Routing

część 1: adresowanie

Sieci komputerowe

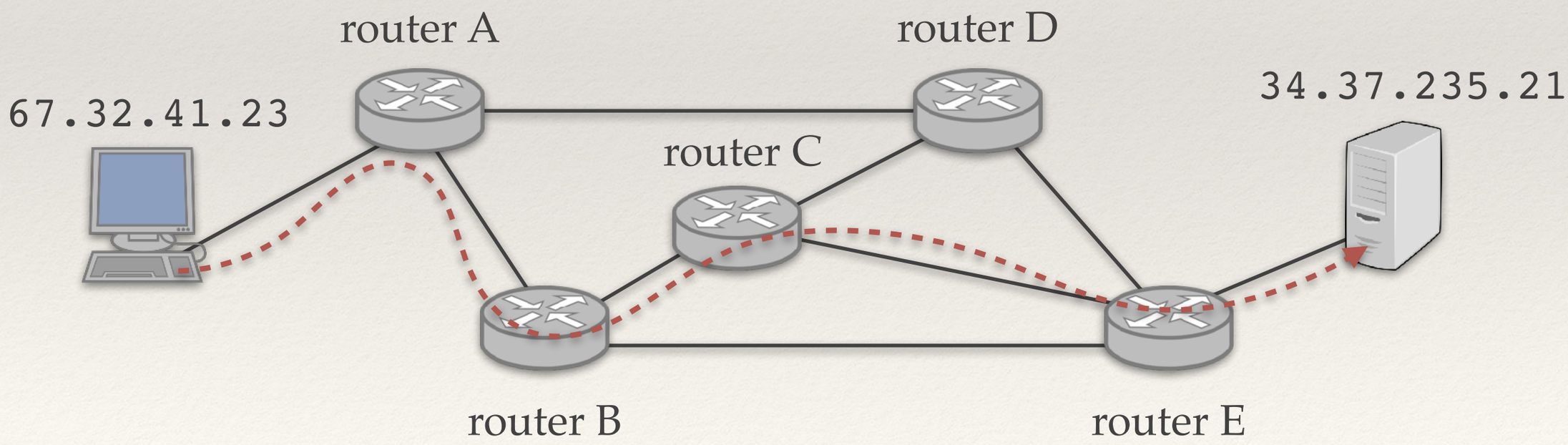
Wykład 2

Marcin Bieńkowski

W poprzednim odcinku

Jak przesyłać dane przez sieć

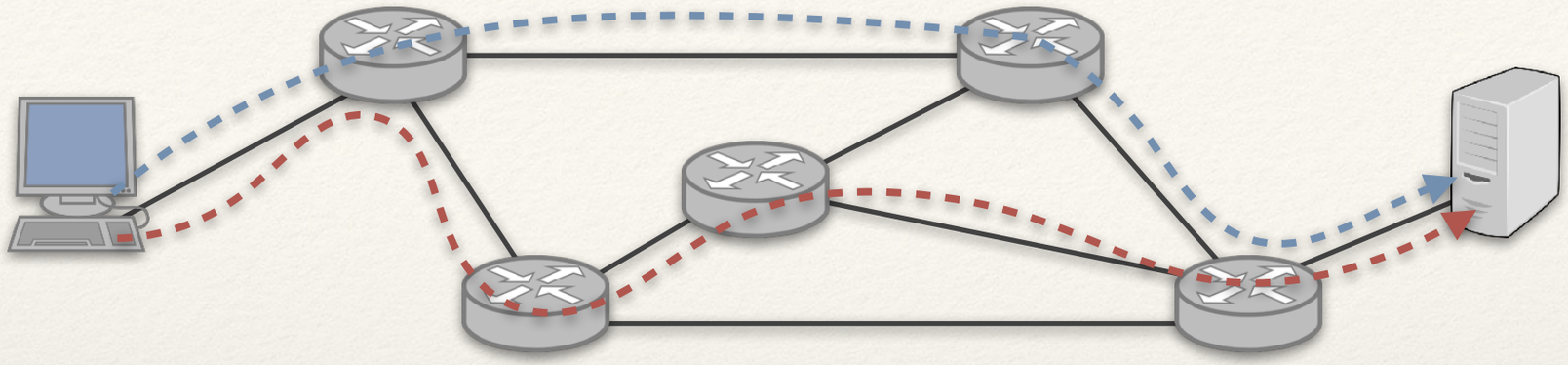
- Chcemy przesyłać między aplikacjami strumień danych.
- **Globalne adresowanie**: w Internecie każda karta sieciowa ma unikatowy 4-bajtowy adres IP.
- Warstwa sieciowa zapewnia globalne dostarczanie danych pomiędzy dwoma dowolnymi kartami sieciowymi = interfejsami sieciowymi.



Przełączanie pakietów

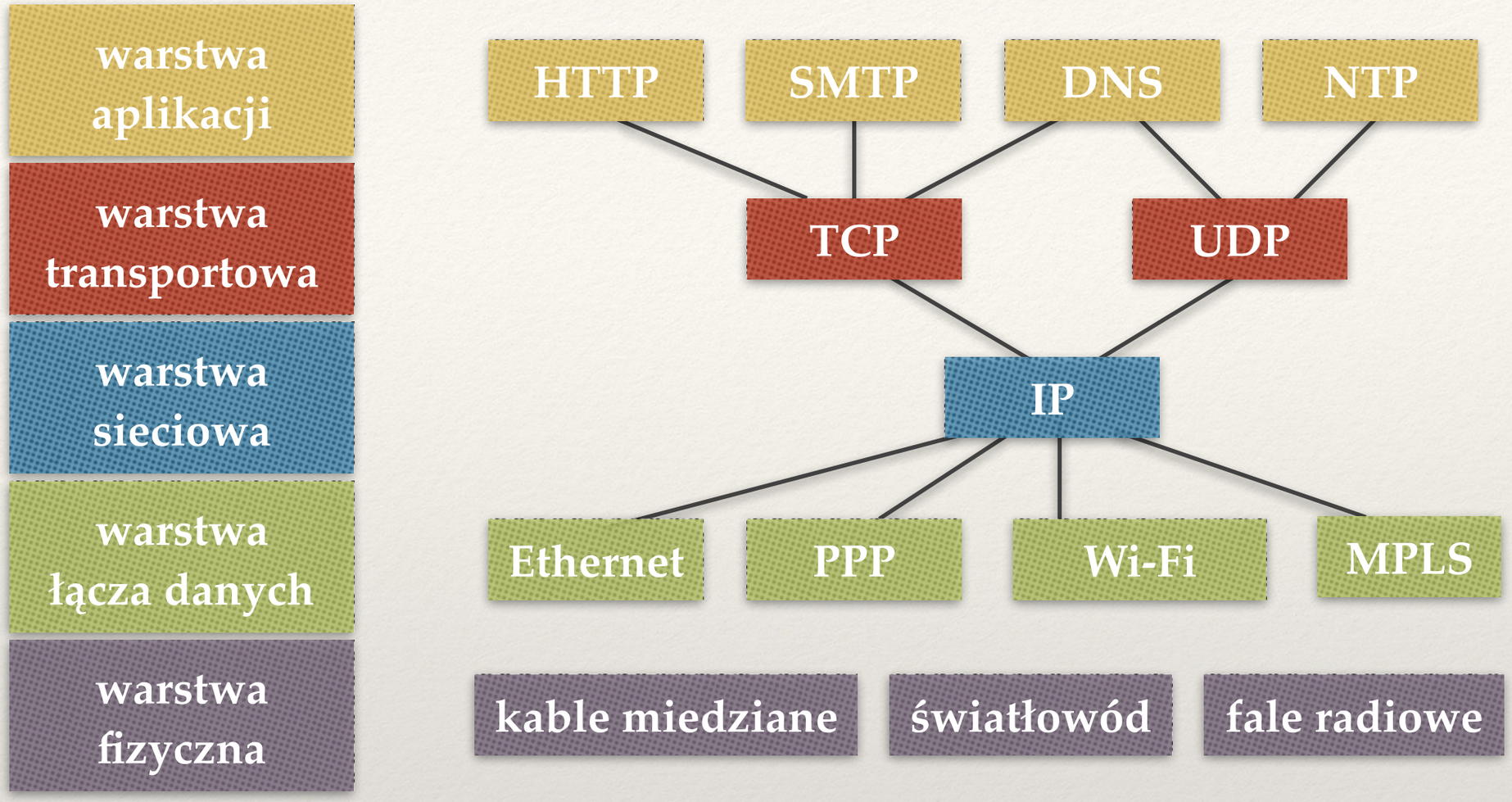
- Wysyłany strumień danych dzielimy na małe porcje: **pakiety.**
- Dane pakietu = fragment strumienia danych.
- Nagłówek pakietu = informacje kontrolne, m.in. adres źródłowy i docelowy.
- Każdy pakiet przesyłany niezależnie.

Routing



- Routing (trasowanie) = wybór trasy dla danego pakietu.
- Router tylko przekazuje pakiet dalej.
- Router nie wie nic o oryginalnym strumieniu danych.
- Router podejmuje decyzję na podstawie nagłówka pakietu w oparciu o **tablice routingu**.

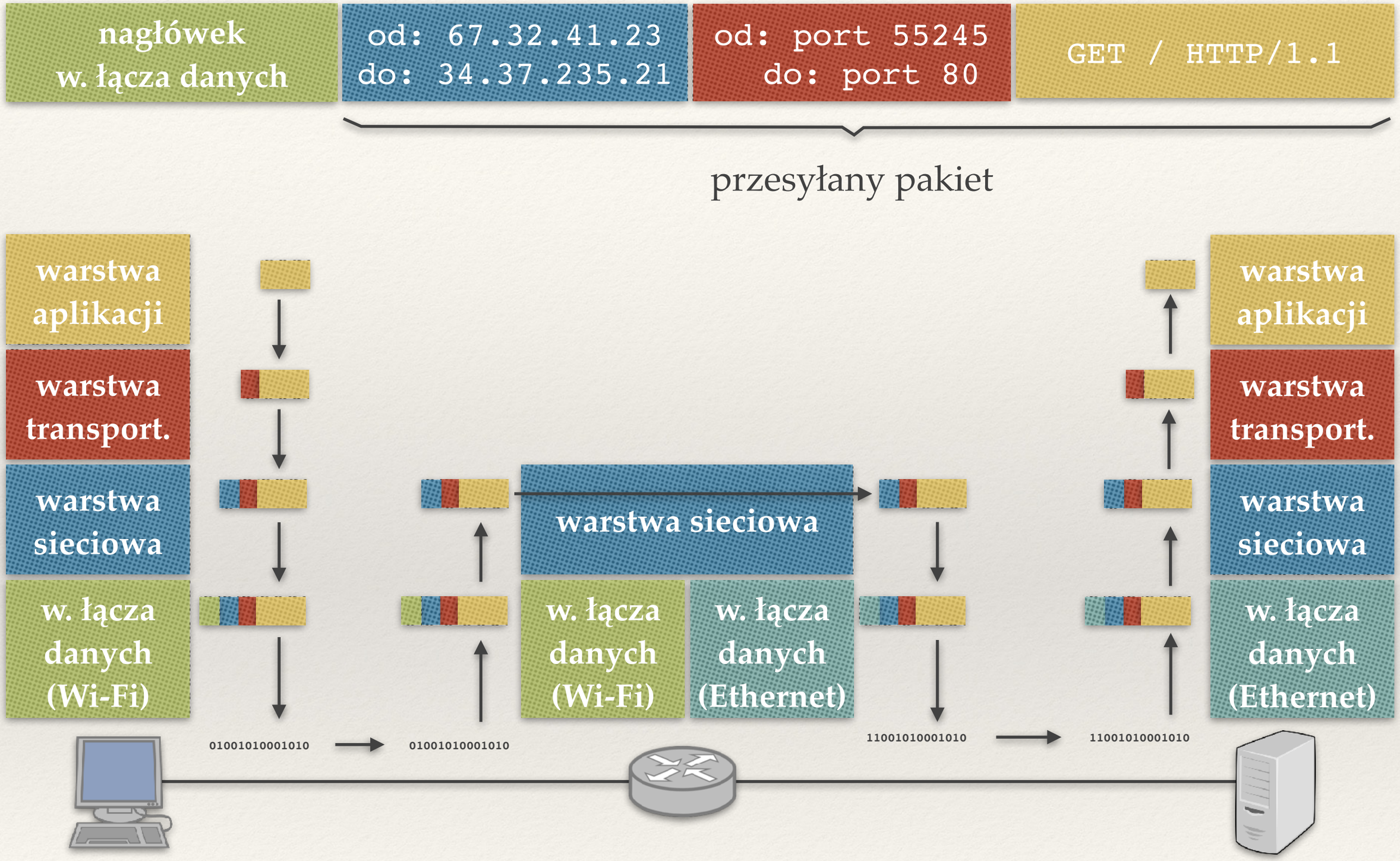
Protokoły w Internecie



Warstwa sieciowa w Internecie: tylko jeden protokół (IP).

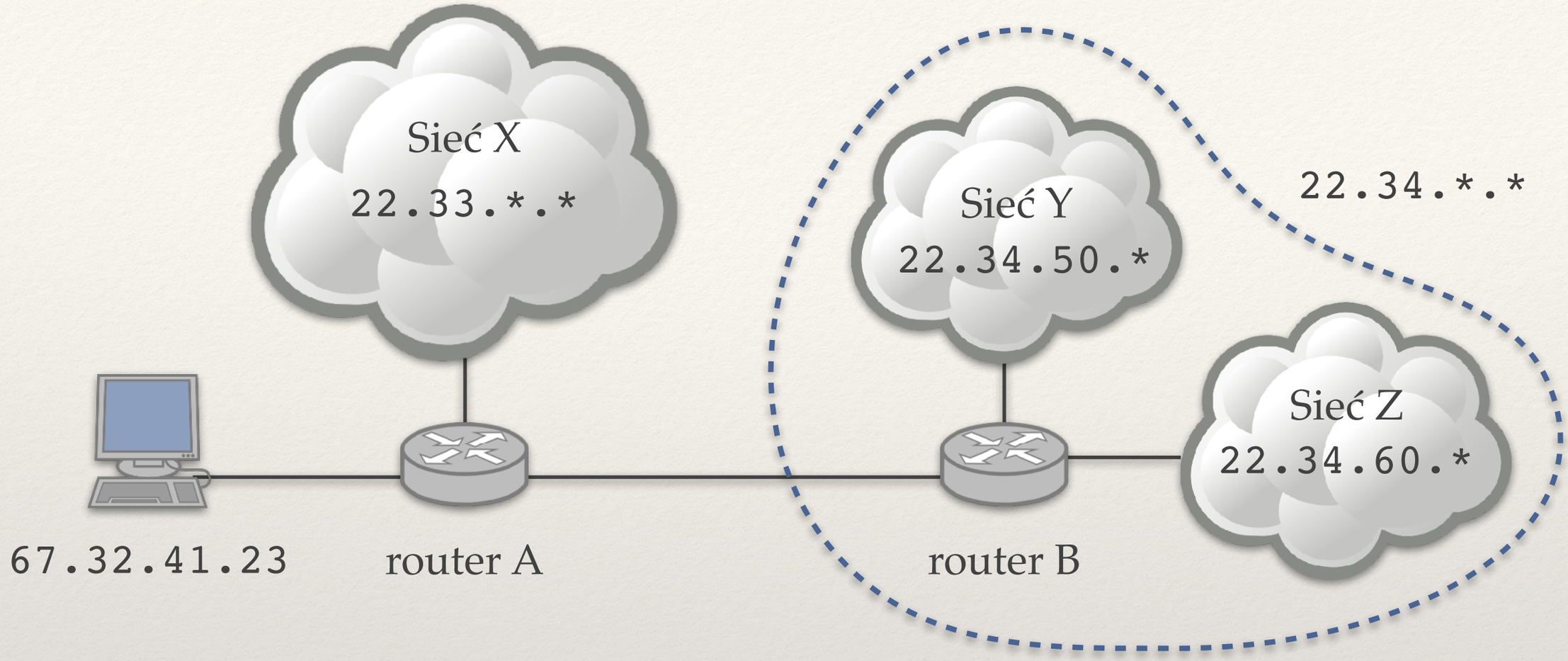
- Zaimplementowany na każdym urządzeniu.
- Definiuje zawodną, bezpołączeniową usługę umożliwiającą przesłanie pakietu między dwoma dowolnymi urządzeniami w sieci.

Internetowy model warstwowy



Adresowanie

Adresy IP



- Każda karta sieciowa ma unikatowy 4-bajtowy adres.
- Adresy mają hierarchiczną strukturę:
  - Router *A* nie musi znać trasy do sieci *Y* i *Z* osobno; wystarczy, że wie że pakiety do `22.34.*.*` powinien wysyłać do routera *B*.

CIDR

- **Notacja CIDR** (*Classless Inter-Domain Routing*)
  - opisuje zakres adresów IP posiadających wspólny prefiks za pomocą pary (pierwszy adres z zakresu, długość prefiksu).

- Adresy IP zaczynające się od prefiksu
  `10011100.00010001.00000100.0010`
  - pierwszy adres z zakresu:
    `10011100.00010001.00000100.0010`**`0000`** `= 156.17.4.32`
  - długość prefiksu: 28 bitów
  - zapis: `156.17.4.32/28`

CIDR: przykłady

- `156.17.4.32/28` = adresy zaczynające się od prefiksu `156.17.4.0010`:
  - `156.17.4.0010`**`0000`** = `156.17.4.32` (pierwszy adres)
  - `156.17.4.0010`**`0001`** = `156.17.4.33`
  - …
  - `156.17.4.0010`**`1110`** = `156.17.4.46`
  - `156.17.4.0010`**`1111`** = `156.17.4.47` (ostatni adres)
  - Razem: $2^{32-28} = 2^4 = 16$ adresów.

CIDR: przykłady

- `156.17.4.32/28` = adresy zaczynające się od prefiksu `156.17.4.0010`:
  - `156.17.4.00100000` = `156.17.4.32` (pierwszy adres)
  - `156.17.4.00100001` = `156.17.4.33`
  - …
  - `156.17.4.00101110` = `156.17.4.46`
  - `156.17.4.00101111` = `156.17.4.47` (ostatni adres)
  - Razem: $2^{32-28} = 2^4 = 16$ adresów.
- `0.0.0.0/0` = wszystkie adresy IP

CIDR: przykłady

- `156.17.4.32/28` = adresy zaczynające się od prefiksu `156.17.4.0010`:
  - `156.17.4.00100000` = `156.17.4.32` (pierwszy adres)
  - `156.17.4.00100001` = `156.17.4.33`
  - …
  - `156.17.4.00101110` = `156.17.4.46`
  - `156.17.4.00101111` = `156.17.4.47` (ostatni adres)
  - Razem: $2^{32-28} = 2^4 = 16$ adresów.

- `0.0.0.0/0` = wszystkie adresy IP
- `34.56.78.90/32` = jeden konkretny adres IP.

Jeśli notacja CIDR opisuje sieć

- Ostatni adres jest zarezerwowany: **adres rozgłoszeniowy (*broadcast*)**.
  - Pakiet wysłany na ten adres dotrze do wszystkich adresów IP z zakresu.

- Pierwszy adres jest zarezerwowany: tzw. **adres sieci.**
  - Względy historyczne (to był początkowo adres rozgłoszeniowy).

- Reszta adresów może być przypisana do kart sieciowych w tej sieci.

Podsieci

`156.17.4.32/28` to zbiór 16 adresów:

- `156.17.4.00100000 = 156.17.4.32`
- …
- `156.17.4.00100111 = 156.17.4.39`

`156.17.4.32/29`

- `156.17.4.00101000 = 156.17.4.40`
- …
- `156.17.4.00101111 = 156.17.4.47`

`156.17.4.40/29`

Adres IP potrzebuje kontekstu

Czy adres `156.17.4.95` jest adresem rozgłoszeniowym?

Adres IP potrzebuje kontekstu

Czy adres `156.17.4.95` jest adresem rozgłoszeniowym?

- Tak w `156.17.4.80/28` = {`156.17.4.80`, ..., `156.17.4.95`}.
- Tak w `156.17.4.64/27` = {`156.17.4.64`, ...,`156.17.4.95`}.
- Nie w `156.17.4.64/26` = {`156.17.4.64`, ..., `156.17.4.127`}.

CIDR dla pojedynczych adresów IP

- Sieć X = `156.17.4.64/26` = {`156.17.4.64`, ..., `156.17.4.127`}
  - Adres A = `156.17.4.95` z sieci X.
  - A zapisujemy jako `156.17.4.95/26`.

CIDR dla pojedynczych adresów IP

- Sieć X = `156.17.4.64/26` = {`156.17.4.64`, ..., `156.17.4.127`}
  - Adres A = `156.17.4.95` z sieci X.
  - A zapisujemy jako `156.17.4.95/26`.

- Operacja odwrotna: mamy adres A = `156.17.4.95/26`.

  Jaka sieć X zawiera A?

  - Pierwszy adres z sieci X to

    $Y = A \;\&\; 1^{26}\,0^{6}$

    $= \texttt{156.17.4.01}\mathbf{011111} \;\&\; 1^{26}\,0^{6}$

    $= \texttt{156.17.4.01}\mathbf{000000}$

    $= \texttt{156.17.4.64}$.
  - Czyli X = `156.17.4.64/26`.

CIDR dla pojedynczych adresów IP

- Sieć X = `156.17.4.64/26` = {`156.17.4.64`, ..., `156.17.4.127`}
  - Adres A = `156.17.4.95` z sieci X.
  - A zapisujemy jako `156.17.4.95/26`.

- Operacja odwrotna: mamy adres A = `156.17.4.95/26`.

  Jaka sieć X zawiera A?

  - Pierwszy adres z sieci X to

    $Y = A \;\&\; 1^{26}\,0^{6}$

    $= \texttt{156.17.4.01}\mathbf{011111} \;\&\; 1^{26}\,0^{6}$

    $= \texttt{156.17.4.01}\mathbf{000000}$

    $= \texttt{156.17.4.64}.$
  - Czyli X = `156.17.4.64/26`.

Zostawiamy 26 bitów z A
i zerujemy pozostałe.

Długość prefiksu = maska (pod)sieci

- `/26` zapisujemy czasem jako `255.255.255.192`
  = `11111111.11111111.11111111.11000000`.

- Niektóre programy potrzebują takiej notacji lub/i w takiej wyświetlają.

Klasy adresów

- Jeśli nie podamy długości prefiksu, niektóre starsze polecenia wywnioskują ją z adresu IP:

```
ifconfig en0 10.0.0.1
= ip addr add 10.0.0.1/8 dev en0
```

- Przyczyny historyczne (klasy adresów IP).
  - Adres IP zaczyna się od `0` → długość prefiksu `/8` (klasa A).
  - Adres IP zaczyna się od `10` → długość prefiksu `/16` (klasa B).
  - Adres IP zaczyna się od `110` → długość prefiksu `/24` (klasa C).

Pętla zwrotna = sieć 127.0.0.0/8

- Interfejs `lo` (*loopback*).
- Łącząc się z dowolnym adresem z tej sieci (zazwyczaj z `127.0.0.1`), łączymy się z lokalnym komputerem.
- Testowanie aplikacji sieciowych bez połączenia z siecią.

Przykład konfiguracji

```
$> ip addr

enp2s0: <BROADCAST,MULTICAST> mtu 1500 state UP
        inet 156.17.4.30/24 brd 156.17.4.255 scope global

lo:     <LOOPBACK> mtu 65536 state UNKNOWN
        inet 127.0.0.1/8 scope host lo

wg0:    <BROADCAST,MULTICAST> mtu 1420 state UNKNOWN
        inet 192.168.15.1/28 scope global
```

Routing pakietów IP

Nagłówek pakietu IP

- 4 x IHL = długość nagłówka w bajtach.
- Protokół = co jest przechowywane w danych pakietu (np. 1 = ICMP, 6 = TCP, 17 = UDP).
- TTL = czas życia pakietu.

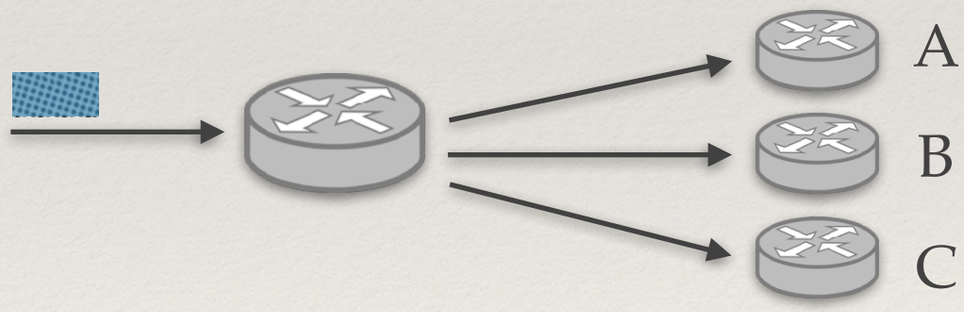
Przetwarzanie pakietu w routerze

- Obliczenie portu wyjściowego (wyjściowej „karty sieciowej”):
  - na podstawie adresu docelowego pakietu i posiadanej tablicy routingu.
- Aktualizacja nagłówka:
  - zmniejszenie TTL o 1; jeśli TTL = 0, to pakiet wyrzucany;
  - ponowne wyliczenie sumy kontrolnej pakietu.
- Przekazanie pakietu do kolejki wyjściowej.

Tablice routingu

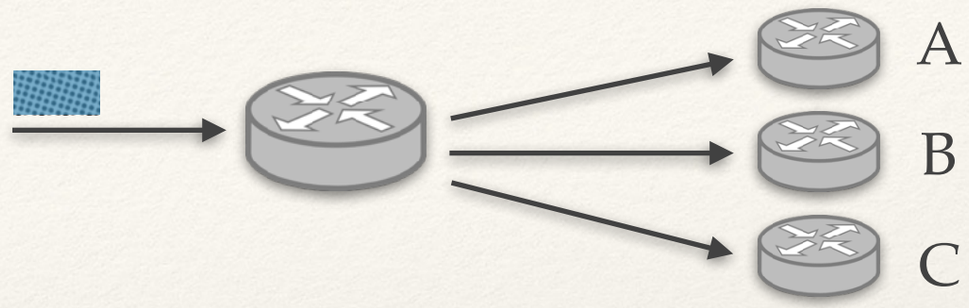
- Tablica routingu zawiera reguły typu „jeśli adres docelowy pakietu zaczyna się od prefiksu *A*, to wyślij pakiet do *X*“.
- Pakiet niepasujący do żadnej reguły jest odrzucany.

prefiks CIDR	akcja
`0.0.0.0/0`	do routera A
`156.17.4.0/24`	do routera B
`156.17.4.128/25`	do routera C
`156.17.4.128/26`	do routera B

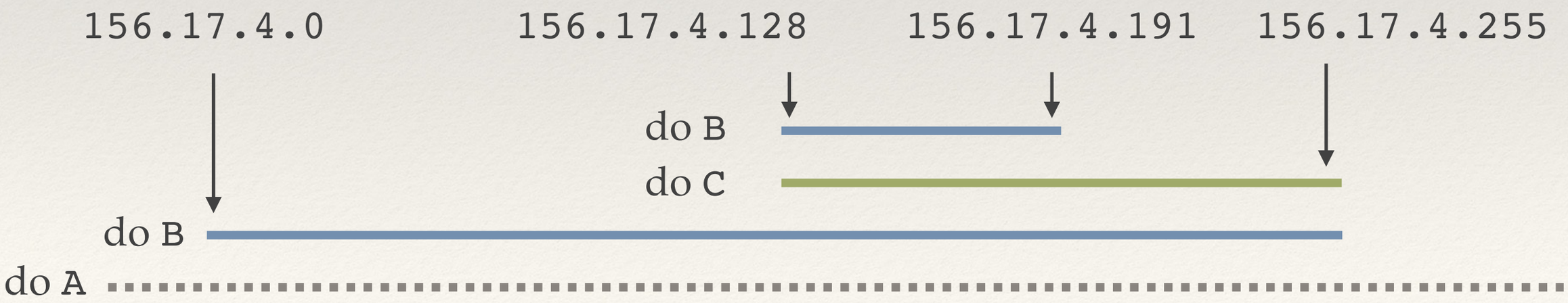


Reguła najdłuższego pasującego prefiksu

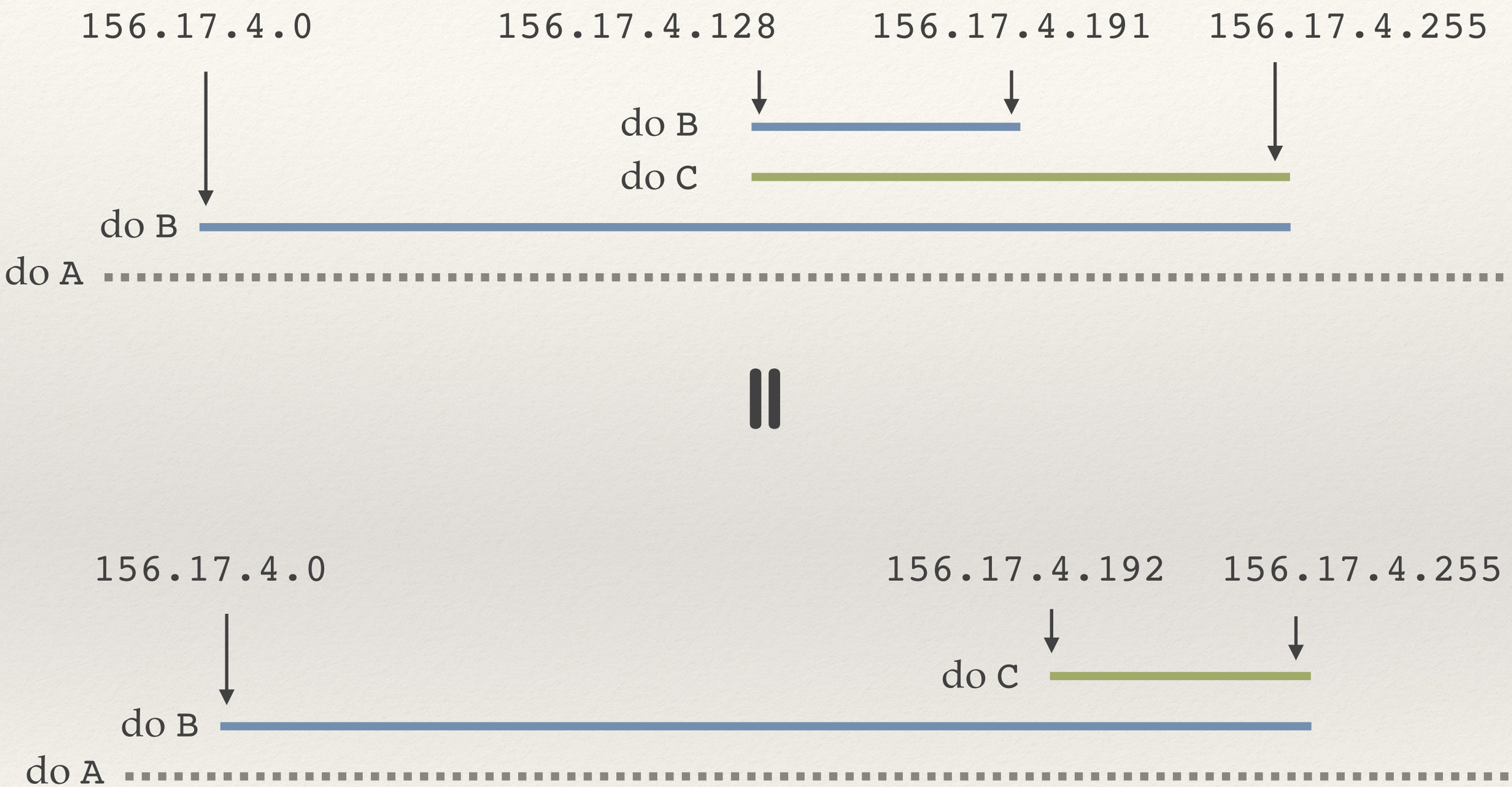
prefiks CIDR	akcja
0.0.0.0/0	do routera A
156.17.4.0/24	do routera B
156.17.4.128/25	do routera C
156.17.4.128/26	do routera B



- Jeśli więcej niż jedna reguła pasuje, wybierana jest ta, która jest **najdłuższym prefiksem** (najbardziej „konkretna reguła“).
- 0.0.0.0/0 = reguła (trasa) domyślna.



Równoważne tablice routingu

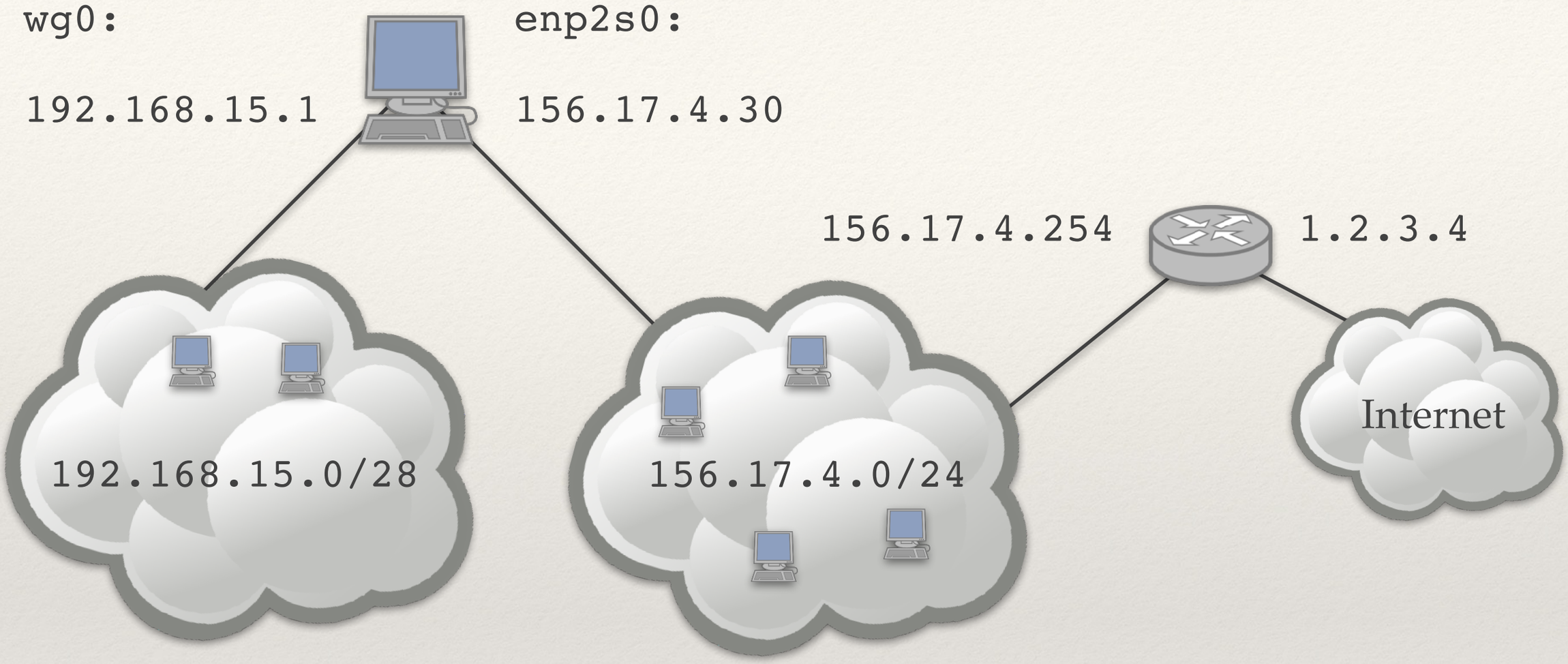


Następny krok

- Akcja z tablicy routingu = wysłanie pakietu do:
  - osiągalnej bezpośrednio przez interfejs **sieci** ***S***
    (jeśli adres docelowy jest w *S*).
  - osiągalnego bezpośrednio przez interfejs **routera** ***X*** (w p.p.).

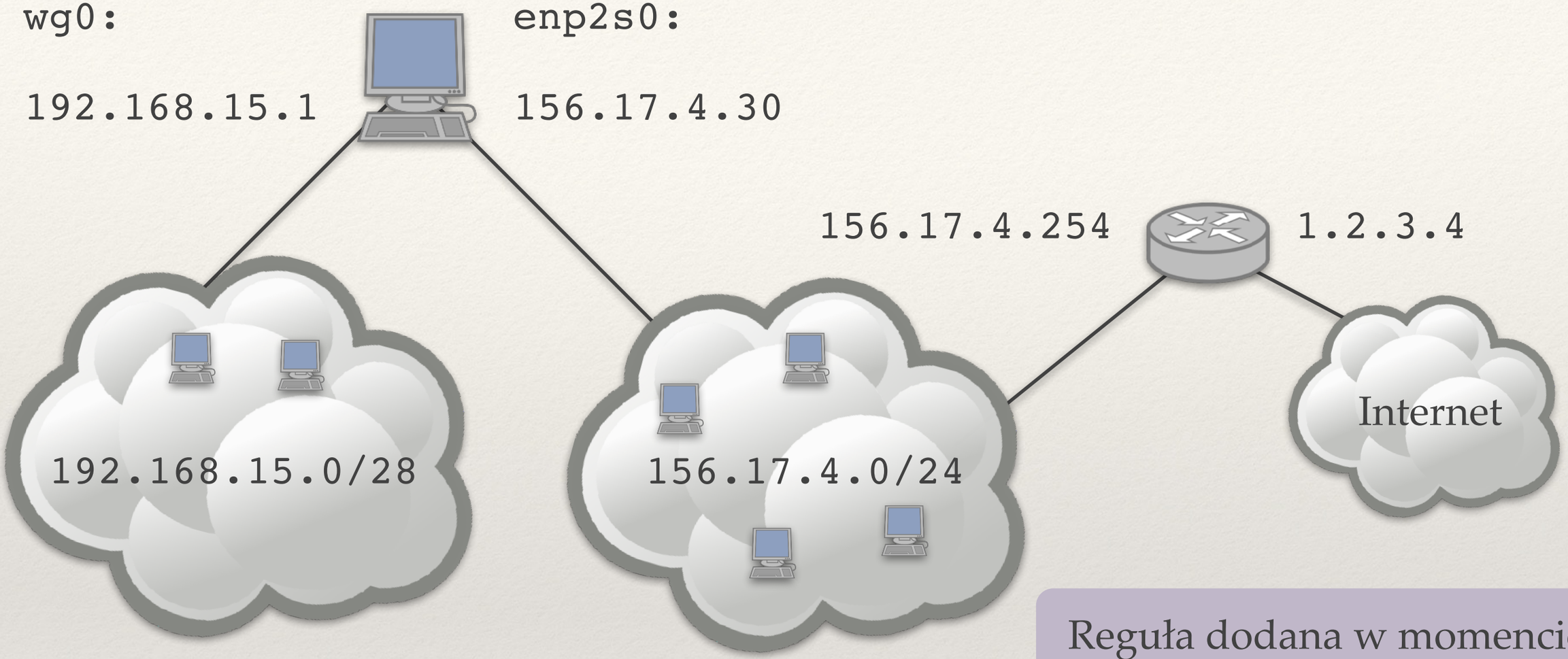
- Bezpośrednio = warstwa sieciowa nie bierze udziału
  w przesyłaniu, choć pakiet może być przesyłany między wieloma urządzeniami.

Przykładowa tablica routingu




```
$> ip route
156.17.4.0/24                          dev enp2s0  src 156.17.4.30
192.168.15.0/28                        dev wg0     src 192.168.15.1
0.0.0.0/0          via 156.17.4.254    dev enp2s0
```

Przykładowa tablica routingu



Reguła dodana w momencie przypisania IP do `enp2s0`

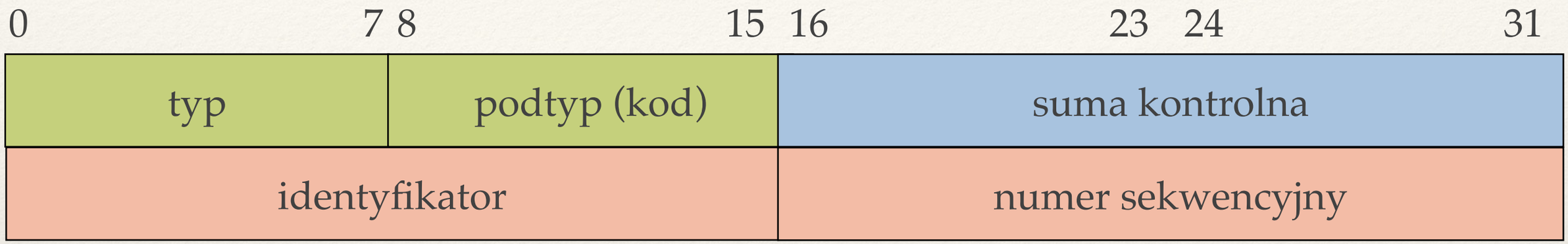
```
$> ip route
156.17.4.0/24                          dev enp2s0  src 156.17.4.30
192.168.15.0/28                        dev wg0     src 192.168.15.1
0.0.0.0/0          via 156.17.4.254    dev enp2s0
```

ICMP

ICMP (*Internet Control Message Protocol*)

- Protokół pomocniczy warstwy sieciowej.
- Pakiety ICMP są enkapsulowane w pakietach IP (są danymi w pakiecie IP).

Nagłówek ICMP



- Bajty 5-8 mogą nie występować.

Ping

- **Wysyła pakiet ICMP o typie 8 (*echo request*).**
  - W danych ICMP jest m.in. znacznik czasowy.
- **Odbiorca odsyła pakiet ICMP o typie 0 (*echo reply*).**
  - W danych ICMP są dokładnie te same pola co w żądaniu.
  - Na tej podstawie można wyznaczyć RTT.

```
PING whitehouse.gov (104.89.18.154): 56 data bytes
from 104.89.18.154: icmp_seq=0 ttl=58 time=65.080 ms
from 104.89.18.154: icmp_seq=1 ttl=58 time=67.033 ms
from 104.89.18.154: icmp_seq=2 ttl=58 time=68.533 ms
```

Traceroute

Jak `traceroute -I` wylicza ścieżkę do *X* (docelowego adresu IP)?

- Niech k = odległość do X.
- `traceroute -I` wysyła pakiety ICMP *echo request* z kolejnymi wartościami TTL (1, 2, 3, …, 30).

Traceroute

Jak `traceroute -I` wylicza ścieżkę do *X* (docelowego adresu IP)?

- Niech k = odległość do *X*.
- `traceroute -I` wysyła pakiety ICMP *echo request* z kolejnymi wartościami TTL (1, 2, 3, …, 30).
- Jeśli wyślemy pakiet z TTL = *j:*
  - Jeśli $j < k$, to pakiet dotrze do *j*-tego routera na trasie do celu. Router wyrzuci pakiet i odeśle nam pakiet ICMP *time exceeded* (typ 11, podtyp 0).

Traceroute

Jak `traceroute -I` wylicza ścieżkę do *X* (docelowego adresu IP)?

- Niech k = odległość do *X*.
- `traceroute -I` wysyła pakiety ICMP *echo request* z kolejnymi wartościami TTL (1, 2, 3, …, 30).
- Jeśli wyślemy pakiet z TTL = *j:*
  - Jeśli $j < k$, to pakiet dotrze do *j*-tego routera na trasie do celu. Router wyrzuci pakiet i odeśle nam pakiet ICMP *time exceeded* (typ 11, podtyp 0).
  - Jeśli $j \geq k$, to komputer docelowy odpowie pakiet ICMP *echo reply*.

Traceroute

- `traceroute` z opcją `-I` wysyła pakiety ICMP *echo request*.
  - Jeśli dotrą do komputera docelowego, to dostaniemy pakiet ICMP *echo reply*.

- `traceroute` bez opcji `-I` wysyła pakiety UDP do rzadko używanego portu.
  - Jeśli dotrą do komputera docelowego, to dostaniemy pakiet ICMP *port unreachable*.

Traceroute: przykład

```
traceroute to example.org (104.18.2.24)
 1  _gateway (172.16.16.254)  0.160 ms
 2  info.wask.wroc.pl (156.17.4.254)  1.003 ms
 3  matchem-vprn509-curie-uni.wask.wroc.pl (156.17.252.26) 0.406
 4  uwrvprn509-unir2.wask.wroc.pl (156.17.252.37)  0.610 ms
 5  unir2-uwrvprn509.wask.wroc.pl (156.17.252.36)  0.620 ms
 6  archi3-uni2.wask.wroc.pl (156.17.254.66)  2.411 ms
 7  156.17.254.116 (156.17.254.116)  1.000 ms
 8  156.17.254.119 (156.17.254.119)  0.800 ms
 9  212.191.238.214 (212.191.238.214)  1.105 ms
10  13335-wa1-ix-02.equinix.com (195.182.218.66)  15.714 ms
11  162.158.100.27 (162.158.100.27)  7.508 ms
12  104.18.2.24 (104.18.2.24)  6.678 ms
```

Traceroute: przykład

```
traceroute to example.org (104.18.2.24)
 1  _gateway (172.16.16.254)  0.160 ms
 2  info.wask.wroc.pl (156.17.4.254)  1.003 ms
 3  matchem-vprn509-curie-uni.wask.wroc.pl (156.17.252.26) 0.406
 4  uwrvprn509-unir2.wask.wroc.pl (156.17.252.37)  0.610 ms
 5  unir2-uwrvprn509.wask.wroc.pl (156.17.252.36)  0.620 ms
 6  archi3-uni2.wask.wroc.pl (156.17.254.66)  2.411 ms
 7  156.17.254.116 (156.17.254.116)  1.000 ms
 8  156.17.254.119 (156.17.254.119)  0.800 ms
 9  212.191.238.214 (212.191.238.214)  1.105 ms
10  13335-wa1-ix-02.equinix.com (195.182.218.66)  15.714 ms
11  162.158.100.27 (162.158.100.27)  7.508 ms
12  104.18.2.24 (104.18.2.24)  6.678 ms
```

demonstracja

Programowanie gniazd (wstęp)

Gniazda

Interfejs programistyczny do nadawania i odbierania pakietów.

- Umożliwiają podawanie **danych** do umieszczenia w datagramach UDP lub segmentach TCP.

dostęp do niektórych pól za pomocą funkcji gniazd

- **Gniazda surowe**: umożliwiają podawanie danych do umieszczenia bezpośrednio w danych pakietu IP.

Kolejność bajtów w liczbach całkowitych

- Liczba całkowita (np `0x4A3B2C1D`) jest przechowywana inaczej na różnych architekturach. Przykładowo:
  - PowerPC: `0x4A`, `0x3B`, `0x2C`, `0x1D` (*big endian*).
  - Intel x86: `0x1D`, `0x2C`, `0x3B`, `0x4A` (*little endian*).

- W nagłówkach pakietów są liczby, protokoły ustalają „sieciową kolejności bajtów“ równą *big endian*.

- Do konwersji służą funkcje `htons`, `htonl`, `ntohs`, `ntohl`.

Sprawdzanie błędów

- Funkcje dotyczące gniazd często zwracają błędy.
- Zwrócona wartość mniejsza od 0 zazwyczaj oznacza błąd.
- Kod błędu: `errno` → jako komunikat: `strerror(errno)`.

Tworzenie gniazda surowego

```
#include <arpa/inet.h>
int sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
```

- Gniazdo surowe otrzymuje **kopię wszystkich pakietów danego protokołu** (w tym przypadku ICMP).
- `sockfd` jest deskryptorem gniazda (takim jak deskryptor pliku czy potoku).

Programowanie gniazd (odbieranie pakietów)

Odbieranie pakietu z gniazda

`recvfrom()` odbiera kolejny pakiet z kolejki związanej z gniazdem.

```
struct sockaddr_in    sender;
socklen_t             sender_len = sizeof(sender);
uint8_t               buffer [IP_MAXPACKET];

ssize_t packet_len = recvfrom (
    sockfd,
    buffer,
    IP_MAXPACKET,
    0,
    (struct sockaddr*)&sender,
    &sender_len
);
```

pakiet jako ciąg bajtów

informacje o nadawcy

Internetowa struktura adresowa

```
struct sockaddr_in {
  sa_family_t      sin_family;
  in_port_t        sin_port;
  struct in_addr   sin_addr;
  …
}
```

zdefiniowana w `netinet/in.h`

Zamiana struktury adresowej na napis z adresem IP:

```
char sender_ip_str[INET_ADDRSTRLEN];
inet_ntop(
  AF_INET, &(sender.sin_addr),
  sender_ip_str, sizeof(sender_ip_str)
);
```

Odczyt nagłówka IP

```
struct ip
{
  unsigned int ip_hl:4;
  …
  u_int32_t saddr;
  u_int32_t daddr;
};
```

zdefiniowany w `netinet/ip.h`

```
struct ip* ip_header    = (struct ip*) buffer;
uint8_t*   icmp_packet = buffer + 4 * ip_header->ip_hl;
```

Odczyt nagłówka ICMP

```
struct icmp
{
  uint8_t icmp_type;
  uint8_t icmp_code;
  …
};
```

zdefiniowany w `netinet/ip_icmp.h`

```
struct ip*   ip_header   = (struct ip*) buffer;
uint8_t*     icmp_packet = buffer + 4 * ip_header->ip_hl;
struct icmp* icmp_header = (struct icmp*) icmp_packet
```

Kod odbierający pakiety

```
int sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);

for (;;) {
  struct sockaddr_in    sender;
  socklen_t             sender_len = sizeof(sender);
  uint8_t               buffer[IP_MAXPACKET];
  ssize_t packet_len = recvfrom(sockfd, buffer, IP_MAXPACKET, 0,
                                (struct sockaddr*)&sender, &sender_len);
  char ip_str[INET_ADDRSTRLEN];
  inet_ntop (AF_INET, &(sender.sin_addr), ip_str, sizeof(ip_str));
  printf ("IP packet with ICMP content from: %s\n", ip_str);
  struct ip*  ip_header = (struct ip*) buffer;
  ssize_t     ip_header_len = 4 * ip_header->ip_hl;
  // IP header = buffer [0, ip_header_len-1]
  // IP data   = buffer [ip_header_len, packet_len-1]
}
```

Kod odbierający pakiety

```
int sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);

for (;;) {
  struct sockaddr_in    sender;
  socklen_t             sender_len = sizeof(sender);
  uint8_t               buffer[IP_MAXPACKET];
  ssize_t packet_len = recvfrom(sockfd, buffer, IP_MAXPACKET, 0,
                                (struct sockaddr*)&sender, &sender_len);
  char ip_str[INET_ADDRSTRLEN];
  inet_ntop (AF_INET, &(sender.sin_addr), ip_str, sizeof(ip_str));
  printf ("IP packet with ICMP content from: %s\n", ip_str);
  struct ip*  ip_header = (struct ip*) buffer;
  ssize_t     ip_header_len = 4 * ip_header->ip_hl;
  // IP header = buffer [0, ip_header_len-1]
  // IP data   = buffer [ip_header_len, packet_len-1]
}
```

icmp_receive.c
kod (z obsługą błędów) na stronie wykładu

demonstracja

Tryb nieblokujący

- **Funkcja `recvfrom()`:**
  - Standardowe wywołanie blokuje, aż w gnieździe będzie pakiet.
  - Zazwyczaj nie chcemy czekać więcej niż x milisekund.

- **Tryb nieblokujący:**
  - Czwarty parametr `recvfrom()` równy `MSG_DONTWAIT`.
  - Jeśli w gnieździe nie ma pakietów, to `recvfrom()` kończy działanie zwracając -1, zaś `errno = EWOULDBLOCK`.

Tryb nieblokujący

- **Funkcja `recvfrom()`:**
  - Standardowe wywołanie blokuje, aż w gnieździe będzie pakiet.
  - Zazwyczaj nie chcemy czekać więcej niż x milisekund.

- **Tryb nieblokujący:**
  - Czwarty parametr `recvfrom()` równy `MSG_DONTWAIT`.
  - Jeśli w gnieździe nie ma pakietów, to `recvfrom()` kończy działanie zwracając -1, zaś `errno = EWOULDBLOCK`.

- **Aktywne czekanie**:
  - Wywołujemy w pętli cały czas `recvfrom(sockfd,_,_,MSG_DONTWAIT,_,_)`.
  - Sprawdzamy, ile czasu upłynęło od ostatniego odczytu.
  - Wada: 100% zużycie procesora.

Funkcja poll()

```
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 != 0`.
  - Pierwsze wywołanie `recvfrom(sockfd, …)` nie zablokuje.
  - W gnieździe `sockfd` może być więcej niż jeden pakiet → można je odczytać w trybie nieblokującym.

Programowanie gniazd (wysyłanie pakietów)

Konstrukcja pakietu ICMP do wysłania

```
struct icmp header;
header.icmp_type = ICMP_ECHO;
header.icmp_code = 0;
header.icmp_hun.ih_idseq.icd_id =
header.icmp_hun.ih_idseq.icd_seq =
header.icmp_cksum = 0;
header.icmp_cksum = compute_icmp_checksum (
        (u_int16_t*)&header, sizeof(header));
```

unikatowy identyfikator,
np. 16 mniej znaczących bitów z PID

numer sekwencyjny

- W przypadku ICMP *echo request* wystarczy sam nagłówek.

`icmp_checksum.c`
kod na stronie wykładu liczący sumę kontrolną

Adresowanie

Wpisujemy adres odbiorcy do struktury adresowej:

```
struct sockaddr_in recipient = {0};
recipient.sin_family = AF_INET;
inet_pton (AF_INET, "adres_ip", &recipient.sin_addr);
```

Opcje gniazda

```
ttl = 42;
setsockopt (sockfd, IPPROTO_IP, IP_TTL, &ttl, sizeof(int));
```

- Brak bezpośredniego dostępu do nagłówka IP.
- Ustawienie TTL wymaga `setsockopt()`.

Wysyłanie pakietu przez gniazdo

```
ssize_t bytes_sent = sendto (
  sockfd,
  &header,
  sizeof(header),
  0,
  (struct sockaddr*)&recipient,
  sizeof(recipient)
);
```

Lektura dodatkowa

- Kurose & Ross: rozdział 4.
- Tanenbaum: rozdział 5.
- Stevens: rozdział 25.

- Dokumentacja IP i ICMP:
  - https://web.archive.org/web/20220412004810/http://www.networksorcery.com/enp/protocol/ip.htm
  - https://web.archive.org/web/20220410210745/http://www.networksorcery.com/enp/protocol/icmp.htm

Zagadnienia

- Z czego wynika hierarchia adresów IP? Jaki ma wpływ na konstrukcję tablic routingu?
- Jakie są zasady notacji CIDR?
- Co to jest adres rozgłoszeniowy?
- Co to jest maska (pod)sieci?
- Opisz sieci IP klasy A, B i C.
- Co to jest interfejs pętli zwrotnej (*loopback*)?
- Do czego służy pole TTL w pakiecie IP? Do czego służy pole protokół?
- Jakie reguły zawierają tablice routingu?
- Na czym polega reguła najdłuższego pasującego prefiksu?
- Co to jest trasa domyślna?
- Do czego służy protokół ICMP? Jakie znasz typy pakietów ICMP?
- Jak działa polecenie `ping`?
- Jak działa polecenie `traceroute`?
- Dlaczego do tworzenia gniazd surowych wymagane są uprawnienia administratora?
- Co to jest sieciowa kolejność bajtów?
- Co robią funkcje `socket()`, `recvfrom()` i `sendto()`?
- Jakie informacje zawiera struktura adresowa `sockaddr_in`?
- Co to jest tryb blokujący i nieblokujący? Co to jest aktywne czekanie?
- Jakie jest działanie funkcji `poll()`?