PTC 3450 - Aula 11

3.4 Princípios da transferência de dados confiável

(Kurose, Seções 3.4 e 3.5)

28/04/2017

Capítulo 3: conteúdo

3.1 serviços da camada de transporte

3.2 multiplexação e desmultiplexação

3.3 transporte sem conexão: UDP

3.4 princípios da transferência de dados confiável

3.5 transporte conectado a transporte: TCP

- estrutura dos segmentos
- transferência de dados confiável
- controle de fluxo
- gerenciamento de conexão

3.6 princípios do controle de congestionamento

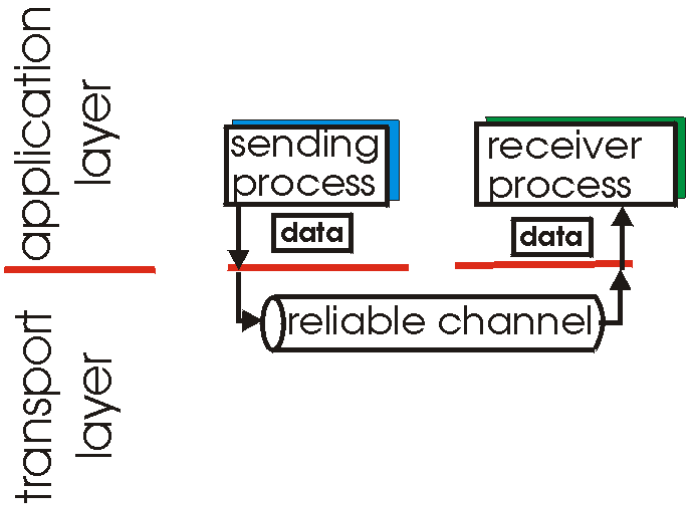
3.7 controle de congestionamento no TCP

RDT – *Reliable Data Transfer*

- Um dos aspectos que permeiam todas a pilha de protocolos é o de *transferência confiável de dados* (RDT – *reliable data transfer*)
- Entre fonte e destino, pacotes podem passar por diversos enlaces diferentes, com características e probabilidade de erros muito diferentes
  - Podem ser descartados em cada um dos roteadores no meio da rota
  - Podem ser corrompidos
  - Podem percorrer rotas diferentes, tendo atrasos diferentes
- Mesmo assim, muitas aplicações são viáveis apenas se garante-se que todos os pacotes sejam entregues e em ordem: comércio eletrônico, e-mail, transferência de arquivos de dados, etc.
- Nessa aula, vamos estudar técnicas que são utilizadas por diversos protocolos para implementar o RDT
- Em seguida, como exemplo, vamos ver como o TCP implementa essas técnicas

Princípios da transferência confiável de dados

- RDT usada nas camadas de aplicação, transporte e enlace
  - Tópico de importância central na área de redes!

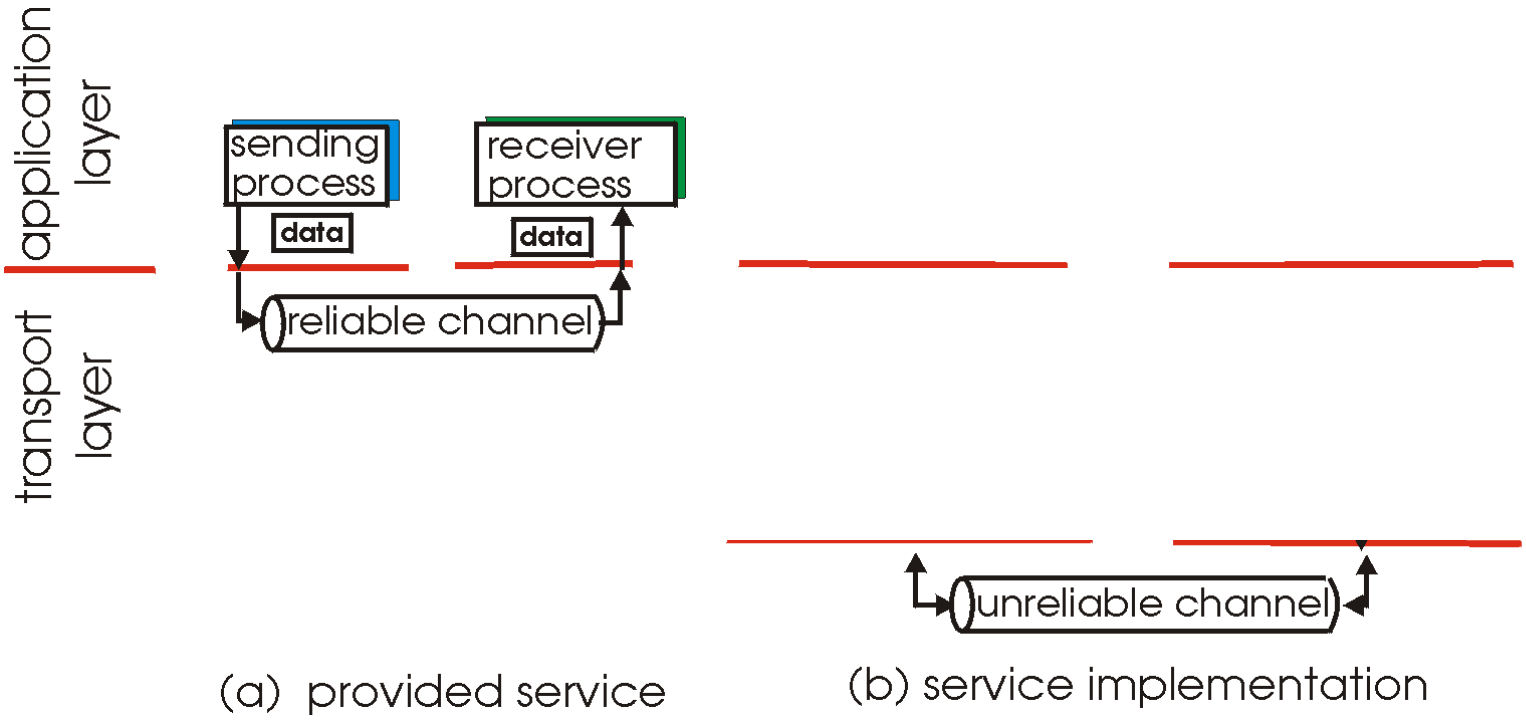


(a) provided service

- características do canal não confiável determinarão complexidade do protocolo de transferência de dados confiável (rdt - *reliable data transfer protocol*)

Princípios da transferência confiável de dados

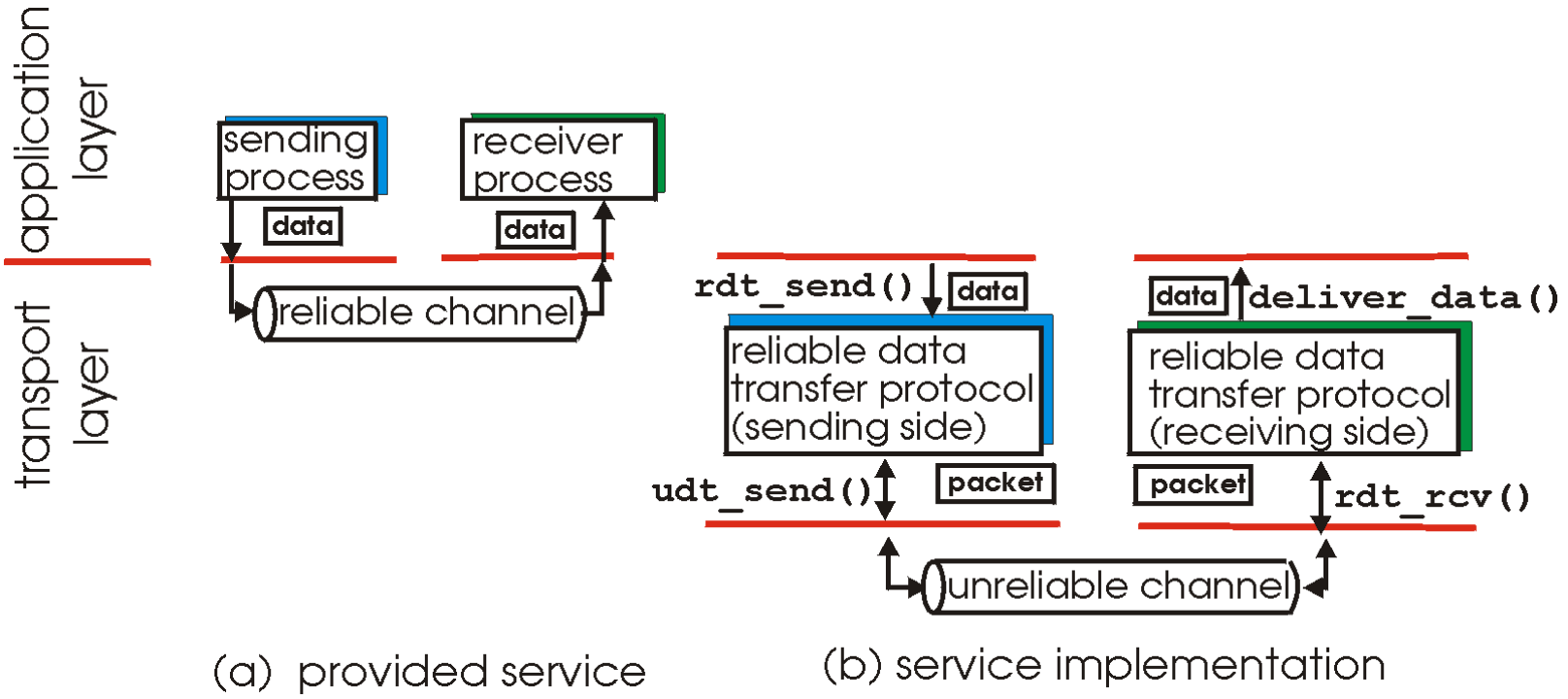
- importante nas camadas de aplicação, transporte e enlace
  - na lista top-10 dos tópicos mais importantes em redes!



- características do canal não confiável determinarão complexidade do protocolo de transferência de dados confiável (RDT - *reliable data transfer protocol*)

Princípios da transferência confiável de dados

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

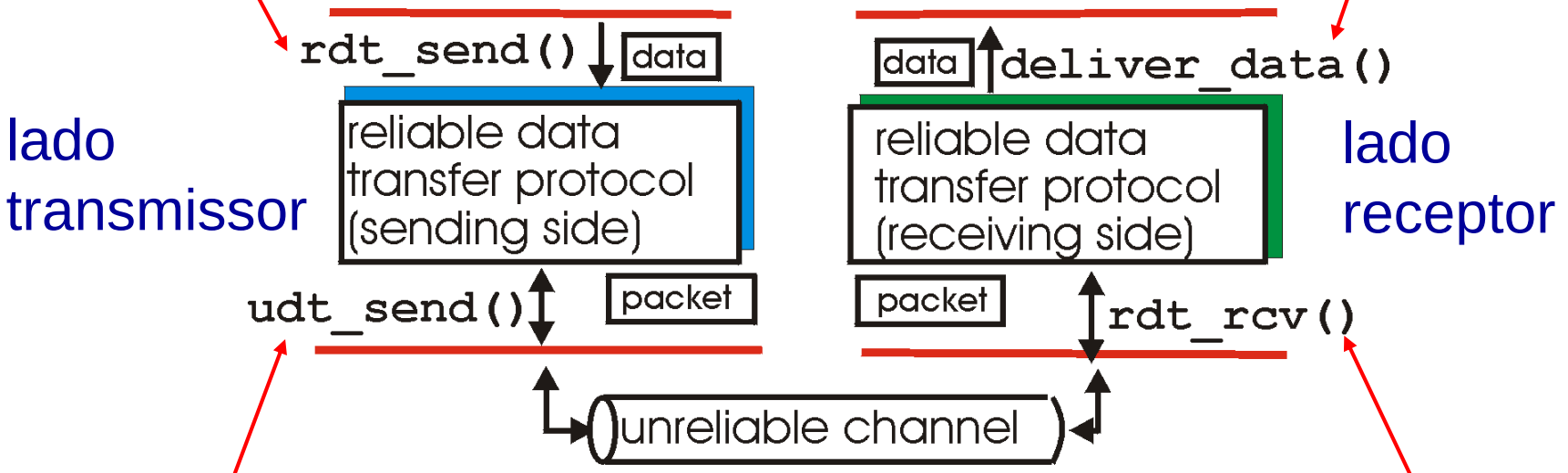


- características do canal não confiável determinarão complexidade do protocolo de transferência de dados confiável (RDT - *reliable data transfer protocol*)

Transferência confiável de dados: começando

`rdt_send()`: chamada de cima, por aplicação. Passados dados a serem entregues à camada superior do destinatário *(reliable data transfer)*

`deliver_data()`: chamado pelo **rdt** para entregar dados para camada superior



`udt_send()`: chamado pelo rdt, para transferir pacote sobre canal não confiável ao destinatário *(unreliable data transfer)*

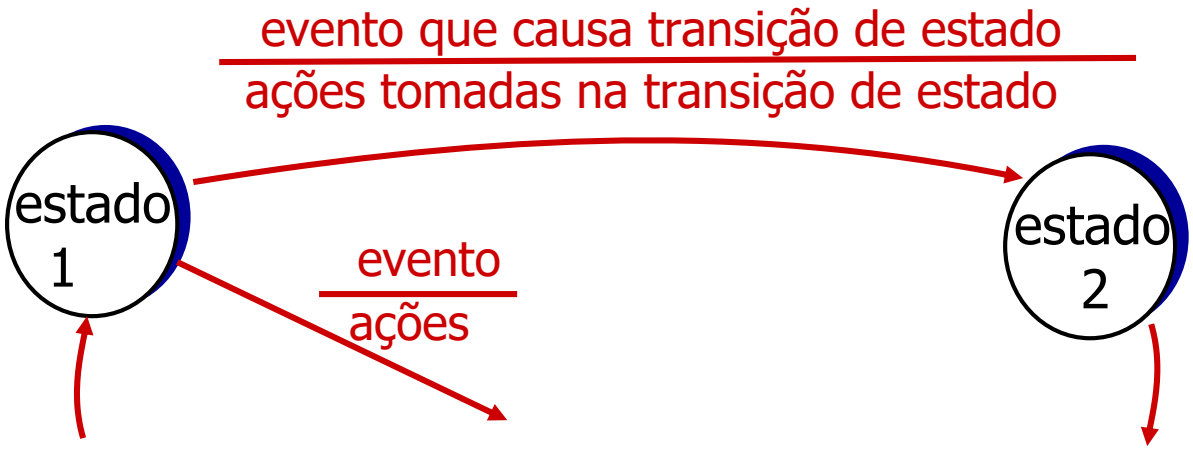
`rdt_rcv()`: chamado quando pacote chega no lado rec. do canal

Transferência confiável de dados: começando

vamos:

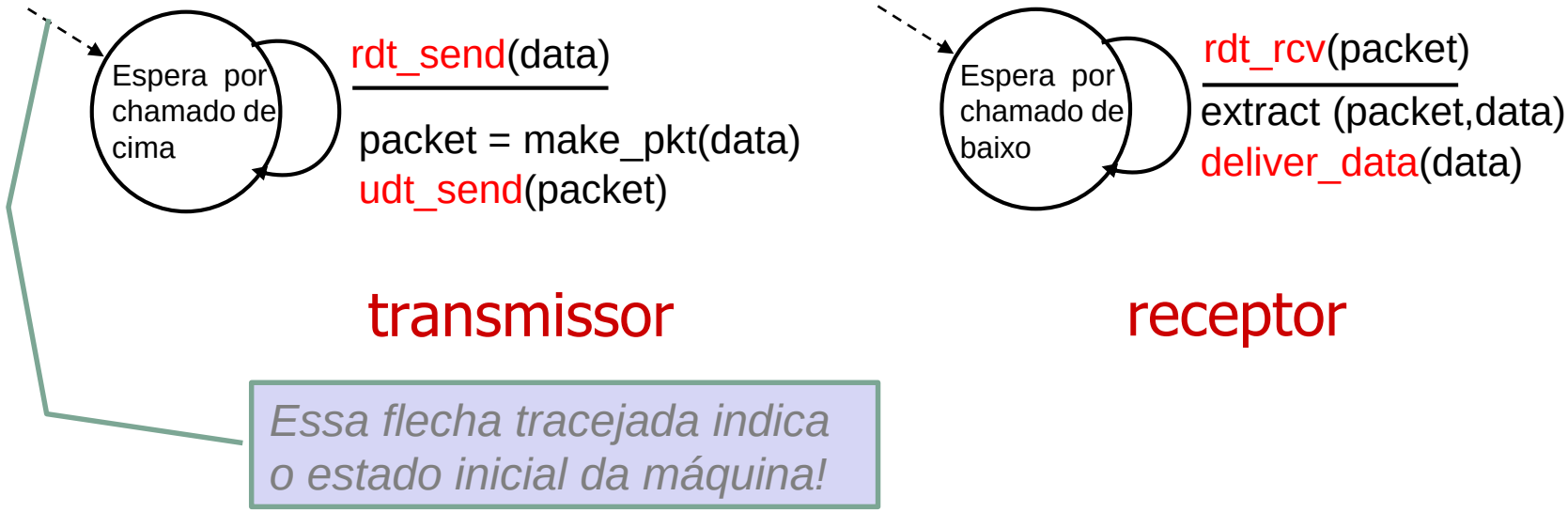
- desenvolver lados transmissor e receptor do protocolo RDT de forma incremental
- considerar apenas *transferência unidirecional de dados*
  - mas informações de controle fluirão em ambas direções!
- usar máquinas de estado finito (FSM – *Finite State Machine*) para especificar algoritmos no transmissor e receptor

estado: quando nesse "estado" próximo estado unicamente determinado pelo próximo evento



rdt1.0: RDT sobre canal confiável

- canal subjacente perfeitamente confiável
  - não há erros em bits
  - não há perdas de pacotes
- separar FSMs para transmissor e receptor:
  - remetente envia dados para o canal subjacente
  - destinatário lê dados do canal subjacente



rdt2.0: canal com erros em bits

- canal subjacente pode corromper bits em pacote, mas não perde ou muda ordem de pacotes
  - podemos usar códigos (por exemplo, CRC) para detectar erros em bits
  - *a* questão: como se recuperar de erros?

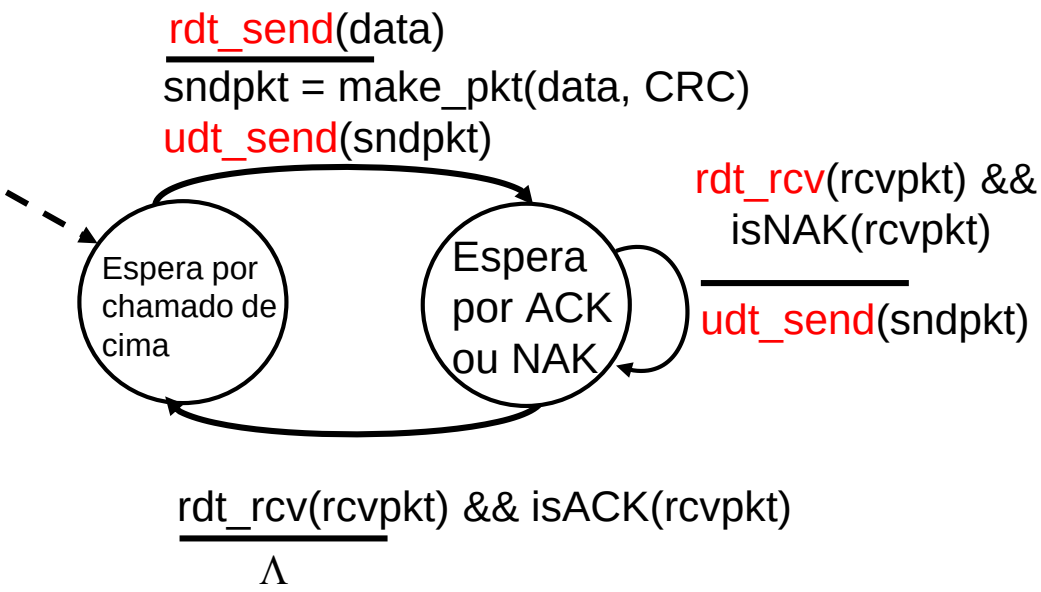
Como humanos se recuperam de "erros" durante conversação?

rdt2.0: canal com erros em bits

- canal subjacente pode corromper bits em pacote, mas não perde ou muda ordem de pacotes
  - podemos usar códigos (por exemplo, CRC) para detectar erros em bits
  - *a* questão: como se recuperar de erros?
  - *positive acknowledgements (ACKs):* receptor explicitamente conta a transmissor que pacote recebido não possui erros
  - *negative acknowledgements (NAKs):* receptor explicitamente conta a transmissor que pacote tem erros
  - transmissor retransmite pacote quando recebe NAK
  - Protocolos ARQ (*Automatic Repeat reQuest*)

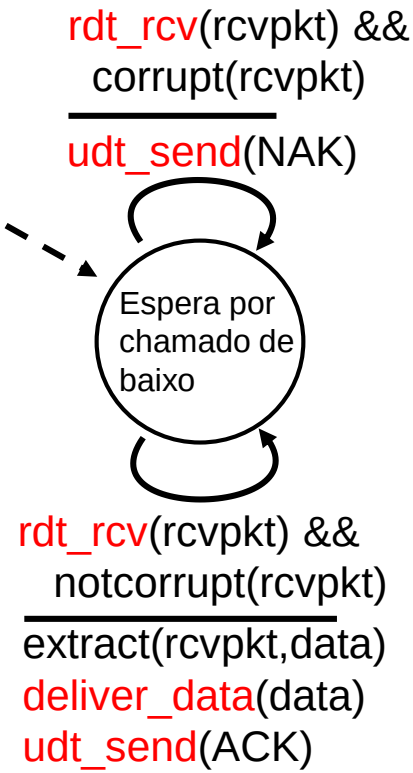
- novos mecanismos em `rdt2.0` (evolução do `rdt1.0`):
  - detecção de erros
  - realimentação: mensagens de controle (ACK, NAK) do receptor para transmissor
  - retransmissão: pacotes com erro são retransmitidos

rdt2.0: especificação por FSM

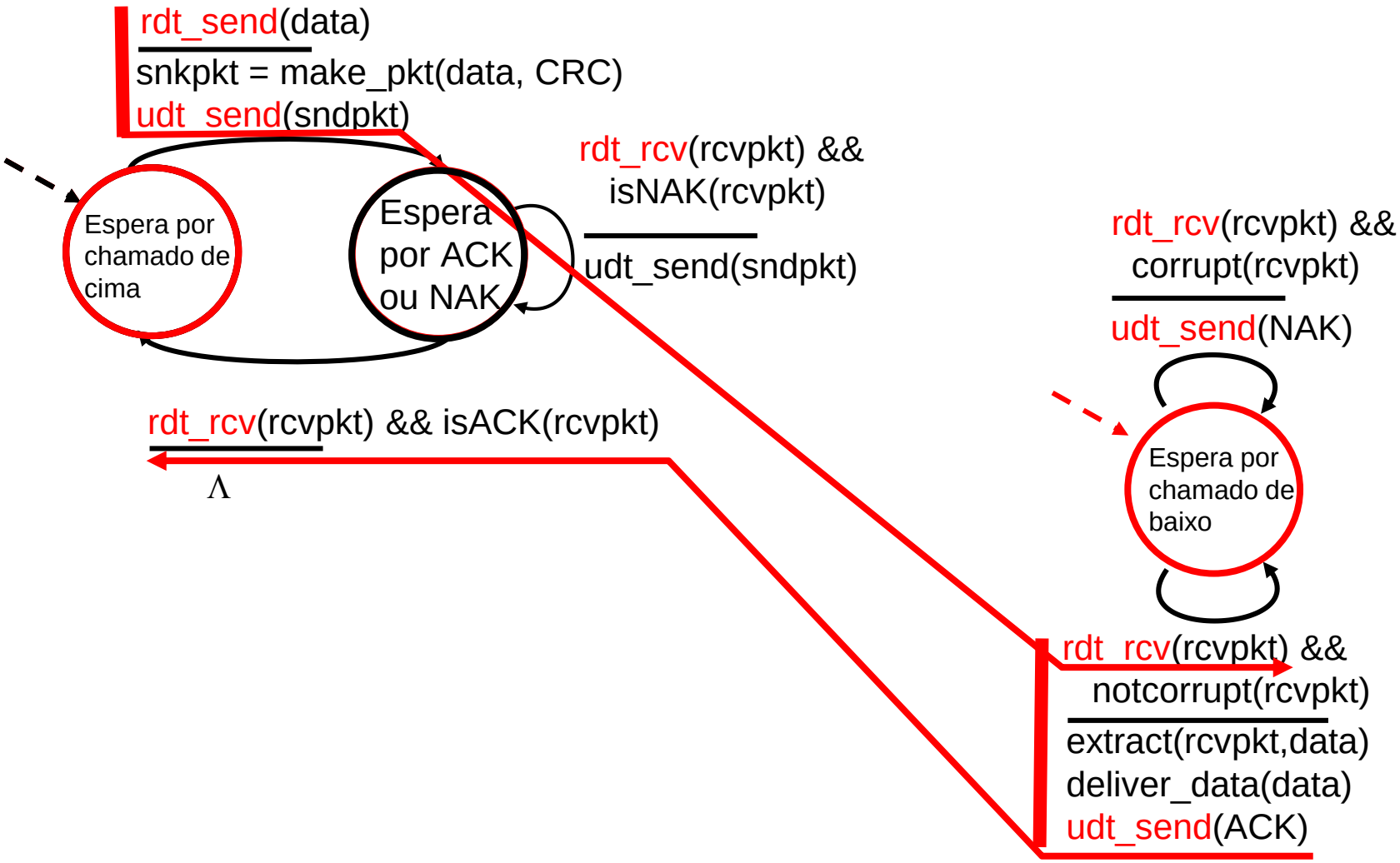


Transmissor

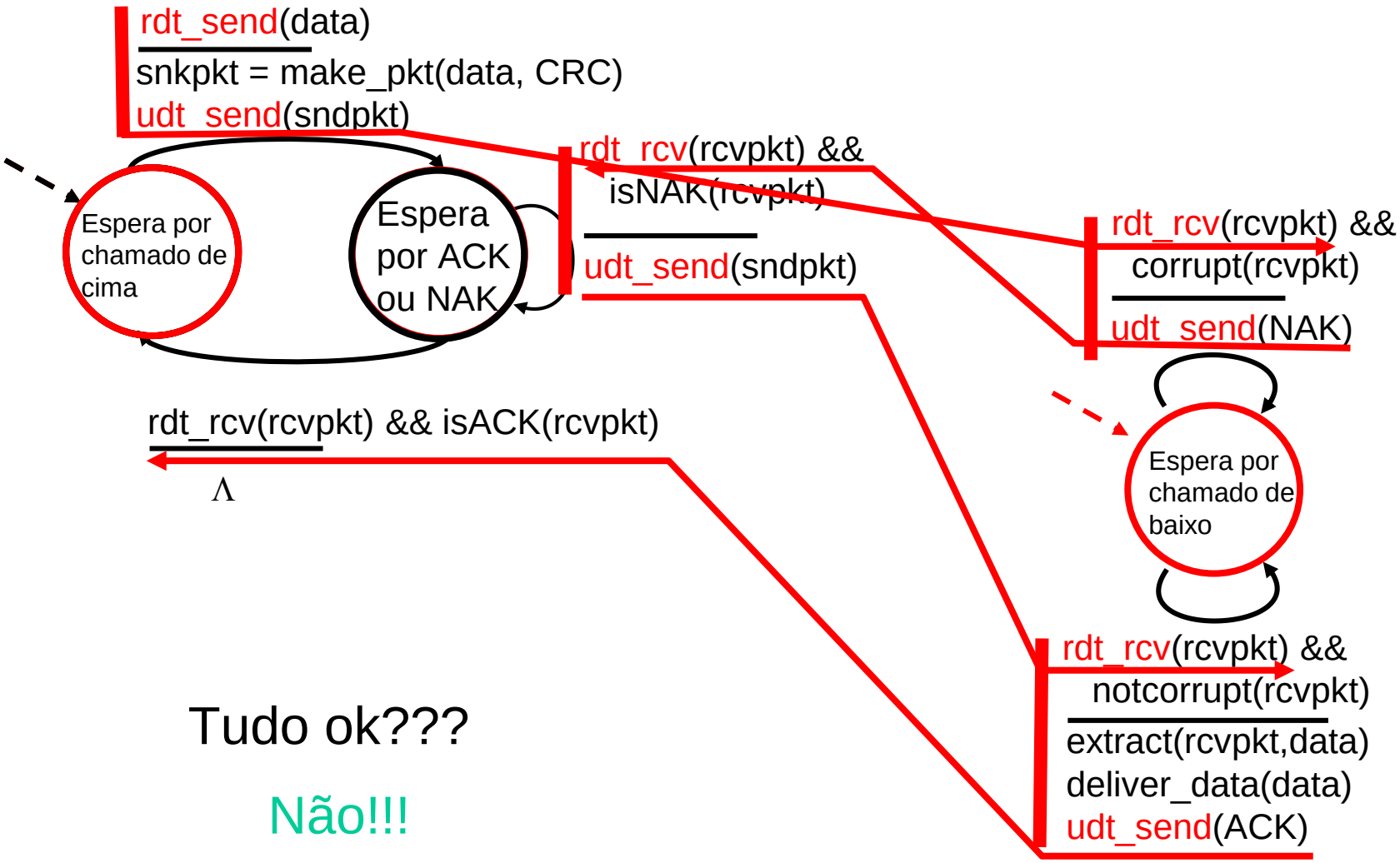
Receptor



rdt2.0: operação sem erros



rdt2.0: cenário com erro



Tudo ok???

Não!!!

rdt2.0 tem um defeito fatal!

o que acontece se ACK/NAK corrompido?

- Transmissor não sabe o que aconteceu no receptor!
- Pode simplesmente retransmitir?
- Não! Possível duplicata!
- Como resolver?

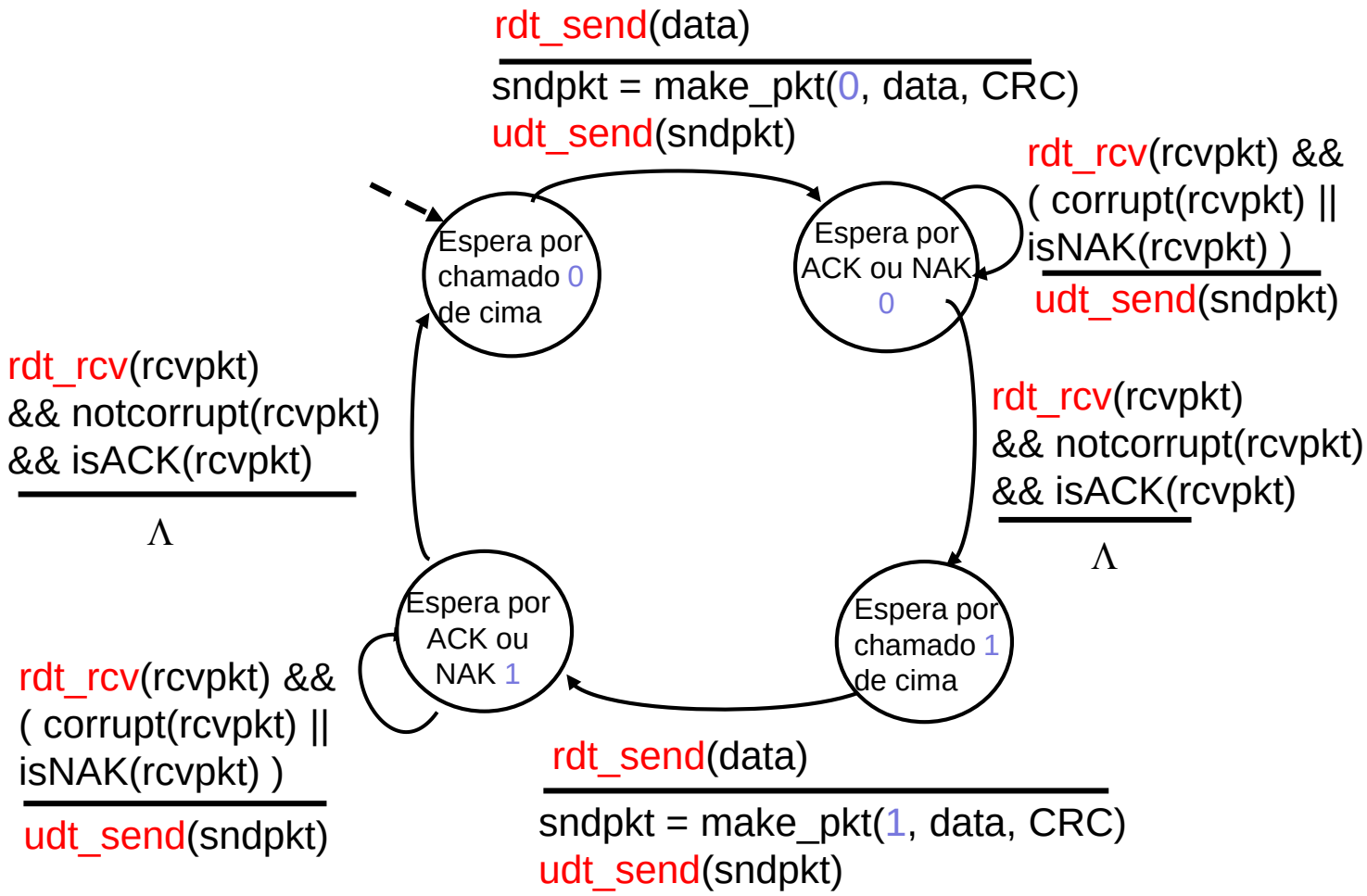
lidando com duplicatas:

- Transmissor retransmite pacote corrente se ACK/NAK corrompido
- Transmissor adiciona *número sequencial* a cada pacote
- Receptor descarta (não entrega) pacote duplicado

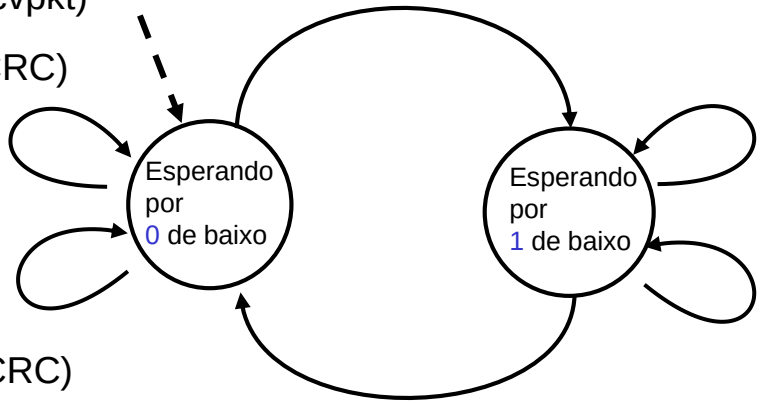
stop and wait

Transmissor envia um pacote, então espera por resposta do receptor

rdt2.1: Transmissor, lidando com ACK/NAKs corrompidos



rdt2.1: Receptor, lidando com ACK/NAKs corrompidos



rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, CRC)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && corrupt(rcvpkt)

sndpkt = make_pkt(NAK, CRC)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq1(rcvpkt)

sndpkt = make_pkt(ACK, CRC)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && corrupt(rcvpkt)

sndpkt = make_pkt(NAK, CRC)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq0(rcvpkt)

sndpkt = make_pkt(ACK, CRC)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, CRC)
udt_send(sndpkt)

rdt2.1: Discussão

Transmissor:

- # seq adicionado ao pacote
- 2 #'s seq. (0,1) são suficientes. Por que?
- Precisa verificar se ACK/NAK recebido corrompido
- Dobro do número de estados
  - Estado precisa "lembrar" se pacote "esperado" deve ter # seq 0 ou 1

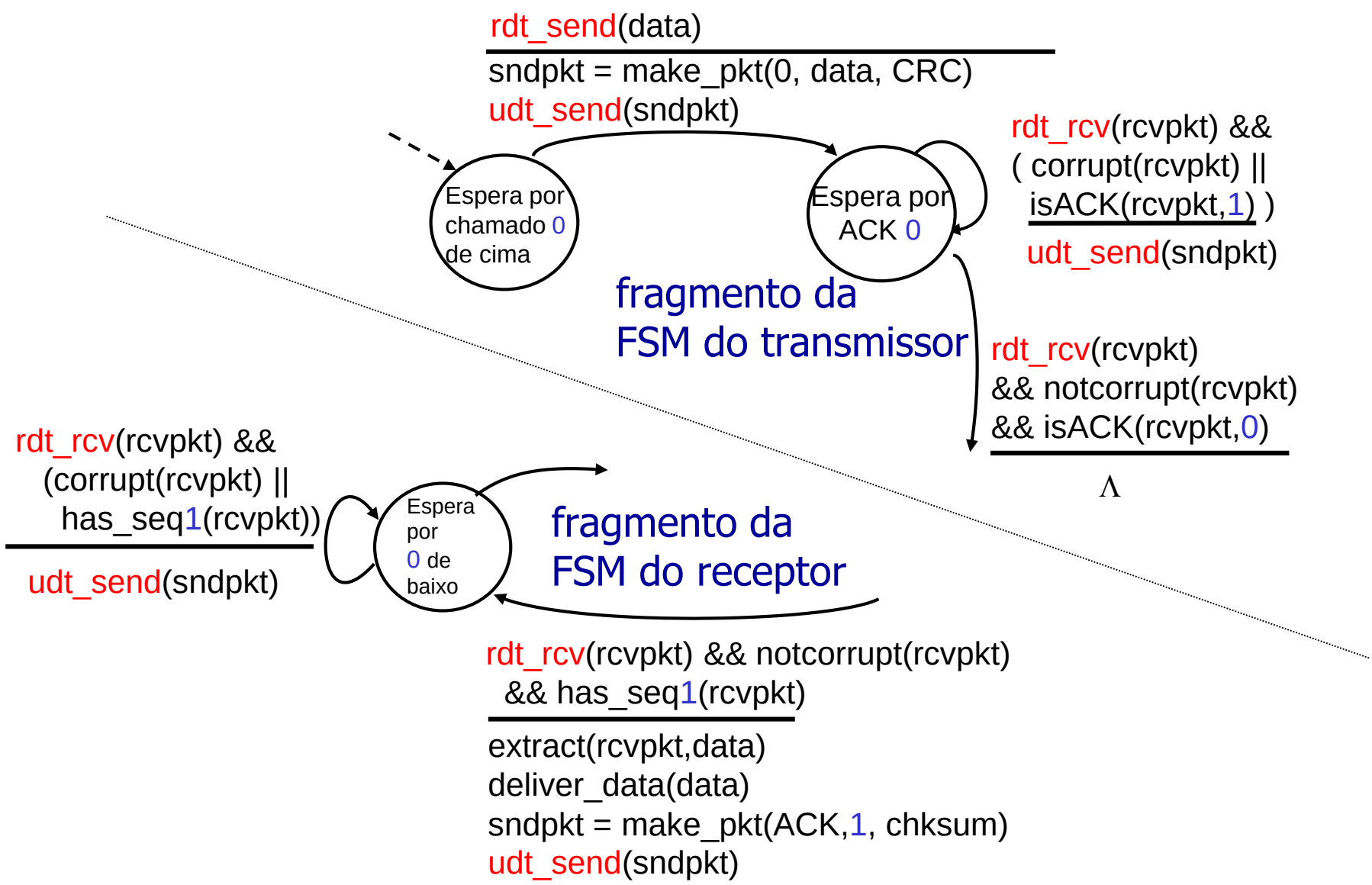
Receptor:

- Precisa checar se pacote recebido é duplicata
  - estado indica se 0 ou 1 é # seq do pacote esperado
- Precisa acrescentar dígitos de verificação de paridade no ACK ou NAK
- ACK ou NAK não precisa de número sequencial

rdt2.2: Um protocolo sem NAK

- Mesma funcionalidade do rdt2.1, usando apenas ACKs
- Em vez de NAK, receptor envia ACK para último pacote recebido OK
  - Receptor precisa incluir *explicitamente* número sequencial do pacote associado ao ACK
- ACK duplicado no transmissor resulta na mesma ação do NAK: *retransmite pacote atual*

rdt2.2: FSM para transmissor e receptor



rdt3.0: Canais com erros e perdas de pacotes

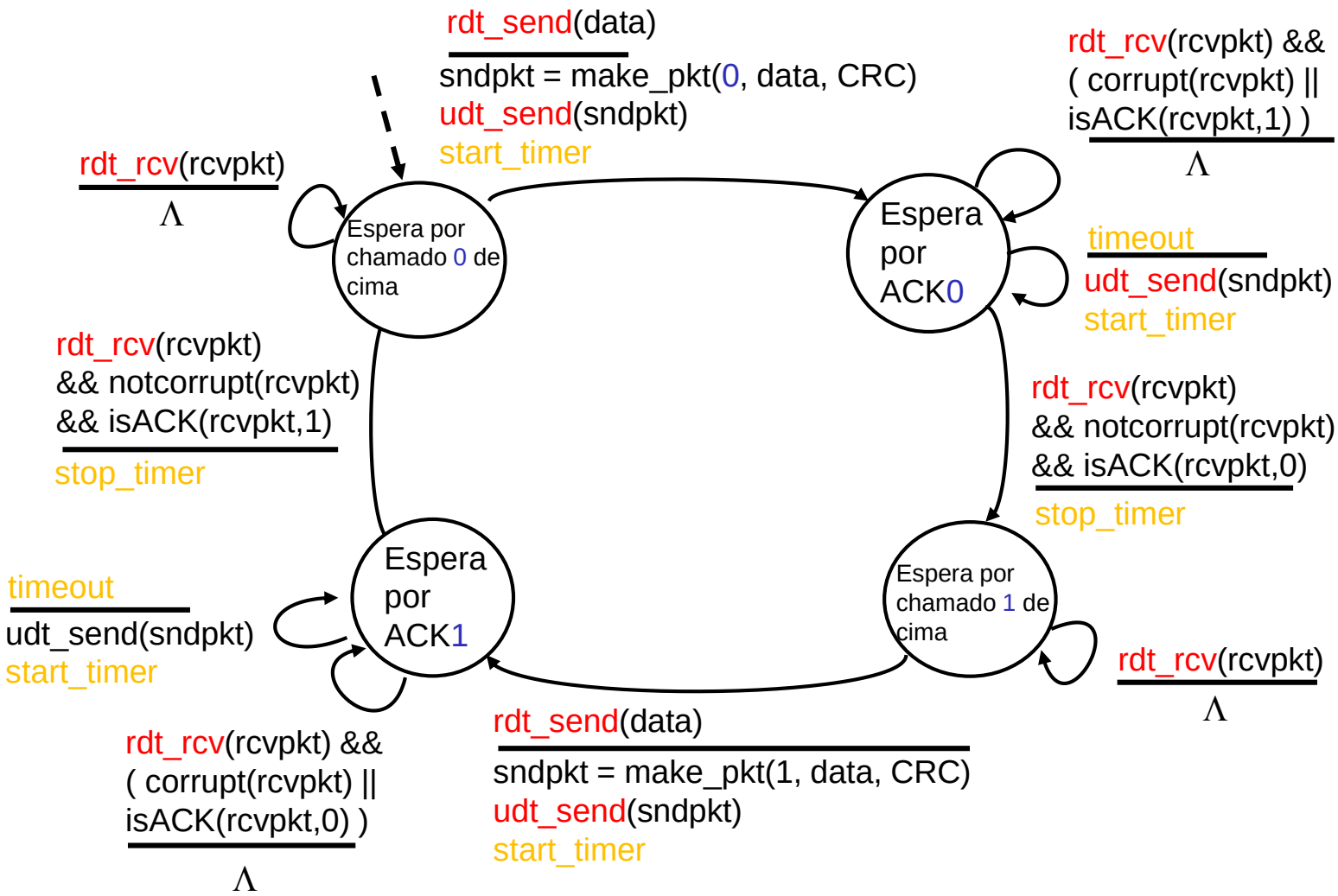
nova hipótese: canal subjacente também pode perder pacotes (dados, ACKs)

- *CRC*, # seq., ACKs, retransmissões ajudarão… mas não são suficientes

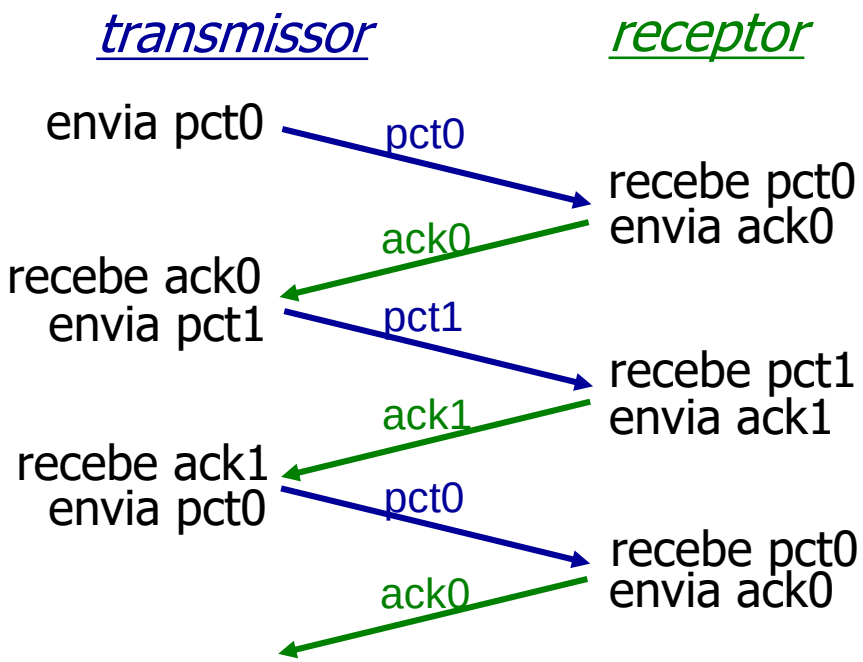
abordagem: transmissor espera tempo "razoável" por ACK

- retransmite se ACK não é recebido nesse tempo
- se pacote (ou ACK) apenas atrasado (não perdido):
  - retransmissão será duplicada, mas #'s seq. já lidam com isso
  - destinatário precisa especificar # seq de pacote sendo ACKed
- requer *temporizador* de contagem regressiva

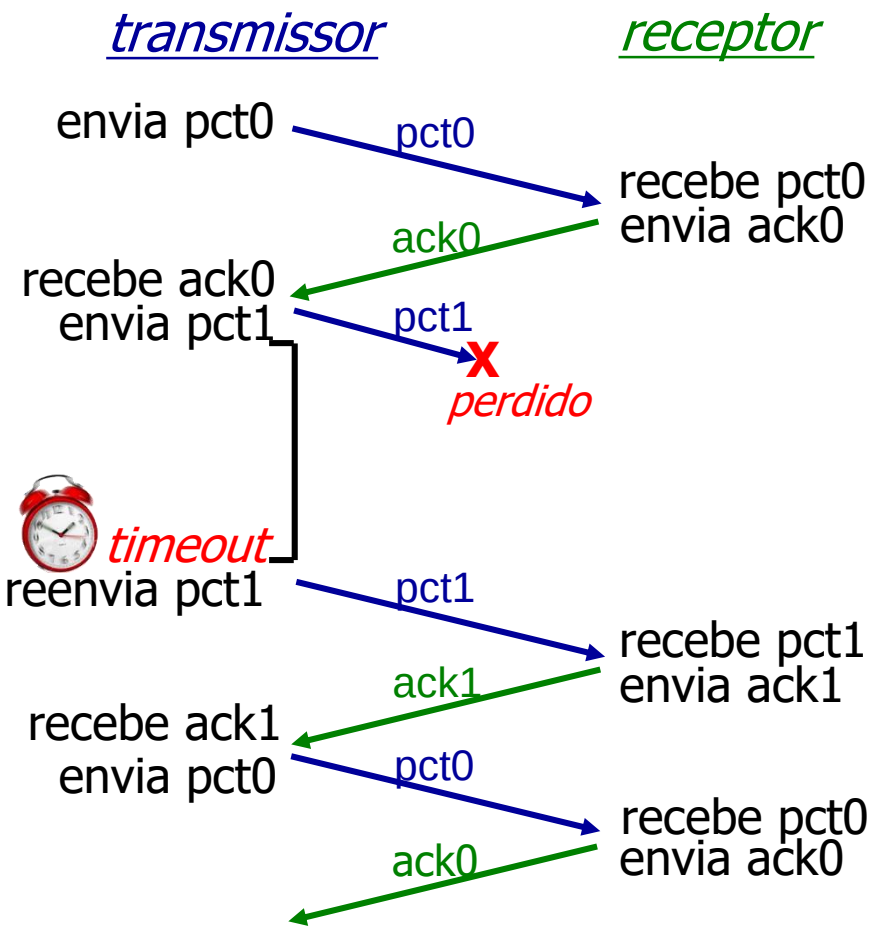
rdt3.0: FSM do Transmissor (Receptor é Exercício)



rdt3.0 em ação (*protocolo bit alternante*)

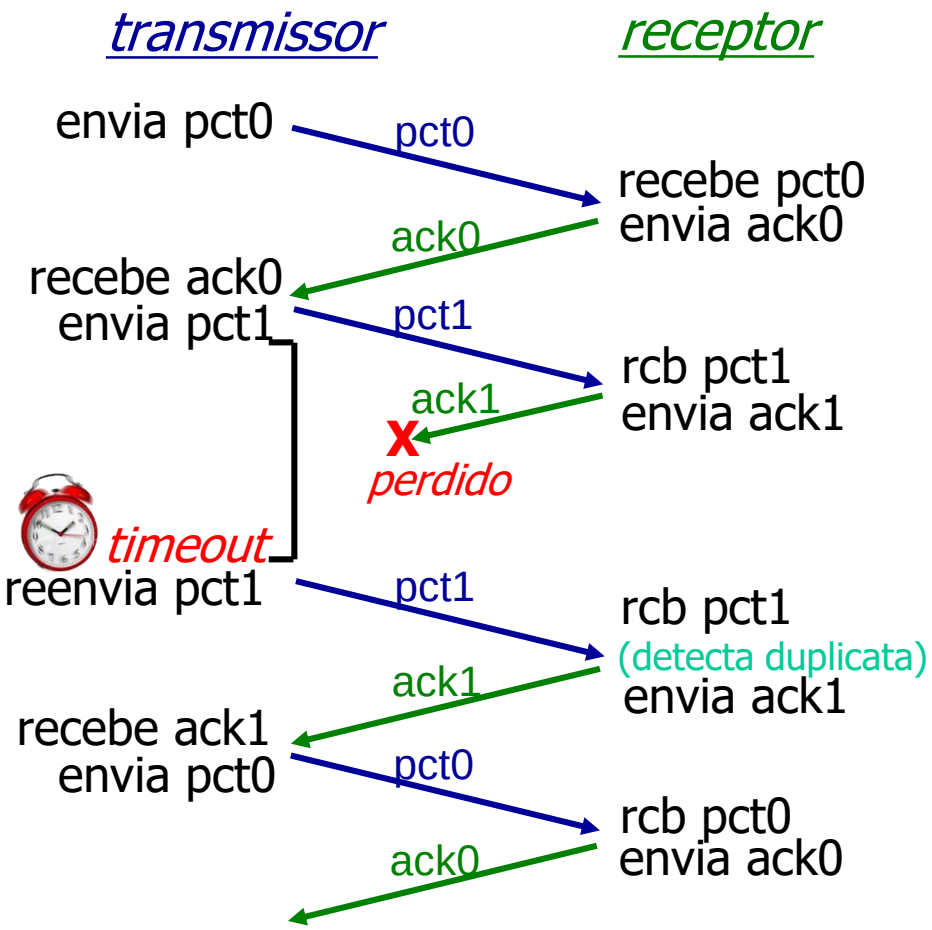


(a) sem perda

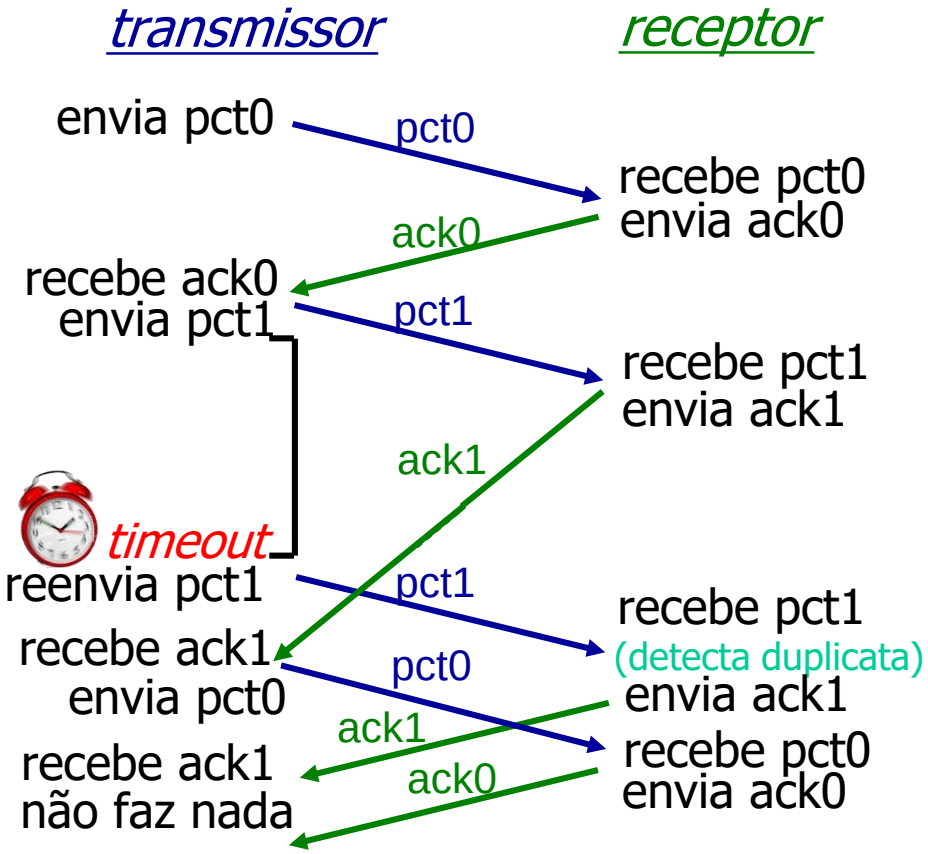


(b) perda de pacote

rdt3.0 em ação



(c) perda de ACK



(d) *timeout* prematuro/ ACK atrasado

Desempenho do rdt 3.0

- rdt 3.0 está correto, mas desempenho sofrível!
- *Exemplo: Enlace de 1 Gbps, atraso de propagação 15 ms, pacote de 8 000 bits*
- Atraso de Transmissão:

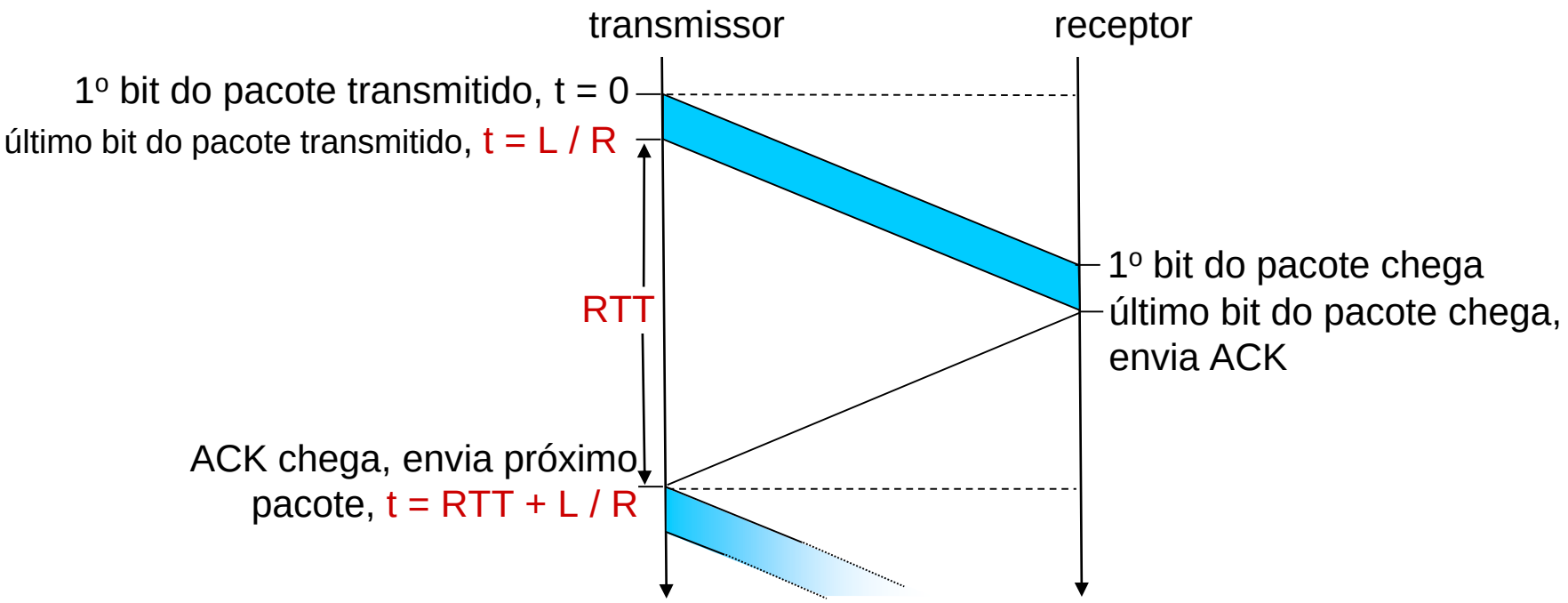
$$d_{\text{trans}} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/s}} = 8 \text{ ms}$$

- $U_{\text{transmissor}}$: *utilização* – fração do tempo em que transmissor está ocupado enviando

$$U_{\text{transmissor}} = \frac{\frac{L}{R}}{\text{RTT} + \frac{L}{R}} = \frac{0.008}{30.008} = 0.00027$$

  - Vazão
    - Um pacote enviado a cada aproximadamente 30 ms: vazão de 267 kbps em um enlace de 1 Gbps (!!!)
- Protocolo de rede limita uso de recurso físico!
- Preço pago para ter RDT (muito caro?)

rdt3.0: operação *stop-and-wait*

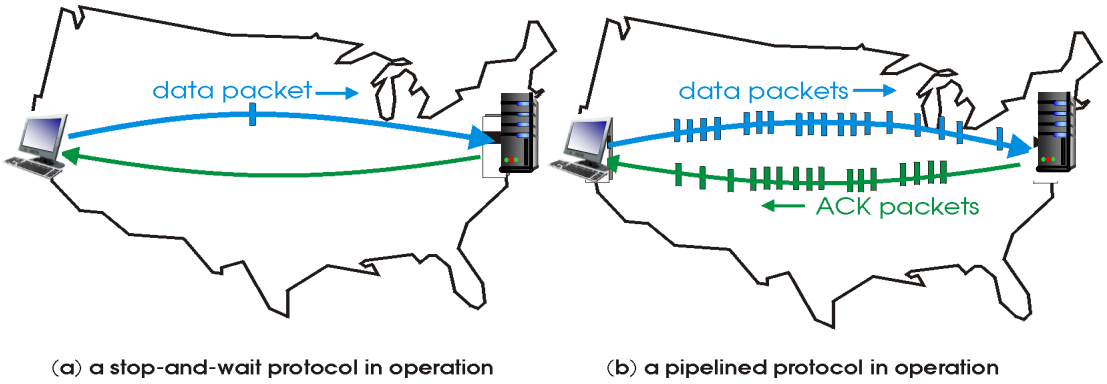


$$U_{\text{transmissor}} = \frac{\frac{L}{R}}{\text{RTT} + \frac{L}{R}} = \frac{0.008}{30.008} = 0.00027$$

Protocolos com paralelismo *(pipelining)*

Pipelining: transmissor permite múltiplos, pacotes "*in-flight*", ainda não reconhecidos (*acknowledged*)

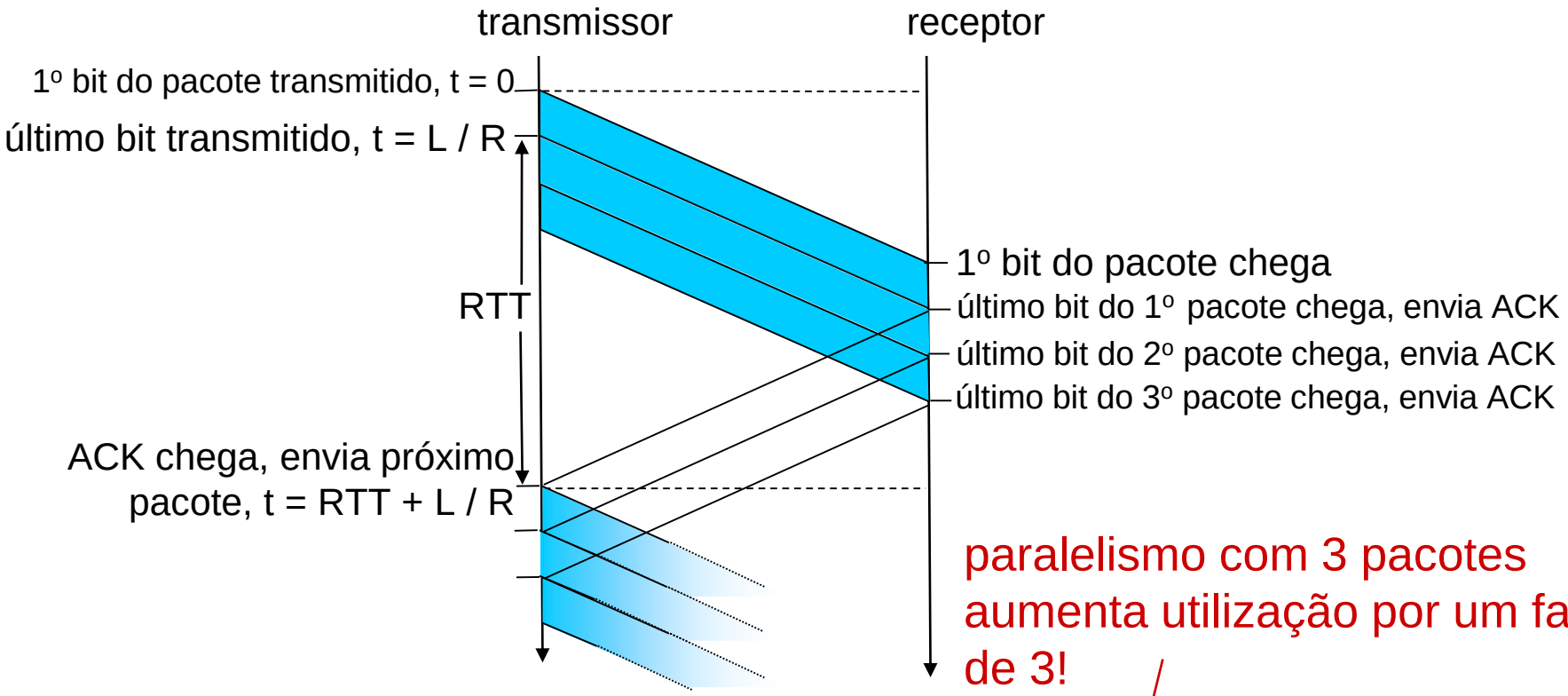
- intervalo dos números sequenciais precisa ser aumentada
- *buffers* no transmissor e/ou receptor



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

- 2 formas genéricas de protocolos com paralelismo: *go-Back-N, repetição seletiva*

Paralelismo: utilização aumentada



paralelismo com 3 pacotes aumenta utilização por um fator de 3!

Qual a utilização do remetente agora?

$$U_{\text{transmissor}} = \frac{3\frac{L}{R}}{\text{RTT} + \frac{L}{R}} = \frac{0.024}{30.008} = 0.00081$$

Protocolos com paralelismo: visão geral

Go-back-N (GBN):

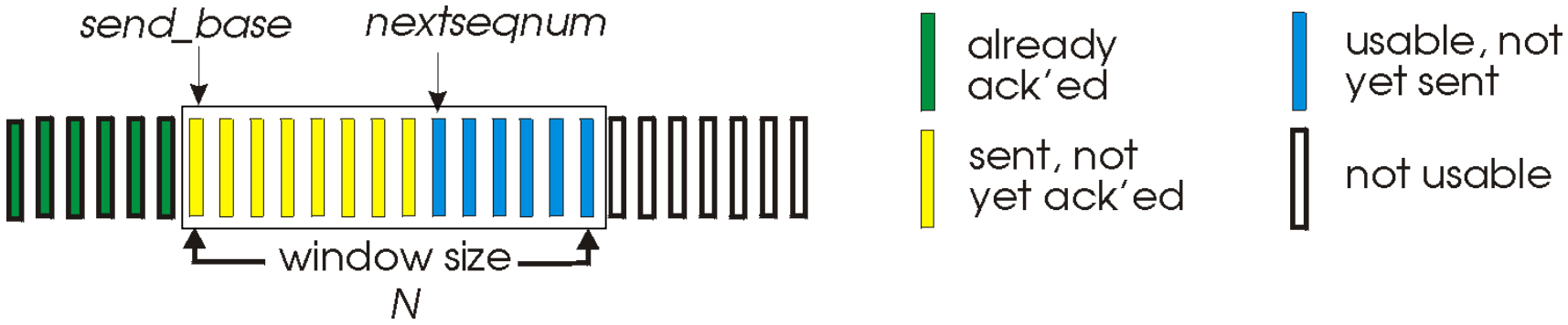
- Transmissor pode ter até *N* pacotes que ainda não retornaram ACKs
- Receptor envia *ACK acumulativo*
  - ACK *x* implica que todos os pacotes até *x* foram recebidos
- Transmissor tem apenas um temporizador associado ao pacote não reconhecido mais antigo
  - quando tempo expira, retransmite *todos* pacotes não reconhecidos

Repetição Seletiva (RS):

- Transmissor pode ter até *N* pacotes que ainda não retornaram ACKs
- Receptor envia *ACK individual* para cada pacote
- Transmissor mantém temporizador para cada pacote ainda não reconhecido
  - quando tempo expira, retransmite apenas aquele pacote associado com o temporizador expirado

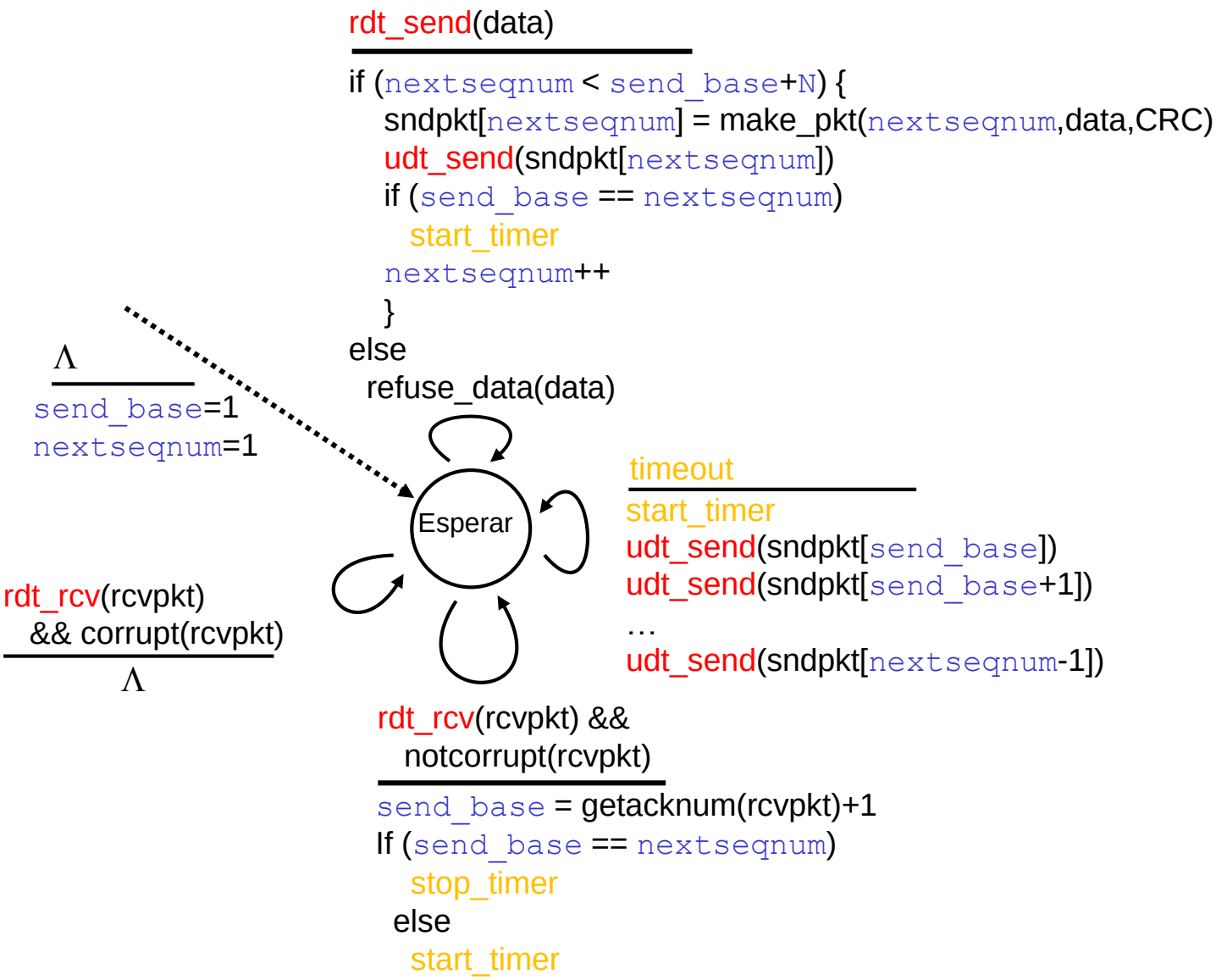
Go-Back-N: Transmissor

- número sequencial de *k*-bits no cabeçalho do pacote
- "janela" de até *N* pacotes consecutivos ainda não reconhecidos permitida

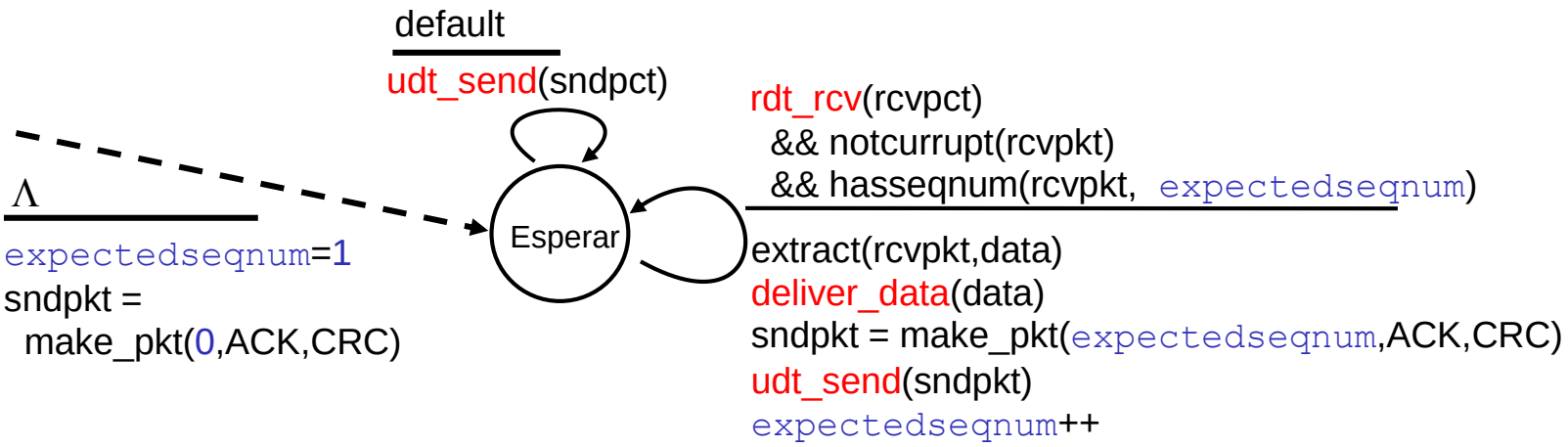


- ACK(*n*): quando recebido, reconhece recepção de todos os pacotes com # seq menor ou igual a *n* – *"ACK acumulativo"*
  - pode receber ACKs duplicados (ver receptor)
- temporizador para pacote não reconhecido mais antigo
- *timeout(n):* retransmite pacote *n* e todos pcts com # seq mais alto na janela
- É um *protocolo de janela deslizante*

GBN: FSM estendido (com variáveis) do transmissor



GBN: FSM estendido do receptor

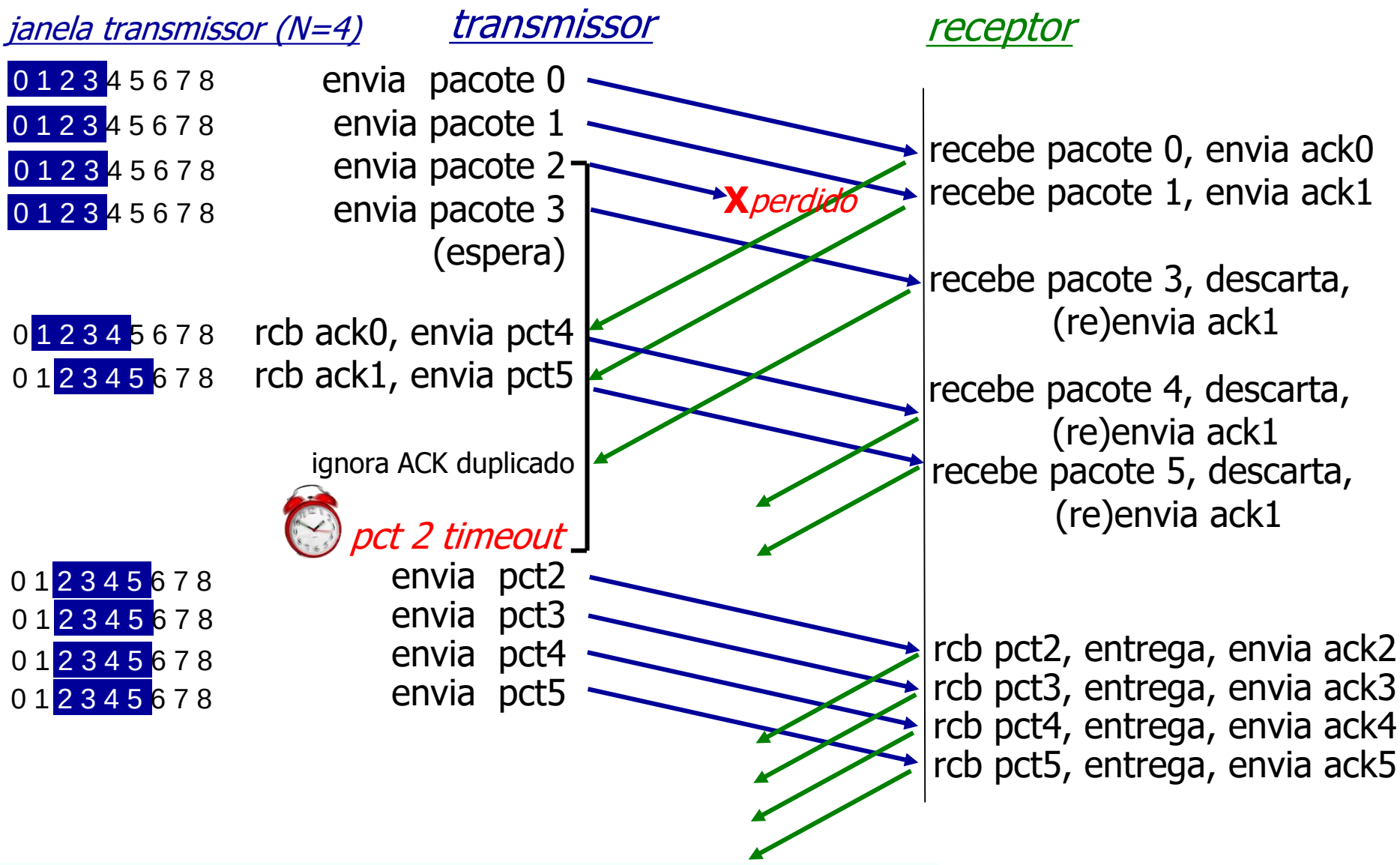


Sempre envia ACK para pacote corretamente recebido com maior # seq *em ordem (sem lacunas)*

- pode gerar ACKs duplicados
- precisa apenas guardar `expectedseqnum`

- pacotes fora de ordem:
  - descarta (não armazena): *não há buffer no destinatário!*
  - reenvia ACK pacote com maior número sequencial em ordem

GBN em ação



Ver: http://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_6/video_applets/GBNindex.html