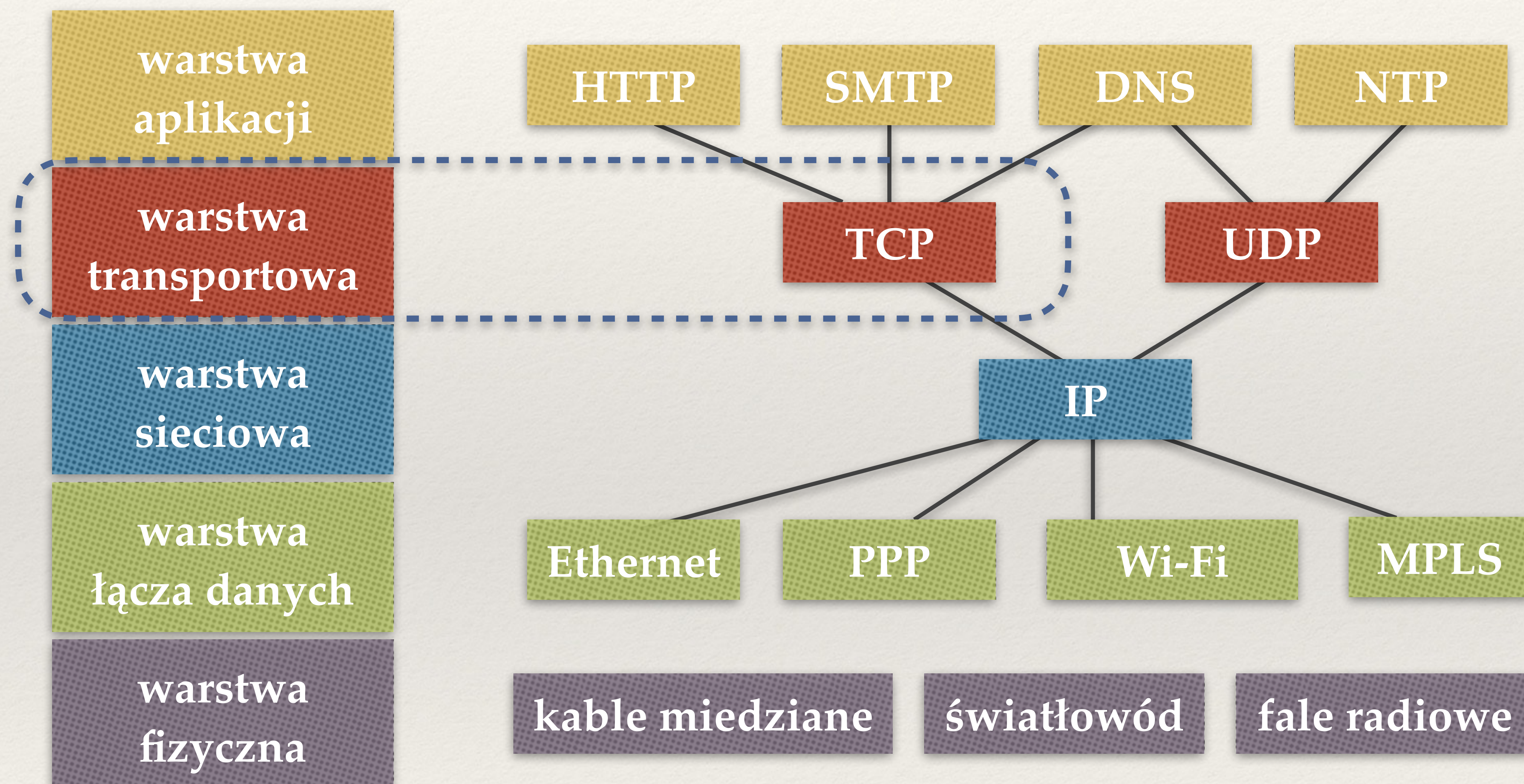
Transport
część 2: protokół TCP

Sieci komputerowe

Wykład 7

Marcin Bieńkowski

Protokoły w Internecie



W poprzednim odcinku: niezawodny transport

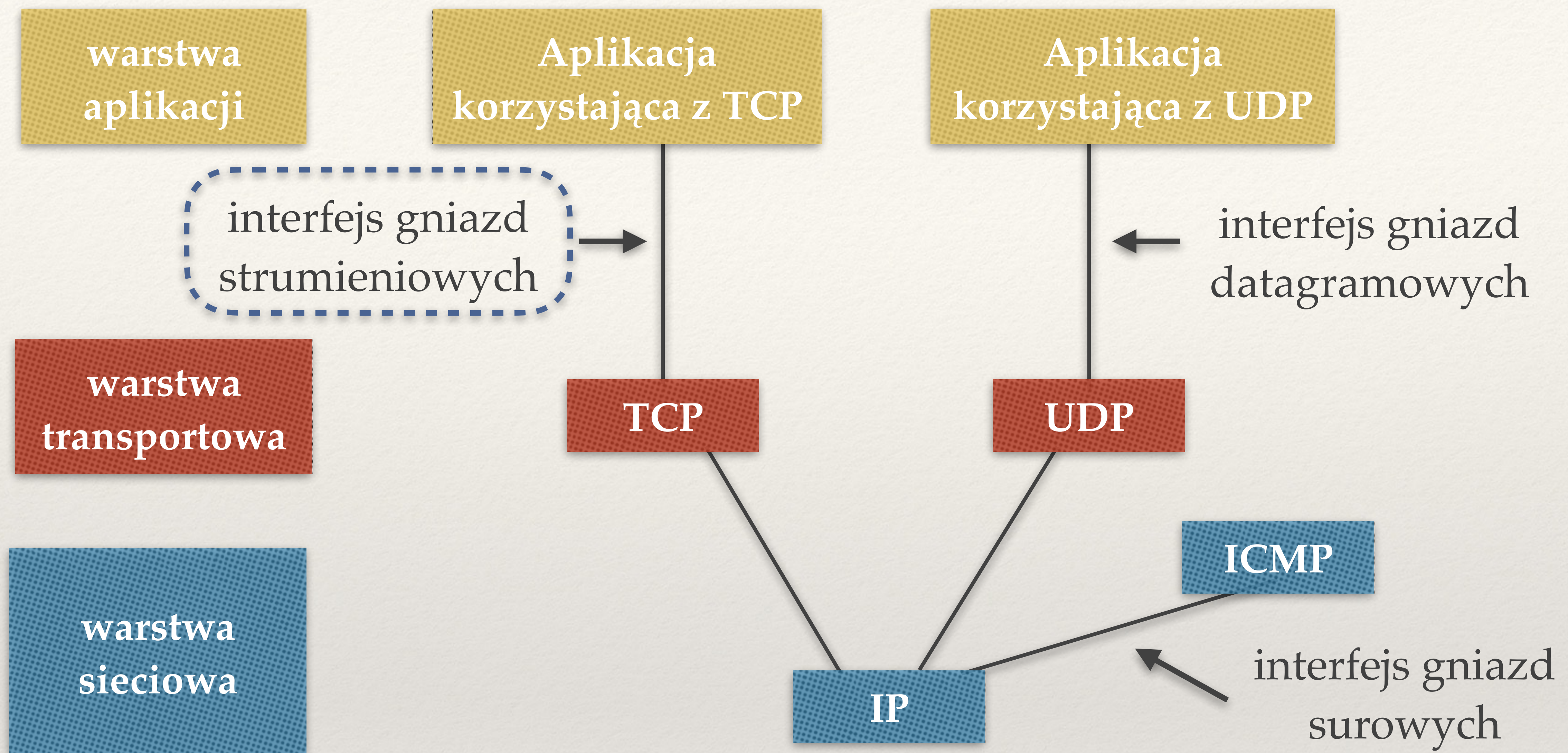
- Segmentacja.
- Algorytmy niezawodnego dostarczania danych: stop-and-wait, go-back-N, wybiórcze ponawianie, okno przesuwne nadawcy / odbiorcy, potwierdzanie skumulowane.
- Niezawodny transport w TCP: okna przesuwne + potwierdzanie skumulowane.
- Kontrola przepływu w TCP: okno oferowane.

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

- 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

Gniazda nasłuchujące.

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

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`.

Gniazdo opisywane (między innymi) przez cztery elementy:
lokalne IP, lokalny port, zdalne IP, zdalny port.

Gniazda TCP: dwa typy gniazd

Gniazda nasłuchujące.

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

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`.

Gniazdo opisywane (między innymi) przez cztery elementy: lokalne IP, lokalny port, zdalne IP, zdalny port.

demonstracja

Dobrze znane porty

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

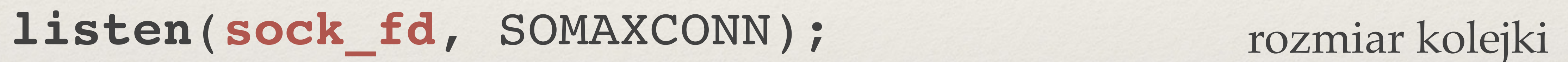
```
struct sockaddr_in server_address = {0};
server_address.sin_family       = AF_INET;
server_address.sin_port         = htons(32345);
server_address.sin_addr.s_addr = htonl(INADDR_ANY);

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

Oczekiwanie na nawiązanie połączenia (serwer)

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

- **TDP**: trzeba najpierw nawiązać połączenie.
  - Tworzenie kolejki na nawiązane, ale nie obsłużone połączenia:

```
listen(sock_fd, SOMAXCONN);
```



  - Pobieranie nawiązanego połączenia z kolejki:

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

gniazdo połączone z klientem

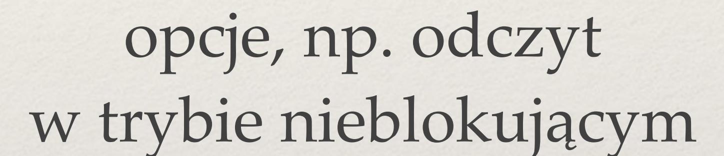
gniazdo do odbierania kolejnych połączeń przez `accept()`

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, SOMAXCONN);
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

```
uint8_t buffer[BUFFER_SIZE];

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



- Nie musimy odczytywać informacji o nadawcy jak w UDP.
- `recv(s,b,x,0) = read(s,b,x).`

Wysyłanie danych przez gniazdo

```
uint8_t buffer[…];

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

- Do wysłania danych pod konkretny adres wystarczy gniazdo `connected_sock_fd`.
- `send(s,b,x,0) = write(s,b,x)`.

Ogólna budowa serwera TCP (powtórzenie)

```
int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
bind(sock_fd, …);
listen(sock_fd, SOMAXCONN);
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);
```

Ogólna budowa serwera TCP (powtórzenie)

```
int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
bind(sock_fd, …);
listen(sock_fd, SOMAXCONN);
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);
```

kod `tcp_server.c` na stronie wykładu

demonstracja

Ogólna budowa klienta TCP

```
int sock_fd = socket(AF_INET, SOCK_STREAM, 0);

// Stworzenie i wypełnienie struktury server_address.

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

// Komunikacja z serwerem za pomocą send() i recv()
// wykorzystująca sockfd.
```

Ogólna budowa klienta TCP

```
int sock_fd = socket(AF_INET, SOCK_STREAM, 0);

// Stworzenie i wypełnienie struktury server_address.

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

// Komunikacja z serwerem za pomocą send() i recv()
// wykorzystująca sockfd.
```

kod `tcp_client.c` na stronie wykładu

demonstracja

Nawiązywanie połączenia (klient)

- **UDP:**
  - Tworzymy gniazdo i od razu możemy wysyłać dane.
  - Przy każdym wywołaniu `sendto()` trzeba podać strukturę adresową serwera.

Nawiązywanie połączenia (klient)

- **UDP:**
  - Tworzymy gniazdo i od razu możemy wysyłać dane.
  - Przy każdym wywołaniu `sendto()` trzeba podać strukturę adresową serwera.

- **TCP:**
  - Tworzymy gniazdo i potem nawiązujemy połączenie za pomocą `connect()`.
  - Późniejsze wywołania `send()` bez podawania struktury adresowej.

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					

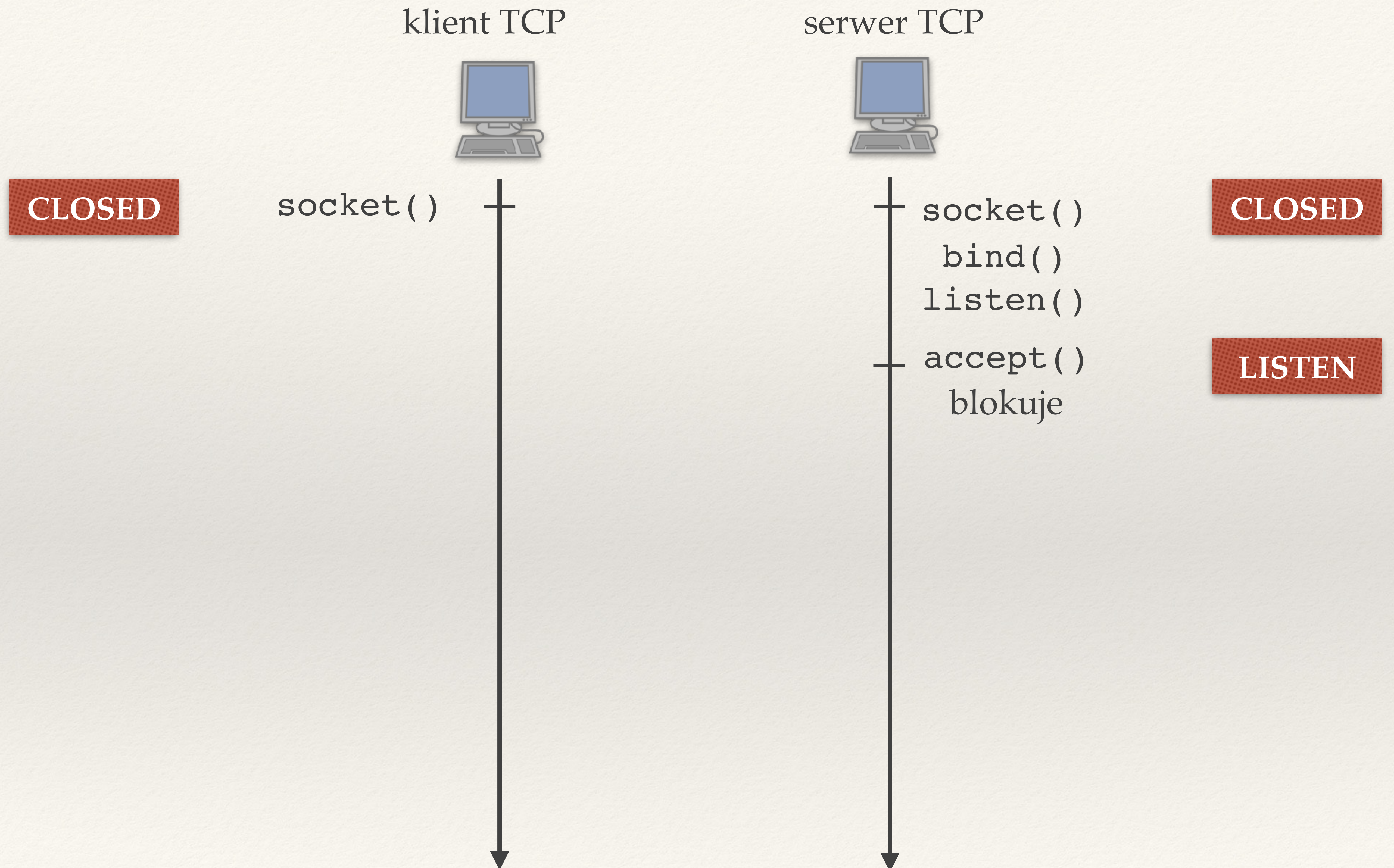
Flagi = zapalone bity.

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

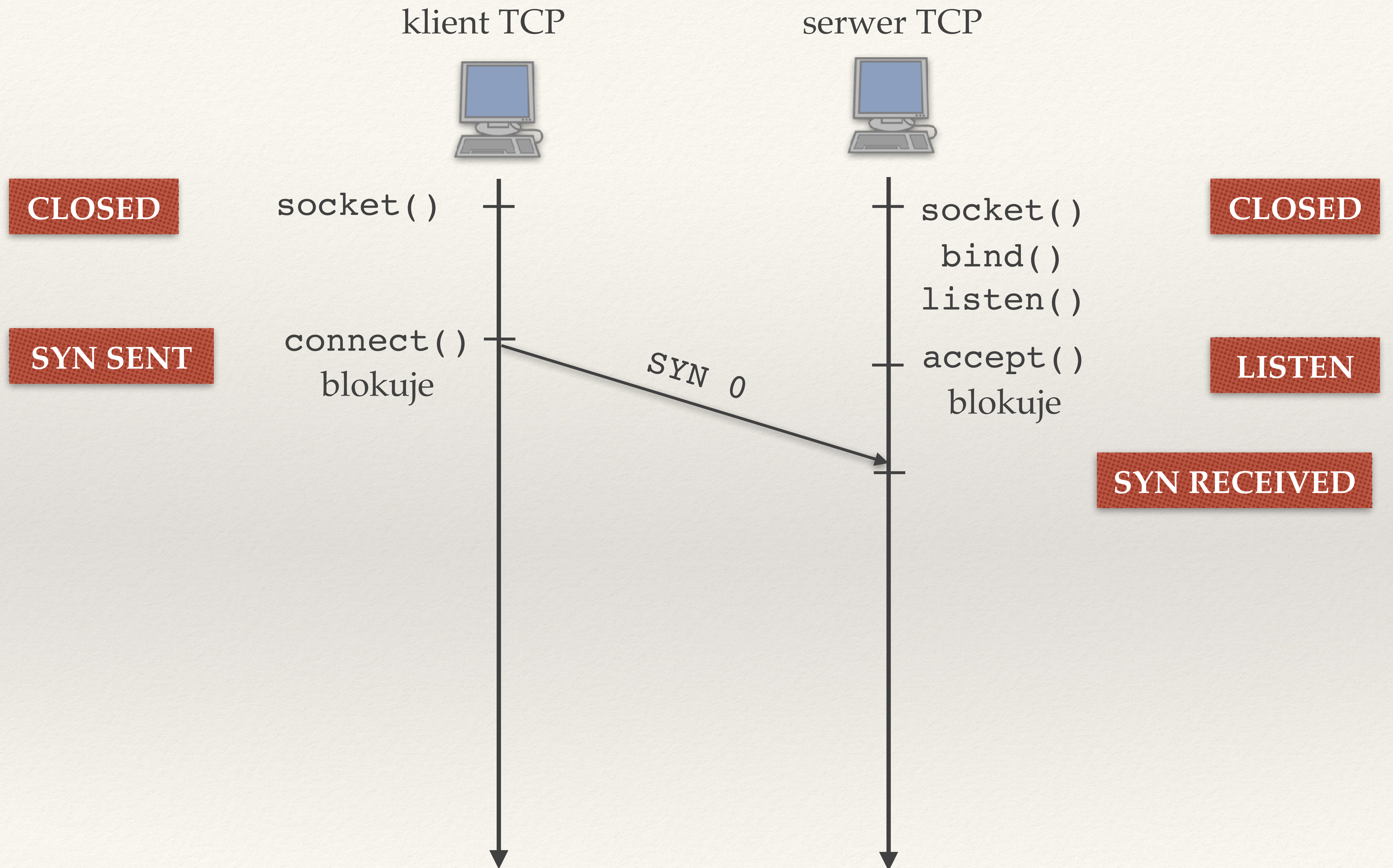
Cykl życia połączenia

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

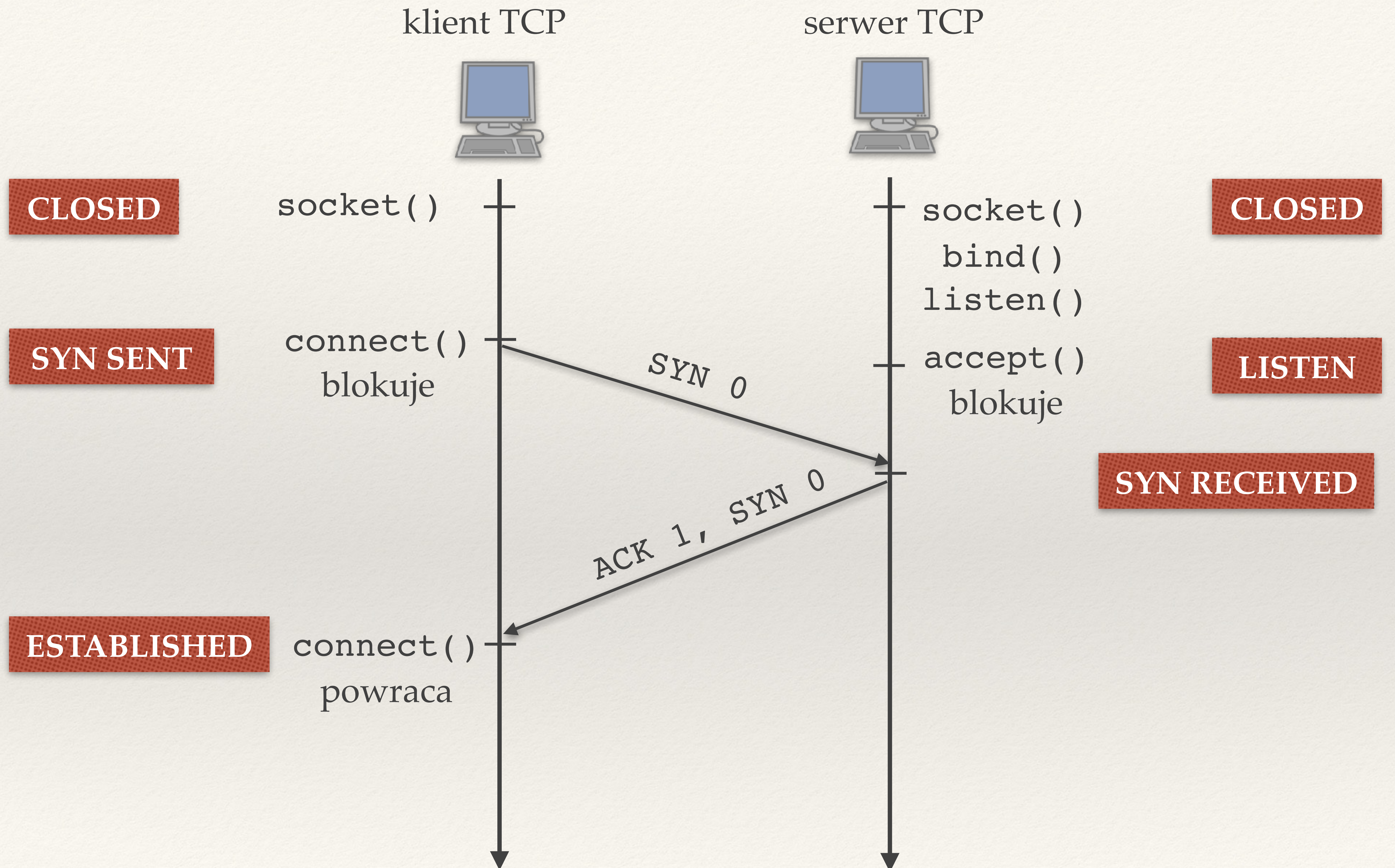
Trójfazowe nawiązywanie połączenia (1)



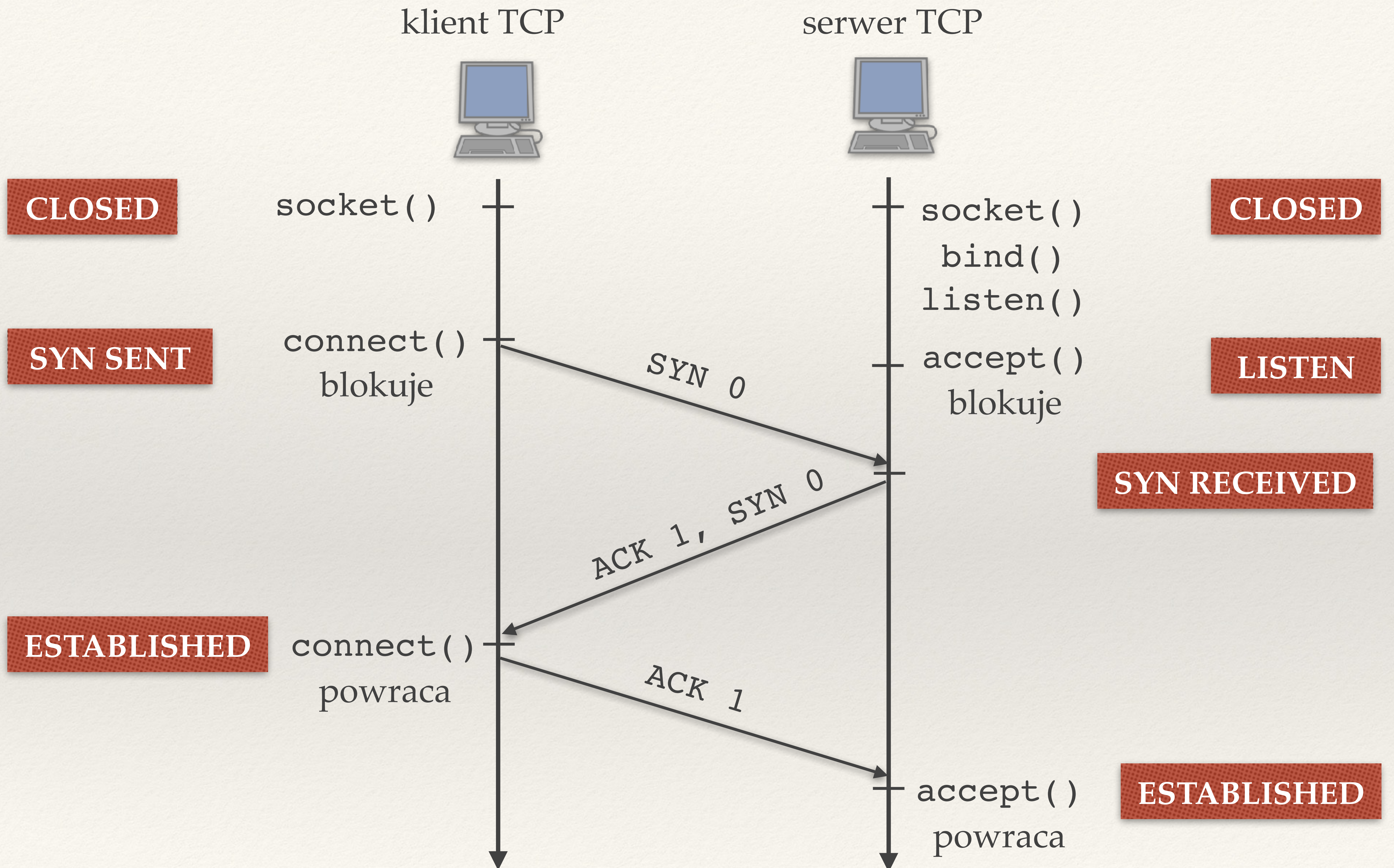
Trójfazowe nawiązywanie połączenia (1)



Trójfazowe nawiązywanie połączenia (1)



Trójfazowe nawiązywanie połączenia (1)



Trójfazowe nawiązywanie połączenia (2)

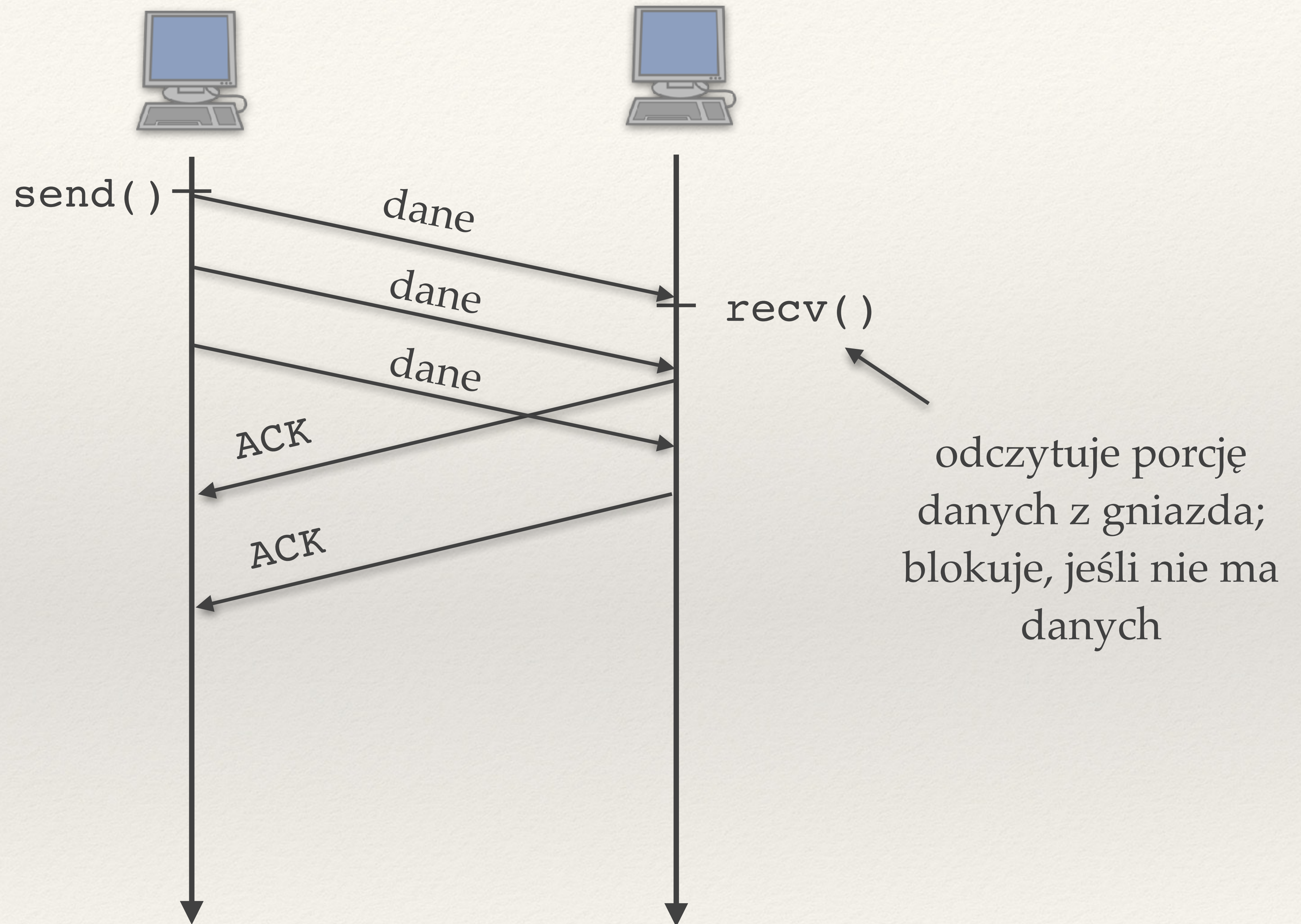
- **Otwarcie bierne** = przejście do stanu `LISTEN`, nie wysyła pakietu, wykonuje serwer TCP.
- **Otwarcie czynne** = wysłanie segmentu `SYN`, przejście do stanu `SYN_SENT`, wykonuje klient TCP.

Trójfazowe nawiązywanie połączenia (2)

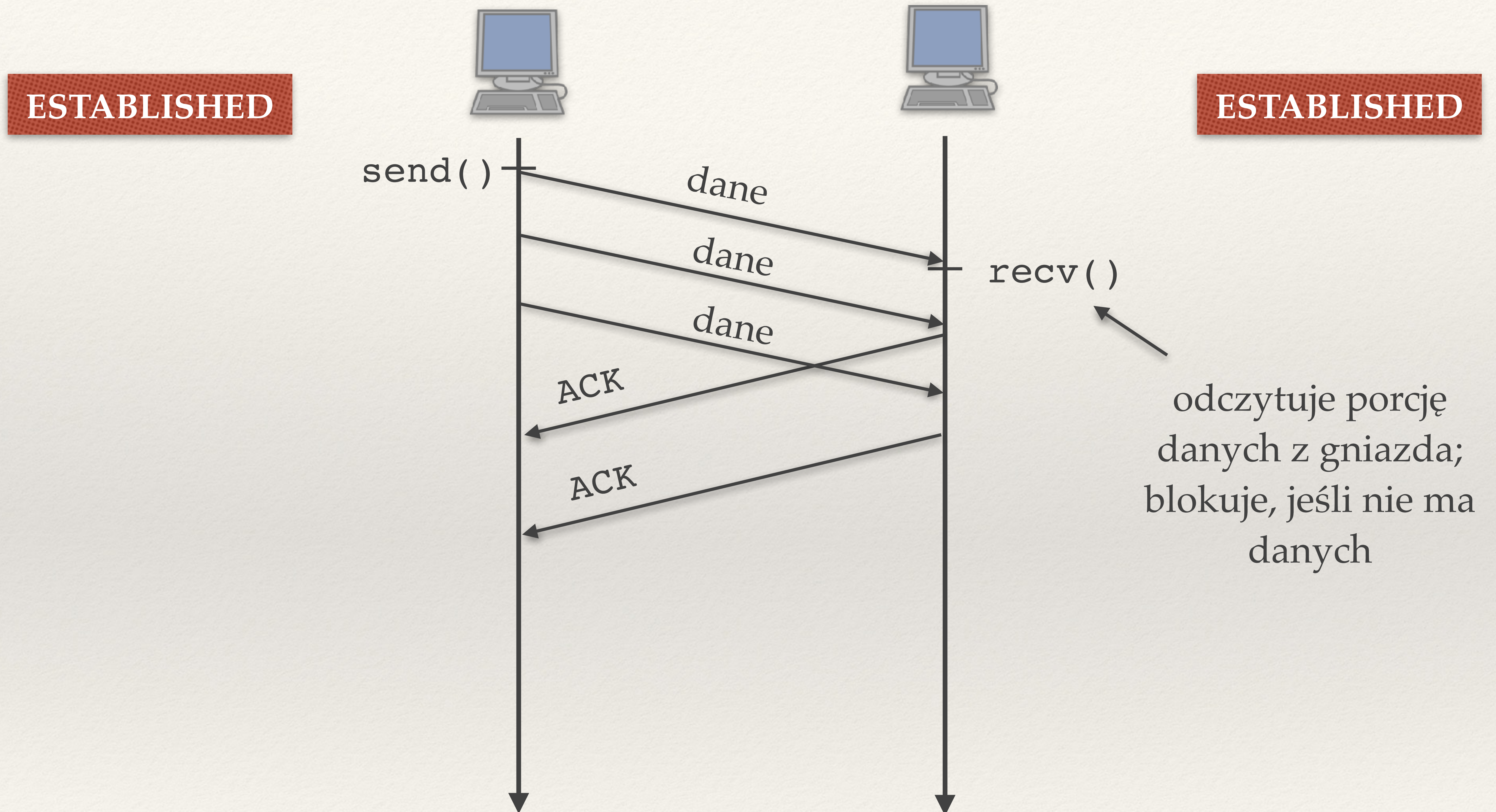
- **Otwarcie bierne** = przejście do stanu `LISTEN`, nie wysyła pakietu, wykonuje serwer TCP.
- **Otwarcie czynne** = wysłanie segmentu `SYN`, przejście do stanu `SYN_SENT`, wykonuje klient TCP.

- W pierwszych segmentach `SYN` nie jest wysyłany numer 0, tylko początkowy numer sekwencji (*initial sequence number*, ISN).
  - Losowy, trudny do zgadnięcia.
  - Łatwo sfałszować źródłowy adres IP i wysłać pierwszy SYN, ale bez znajomości ISN, nie wyślemy pierwszego `ACK` i nie rozpoczniemy komunikacji TCP.

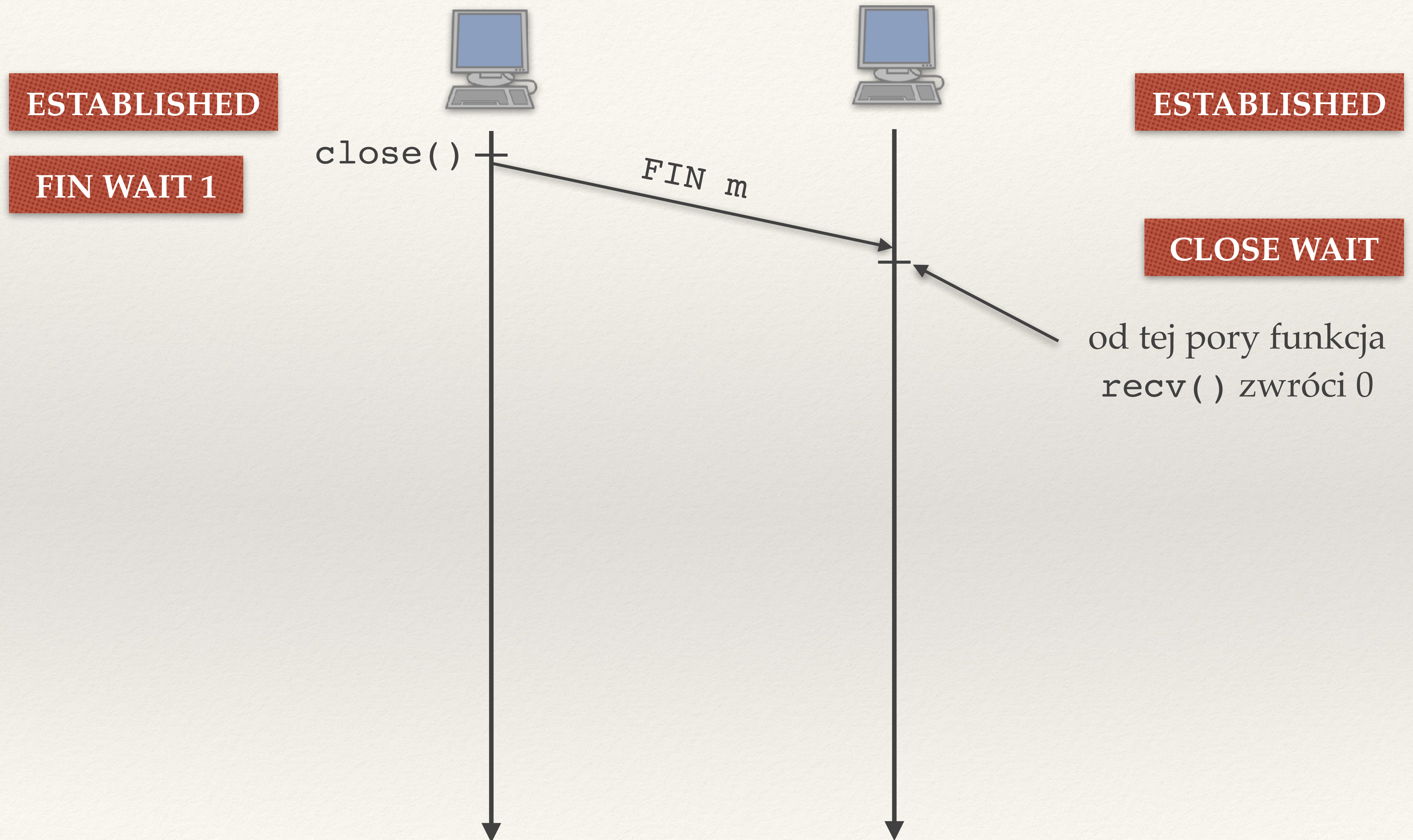
Przesyłanie danych



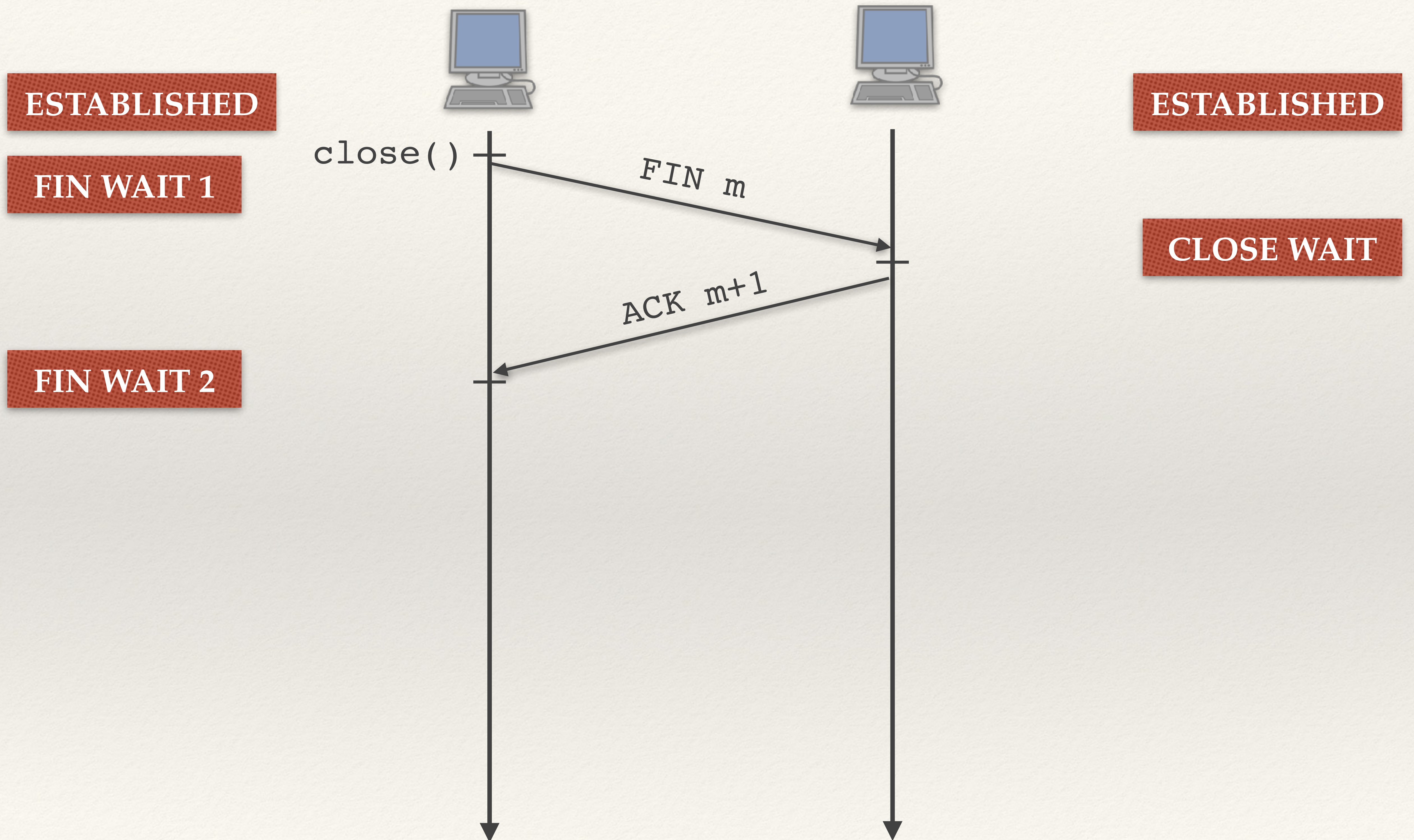
Przesyłanie danych



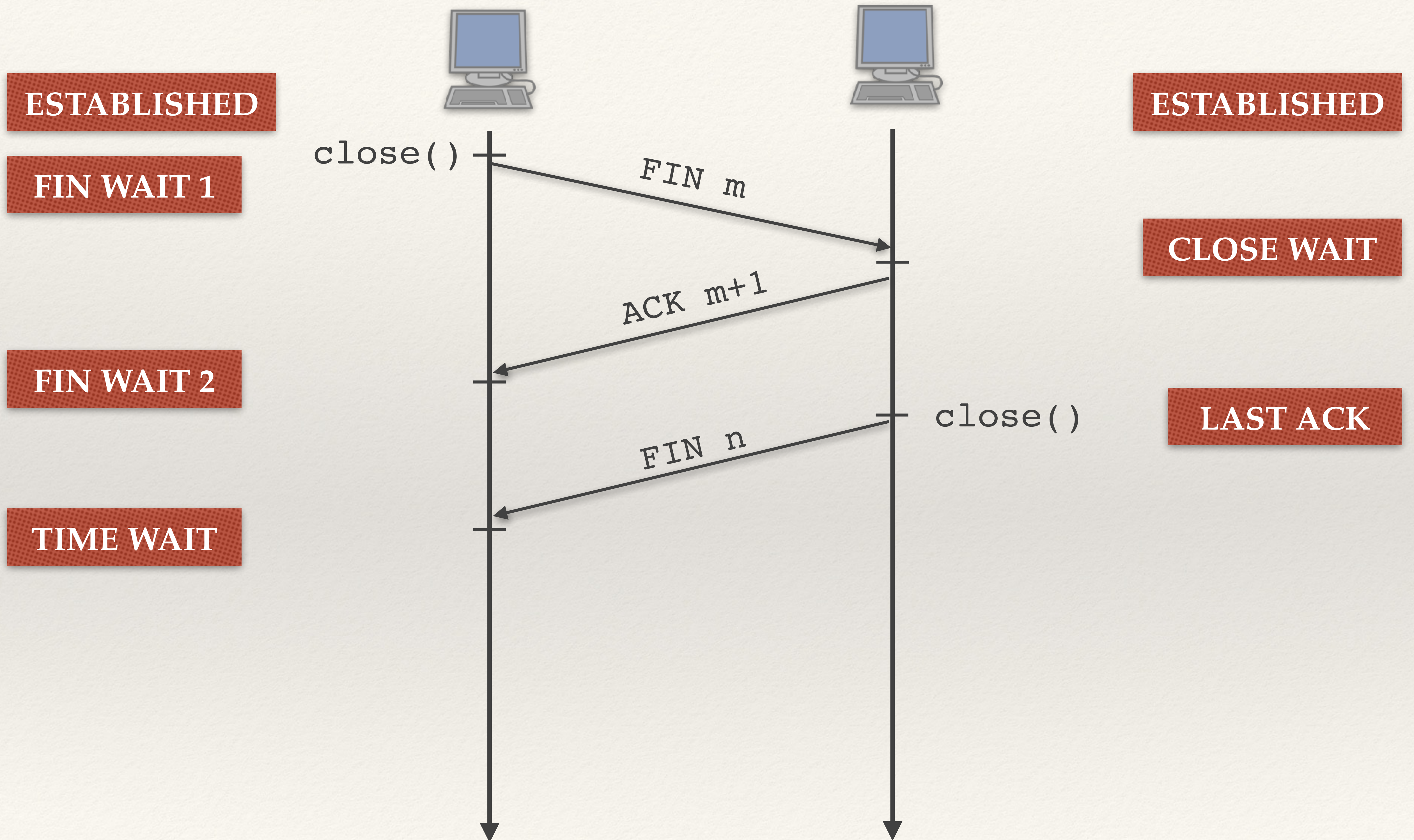
Czterofazowe kończenie połączenia



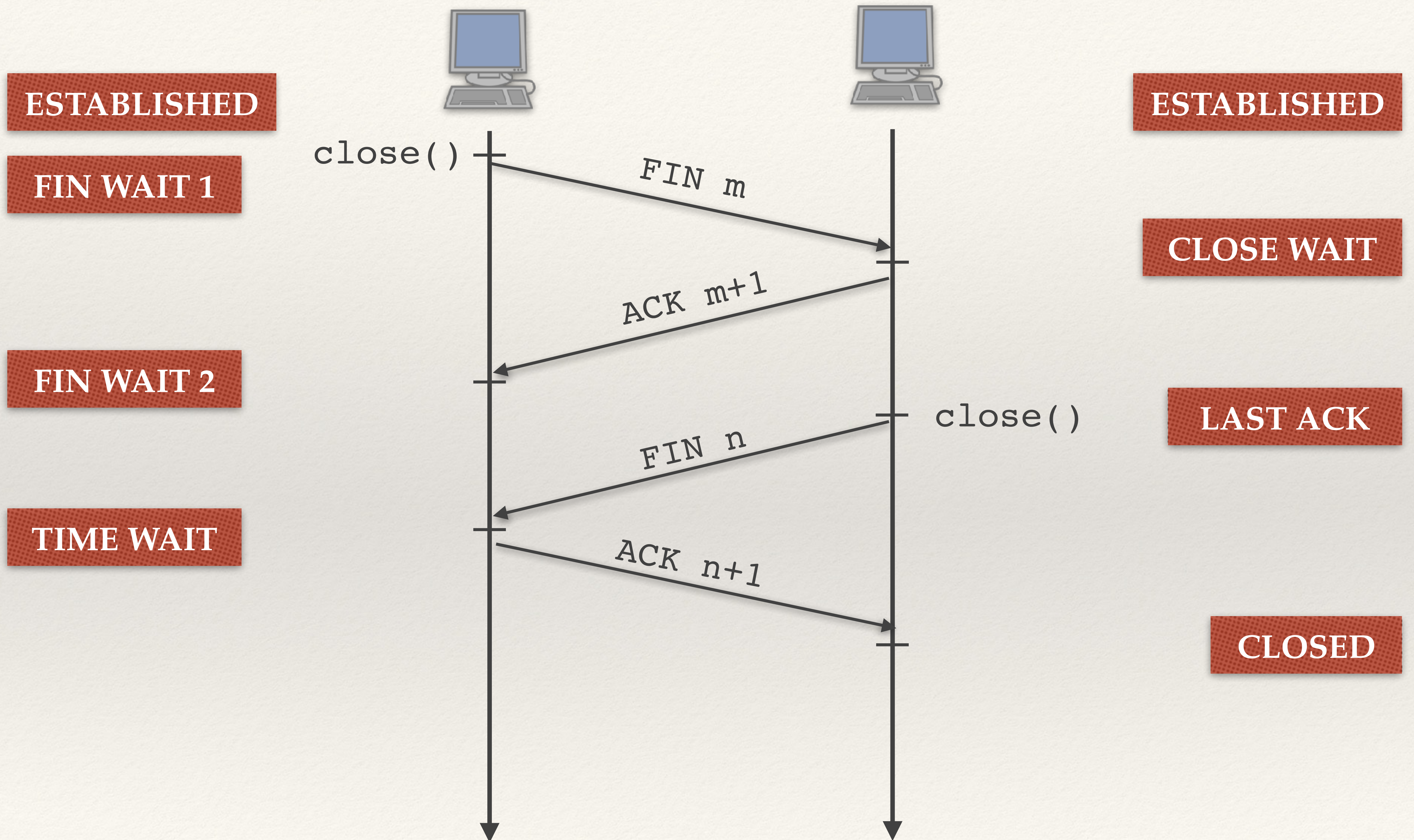
Czterofazowe kończenie połączenia



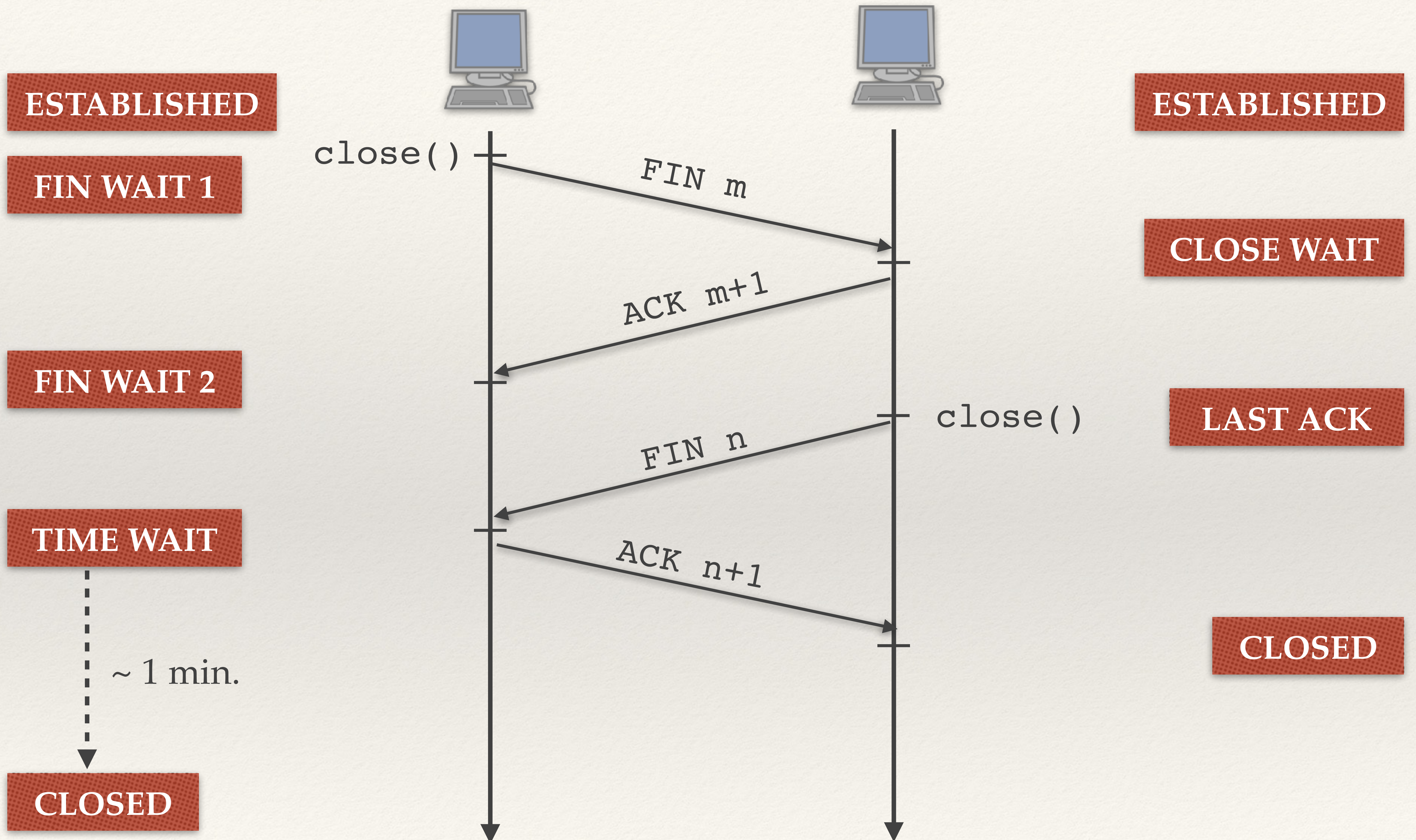
Czterofazowe kończenie połączenia



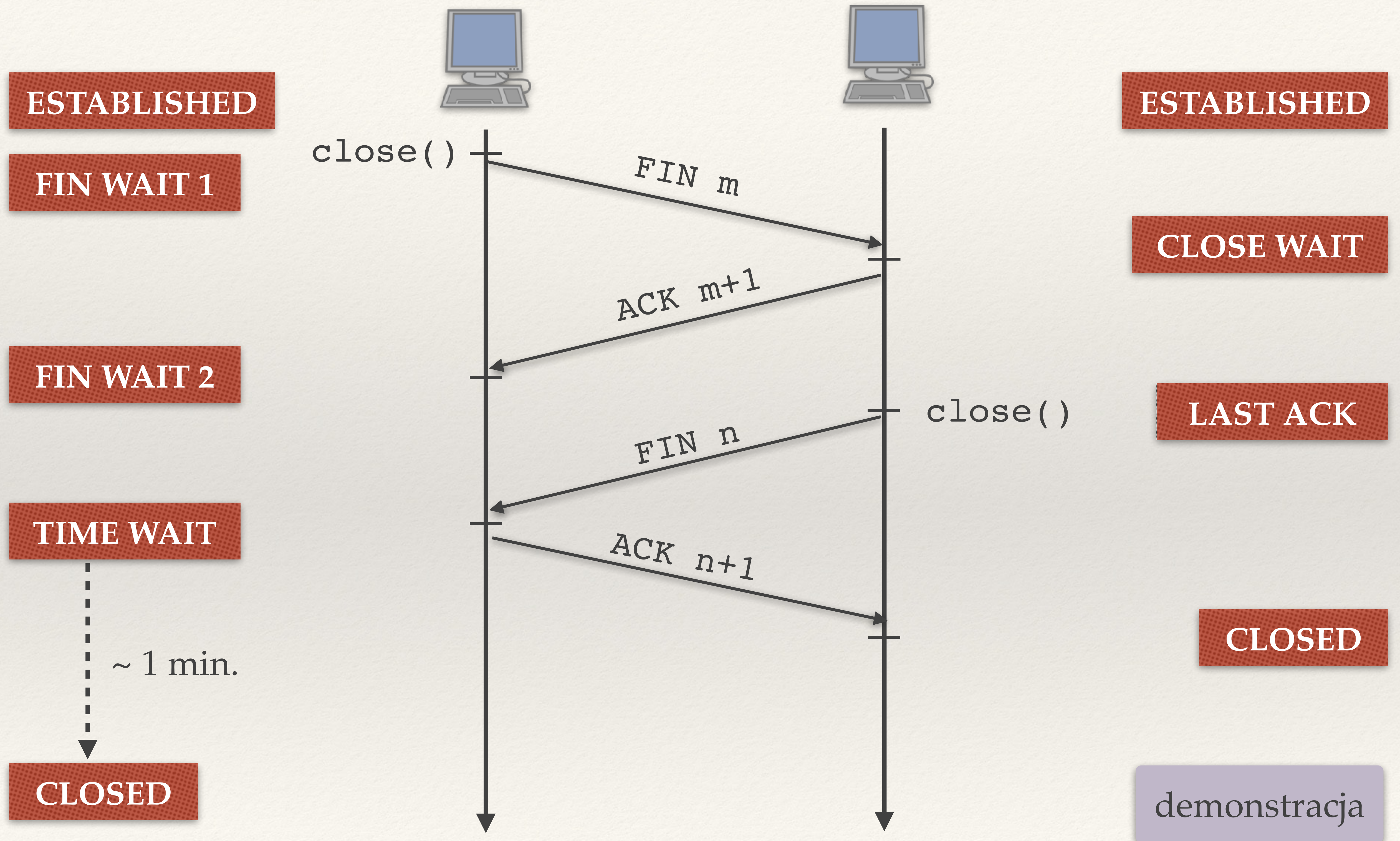
Czterofazowe kończenie połączenia



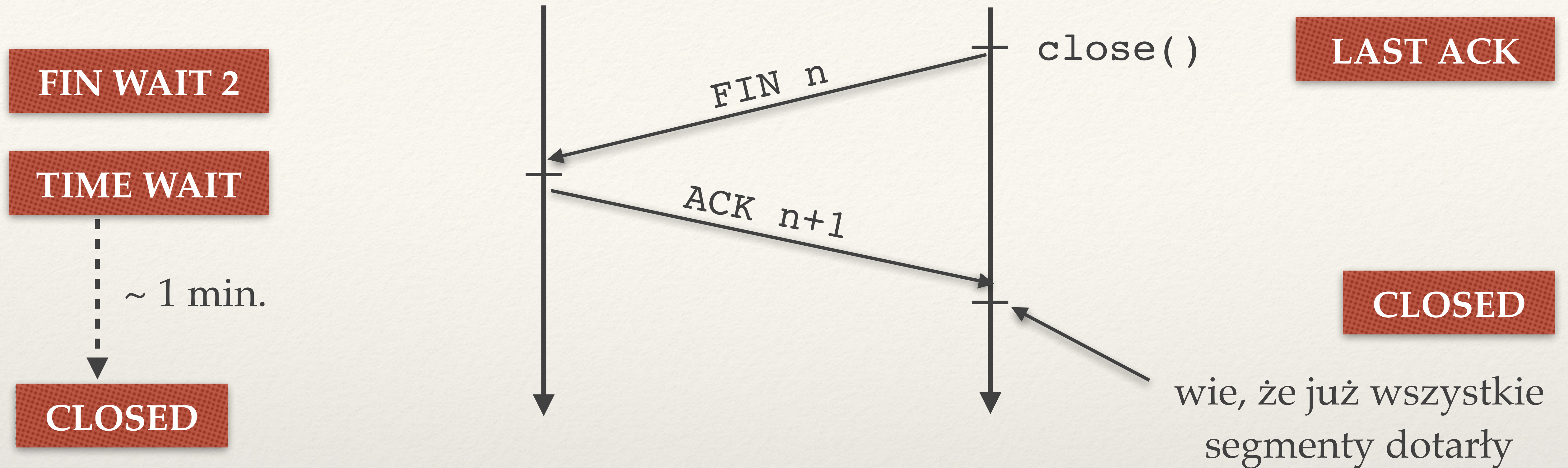
Czterofazowe kończenie połączenia



Czterofazowe kończenie połączenia

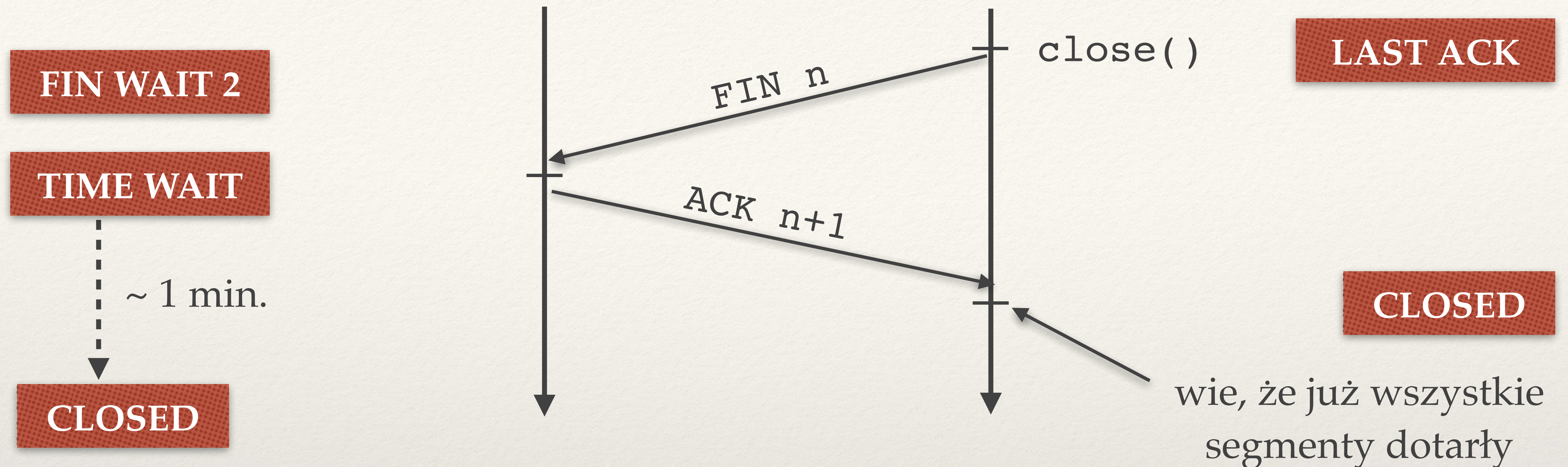


Stan TIME WAIT



- Lewa strona nie wie, czy prawa strona dostała jej `ACK`.

Stan TIME WAIT



- Lewa strona nie wie, czy prawa strona dostała jej `ACK`.
- Jeśli końcowy `ACK` nie dotrze, prawa strona wyśle `FIN` jeszcze raz. Lewa strona chce go poprawnie obsłużyć.
- Nie chcemy też żeby można było szybko utworzyć połączenie TCP o takich samych parametrach (IP + porty) → duplikaty starych segmentów zostałyby uznane za należące do nowego połączenia.

Stany TCP: sytuacje nietypowe

unusual event
client/receiver path
server/sender path
(Start) CLOSED
CONNECT/SYN (Step 1 of the 3-way-handshake)
CLOSE/-
LISTEN/-
CLOSE/-
LISTEN
(Step 2 of the 3-way-handshake) SYN/SYN+ACK
RST/-
SEND/SYN
SYN RECEIVED
SYN/SYN+ACK (simultaneous open)
SYN SENT
Data exchange occurs
ACK/-
ESTABLISHED
SYN+ACK/ACK
(Step 3 of the 3-way-handshake)
CLOSE/FIN
CLOSE/FIN
FIN/ACK
Active CLOSE
Passive CLOSE
FIN WAIT 1
FIN/ACK
CLOSING
CLOSE WAIT
FIN+ACK/ACK
ACK/-
ACK/-
CLOSE/FIN
FIN WAIT 2
TIME WAIT
LAST ACK
FIN/ACK
Timeout
ACK/-
(Go back to start) CLOSED

Obrazek ze strony https://en.wikipedia.org/wiki/Transmission_Control_Protocol

Wysyłanie większych danych

Funkcja send()

Jak działa `tcp_client.c` przy wysyłaniu dużej ilości danych?

- `send(…, …, n, …)` zwraca, ile bajtów zapisano do bufora wysyłkowego.

Funkcja send()

Jak działa `tcp_client.c` przy wysyłaniu dużej ilości danych?

- `send(…, …, n, …)` zwraca, ile bajtów zapisano do bufora wysyłkowego.
- To może być mniej niż *n* i nie jest to błąd.
- Wysłanych przez jądro może być jeszcze mniej…
- … a odebranych przez `recv()` jeszcze mniej.

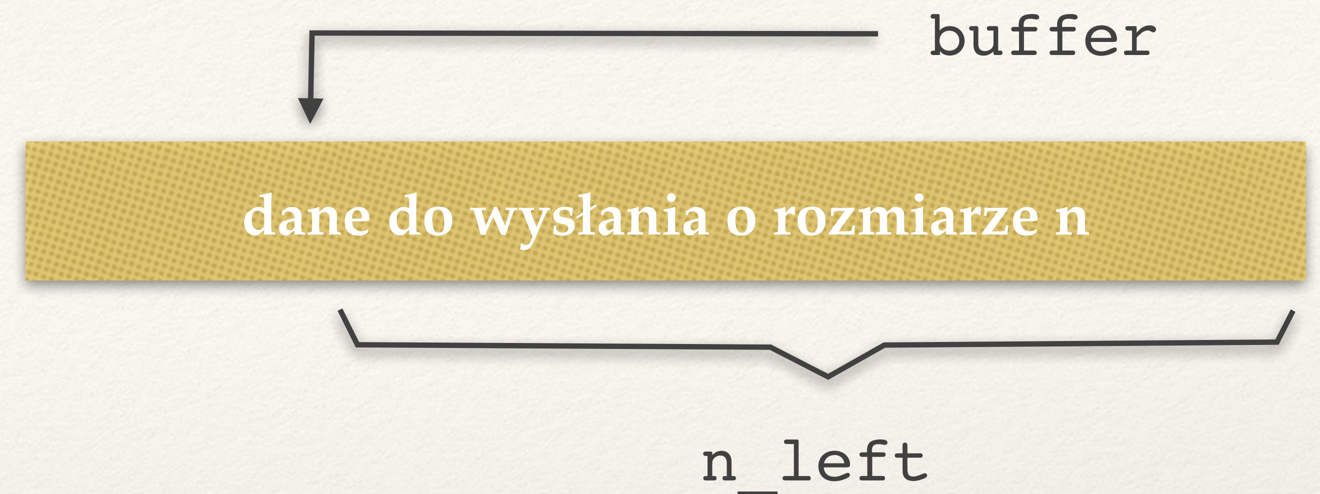
Funkcja send()

Jak działa `tcp_client.c` przy wysyłaniu dużej ilości danych?

- `send(…, …, n, …)` zwraca, ile bajtów zapisano do bufora wysyłkowego.
- To może być mniej niż *n* i nie jest to błąd.
- Wysłanych przez jądro może być jeszcze mniej…
- … a odebranych przez `recv()` jeszcze mniej.

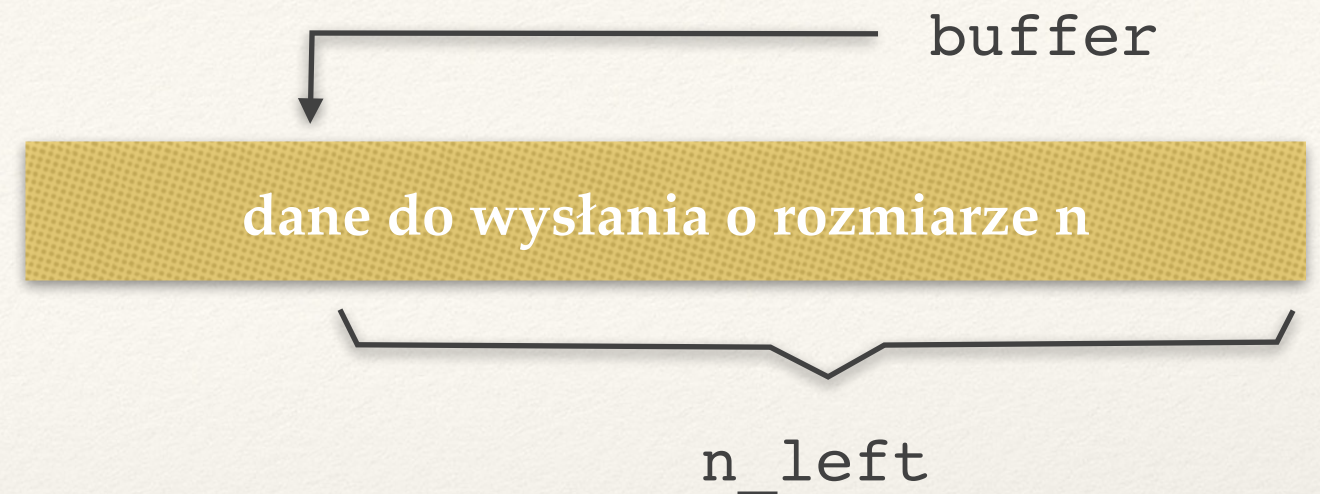
demonstracja

Poprawka: wysyłanie do skutku




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

Poprawka: wysyłanie do skutku




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

kod `tcp_client_fixed.c` na stronie wykładu

demonstracja

Funkcja recv()

Czy `tcp_client_fixed.c` naprawia problem?

- `tcp_server.c` wciąż czyta tylko część z wysyłanych danych.

Funkcja recv()

Czy `tcp_client_fixed.c` naprawia problem?

- `tcp_server.c` wciąż czyta tylko część z wysyłanych danych.
- Dodatkowo aplikacja serwera zamyka gniazdo, a klient wciąż wysyła dane.
  - Serwer odbiera segment danych po zamknięciu gniazda.
  - Serwer wysyła segment `RST`, klient go odbiera.
  - Zapis do gniazda, które otrzymało `RST` → otrzymanie `SIGPIPE` od jądra.
  - `SIGPIPE` można zignorować, ale to nie naprawia problemu.

Funkcja recv()

Czy `tcp_client_fixed.c` naprawia problem?

- `tcp_server.c` wciąż czyta tylko część z wysyłanych danych.
- Dodatkowo aplikacja serwera zamyka gniazdo, a klient wciąż wysyła dane.
  - Serwer odbiera segment danych po zamknięciu gniazda.
  - Serwer wysyła segment `RST`, klient go odbiera.
  - Zapis do gniazda, które otrzymało `RST` → otrzymanie `SIGPIPE` od jądra.
  - `SIGPIPE` można zignorować, ale to nie naprawia problemu.

demonstracja

Problem: nie zdefiniowaliśmy protokołu komunikacji

Do jakiego momentu `recv()` powinno czytać dane?

- Opcja 1: na początku wysyłamy rozmiar danych.
- Opcja 2: ustalamy **znacznik końca rekordu** i czytamy dane przez `recv()` aż do napotkania takiego znacznika.
  - Opcja 2a: czytamy aż `recv()` zwróci 0.
    Działa jeśli protokół to jedno pytanie i jedna odpowiedź.

Problem: nie zdefiniowaliśmy protokołu komunikacji

Do jakiego momentu `recv()` powinno czytać dane?

- Opcja 1: na początku wysyłamy rozmiar danych.
- Opcja 2: ustalamy **znacznik końca rekordu** i czytamy dane przez `recv()` aż do napotkania takiego znacznika.
  - Opcja 2a: czytamy aż `recv()` zwróci 0.
    Działa jeśli protokół to jedno pytanie i jedna odpowiedź.

kod `tcp_server_fixed.c` na stronie wykładu

demonstracja

Uwagi końcowe

- `tcp_server_fixed.c` jest pewną (nie jedyną) możliwością ustalenia protokołu komunikacyjnego.
- W rzeczywistych zastosowaniach potrzebujemy m.in.:
  - obsługi wielu klientów jednocześnie:
    osobne wątki lub/i wykorzystanie funkcji typu `poll()`,
  - rozłączania wolnych klientów po jakimś czasie.

Lektura dodatkowa

- Kurose & Ross: rozdział 3
- Tanenbaum: rozdział 6
- Stevens: rozdziały 3–6, 13, 27
- Beej's Guide to Network Programming: 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 funkcję 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.