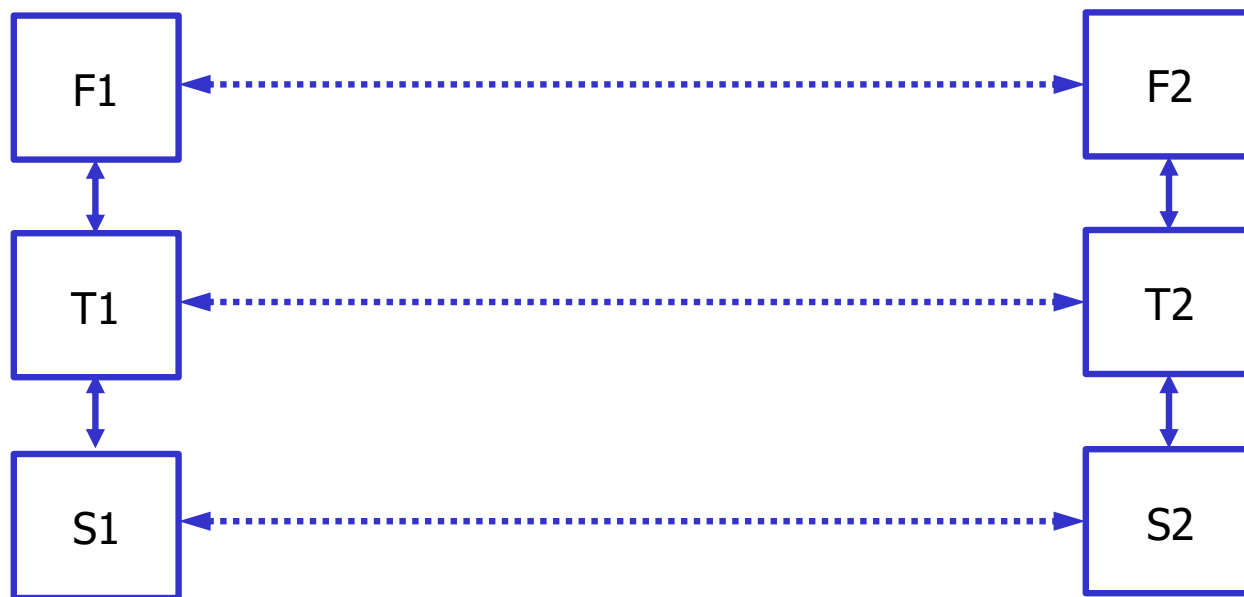


Metodyka programowania

Komunikator

Warstwowy model komunikacji

Współczesne systemy telekomunikacyjne są projektowane i budowane w oparciu o warstwowy model komunikacji

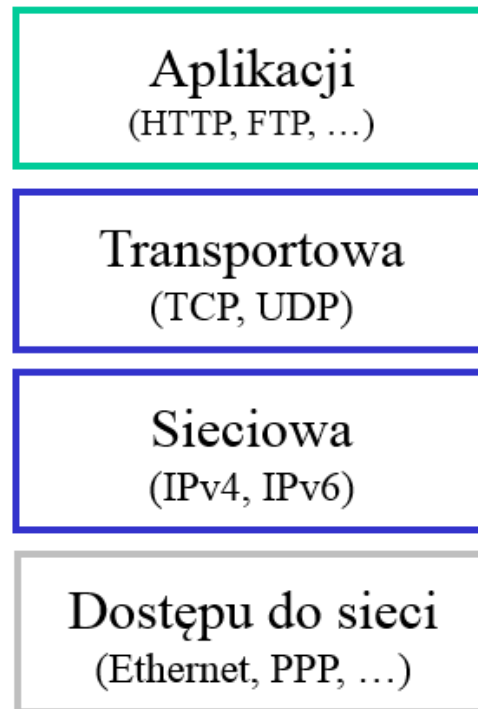


Warstwowy model komunikacji,
oznaczenia: F – filozof, T – tłumacz, S - sekretarz

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.

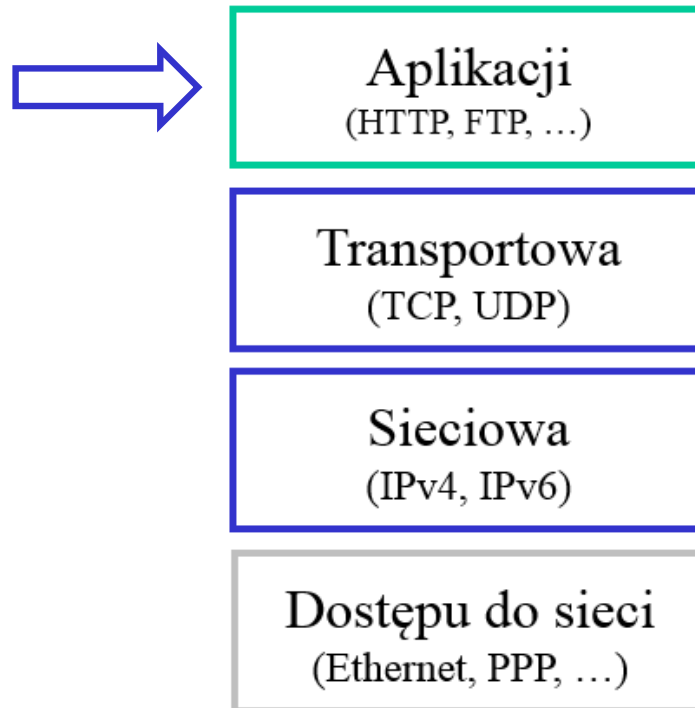
Model DoD definiuje 4 warstwy sieci ich zadania.



Model DoD

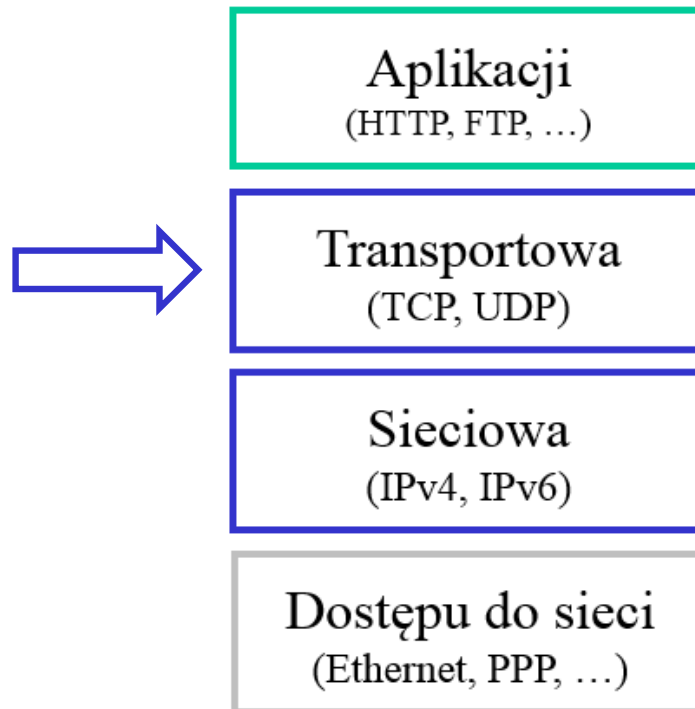
- Warstwa aplikacji

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.



Model DoD

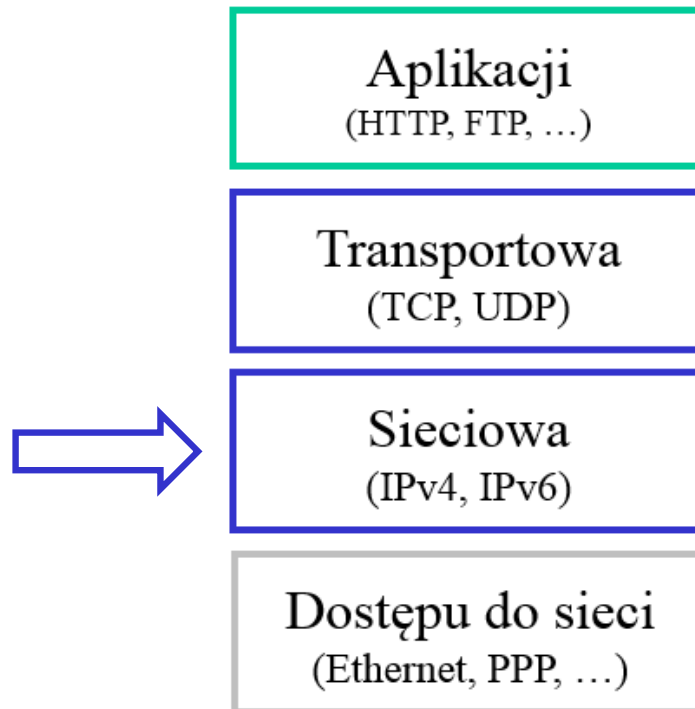
- Warstwa transportowa
Zapewnia poprawność i kompletność danych przesyłanych między hostami oraz kieruje dane do właściwych aplikacji. Najczęściej używane protokoły transportowe to TCP i UDP, ale są też inne, np. QUIC. Adresem w tej warstwie jest numer portu.



Model DoD

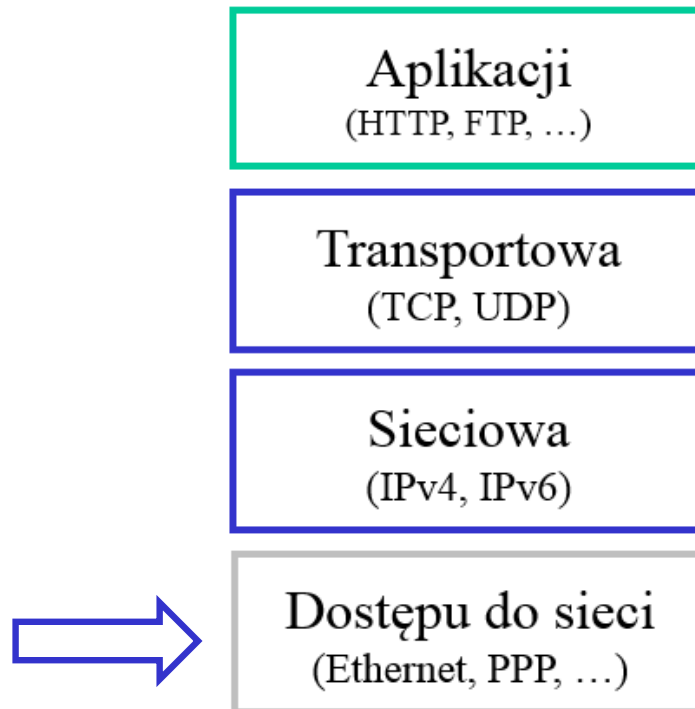
- Warstwa sieciowa

Kluczowa warstwa Internetu, odpowiedzialna za znajdowanie drogi (ang. route) między hostami. Wykorzystuje protokół IP do przysyłania danych, a wewnątrz wiele różnych protokołów (np. RIP, OSPF, BGP), dzięki którym routery znajdują drogę dla pakietów IP. Adresem w tej warstwie jest adres IP.

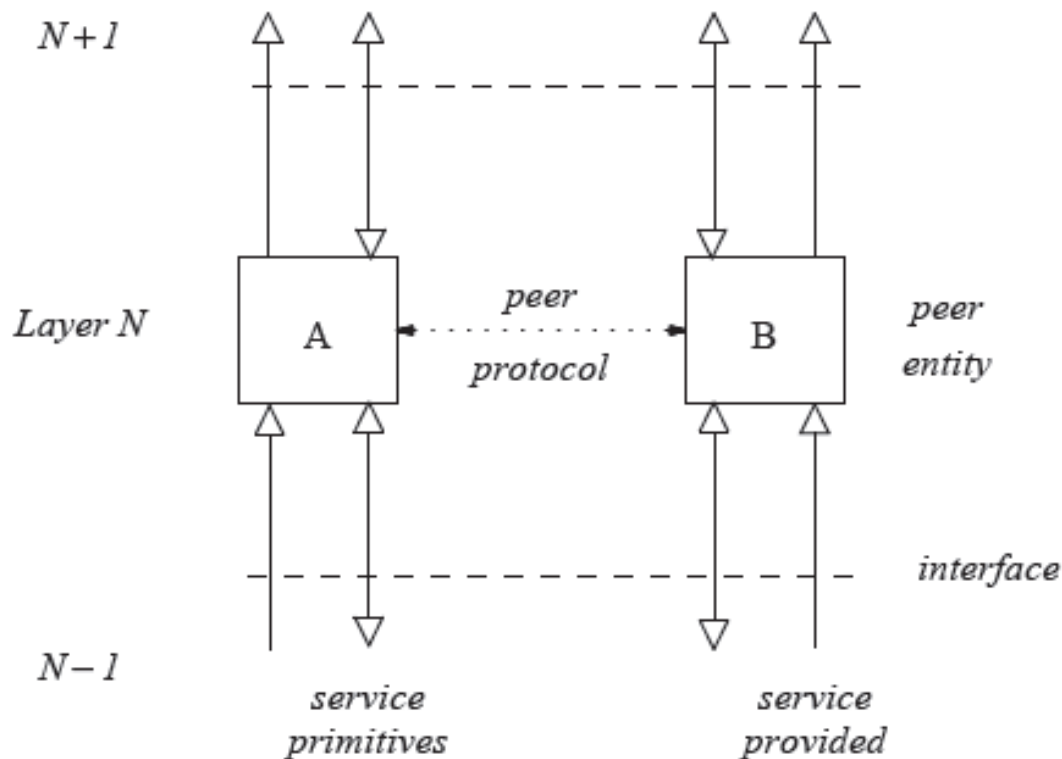


Model DoD

- 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.



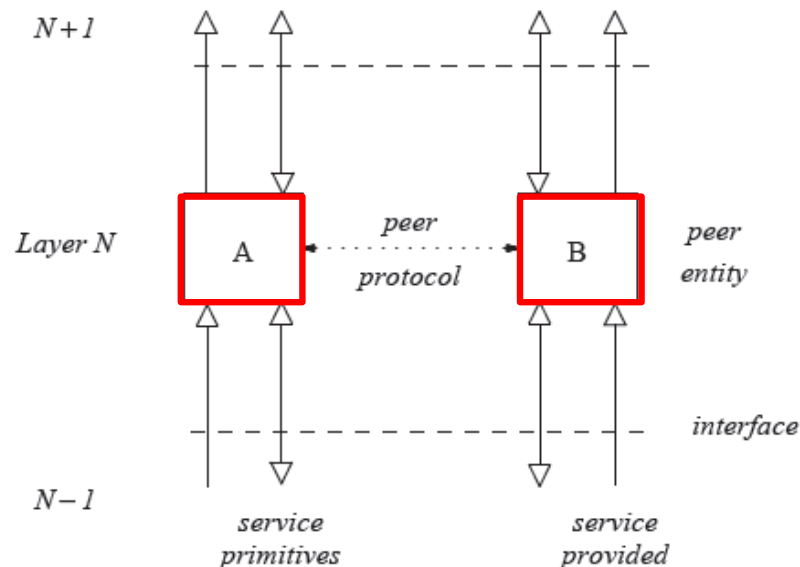
Komunikacja w modelu warstwowym



Komunikacja w modelu warstwowym i usługi elementarne

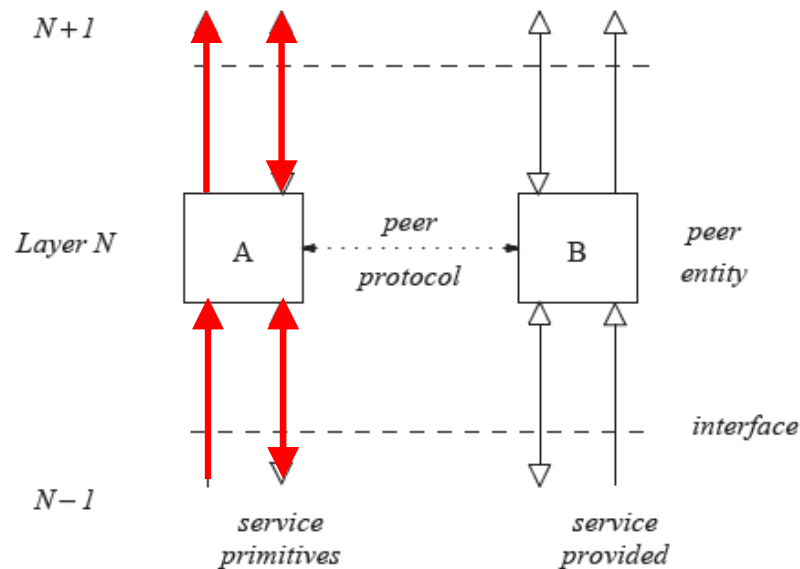
Komunikacja w modelu warstwowym

- 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).



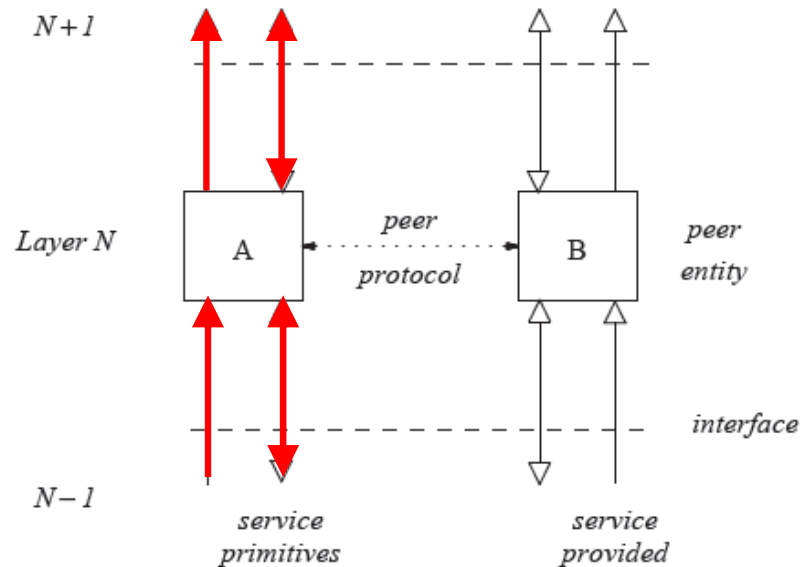
Komunikacja w modelu warstwowym

- 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.



Komunikacja w modelu warstwowym

- 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.



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.

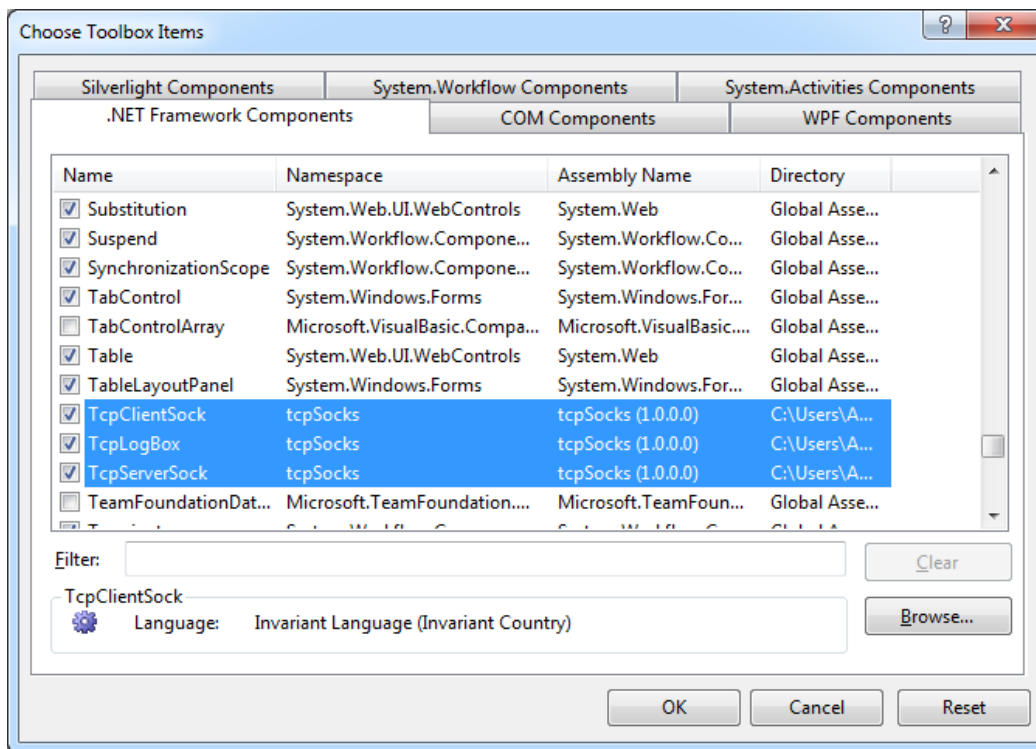
Protokół TCP

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

- **Open** – otwarcie na połączenia klientów (serwer) albo połączenie z serwerem (klient),
- **Connect** – informacja o nawiązaniu połączenia,
- **Send** – wysłanie danych,
- **Read** – informacja o odebraniu danych,
- **Close** – zamknięcie połączenia,
- **Disconnect** – informacja o zamknięciu połączenia,
- **Error** – informacja o błędzie.

Pakiet TcpSocks

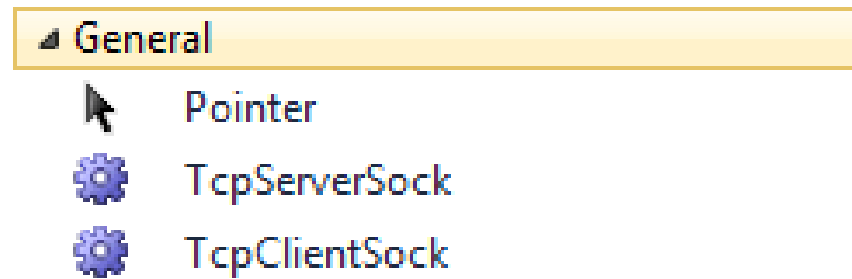
Pakiet TcpSocks zawiera komponenty do komunikacji TCP (serwer i klient) oraz pomocnicze (okienko logu i splitter). Nie ma wersji instalacyjnej, należy go dodać do przybornika.



Komponenty internetowe TcpSocks

Komponenty **zarządzające** połączeniami TCP/IP

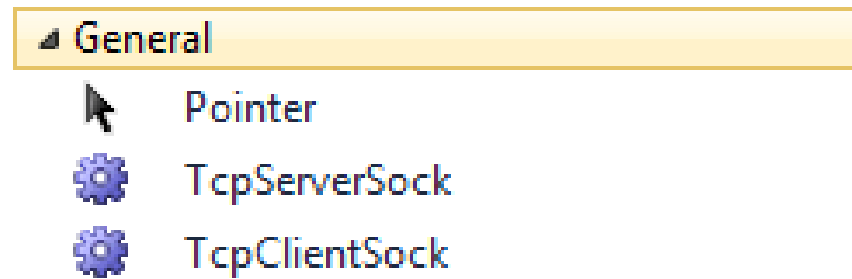
- Wykorzystują technologię Windows Socket, będącą implementacją technologii Berkeley Socket:
 - TcpClientSock
 - TcpServerSock
- Udostępniają wszystkie pasywne i aktywne usługi elementarne protokołu TCP w postaci metod i zdarzeń



Komponenty internetowe TcpSocks

Komponenty są dedykowane do komunikacji tekstowej

- Konwertują wysyłaną wiadomość typu String na strumień bajtów, a po drugiej stronie łącza konwertują strumień bajtów z powrotem na typ String.
- Zapewniają, że każda wiadomość wysłana jednorazowo metodą Send zostanie w całości przekazana po drugiej stronie łącza 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

TcpClientSock – 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,

TcpClientSock – zdarzenia

Connect – informuje o połączeniu z serwerem
(na skutek wywołania metody **Open**)

- Connect(object sender, TcpSockThread Sock)
 - sender – komponent zgłaszający zdarzenie
 - Sock – gniazdko TCP/IP, przez który można wysłać wiadomość

Disconnect – informuje o rozłączeniu z serwerem (np. na skutek wywołania metody **Close**)

TcpClientSock – zdarzenia

Error – informuje o wystąpieniu błędu
(np. niemożność połączenia, zerwanie połączenia, ...)

- `Error(object sender, TcpSockThread Sock, int code)`
 - sender – komponent zgłaszający zdarzenie
 - Sock – gniazdko TCP/IP
 - code – numer błędu TCP/IP

TcpClientSock – zdarzenia

Read – informuje o odebraniu wiadomości

- Read(object sender, TcpSocketThread Sock, string msg)
 - sender – komponent zgłaszający zdarzenie
 - Sock – gniazdko TCP/IP , przez który można wysłać odpowiedź
 - msg – odebrana wiadomość

TcpServerSock – właściwości

IsOpen – (Boolean) czy port serwera jest otwarty

Socks – (TcpSockThread[]) lista aktualnie połączonych gniazdek klienckich

TcpServerSock – metody

Open – otwiera port TCP, argumenty:

- String **Address** – adres IP serwera,
- Int32 **Port** – numer portu TCP serwera

Close – zamyka port TCP (również trwające połączenia)

Send – wysyła wiadomość, ma trzy przeciążone warianty:

- Send(String **Msg**)
- Send(String **Msg**, TcpSockThread Except)
- Send(String **Msg**, SelectSock isValid)

Argumenty

- **Msg** – wiadomość do wysłania,
- Except – gniazdko, do którego wiadomość nie zostanie wysłana
- isValid – funkcja decydująca, czy wiadomość ma być wysłana

TcpServerSock – zdarzenia

Connect – informuje o połączeniu z klientem
(na skutek wywołania metody **Open**)

- Connect(object sender, TcpSockThread Sock)
 - sender – komponent zgłaszający zdarzenie
 - Sock – gniazdko TCP/IP, przez który można wysłać wiadomość

Disconnect – informuje o rozłączeniu z klientem (np. na skutek wywołania metody **Close**)

TcpServerSock – zdarzenia

Error – informuje o wystąpieniu błędu
(np. brak możliwości otwarcia portu)

- `Error(object sender, TcpSockThread Sock, int code)`
 - `sender` – komponent zgłaszający zdarzenie
 - `Sock` – gniazdko TCP/IP klienta (może być null)
 - `code` – numer błędu TCP/IP

TcpServerSock – zdarzenia

Read – informuje o odebraniu wiadomości

- Read(object sender, TcpSockThread Sock, string msg)
 - sender – komponent zgłaszający zdarzenie
 - Sock – gniazdko TCP/IP, przez który można wysłać odpowiedź
 - msg – odebrana wiadomość

TcpSocketThread

Reprezentuje zakończenie połączenia Windows Socket w komponentach TcpClientSock i TcpServerSock.

Obiekt klasy TcpSocketThread jest przekazywany jako jeden z argumentów (Sock) do funkcji obsługi zdarzeń, m.in. Connect i Read.

TcpSocketThread

Właściwości:

- **LocalIP**
- **LocalPort**
- **RemoteIp**
- **RemotePort**
- **Data** – (Object) dowolne dane, które można "podczepić" pod gniazdko i później wykorzystywać;
Przy odczycie wymagane rzutowanie

TcpSocketThread

Metody:

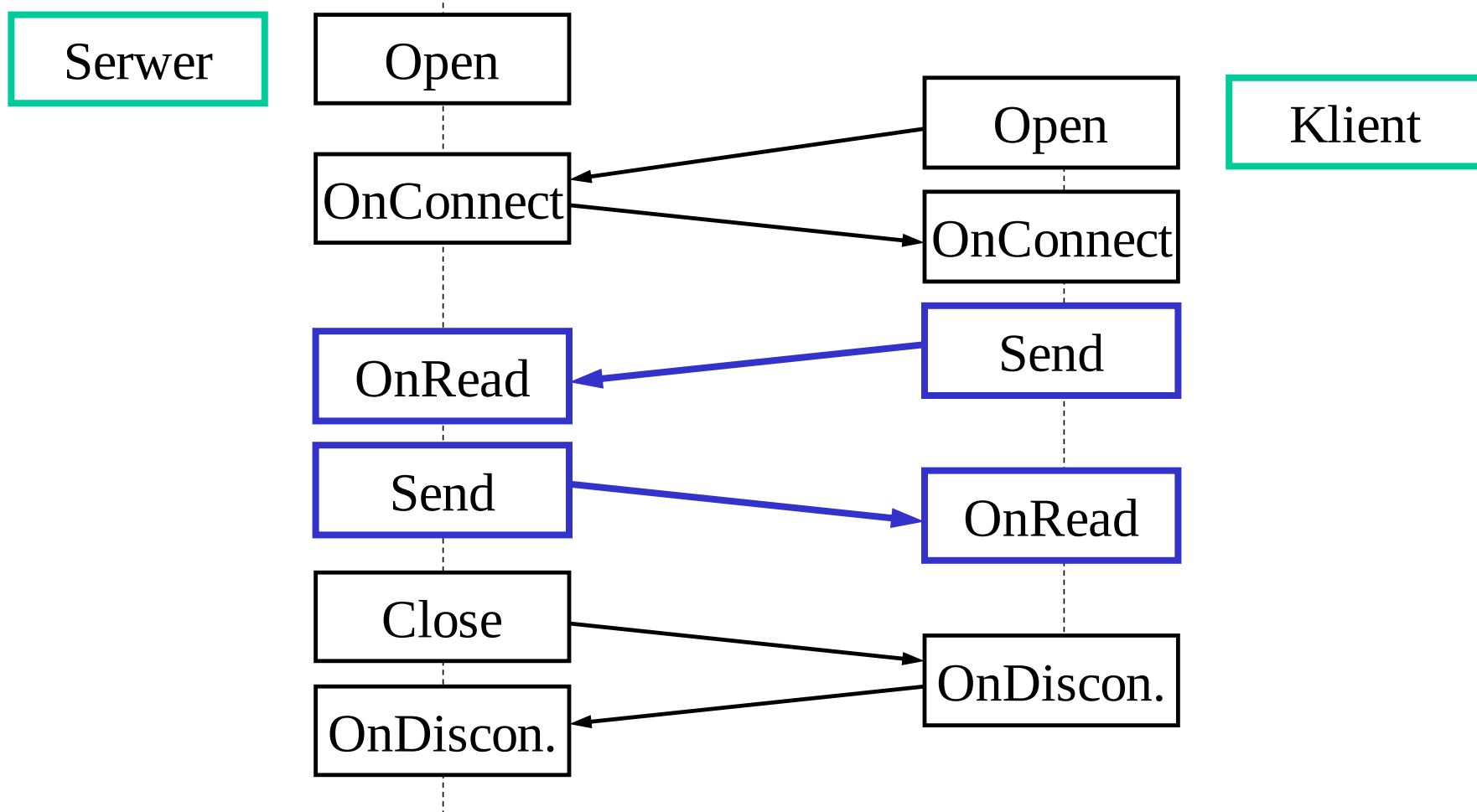
- **Close** – zamyka połączenie
- **Send** – wysyła wiadomość

Kolejność czynności

Zasady

- 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 czynności

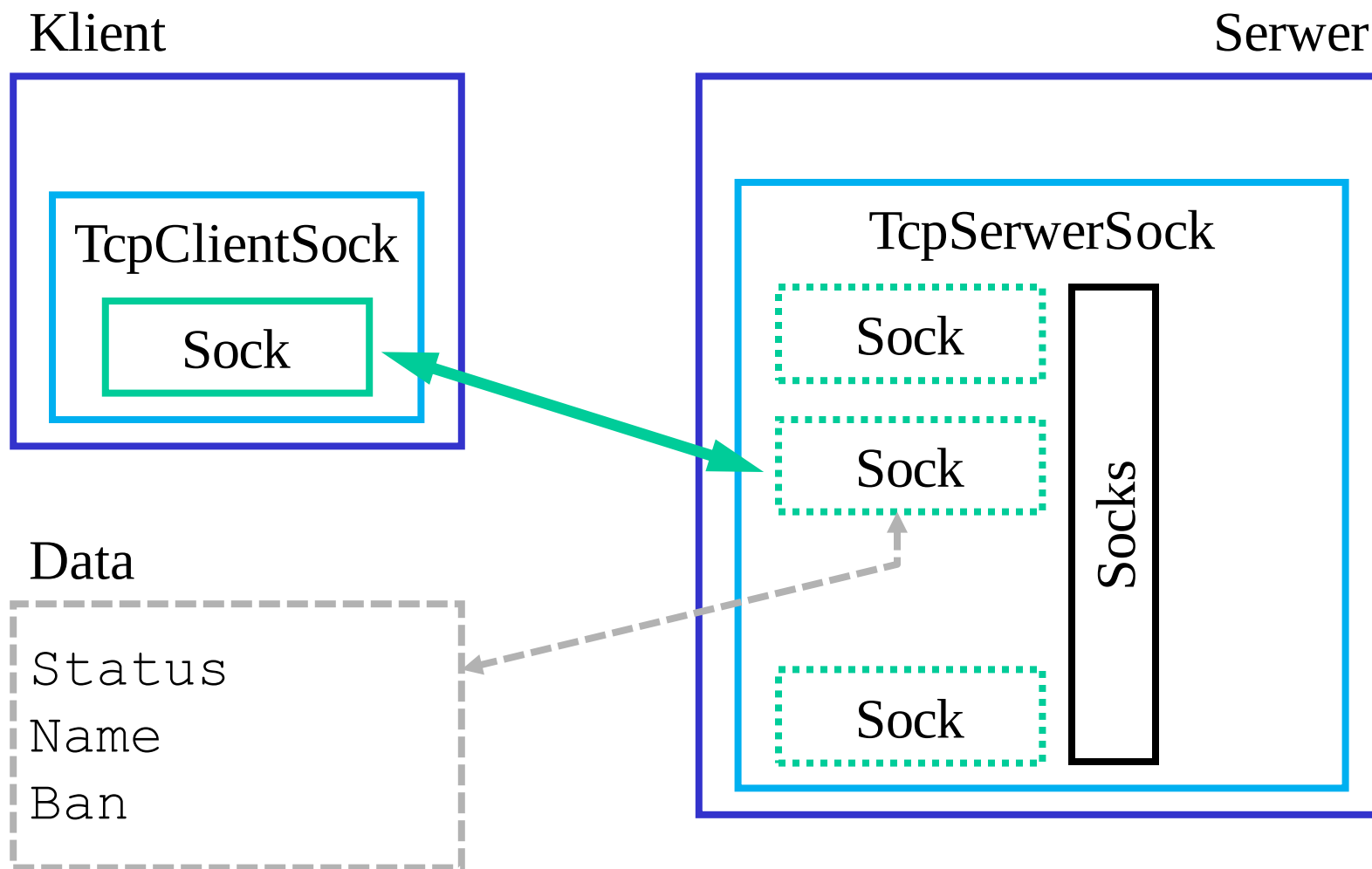


Kolejność czynności

Kolejność czynności

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

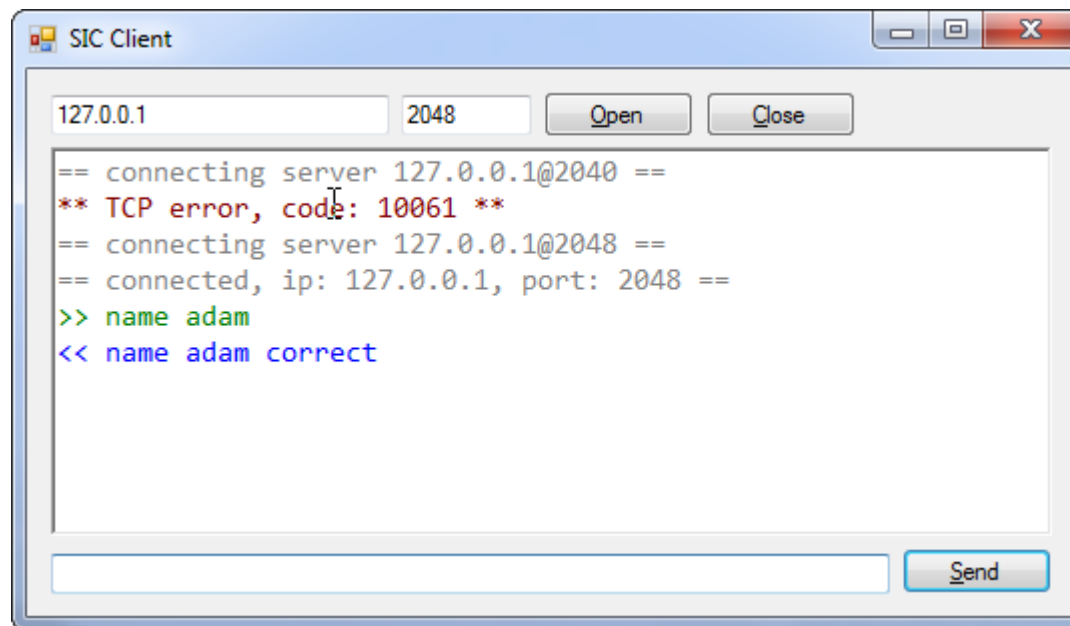
Struktura połączenia



TcpLogBox

TcpLogBox

- Wyświetla zapis komunikacji, używając różnych kolorów i symboli do wyróżnienia komunikatów różnych typów. Wewnętrznie używa komponentu RichTextBox.
- Do wyświetlania komunikatów służą dedykowane metody



TcpLogBox

TcpLogBox, 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.

TcpLogBox

Metody AddError, AddSystem, AddTx, AddRx i AddPriv

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

TcpLogBox

Metoda Add

- Wyświetla komunikat we wskazanym kolorze i nie dodaje do komunikatu żadnych symboli (ale można to zrobić samemu, korzystając z łańcucha formatującego).
- Ma trzy argumenty:
 - Color **color** – kolor tekstu,
 - String **format** – treść lub łańcuch formatujący treści,
 - Object[] **data** – zmienne do wstawienia do łańcucha formatującego (opcjonalnie).

TcpLogBox

Ze 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);
```

TcpLogBox

- 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, "## Err {0} ##", code);  
tcpLog.Add(Color.Navy, "<- {0}", message);
```

TcpMessageSplitter

TcpMessageSplitter

- Dzieli komunikat na części, ułatwiając jego interpretację, opcjonalnie zmieniając wielkość liter (na małe albo duże, do wyboru) i konwertując tekst na liczby.

TcpMessageSplitter

Konstruktor

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

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

TcpMessageSplitter

Metoda Split

- Dzieli komunikat na fragmenty:
 - **Cmd** (ang. command), pierwsze słowo to polecenie,
 - **Arg** (ang. argument), każde kolejne słowo to n-ty argument,
 - **Rem** (ang. remainder) cały tekst od n-tego słowa to reszta, 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, to zwracany jest pusty łańcuch lub wartość zero (nie wystąpi błąd!).
- W resztach Rem nie ma zmiany wielkości liter, to dotyczy tylko polecenia Cmd i argumentów Arg.

TcpMessageSplitter

Metoda Split

- Dla przykładu, komunikat

```
ERR 10061 TCP error
```

zostanie podzielony w następujący sposób:

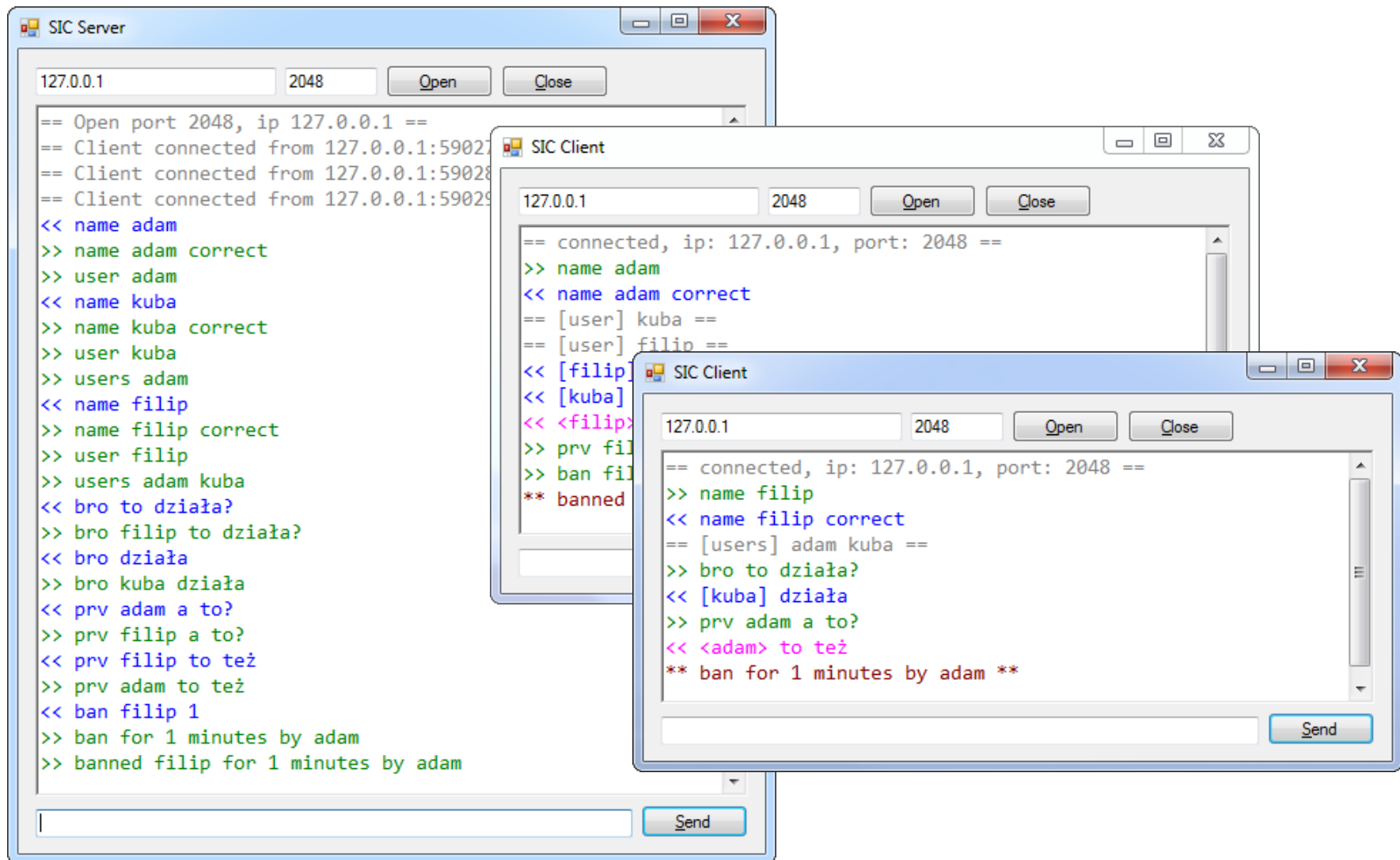
```
splitter.Cmd:      "err"  
splitter.Num[1]: 10061  
splitter.Rem[2]: "TCP error"
```

Protokół SIC

Protokół SIC (ang. Simple Internet Communicator)

- 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

Protokół SIC



The image displays three overlapping windows illustrating the SIC protocol. The top-left window is the 'SIC Server', the middle window is the 'SIC Client', and the bottom-right window is another 'SIC Client'.

SIC Server

```
127.0.0.1 2048 Open Close
== Open port 2048, ip 127.0.0.1 ==
== Client connected from 127.0.0.1:59028 ==
== Client connected from 127.0.0.1:59028 ==
== Client connected from 127.0.0.1:59028 ==
<< name adam
>> name adam correct
>> user adam
<< name kuba
>> name kuba correct
>> user kuba
>> users adam
<< name filip
>> name filip correct
>> user filip
>> users adam kuba
<< bro to działa?
>> bro filip to działa?
<< bro działa
>> bro kuba działa
<< prv adam a to?
>> prv filip a to?
<< prv filip to też
>> prv adam to też
<< ban filip 1
>> ban for 1 minutes by adam
>> banned filip for 1 minutes by adam
```

SIC Client

```
127.0.0.1 2048 Open Close
== connected, ip: 127.0.0.1, port: 2048 ==
>> name adam
<< name adam correct
== [user] kuba ==
== [user] filip ==
<< [filip]
<< [kuba]
<< <filip>
>> prv filip
>> ban filip
** banned
```

SIC Client

```
127.0.0.1 2048 Open Close
== connected, ip: 127.0.0.1, port: 2048 ==
>> name filip
<< name filip correct
== [users] adam kuba ==
>> bro to działa?
<< [kuba] działa
>> prv adam a to?
<< <adam> to też
** ban for 1 minutes by adam **
```

Protokół SIC

Komunikaty SIC

- Zawsze zaczynają się od polecenia (np. name), po którym należy podać odpowiednie do argumenty (np. name adam).
- Polecenie może być zastąpione przez równoważny mu alias (np. zamiast name adam można użyć 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.

Protokół SIC

Komunikaty klienta

- name <user-name> – wybór nazwy użytkownika
- bro <message> – wiadomość do wszystkich
- prv <user-name> <message> – wiadomość prywatna
- ban <user-name> <minutes> – zablokowanie użytkownika
- bye – rozłączenie z serwerem

Protokół SIC

Komunikaty klienta, aliasy

- name - nick, &
- bro - send, *
- prv - @, /
- ban - \$
- bye - !

Protokół SIC

Komunikaty serwera

- name <user-name> – potwierdzenie wyboru nazwy
- users <user-name> <user-name> – aktualna lista už.
- user <user-name> – informacja o dołączeniu nowego už.
- bro <user-name> <message> – wiadomość do wszystkich
- prv <user-name> <message> – wiadomość prywatna
- ban <details> – informacja o banie (do zablokowanego)
- baned <details> – informacja o banie (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 używa podstawowej formy polecenia