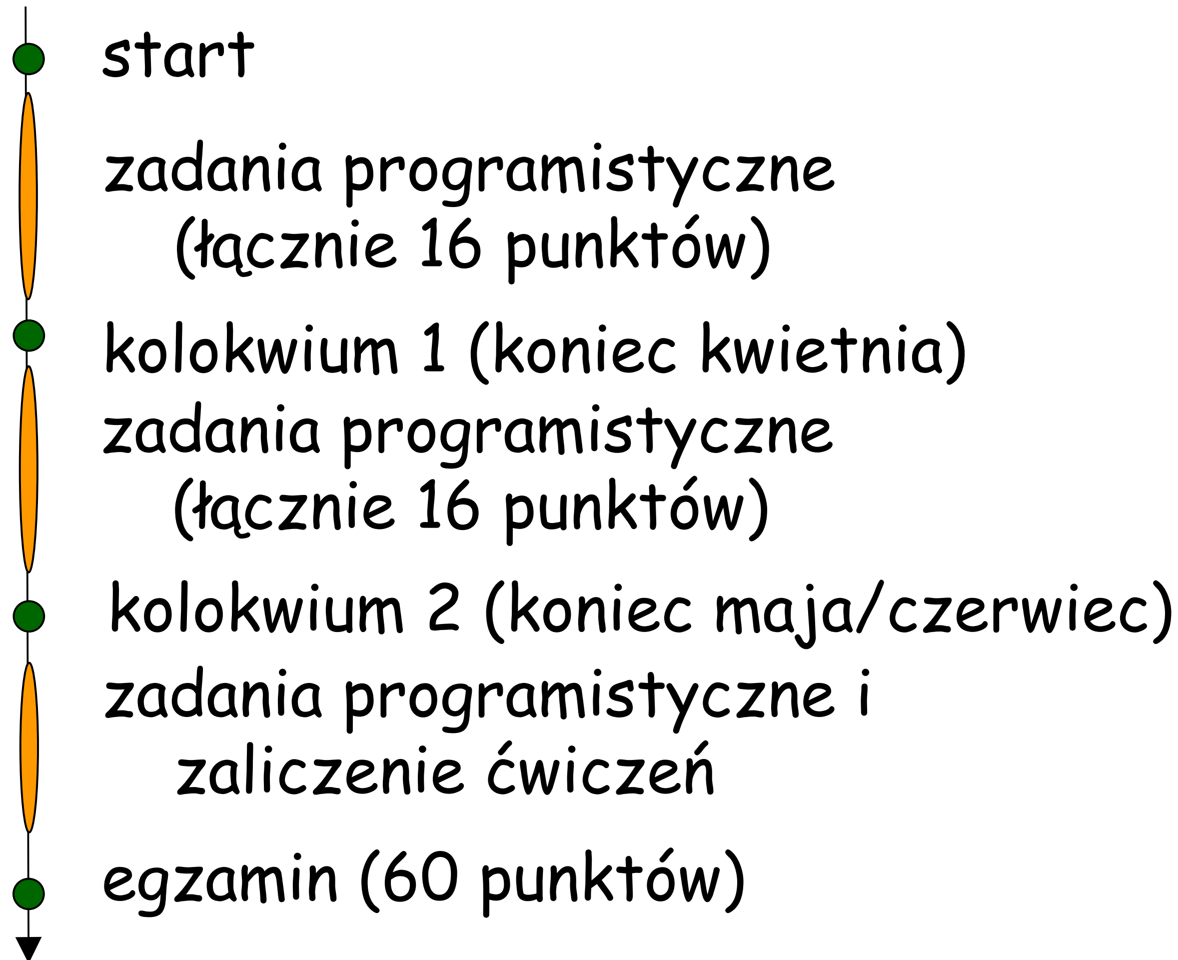
Plan całości wykładu

- Wprowadzenie (2 wykłady)
- Warstwa aplikacji (2 wykłady)
- Warstwa transportu (2-3 wykłady)
- Warstwa sieci (2-3 wykłady)
- Warstwa łącza i sieci lokalne (3 wykłady)
- jeśli zostanie czasu...
  - sieci radiowe
  - komunikacja audio/wideo
  - zarządzanie sieciami

Plan czasowy wykładu i ćwiczeń



Literatura do warstwy transportu

Rozdział 3, *Computer Networking: A Top-Down Approach Featuring the Internet*, wydanie 2 lub 3, J. Kurose, K. Ross, Addison-Wesley, 2004

Rozdziały 3.5, 6.2, 8.3, *Sieci komputerowe – podejście systemowe*, L. Peterson, B. Davie, Wyd. Nakom, Poznań, 2000

Rozdziały 17, 18, 20, 21, *Biblia TCP/IP, tom 1*, R. Stevens, Wyd. RM, Warszawa, 1998

Warstwa transportu

Cele:

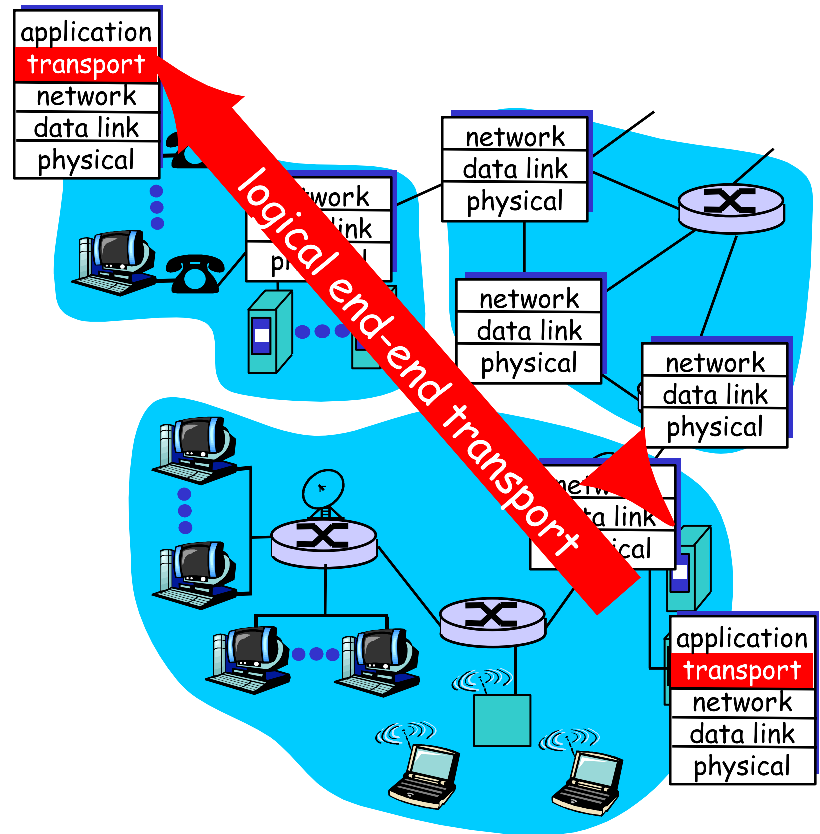
- zrozumienie podstawowych mechanizmów transportowych:
  - multipleksacja/demultipleksacja
  - niezawodna komunikacja
  - kontrola przepływu
  - kontrola przeciążenia
- poznanie mechanizmów transportowych Internetu
  - UDP: transport bezpołączeniowy
  - TCP: transport połączeniowy
  - kontrola przeciążenia TCP

Mapa wykładu

- Usługi warstwy transportu
- Multipleksacja i demultipleksacja
- Transport bezpołączeniowy: UDP
- Zasady niezawodnej komunikacji danych
- Transport połączeniowy: TCP
  - struktura segmentu
  - niezawodna komunikacja
  - kontrola przepływu
  - zarządzanie połączeniem
- Mechanizmy kontroli przeciążenia
- Kontrola przeciążenia w TCP

Usługi i protokoły warstwy transportu

- *logiczna komunikacja* pomiędzy procesami aplikacji działającymi na różnych hostach
- protokoły transportowe działają na systemach końcowych
  - nadawca: dzieli komunikat aplikacji na segmenty, przekazuje segmenty do warstwy sieci
  - odbiorca: łączy segmenty w komunikat, który przekazuje do warstwy aplikacji
- więcej niż jeden protokół transportowy
  - Internet: TCP oraz UDP



Warstwy transportu i sieci

- *warstwa sieci:* logiczna komunikacja pomiędzy hostami
- *warstwa transportu:* logiczna komunikacja pomiędzy procesami
  - korzysta z oraz uzupełnia usługi warstwy sieci

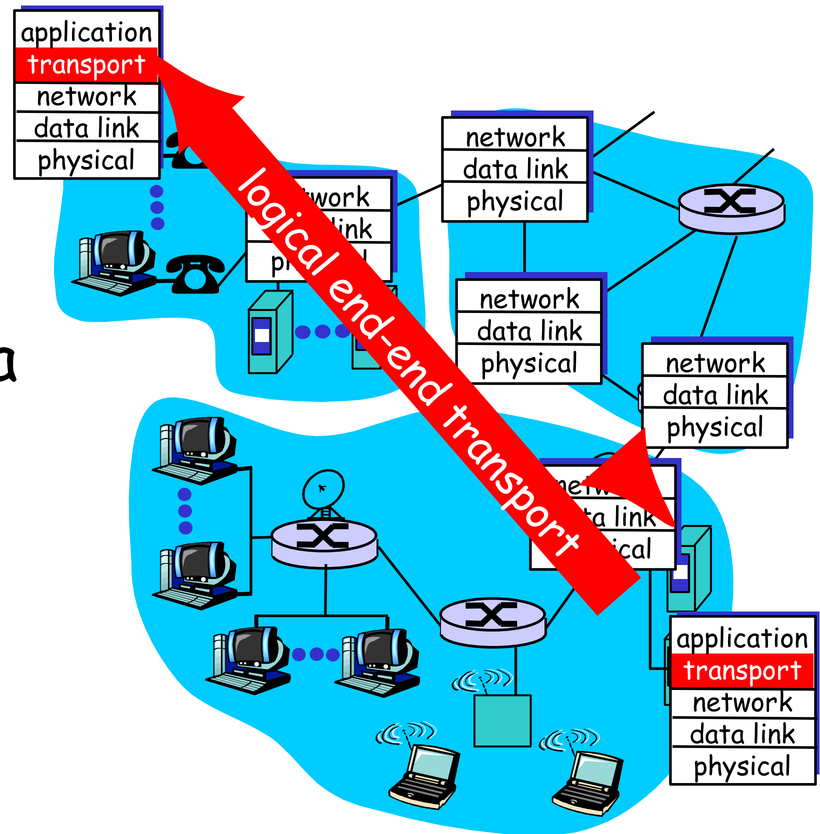
Analogia:

pracownicy firmy zamawiają pizzę

- procesy = pracownicy
- komunikaty = pizze
- hosty = firma i pizzeria
- protokół transportowy = zamawiający pracownik
- protokół sieci = doręczyciel pizzy

Protokoły transportowe Internetu

- niezawodna, uporządkowana komunikacja (TCP)
  - kontrola przeciążenia
  - kontrola przepływu
  - tworzenie połączenia
- zawodna, nieuporządkowana komunikacja (UDP)
  - proste rozszerzenie usługi "best-effort" IP
- niedostępne usługi:
  - gwarancje maksymalnego opóźnienia
  - gwarancje minimalnej przepustowości



Mapa wykładu

- Usługi warstwy transportu
- Multipleksacja i demultipleksacja
- Transport bezpołączeniowy: UDP
- Zasady niezawodnej komunikacji danych
- Transport połączeniowy: TCP
  - struktura segmentu
  - niezawodna komunikacja
  - kontrola przepływu
  - zarządzanie połączeniem
- Mechanizmy kontroli przeciążenia
- Kontrola przeciążenia w TCP

Multipleksacja/demultipleksacja

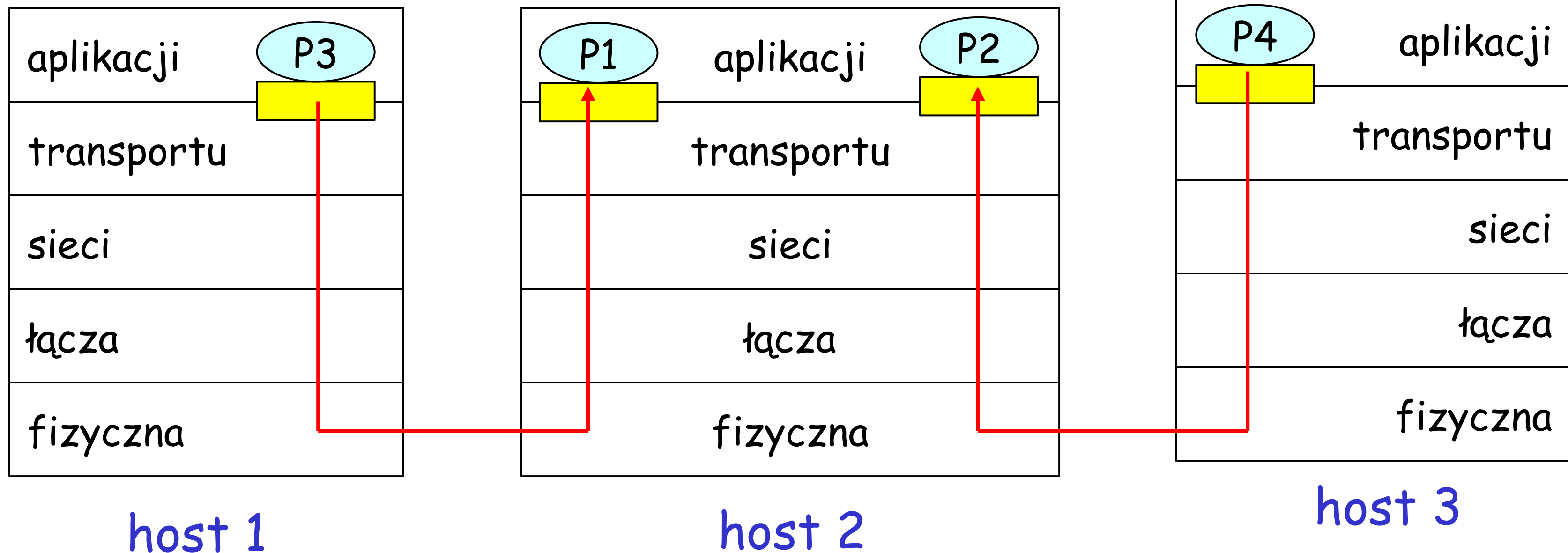
Demultipleksacja u odbiorcy

przekazywanie otrzymanych segmentów do właściwych gniazd

Multipleksacja u nadawcy

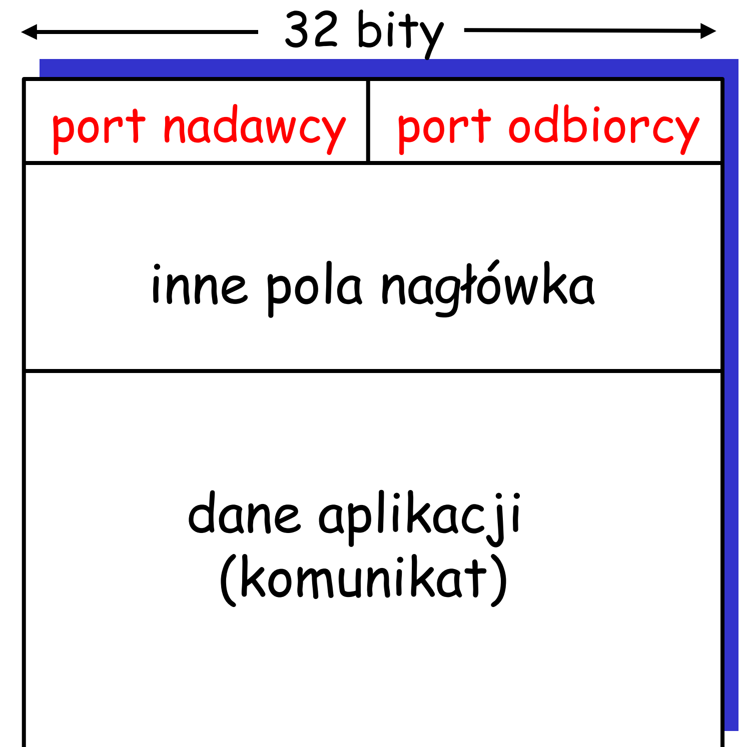
zbieranie danych z wielu gniazd, dodanie nagłówka (używanego później przy demultipleksacji)

= gniazdo = proces



Jak działa demultipleksacja

- host otrzymuje pakiety IP
  - każdy pakiet ma adres IP nadawcy, adres IP odbiorcy
  - każdy pakiet zawiera jeden segment warstwy transportu
  - każdy segment ma port nadawcy i odbiorcy (pamiętać: powszechnie znane numery portów dla określonych aplikacji)
- host używa adresu IP i portu żeby skierować segment do odpowiedniego gniazda



format segmentu TCP/UDP

Demultipleksacja bezpołączeniowa

- Gniazda są tworzone przez podanie numeru portu:

```
DatagramSocket mojeGniazdo1 =
   new DatagramSocket(99111);
DatagramSocket mojeGniazdo2 =
   new DatagramSocket(99222);
```

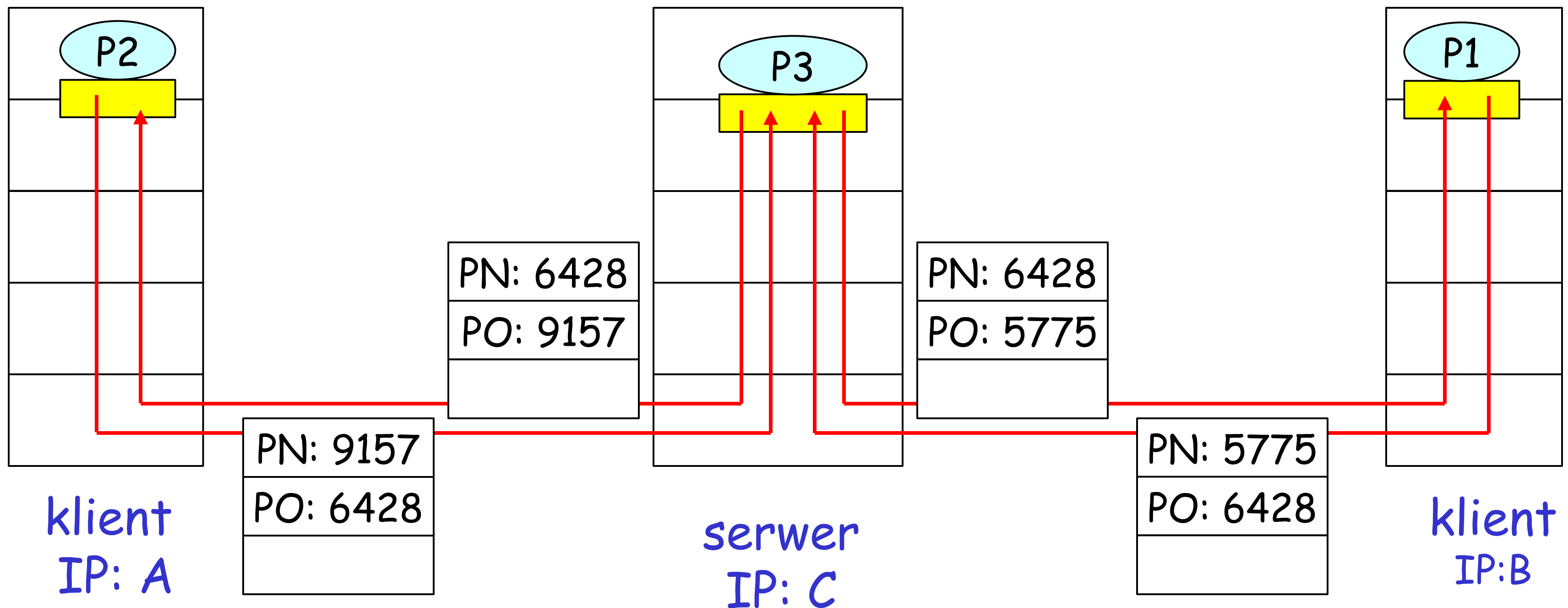
- Gniazdo UDP jest identyfikowane przez parę:

(adres IP odbiorcy, port odbiorcy)

- Kiedy host otrzymuje segment UDP:
  - sprawdza port odbiorcy w segmencie
  - kieruje segment UDP do gniazda z odpowiednim numerem portu
- Datagramy IP z różnymi adresami IP lub portami nadawcy są kierowane do tego samego gniazda

Demultipleksacja bezpołączeniowa (c.d.)

```
DatagramSocket gniazdoSerwera = new DatagramSocket(6428);
```

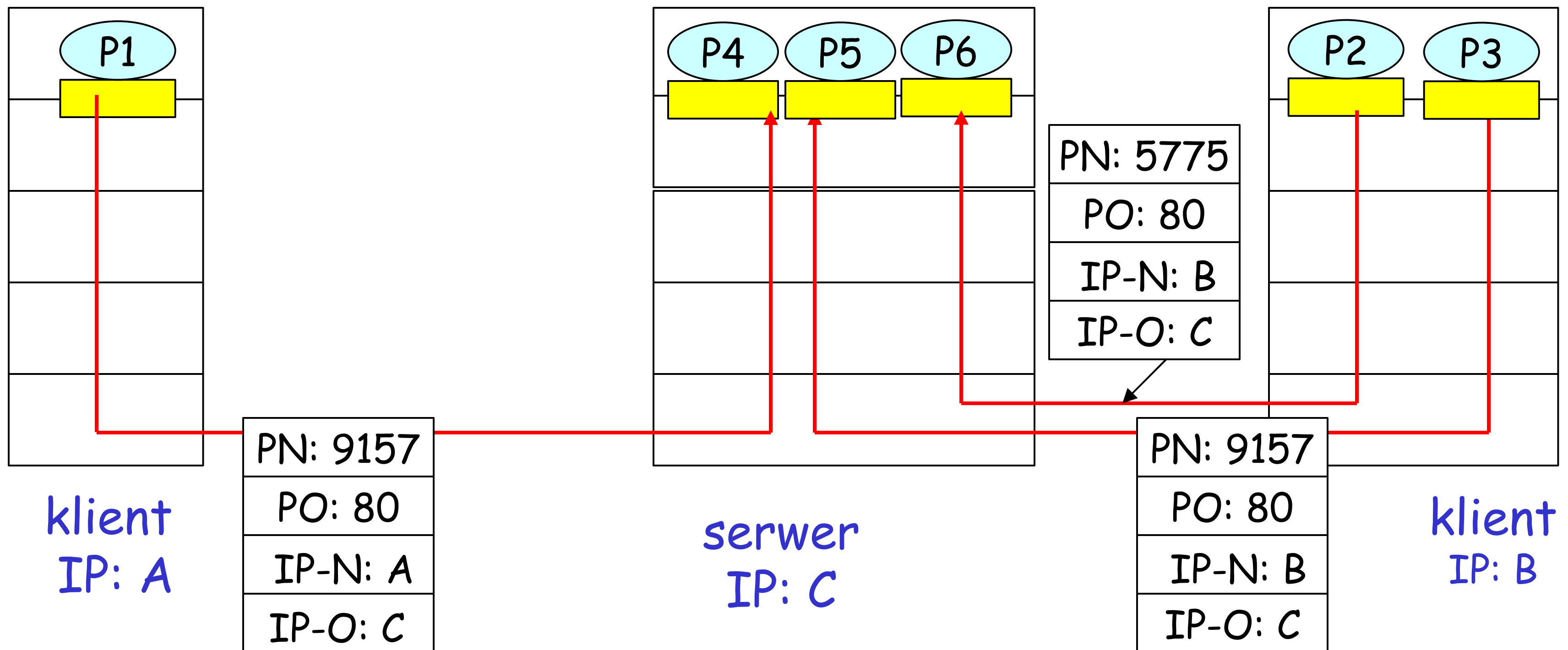


Port nadawcy (PN) jest „adresem zwrotnym".

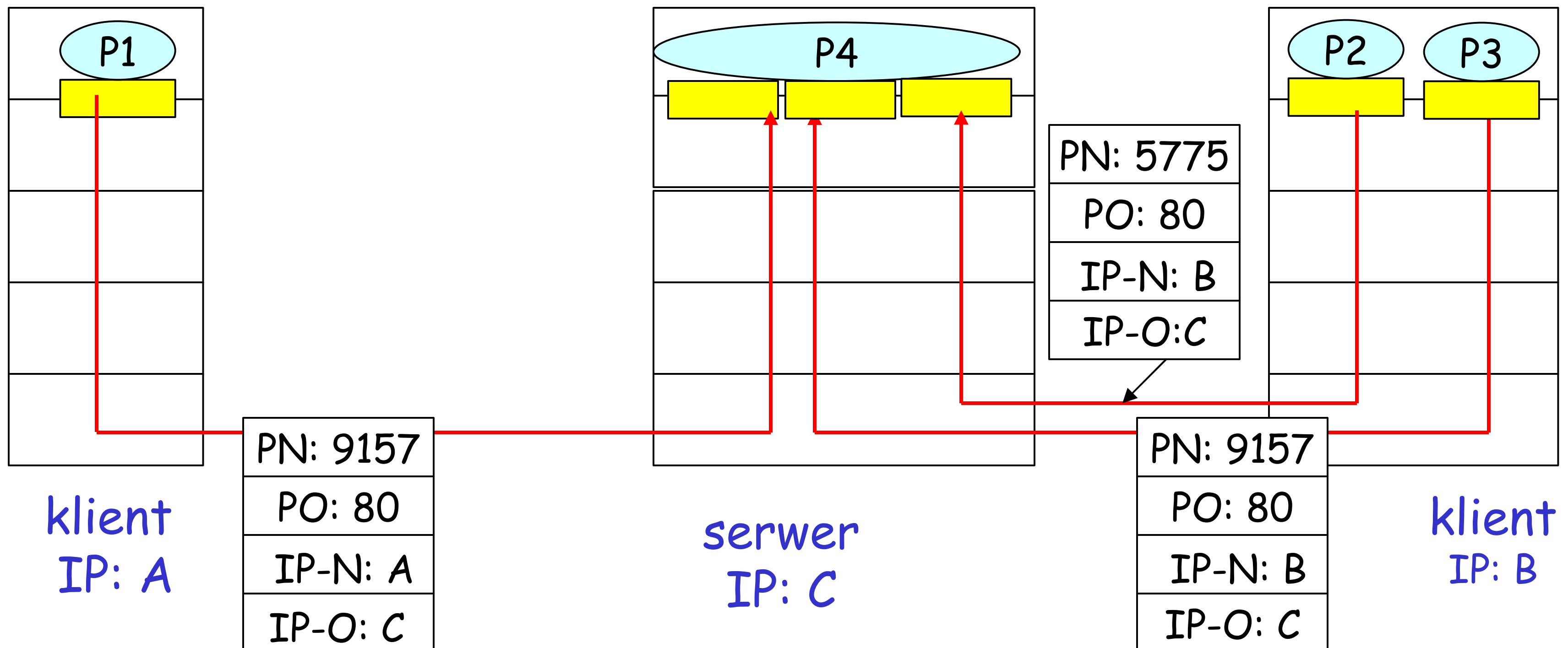
Demultipleksacja połączeniowa

- Gniazdo TCP jest określane przez cztery wartości:
  - adres IP nadawcy
  - port nadawcy
  - adres IP odbiorcy
  - port odbiorcy
- Host odbierający używa wszystkich 4 wartości, żeby skierować segment do właściwego gniazda
- Uwaga: host sprawdza także 5 wartość: protokół
- Host serwera może obsługiwać wiele gniazd TCP jednocześnie:
  - każde gniazdo ma inne 4 wartości
- Serwery WWW mają oddzielne gniazda dla każdego klienta
  - HTTP z nietrwałymi połączeniami wymaga oddzielnego gniazda dla każdego żądania

Demultipleksacja połączeniowa (c.d)



Demultipleksacja połączeniowa i serwer wielowątkowy



Porty komunikacyjne

- Numer przydzielony przez system: 0
  - po wywołaniu bind system wybiera numer portu 1024-5000 (znaleźć *go* można po wywołaniu *getsockname()*)
- Porty zarezerwowane: 1-1023
  - Porty dobrze znane: 1-255 (/etc/services)
  - Porty zwyczajowo zarezerwowane dla Unixa BSD: 256-511
  - Przydzielane przez rresvport: 512-1023
- Porty wolne 1024-65535

Mapa wykładu

- Usługi warstwy transportu
- Multipleksacja i demultipleksacja
- Transport bezpołączeniowy: UDP
- Zasady niezawodnej komunikacji danych
- Transport połączeniowy: TCP
  - struktura segmentu
  - niezawodna komunikacja
  - kontrola przepływu
  - zarządzanie połączeniem
- Mechanizmy kontroli przeciążenia
- Kontrola przeciążenia w TCP

UDP: User Datagram Protocol [RFC 768]

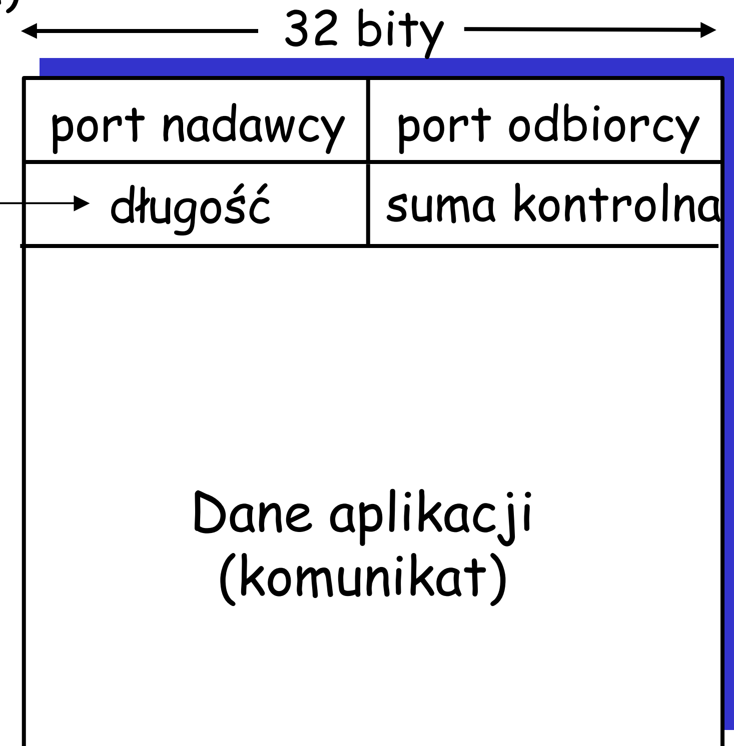
- "bez bajerów", "odchudzony" protokół transportowy Internetu
- usługa typu "best effort", segmenty UDP mogą zostać:
  - zgubione
  - dostarczone do aplikacji w zmienionej kolejności
- *bezpołączeniowy:*
  - nie ma inicjalizacji między nadawcą i odbiorcą UDP
  - każdy segment UDP jest obsługiwany niezależnie od innych

Czemu istnieje UDP?

- nie ma inicjalizacji połączenia (co może zwiększać opóźnienie)
- prosty: nie ma stanu połączenia u nadawcy ani odbiorcy
- mały nagłówek segmentu
- nie ma kontroli przeciążenia: UDP może słać dane tak szybko, jak chce

Więcej o UDP

- Często używane do komunikacji strumieniowej
  - tolerującej straty
  - wrażliwej na opóźnienia
- Inne zastosowania UDP
  - DNS
  - SNMP
- niezawodna komunikacja po UDP: dodać niezawodność w warstwie aplikacji
  - Praca domowa



Format segmentu UDP

Suma kontrolna UDP

Cel: odkrycie „błędów” (n.p., odwróconych bitów) w przesłanym segmencie

Nadawca:

- traktuje zawartość segmentu jako ciąg 16-bitowych liczb całkowitych
- suma kontrolna: dodawanie (i potem negacja sumy) zawartości segmentu
- nadawca wpisuje wartość sumy kontrolnej do odpowiedniego pola nagłówka UDP

Odbiorca:

- oblicza sumę kontrolną odebranego segmentu
- sprawdza, czy obliczona suma kontrolna jest równa tej, która jest w nagłówku:
  - NIE - wykryto błąd
  - TAK - Nie wykryto błędu. *Ale może błąd jest i tak?* Wrócimy do tego ….

Przykład sumy kontrolnej

- Uwaga
  - Dodając liczby, reszta z dodawania najbardziej znaczących bitów musi zostać dodana do wyniku (zawinięta, przeniesiona na początek)
- Przykład: suma kontrolna dwóch liczb 16-bitowych

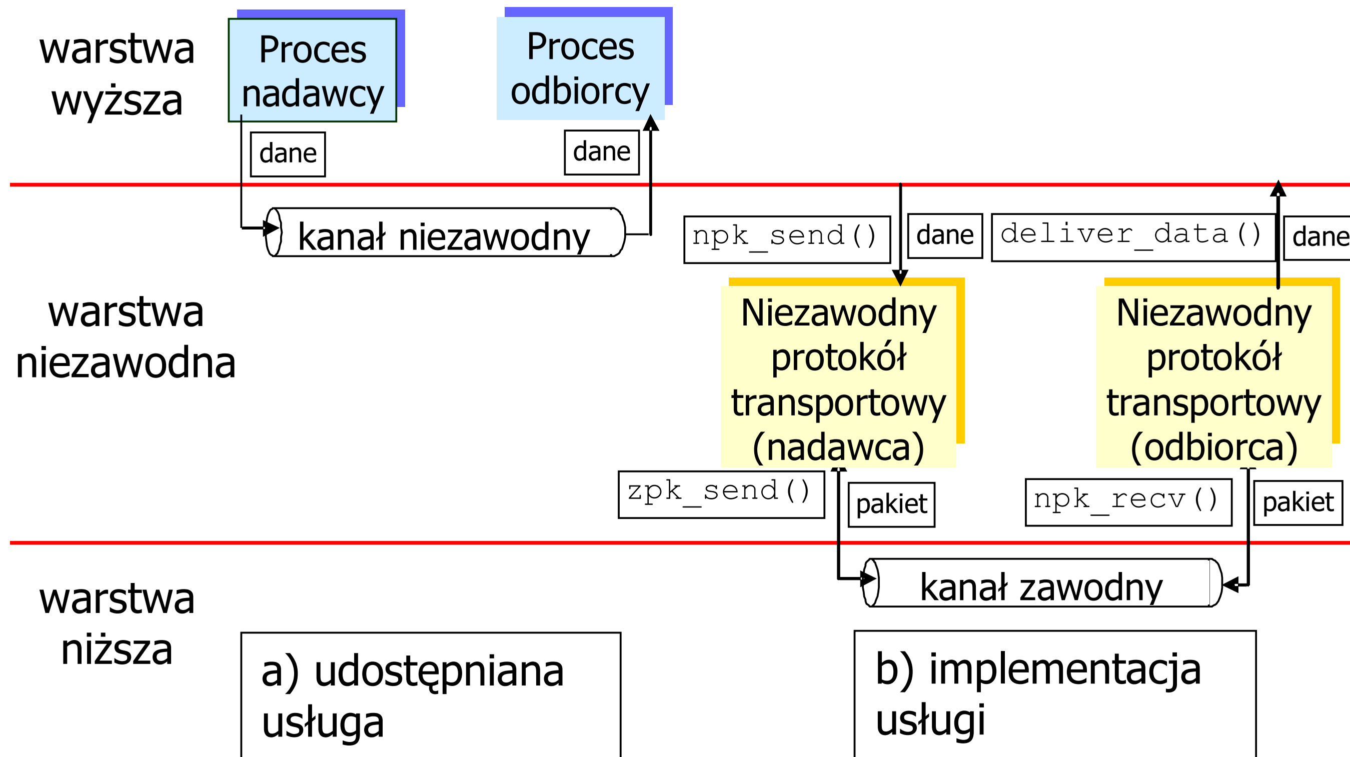
		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
zawinięcie	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
suma		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
suma kontrolna		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Mapa wykładu

- Usługi warstwy transportu
- Multipleksacja i demultipleksacja
- Transport bezpołączeniowy: UDP
- Zasady niezawodnej komunikacji danych
- Transport połączeniowy: TCP
  - struktura segmentu
  - niezawodna komunikacja
  - kontrola przepływu
  - zarządzanie połączeniem
- Mechanizmy kontroli przeciążenia
- Kontrola przeciążenia w TCP

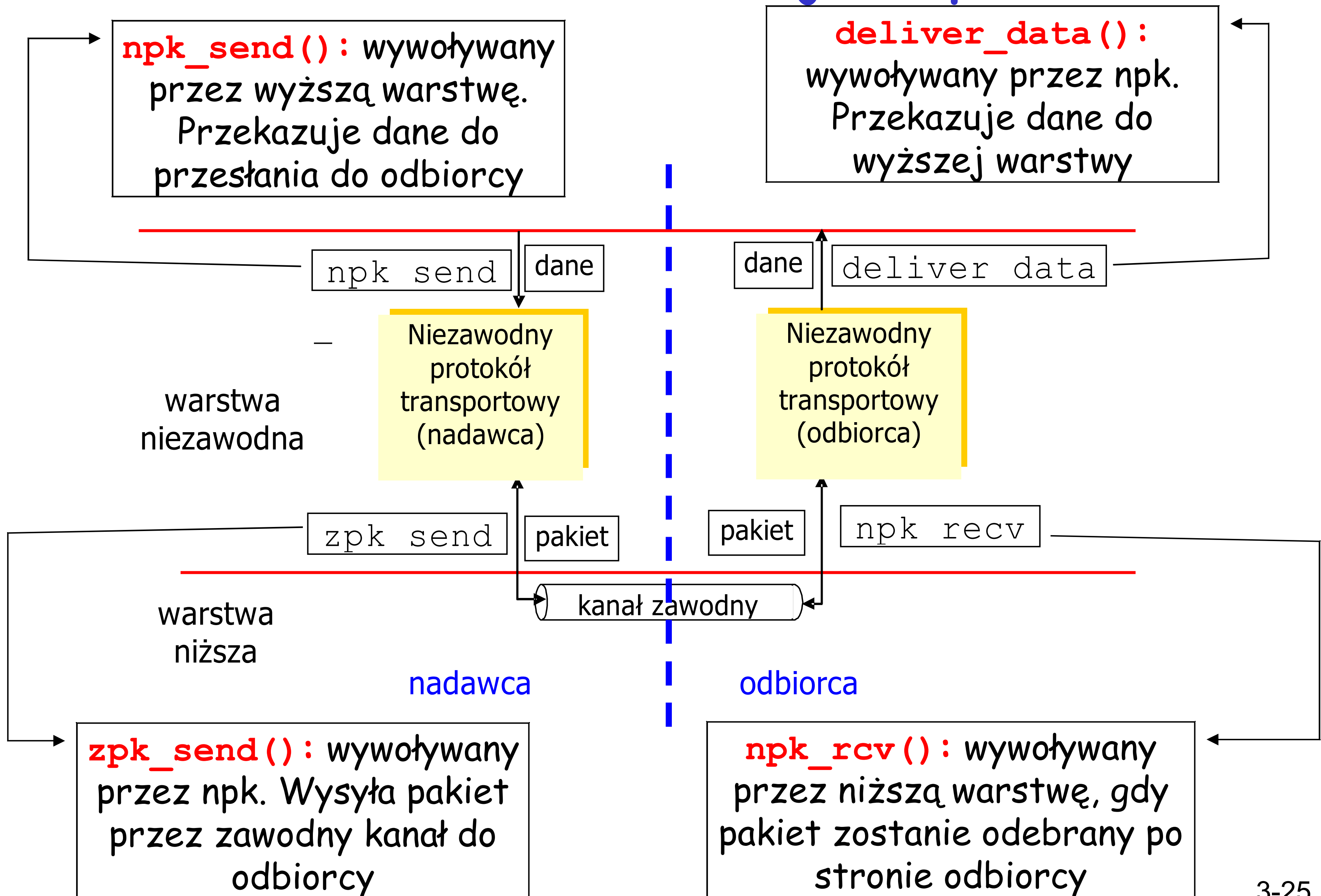
Zasady niezawodnej komunikacji danych

- Ważne w warstwie aplikacji, transportu i łącza
- Jeden z najważniejszych tematów w dziedzinie sieci!



- charakterystyka zawodnego kanału określa złożoność niezawodnego protokołu komunikacji (***npk***)

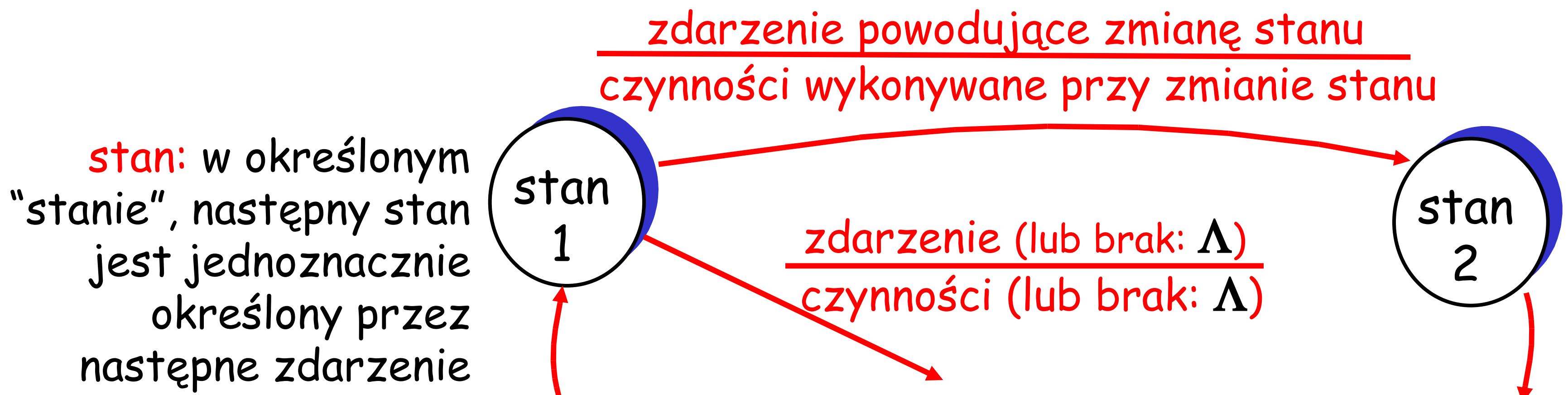
Niezawodna komunikacja (npk)



Niezawodna komunikacja: początki

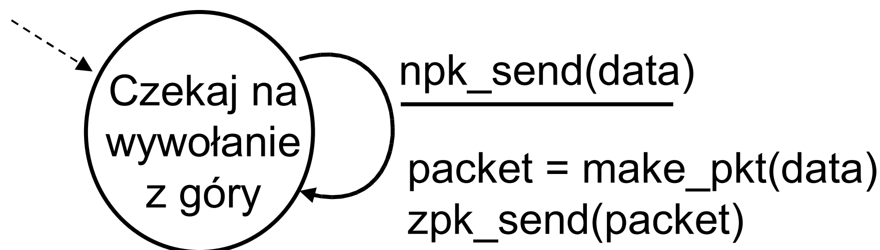
Co zrobimy:

- stopniowo zaprojektujemy nadawcę i odbiorcę niezawodnego protokołu komunikacji (npk)
- komunikacja danych tylko w jedną stronę
  - ale dane kontrolne w obie strony!
- użyjemy automatów skończonych (AS) do specyfikacji nadawcy, odbiorcy

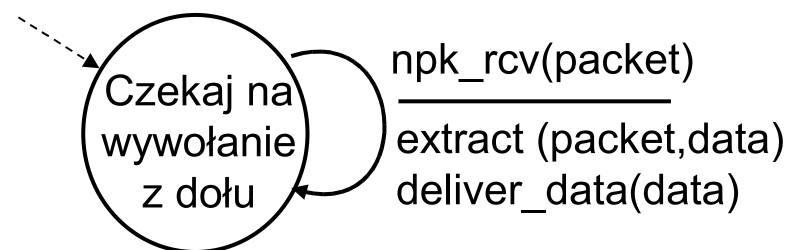


Npk1.0: niezawodna komunikacja przez niezawodny kanał

- używany kanał jest w pełni niezawodny
  - nie ma błędów bitowych
  - pakiety nie są tracone
- oddzielne AS dla nadawcy, odbiorcy:
  - nadawca wysyła dane przez kanał
  - odbiorca odbiera dane z kanału



nadawca

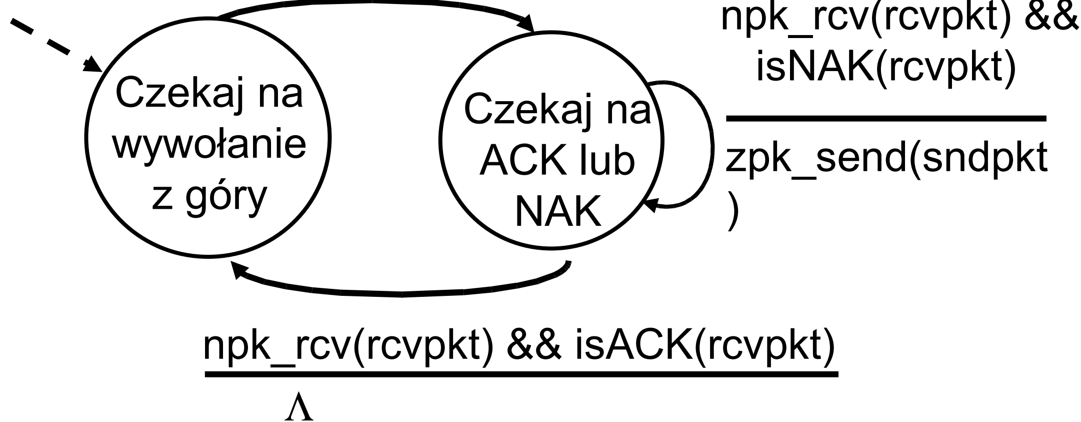


odbiorca

Npk2.0: kanał z błędami bitowymi

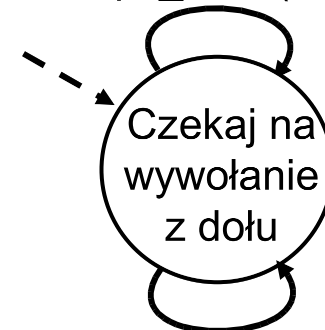
- kanał może zmieniać bity w pakiecie
  - suma kontrolna pozwala rozpoznać błędy bitowe
- *pytanie*: jak naprawić błąd:
  - *potwierdzenia (ang. acknowledgement, ACKs):* odbiorca zawiadamia nadawcę, że pakiet jest dotarł bez błędu
  - *negatywne potwierdzenia (NAKs):* odbiorca zawiadamia nadawcę, że pakiet ma błędy
  - nadawca retransmituje pakiet po otrzymaniu NAK
- nowe mechanizmy w npk2.0:
  - rozpoznawanie błędów
  - informacja zwrotna od odbiorcy: komunikaty kontrolne (ACK,NAK) odbiorca->nadawca

npk2.0: specyfikacja AS

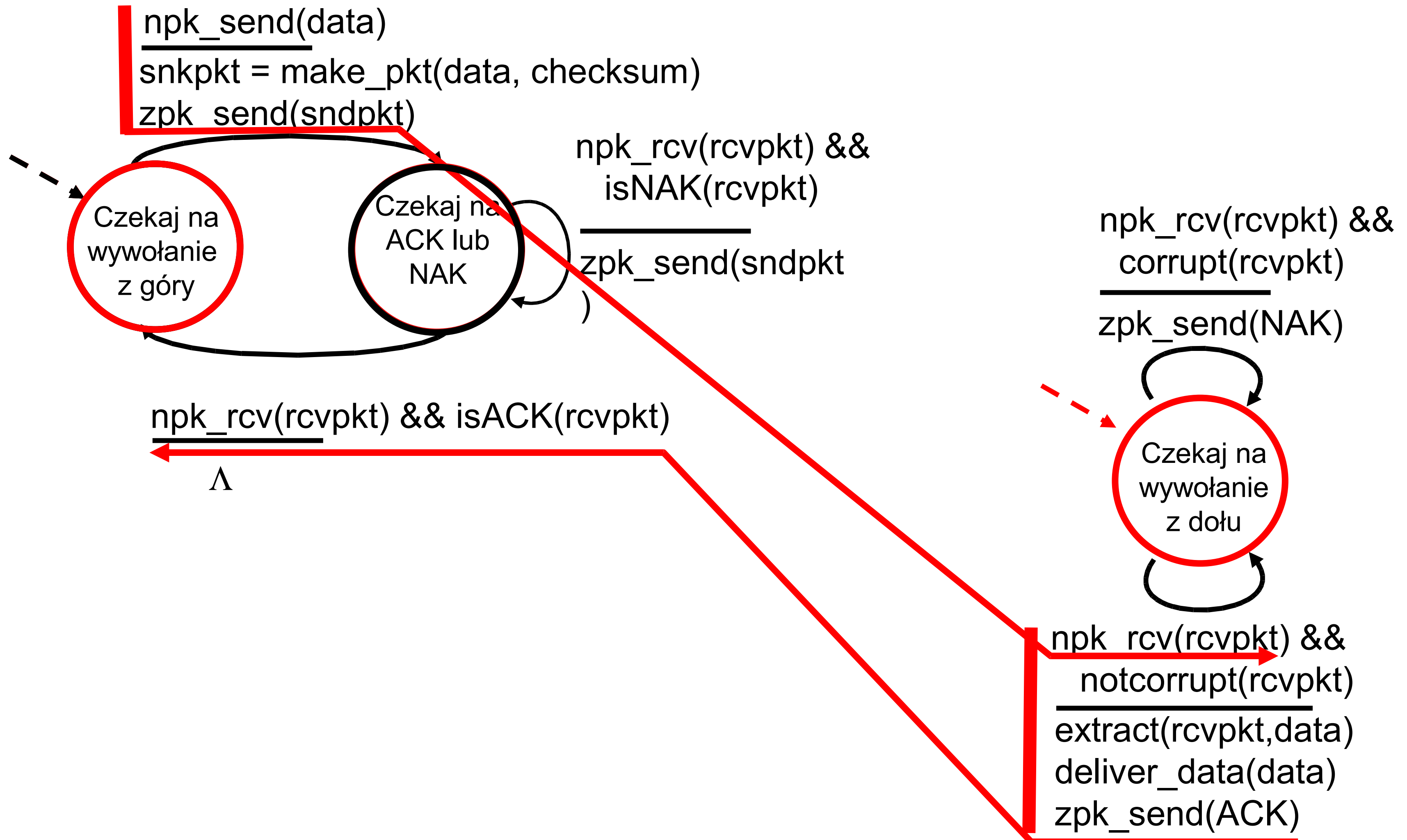


nadawca

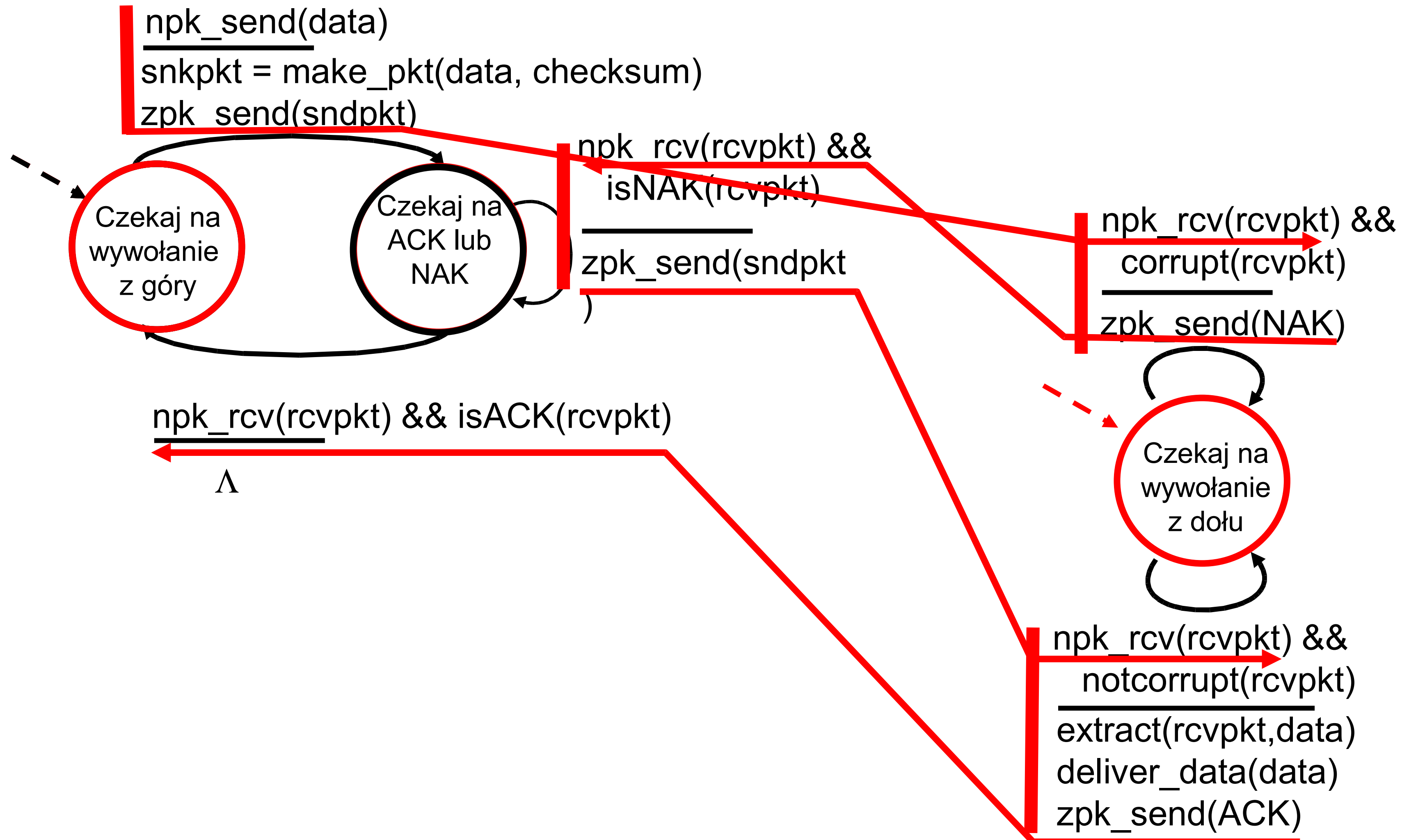
odbiorca



npk2.0: działanie bez błędów



npk2.0: działanie z błędami



npk2.0 ma fatalny błąd!

Co się stanie, gdy ACK/NAK będzie miał błąd?

- nadawca nie wie, co się stało u odbiorcy!
- nie można po prostu zawsze retransmitować: możliwe jest wysłanie pakietu podwójnie (duplikatu).

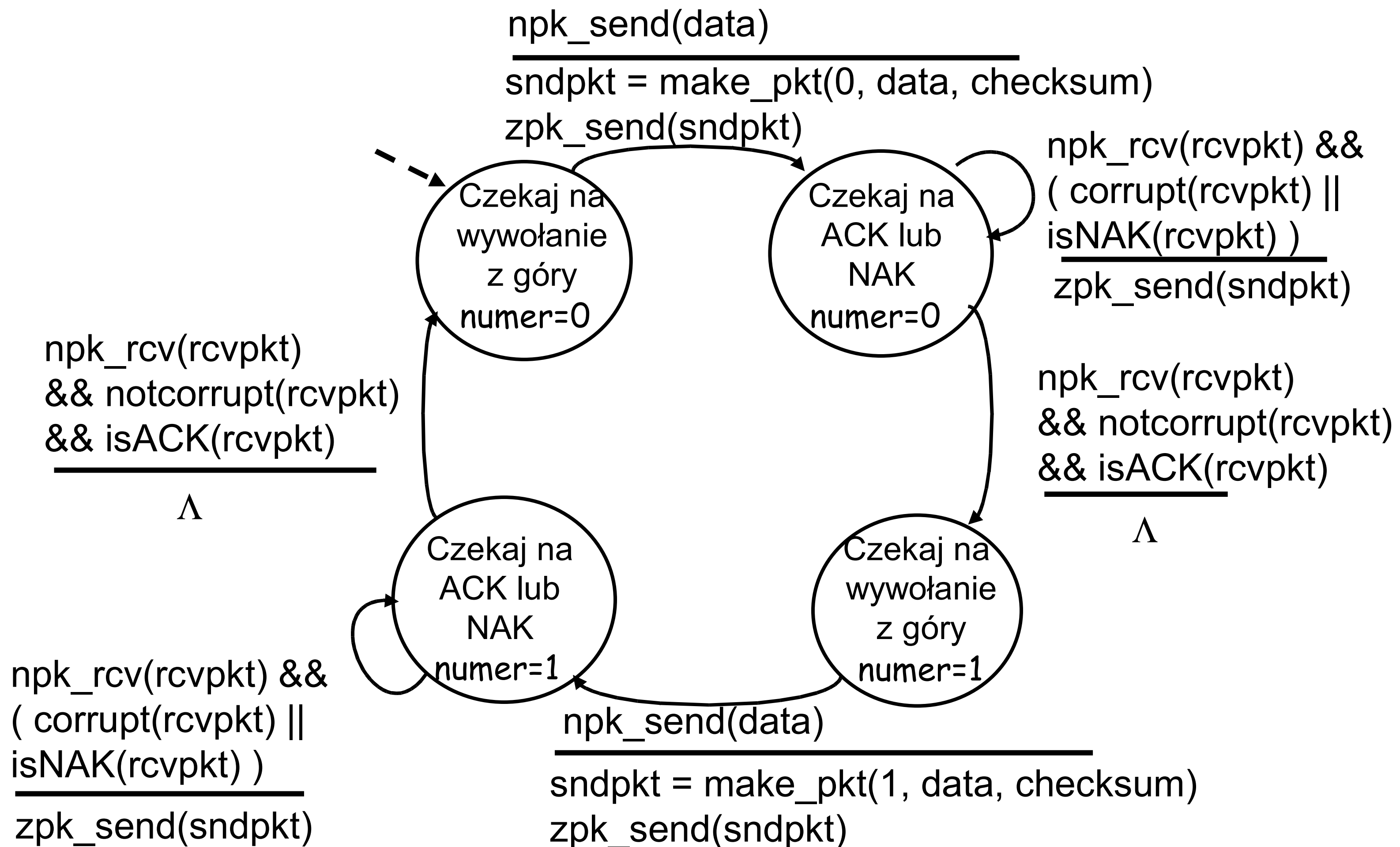
Obsługa duplikatów:

- nadawca dodaje *numer sekwencyjny* do każdego pakietu
- nadawca retransmituje aktualny pakiet, jeśli ACK/NAK ma błąd
- odbiorca wyrzuca (nie przekazuje wyżej) zduplikowane pakiety

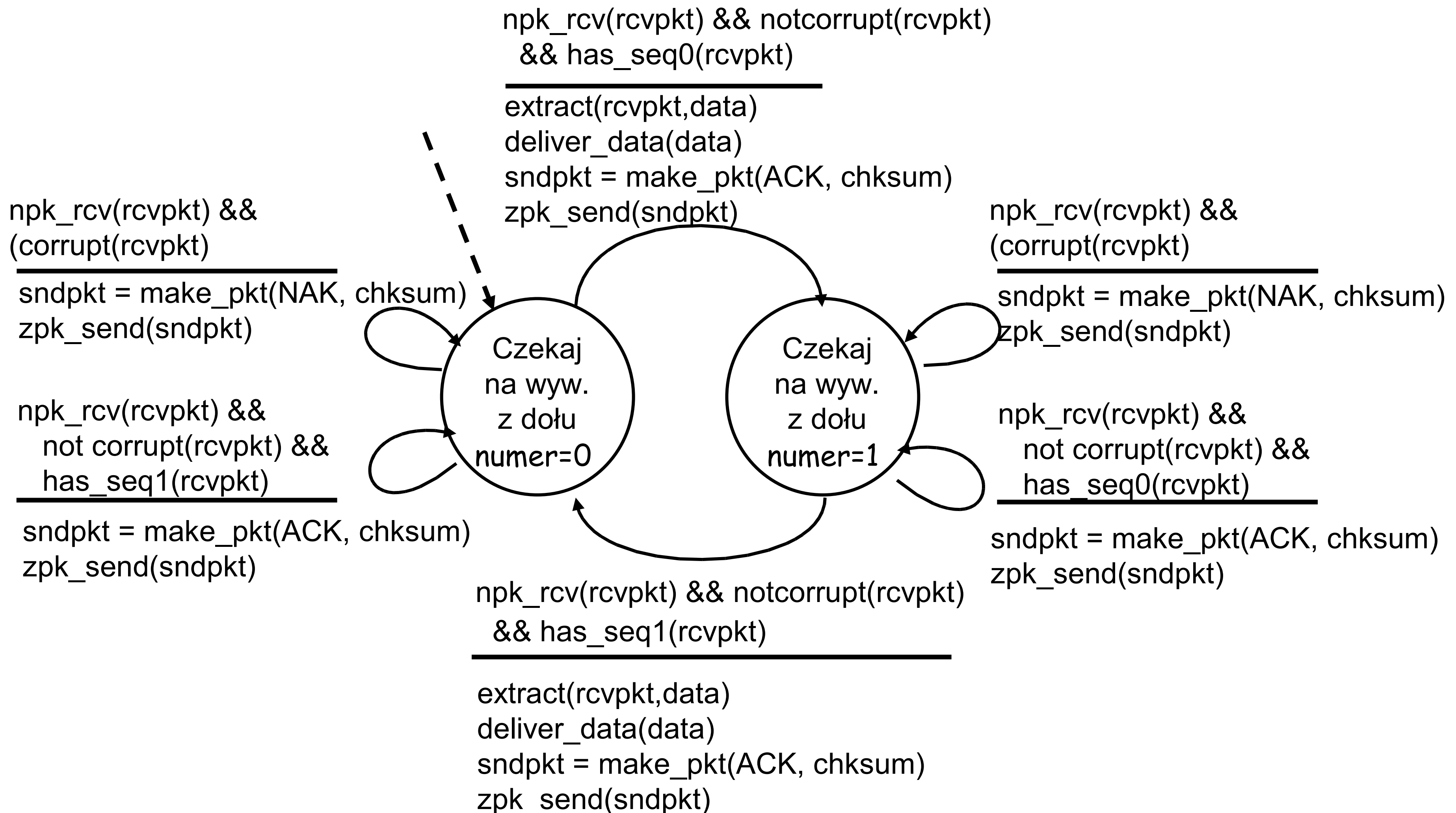
wstrzymaj i czekaj

Nadawca wysyła jeden pakiet, potem czeka na odpowiedź odbiorcy

npk2.1: nadawca, obsługuje błędne ACK/NAK



npk2.1: odbiorca, obsługuje błędne ACK/NAK



npk2.1: dyskusja

Nadawca:

- Dodaje numer sekwencyjny do pakietu
- Dwa numery (0,1) wystarczą. Dlaczego?
- musi sprawdzać, czy ACK/NAK jest poprawny
- dwa razy więcej stanów (niż w npk2.0)
  - stan musi "pamiętać" aktualny numer sekwencyjny (0 lub 1)

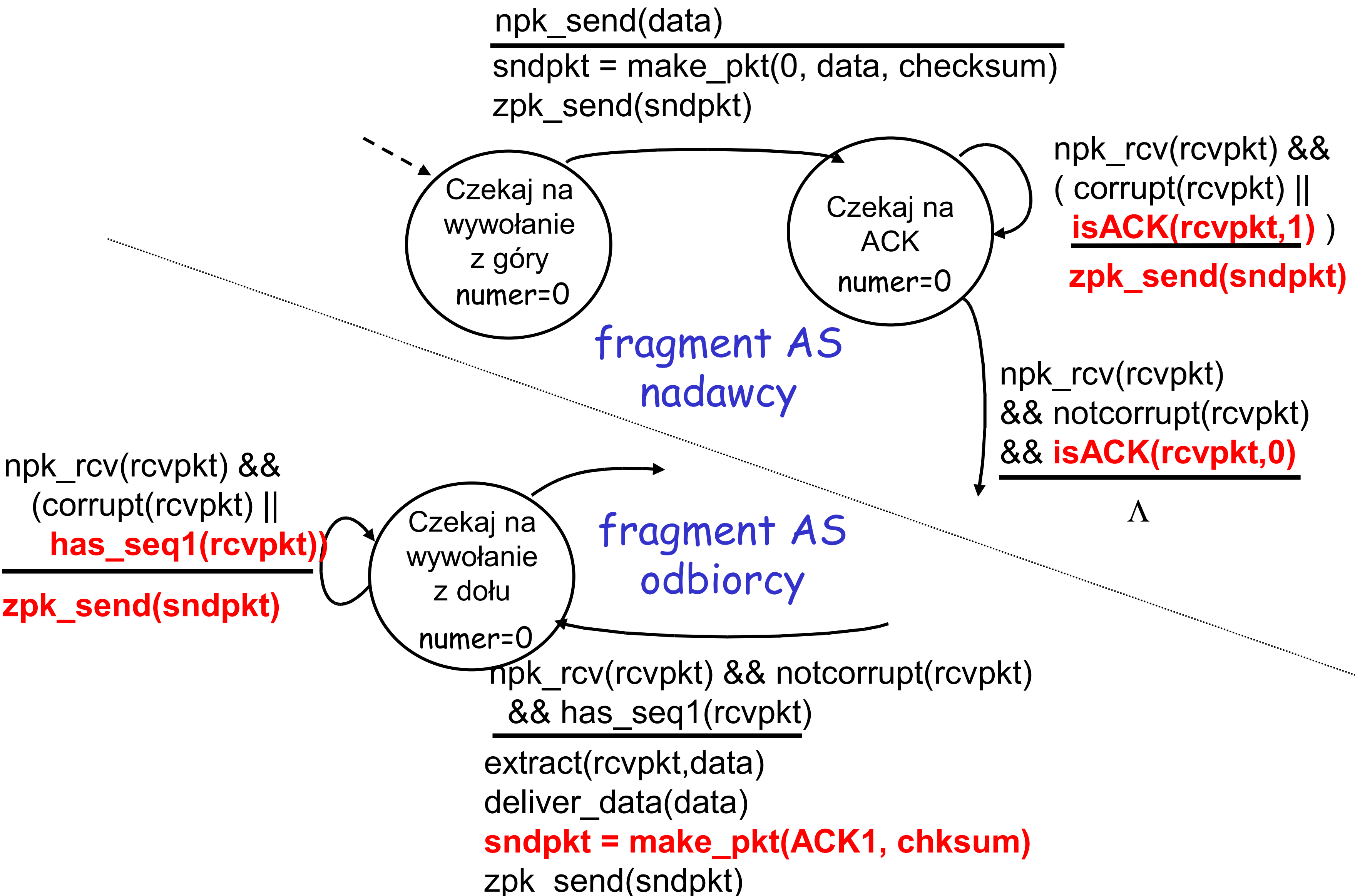
Odbiorca:

- musi sprawdzać, czy odebrany pakiet jest duplikatem
  - stan wskazuje, czy oczekuje numeru sekwencyjnego 0, czy 1
- uwaga: odbiorca może *nie wiedzieć* czy ostatni ACK/NAK został poprawnie odebrany przez nadawcę

npk2.2: protokół bez negatywnych potwierdzeń (NAK)

- ta sama funkcjonalność co w npk2.1, używając tylko zwykłych potwierdzeń (ACK)
- zamiast NAK, odbiorca wysyła ACK za ostatni poprawnie odebrany pakiet
  - odbiorca musi *dodać* numer sekwencyjny pakietu, który jest potwierdzany
- powtórne ACK u nadawcy powoduje tę samą czynność co NAK: *retransmisję ostatnio wysłanego pakietu*

npk2.2: fragmenty nadawcy, odbiorcy



npk3.0: kanał z błędami *oraz* stratami

Nowe założenie:

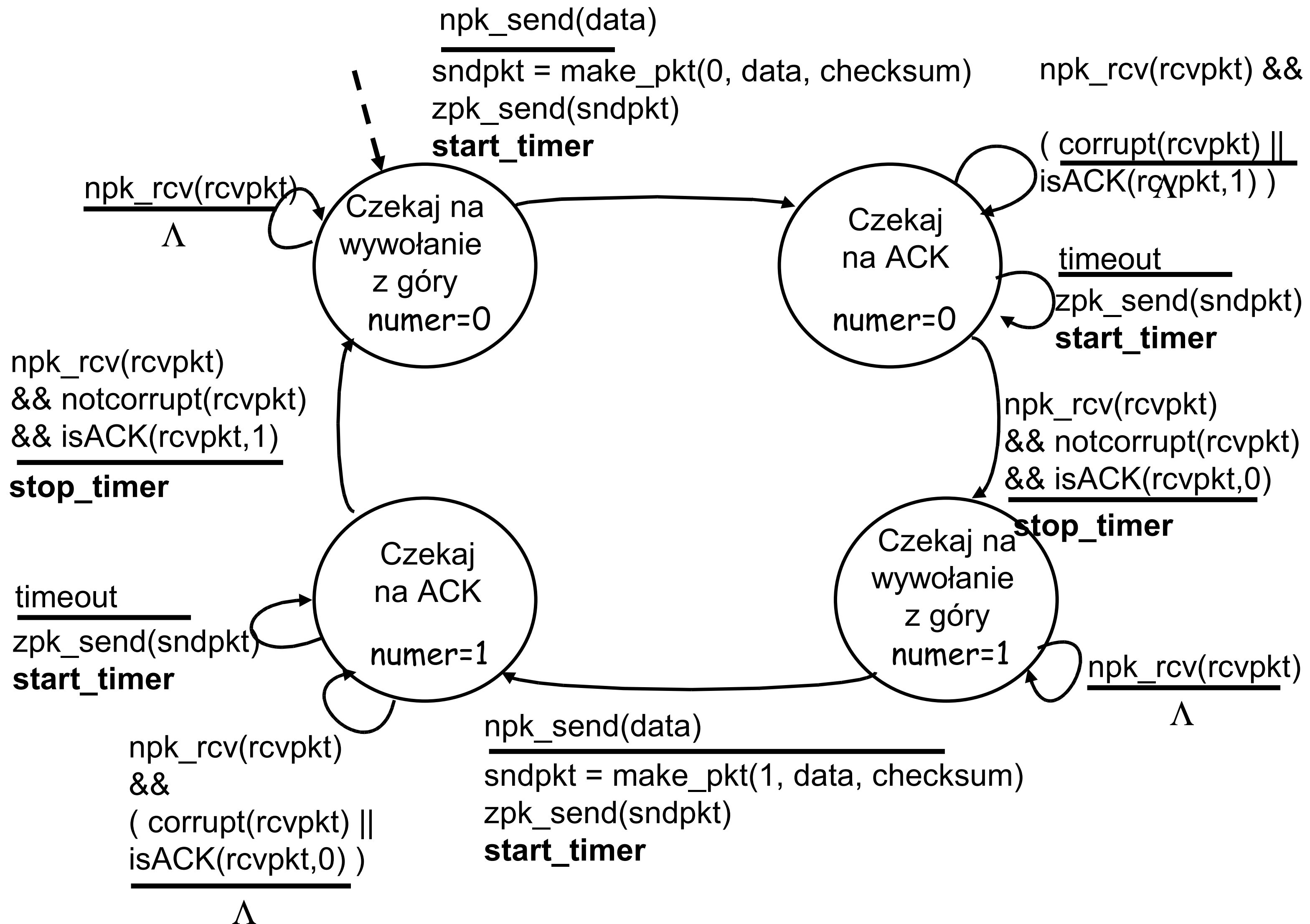
używany kanał może gubić pakiety (z danymi lub ACK)

- suma kontrolna, numery sekwencyjne, potwierdzenia, retransmisje będą pomocne, ale nie wystarczą

Podejście: nadawca czeka przez "rozsądny" czas na potwierdzenie ACK

- retransmituje, jeśli nie otrzyma ACK w tym czasie
- jeśli pakiet (lub ACK) jest tylko opóźniony, ale nie stracony:
  - retransmisja będzie duplikatem, ale za pomocą numerów sekwencyjnych już to obsługujemy
  - odbiorca musi określić numer sekwencyjny pakietu, który jest potwierdzany
- wymagany jest licznik czasu

npk3.0 nadawca



npk3.0 w działaniu

nadawca | odbiorca

wyślij pkt_0

pkt_0

odbierz pkt_0
wyślij ACK_0

ACK_0

odbierz ACK_0
wyślij pkt_1

pkt_1

odbierz pkt_1
wyślij ACK_1

ACK_1

odbierz ACK_1
wyślij pkt_0

pkt_0

odbierz pkt_0
wyślij ACK_0

ACK_0

działanie bez strat

nadawca | odbiorca

wyślij pkt_0

pkt_0

odbierz pkt_0
wyślij ACK_0

ACK_0

odbierz ACK_0
wyślij pkt_1

pkt_1

(strata)

timeout, retransmituj pkt_1

pkt_1

odbierz pkt_1
wyślij ACK_1

ACK_1

odbierz ACK_1
wyślij pkt_0

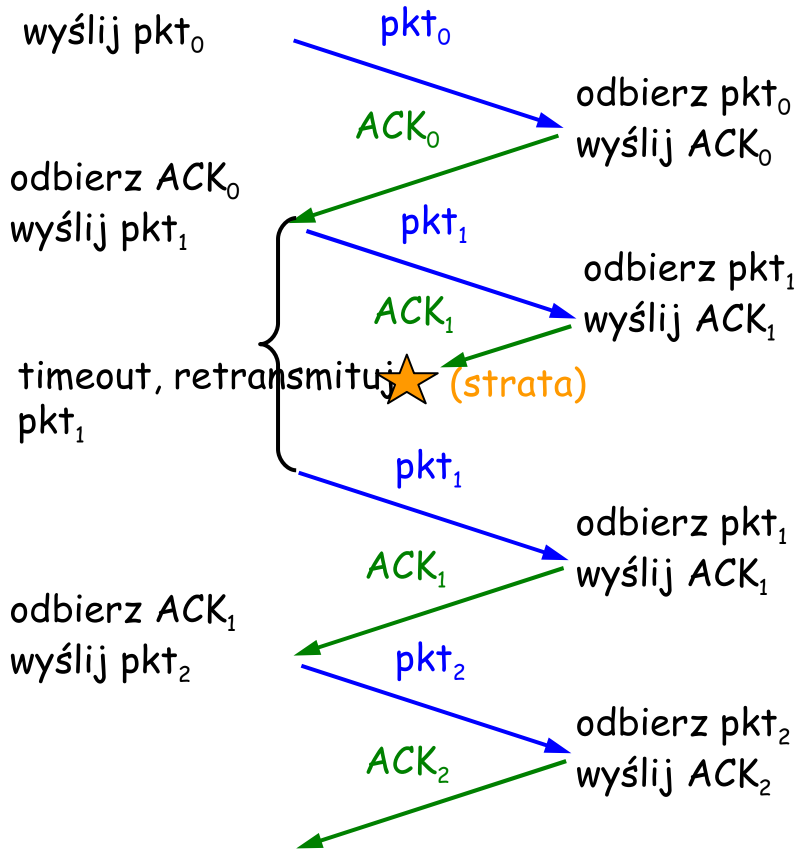
pkt_0

odbierz pkt_2
wyślij ACK_0

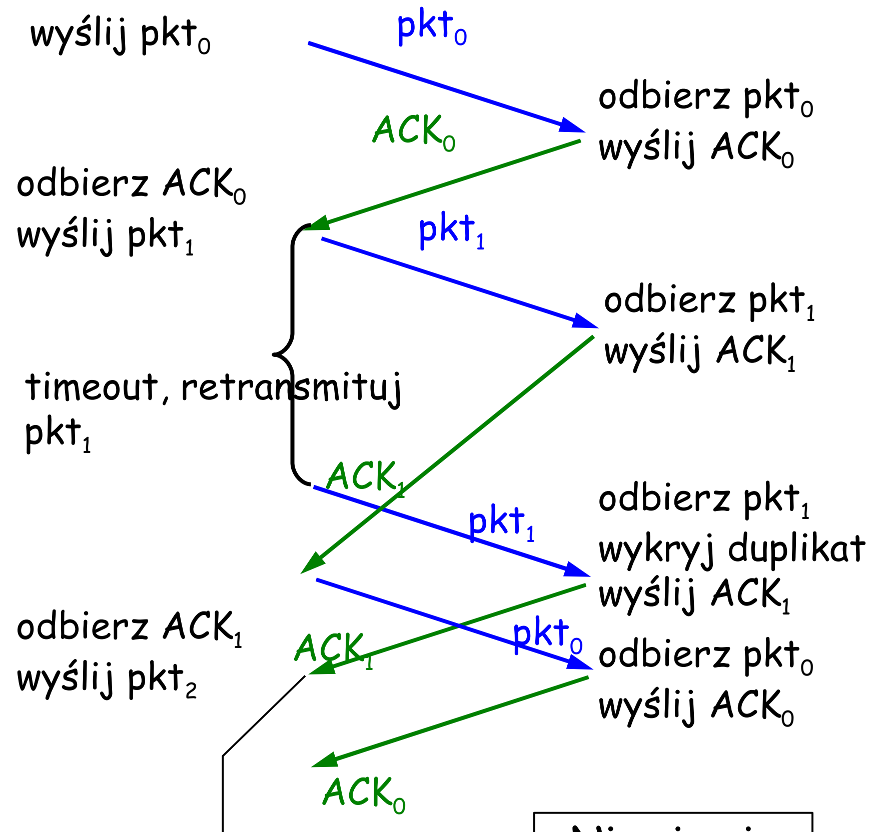
ACK_0

działanie ze stratą pakietu

npk3.0 w działaniu



działanie ze stratą ACK



za wczesny timeout

Wydajność npk3.0

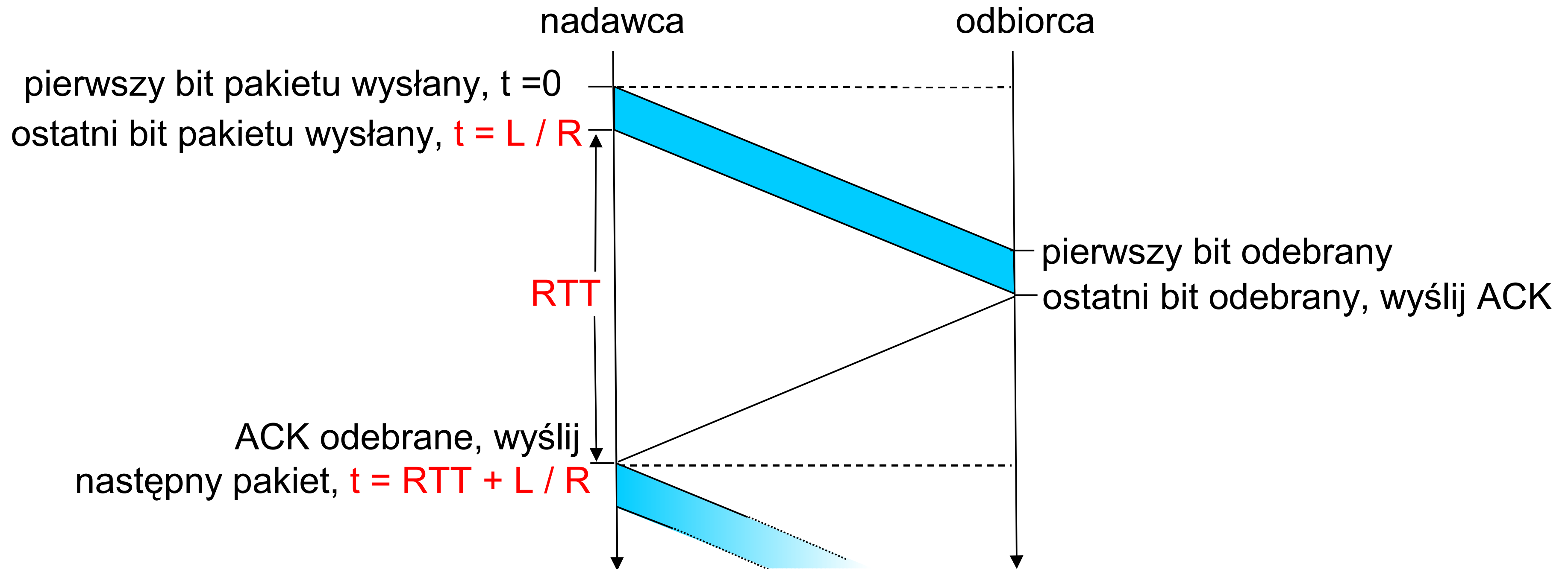
- npk3.0 działa, ale wydajność ma bardzo kiepską
- przykład: link 1 Gb/s, opóźnienie k-k 15 ms, pakiet 1KB:

$$T_{transmisji} = \frac{L \text{ (rozmiar pakietu w b)}}{R \text{ (przepustowość, b/s)}} = \frac{8\text{kb/pkt}}{10^9 \text{ b/s}} = 8 \text{ mikros.}$$

$$W_{nadawcy} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

  - $W_{nadawcy}$: wykorzystanie – procent czasu, w jakim nadawca nadaje
  - pakiet rozmiaru 1KB co 30 ms -> przepustowość 33kB/s przez łącze 1 Gb/s
  - protokół ogranicza wykorzystanie fizycznych zasobów łącza!

npk3.0: działanie wyślij i czekaj

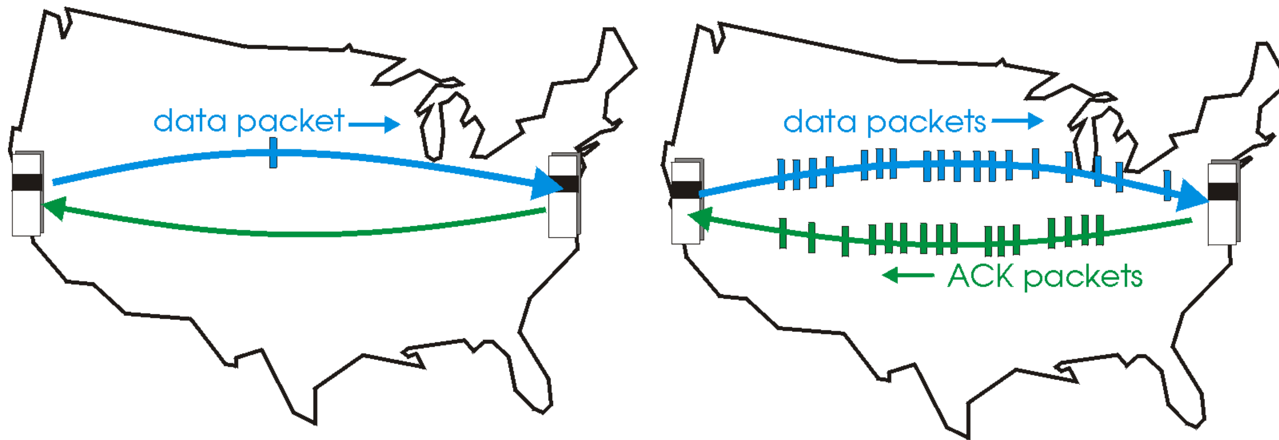


$$W_{nadawcy} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Protokoły "wysyłające grupowo"

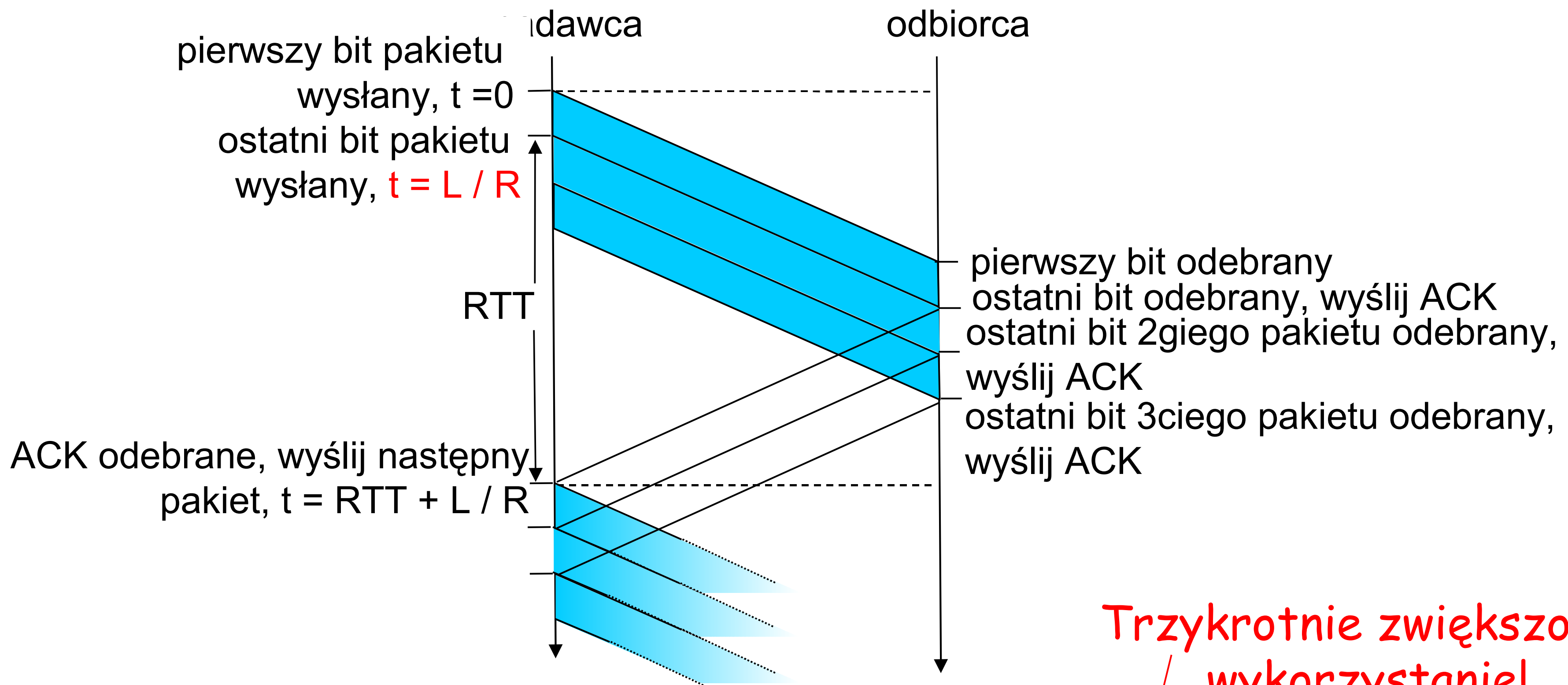
Wysyłanie grupowe: nadawca wysyła wiele pakietów bez czekania na potwierdzenie

- trzeba zwiększyć zakres numerów sekwencyjnych
- trzeba mieć bufor u nadawcy i/lub odbiorcy



- Dwa podstawowe rodzaje protokołów wysyłania grupowego: *wróć o N, selektywne powtarzanie*

Wysyłanie grupowe: zwiększone wykorzystanie

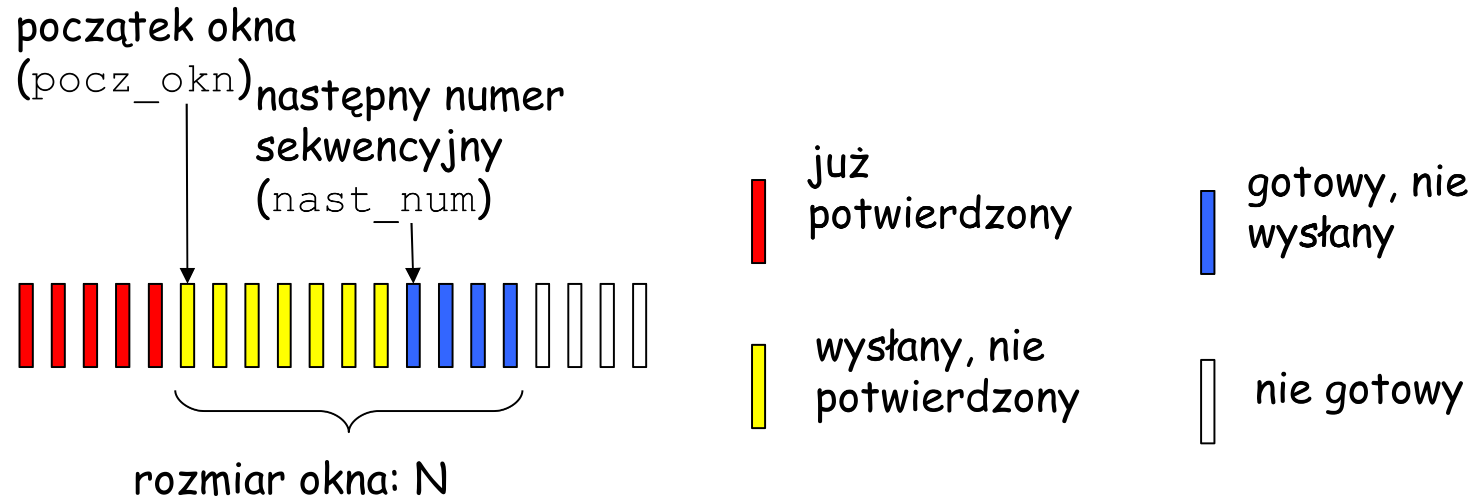


$$W_{nadawcy} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Wróć o N (WN)

Nadawca:

- k bitów na numer sekwencyjny w nagłówku pakietu
- wysyła "okno" co najwyżej N kolejnych, niepotwierdzonych pakietów



- ACK(n): potwierdza wszystkie pakiety aż do (i łącznie z) pakietem o numerze sekwencyjnym *n* - "skumulowany ACK"
  - może otrzymywać duplikaty potwierdzeń (patrz odbiorca)
- potrzebny jest zegar – jeden dla całego okna
- *timeout:* retransmisja wszystkich niepotwierdzonych pakietów w oknie, czyli od `pocz_okn` do `nast_num`

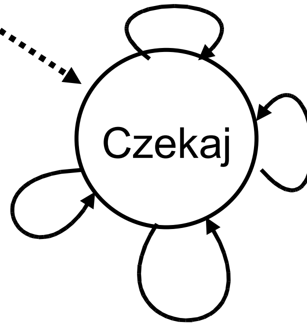
WN: rozszerzony AS nadawcy

npk_send(dane)

if (nast_num < pocz_okn+N)
{
sndpkt[nast_num] = make_pkt(nast_num, dane, suma_kontr)
zpk_send(sndpkt[nast_num])
if (pocz_okn == nast_num)
start_timer
nast_num++
}
else
refuse_data(dane)

Λ

pocz_okn=1
nast_num=1



timeout

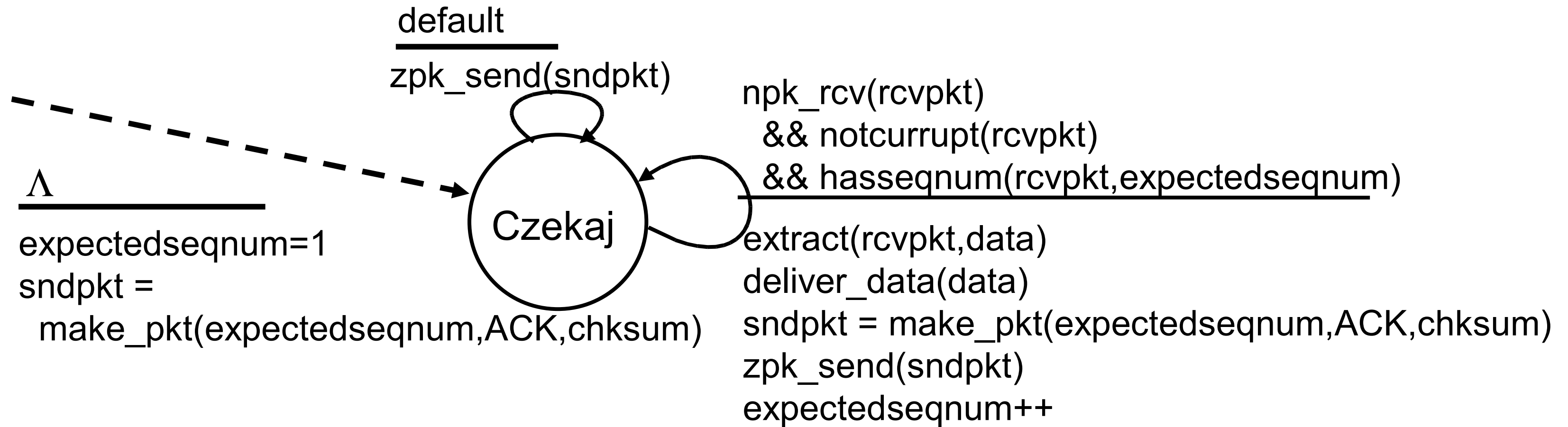
start_timer
zpk_send(sndpkt[pocz_okn])
zpk_send(sndpkt[pocz_okn+1])
…
zpk_send(sndpkt[nast_num-1])

npk_rcv(rcvpkt)
&& corrupt(rcvpkt)

npk_rcv(rcvpkt) &&
notcorrupt(rcvpkt)

pocz_okn = numer_ACK(rcvpkt) + 1
If (pocz_okn == nast_num)
stop_timer
else
start_timer

WN: rozszerzony AS odbiorcy



tylko ACK: zawsze wysyła ACK dla ostatniego poprawnie odebranego pakietu spośród pakietów odebranych *w kolejności*

- może generować zduplikowane ACK
- trzeba pamiętać tylko **`expectedseqnum`**

- pakiety nie w kolejności:
  - są wyrzucane -> nie ma buforowania u odbiorcy!
  - Wysyłane jest ponownie ACK z numerem sekwencyjnym ostatniego pakiety odebranego w kolejności

WN w działaniu

N = 4