

---

# Transport

## część 2: protokół TCP

Sieci komputerowe

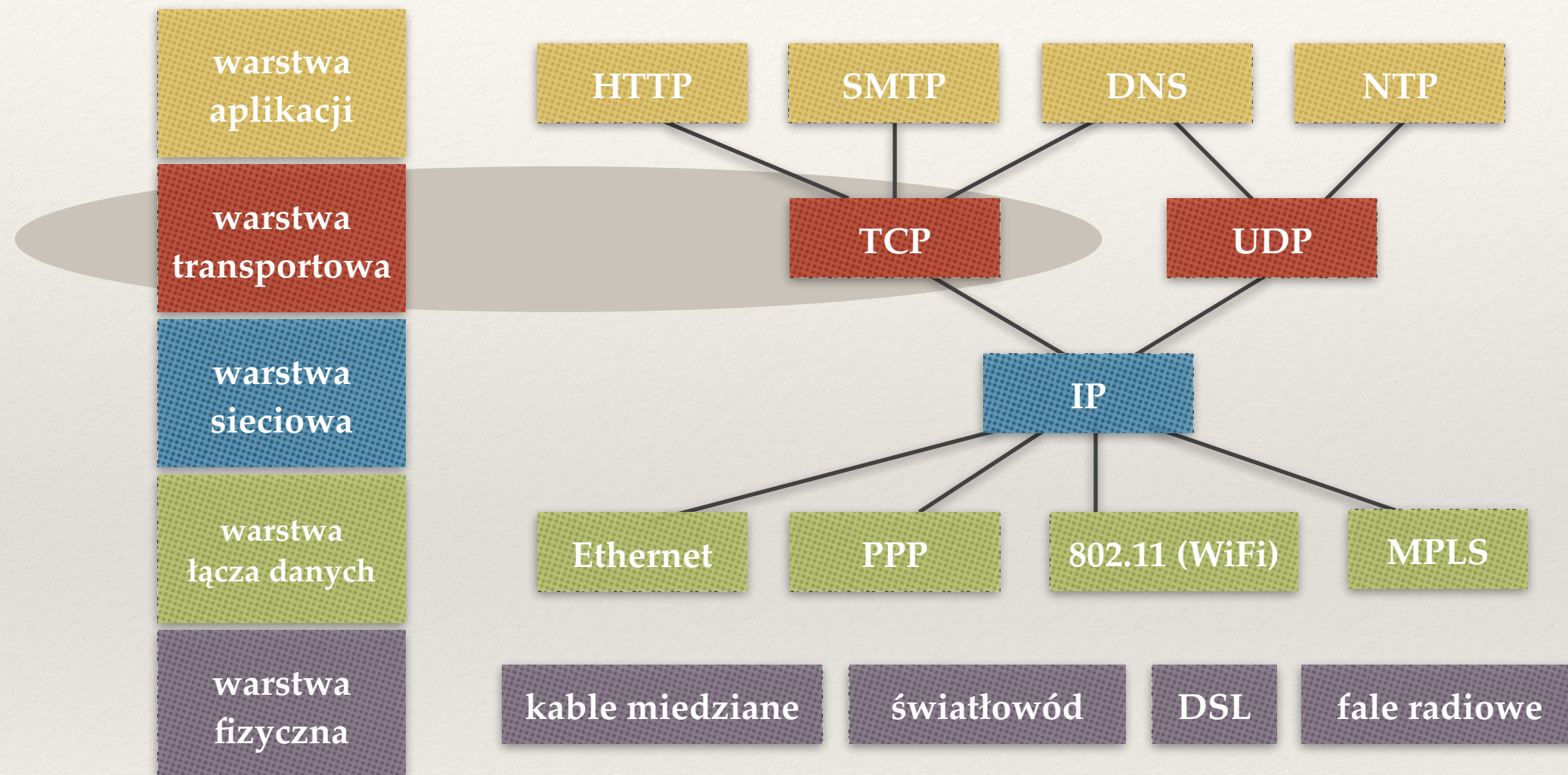
Wykład 7

---

*Marcin Bieńkowski*



# Protokoły w Internecie





# W poprzednim odcinku: niezawodny transport

---

- ❖ Segmentacja.
- ❖ Algorytmy niezawodnego dostarczania danych: stop-and-wait, okno przesuwne nadawcy, okno przesuwne odbiorcy.
- ❖ Potwierdzanie: go-back-N, selektywne, skumulowane.
- ❖ **Niezawodny transport w TCP:** okna przesuwne + potwierdzanie skumulowane.
- ❖ **Kontrola przepływu w TCP:** odbiorca wysyła rozmiar wolnego miejsca w oknie (tzw. okno oferowane) → reguluje rozmiar okna nadawcy.



# Dzisiaj

---

- ❖ Programowanie gniazd TCP.
- ❖ Implementacja TCP.



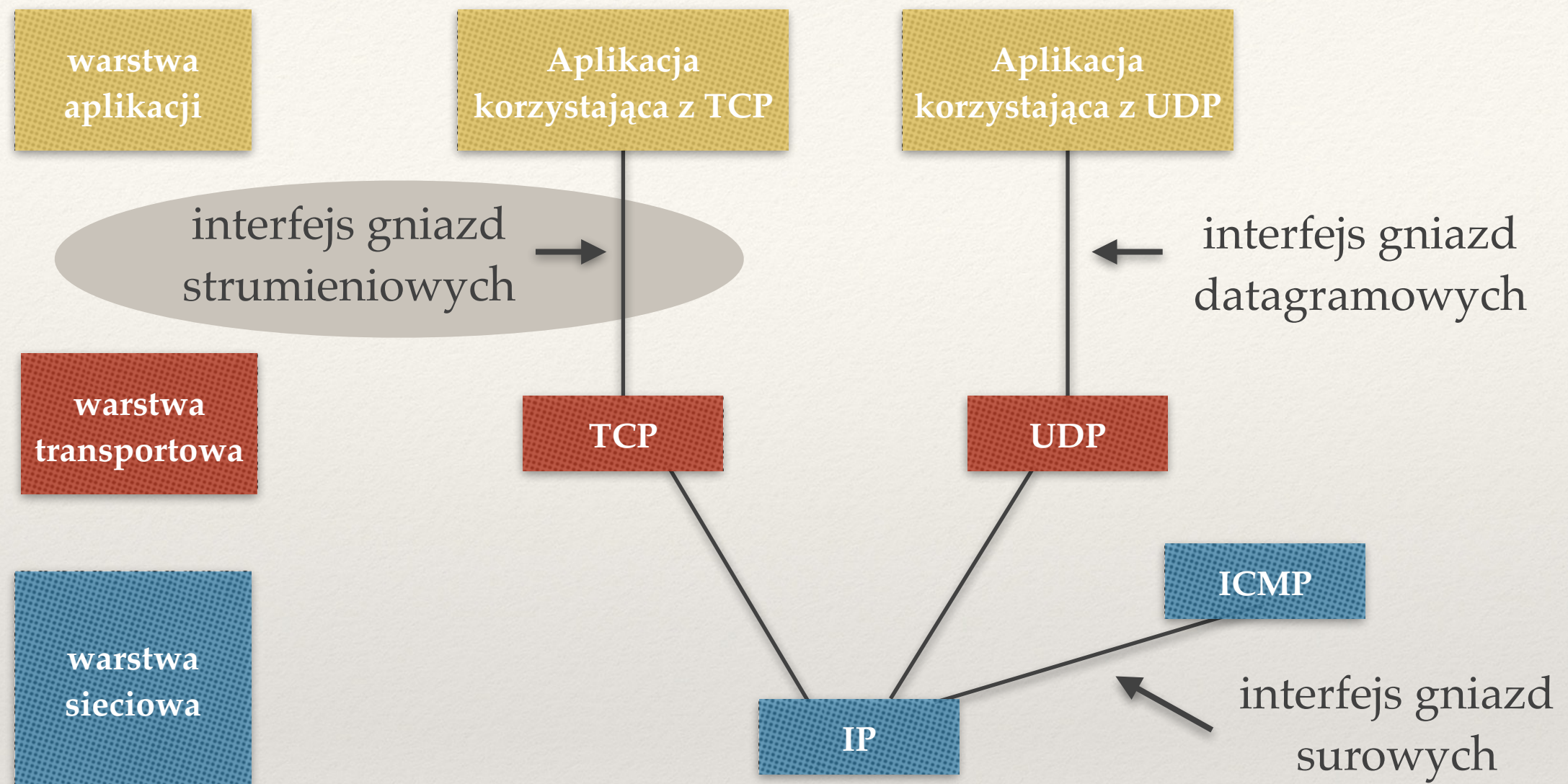
---

# Programowanie gniazd

---



# Interfejs programistyczny



- ❖ Interfejs programistyczny *BSD sockets*.
- ❖ Przystępne wprowadzenie: Beej's Guide to Network Programming.



# Komunikacja

---

## Komunikacja bezpołączeniowa

- ❖ Strony nie utrzymują stanu.
- ❖ Przykładowo: zwykła poczta.

## Komunikacja połączeniowa

- ❖ Na początku strony wymieniają komunikaty *nawiązujące połączenie*.
- ❖ Późniejsza komunikacja wygodniejsza niż w przypadku bezpołączeniowym.
- ❖ Na końcu trzeba *zakończyć połączenie*.
- ❖ Przykładowo: telefon.



# Gniazda UDP

---

- ❖ Gniazdo jest związane z konkretnym procesem.
- ❖ Gniazdo identyfikowane przez lokalny adres IP + lokalny port.
- ❖ Gniazdo nie posiada stanu.
- ❖ Gniazdo nie jest „połączone” z innym gniazdem.
- ❖ Nie ma różnicy między klientem i serwerem: po pierwszym wywołaniu `sendto( )` gniazdo klienta otrzymuje od jądra numer portu i zachowuje się identycznie jak gniazdo serwera.



# Gniazda TCP: dwa typy gniazd

---

## TCP: gniazda nasłuchujące

- ❖ Dla serwera, tylko do nawiązywania połączeń
- ❖ Tylko jedna strona gniazda (lokalna) ma przypisany adres:  
`172.16.16.14:80 — *:*`

## TCP: gniazda połączone

- ❖ Tworzone dla klienta i serwera po połączeniu, do wymiany właściwych danych.
  - ♦ Gniazdo serwera: `172.16.16.14:80 — 22.33.44.55:44444`
  - ♦ Gniazdo klienta: `22.33.44.55:44444 — 172.16.16.14:80`

TCP: gniazdo opisywane (między innymi) przez cztery elementy:  
lokalny IP, lokalny port, zdalny IP, zdalny port.

demonstracja



# Dobrze znane porty

---

Skąd wiemy, że powinniśmy się łączyć właśnie z portem 80?

- ❖ Dobrze znane porty (*well known ports*)
- ❖ Niektóre usługi mają porty zarezerwowane przez standardy:
  - ♦ 22 - port SSH,
  - ♦ 80 - port HTTP,
  - ♦ 443 - port HTTPS,
  - ♦ → `/etc/services`.



---

# Podstawowe funkcje

---



# Tworzenie gniazda TCP

---

```
#include <arpa/inet.h>  
int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
```



# Wiązanie gniazda z adresem i portem (serwer)

---

Struktura adresowa i `bind()` identycznie jak w UDP.

```
struct sockaddr_in server_address = {0};

server_address.sin_family      = AF_INET;
server_address.sin_port      = htons(32345);
server_address.sin_addr.s_addr = htonl(INADDR_ANY);

bind(
    sockfd,
    (struct sockaddr*)&server_address,
    sizeof(server_address)
);
```



# Oczekiwanie na nawiązanie połączenia (serwer)

---

- ❖ UDP: bezpośrednio po `bind( )` można odbierać i wysyłać dane.



# Oczekiwanie na nawiązanie połączenia (serwer)

---

- ❖ UDP: bezpośrednio po `bind( )` można odbierać i wysyłać dane.
- ❖ TDP: trzeba najpierw nawiązać połączenie.

- ✦ Tworzenie kolejki na nawiązane, ale nie obsłużone połączenia:

```
listen(sock_fd, backlog);
```

- ❖ Pobieranie nawiązanego połączenia z kolejki:

```
int connected_sock_fd = accept(sock_fd, NULL, NULL);
```



# Oczekiwanie na nawiązanie połączenia (serwer)

---

- ❖ UDP: bezpośrednio po `bind()` można odbierać i wysyłać dane.
- ❖ TDP: trzeba najpierw nawiązać połączenie.

- ✦ Tworzenie kolejki na nawiązane, ale nie obsłużone połączenia:

```
listen(sock_fd, backlog);
```

rozmiar kolejki (np. 64)



- ❖ Pobieranie nawiązanego połączenia z kolejki:

```
int connected_sock_fd = accept(sock_fd, NULL, NULL);
```



# Oczekiwanie na nawiązanie połączenia (serwer)

- ❖ UDP: bezpośrednio po `bind( )` można odbierać i wysyłać dane.
- ❖ TDP: trzeba najpierw nawiązać połączenie.

- ✦ Tworzenie kolejki na nawiązane, ale nie obsłużone połączenia:

```
listen(sock_fd, backlog);
```

rozmiar kolejki (np. 64)



- ❖ Pobieranie nawiązanego połączenia z kolejki:

```
int connected_sock_fd = accept(sock_fd, NULL, NULL);
```

gniazdo połączone z  
klientem



gniazdo do odbierania kolejnych  
połączeń przez `accept( )`





# Ogólna budowa serwera TCP

---

```
int sock_fd = socket(AF_INET, SOCK_STREAM, 0);  
// stworzenie i wypełnienie struktury server_address  
bind(sock_fd, (struct sockaddr*)&server_address,  
      sizeof(server_address));  
  
listen(sock_fd, 64)  
  
for (;;) {  
    int connected_sock_fd = accept(sock_fd, NULL, NULL);  
  
    // wysyłanie i odbieranie danych przez connected_sock_fd  
  
    close(connected_sock_fd);  
}  
  
close(sock_fd);
```



# Odbieranie danych z gniazda

---

```
u_int8_t buffer[BUFFER_SIZE+1];
```

```
ssize_t bytes_read = recv(  
    connected_sock_fd,  
    buffer,  
    BUFFER_SIZE,  
    0, ←  
);
```


miejsce na opcje,  
np. odczyt w trybie nieblokującym



# Odbieranie danych z gniazda

---

```
u_int8_t buffer[BUFFER_SIZE+1];
```

```
ssize_t bytes_read = recv(  
    connected_sock_fd,  
    buffer,  
    BUFFER_SIZE,  
    0,   
);
```

miejsce na opcje,  
np. odczyt w trybie nieblokującym

- ❖ W UDP potrzebowaliśmy odczytać również informacje o nadawcy.
- ❖ W TCP do odesłania danych wystarczy gniazdo `connected_sock_fd`.
- ❖ `recv(s,b,x,0) = read(s,b,x)`



# Wysyłanie danych przez gniazdo

---

```
u_int8_t buffer[...];  
  
ssize_t bytes_sent = send(  
    connected_sock_fd,  
    buffer,  
    reply_length,  
    0,  
);
```



# Wysyłanie danych przez gniazdo

---

```
u_int8_t buffer[...];
```

```
ssize_t bytes_sent = send(  
    connected_sock_fd,  
    buffer,  
    reply_length,  
    0,  
);
```

❖ `send(s,b,x,0) = write(s,b,x)`



# Demonstracja serwera TCP (1)

---

`tcp_server_echo.c + telnet/netcat`

kod serwera na stronie wykładu

demonstracja



# Nawiązywanie połączenia (klient)

---

- ❖ **UDP:** bezpośrednio po stworzeniu gniazda, klient może przez nie przesyłać dane.

Przy każdym wywołaniu `sendto ( )` trzeba podać strukturę adresową serwera.



# Nawiązywanie połączenia (klient)

---

- ❖ **UDP:** bezpośrednio po stworzeniu gniazda, klient może przez nie przesyłać dane.

Przy każdym wywołaniu `sendto ( )` trzeba podać strukturę adresową serwera.

- ❖ **TDP:** po stworzeniu gniazda po stronie klienta trzeba najpierw nawiązać połączenie z serwerem:

```
connect( sock_fd, ( struct sockaddr* ) &server_address,  
        sizeof(server_address) );
```

Późniejsze wywołania `send ( )` bez podawania struktury adresowej.



# Demonstracja serwera TCP (2)

---

`tcp_server_echo.c + tcp_client_echo.c`

kod klienta na stronie wykładu

demonstracja



---

# Implementacja TCP

---



# Flagi w segmencie TCP

|  |     |     |             |       |                         |       |  |    |  |
|--|-----|-----|-------------|-------|-------------------------|-------|--|----|--|
| 0  |     | 7 8 |             | 15 16 |                         | 23 24 |  | 31 |  |
| port źródłowy  |     |     |             |       | port docelowy           |       |  |    |  |
| numer sekwencyjny (numer pierwszego bajtu w segmencie) |     |     |             |       |                         |       |  |    |  |
| numer ostatniego potwierdzanego bajtu + 1              |     |     |             |       |                         |       |  |    |  |
| offset   | 000 | ECN | U-A-P-R-S-F |       | oferowane okno          |       |  |    |  |
| suma kontrolna   |     |     |             |       | wskaźnik pilnych danych |       |  |    |  |
| dodatkowe opcje, np. potwierdzanie selektywne          |     |     |             |       |                         |       |  |    |  |

**Flagi = zapalone bity.**

- ❖ SYN = synchronize (do nawiązywania połączenia).
- ❖ ACK = pole „numer potwierdzanego bajtu” ma znaczenie.
- ❖ FIN = finish (do kończenia połączenia).



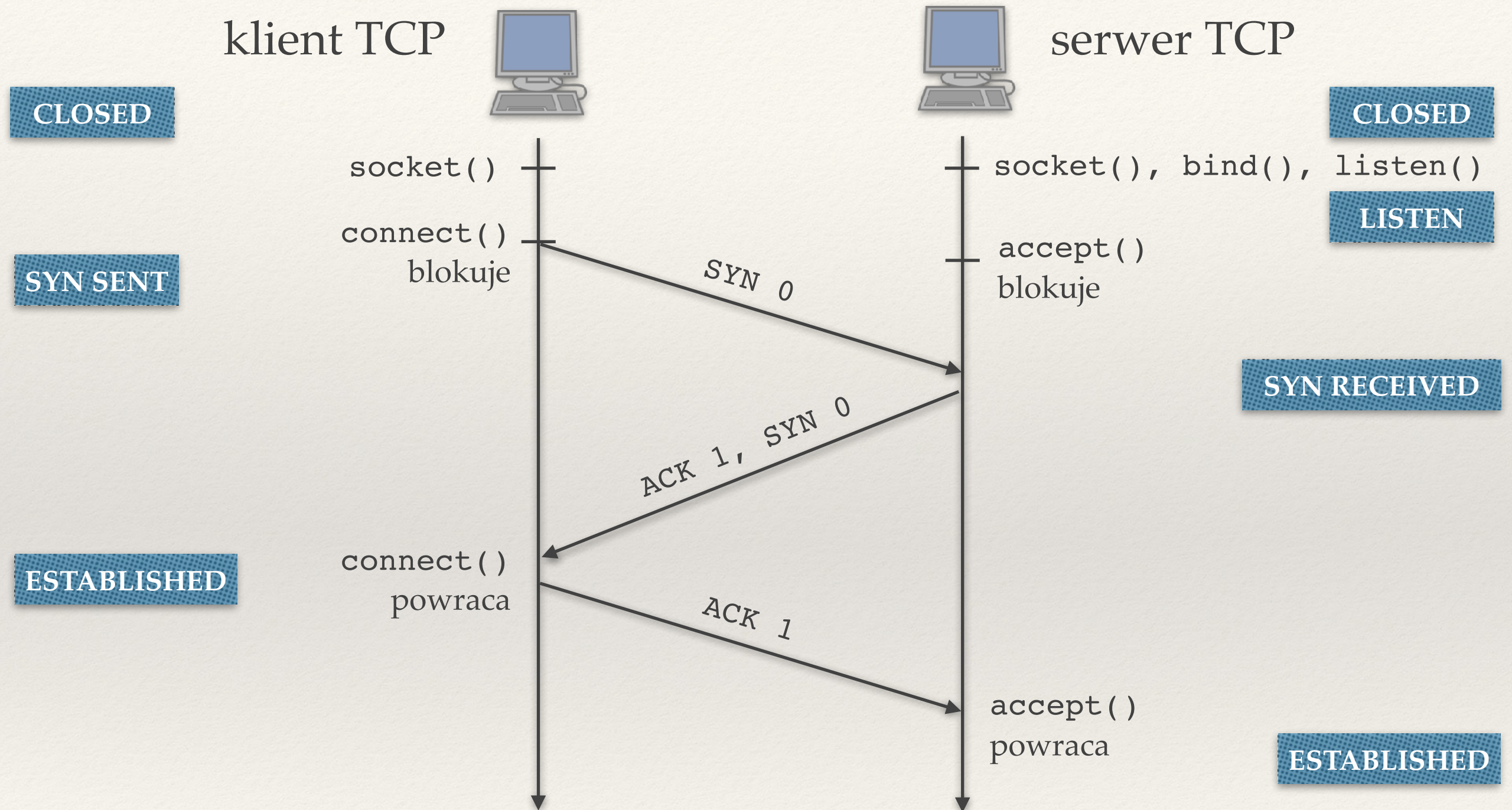
# Cykl życia połączenia

---

- ❖ Trójfazowe nawiązywanie połączenia.
- ❖ Przesyłanie danych.
- ❖ Czterofazowe kończenie połączenia.



# Trójfazowe nawiązywanie połączenia (1)





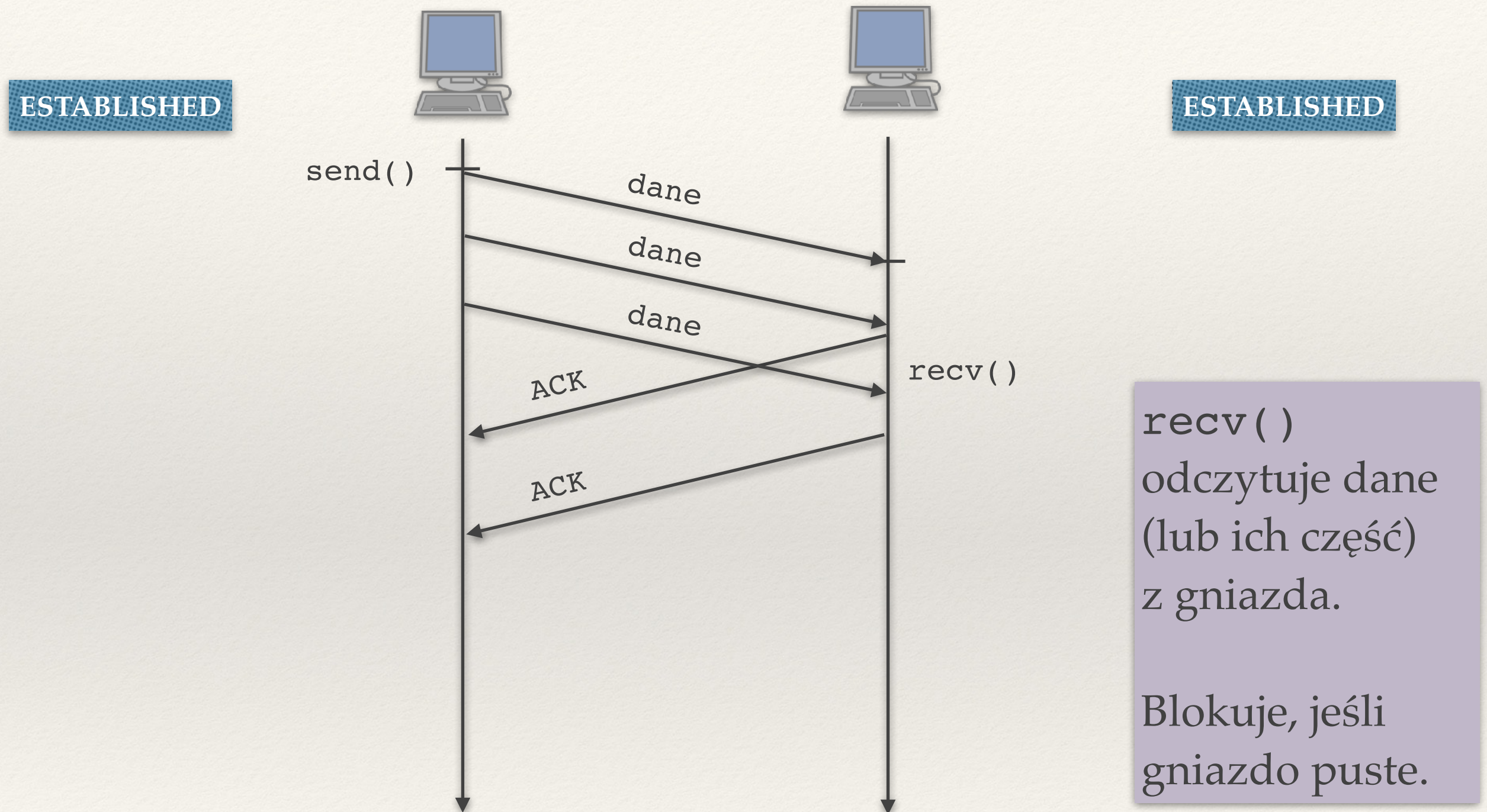
# Trójfazowe nawiązywanie połączenia (2)

---

- ❖ Przejście do stanu `LISTEN` = otwarcie bierne (nie wysyła pakietu), wykonuje serwer TCP.
- ❖ Przejście do stanu `SYN_SENT` = otwarcie czynne (wysyła segment `SYN`), wykonuje klient TCP.
- ❖ W rzeczywistości w segmencie `SYN` nie jest wysyłany numer 0, tylko początkowy numer sekwencyjny:
  - ♦ losowy, trudny do zgadnięcia → zapobiega podszywaniu się!
  - ♦ łatwo sfałszować źródłowy adres IP, ale trudno z takiego adresu rozpocząć komunikację TCP.

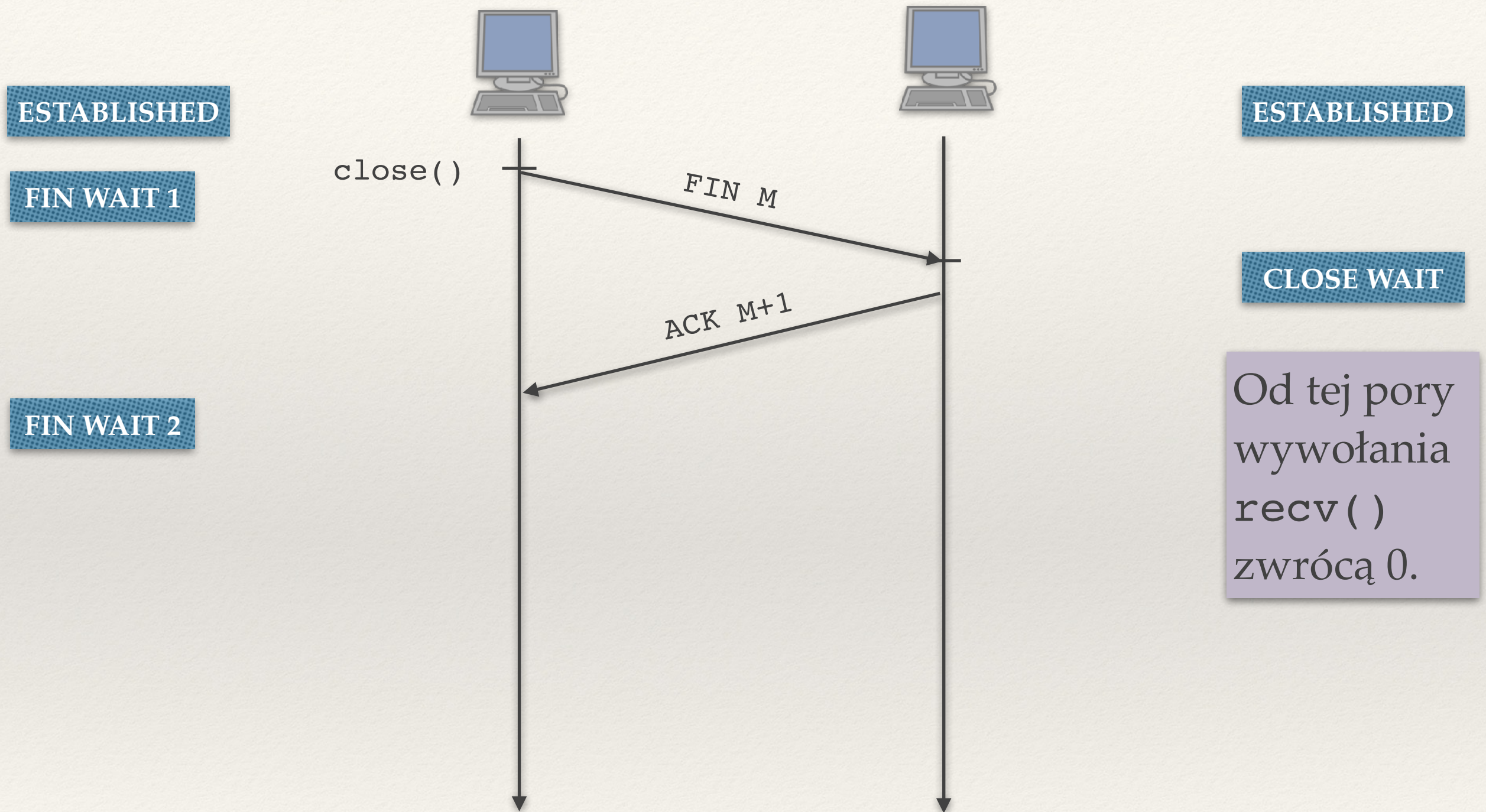


# Przesyłanie danych



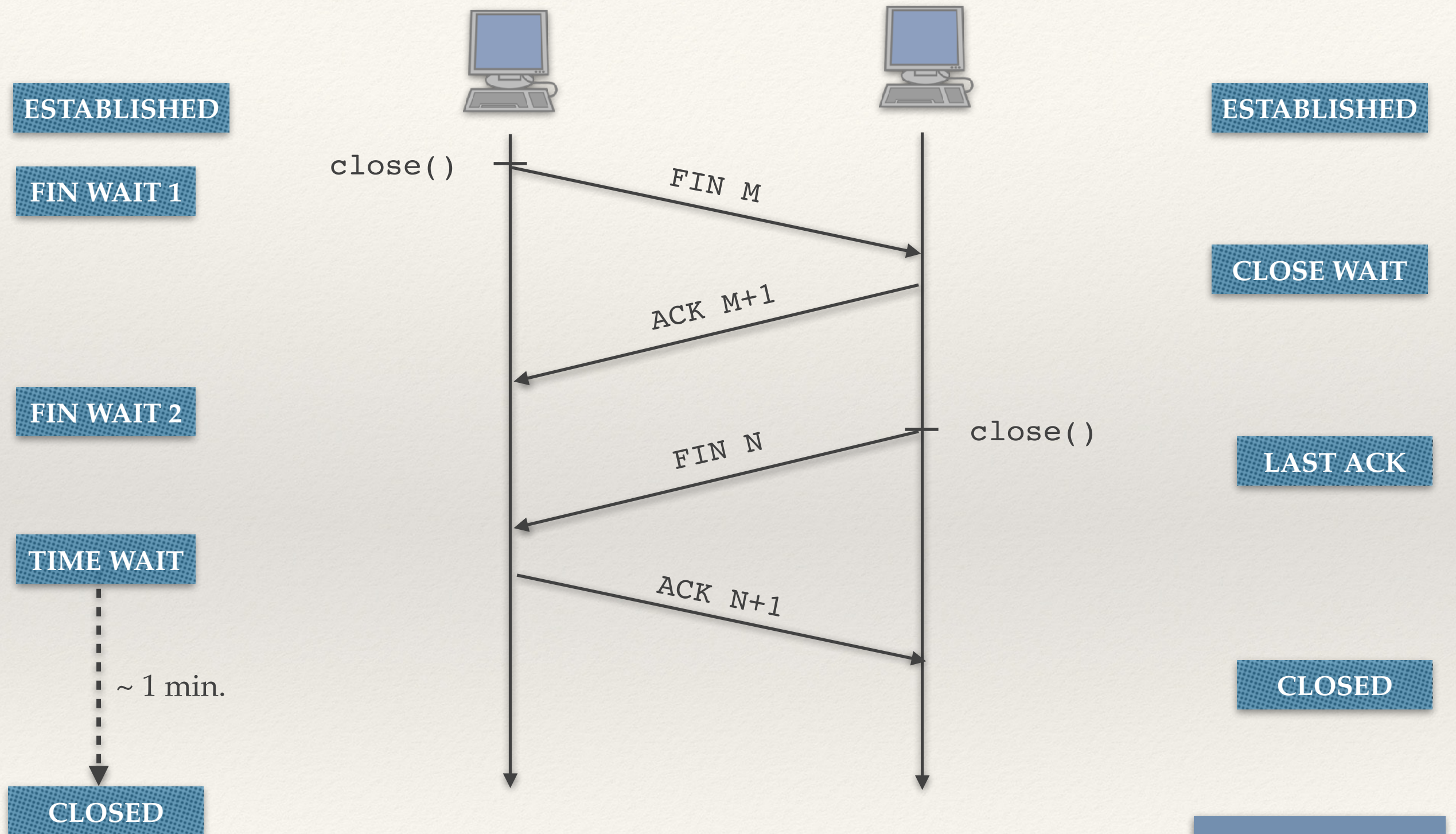


# Czterofazowe kończenie połączenia (1)





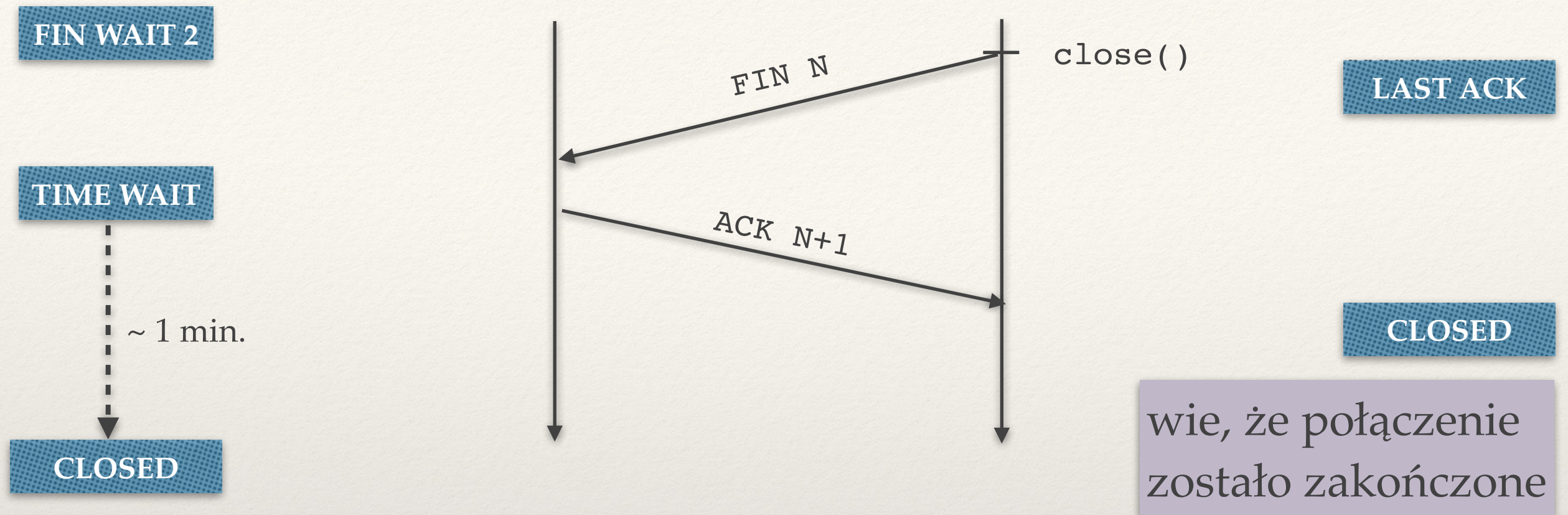
# Czterofazowe kończenie połączenia (2)



demonstracja



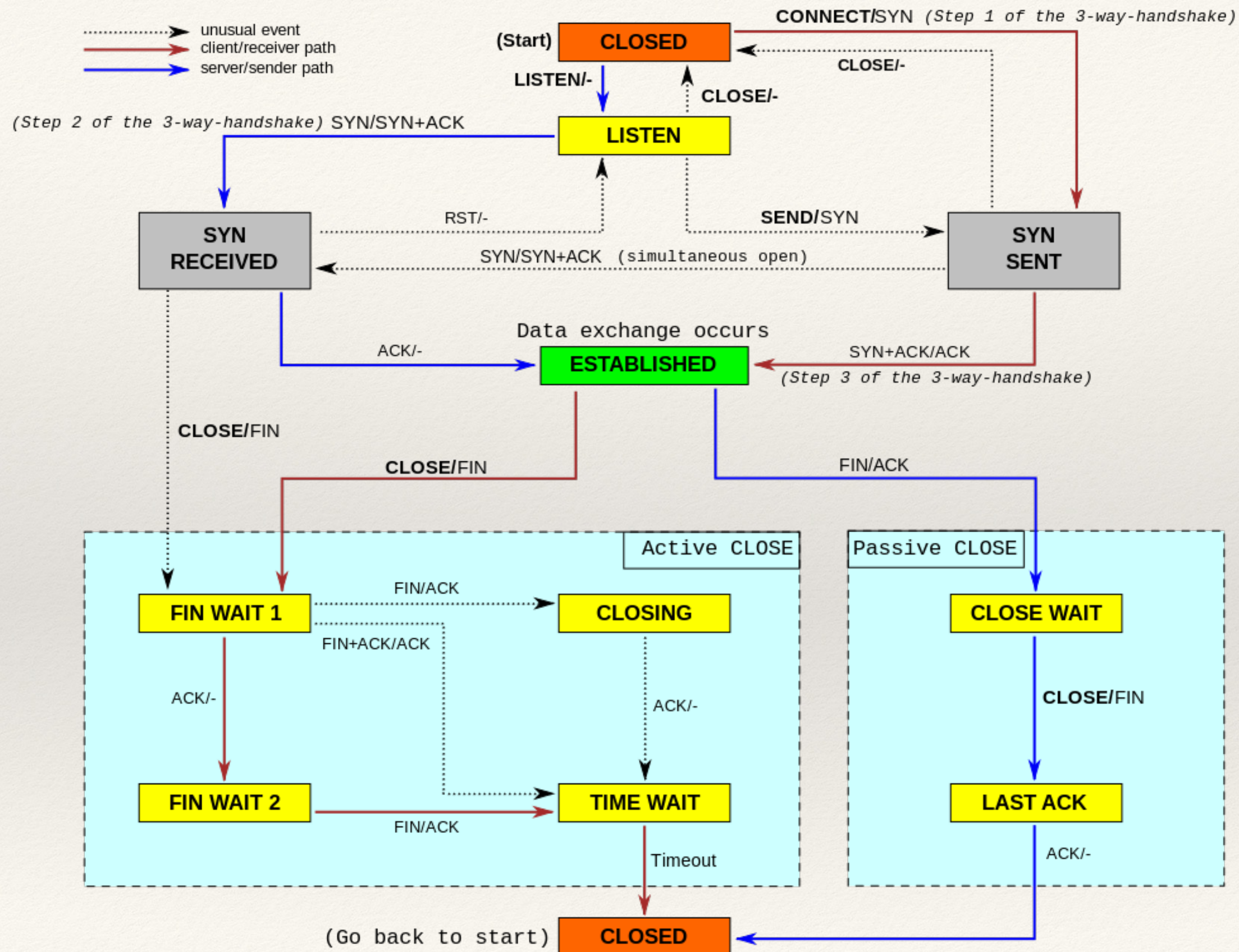
# Po co jest stan TIME WAIT?



- ❖ Lewa strona nie wie, czy prawa strona dostała jej ACK.
- ❖ Końcowy ACK nie dociera → prawa strona wysyła FIN jeszcze raz → lewa strona chce go poprawnie obsłużyć.
- ❖ Dodatkowy cel: nie chcemy żeby ktoś szybko utworzył połączenie TCP o takich samych parametrach (IP + porty) → stare duplikaty segmentów mogłyby być uznane za należące do nowego połączenia.



# Stany TCP: sytuacje nietypowe





---

# Wysyłanie większych danych

---



# Funkcja `send()`

---

Czy nasz klient `tcp_client_echo.c` działa poprawnie?

- ❖ Wyślijmy 1 mln bajtów lub więcej.

demonstracja



# Funkcja `send()`

---

Czy nasz klient `tcp_client_echo.c` działa poprawnie?

- ❖ Wyślijmy 1 mln bajtów lub więcej.
- ❖ `send ( )` może zwrócić mniej i nie jest to błąd!

demonstracja



# Funkcja send()

---

Czy nasz klient `tcp_client_echo.c` działa poprawnie?

- ❖ Wyślijmy 1 mln bajtów lub więcej.
- ❖ `send ( )` może zwrócić mniej i nie jest to błąd!
- ❖ `send ( )` zwraca, *ile bajtów zapisano do bufora wysyłkowego*.

demonstracja



# Funkcja send()

---

Czy nasz klient `tcp_client_echo.c` działa poprawnie?

- ❖ Wyślijmy 1 mln bajtów lub więcej.
- ❖ `send ( )` może zwrócić mniej i nie jest to błąd!
- ❖ `send ( )` zwraca, *ile bajtów zapisano do bufora wysyłkowego*.
  - ♦ wysłanych przez jądro może być jeszcze mniej,

demonstracja



# Funkcja send()

---

Czy nasz klient `tcp_client_echo.c` działa poprawnie?

- ❖ Wyślijmy 1 mln bajtów lub więcej.
- ❖ `send ( )` może zwrócić mniej i nie jest to błąd!
- ❖ `send ( )` zwraca, *ile bajtów zapisano do bufora wysyłkowego*.
  - ♦ wysłanych przez jądro może być jeszcze mniej,
  - ♦ a odebranych przez serwer jeszcze mniej!

demonstracja



# Wysyłanie do skutku

---

```
size_t n_left = n;
while (n_left > 0) {
    ssize_t bytes_sent = send(sockfd, buffer, n_left, 0);
    if (bytes_sent < 0)
        ERROR("send error");
    printf("%ld bytes sent\n", bytes_sent);
    n_left -= bytes_sent;
    buffer += bytes_sent;
}
```

kod klienta `tcp_client_echo_fixed.c` na stronie wykładu



# Segment RST

---

- ❖ Segment z flagą RST (reset): wysyłany kiedy wystąpi błąd.
  - ✦ Przykładowo w odpowiedzi na dowolny segment wysłany do zamkniętego portu.
- ❖ Po otrzymaniu takiego segmentu z gniazda nie można już korzystać.



# Funkcja `recv()`

---

**`tcp_server_echo.c + tcp_client_echo_fixed.c`**

- ❖ Wyślijmy 1 mln bajtów lub więcej



# Funkcja `recv()`

---

**`tcp_server_echo.c + tcp_client_echo_fixed.c`**

- ❖ Wyślijmy 1 mln bajtów lub więcej
- ❖ Dlaczego `tcp_server_echo.c` czyta tylko część z nich?



# Funkcja `recv()`

---

## `tcp_server_echo.c + tcp_client_echo_fixed.c`

- ❖ Wyślijmy 1 mln bajtów lub więcej
- ❖ Dlaczego `tcp_server_echo.c` czyta tylko część z nich?
- ❖ Aplikacja serwera zamyka gniazdo
  - serwer otrzymuje kolejne segmenty od klienta
  - warstwa TCP po stronie serwera wysyła RST
  - gniazdo po stronie klienta psuje się
  - klient zapisuje do zepsutego gniazda
  - klient otrzymuje sygnał SIGPIPE.

demonstracja



# Na czym polega problem?

---

- ❖ Do jakiego momentu `recv ( )` powinno czytać dane?
- ❖ **Problem:** nie zdefiniowaliśmy protokołu komunikacji!



# Na czym polega problem?

---

- ❖ Do jakiego momentu `recv ( )` powinno czytać dane?
- ❖ **Problem:** nie zdefiniowaliśmy protokołu komunikacji!
- ❖ **Podejście nr 1:** na początku wysyłamy rozmiar danych.



# Na czym polega problem?

---

- ❖ Do jakiego momentu `recv ( )` powinno czytać dane?
- ❖ **Problem:** nie zdefiniowaliśmy protokołu komunikacji!
- ❖ **Podejście nr 1:** na początku wysyłamy rozmiar danych.
- ❖ **Podejście nr 2:** ustalamy znacznik końca rekordu a potem:

```
while (znacznik nie napotkany) {  
    recv(porcja danych)  
    przetworz(porcja danych)  
}
```



# Na czym polega problem?

---

- ❖ Do jakiego momentu `recv ( )` powinno czytać dane?
- ❖ **Problem:** nie zdefiniowaliśmy protokołu komunikacji!
- ❖ **Podejście nr 1:** na początku wysyłamy rozmiar danych.
- ❖ **Podejście nr 2:** ustalamy znacznik końca rekordu a potem:

```
while (znacznik nie napotkany) {  
    recv(porcja danych)  
    przetworz(porcja danych)  
}
```

Nie chcemy czekać  
sumarycznie dłużej niż  
 $x$  sekund

Jeśli napotkamy znacznik, to przerywamy.



# Przypomnienie: funkcja poll()

---

Czekanie maksymalnie  $x$  milisekund na pakiet w gnieździe sockfd.



# Przypomnienie: funkcja poll()

---

Czekanie maksymalnie  $x$  milisekund na pakiet w gnieździe sockfd.

```
struct pollfd ps;  
    ps.fd = sockfd;  
    ps.events = POLLIN;  
    ps.revents = 0;  
int ready = poll(&ps, 1, x);
```



# Przypomnienie: funkcja poll()

Czekanie maksymalnie  $x$  milisekund na pakiet w gnieździe `sockfd`.

```
struct pollfd ps;  
    ps.fd = sockfd;  
    ps.events = POLLIN;  
    ps.revents = 0;  
int ready = poll(&ps, 1, x);
```

- ❖ `ready = 0` → nastąpił timeout (po  $x$  milisekundach).
- ❖ `ready < 0` → wystąpił błąd (zazwyczaj przerwanie sygnałem)
- ❖ `ready > 0` → `ready` obserwowanych deskryptorów “gotowych do odczytu”.
  - ❖ Trzeba sprawdzić, czy `ps.revents == POLLIN`!
  - ❖ Najbliższe wywołanie `recv(sockfd, ...)` nie zablokuje.

kod klienta `tcp_server_echo_poll.c` na stronie wykładu



# Przypomnienie: funkcja poll()

Czekanie maksymalnie  $x$  milisekund na pakiet w gnieździe `sockfd`.

```
struct pollfd ps;  
    ps.fd = sockfd;  
    ps.events = POLLIN;  
    ps.revents = 0;  
int ready = poll(&ps, 1, x);
```

Obserwowaliśmy 1,  
więc `ready = 1`

- ❖ `ready = 0` → nastąpił timeout (po  $x$  milisekundach).
- ❖ `ready < 0` → wystąpił błąd (zazwyczaj przerwanie sygnałem)
- ❖ `ready > 0` → `ready` obserwowanych deskryptorów “gotowych do odczytu”.
  - ❖ Trzeba sprawdzić, czy `ps.revents == POLLIN`!
  - ❖ Najbliższe wywołanie `recv(sockfd, ...)` nie zablokuje.

kod klienta `tcp_server_echo_poll.c` na stronie wykładu



# Lektura dodatkowa

---

- ❖ Kurose, Ross: rozdział 3
- ❖ Tanenbaum: rozdział 6
- ❖ Stevens: rozdziały 3–6, 13, 27
- ❖ Beej's Guide to Network Programmin:  
`https://beej.us/guide/bgnet/`



# Zagadnienia

---

- ❖ Co to jest gniazdo?
- ❖ Czym różni się gniazdo nasłuchujące od gniazda połączonego? Czy w protokole UDP mamy gniazda połączone?
- ❖ Co robią funkcje jądra `bind()`, `listen()`, `accept()`, `connect()`?
- ❖ Czym różni się komunikacja bezpołączeniowa od połączeniowej?
- ❖ Czym różni się otwarcie bierne od otwarcia aktywnego? Czy serwer może wykonać otwarcie aktywne?
- ❖ Do czego służą flagi SYN, ACK, FIN i RST stosowane w protokole TCP?
- ❖ Opisz trójstopniowe nawiązywanie połączenia w TCP. Jakie informacje są przesyłane w trakcie takiego połączenia?
- ❖ Dlaczego przesyłanych bajtów nie numeruje się od zera?
- ❖ Jakie segmenty są wymieniane podczas zamykania połączenia w protokole TCP?
- ❖ Co zwraca funkcja `recv()` wywołana na gnieździe w blokującym i nieblokującym trybie?
- ❖ Po co wprowadzono stan `TIME_WAIT`?
- ❖ Na podstawie diagramu stanów TCP opisz możliwe scenariusze nawiązywania i kończenia połączenia.