

# Capítulo 3: Camada de Transporte

## Metas do capítulo:

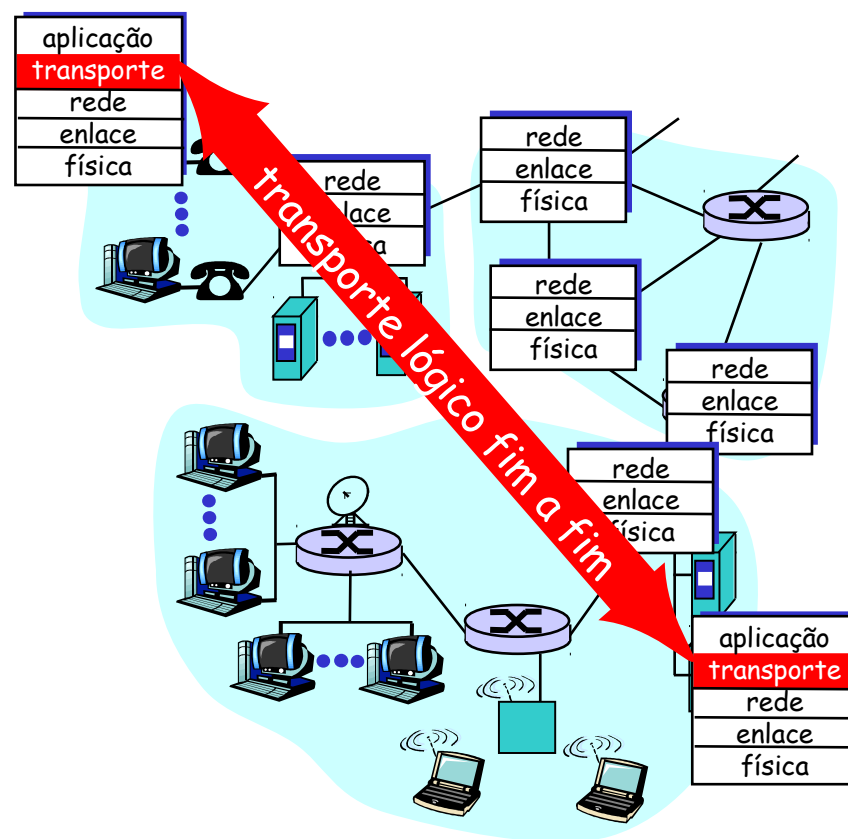
- compreender os princípios atrás dos serviços da camada de transporte:
  - multiplexação/demultiplexação
  - transferência confiável de dados
  - controle de fluxo
  - controle de congestionamento
- aprender os protocolos da camada de transporte da Internet:
  - UDP: transporte sem conexão
  - TCP: transporte orientado a conexões
  - Controle de congestionamento do TCP

# Conteúdo do Capítulo 3

- ▢ 3.1 Serviços da camada de transporte
- ▢ 3.2 Multiplexação e demultiplexação
- ▢ 3.3 UDP: Transporte não orientado a conexão
- ▢ 3.4 Princípios da transferência confiável de dados
- ▢ 3.5 Transporte orientado a conexão: TCP
  - ▢ transferência confiável
  - ▢ controle de fluxo
  - ▢ gerenciamento de conexões
- ▢ 3.6 Princípios de controle de congestionamento
- ▢ 3.7 Controle de congestionamento do TCP

# Serviços e protocolos de transporte

- provê *comunicação lógica* entre processos de aplicação executando em hospedeiros diferentes
- protocolos de transporte executam em sistemas finais:
  - lado transmissor: quebra as mensagens das aplicações em *segmentos*, repassa-os para a camada de rede
  - lado receptor: remonta as mensagens a partir dos segmentos, repassa-as para a camada de aplicação
- existem mais de um protocolo de transporte disponível para as aplicações
  - Internet: TCP e UDP



# Camadas de Transporte x rede

- *camada de rede:*  
comunicação lógica entre hospedeiros
- *camada de transporte:*  
comunicação lógica entre processos
  - depende de, estende serviços da camada de rede

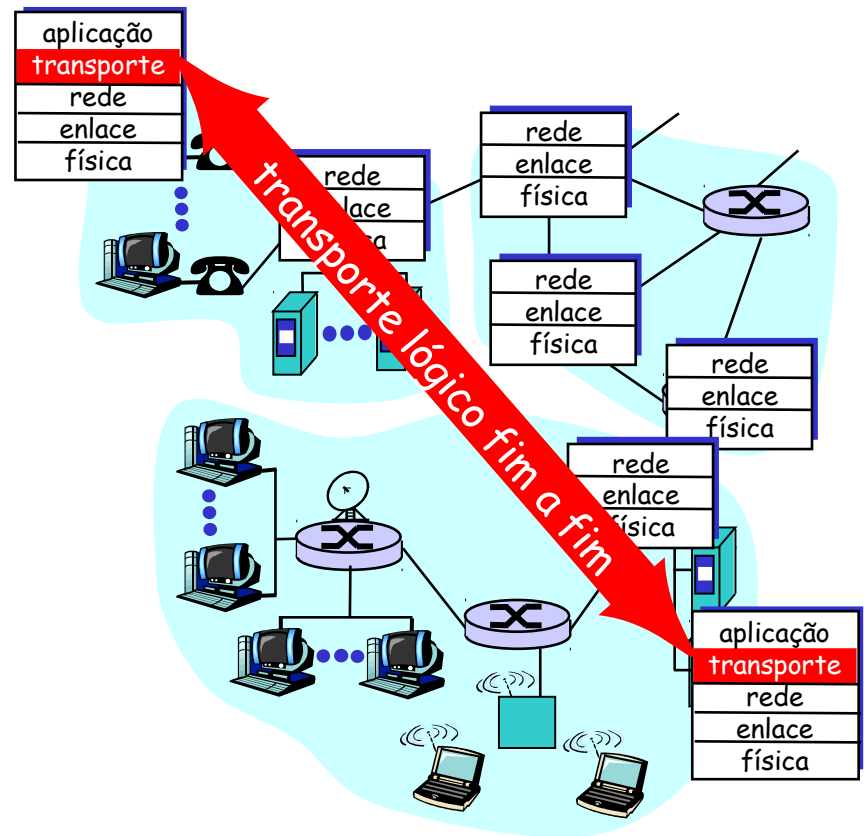
## Analogia doméstica:

*12 crianças enviando cartas para 12 crianças*

- processos = crianças
- mensagens da apl. = cartas nos envelopes
- hospedeiros = casas
- protocolo de transporte = Ann e Bill
- protocolo da camada de rede = serviço postal

# Protocolos da camada de transporte Internet

- ▮ entrega confiável, ordenada (TCP)
  - ▮ controle de congestionamento
  - ▮ controle de fluxo
  - ▮ estabelecimento de conexão ("setup")
- ▮ entrega não confiável, não ordenada: UDP
  - ▮ extensão sem "frescuras" do "melhor esforço" do IP
- ▮ serviços não disponíveis:
  - ▮ garantias de atraso
  - ▮ garantias de largura de banda




# Conteúdo do Capítulo 3

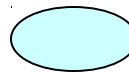
- 3.1 Serviços da camada de transporte
- 3.2 Multiplexação e demultiplexação
- 3.3 UDP: Transporte não orientado a conexão
- 3.4 Princípios da transferência confiável de dados
- 3.5 Transporte orientado a conexão: TCP
  - transferência confiável
  - controle de fluxo
  - gerenciamento de conexões
- 3.6 Princípios de controle de congestionamento
- 3.7 Controle de congestionamento do TCP

# Multiplexação/demultiplexação

## Demultiplexação no receptor:

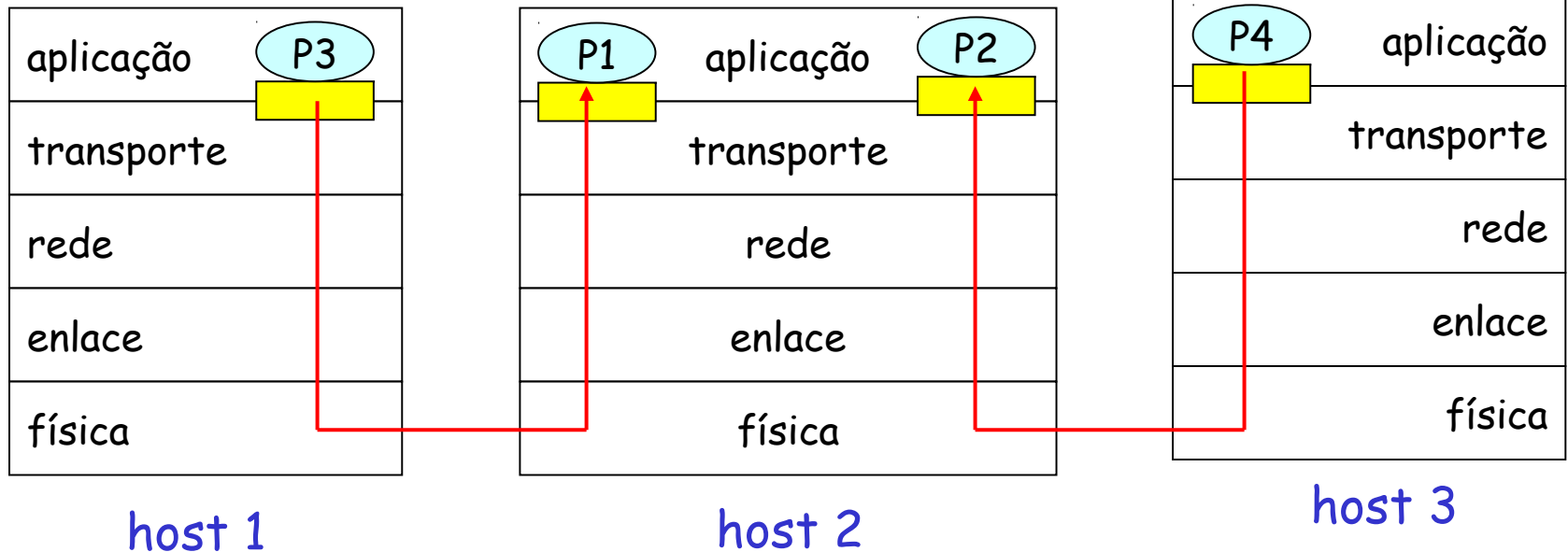
Entrega dos segmentos recebidos ao socket correto

 = socket

 = processo

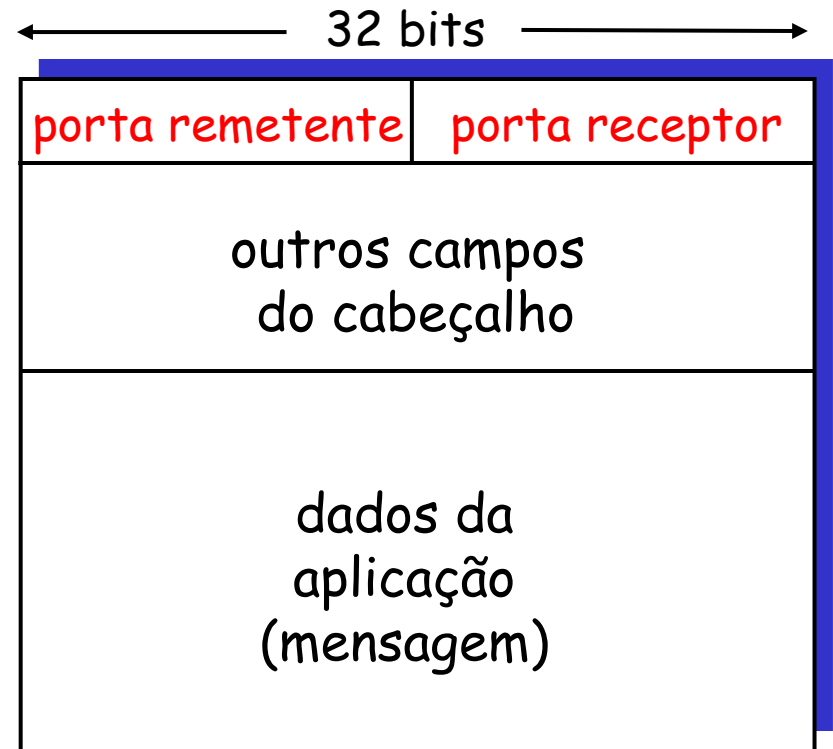
## Multiplexação no transm.:

reúne dados de muitos sockets, envelope os dados com o cabeçalho (usado posteriormente para a demultiplexação)



# Como funciona a demultiplexação

- host recebe os datagramas IP
  - cada datagrama possui os endereços IP da origem e do destino
  - cada datagrama transporta 1 segmento da camada de transporte
  - cada segmento possui números das portas origem e destino (lembre: números de portas bem conhecidas para aplicações específicas)
- host usa os endereços IP e os números das portas para direcionar o segmento ao socket apropriado



formato de segmento  
TCP/UDP



# Demultiplexação sem Conexões

- Crie sockets com números de porta:

```
DatagramSocket mySocket1 =  
    new DatagramSocket(99111);  
DatagramSocket mySocket2 =  
    new DatagramSocket(99222);
```

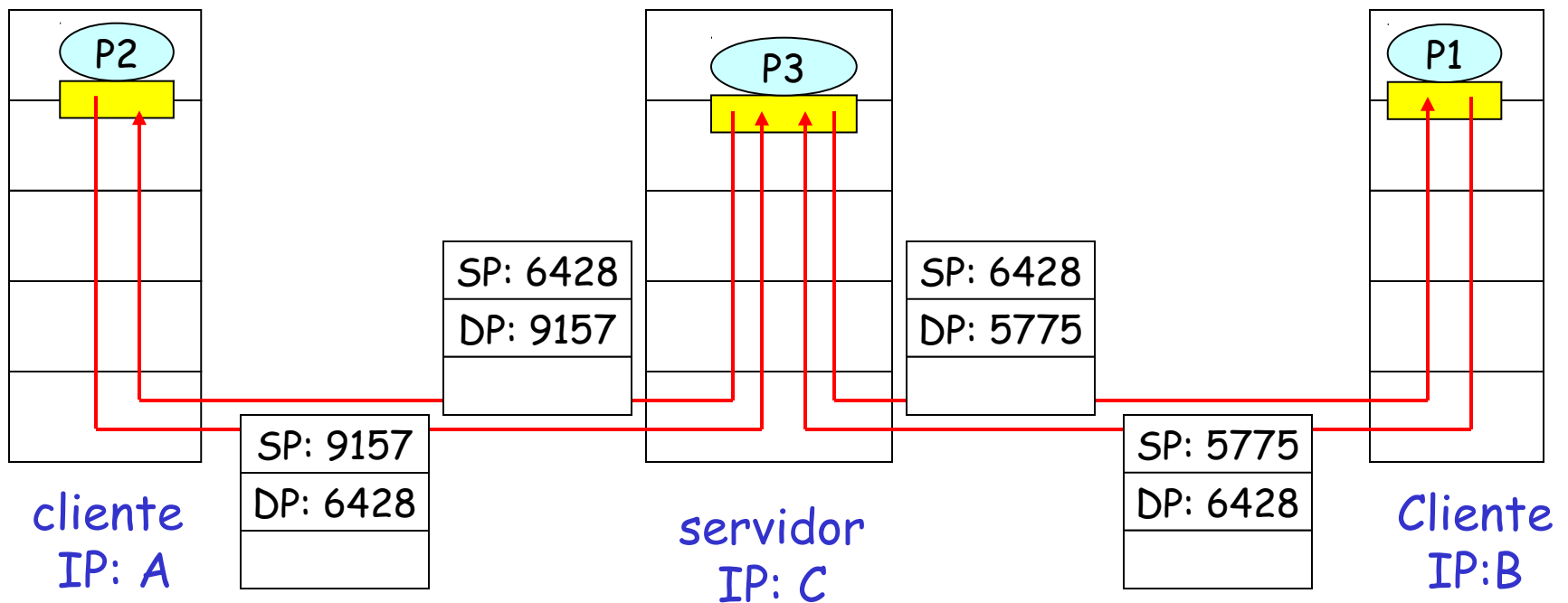
- socket UDP identificado pela dupla:

(end IP dest, no. da porta destino)

- Quando host recebe segmento UDP:
  - verifica no. da porta de destino no segmento
  - encaminha o segmento UDP para o socket com aquele no. de porta
- Datagramas IP com diferentes endereços IP origem e/ou números de porta origem são encaminhados para o mesmo socket

# Demultiplexação sem Conexões (cont)

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

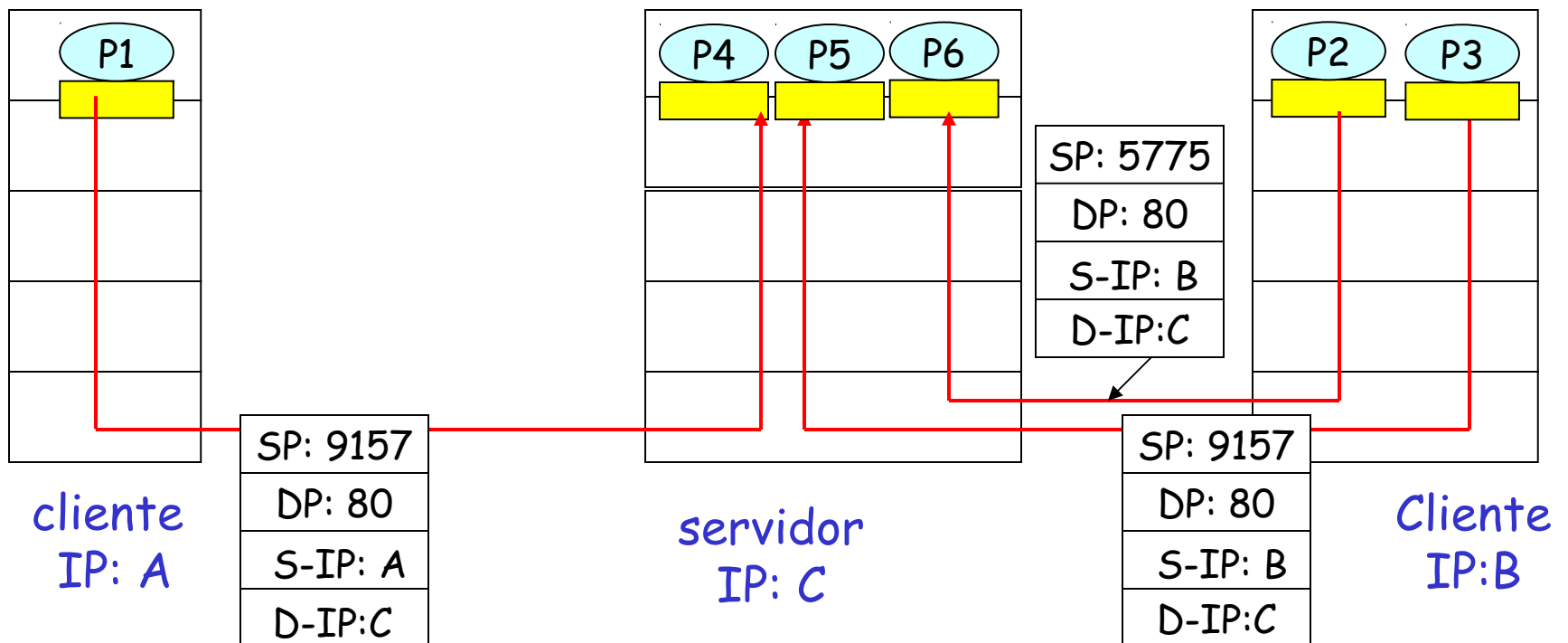


SP (*source port*) provê "endereço de retorno"

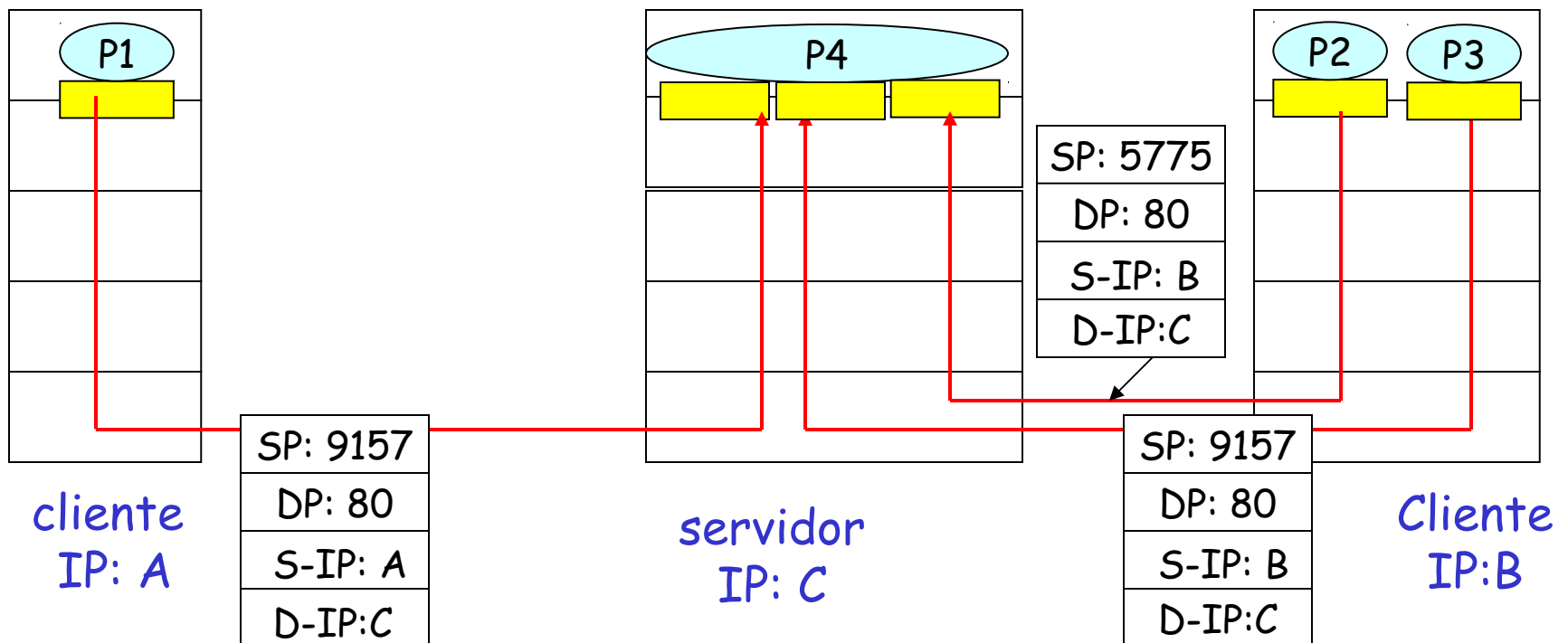
# Demultiplexação Orientada a Conexões

- ▮ Socket TCP identificado pela 4-dupla:
  - ▮ endereço IP origem
  - ▮ número da porta origem
  - ▮ endereço IP destino
  - ▮ número da porta destino
- ▮ receptor usa todos os quatro valores para direcionar o segmento para o socket apropriado
- ▮ Servidor pode dar suporte a muitos sockets TCP simultâneos:
  - ▮ cada socket é identificado pela sua própria 4-dupla
- ▮ Servidores Web têm sockets diferentes para cada conexão cliente
  - ▮ HTTP não persistente terá sockets diferentes para cada pedido

# Demultiplexação Orientada a Conexões (cont)



# Demultiplexação Orientada a Conexões: Servidor Web com Threads



# Conteúdo do Capítulo 3

- ▢ 3.1 Serviços da camada de transporte
- ▢ 3.2 Multiplexação e demultiplexação
- ▢ 3.3 UDP: Transporte não orientado a conexão
- ▢ 3.4 Princípios da transferência confiável de dados
- ▢ 3.5 Transporte orientado a conexão: TCP
  - ▢ transferência confiável
  - ▢ controle de fluxo
  - ▢ gerenciamento de conexões
- ▢ 3.6 Princípios de controle de congestionamento
- ▢ 3.7 Controle de congestionamento do TCP

# UDP: User Datagram Protocol [RFC 768]

- ▮ Protocolo de transporte da Internet mínimo, "sem frescura",
- ▮ Serviço "melhor esforço", segmentos UDP podem ser:
  - ▮ perdidos
  - ▮ entregues à aplicação fora de ordem do remesso
- ▮ *sem conexão:*
  - ▮ não há "setup" UDP entre remetente, receptor
  - ▮ tratamento independente de cada segmento UDP

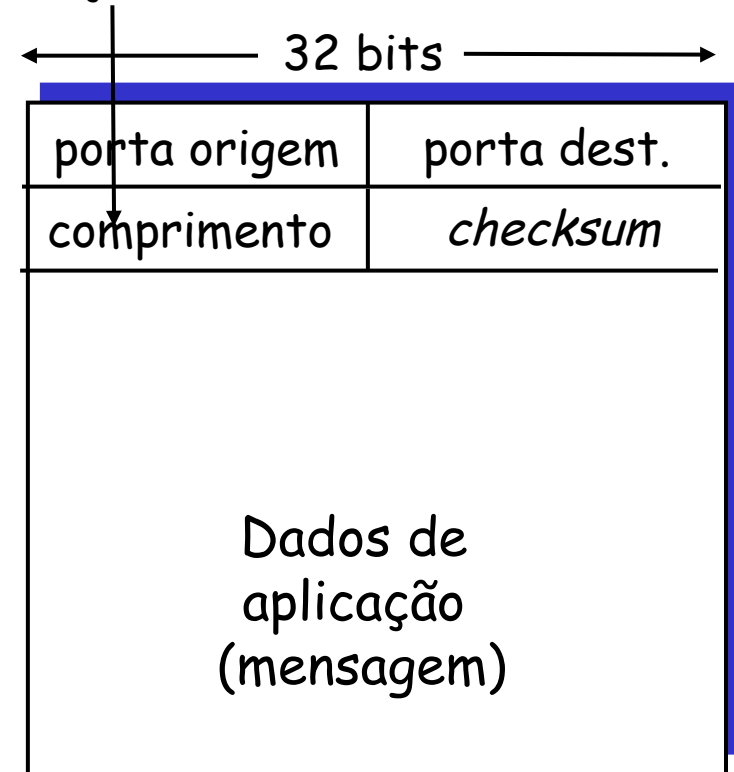
## Por quê existe um UDP?

- ▮ elimina estabelecimento de conexão (o que pode causar retardo)
- ▮ simples: não se mantém "estado" da conexão no remetente/receptor
- ▮ pequeno cabeçalho de segmento
- ▮ sem controle de congestionamento: UDP pode transmitir o mais rápido possível

# Mais sobre UDP

- muito utilizado para apls. de meios contínuos (voz, vídeo)
  - tolerantes de perdas
  - sensíveis à taxa de transmissão
- outros usos de UDP (por quê?):
  - DNS (nomes)
  - SNMP (gerenciamento)
- transferência confiável com UDP: incluir confiabilidade na camada de aplicação
  - recuperação de erro específica à apl.!

Comprimento em bytes do segmento UDP, incluindo cabeçalho



Formato do segmento UDP



# Checksum UDP

Meta: detectar "erro" (e.g., bits invertidos) no segmento transmitido

## Remetente:

- ▢ trata conteúdo do segmento como sequência de inteiros de 16-bits
- ▢ campo checksum zerado
- ▢ checksum: soma (adição usando complemento de 1) do conteúdo do segmento
- ▢ remetente coloca *complemento do valor da soma* no campo checksum de UDP

## Receptor:

- ▢ calcula checksum do segmento recebido
- ▢ verifica se checksum computado é zero:
  - ▢ NÃO - erro detectado
  - ▢ SIM - nenhum erro detectado. *Mas ainda pode ter erros? Veja depois ....*

# Exemplo do Checksum Internet

## □ Note

- Ao adicionar números, o transbordo do bit mais significativo deve ser adicionado o resultado
- Exemplo: adição de dois inteiros de 16-bits

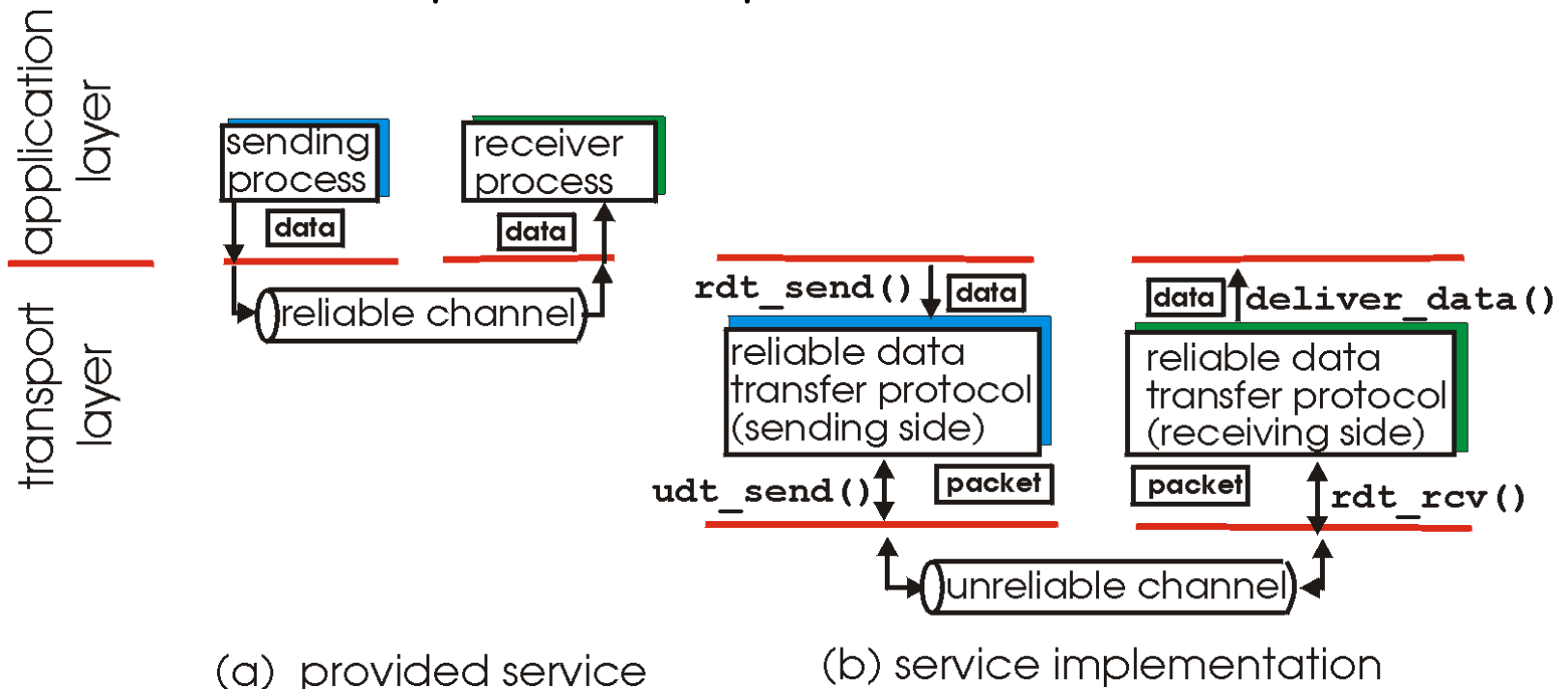
	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																
transbordo	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
	<hr/>															
soma	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

# Conteúdo do Capítulo 3

- 3.1 Serviços da camada de transporte
- 3.2 Multiplexação e demultiplexação
- 3.3 UDP: Transporte não orientado a conexão
- 3.4 Princípios da transferência confiável de dados
- 3.5 Transporte orientado a conexão: TCP
  - transferência confiável
  - controle de fluxo
  - gerenciamento de conexões
- 3.6 Princípios de controle de congestionamento
- 3.7 Controle de congestionamento do TCP

# Princípios de Transferência confiável de dados (rdt)

- importante nas camadas de transporte, enlace
- na lista dos 10 tópicos mais importantes em redes!

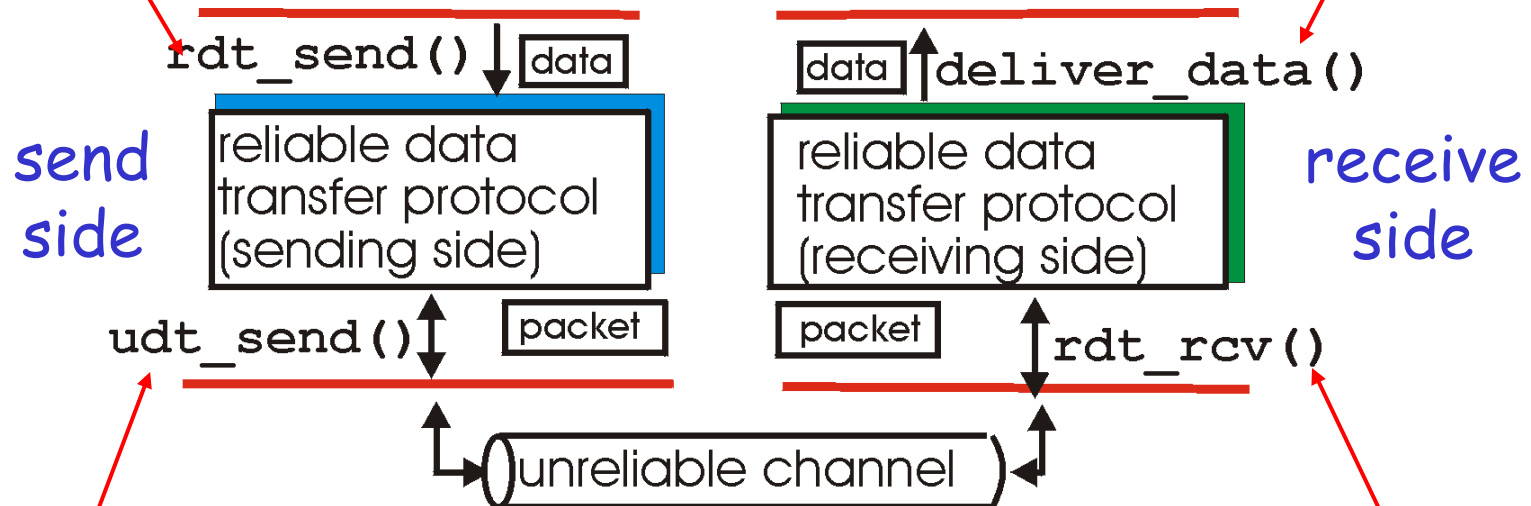


- características do canal não confiável determinam a complexidade de um protocolo de transferência confiável de dados (rdt)

# Transferência confiável de dados (rdt): como começar

**rdt\_send()** : chamada de cima, (p.ex., pela apl.). Dados recebidos p/ entregar à camada sup. do receptor

**deliver\_data()** : chamada por rdt p/ entregar dados p/ camada superior



**udt\_send()** : chamada por rdt, p/ transferir pacote pelo canal ã confiável ao receptor

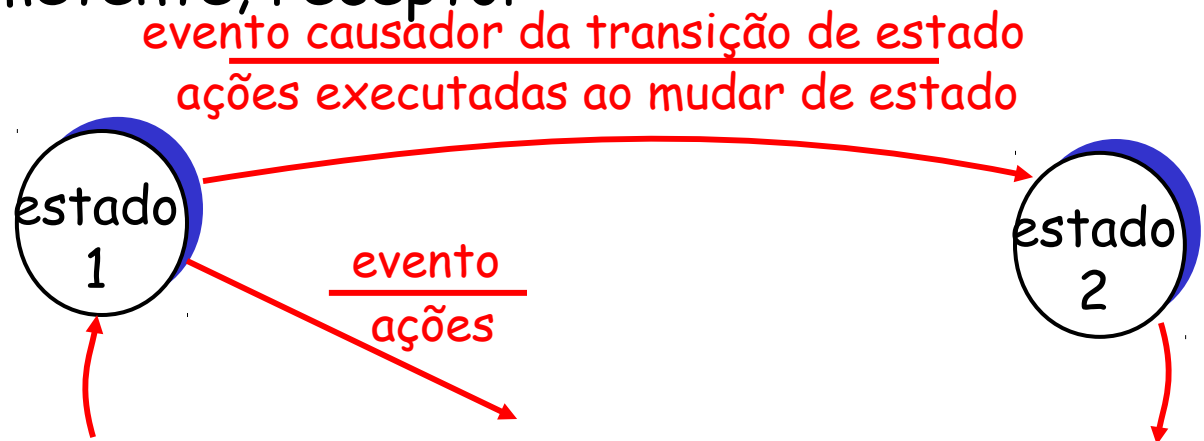
**rdt\_rcv()** : chamada quando pacote chega no lado receptor do canal

# Transferência confiável de dados (rdt): como começar

## Iremos:

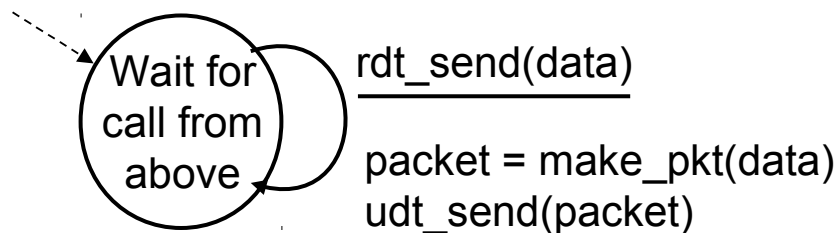
- ▮ desenvolver incrementalmente os lados remetente, receptor do protocolo RDT
- ▮ considerar apenas fluxo unidirecional de dados
  - ▮ mas info de controle flui em ambos os sentidos!
- ▮ Usar máquinas de estados finitos (FSM) p/ especificar remetente, receptor

**estado:** neste "estado" o próximo estado é determinado unicamente pelo próximo evento

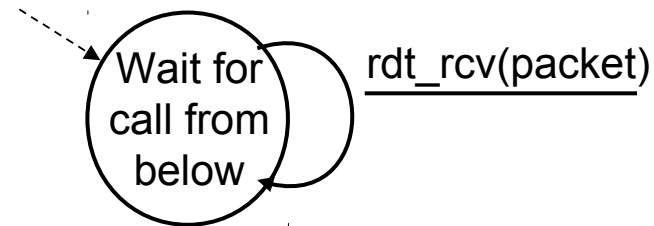


# Rdt1.0: transferência confiável usando um canal confiável

- canal subjacente perfeitamente confiável
  - não tem erros de bits
  - não tem perda de pacotes
- FSMs separadas para remetente e receptor:
  - remetente envia dados pelo canal subjacente
  - receptor recebe dados do canal subjacente



transmissor



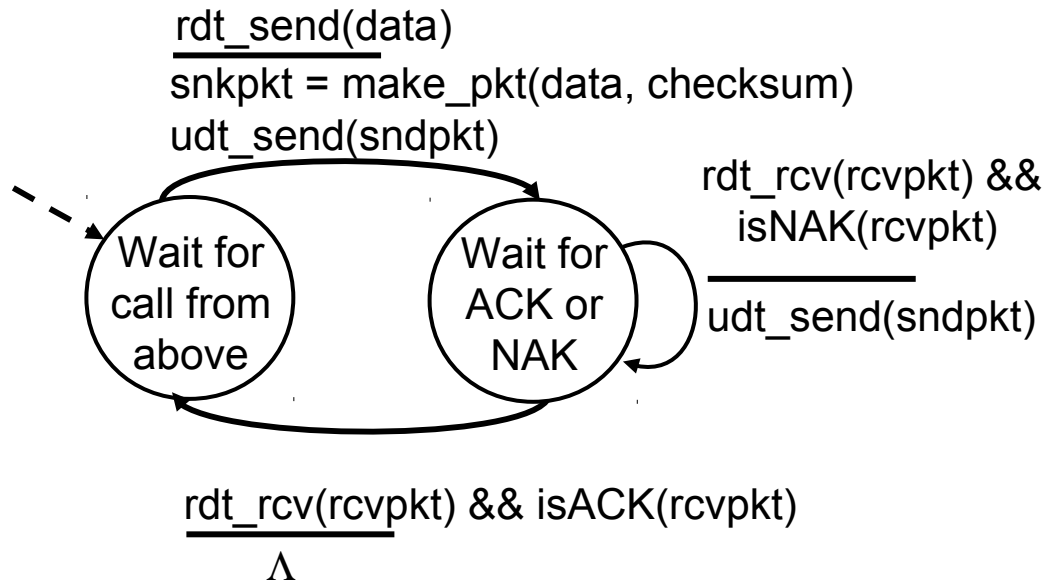
receptor

# Rdt2.0: canal com erros de bits

- canal subjacente pode inverter bits no pacote
  - lembre-se: checksum UDP pode detectar erros de bits
- a questão: como recuperar dos erros?
  - *reconhecimentos (ACKs)*: receptor avisa explicitamente ao remetente que pacote chegou bem
  - *reconhecimentos negativos (NAKs)*: receptor avisa explicitamente ao remetente que pacote tinha erros
  - remetente retransmite pacote ao receber um NAK
  - cenários humanos usando ACKs, NAKs?
- novos mecanismos em rdt2.0 (em relação ao rdt1.0):
  - detecção de erros
  - realimentação pelo receptor: msgs de controle (ACK,NAK) receptor→remetente

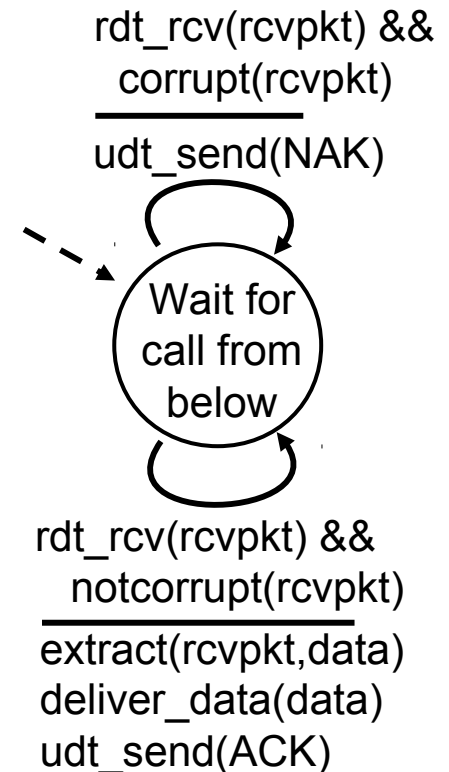


# rdt2.0: especificação da FSM

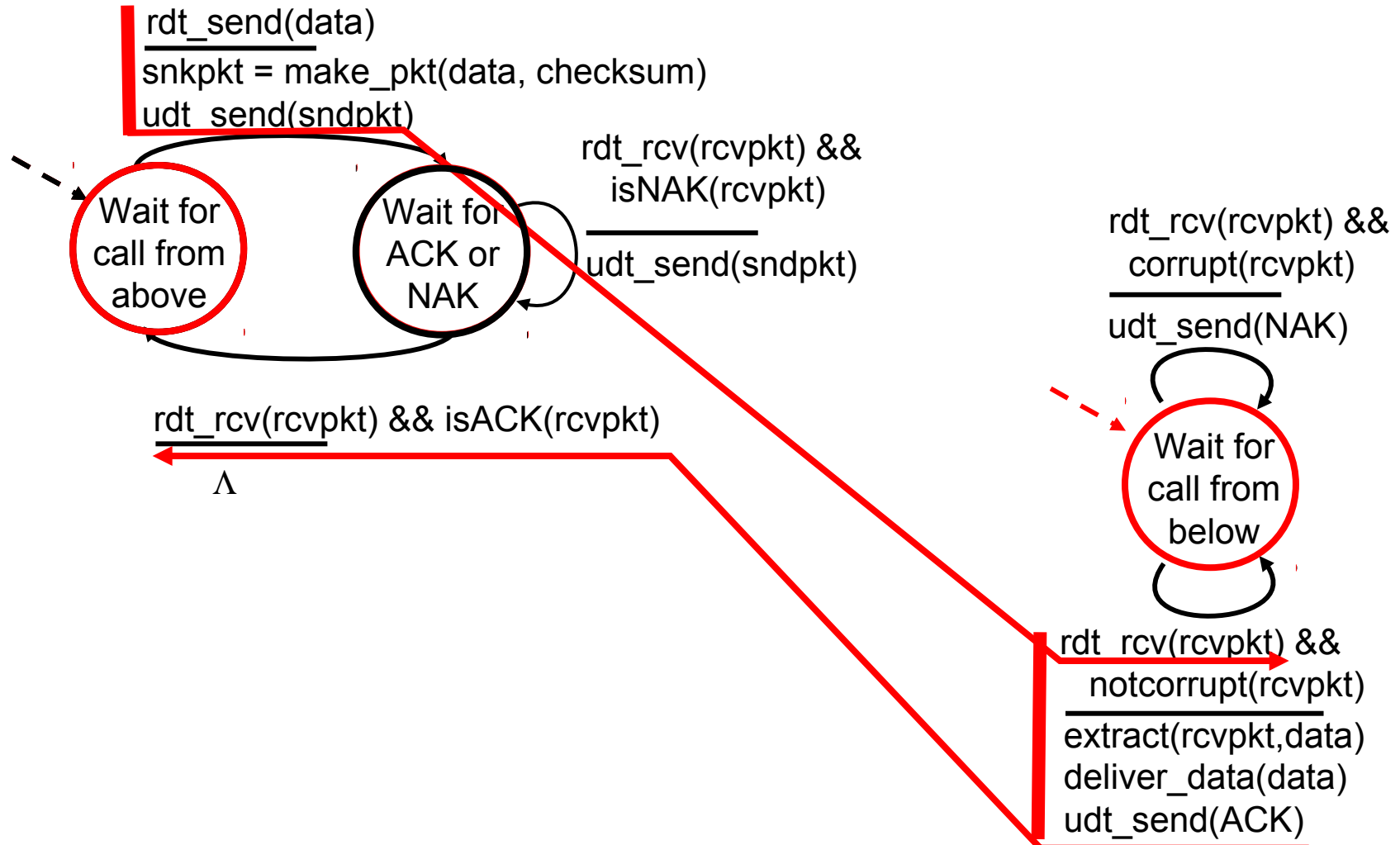


transmissor

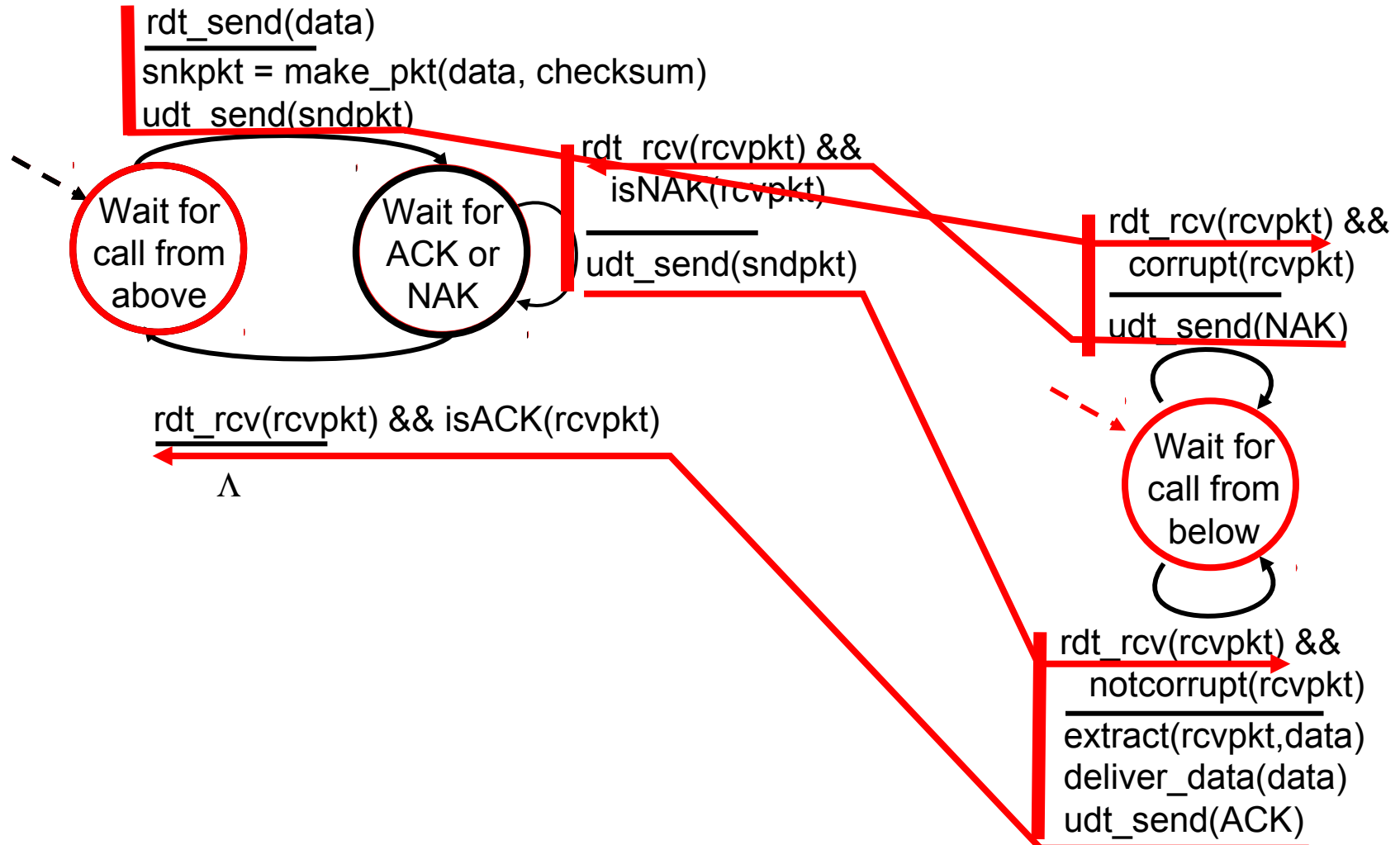
receptor



# rdt2.0: operação sem erros



# rdt2.0: cenário com erros



# rdt2.0 tem uma falha fatal!

## O que acontece se ACK/NAK com erro?

- ▮ Remetente não sabe o que se passou no receptor!
- ▮ não se pode apenas retransmitir: possibilidade de pacotes duplicados

## O que fazer?

- ▮ remetente usa ACKs/NAKs p/ ACK/NAK do receptor? E se perder ACK/NAK do remetente?
- ▮ retransmitir, mas pode causar retransmissão de pacote recebido certo!

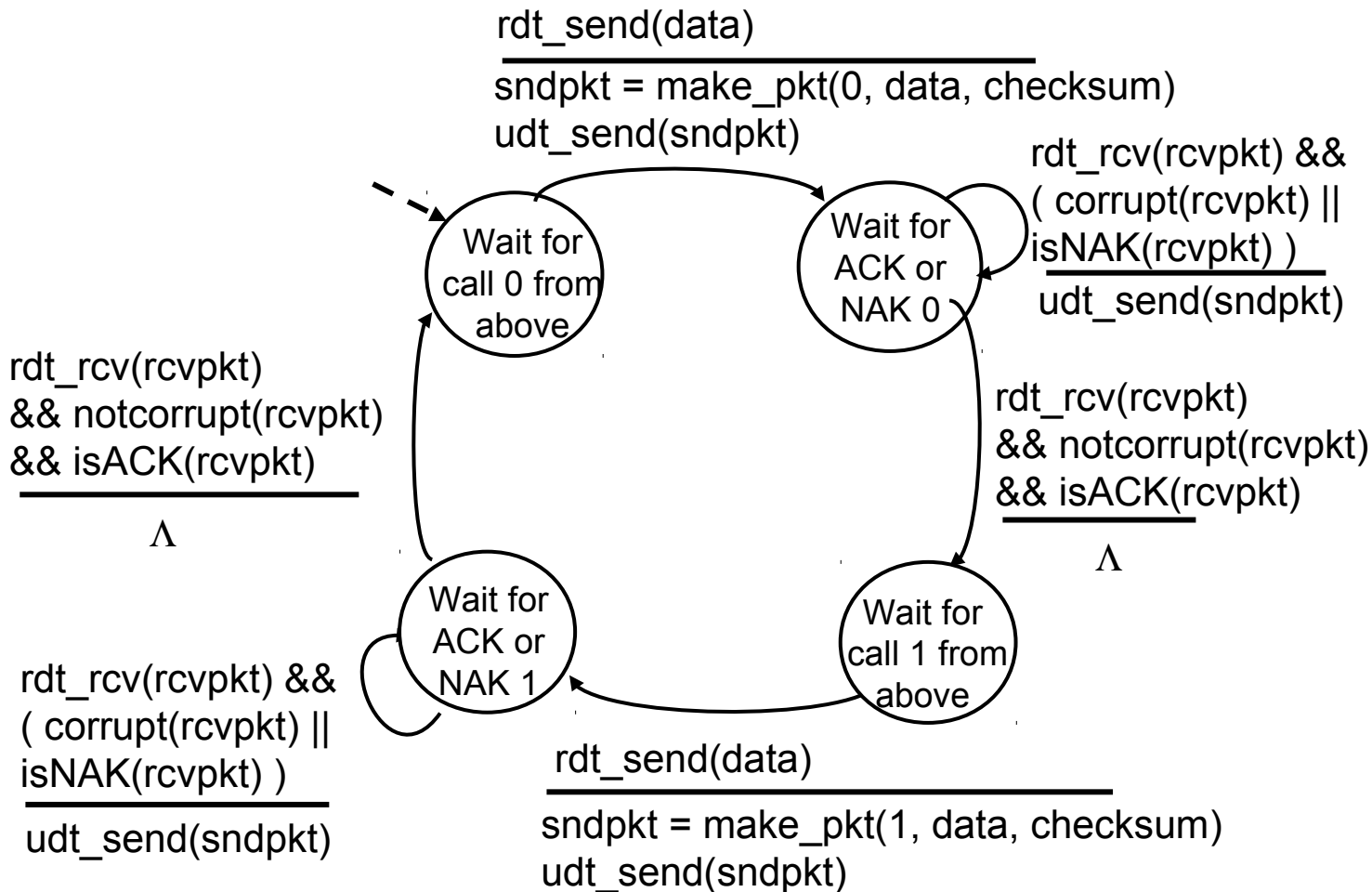
## Lidando c/ duplicação:

- ▮ remetente inclui *número de sequência* p/ cada pacote
- ▮ remetente retransmite pacote atual se ACK/NAK recebido com erro
- ▮ receptor descarta (não entrega) pacote duplicado

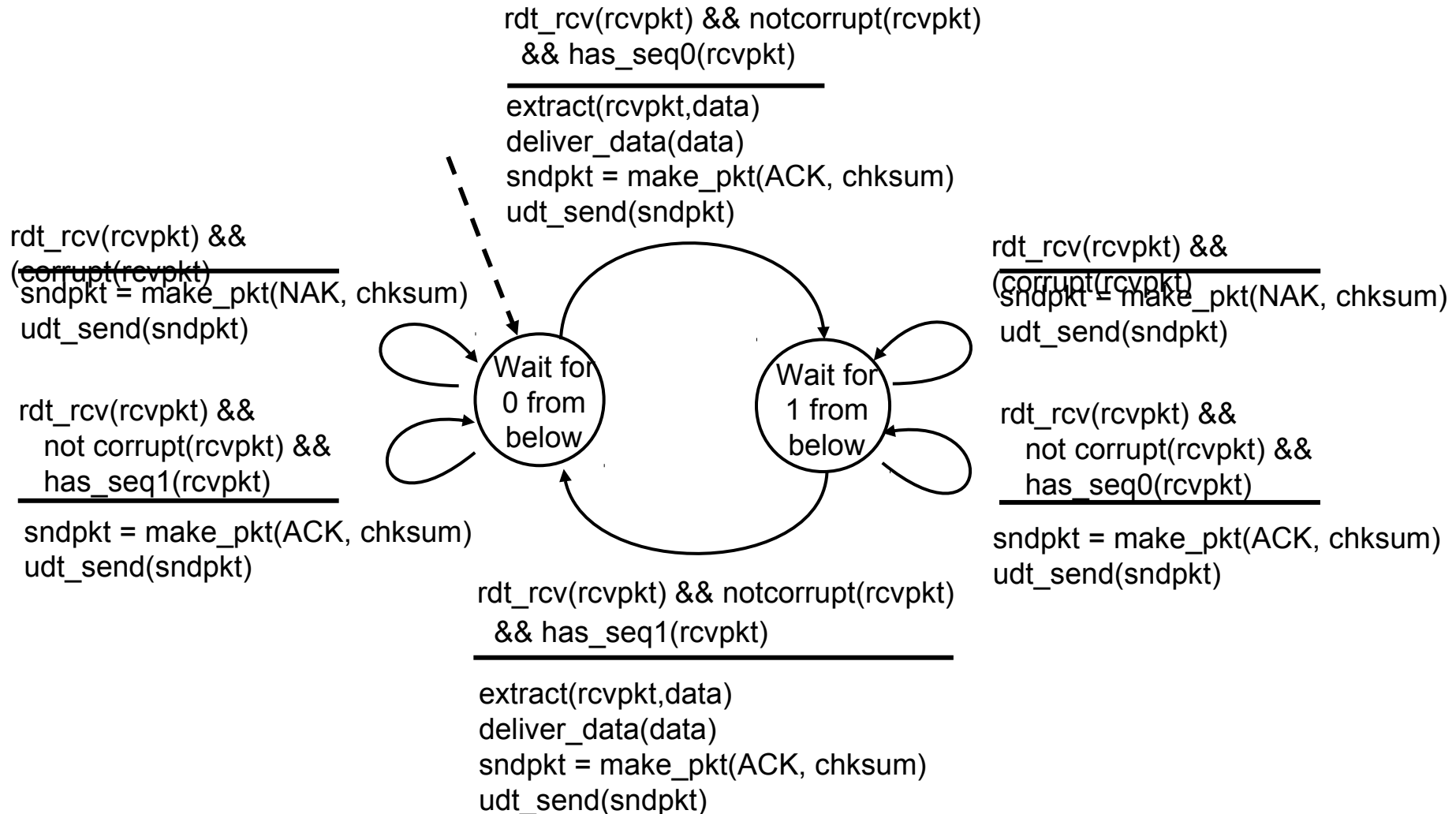
### pára e espera

Remetente envia um pacote, e então aguarda resposta do receptor

# rdt2.1: remetente, trata ACK/NAKs c/ erro



# rdt2.1: receptor, trata ACK/NAKs com erro



# rdt2.1: discussão

## Remetente:

- ▢ no. de seq no pacote
- ▢ bastam dois nos. de seq. (0,1). Por quê?
- ▢ deve checar se ACK/NAK recebido tinha erro
- ▢ duplicou o no. de estados
  - ▢ estado deve "lembrar" se pacote "corrente" tem no. de seq. 0 ou 1

## Receptor:

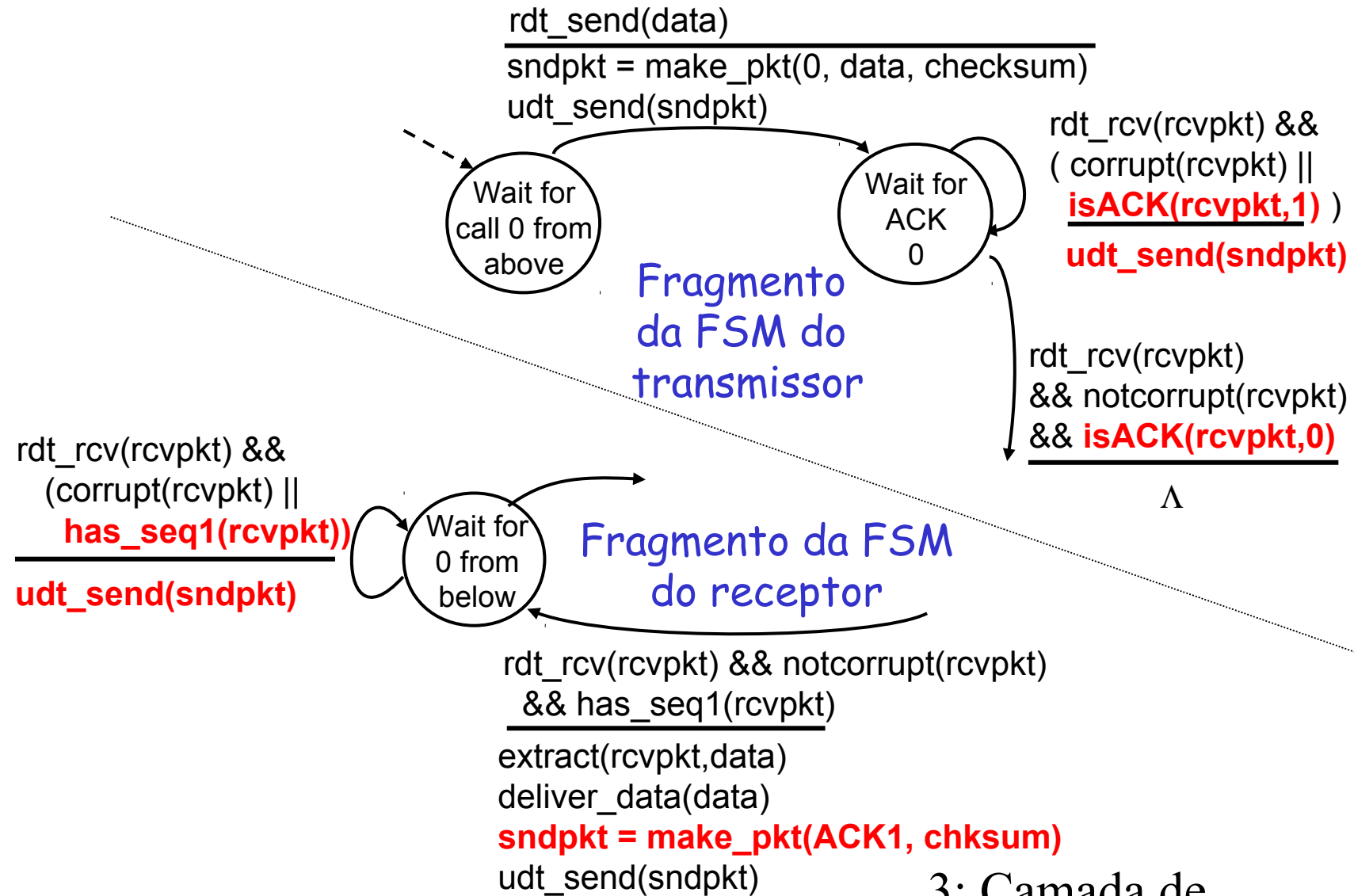
- ▢ deve checar se pacote recebido é duplicado
  - ▢ estado indica se no. de seq. esperado é 0 ou 1
- ▢ note: receptor não tem como saber se último ACK/NAK foi recebido bem pelo remetente

## rdt2.2: um protocolo sem NAKs

- mesma funcionalidade que rdt2.1, só com ACKs
- ao invés de NAK, receptor envia ACK p/ último pacote recebido bem
  - receptor deve incluir *explicitamente* no. de seq do pacote reconhecido
- ACK duplicado no remetente resulta na mesma ação que o NAK: *retransmite pacote atual*



# rdt2.2: fragmentos do transmissor e receptor



# rdt3.0: canais com erros e perdas

Nova suposição: canal subjacente também pode perder pacotes (dados ou ACKs)

- ▮ checksum, no. de seq., ACKs, retransmissões podem ajudar, mas não serão suficientes

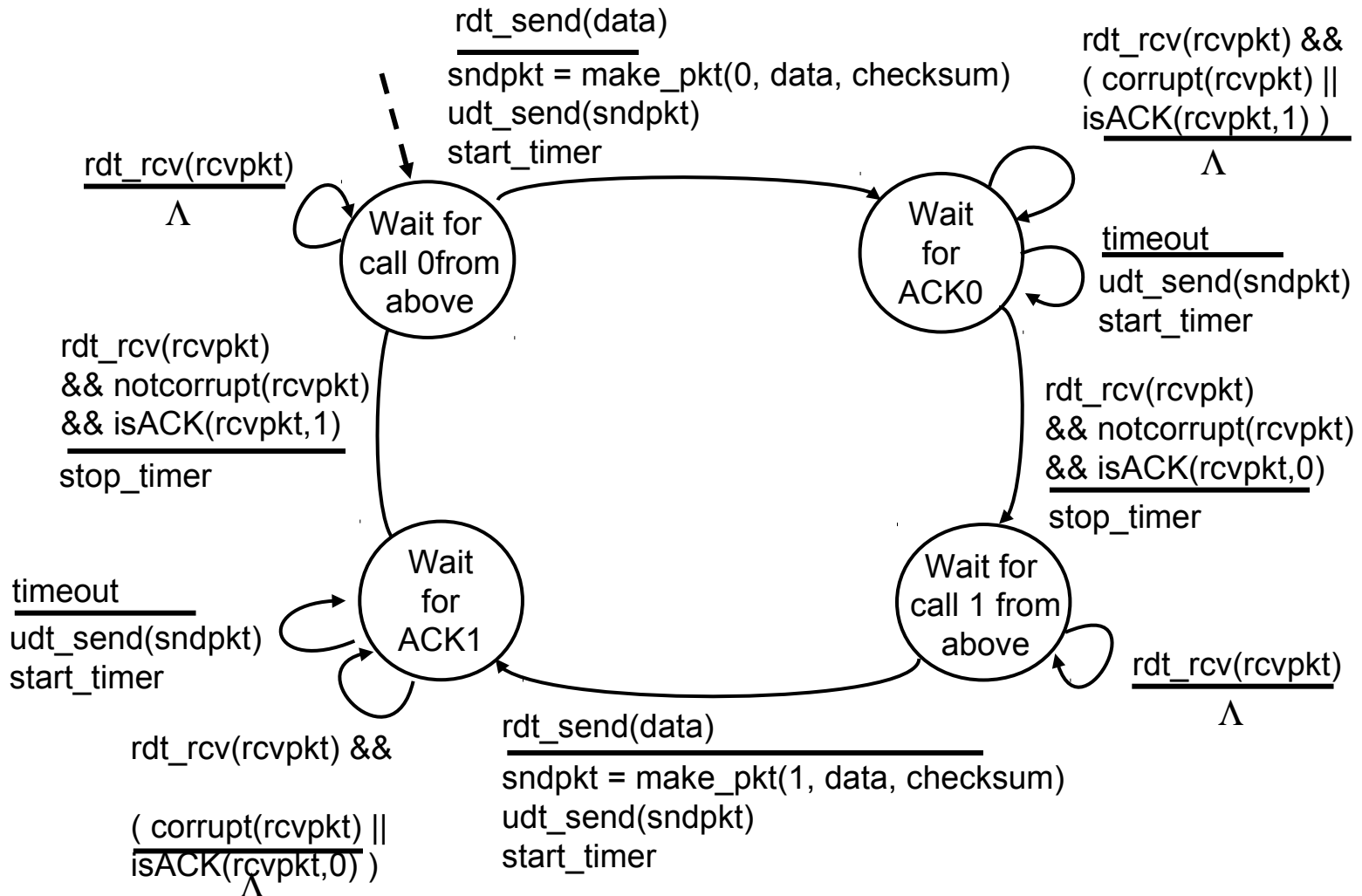
P: como lidar com perdas?

- ▮ remetente espera até ter certeza que se perdeu pacote ou ACK, e então retransmite
- ▮ eca!: desvantagens?

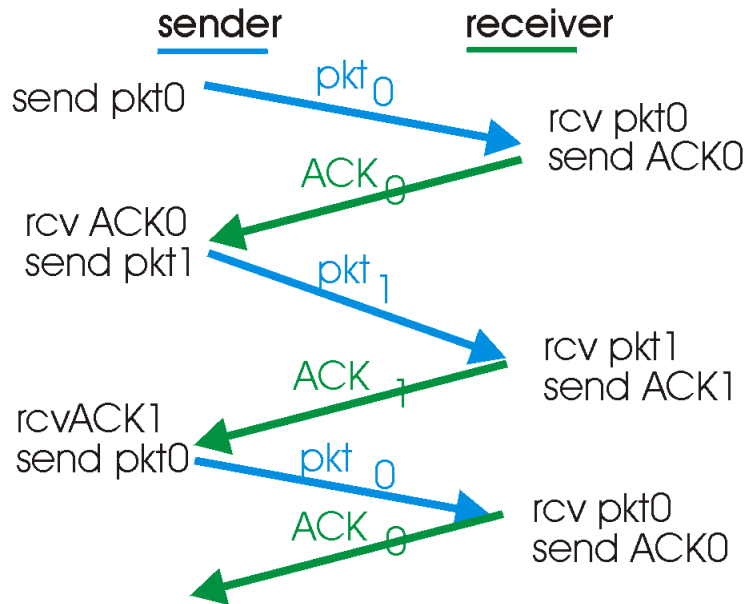
Abordagem: remetente aguarda um tempo "razoável" pelo ACK

- ▮ retransmite se nenhum ACK for recebido neste intervalo
- ▮ se pacote (ou ACK) apenas atrasado (e não perdido):
  - ▮ retransmissão será duplicada, mas uso de no. de seq. já cuida disto
  - ▮ receptor deve especificar no. de seq do pacote sendo reconhecido
- ▮ requer temporizador

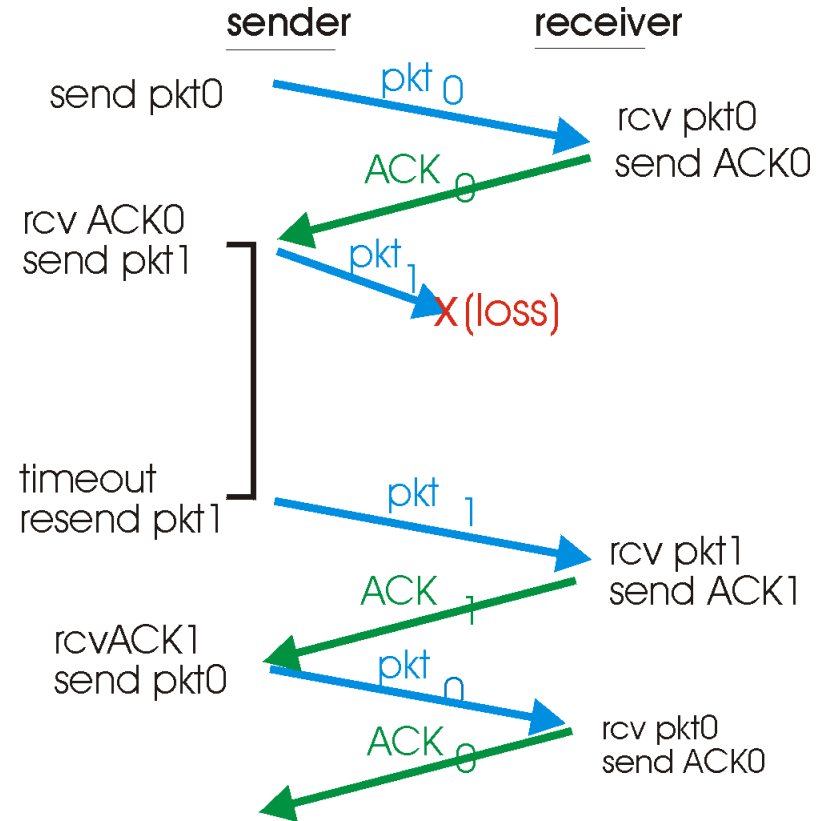
# rdt3.0: remetente



# rdt3.0 em ação

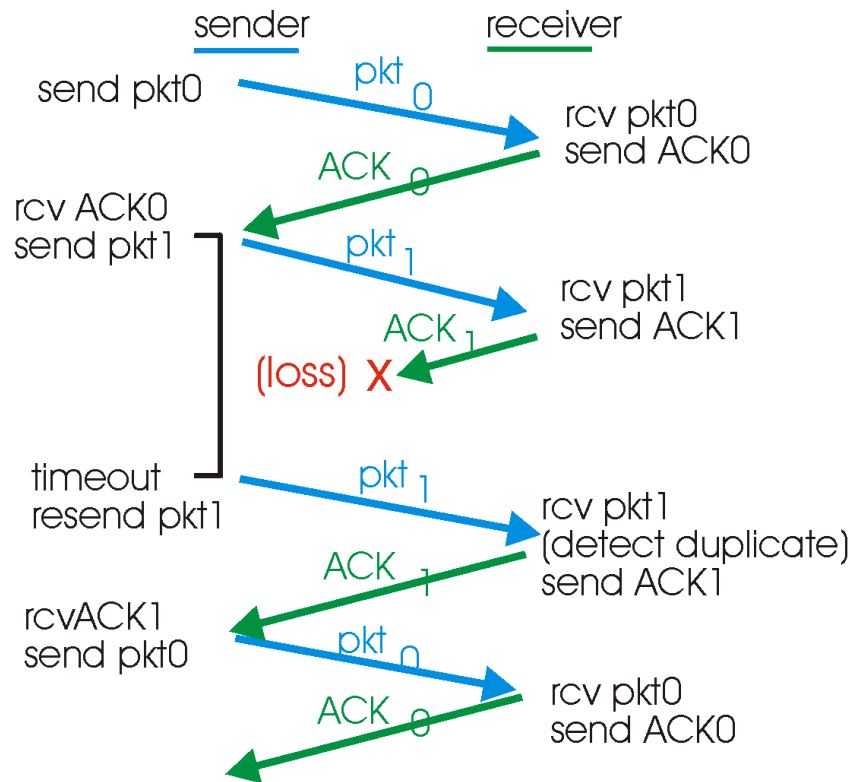


(a) operation with no loss

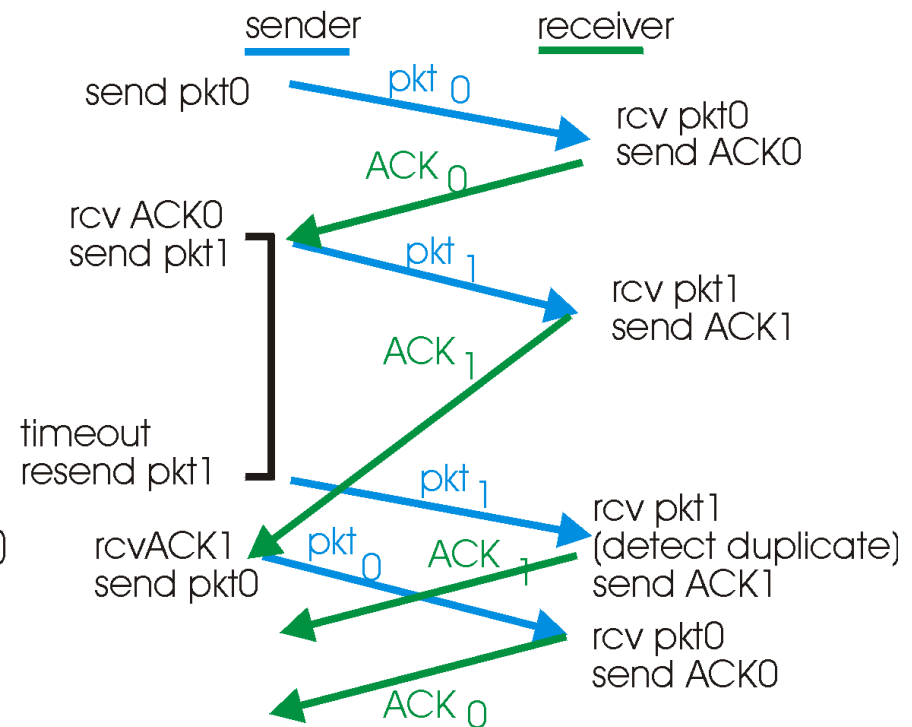


(b) lost packet

# rdt3.0 em ação



(c) lost ACK



(d) premature timeout

# Desempenho de rdt3.0

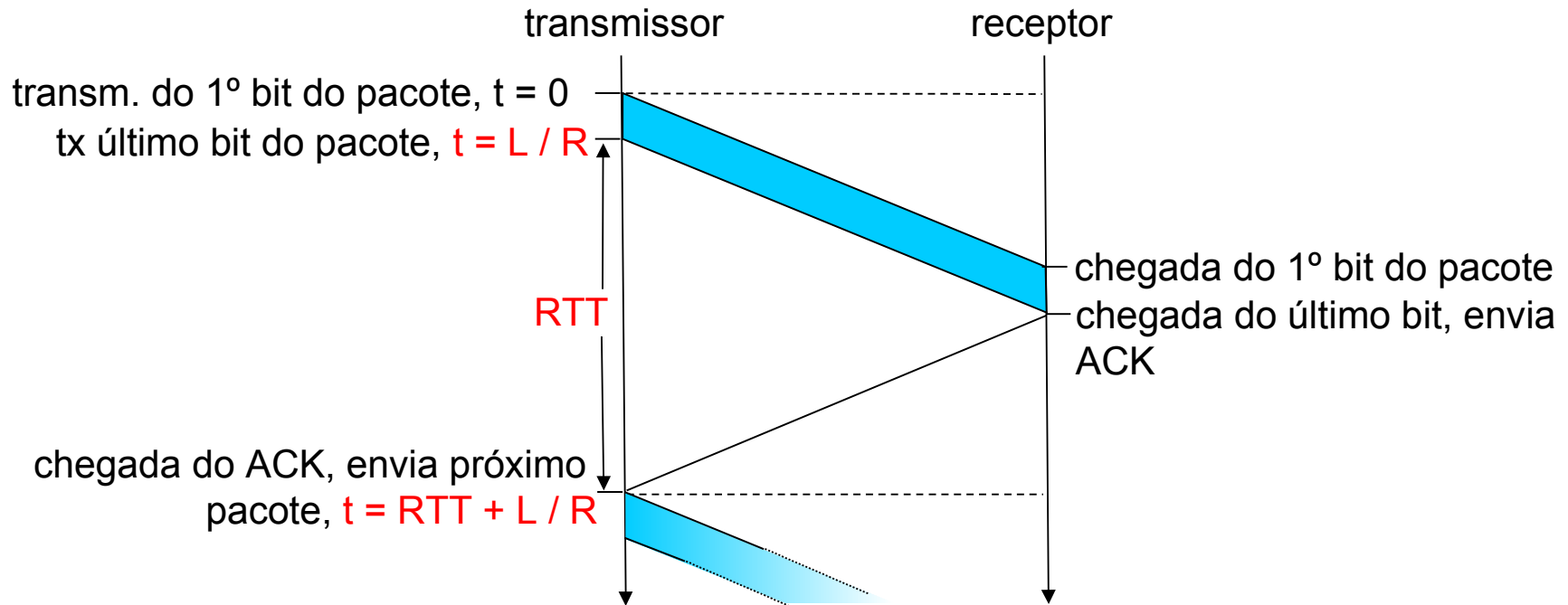
- rdt3.0 funciona, porém seu desempenho é muito ruim
- exemplo: enlace de 1 Gbps, retardo fim a fim de 15 ms, pacote de 1KB:

$$T_{\text{transmitir}} = \frac{8\text{kb/pacote}}{10^{**9} \text{ b/seg}} = 8 \text{ microseg}$$

$$U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

- pac. de 1KB a cada 30 mseg -> vazão de 33kB/seg num enlace de 1 Gbps
- protocolo limita uso dos recursos físicos!

# rdt3.0: stop-and-wait operation

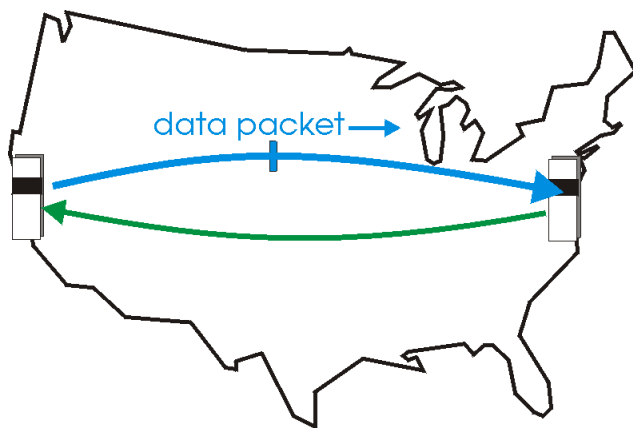


$$U_{tx} = \frac{L/R}{RTT + L/R} = \frac{0,008}{30,008} = 0,00027$$

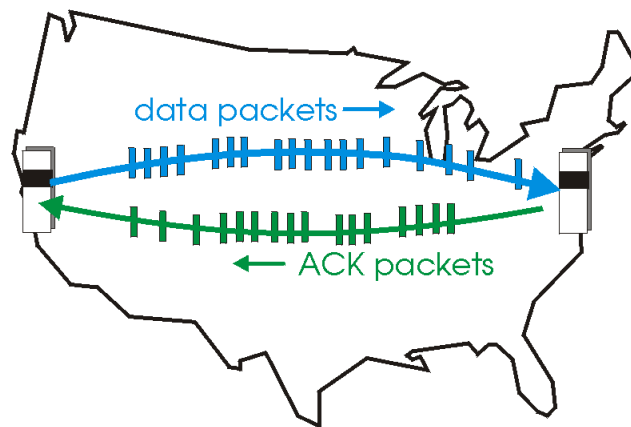
# Protocolos "com paralelismo" (pipelined)

**Paralelismo (pipelining):** remetente admite múltiplos pacotes "em trânsito", ainda não reconhecidos

- faixa de números de seqüência deve ser aumentada
- buffers no remetente e/ou no receptor



(a) a stop-and-wait protocol in operation

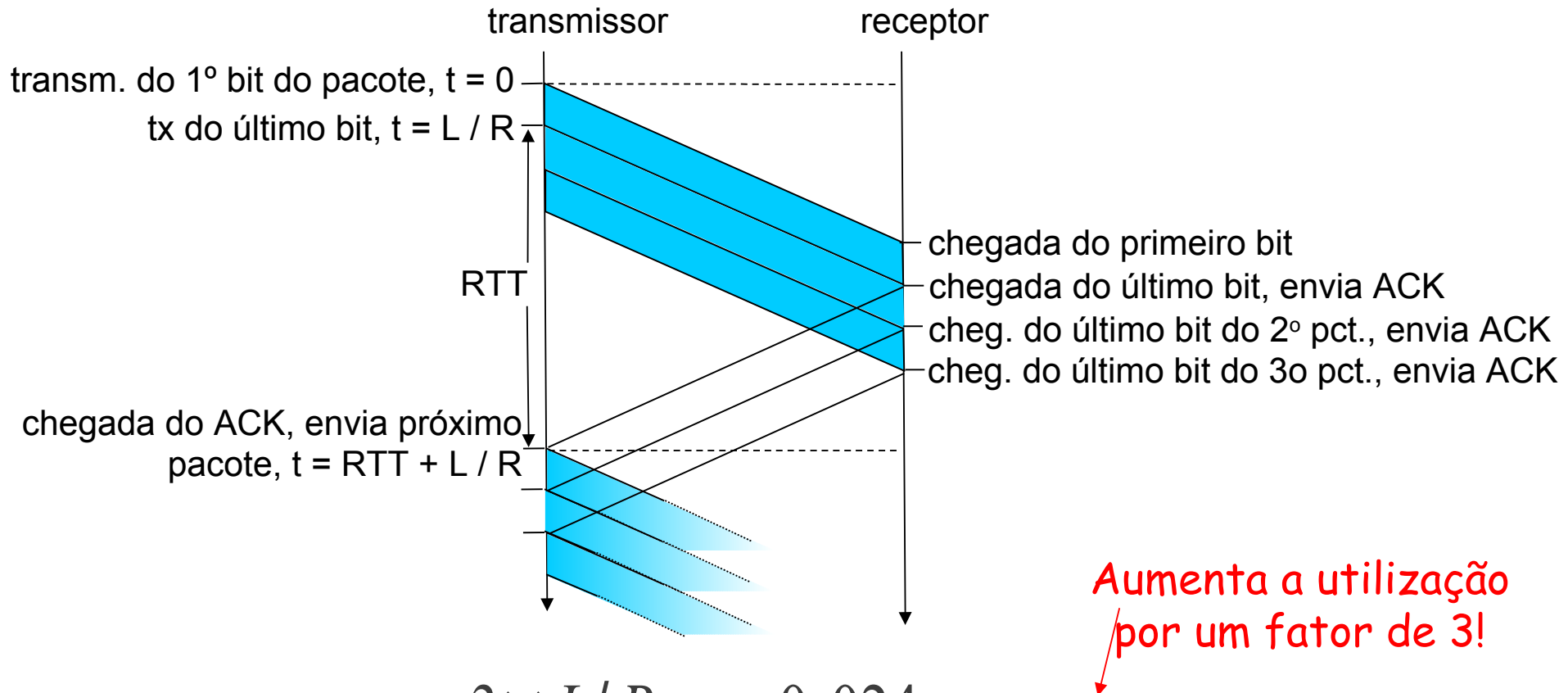


(b) a pipelined protocol in operation

- Duas formas genéricas de protocolos com paralelismo:  
*Go-back-N, retransmissão seletiva*



# Paralelismo: maior utilização

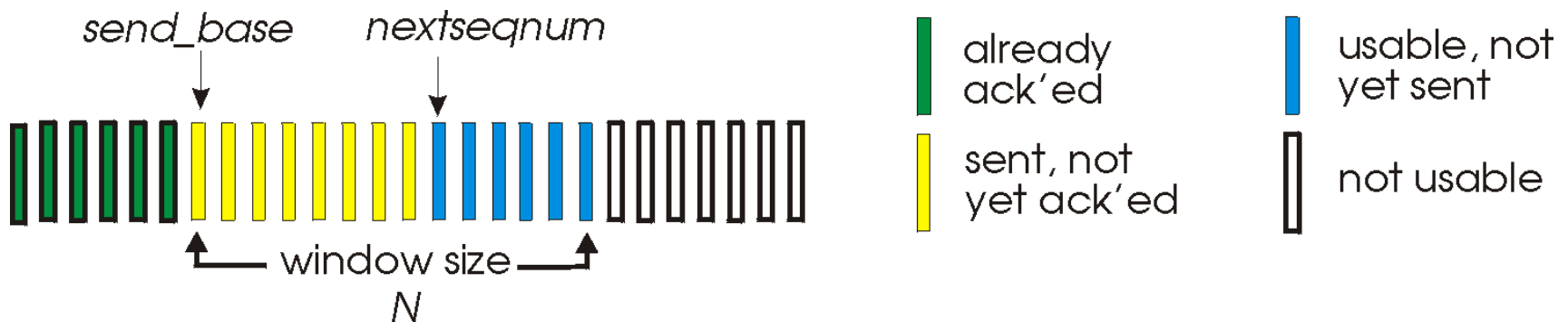


$$U_{tx} = \frac{3 \times L/R}{RTT + L/R} = \frac{0,024}{30,008} = 0,0008$$

# Go-back-N (GBN)

## Remetente:

- no. de seq. de k-bits no cabeçalho do pacote
- admite "janela" de até N pacotes consecutivos não reconhecidos



- ACK(n): reconhece todos pacotes, até e inclusive no. de seq n - "ACK cumulativo"
  - pode receber ACKs duplicados (veja receptor)
- temporizador para cada pacote em trânsito
- *timeout(n)*: retransmite pacote n e todos os pacotes com no. de seq maiores na janela

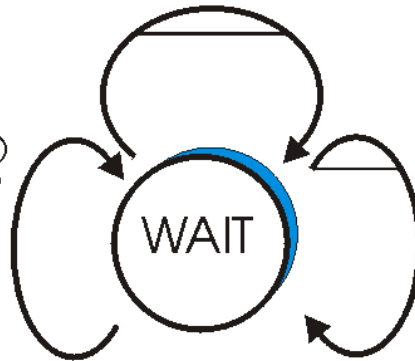
# GBN: FSM estendida do remetente

rdt\_send(data)

```
if (nextseqnum < base+N) {  
    compute chksum  
    make_pkt(sndpkt(nextseqnum)),nextseqnum,data,chksum)  
    udt_send(sndpkt(nextseqnum))  
    if (base == nextseqnum)  
        start_timer  
    nextseqnum = nextseqnum + 1  
}  
else  
    refuse_data(data)
```

rdt\_rcv(rcv\_pkt) && notcorrupt(rcvpkt)

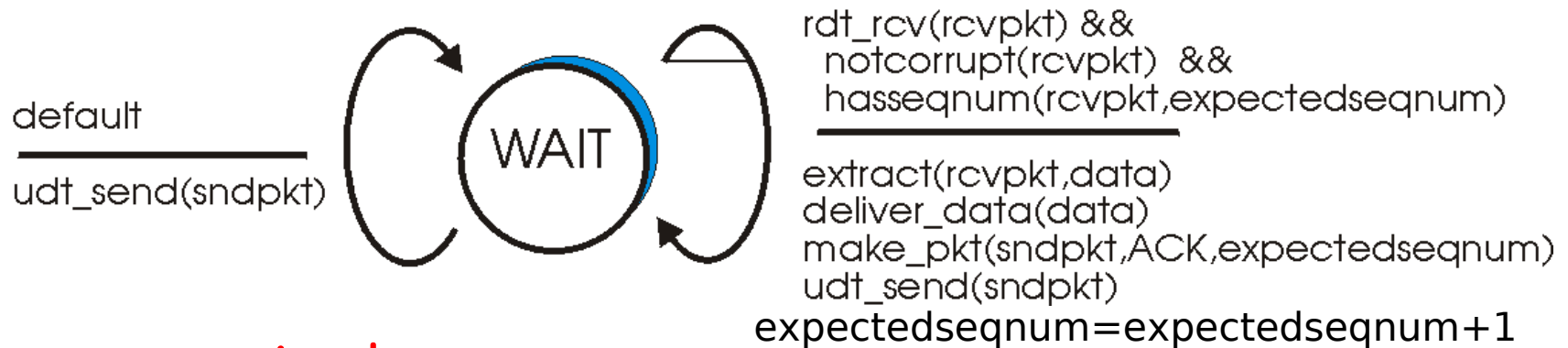
```
base = getacknum(rcvpkt)+1  
if (base == nextseqnum)  
    stop_timer  
else  
    start_timer
```



timeout

```
start_timer  
udt_send(sndpkt(base))  
udt_send(sndpkt(base+1))  
.....  
udt_send(sndpkt(nextseqnum-1))
```

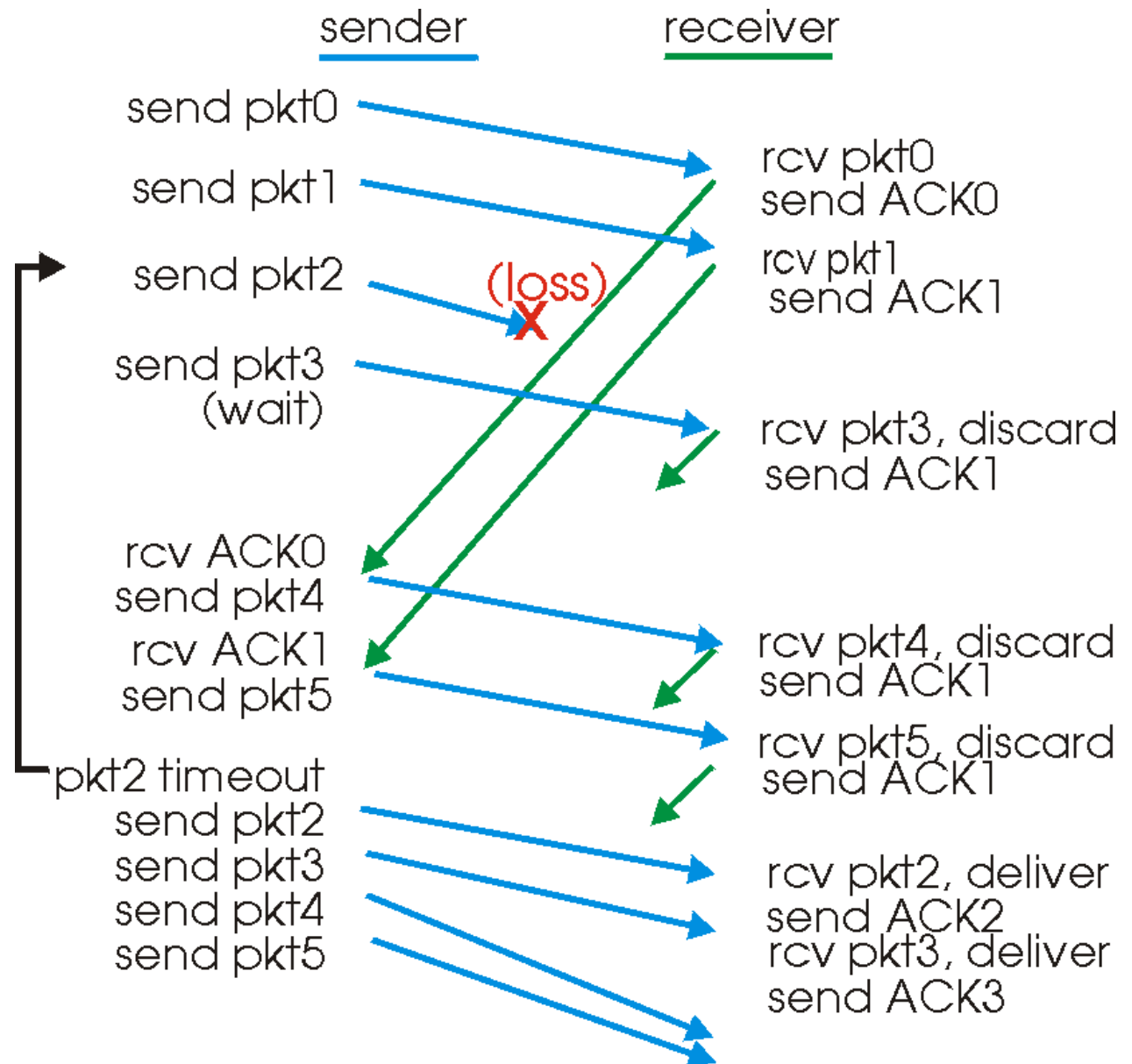
# GBN: FSM estendida do receptor



## receptor simples:

- usa apenas ACK: sempre envia ACK para pacote recebido bem com o maior no. de seq. *em-ordem*
  - pode gerar ACKs duplicados
  - só precisa se lembrar do **expectedseqnum**
- pacote fora de ordem:
  - descarta (não armazena) -> **receptor não usa buffers!**
  - manda ACK de pacote com maior no. de seq em-ordem

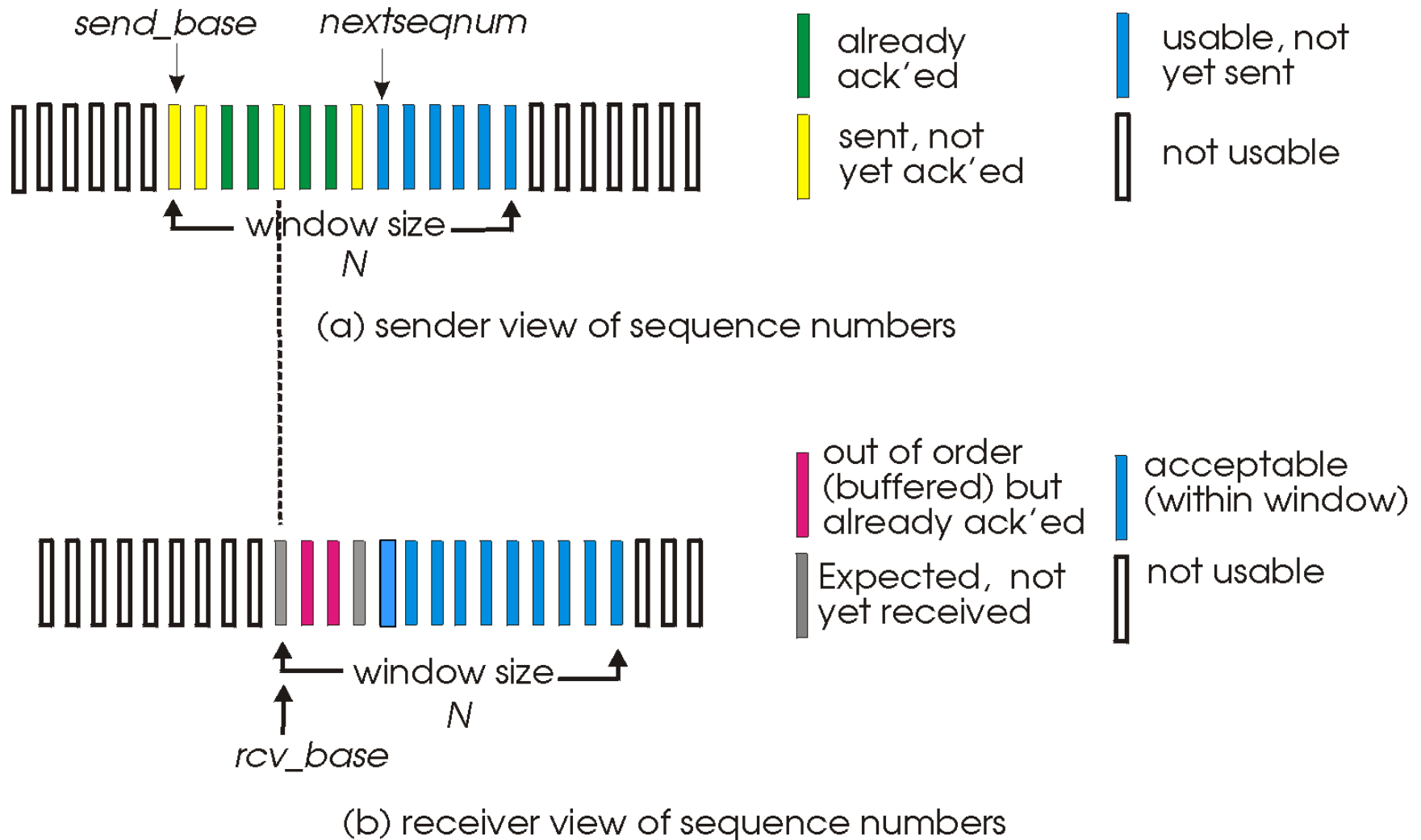
# GBN em ação



# Retransmissão seletiva

- ▢ receptor reconhece *individualmente* todos os pacotes recebidos corretamente
  - ▢ armazena pacotes no buffer, conforme necessário, para posterior entrega em-ordem à camada superior
- ▢ remetente apenas re-envia pacotes para os quais **ACK** não recebido
  - ▢ temporizador de remetente para cada pacote sem **ACK**
- ▢ janela do remetente
  - ▢ N nos. de seq consecutivos
  - ▢ outra vez limita nos. de seq de pacotes enviados, mas ainda não reconhecidos

# Retransmissão seletiva: janelas de remetente, receptor



# Retransmissão seletiva

## remetente

### dados de cima:

- se próx. no. de seq na janela, envia pacote

### timeout(n):

- reenvia pacote n, reiniciar temporizador

### ACK(n) em

[sendbase, sendbase+N]:

- marca pacote n "recebido"
- se n for menor pacote não reconhecido, avança base da janela ao próx. no. de seq não reconhecido

## receptor

### pacote n em

[rcvbase, rcvbase+N-1]

- envia ACK(n)
- fora de ordem: buffer
- em ordem: entrega (tb. entrega pacotes em ordem no buffer), avança janela p/ próxima pacote ainda não recebido

### pacote n em

[rcvbase-N, rcvbase-1]

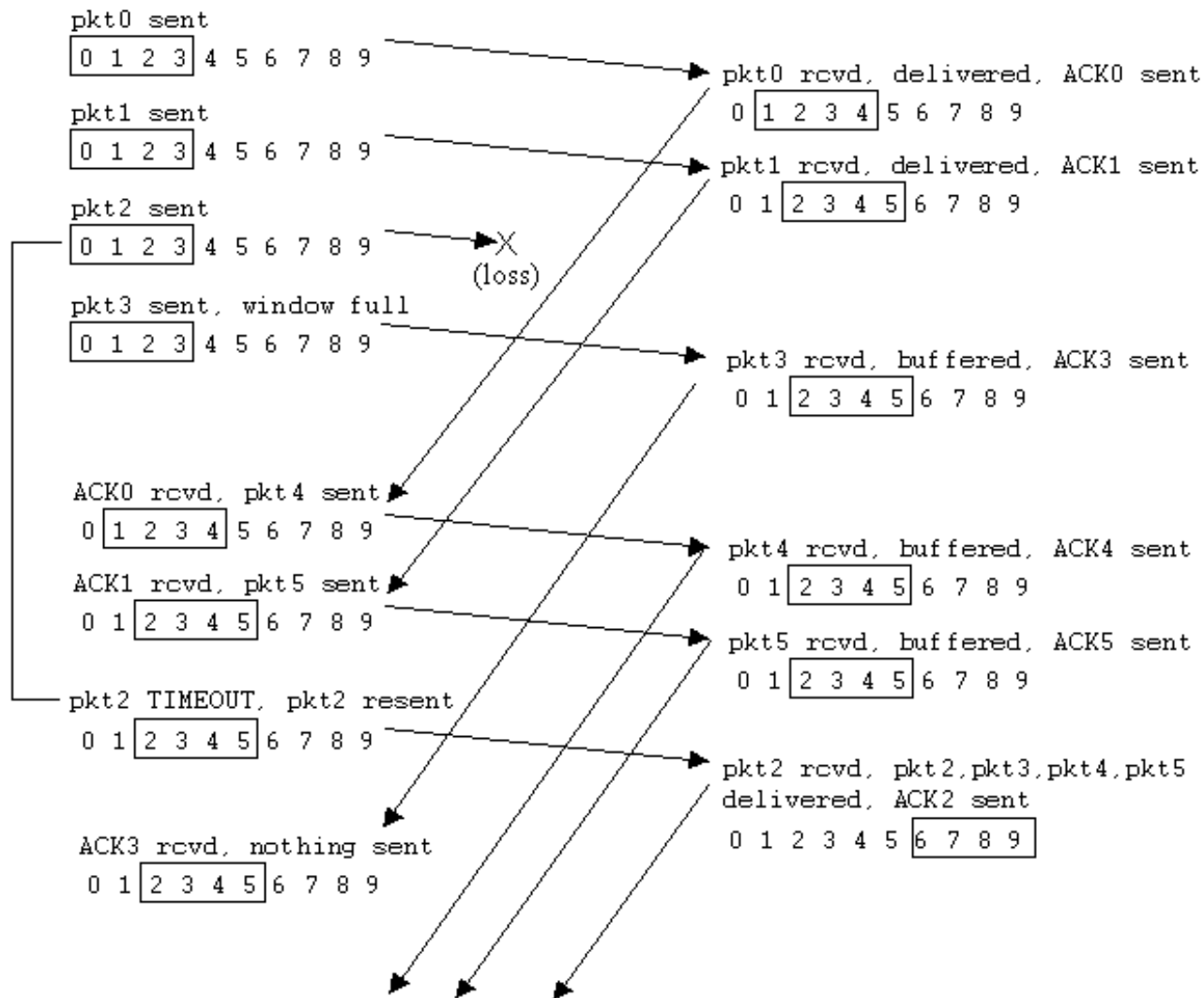
- ACK(n)

### senão:

- ignora



# Retransmissão seletiva em ação



# Retransmissão seletiva: dilema

Exemplo:

- nos. de seq : 0, 1, 2, 3
- tam. de janela = 3

- receptor não vê diferença entre os dois cenários!
- incorretamente passa dados duplicados como novos em (a)

Q: qual a relação entre tamanho de no. de seq e tamanho de janela?

