

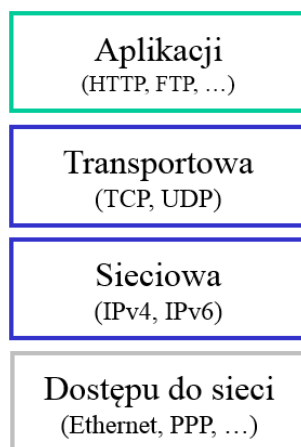
Komunikator

Warstwowy model komunikacji

Internet bazuje na warstwowym modelu komunikacji DoD (ang. *Department of Defense model*), znanym też jako model TCP/IP albo stos protokołów TCP/IP.

Model DoD definiuje 4 warstwy sieci ich zadania:

1. Warstwa aplikacji
Realizuje użyteczne dla człowieka zadania, jak pobieranie stron internetowych w usłudze WWW, wysyłanie i odbieranie poczty elektronicznej, połączenie z konsolą przez SSH itp. Obejmuje zestaw protokołów, jak HTTP, SMTP, POP3, IMAP, SSH, FTP i wiele innych.
2. Warstwa transportowa
Zapewnia poprawność i kompletność danych przesyłanych między hostami oraz kieruje dane do właściwych aplikacji. Najczęściej używane protokoły transportowe to TCP i UDP, ale są też inne, np. QUIC. Adresem w tej warstwie jest numer portu.
3. Warstwa sieciowa
Kluczowa warstwa Internetu, odpowiedzialna za znajdowanie drogi (ang. *route*) między hostami. Wykorzystuje protokół IP do przysyłania danych, a wewnętrznie wiele różnych protokołów (np. RIP, OSPF, BGP), dzięki którym routery znajdują drogę dla pakietów IP. Adresem w tej warstwie jest adres IP.
4. Warstwa dostępu do sieci
Odpowiada za przesyłanie danych między kartami sieciowymi przez fizyczne połączenia kablowe, światłowodowe lub radiowe. Adresem w tej warstwie jest adres MAC.



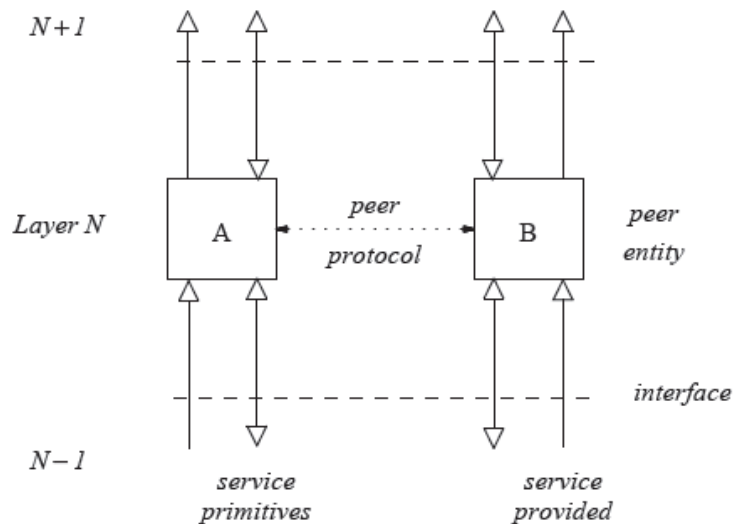
Warstwy modelu DoD

Protokoły internetowe

Moduł każdej warstwy sieciowej (ang. *peer entity*) komunikuje się wyłącznie ze swoim odpowiednikiem po drugiej stronie łącza, posługując się przy tym określonym protokołem komunikacyjnym (ang. *peer protocol*).

Do realizacji swoich zadań protokół n-tej warstwy wykorzystuje usługi elementarne (ang. *service primitives*) świadczone przez warstwę niższą, a jednocześnie świadczy określone usługi elementarne dla warstwy wyższej. Lista usług elementarnych jest różna dla różnych protokołów, nawet jeżeli należą do tej samej warstwy sieciowej.

Usługi elementarne mogą być pasywne, wykonywane z inicjatywy warstwy wyższej, albo aktywne, wykonywane z inicjatywy samego protokołu. W praktyce usługi pasywne to polecenia, które realizuje protokół (np. *wyślij dane*), a aktywne to informacje o zdarzeniach (np. *odebrano dane*). W programowaniu są to, odpowiednio, metody i zdarzenia.



Komunikacja w modelu warstwowym i usługi elementarne

Aby protokół mógł prawidłowo realizować swoje zadania, musi znać usługi elementarne warstwy niższej, ale też uwzględniać ograniczenia tych usług. Dla przykładu: protokół TCP daje gwarancję dostarczania danych z zachowaniem ich kolejności, ale protokół IP działa według zasady *best effort* („robim, co możemy”), co oznacza że może zgubić pakiet, a ponadto pakiety mogą być dostarczane w innej kolejności, niż były wysłane. Protokół TCP musi to uwzględniać, dlatego ma wewnętrzne mechanizmy zapewniające powtarzanie pakietów zgubionych oraz ustawianie pakietów we właściwej kolejności – zanim odebrane dane zostaną przekazane warstwie aplikacji.

Protokół TCP

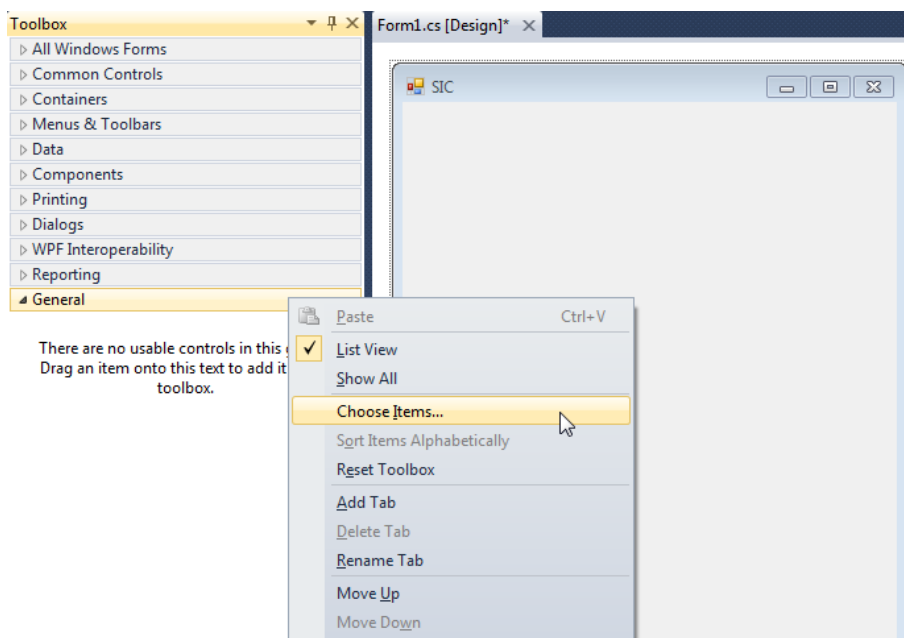
Protokół TCP (ang. *Transmission Control Protocol*) jest protokołem warstwy transportowej. Jest to protokół połączeniowy, działający w relacji klient-serwer. Gwarantuje dostarczenie wszystkich pakietów bez błędów, w całości, z zachowaniem kolejności, bez duplikatów. TCP jest protokołem binarnym bajtowym – przesyła dane binarne, w formie strumienia bajtów, nie interpretując ich w żaden sposób. Moduł TCP może połączyć kilka mniejszych porcji danych wysyłanych przez aplikację w jeden pakiet albo odwrotnie, podzielić większą porcję danych na kilka pakietów.

Protokół TCP definiuje 7 usług elementarnych, 3 pasywne i 4 aktywne:

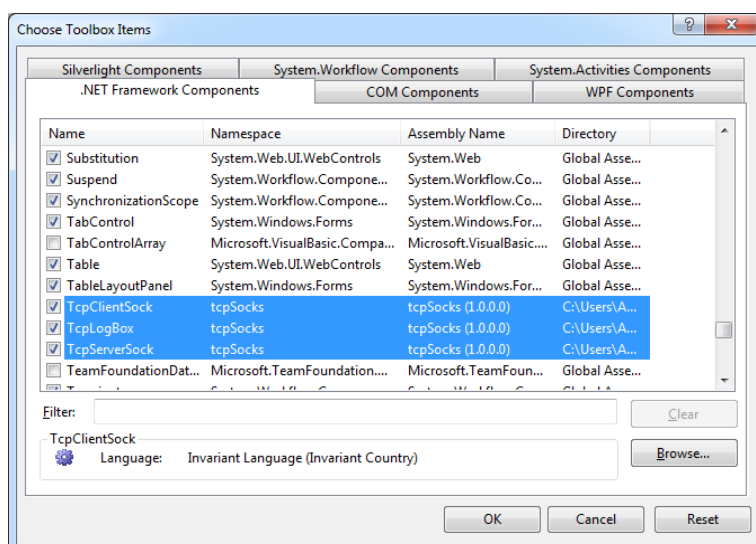
1. Open – otwarcie na połączenia klientów (serwer) albo połączenie z serwerem (klient),
2. Connect – informacja o nawiązaniu połączenia,
3. Send – wysłanie danych,
4. Read – informacja o odebraniu danych,
5. Close – zamknięcie połączenia,
6. Disconnect – informacja o zamknięciu połączenia,
7. Error – informacja o błędzie.

Pakiet TcpSocks

Pakiet komponentów *TcpSocks* nie ma wersji instalacyjnej, aby z niego skorzystać należy ręcznie dodać komponenty do przybornika Visual Studio. W tym celu należy utworzyć projekt *C# Windows Forms (.NET Framework)*, wybrać zakładkę przybornika, np. *General*, kliknąć jej nazwę prawym klawiszem myszy i wybrać polecenie *Choose Items*.



W oknie *Choose Toolbox Items*, które się wtedy wyświetli, należy kliknąć przycisk *Browse*, wskazać plik *tcpSocks.dll* i potwierdzić. Na liście komponentów *.NET Framework* powinny się pojawić trzy nowe komponenty: *TcpClientSock*, *TcpServerSock* i *TcpLogBox*. Po potwierdzeniu zmian przyciskiem *OK*, komponenty powinny pojawić się w przyborniku.



Komponenty internetowe *TcpSocks*

Komponenty *TcpClientSock* i *TcpServerSock* zarządzają połączeniami TCP/IP. Wewnętrznie wykorzystują obiekty klasy *System.Net.Sockets.Socket*, które z kolei korzystają z należącej do Windows technologii Windows Socket, będącej implementacją technologii Berkeley Socket. Komponenty *TcpSock* udostępniają wszystkie pasywne i aktywne usługi elementarne protokołu TCP w postaci metod i zdarzeń.

Komponenty *TcpSock* są dedykowane do komunikacji tekstowej – konwertują wysyłaną wiadomość typu *String* na strumień bajtów, czego wymaga protokół TCP, a po drugiej stronie łączy konwertują strumień bajtów z powrotem na typ *String*. Dodatkowo zapewniają, że każda wiadomość wysłana jednorazowo metodą *Send* zostanie w całości przekazana po drugiej stronie łączy w oddzielnym zdarzeniu *Read* – pomimo tego, że protokół TCP może połączyć kilka krótszych pakietów w jeden większy lub odwrotnie.

TcpClientSock

Właściwości

- **IsOpen** – (Boolean) czy trwa połączenie z serwerem

Metody

- **Open** – otwiera połączenie z serwerem, argumenty:
 - String **Address** – adres IP serwera,
 - Int32 **Port** – numer portu TCP serwera
- **Close** – zamyka połączenie z serwerem
- **Send** – wysyła wiadomość, argumenty:
 - String **Msg** – wiadomość,

Zdarzenia

- **Connect** – informuje o połączeniu z serwerem, argumenty:
 - sender – komponent zgłaszający zdarzenie
 - Sock – gniazdko TCP/IP, przez który można wysłać wiadomość,
- **Disconnect** – informuje o rozłączeniu z serwerem,
- **Error** – informuje o wystąpieniu błędu TCP, argumenty:
 - sender – komponent zgłaszający zdarzenie,
 - Sock – gniazdko TCP/IP,
 - **code** – numer błędu TCP/IP,
- **Read** – informuje o odebraniu wiadomości, argumenty:
 - sender – komponent zgłaszający zdarzenie,
 - Sock – gniazdko TCP/IP, przez który można wysłać odpowiedź,
 - **msg** – odebrana wiadomość.

TcpServerSock

Właściwości

- **IsOpen** – (Boolean) czy jest otwarty port TCP
- **Socks** – (TcpSockThread[]) lista aktualnie połączonych gniazdek klienckich

Metody

- **Open** – otwiera port TCP, argumenty:
 - String **Address** – adres IP serwera,
 - Int32 **Port** – numer portu TCP serwera
- **Close** – zamyka port TCP
- **Send** – wysyła wiadomość, ma trzy warianty:
 - Send(String **Msg**) – wysyła wiadomość do wszystkich,
 - Send(String **Msg**, TcpSockThread **Except**) – jw. z wyjątkiem wskazanego
 - Send(String **Msg**, SelectSock **IsValid**) – do wszystkich spełniających warunek

Argumenty

- **Msg** – wiadomość do wysłania,
- **Except** – gniazdko, do którego wiadomość nie zostanie wysłana
- **IsValid** – funkcja, której rezultat decyduje czy wiadomość ma być wysłana

Zdarzenia

- **Connect** – informuje o połączeniu z klientem, argumenty:
 - sender – komponent zgłaszający zdarzenie
 - Sock – gniazdko TCP/IP klienta, przez które można wysłać wiadomość
- **Disconnect** – informuje o rozłączeniu klienta
- **Error** – informuje o wystąpieniu błędu TCP, argumenty:
 - sender – komponent zgłaszający zdarzenie
 - Sock – gniazdko TCP/IP klienta (może być *null*)
 - **code** – numer błędu TCP/IP
- **Read** – informuje o odebraniu wiadomości, argumenty:
 - sender – komponent zgłaszający zdarzenie
 - Sock – gniazdko TCP/IP klienta, przez które można wysłać odpowiedź
 - **msg** – odebrana wiadomość

TcpSockThread

Reprezentuje połączenie Windows Socket w komponentach *TcpClientSock* i *TcpServerSock*. Obiekt klasy *TcpSockThread* jest przekazywany jako jeden z argumentów do funkcji obsługi zdarzeń, m.in. *Connect* i *Read*

Właściwości:

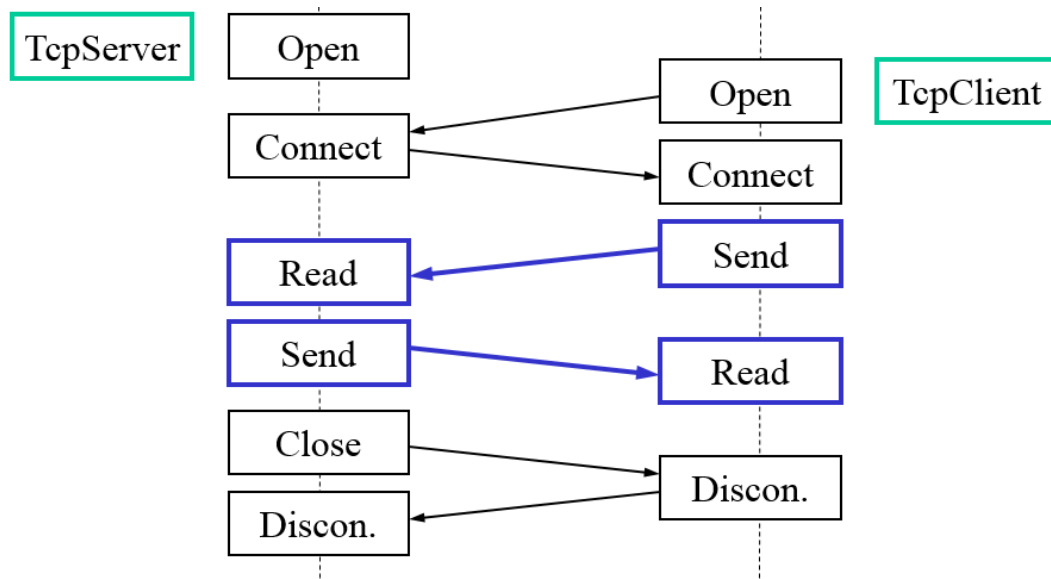
- **LocalIP** – lokalny adres IP
- **LocalPort** – lokalny port
- **RemoteIp** – zdalny adres IP
- **RemotePort** – zdalny port
- **Data** – (Object) dowolne dane programu; Przy odczycie wymagane jest rzutowanie

Metody:

- **Close** – zamyka połączenie
- **Send** – wysyła wiadomość

Kolejność czynności

Protokół TCP jest protokołem połączeniowym, co oznacza że zanim będzie można wysyłać dane, musi zostać ustanowione połączenie pomiędzy klientem a serwerem. Po połączeniu obie strony – klient i serwer – mogą w dowolnej chwili wysyłać dane. Kiedy wymiana danych się zakończy, należy zamknąć niepotrzebne już połączenie. Połączenie może być zamknięte zarówno przez klienta, jak i przez serwer, w dowolnej chwili.



Kolejność działań jest następująca:

1. Serwer używa metody *Open*, otwierając port TCP,
2. Klient używa metody *Open*, aby połączyć się z serwerem,
3. Serwer i klient generują zdarzenie *Connect*, informujące o udanym połączeniu,
4. Klient (albo serwer) wysyła komunikat, używając metody *Send*,
5. Serwer (albo klient) generuje zdarzenie *Read*, informujące o odebraniu komunikatu,
6. Klient (albo serwer) używa metody *Close* aby zakończyć połączenie
7. Serwer i klient generują zdarzenie *Disconnect*, informujące o rozłączeniu,

Komponenty pomocnicze *TcpSocks*

TcpLogBox

Wyświetla zapis komunikacji, używając różnych kolorów i symboli do wyróżnienia komunikatów różnych typów. Wewnętrznie używa komponentu *RichTextBox*.

Do wyświetlania komunikatów służą dedykowane metody.

Metody:

- **AddError** – wyświetla komunikat o błędzie,
- **AddSystem** – wyświetla komunikat systemowy,
- **AddTx** – wyświetla wysłaną wiadomość,
- **AddRx** – wyświetla odebraną wiadomość,
- **AddPrv** – wyświetla odebraną wiadomość prywatną,
- **Add** – wyświetla dowolny komunikat.

Metody *AddError*, *AddSystem*, *AddTx*, *AddRx* oraz *AddPrv* działają podobnie, ale wyświetlają komunikaty w różnych kolorach i dodają różne symbole, np. *AddError* wyświetla komunikaty w kolorze *Maroon* i dodaje znaki ****** na początku i na końcu komunikatu. Wszystkie te metody mają te same dwa argumenty:

- String **format** – treść lub łańcuch formatujący treści,
- Object[] **data** – zmienne do wstawienia do łańcucha formatującego (opcjonalnie).

Metoda *Add* wyświetla komunikat we wskazanym kolorze i nie dodaje do komunikatu żadnych symboli (ale można to zrobić samemu, korzystając z łańcucha formatującego).

Metoda ta ma trzy argumenty:

- Color **color** – kolor tekstu,
- String **format** – treść lub łańcuch formatujący treści,
- Object[] **data** – zmienne do wstawienia do łańcucha formatującego (opcjonalnie).

Z wszystkich tych metod można korzystać na dwa sposoby:

- Wyświetlenie komunikatu o stałej treści:
`tcpLog.AddSystem("Connected");`
- Wyświetlenie komunikatu z wartościami zmiennych – dla każdej zmiennej należy umieścić w łańcuchu formatującym jej indeks, w nawiasach klamrowych:

```
tcpLog.AddError("Error {0}", code);  
tcpLog.AddPrv("{0}): {1}", user, message);
```

W przypadku metody *Add* należy dodać kolor, jako pierwszy argument, oraz opcjonalnie w łańcuchu formatującym symbole, ułatwiające odróżnienie komunikatów różnego typu:

```
tcpLog.Add(Color.Gray, "-- Connected --");  
tcpLog.Add(Color.Red, "## Error {0} ##", code);  
tcpLog.Add(Color.Navy, "<- ({0}): {1}", user, message);
```

TcpMessageSplitter

Dzieli komunikat na części, ułatwiając jego interpretację, opcjonalnie zmieniając wielkość liter (na małe albo duże, do wyboru) i konwertując tekst na liczby. Jest szczególnie przydatny w aplikacji serwera, gdzie ułatwia analizę i sprawdzanie formalnej poprawności komunikatów odbieranych z aplikacji klienckich, ale może się też przydać po stronie klienta.

Konstruktor *TcpMessageSplitter* ma jeden, opcjonalny argument, *TcpMessageCase*, oznaczający czy i w jaki sposób będzie zmieniana wielkość liter w poleceniach i argumentach poleceń. Na potrzeby ćwiczenia należy użyć wartości *TcpMessageCase.Lower*:

```
TcpMessageSplitter splitter =  
    new TcpMessageSplitter(TcpMessageCase.Lower);
```

Metoda *Split* dzieli komunikat na fragmenty: pierwsze słowo komunikatu to polecenie, *Cmd* (ang. *command*), każde kolejne słowo to n-ty argument, *Arg* (ang. *argument*), natomiast cały tekst od n-tego słowa to reszta, *Rem* (ang. *remainder*). Argumenty *Arg* są typu *String*, ale są również dostępne po konwersji na liczby całkowite *Int32* jako *Num* (ang. *number*).

Argumenty *Arg*, liczby *Num* oraz reszty *Rem* są indeksowane, zaczynając od 1 (nie od zera!). Jeżeli zostanie użyty index wyższy, niż liczba słów w komunikacie, to zwracany jest pusty łańcuch lub wartość zero. W resztach *Rem* nie ma zmiany wielkości liter, to dotyczy tylko polecenia *Cmd* i argumentów *Arg*.

Dla przykładu, komunikat

```
ERR 10061 TCP error
```

zostanie podzielony w następujący sposób:

```
splitter.Cmd: "err"  
splitter.Num[1]: 10061  
splitter.Rem[2]: "TCP error"
```

Podobnie komunikat

```
prv user Private message
```

zostanie podzielony w następujący sposób:

```
splitter.Cmd: "prv"  
splitter.Arg[1]: "user"  
splitter.Rem[2]: "Private message"
```

Podział komunikatu na części umożliwia m.in. różne formatowanie komunikatów różnych typów. Wybrane komunikaty można sformatować według własnych preferencji, wszystkie pozostałe będą wyświetlane w sekcji *default*, np.:

```
splitter.Split(msg);  
switch (splitter.Cmd)  
{  
    case "prv":  
        tcpLogBox.AddPrv("[{0}]: {1}", splitter.Arg[1], splitter.Rem[2]);  
        break;  
  
    default:  
        tcpLogBox.AddSystem(msg);  
        break;  
}
```

Protokół SIC

Protokół SIC (ang. *Simple Internet Communicator*), opracowany na potrzeby ćwiczeń przez autora tego opracowania, umożliwia wymianę wiadomości pomiędzy klientami SIC, za pośrednictwem serwera SIC. Wykorzystuje protokół TCP w warstwie transportowej stosu IP. Główną funkcją protokołu jest prowadzenie rozmów, publicznych (widocznych dla wszystkich) i prywatnych (widocznych tylko dla nadawcy i adresata). Użytkownicy są identyfikowani przez nazwy, które sami wybierają.

Komunikaty SIC zawsze zaczynają się od polecenia (np. `name`), po którym należy podać odpowiednie do polecenia argumenty (np. `name adam`). Polecenie może być zastąpione przez równoważny mu alias (np. zamiast `name adam` można użyć aliasów `nick adam` albo `&adam`). Polecenie i kolejne argumenty powinny być rozdzielone spacjami, jedynie dla aliasów w postaci znaków specjalnych (np. `&`) odstęp jest opcjonalny. Jeżeli w poleceniu wystąpi błąd, serwer w odpowiedzi wyśle komunikat z numerem i opisem błędu.

Komunikaty klienta

- `name <user-name>` – wybór nazwy użytkownika, aliasy `nick`, `&`
- `bro <message>` – wysłanie wiadomości do wszystkich, aliasy `send`, `*`
- `prv <user-name> <message>` – wysłanie wiadomości prywatnej, aliasy `@`, `/`
- `ban <user-name> <minutes>` – zablokowanie użytkownika, alias `$`
- `bye` – rozłączenie z serwerem, alias `!`

Komunikaty serwera

- `name <user-name>` – potwierdzenie wyboru nazwy użytkownika
- `users <user-name> <user-name>` – aktualna lista użytkowników
- `user <user-name>` – informacja o dołączeniu nowego użytkownika
- `bro <user-name> <message>` – wiadomość do wszystkich, ze wskazaniem autora
- `prv <user-name> <message>` – wiadomość prywatna, ze wskazaniem autora
- `ban <details>` – informacja o zablokowaniu (do zablokowanego)
- `baned <details>` – informacja o zablokowaniu (do pozostałych)
- `bye <user-name>` – informacja o rozłączeniu użytkownika
- `err <code> <description>` – informacja o błędzie

Niezależnie od tego, czy klient użyje polecenia czy jego aliasu, serwer zawsze wysła komunikat zawierający podstawową formę polecenia

Błędy

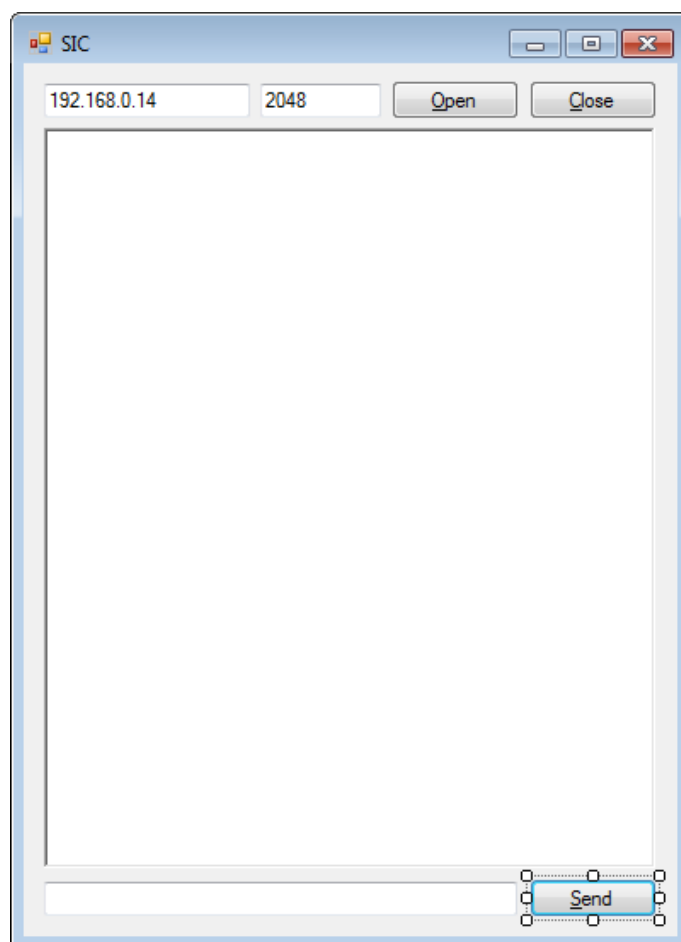
- 1 – nieznane polecenie
- 2 – użytkownik nie przedstawił się
- 3 – nazwa użytkownika jest zajęta
- 4 – użytkownik już się przedstawił
- 5 – nie ma takiego użytkownika
- 6 – użytkownik zablokowany
- 7 – niepoprawny czas zablokowania

Zadania

Celem ćwiczenia jest opracowanie programu komunikatora internetowego, umożliwiającego prowadzenie publicznych i prywatnych rozmów za pośrednictwem serwera SIC, opcjonalnie wzbogaconego przez rozpoznawanie i formatowanie odbieranych komunikatów.

1. Okno aplikacji

- Utworzyć nowy projekt, język C#, typ projektu Windows Forms;
- Umieścić na formularzu komponenty wg wzoru na rysunku poniżej: pola tekstowe na adres IP i numer portu, przyciski *Open* i *Close*, następnie *TcpLogBox*, u dołu pole tekstowe na komunikaty i przycisk *Send* oraz *TcpClientSock* (ten ostatni nie ma widocznej reprezentacji, zatem lokuje się na liście pod formularzem, jak *Timer*);
- Ustawić właściwości okna i komponentów: (1) okno powinno mieć rozmiar stały albo zmieniany tylko w ograniczonym zakresie – wówczas komponenty powinny mieć adekwatnie zmienione zakotwiczenie, (2) okno powinno mieć odpowiedni tytuł, przyciski powinny mieć skróty klawiaturowe, a przycisk *Send* powinien być domyślny dla formularza, (3) pola tekstowe powinny zawierać adres IP i numer portu serwera (poda je prowadzący ćwiczenia), dzięki czemu nie trzeba będzie ich wpisywać przy każdym uruchomieniu aplikacji.



2. Połączenie z serwerem

- Dodać obsługę kliknięcia przycisku *Open* – należy wywołać metodę *Open* komponentu *TcpClientSock*, podając jako argumenty adres IP serwera i numer portu; **Ważne!** Adres IP można podać w postaci łańcucha znaków, wprost z pola tekstowego, ale numer portu musi być liczbą całkowitą, należy zatem użyć metody *Int32.Parse*.
- Dodać obsługę kliknięcia przycisku *Close* – należy wywołać metodę *Close* komponentu *TcpClientSock*, bez argumentów;
- Dodać funkcje obsługi zdarzeń *Connect*, *Disconnect* i *Error* komponentu *TcpClientSock*. Należy do komponentu *TcpLogBox* dodać odpowiedni komunikat, najlepiej używając metody *AddSystem* dla zdarzeń *Connect* i *Disconnect* oraz *AddError* dla zdarzenia *Error*;
- Sprawdzić możliwość połączenia i rozłączenia z serwerem i reakcję na te zdarzenia. Należy również sprowokować błąd połączenia (np. zmieniając adres lub numer portu) i sprawdzić reakcję programu na taką sytuację.

3. Komunikacja

- Dodać obsługę kliknięcia przycisku *Send* – należy wykonać następujące czynności: (1) wywołać metodę *Send* komponentu *TcpClientSock*, podając jako argument treść komunikatu – tekst z pola tekstowego (2) do komponentu *TcpLogBox* dodać komunikat, najlepiej używając metody *AddTx*, jako argument podając ponownie treść wysłanego komunikatu (3) wyczyścić pole tekstowe, do czego służy metoda *Clear* oraz (4) aktywować („sfokusować”) pole tekstowe, do czego służy metoda *Focus*; **Ważne!** Do komunikacji wystarczy wykonanie pierwszych dwóch czynności, ale pozostałe dwie ułatwią rozmowę – użytkownik może wpisać treść, nacisnąć <Enter> żeby ją wysłać, po czym natychmiast może pisać treść kolejnego komunikatu itd.
- Dodać funkcję obsługi zdarzenia *Read* komponentu *TcpClientSock*. Należy do komponentu *TcpLogBox* dodać komunikat, najlepiej używając metody *AddRx*, jako argument podając treść odebranego komunikatu, przekazany do zdarzenia jako *msg*;
- Sprawdzić możliwość komunikacji z innymi uczestnikami ćwiczeń.

4. * Ograniczenie dostępu do przycisków i pól tekstowych

- Korzystając z edytora właściwości komponentów, zmienić właściwość *Enabled* przycisków *Close* i *Send* oraz pola tekstowego komunikatu na *false*;
- Zmienić funkcje obsługi zdarzeń *Connect* i *Disconnect* komponentu *TcpClientSock* w taki sposób, aby dostępne były tylko te przyciski i pola tekstowe, które chwili są potrzebne, np. po połączeniu z serwerem przyciski *Close* i *Send* powinny być dostępne, natomiast *Open* – nie; **Ważne!** Nie należy zmieniać funkcji obsługi kliknięcia przycisków. Np. kliknięcie przycisku *Close* zamyka połączenie, ale nie jest to jedyna możliwa przyczyna zamknięcia połączenia; Jeżeli zmiana stanu przycisków zostanie wpisana do zdarzenia *Disconnect*, to zadziała zawsze, niezależnie od powodu zamknięcia połączenia.

5. * Formatowanie odbieranych komunikatów

- W przestrzeni między funkcjami obsługi zdarzeń, najlepiej na początku klasy *Form1*, dodać deklarację i utworzenie obiektu *TcpMessageSplitter*;
- Zmodyfikować funkcję obsługi zdarzenia *Read*: podzielić odebrany komunikat na części i zależnie od polecenia *Cmd* wyświetlić komunikat w odpowiednim formacie. Można użyć funkcji dedykowanych (np. *AddPrv*) albo uniwersalnej funkcji *Add*. **Ważne!** Należy użyć instrukcji *switch-case* z sekcją *default* – poszczególne sekcje *case* wyświetlą wybrane typy komunikatów, a sekcja *default* wszystkie pozostałe.