyłać z powrotem odpowiednią wiadomość zwrotną. Dodatkowo serwer miał umożliwiać obsługę kilku klientów naraz poprzez tworzenie dla każdego z klientów osobnego wątku.

2. Wprowadzenie

Model TCP/IP jest powszechnie wykorzystywany do transmisji danych przez sieci komputerowe. Implementuje on najważniejsze warstwy modelu OSI, min. transportową(protokół TCP) oraz sieciową(IP). Przy wysyłaniu wiadomości przechodzi ona przez każdą warstwę, gdzie dodawane są odpowiednie nagłówki zależne od użytych protokołów. Następnie dostarczona do odbiorcy wiadomość jest odczytywana przez każdą z warstw w przeciwnej kolejności.

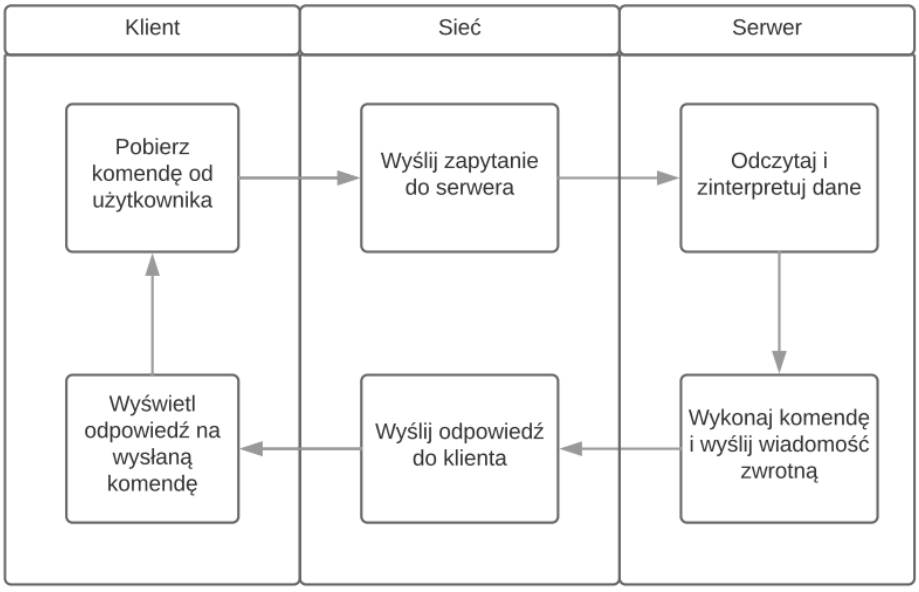
Protokół TCP jest protokołem wymagającym nawiązania połączenia między komunikującymi się stronami - odbiorca potwierdza otrzymanie każdej informacji, kontrolowane jest także, czy wiadomość dotarła bez błędów oraz w odpowiedniej kolejności. Dzięki temu jest używany wszędzie tam, gdzie wymagana jest niezawodność przy przesyłaniu danych.

Model klient – serwer ustala podział ról podczas komunikacji. Jeden serwer zapewnia usługi dla jednego lub więcej klientów wysyłających żądania obsługi. Jest on stosowany między innymi w serwerach WWW, poczty elektronicznej lub systemach zarządzania bazami danych. W celu umożliwienia płynnej obsługi przez serwer wielu klientów wydziela się często do obsługi każdego z nich osobny wątek, co pozwala głównemu wątkowi zająć się przyjmowaniem nowych wątków oraz zarządzaniem najistotniejszymi zadaniami.

3. Opis działania

Aplikacje serwera oraz klienta napisano w języku MASM(Microsoft Macro Assembler) jako aplikacje konsolowe. Język MASM wybrano ze względu na możliwość wywoływania w nim funkcji API Windowsa, w szczególności biblioteki WinSock implementującej komunikację za pomocą protokołu TCP/IP. Po klienta oraz serwera następuje nawiązanie między nimi komunikacji na wolnym gnieździe, tworzony jest wątek obsługujący danego klienta, natomiast serwer czeka na pojawienie się kolejnych próśb nawiązania połączenia.

Schemat komunikacji między serwerem a klientem jest przedstawiony poniżej.



Interpretacja danych przez serwer polega na wydzieleniu z odebranej wiadomości czterech pierwszych znaków jako komendy, pominięciu piątego znaku oraz zapisaniu reszty wiadomości. Komenda porównywana jest z obsługiwanymi przez serwer komendami, następnie jeśli pasuje do którejś z nich podejmowana jest odpowiednia akcja i serwer z powrotem wysyła wiadomość odpowiadającą tej akcji. Jeśli wysłana przez klienta wiadomość nie zostanie zinterpretowana jako komenda, z powrotem wysyłany jest komunikat o nieznanym poleceniu.

3.1. Działanie wspólnej części obydwu programów

Programy napisano z użyciem zestawu instrukcji procesorów z serii .386 i 32-bitowego modelu pamięci. Kod podzielony jest na segmenty zgodnie z konwencją stosowaną w asemblerze. Dyrektywa **invoke** dostępna w MASM-ie pozwala na używanie funkcji API, przy czym w projekcie korzystano głównie z funkcji biblioteki WinSock.

Programy posługują się podobnym zestawem tekstów używanych przy sygnalizacji błędów. Do wyświetlania oraz pobierania wiadomości tekstowych wykorzystane zostały funkcje **StdOut** oraz **StdIn**. Każdemu wywołaniu funkcji WinSocka towarzyszy sprawdzenie, czy wywołanie się powiodło – w przypadku niepowodzenia, za funkcji **WSAGetLastError** do rejestru **eax** pobierany jest kod błędu, który po przeformatowaniu za pomocą funkcji **FormatMessage** wyświetlany jest w oknie wywoływanym za pomocą **MessageBoxA**.

Po uruchomieniu każdego z programów następuje inicjalizacja biblioteki WinSock za pomocą **WSAStartup** do wersji 2. Następnie następuje utworzenie odpowiedniego gniazda funkcją **socket**(gniazdo TCP/IP) oraz zapisanie do struktury addr_in adresu IP serwera oraz używanego portu. Pod koniec działania programu gniazdo to jest zamykane funkcją **closesocket** oraz wywoływana jest funkcja **WSACleanup**.

Klient oraz serwer do komunikacji używają funkcji **send** i **recv**, wykorzystując 128-znakowe bufery.

3.2. Działanie klienta

Po utworzeniu gniazda klient wywołuje funkcję **connect** używając przekazując jej wskaźnik na gniazdo oraz na adres serwera. Jeśli serwer nie odpowiada, aplikacja kliencka wyświetla odpowiednie okno dialogowe, które blokuje działanie programu. Po kliknięciu ‘OK’ następuje kolejna próba połączenia. Jeśli połączenie zostanie zaakceptowane, klient czeka na wiadomość powitalną od serwera, wyświetla ją, a następnie przechodzi do pętli wczytywania od użytkownika tekstu, wysyłania go do serwera i wyświetlania odpowiedzi. Po wpisaniu odpowiedniego polecenia serwer może rozłączyć się z klientem i zakończyć działanie aplikacji klienckiej.

3.3. Działanie serwera

Serwer tworzy gniazdo do nasłuchiwania i przypisuje je do utworzonego wcześniej adresu funkcją **bind**, po czym wywoływana jest funkcja **listen**, która blokuje główny wątek do czasu pojawienia się żądania połączenia ze strony klienta. Połączenie jest akceptowane funkcją **accept**, która zwraca deskryptor gniazda służącego do komunikacji z nowym klientem. Następnie wysyłana jest do niego wiadomość powitalna i serwer tworzy nowy wątek funkcją **CreateThread**, której przekazuje deskryptor gniazda klienta oraz wskaźnik adresu funkcji **ThreadProc**. Następnie serwer wraca do nasłuchiwania w oczekiwaniu na kolejne połączenia.

Funkcja **ThreadProc** odpowiedzialna jest za odbiór i interpretację komend klienta oraz przesłanie odpowiedniej odpowiedzi. Mechanizm interpretacji tekstów opisany powyżej przyśpiesza wykorzystanie funkcji **lstrcmpiA** pozwalającej porównać dwa łańcuchy znaków. Po otrzymaniu odpowiedniej komendy wątek zamyka gniazdo klienta i wywołuje **ExitThread**, aby zakończyć pracę.

4. Instrukcja obsługi

Należy włączyć obie aplikacje i poczekać na wyświetlenie komunikatu o nawiązaniu połączenia. Następnie z konsoli klienta można wysyłać polecenia do serwera, co powoduje jego odpowiednią reakcję oraz otrzymanie wiadomości zwrotnej. Serwer obsługuje następujące komendy:

-echo – wysyła z powrotem do klienta tekst znajdujący się po komendzie

-show – wyświetla w konsoli serwera wiadomość od klienta

-read – wysyła do klienta tekst znajdujący się w zmiennych data1, data2, data3, przykładowa składnia komendy: read data1

-quit – powoduje zamknięcie przez serwer gniazda odpowiedzialnego za komunikację z danym klientem, wysłanie wiadomości o zakończeniu połączenia oraz wyłączenia klienta

5. Wnioski

Napisany program jest stosunkowo prosty w działaniu i ze względu na to ma on zastosowanie tylko dydaktyczne, ale jego stworzenie i tak wymagało sporo wysiłku. Wykorzystanie osobnych wątków do obsługi nowych klientów przez serwer jest bardzo wygodne i pozwala na napisanie bardziej elastycznego programu. Należy jednak uważać na związane z wątkami pułapki, takie jak na przykład jednoczesny zapis do tego samego obszaru pamięci przez dwa wątki. Asembler jako język programowania również nie jest zbyt przyjazny dla nowego użytkownika, ciężki w debugowaniu i wymuszający zastanowienie się nawet przy pisaniu prostych operacji oraz znajomość działania procesora na najniższym poziomie. W zamian za to asembler pozwala na całkowitą kontrolę nad działaniem programu i jego maksymalną optymalizację, o ile programista umie się nim posługiwać.