Sieci komputerowe

Wykład 7: Transport: protokół TCP

Marcin Bieńkowski

Instytut Informatyki
Uniwersytet Wrocławski

W poprzednim odcinku

Niezawodny transport

- Algorytmy niezawodnego dostarczania danych: stop-and-wait, okno przesuwne.
- Potwierdzanie: go-back-N, selektywne, skumulowane.
- Kontrola przepływu: odbiorca wysyła rozmiar oferowanego okna $\rightarrow$ reguluje rozmiar okna nadawcy
- TCP: okno przesuwne + potwierdzanie skumulowane + numerowanie bajtów.

Dzisiaj

Inne szczegóły techniczne TCP
+ programowanie gniazd TCP

Przykład

Łączymy się z serwerem (np. serwerem WWW)

- interaktywnie $\rightarrow$ `telnet`
- nieinteraktywnie $\rightarrow$ `nc`
- Uwaga: oba powyższe programy działają w warstwie aplikacji!

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

- Dobrze znane porty (*well known ports*).
- Niektóre usługi mają porty zarezerwowane przez standardy: przykładowo wszystkie serwery WWW odbierają połączenia na porcie 80.
- $\rightarrow$ `/etc/services`

Przykład

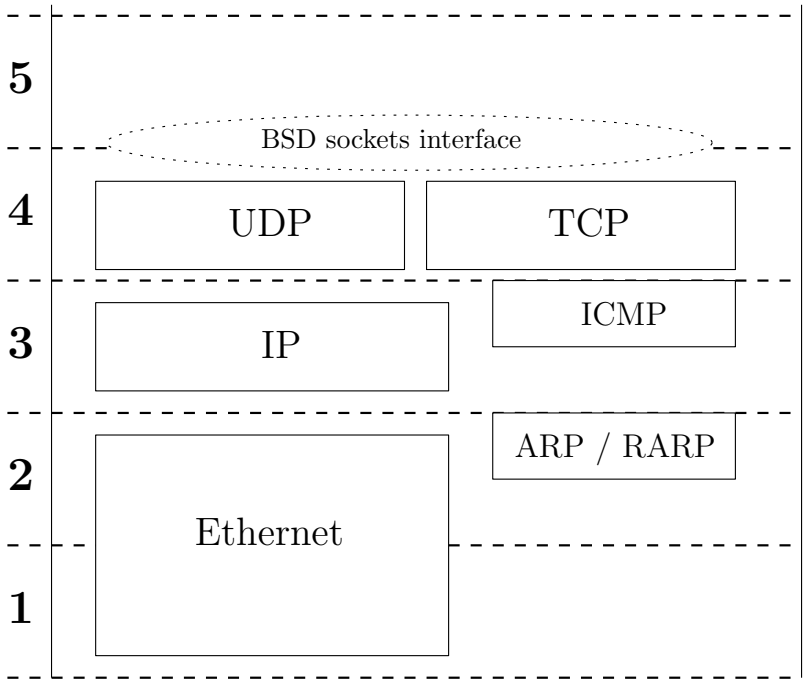
Łączymy się z serwerem (np. serwerem WWW)

- interaktywnie $\rightarrow$ `telnet`
- nieinteraktywnie $\rightarrow$ `nc`
- Uwaga: oba powyższe programy działają w warstwie aplikacji!

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

- Dobrze znane porty (*well known ports*).
- Niektóre usługi mają porty zarezerwowane przez standardy: przykładowo wszystkie serwery WWW odbierają połączenia na porcie 80.
- $\rightarrow$ `/etc/services`

Interfejs programistyczny



Interfejs programistyczny: *BSD sockets*

- Dobre 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 jest zazwyczaj efektywniejsza niż w przypadku bezpołączeniowym.
- Na końcu trzeba *zakończyć połączenie*.
- Przykładowo: telefon.

Gniazda UDP

UDP

- Gniazdo jest związane z konkretnym procesem.
- Gniazdo jest 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.

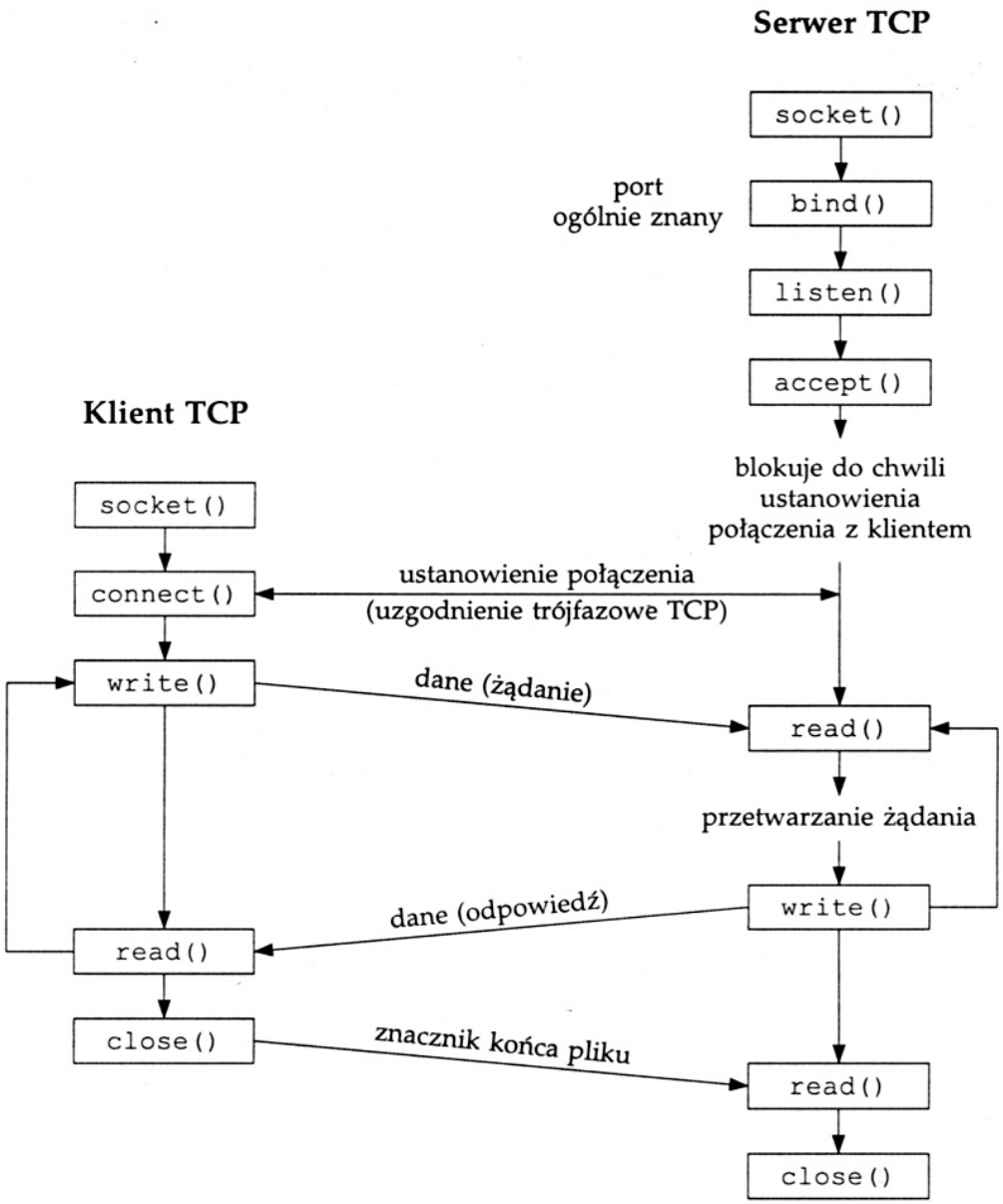
Gniazda TCP

TCP: gniazda nasłuchujące

- Tylko dla serwera
- Tylko do nawiązywania połączeń
- Przykładowo (lokalnie) `156.17.4.30:80` – (zdalnie) `*:*`.

TCP: gniazda połączone

- Tworzone dla klienta i serwera po połączeniu.
- Do wymiany właściwych danych.
- Przykładowo po stronie serwera:
  (lokalnie) `156.17.4.30:80` – (zdalnie) `22.33.44.55:44444`.
- Przykładowo po stronie klienta:
  (lokalnie) `22.33.44.55:44444` – (zdalnie) `156.17.4.30:80`.



Rys. 4.1. Funkcje gniazd dla podstawowej komunikacji klient–serwer

Obrazek z książki *Programowanie usług sieciowych*, R. Stevens

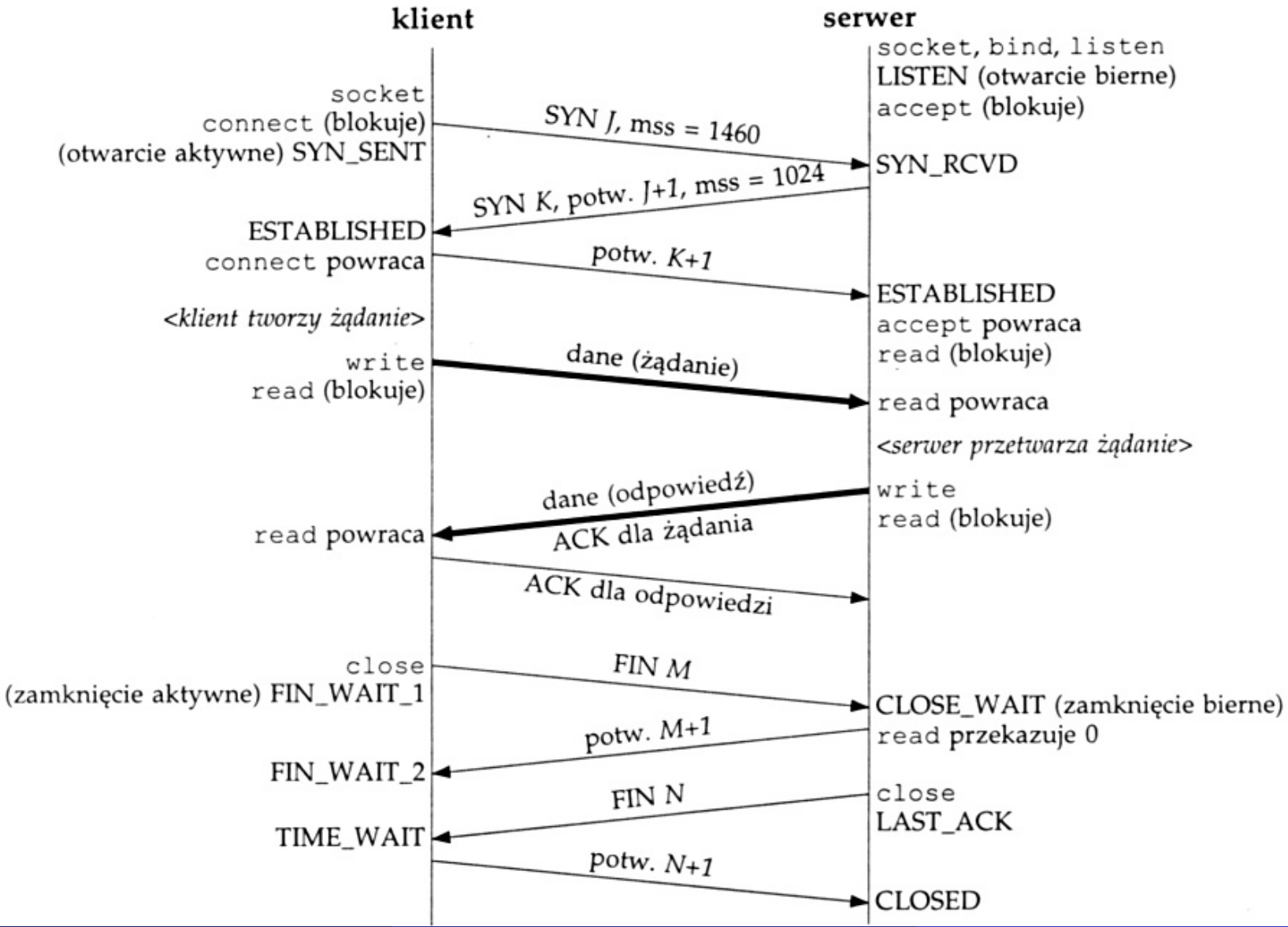
Przykład serwera i klienta TCP

- $\rightarrow$ server1.c + telnet/netcat
- $\rightarrow$ server1.c + client1.c

Cykl życia połączenia (1)

- Trójfazowe nawiązywanie połączenia
- Przesyłanie danych
- Czterofazowe kończenie połączenia

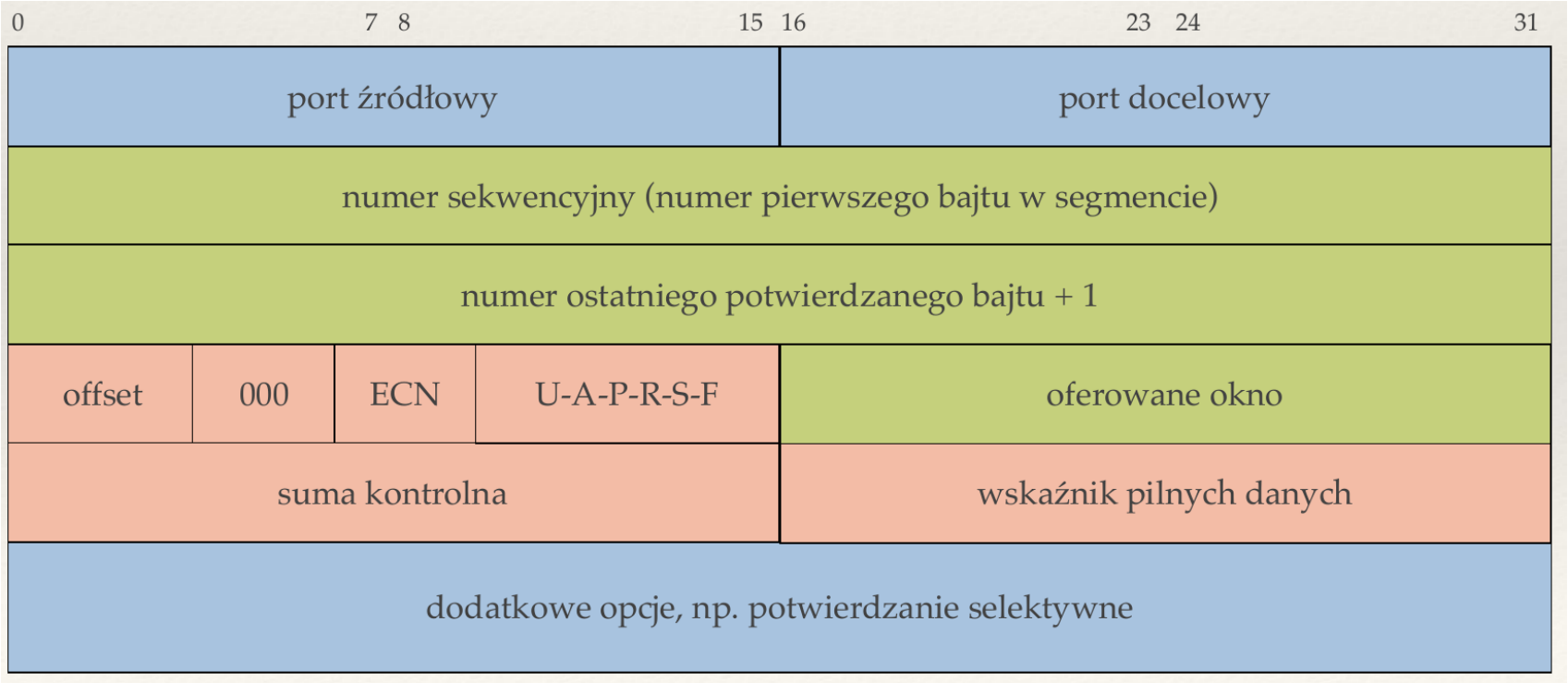
Cykl życia połączenia (2)



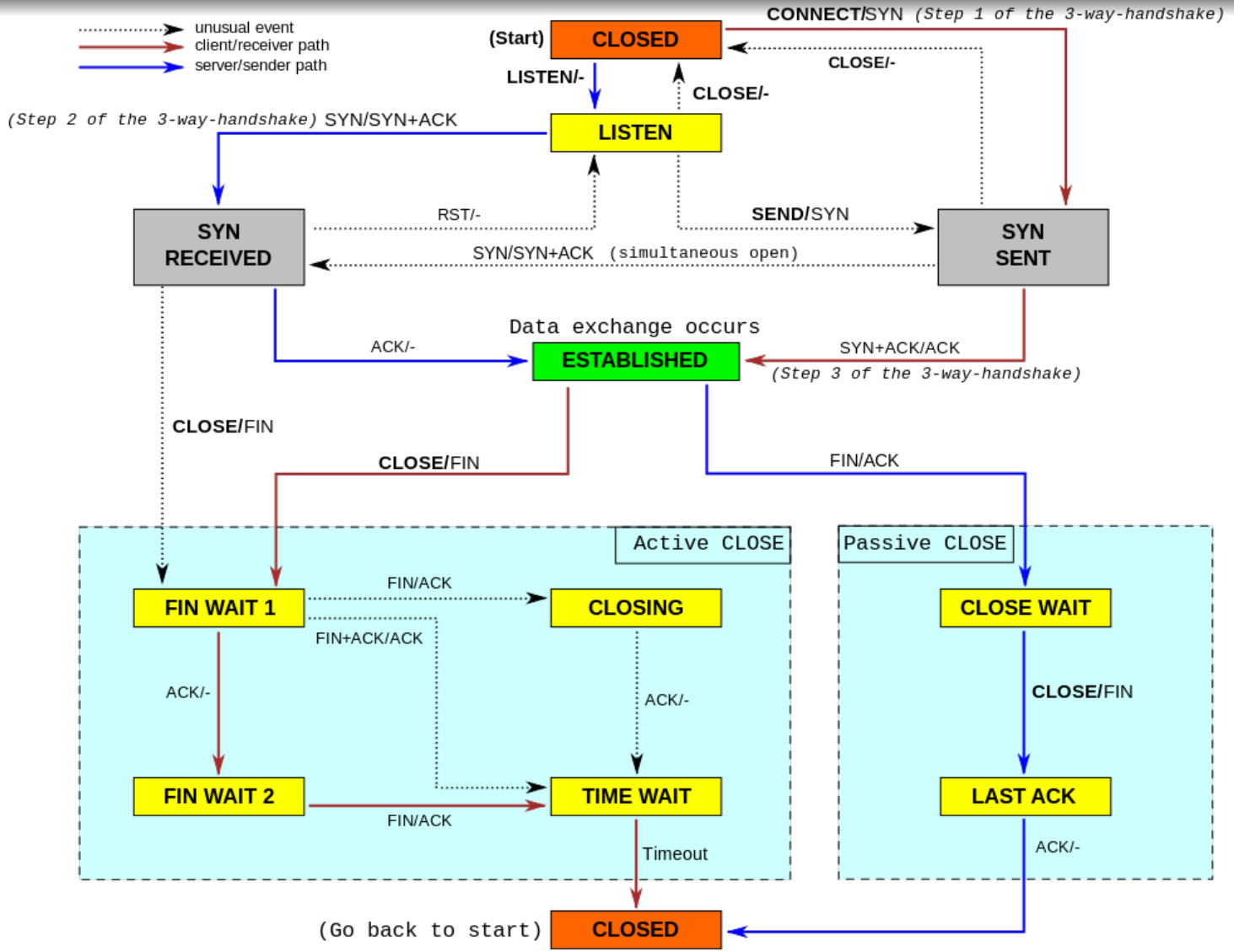
Gniazda TCP (jeszcze raz)

- Po otwarciu biernym `bind()` + `listen()`: tworzone gniazdo nasłuchujące
- Funkcja `accept()` zwraca gniazdo połączone (po trójfazowym nawiązaniu połączenia).
- Funkcja `connect()` zwraca gniazdo połączone po stronie klienta.

Segment 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

Stany TCP



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

Segment RST

- Wysyłany kiedy wystąpi błąd.
- Przykładowo w odpowiedzi na dowolny segment wysłany do zamkniętego portu.

Stan TIME WAIT

- Kończenie połączenia nie jest do symetryczne:
  - strona robiąca zamknięcie bierne: pełna wiedza (dostaje ACK na wysłany FIN)
  - strona robiąca zamknięcie aktywne: nie wie czy druga strona dostała jej ACK $\rightarrow$ stan TIME WAIT
- TIME WAIT utrzymuje się przez 1-4 min. (2 $\times$ max. czas życia segmentu).

  Cel 1: Prawidłowe zakończenie połączenia TCP w przypadku błędów: jeśli końcowy ACK nie dociera, to druga strona wyśle FIN jeszcze raz $\rightarrow$ chcemy go poprawnie obsłużyć.

  Cel 2: Usunięcie starych duplikatów segmentów z sieci.

Funkcja send()

Czy nasz klient działa poprawnie?

- Wyślijmy 100.000 bajtów...
- Po pierwsze `send()` może wysłać mniej i to nie jest błąd!
- Co więcej, `send()` zapisuje dane tylko do bufora wysyłkowego, zakończenie tej funkcji nie oznacza faktycznego wysłania.

Funkcja send()

Czy nasz klient działa poprawnie?

- Wyślijmy 100.000 bajtów...
- Po pierwsze `send()` może wysłać mniej i to nie jest błąd!
- Co więcej, `send()` zapisuje dane tylko do bufora wysyłkowego, zakończenie tej funkcji nie oznacza faktycznego wysłania.

Funkcja recv()

Nowa wersja klienta

- $\rightarrow$ server1.c + client2.c
- Wyślijmy 1.000.000 bajtów...
- Klient zostaje zabity przez SIGPIPE?!
  - klient: `send()` zapisuje część lub całość do bufora wysyłkowego.
  - TCP po stronie klienta wysyła kilka segmentów (np. po 16 KB).
  - TCP po stronie serwera potwierdza część lub wszystkie segmenty
  - serwer: `recv()` odczytuje pewien fragment bufora odbiorczego (np. 32 KB)
  - serwer: wysyła odpowiedź i zamyka połączenie.
  - klient: `send()` wysyła kolejny segment,
    - stos TCP po stronie serwera odpowiada segmentem RST.
    - stos TCP po stronie klienta zamyka gniazdo.
  - klient: `send()` wysyła kolejny segment usiłując zapisać do zamkniętego gniazda $\rightarrow$ otrzymuje sygnał `SIGPIPE`.

Funkcja recv()

Nowa wersja klienta

- $\rightarrow$ server1.c + client2.c
- Wyślijmy 1.000.000 bajtów...
- Klient zostaje zabity przez SIGPIPE?!
  - klient: `send()` zapisuje część lub całość do bufora wysyłkowego.
  - TCP po stronie klienta wysyła kilka segmentów (np. po 16 KB).
  - TCP po stronie serwera potwierdza część lub wszystkie segmenty
  - serwer: `recv()` odczytuje pewien fragment bufora odbiorczego (np. 32 KB)
  - serwer: wysyła odpowiedź i zamyka połączenie.
  - klient: `send()` wysyła kolejny segment,
    - stos TCP po stronie serwera odpowiada segmentem RST.
    - stos TCP po stronie klienta zamyka gniazdo.
  - klient: `send()` wysyła kolejny segment usiłując zapisać do zamkniętego gniazda $\rightarrow$ otrzymuje sygnał `SIGPIPE`.

Funkcja recv()

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

- Nie zdefiniowaliśmy protokołu komunikacji.
- Podejście nr 1: ustalamy znacznik końca rekordu (koniec wiersza) i czytamy do tego znacznika.
- Podejście nr 2: na początku wysyłamy rozmiar danych.

Nieblokujący odczyt (1)

- Wywołanie funkcji `recv()` w przypadku gniazda w ktorym nic nie ma blokuje do momentu, kiedy coś się tam pojawi.
  - Co z klientami, którzy chcą zablokować dostęp do usługi?
- Tak jak w przypadku gniazd UDP możemy wykorzystać flagę `MSG_DONTBLOCK`.
- Jak często sprawdzać, czy coś jest w gnieździe?
  - Często $\rightarrow$ duże zużycie CPU
  - Rzadko $\rightarrow$ potencjalnie duże opóźnienie
  - A może usnąć do momentu aż coś się pojawi?

Nieblokujący odczyt (2)

- Funkcja `select()`: obserwuje jedno lub wiele gniazd maksymalnie przez zadany czas,
- Powrót z funkcji jeśli coś jest gotowe (np. do odczytu).
- Zwraca liczbę gotowych do odczytu gniazd, 0 jeśli nastąpił timeout.
- Kolejne wywołanie `recv()` na gotowym gnieździe nie zablokuje (choć może odczytać mniej bajtów niż chcemy).
- $\rightarrow$ `server2.c`

Lektura dodatkowa

- Kurose, Ross: rozdział 3
- Tanenbaum: rozdział 6
- Stevens: rozdziały 3-6, 13, 27