

---

# Routing

## część 1: adresowanie

Sieci komputerowe

Wykład 2

---

*Marcin Bieńkowski*

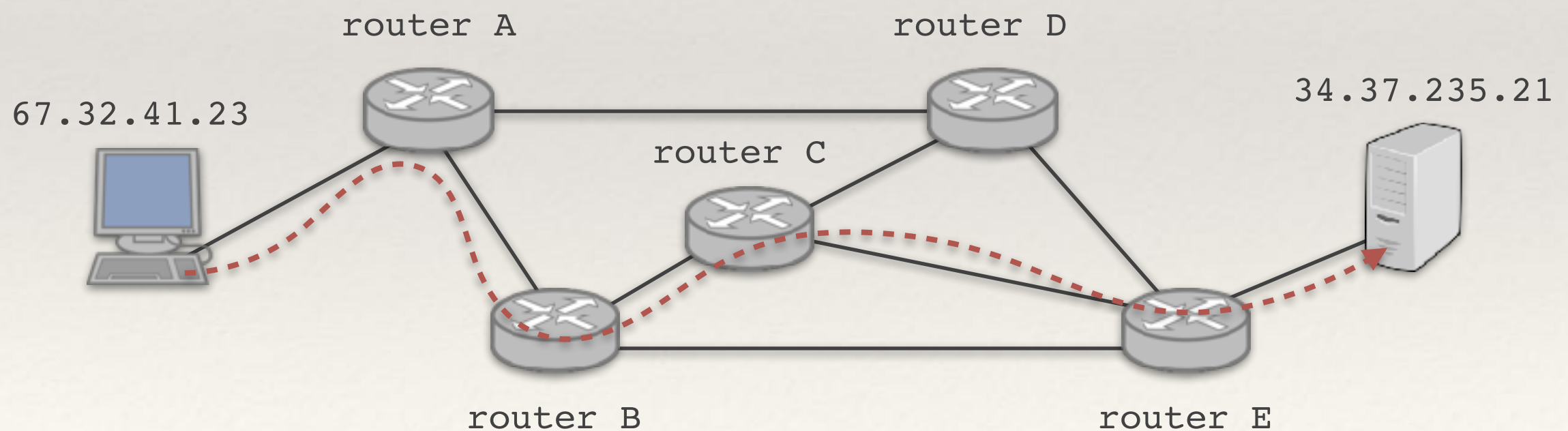
---

W poprzednim odcinku

---

# Jak przesyłać dane przez sieć

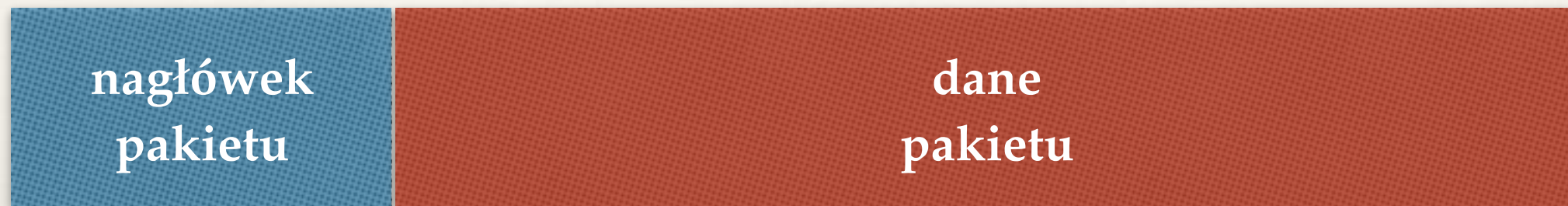
- ❖ Chcemy przesyłać między aplikacjami strumień danych.
- ❖ **Globalne adresowanie:** w Internecie każda karta sieciowa ma unikatowy 4-bajtowy adres IP.
- ❖ Warstwa sieciowa zapewnia globalne dostarczanie danych pomiędzy dwoma dowolnymi komputerami (kartami sieciowymi = interfejsami sieciowymi).



# Przełączanie pakietów

---

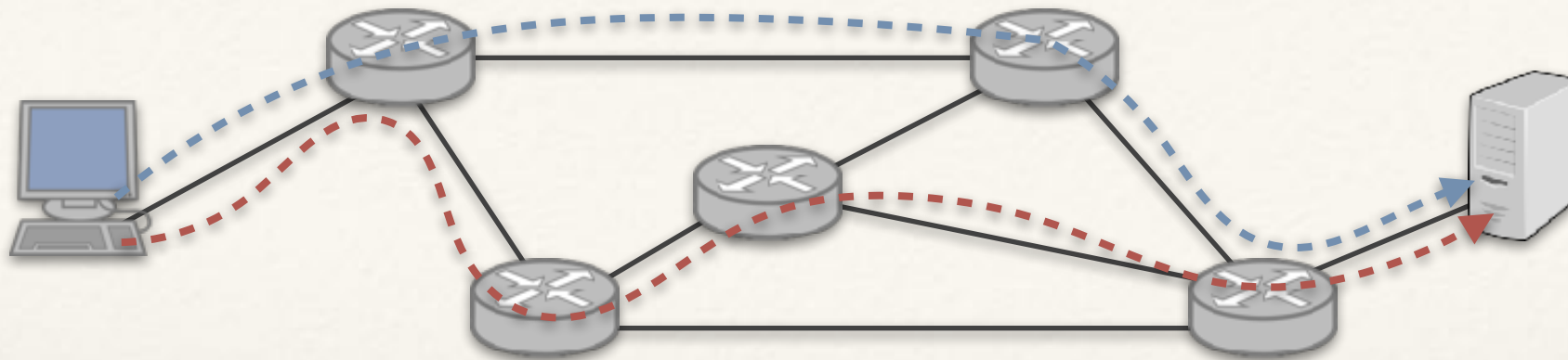
- ❖ Wysyłany strumień danych dzielimy na małe porcje: **pakiety**.



- ❖ Dane pakietu = fragment strumienia danych.
- ❖ Nagłówek pakietu = informacje kontrolne, m.in. adres źródłowy i docelowy.
- ❖ Każdy pakiet przesyłany niezależnie.

# Routing

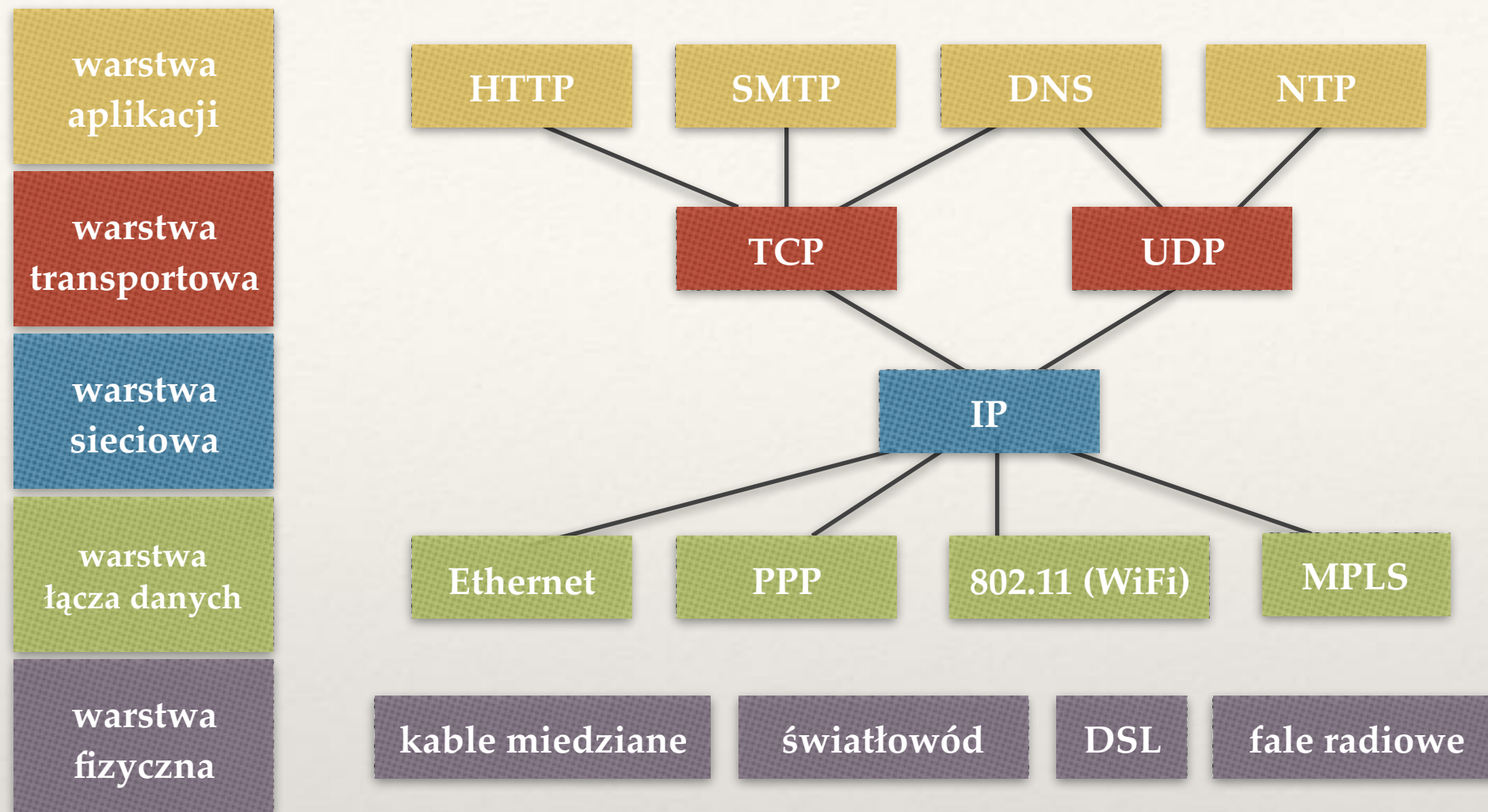
---



- ❖ Routing (trasowanie) = wybór trasy dla danego pakietu.
- ❖ W Internecie:
  - ❖ 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**.



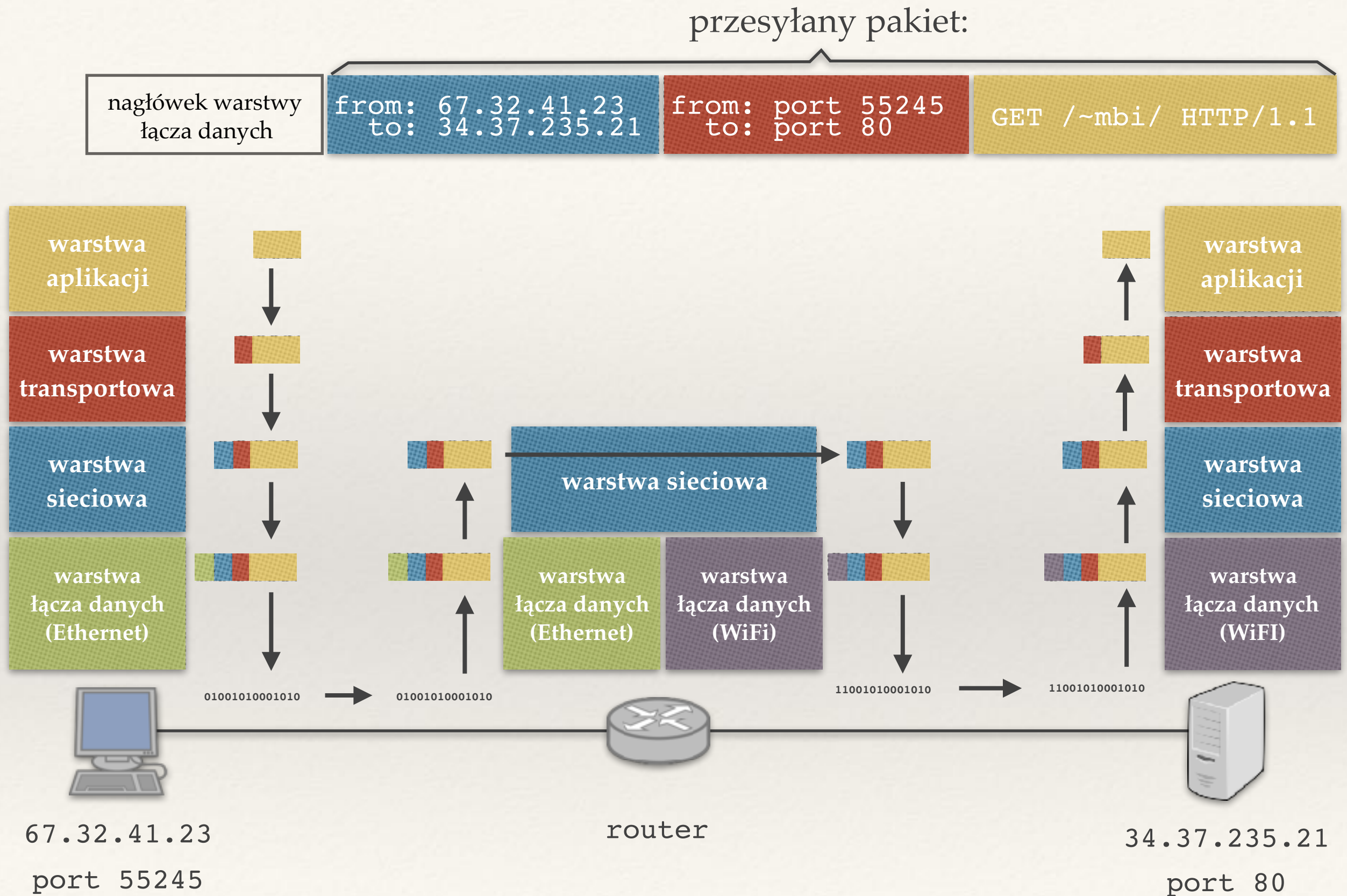
# Protokoły w Internecie



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

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

# Internetowy model warstwowy



---

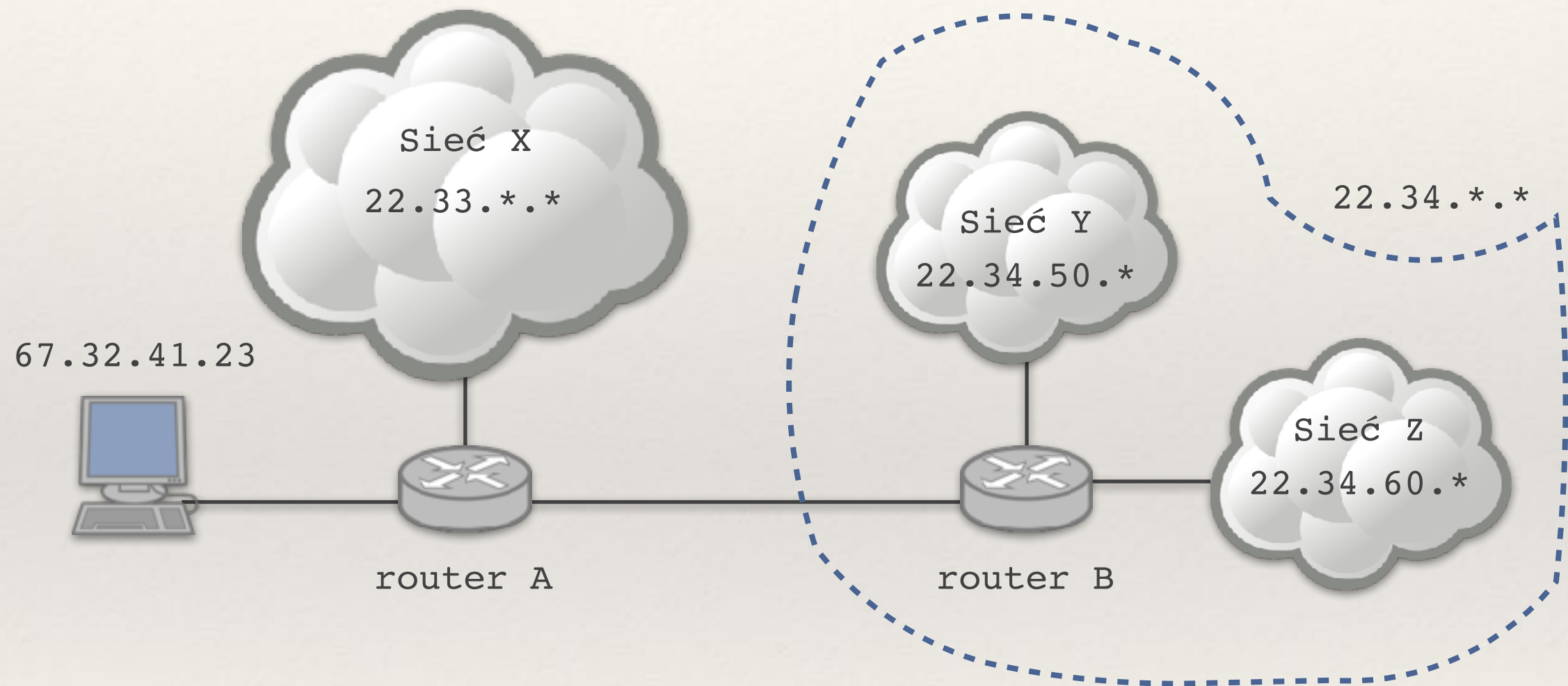
# Adresowanie

---



# Adresy IP

- ❖ Każda karta sieciowa ma unikatowy 4-bajtowy adres.
- ❖ Hierarchiczna struktura adresów:



- ❖ 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 (1)

---

- ❖ Notacja **CIDR** (Classless Inter-Domain Routing): opisuje zakres adresów IP posiadających wspólny prefiks za pomocą pary (pierwszy adres z zakresu, długość prefiksu).
- ❖ **Przykład:** 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 (2)

---

**Przykład 1:**  $156.17.4.32/28$  = adresy zaczynające się od prefiksu  $156.17.4.0010$ :

- ❖  $156.17.4.00100000$  =  $156.17.4.32$  (pierwszy adres)
- ❖  $156.17.4.00100001$  =  $156.17.4.33$  (drugi adres)
- ❖ ...
- ❖  $156.17.4.00101110$  =  $156.17.4.46$  (przedostatni adres)
- ❖  $156.17.4.00101111$  =  $156.17.4.47$  (ostatni adres)

Razem:  $2^{32-28} = 2^4 = 16$  adresów.

**Przykład 2:**  $0.0.0.0/0$  = wszystkie adresy IP

**Przykład 3:**  $34.56.78.90/32$  = jeden konkretny adres IP

# CIDR a sieci

---

Notację CIDR najczęściej stosujemy, żeby opisać konkretną sieć.

Dla sieci 156.17.4.32/28:

- ❖ Ostatni adres jest zarezerwowany: **adres rozgłoszeniowy (broadcast)**.
  - ✦ Pakiet wysłany na adres rozgłoszeniowy dotrze do wszystkich adresów IP z zakresu.
- ❖ Pierwszy adres jest zarezerwowany: tzw. **adres sieci**.
  - ✦ Względy historyczne (to był początkowo adres rozgłoszeniowy).
- ❖ Pozostałe  $16 - 2 = 14$  adresów IP może być przypisane do komputerów (kart sieciowych) w tej sieci.



# Podsieci

---

$156.17.4.32/28$  = zbiór następujących adresów

- ❖  $156.17.4.0010\mathbf{0}000 = 156.17.4.32$
  - ❖ ...
  - ❖  $156.17.4.0010\mathbf{0}111 = 156.17.4.39$
- $156.17.4.32/29$
- 
- ❖  $156.17.4.0010\mathbf{1}000 = 156.17.4.40$
  - ❖ ...
  - ❖  $156.17.4.0010\mathbf{1}111 = 156.17.4.47$
- $156.17.4.40/29$

# Adres IP to za mało!

---

- ❖ Czy adres 156.17.4.95 jest adresem rozgłoszeniowym?

# Adres IP to za mało!

---

- ❖ Czy adres 156.17.4.95 jest adresem rozgłoszeniowym?
  - ❖ Tak w sieci 156.17.4.80/28 = {156.17.4.80, ..., 156.17.4.95}
  - ❖ Tak w sieci 156.17.4.64/27 = {156.17.4.64, ..., 156.17.4.95}
  - ❖ Nie w sieci 156.17.4.64/26 = {156.17.4.64, ..., 156.17.4.127}

# Adres IP to za mało!

---

- ❖ Czy adres  $156.17.4.95$  jest adresem rozgłoszeniowym?
  - ❖ Tak w sieci  $156.17.4.80/28 = \{156.17.4.80, \dots, 156.17.4.95\}$
  - ❖ Tak w sieci  $156.17.4.64/27 = \{156.17.4.64, \dots, 156.17.4.95\}$
  - ❖ Nie w sieci  $156.17.4.64/26 = \{156.17.4.64, \dots, 156.17.4.127\}$
- ❖ Przy podawaniu dowolnego adresu IP powinniśmy też podać długość prefiksu określającego jego sieć, np.  $156.17.4.95/26$ .



# CIDR (3)

---

- ❖ Notację CIDR rozszerzamy na wszystkie adresy IP. Przykładowo:
  - ♦  $156.17.4.32/28$  = cały zakres 16 adresów.
  - ♦  $156.17.4.33/28$  = pierwszy adres dla komp. w sieci  $156.17.4.32/28$ .
  - ♦  $156.17.4.34/28$  = drugi adres dla komputera w sieci  $156.17.4.32/28$ .
  - ♦ ...
  - ♦  $156.17.4.46/28$  = ostatni adres dla komputera w sieci  $156.17.4.32/28$ .
  - ♦  $156.17.4.47/28$  = adres rozgłoszeniowy w sieci  $156.17.4.32/28$ .
  - ♦ Przy tej konwencji rezerwowanie pierwszego adresu na adres sieci znowu ma sens.

# CIDR (3)

---

- ❖ Notację CIDR rozszerzamy na wszystkie adresy IP. Przykładowo:
  - ♦  $156.17.4.32/28$  = cały zakres 16 adresów.
  - ♦  $156.17.4.33/28$  = pierwszy adres dla komp. w sieci  $156.17.4.32/28$ .
  - ♦  $156.17.4.34/28$  = drugi adres dla komputera w sieci  $156.17.4.32/28$ .
  - ♦ ...
  - ♦  $156.17.4.46/28$  = ostatni adres dla komputera w sieci  $156.17.4.32/28$ .
  - ♦  $156.17.4.47/28$  = adres rozgłoszeniowy w sieci  $156.17.4.32/28$ .
  - ♦ Przy tej konwencji rezerwowanie pierwszego adresu na adres sieci znowu ma sens.
- ❖ Długość prefiksu nazywamy maską (pod)sieci.
  - ♦ Czasem zapisywany w postaci bitowej  $/28 = /255.255.255.240$  (28 jedynek).
  - ♦ Aby uzyskać adres sieci robimy logiczny AND adresu IP z maską sieci.

# Klasy adresów

---

- ❖ Jeśli nie podamy maski sieci, niektóre polecenia (np. `ifconfig`) wydedukują ją z adresu IP.
  - ♦ `ifconfig eth0 10.0.0.1`  
= `ifconfig eth0 10.0.0.1 netmask 255.0.0.0`
- ❖ Względy historyczne (klasy adresów IP).
  - ♦ Adres IP zaczyna się od 0 → maska podsieci = /8 (klasa A).
  - ♦ Adres IP zaczyna się od 10 → maska podsieci = /16 (klasa B).
  - ♦ Adres IP zaczyna się od 110 → maska podsieci = /24 (klasa C).

# Pętla lokalna = 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

---

```
eth0:    <BROADCAST,MULTICAST> mtu 1500 state UP
          link/ether d8:cb:8a:34:a4:66 brd ff:ff:ff:ff:ff:ff
          inet 156.17.4.30/24 brd 156.17.4.255 scope global eth0

lo:      <LOOPBACK> mtu 65536 state UNKNOWN
          link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
          inet 127.0.0.1/8 scope host lo

tun0     <POINTOPOINT,MULTICAST,NOARP> mtu 1500 state UNKNOWN
          link/none
          inet 172.28.0.1/16 brd 172.28.255.255 scope global tun0
```

---

# Routing pakietów IP

---

# Nagłówek pakietu IP

0	7	8	15	16	23	24	31
wersja	IHL	typ usługi	całkowita długość pakietu				
pola związane z fragmentacją pakietu							
TTL		protokół	suma kontrolna nagłówka IP				
źródłowy adres IP							
docelowy adres IP							

- ❖  $4 \times \text{IHL} = \text{długość nagłówka w bajtach}$ .
- ❖ Protokół = datagram jakiego protokołu przechowywany jest w danych pakietu (np. 1 = ICMP, 6 = TCP, 17 = UDP).
- ❖ TTL = czas życia pakietu.

# Przetwarzanie pakietu w routerze

---

## 1. Obliczenie portu wyjściowego:

- ♦ na podstawie adresu docelowego pakietu;
- ♦ wykorzystując tablicę routingu.

## 2. Aktualizacja nagłówka:

- ♦ zmniejszenie TTL o 1; jeśli  $TTL = 0$ , to pakiet wyrzucany;
- ♦ ponowne wyliczenie sumy kontrolnej pakietu.

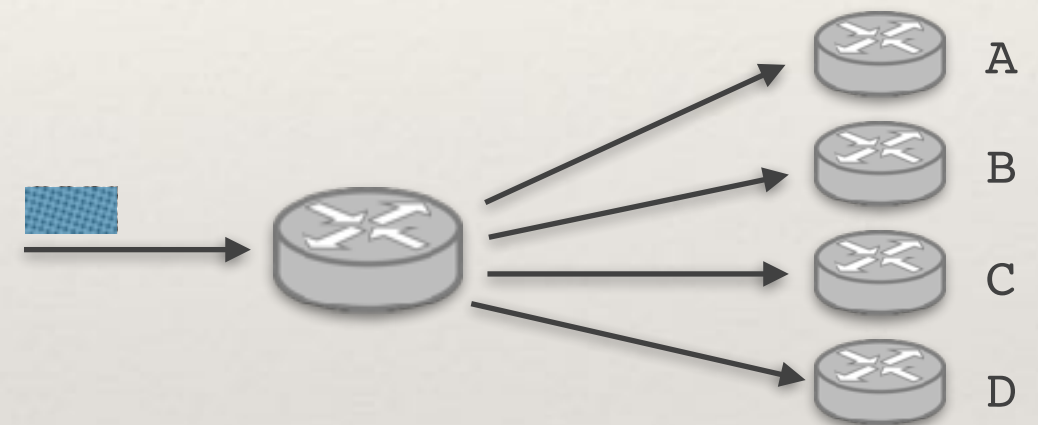
## 3. Przekazanie pakietu do kolejki wyjściowej.



# Tablice routingu

- ❖ Tablica routingu zawiera reguły typu „jeśli adres docelowy pakietu zaczyna się od prefiksu  $A$ , to wyślij pakiet do  $X$ ”.

prefiks CIDR	akcja
10.20.30.0/24	do routera A
8.0.0.0/8	do routera B
9.9.9.0/24	do routera C
156.17.0.0/16	do routera C
156.18.0.0/16	do routera D

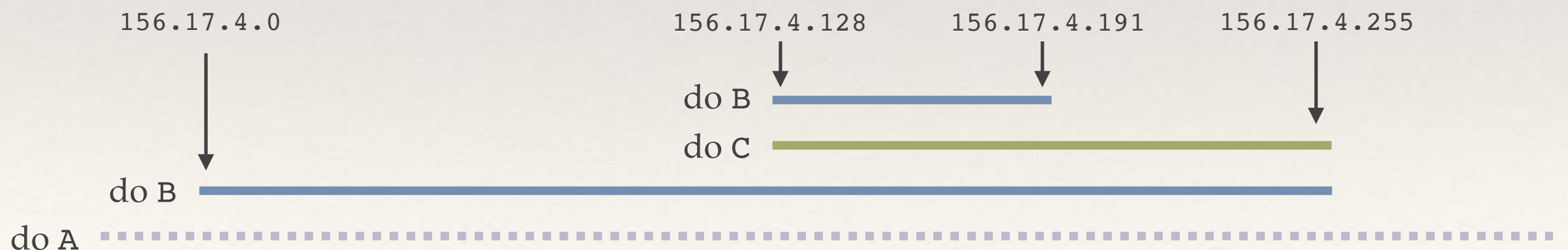
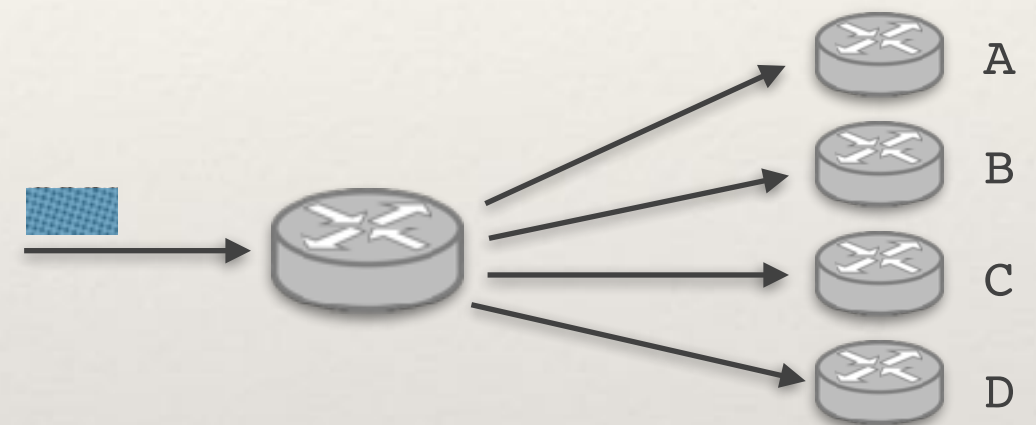


- ❖ Pakiet niepasujący do żadnej reguły jest odrzucany.

# Reguła najdłuższego pasującego prefiksu

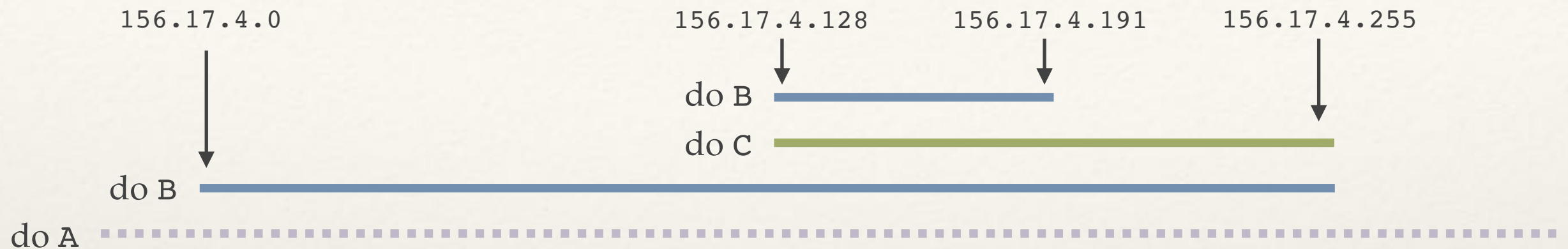
- ❖ Jeśli więcej niż jedna reguła pasuje, wybierana jest ta, która jest **najdłuższym prefiksem** (najbardziej „konkretna reguła”).
- ❖  $0.0.0.0/0$  = reguła (trasa) domyślna.

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



# Równoważne tablice routingu

---



||



# Następny krok

---

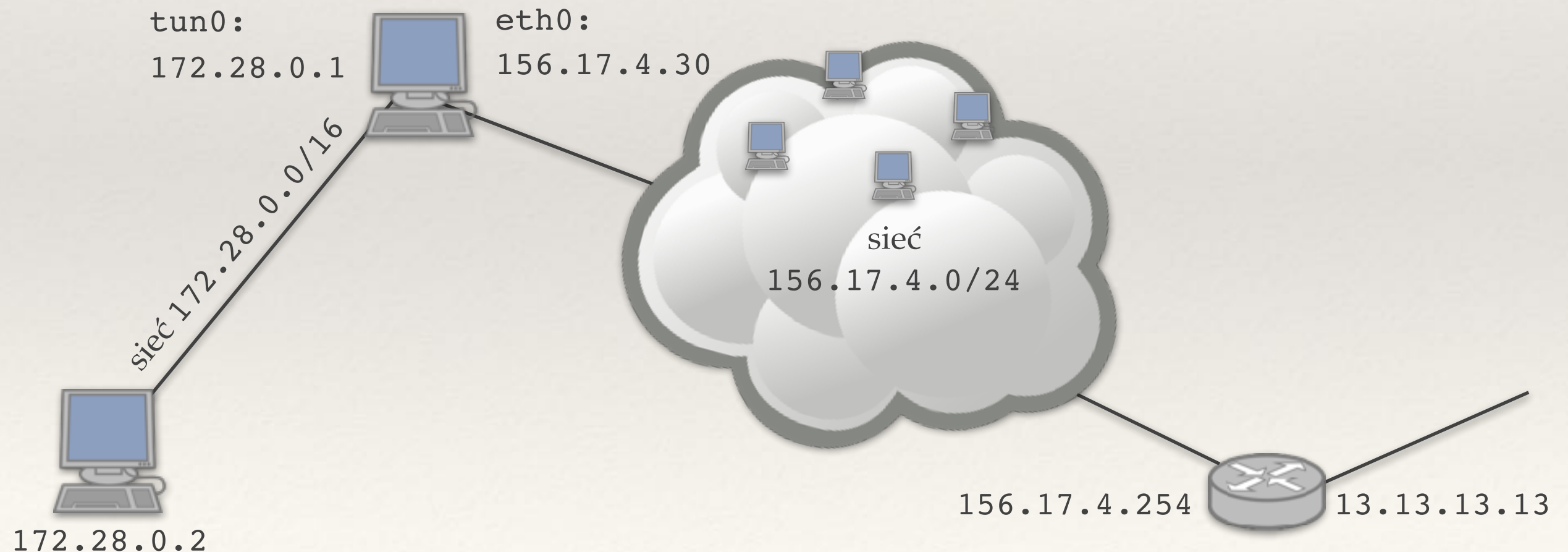
- ❖ **Akcja z tablicy routingu** = wysłanie pakietu do
  - ♦ osiągalnego bezpośrednio przez interfejs sieciowy routera X ...
  - ♦ ... albo do osiągalnej bezpośrednio przez interfejs sieci S.
- ❖ **Bezpośrednio** = warstwa sieciowa nie bierze udziału w przesyłaniu, choć pakiet może być przesyłany między wieloma urządzeniami.



# Przykładowa tablica routingu w komputerze

```
$> ip route
```

```
156.17.4.0/24          dev eth0  src 156.17.4.30
172.28.0.0/16          dev tun0  src 172.28.0.1
0.0.0.0/0              via 156.17.4.254 dev eth0
```



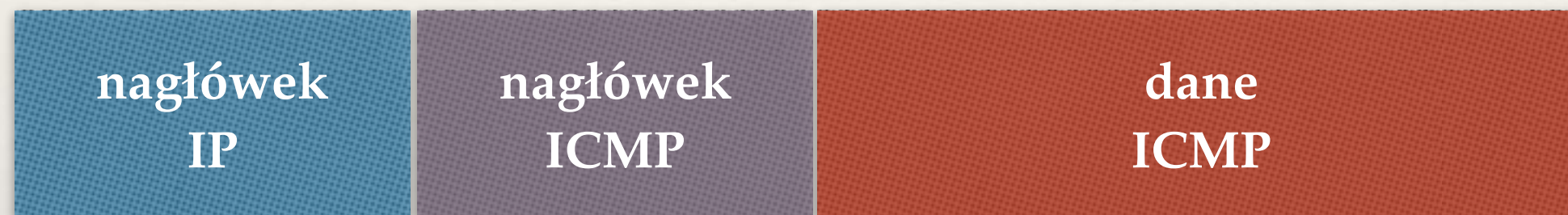
---

# ICMP

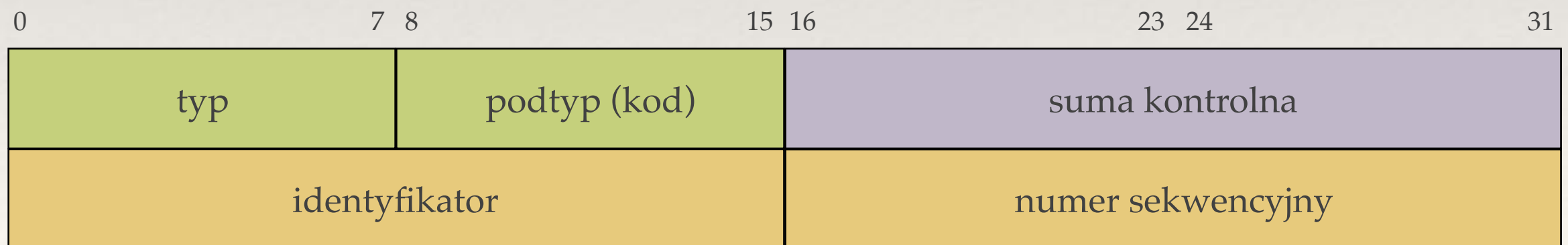
---

# ICMP (Internet Control Message Protocol)

- ❖ Protokół pomocniczy warstwy trzeciej.
- ❖ Pakiety ICMP są enkapsulowane w pakietach IP (stanowią pole danych w pakiecie IP).



- ❖ Nagłówek ICMP (bajty 5-8 mogą nie występować):



# Ping

---

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

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



# Traceroute (1)

---

- ❖ Pakiet IP z startowym  $TTL = j$ , zostaje odrzucony przez  $j$ -ty router na trasie do celu. Router ten odeśle komunikat ICMP *time exceeded* (typ 11, podtyp 0).
- ❖ `traceroute` wyświetla ścieżkę do docelowego adresu IP:
  - ♦ Pakiety mają coraz większe TTL.
  - ♦ Wysyłamy ICMP echo request → komputer docelowy odpowie komunikatem ICMP *echo reply*.
  - ♦ Albo pakiet UDP do rzadko używanego portu → komputer docelowy odpowie komunikatem ICMP *port unreachable*.

# Traceroute (2)

---

traceroute to www.ii.uni.wroc.pl (156.17.4.3), 64 hops max, 52 byte packets

```
1  livebox (192.168.1.1)  3.151 ms
2  wro-bng2.tpnet.pl (80.50.18.74)  31.965 ms
3  wro-r2.tpnet.pl (80.50.122.73)  31.870 ms
4  lodz-ar3.tpnet.pl (213.25.5.206)  62.835 ms
5  z-tpnetu.lodz-gw.rtr.pionier.gov.pl (80.50.231.26)  37.103 ms
6  lodz-gw2.z-lodz-gw.rtr.pionier.gov.pl (212.191.126.77)  37.606 ms
7  z-lodz-gw.wroclaw.10gb.rtr.pionier.gov.pl (212.191.225.34)  44.687 ms
8  rolnik-karkonosz.wask.wroc.pl (156.17.254.112)  46.707 ms
9  archi-rolnik.wask.wroc.pl (156.17.254.108)  47.936 ms
10 matchem-archi.wask.wroc.pl (156.17.254.142)  47.986 ms
11 gwuwrmatchem.uni.wroc.pl (156.17.252.33)  49.342 ms
12 www.ii.uni.wroc.pl (156.17.4.3)  48.511 ms
```

---

# Programowanie gniazd (wstęp)

---

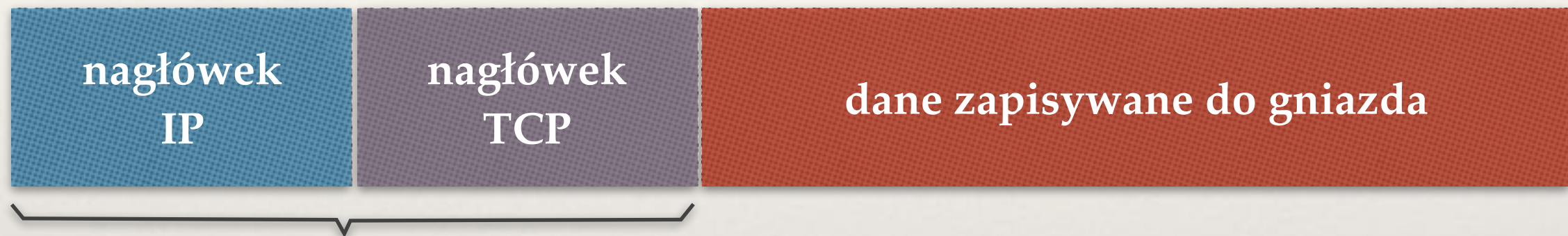


# Gniazda

---

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 wymagają „sieciowej kolejności bajtów” (*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 podobnym do deskryptora 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);  
u_int8_t           buffer [IP_MAXPACKET];
```

```
ssize_t packet_len = recvfrom (  
    sockfd,  
    buffer, ← pakiet jako ciąg bajtów  
    IP_MAXPACKET,  
    0,  
    (struct sockaddr*) &sender, } ← informacje o nadawcy  
    &sender_len  
);
```



# Internetowa struktura adresowa

---

```
struct sockaddr_in {  
    sa_family_t    sin_family;  
    in_port_t      sin_port;  
    struct in_addr  sin_addr;  
    // tutaj zera  
}
```

zdefiniowana w `netinet/in.h`

```
char sender_ip_str[20];  
inet_ntop (  
    AF_INET,  
    &(sender.sin_addr),  
    sender_ip_str,  
    sizeof(sender_ip_str)  
);
```

zamienia strukturę adresową  
w sender na napis z adresem IP

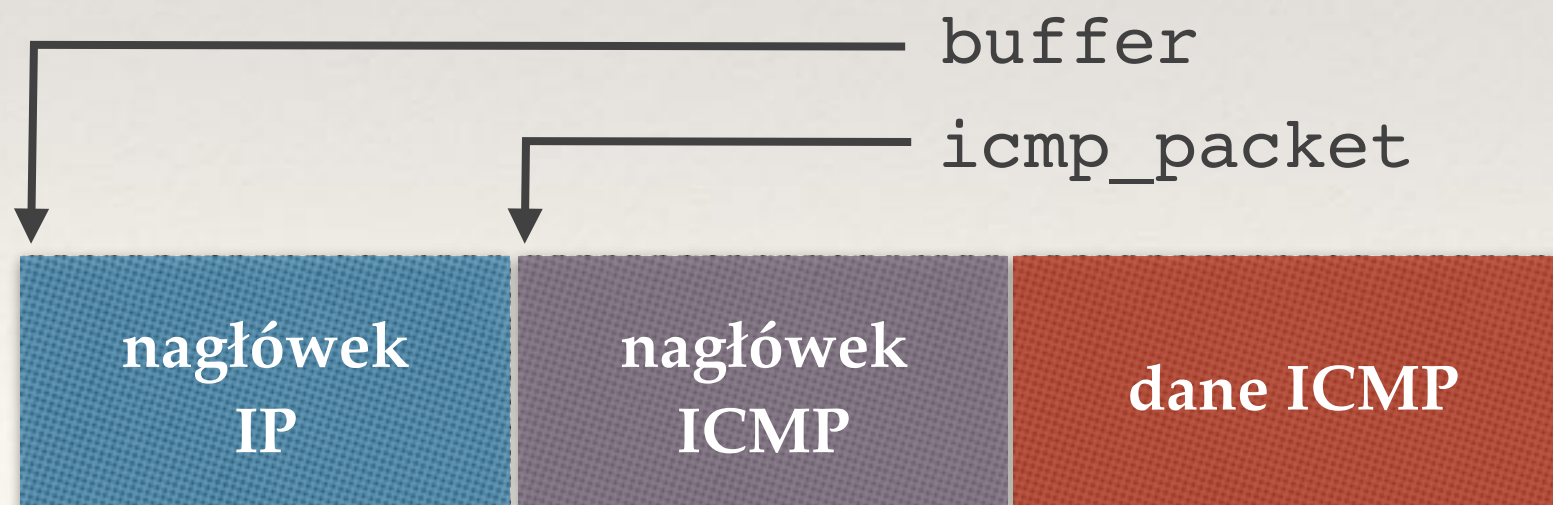
# Odczyt nagłówka IP

```
struct iphdr
```

```
{  
    unsigned int version:4;  
    unsigned int ihl:4;  
    ...  
    u_int32_t saddr;  
    u_int32_t daddr;  
};
```

zdefiniowana w `netinet/ip.h`

```
struct iphdr* ip_header = (struct iphdr*) buffer;  
u_int8_t* icmp_packet = buffer + 4 * ip_header->ihl;
```



# Odczyt nagłówka ICMP

```
struct icmphdr
```

zdefiniowana w `netinet/ip_icmp.h`

```
{  
    u_int8_t type;  
    u_int8_t code;  
    ...  
};
```

```
struct iphdr*    ip_header    = (struct iphdr*) buffer;  
u_int8_t*        icmp_packet  = buffer + 4 * ip_header->ihl;  
struct icmphdr* icmp_header  = (struct icmphdr*) 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);
    u_int8_t buffer[IP_MAXPACKET];

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

    char ip_str[20];          // sender IP
    inet_ntop (AF_INET, &(sender.sin_addr), ip_str, sizeof(ip_str));
    printf ("IP packet with ICMP content from: %s\n", ip_str);

    struct iphdr* ip_header = (struct iphdr*) buffer;
    ssize_t ip_header_len = 4 * ip_header->ihl;

    // IP header = buffer [0, ip_header_len-1]
    // IP data    = buffer [ip_header_len, packet_len-1]
}
```

Brak obsługi błędów!

demonstracja

[cały kod programu](#)



# Tryb nieblokujący

---

## ❖ Funkcja `recvfrom()`:

- ♦ Standardowe wywołanie blokuje aż w gnieździe będzie pakiet.
- ♦ Zazwyczaj nie chcemy czekać więcej niż  $x$  sekund.

## ❖ 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  $x$  sekund.

## ❖ 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 select()

Czekanie maksymalnie x sekund na pakiet w gnieździe sockfd.

```
fd_set descriptors;  
FD_ZERO (&descriptors);  
FD_SET (sockfd, &descriptors);  
struct timeval tv; tv.tv_sec = x; tv.tv_usec = 0;  
int ready = select (sockfd+1, &descriptors, NULL, NULL, &tv);
```

- ❖ `ready < 0` → wystąpił błąd.
- ❖ `ready = 0` → nastąpił timeout (po x sekundach).
- ❖ `ready > 0` → ready obserwowanych deskryptorów gotowych do odczytu.
  - ❖ Obserwowaliśmy tylko jeden, więc `ready = 1`.
  - ❖ 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)

---

# Tworzenie danych do wysyłki

Konstruujemy komunikat ICMP do wysłania. W przypadku ICMP *echo request* wystarczy sam nagłówek.

```
struct icmphdr icmp_header;  
icmp_header.type = ICMP_ECHO;  
icmp_header.code = 0;  
icmp_header.un.echo.id = ???;  
icmp_header.un.echo.sequence = ???;  
icmp_header.checksum = 0;  
icmp_header.checksum = compute_icmp_checksum (  
    (u_int16_t*)&icmp_header, sizeof(icmp_header));
```

unikatowy identyfikator, np. PID

numer sekwencyjny

funkcja obliczająca 16-bitową sumę kontrolną w kodzie uzupełnień do jedności

# Adresowanie

---

Wpisujemy adres odbiorcy do struktury adresowej:

```
struct sockaddr_in recipient;  
bzero (&recipient, sizeof(recipient));  
recipient.sin_family = AF_INET;  
inet_pton(AF_INET, "adres_ip", &recipient.sin_addr);
```



# Opcje gniazda

---

Pole TTL jest w nagłówku IP → brak bezpośredniego dostępu. Zmiana wywołaniem:

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

# Wysyłanie pakietu przez gniazdo

---

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

# Lektura dodatkowa

---

- ❖ Kurose & Ross: rozdział 4.
- ❖ Tanenbaum: rozdział 5.
- ❖ Stevens: rozdział 25.
- ❖ Dokumentacja IP i ICMP:
  - ♦ <http://www.networksorcery.com/enp/protocol/ip.htm>
  - ♦ <http://www.networksorcery.com/enp/protocol/icmp.htm>