Capítulo 3: Camada de Transporte

Metas do capítulo:

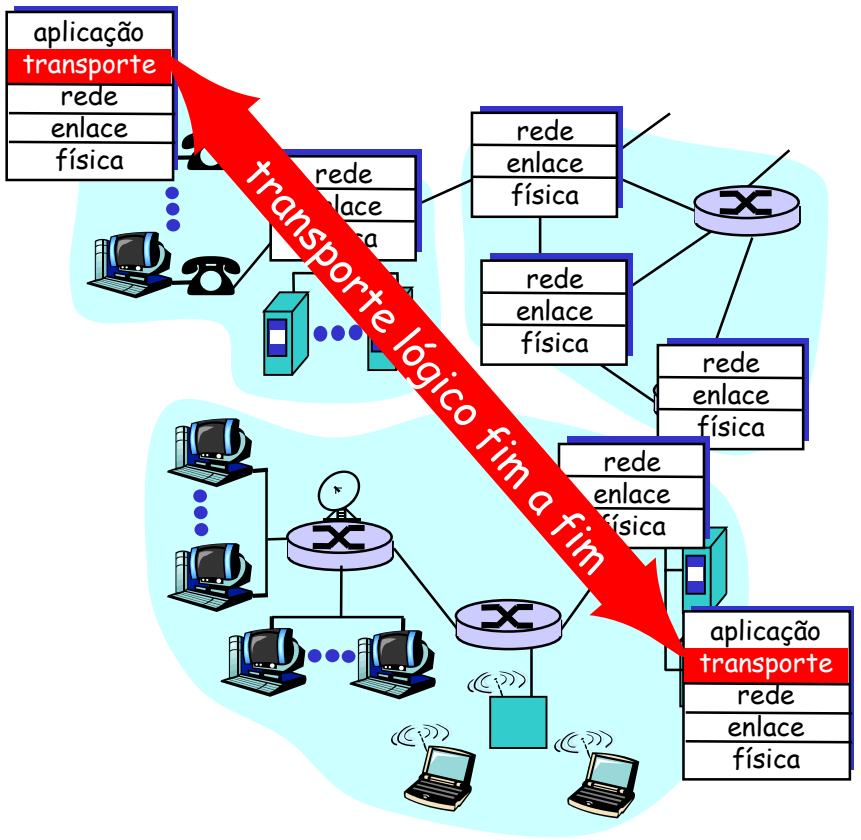
- compreender os princípios atrás dos serviços da camada de transporte:
  - multiplexação/ demultiplexação
  - transferência confiável de dados
  - controle de fluxo
  - controle de congestionamento
- instanciação e implementação na Internet

Resumo do Capítulo:

- serviços da camada de transporte
- multiplexação/demultiplexação
- transporte sem conexão: UDP
- princípios de transferência confiável de dados
- transporte orientado a conexão: TCP
  - transferência confiável
  - controle de fluxo
  - gerenciamento de conexões
- princípios de controle de congestionamento
- controle de congestionamento em TCP

Serviços e protocolos de transporte

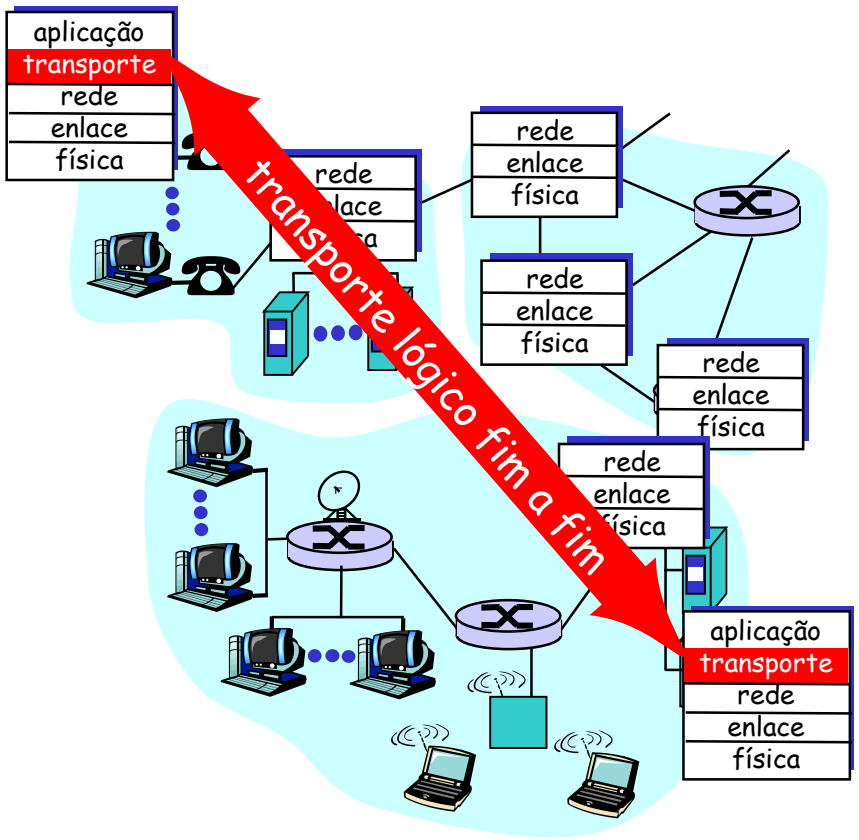
- provê *comunicação lógica* entre processos de aplicação executando em hospedeiros diferentes
- protocolos de transporte executam em sistemas terminais
- serviços das camadas de transporte X rede:
- *camada de rede :* dados transferidos entre sistemas
- *camada de transporte:* dados transferidos entre processos
  - ✓ depende de, estende serviços da camada de rede



Protocolos da camada de transporte

Serviços de transporte na Internet:

- entrega confiável, ordenada, ponto a ponto (TCP)
  - ✓ congestionamento
  - ✓ controle de fluxo
  - ✓ estabelecimento de conexão (setup)
- entrega não confiável, ("melhor esforço"), não ordenada, ponto a ponto ou multiponto: UDP
- serviços não disponíveis:
  - ✓ tempo-real
  - ✓ garantias de banda
  - ✓ multiponto confiável

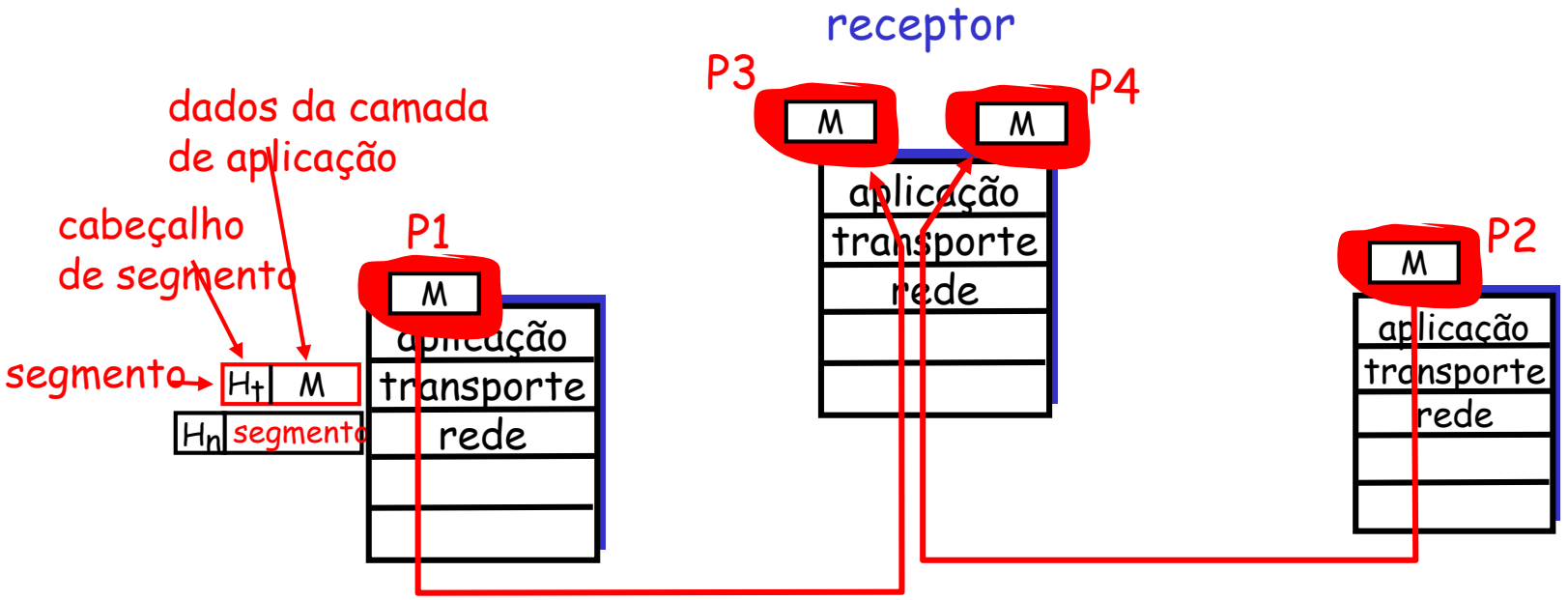


Multiplexação/demultiplexação

Lembrança: *segmento* - unidade de dados trocada entre entidades da camada de transporte

- ✓ = TPDU: transport protocol data unit

Demultiplexação: entrega de segmentos recebidos para os processos da camada de apl corretos

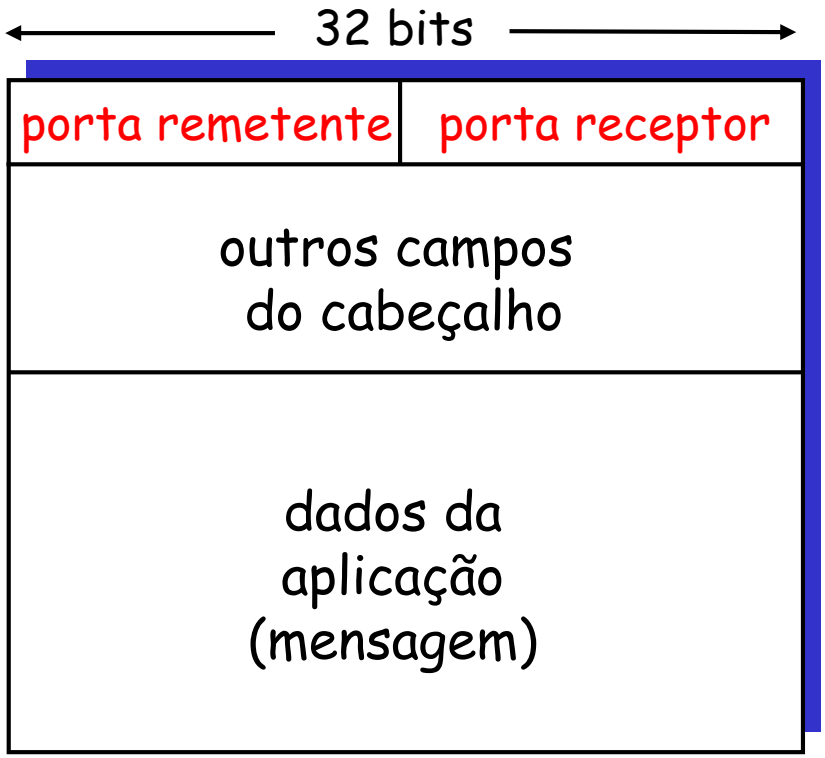


Multiplexação/demultiplexação

Multiplexação:
juntar dados de múltiplos processos de apl, envelopando dados com cabeçalho (usado depois para demultiplexação)

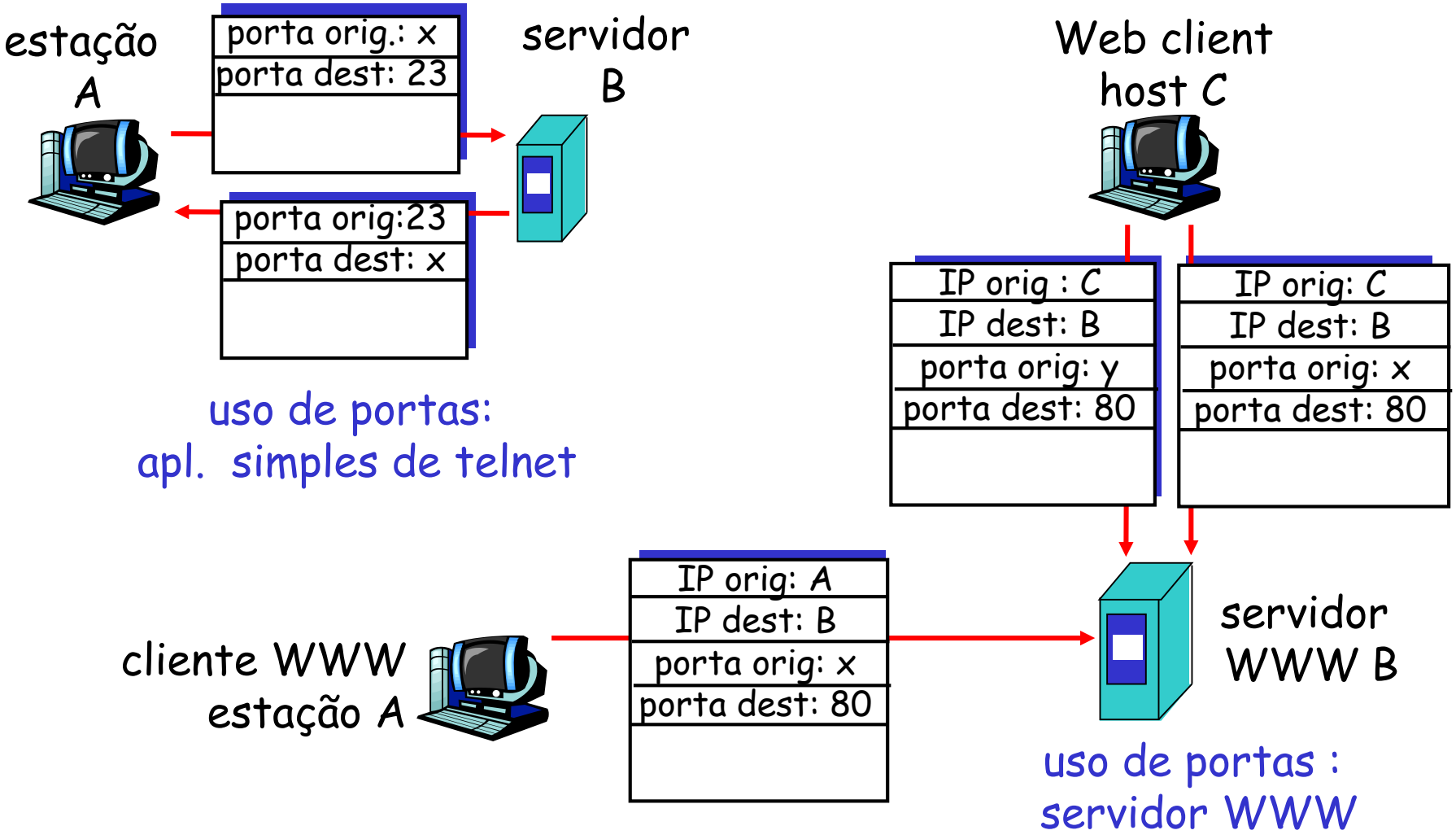
multiplexação/demultiplexação:

- baseadas em números de porta e endereços IP de remetente e receptor
  - ✓ números de porta de remetente/receptor em cada segmento
  - ✓ lembrete: número de porta bem conhecido para aplicações específicas



formato de segmento TCP/UDP

Multiplexação/demultiplexação: exemplos



UDP: User Datagram Protocol [RFC 768]

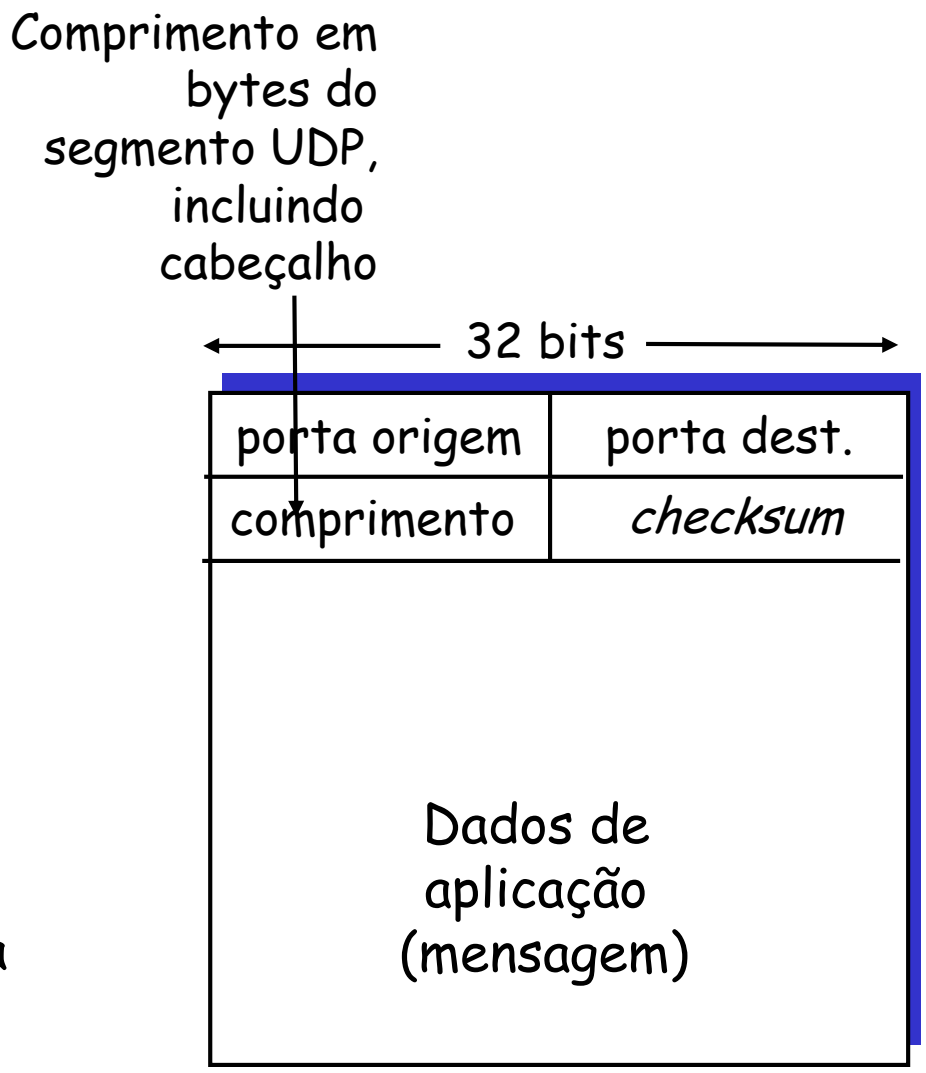
- Protocolo de transporte da Internet mínimo, "sem frescura",
- Serviço "melhor esforço", segmentos UDP podem ser:
  - ✓ perdidos
  - ✓ entregues à aplicação fora de ordem do remesso
- *sem conexão:*
  - ✓ não há "setup" UDP entre remetente, receptor
  - ✓ tratamento independente de cada segmento UDP

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.!



UDP segment format

Checksum UDP

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

Remetente:

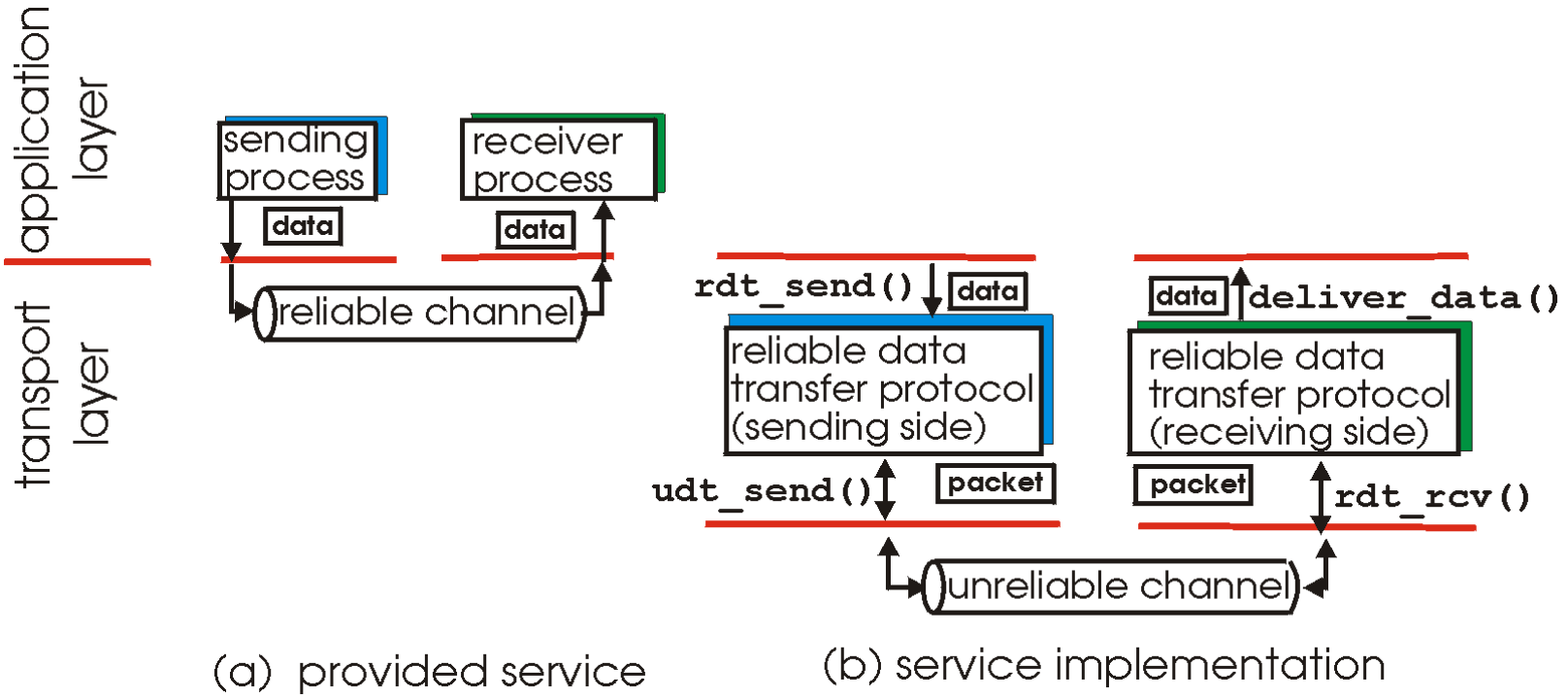
- trata conteúdo do segmento como seqüência de inteiros de 16-bits
- campo checksum zerado
- checksum: soma (adição usando complemento de 1) do conteúdo do segmento
- remetente coloca *complemento do valor da soma* no campo checksum de UDP

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 ….

Princípios de Transferência confiável de dados (rdt)

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

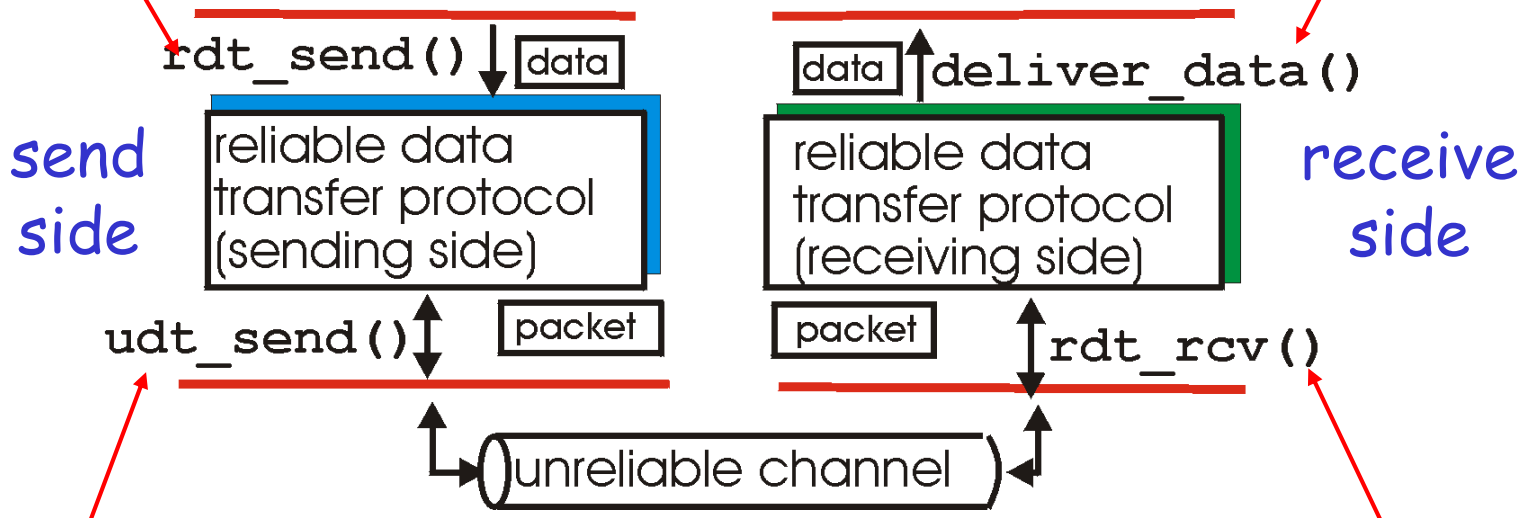


- características do canal não confiável determinam a complexidade de um protocolo de transferência confiável de dados (rdt)

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 ñ confiável ao receptor

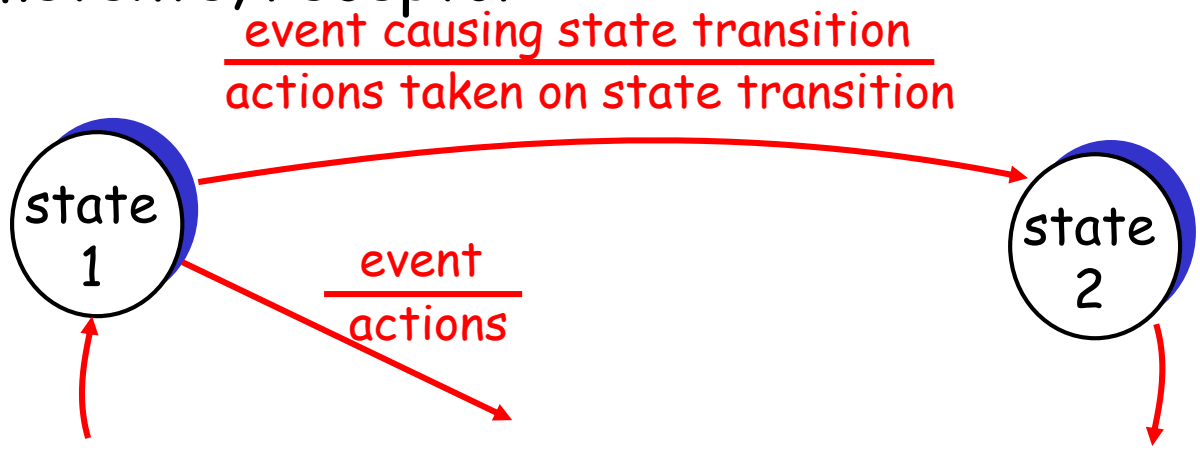
`rdt_rcv()`: chamada quando pacote chega chega no lado receptor do canal

Transferência confiável de dados (rdt): como começar

Iremos:

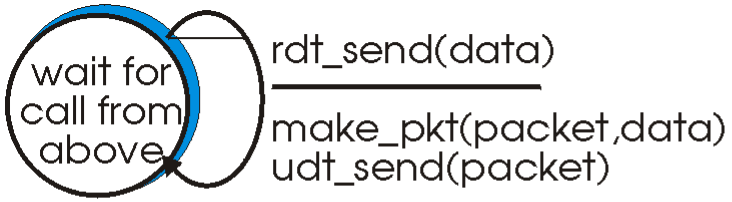
- desenvolver incrementalmente os lados remetente, receptor do protocolo RDT
- considerar apenas fluxo unidirecional de dados
  - ✓ mas info de controle flui em ambos os sentidos!
- Usar máquinas de estados finitos (FSM) p/ especificar remetente, receptor

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

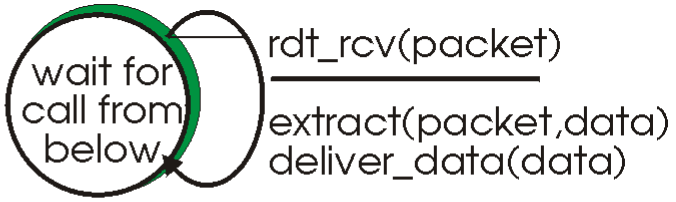


Rdt1.0: transferência confiável usando um canal confiável

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



(a) rdt1.0: sending side

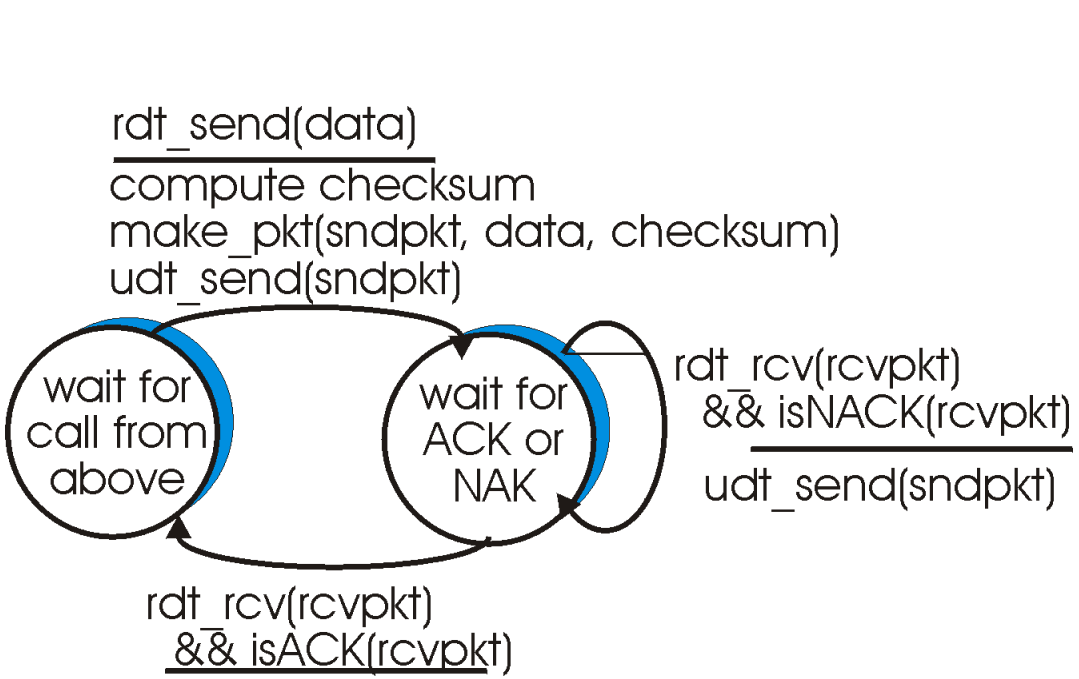


(b) rdt1.0: receiving side

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`**):
  - deteção de erros
  - realimentação pelo receptor: msgs de controle (ACK,NAK) receptor->remetente

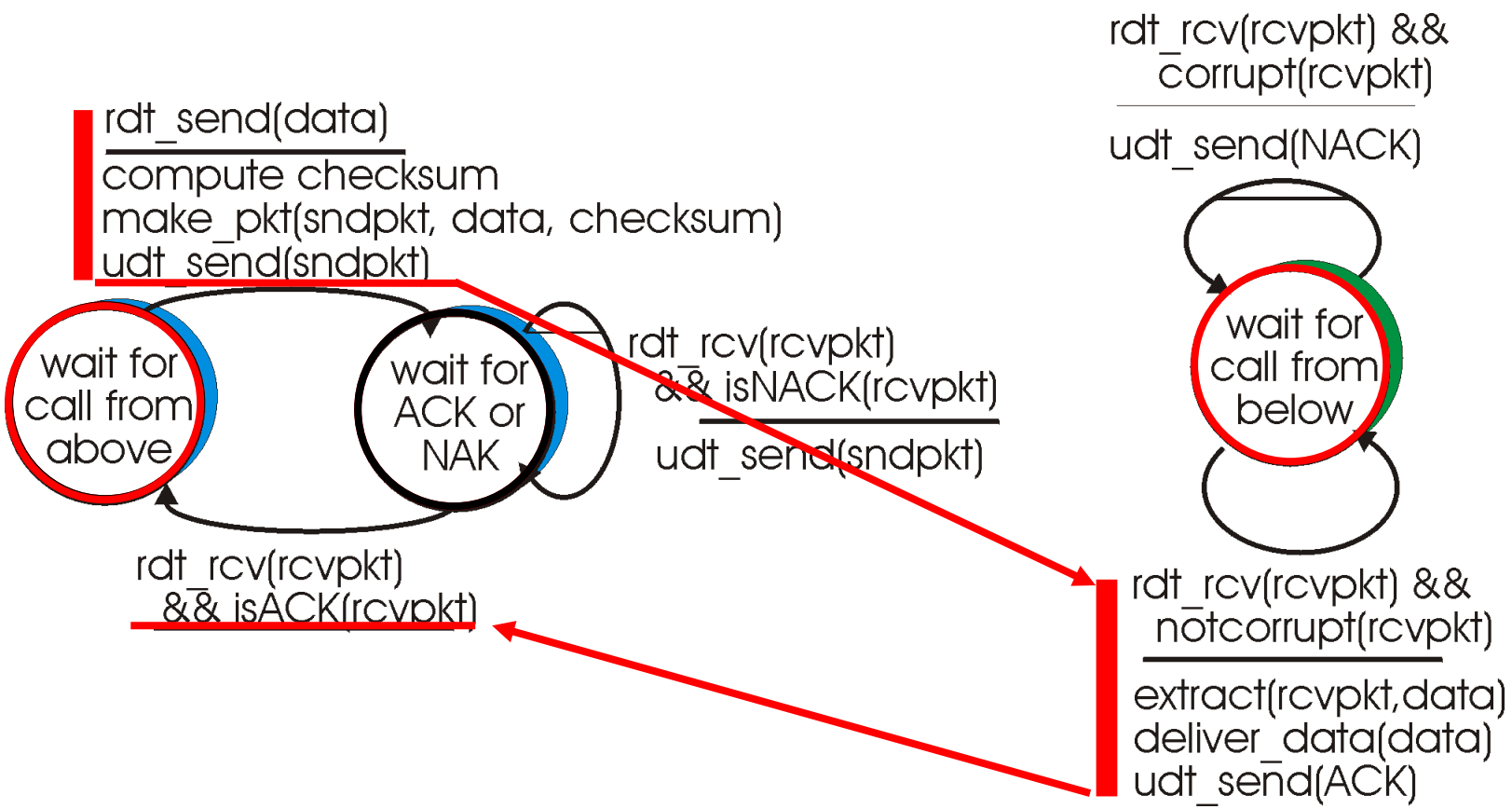
rdt2.0: especificação da FSM



FSM do remetente

FSM do receptor

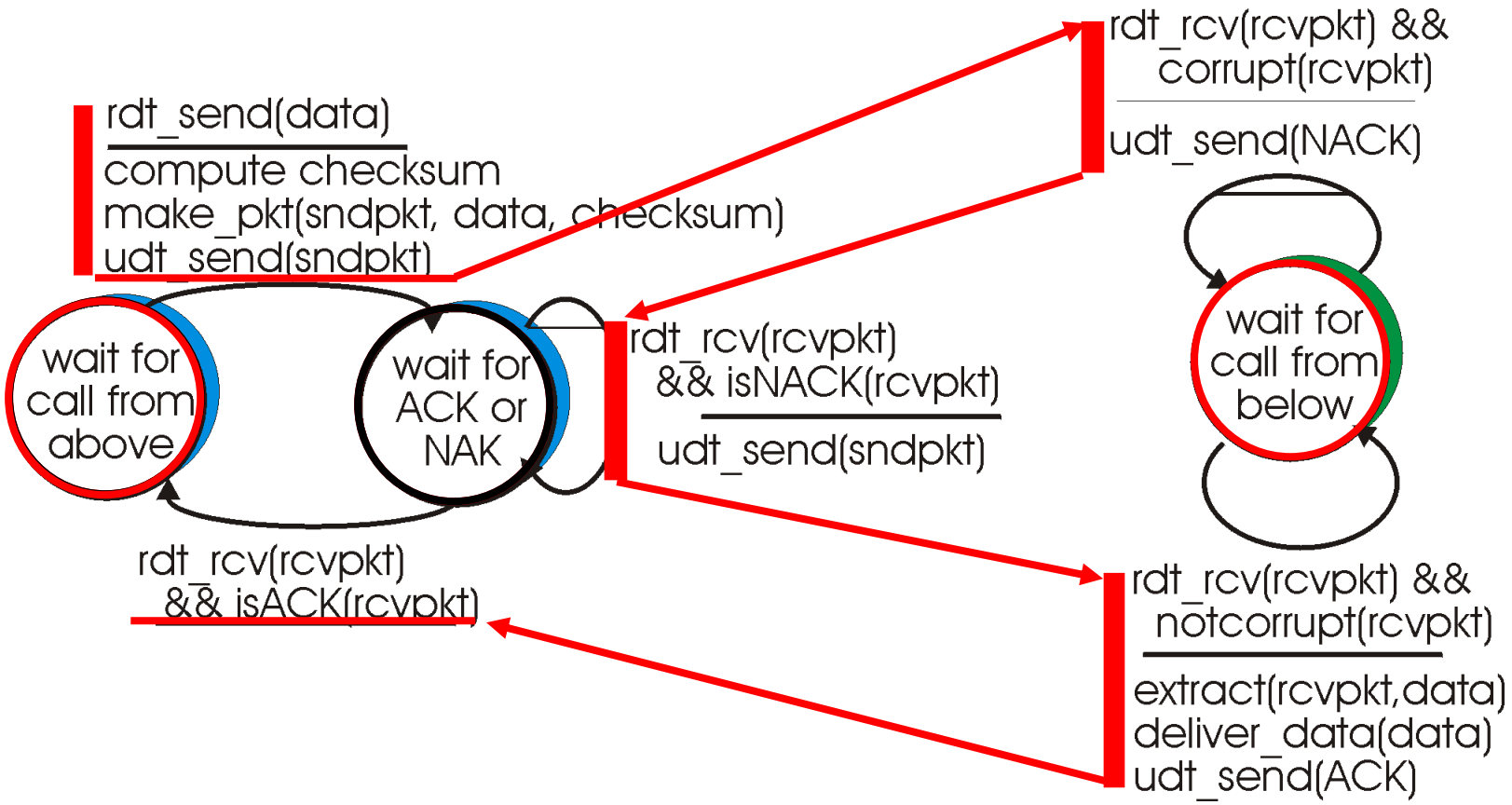
rdt2.0: em ação (sem erros)



FSM do remetente

FSM do receptor

rdt2.0: em ação (cenário de erro)



FSM do remetente

FSM do receptor

rdt2.0 tem uma falha fatal!

O que acontece se ACK/NAK com erro?

- Remetente não sabe o que 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!

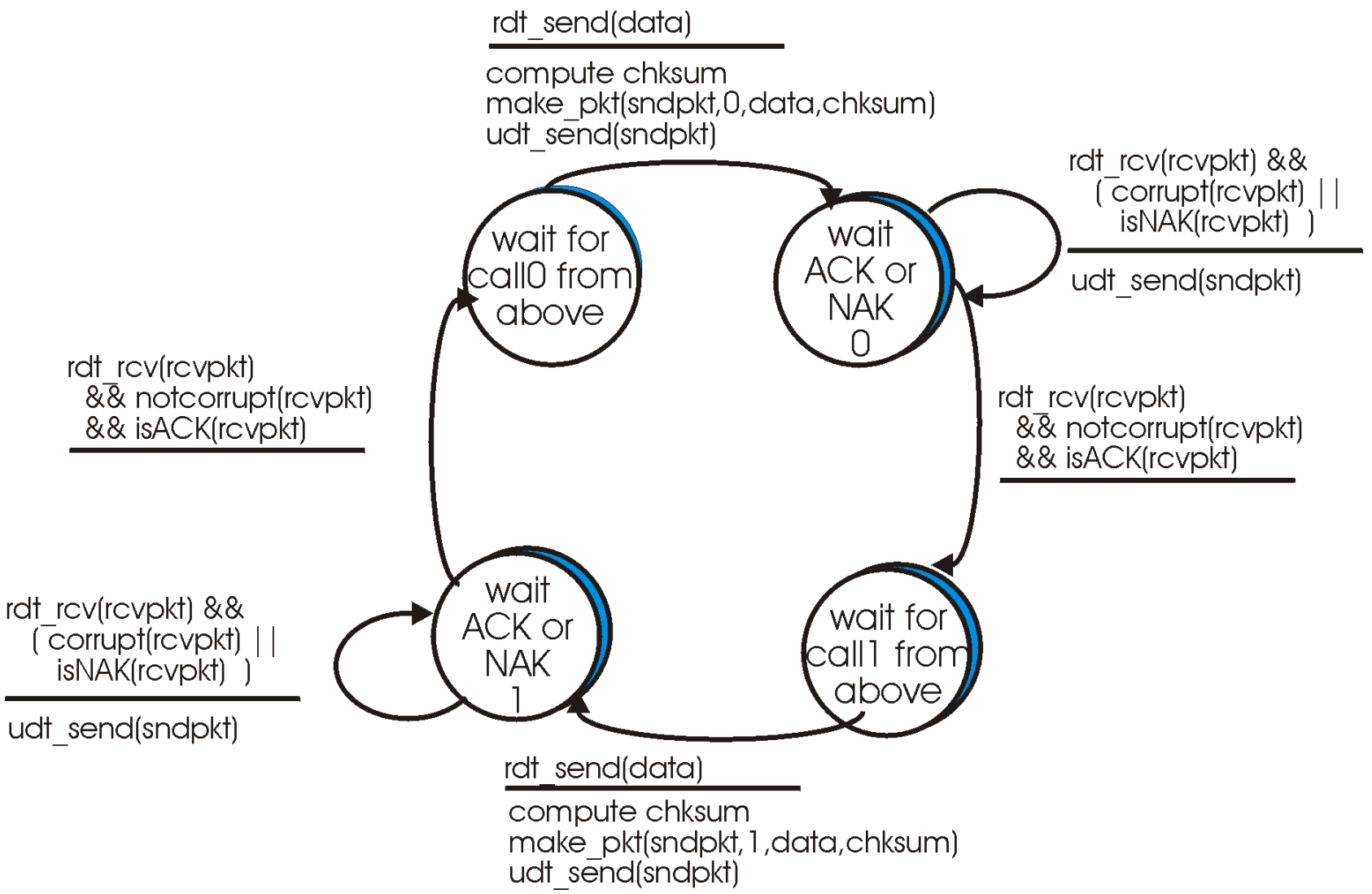
Lidando c/ duplicação:

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

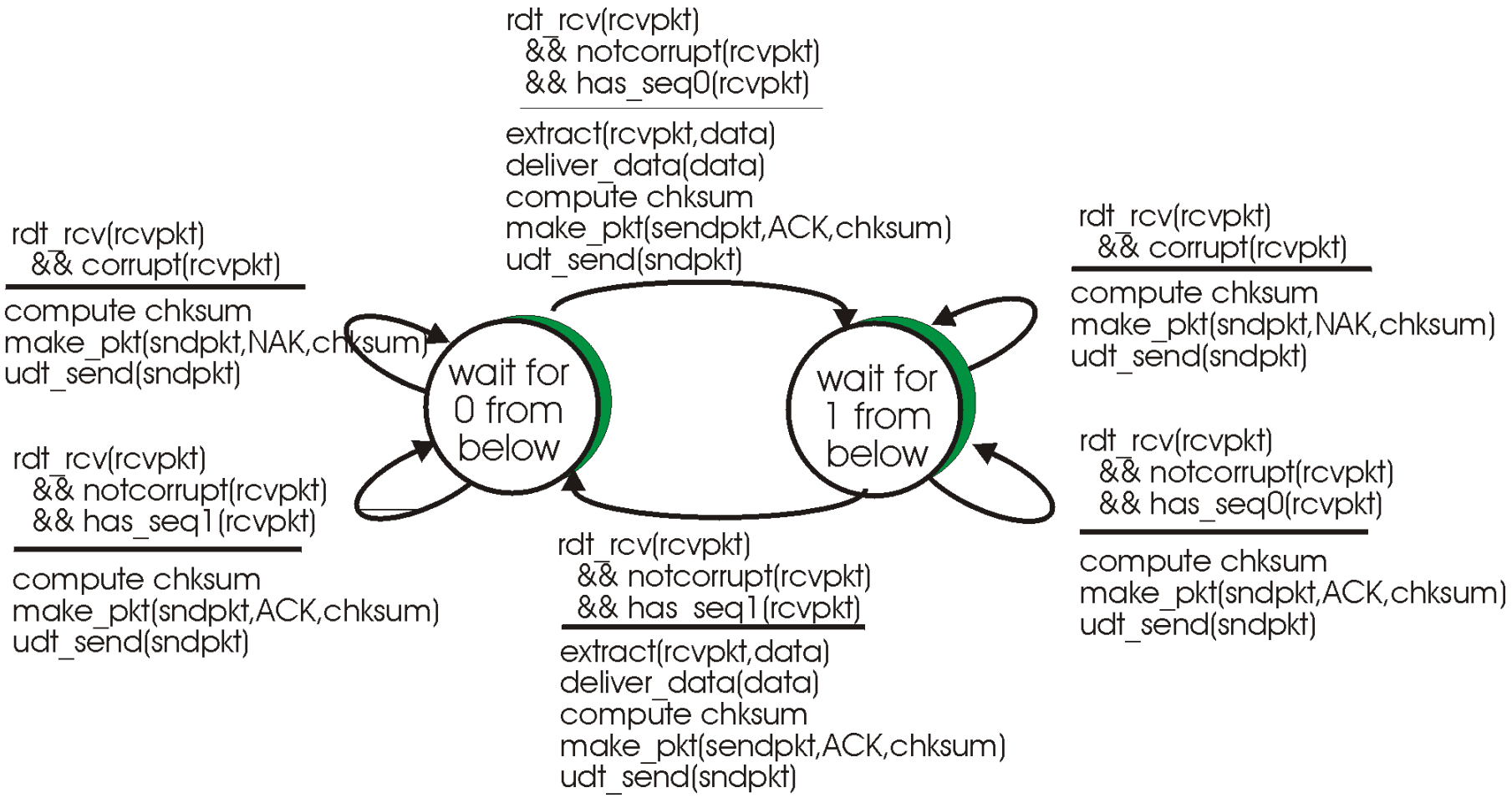
para e espera

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

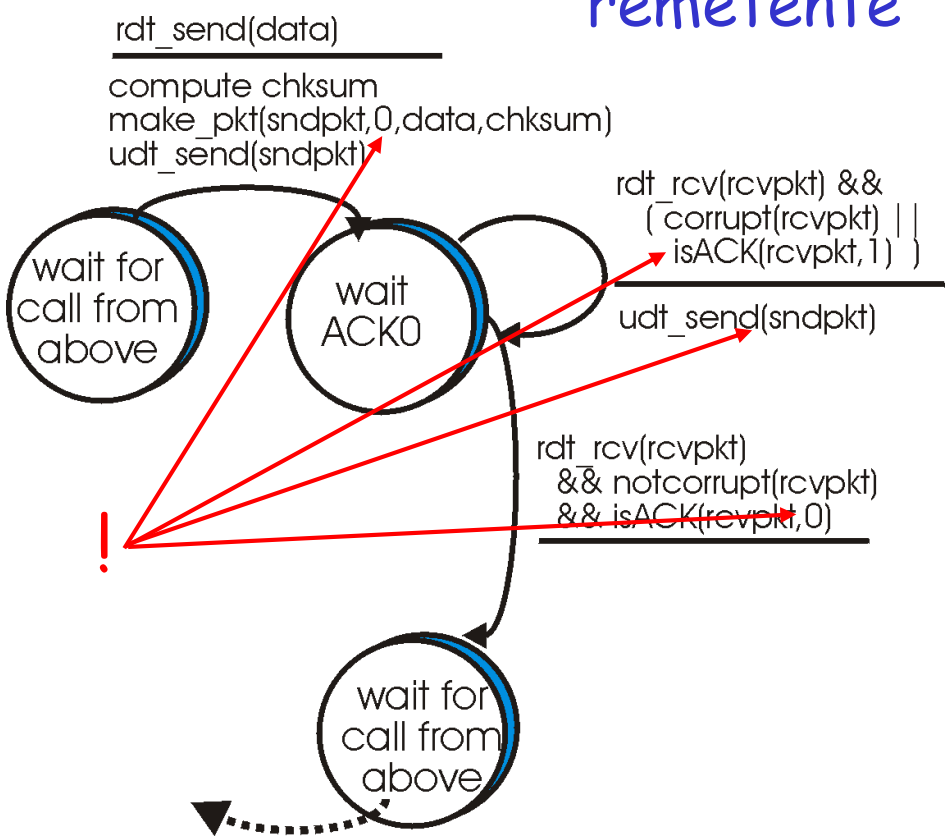
Receptor:

- 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*

FSM do remetente



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

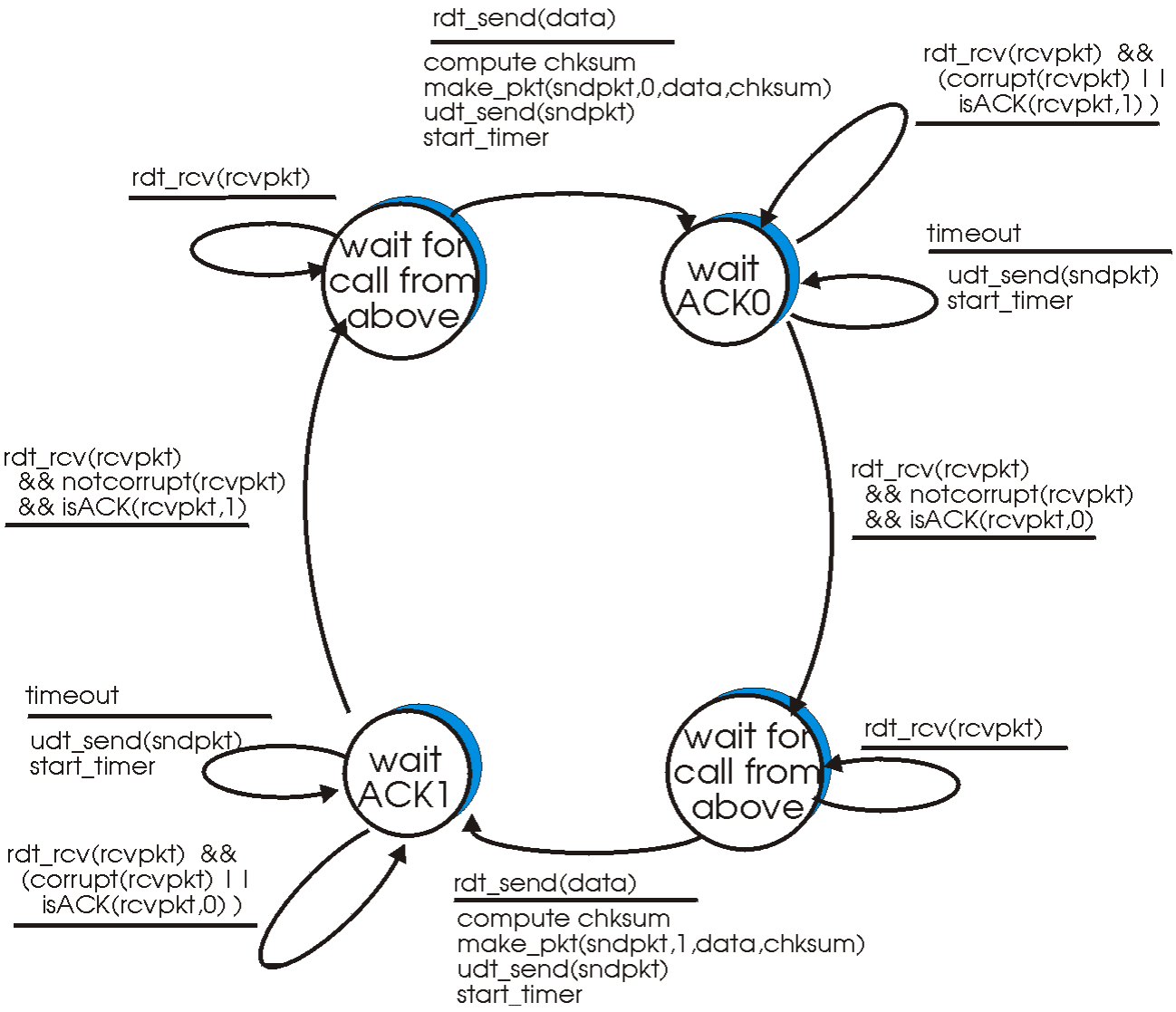
P: como lidar com perdas?

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

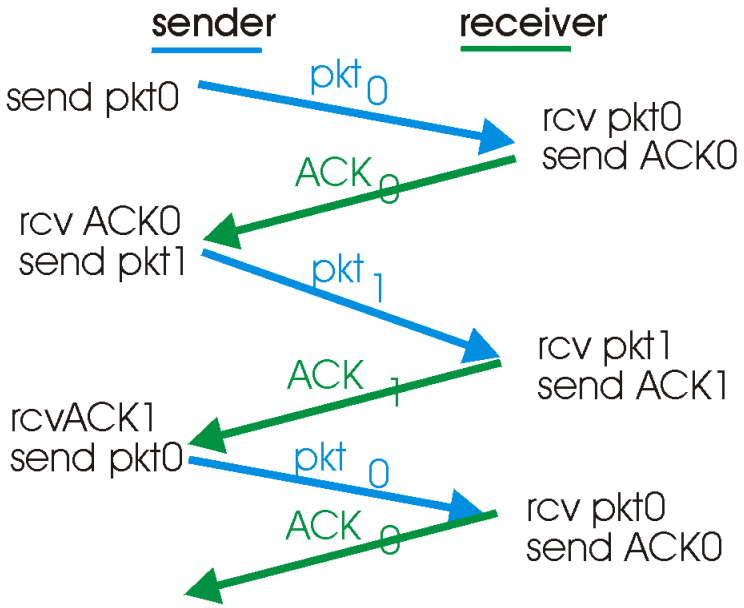
Abordagem: remetente aguarda um tempo "razoável" pelo ACK

- ➢ retransmite e nenhum ACK 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

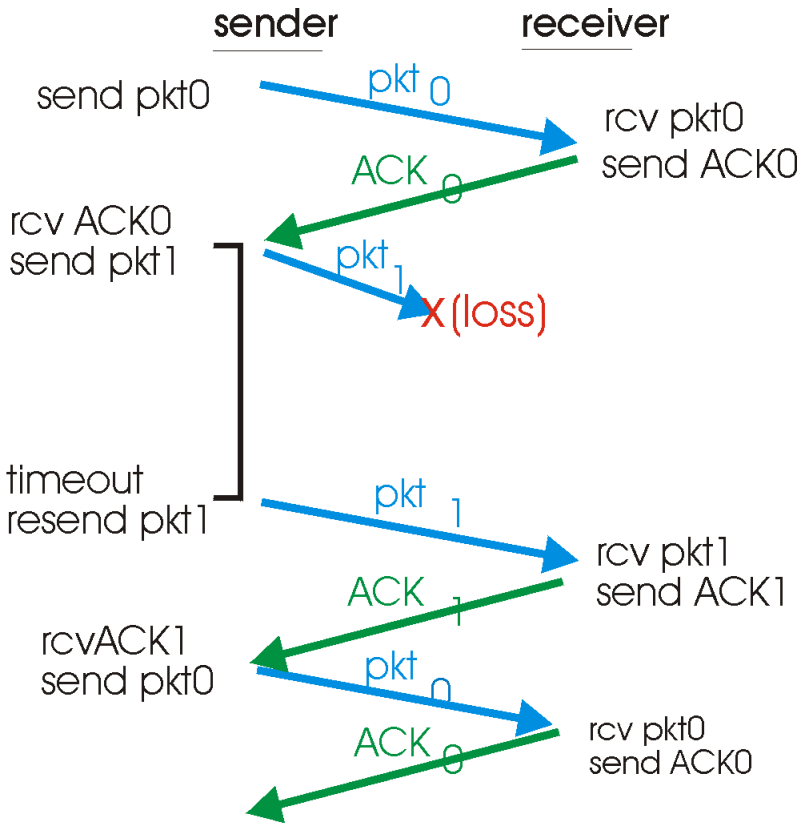
rdt3.0: remetente



rdt3.0 em ação

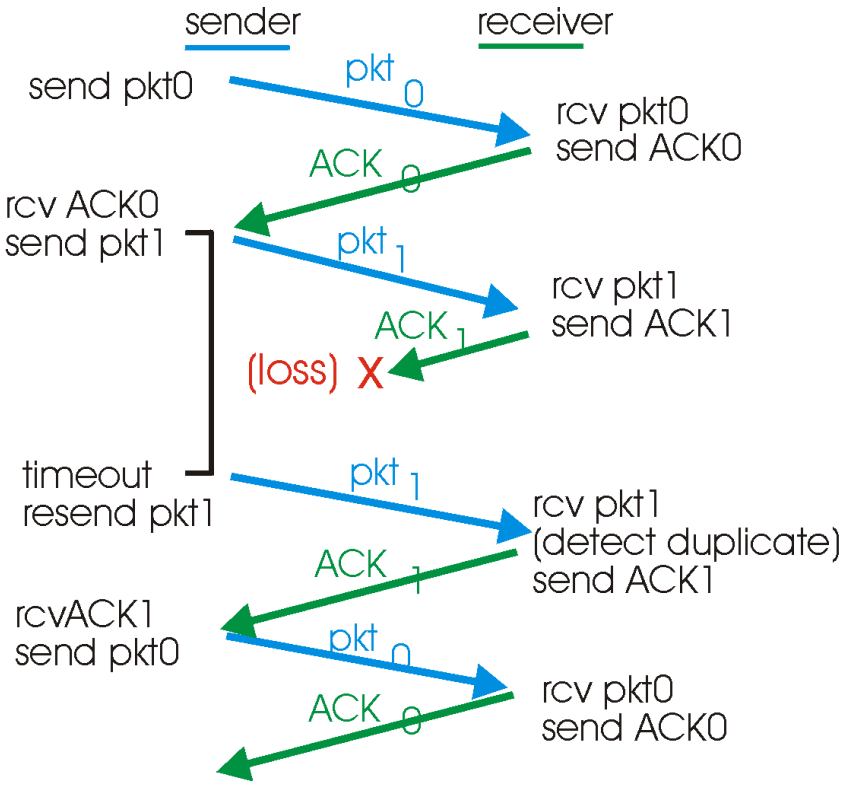


(a) operation with no loss

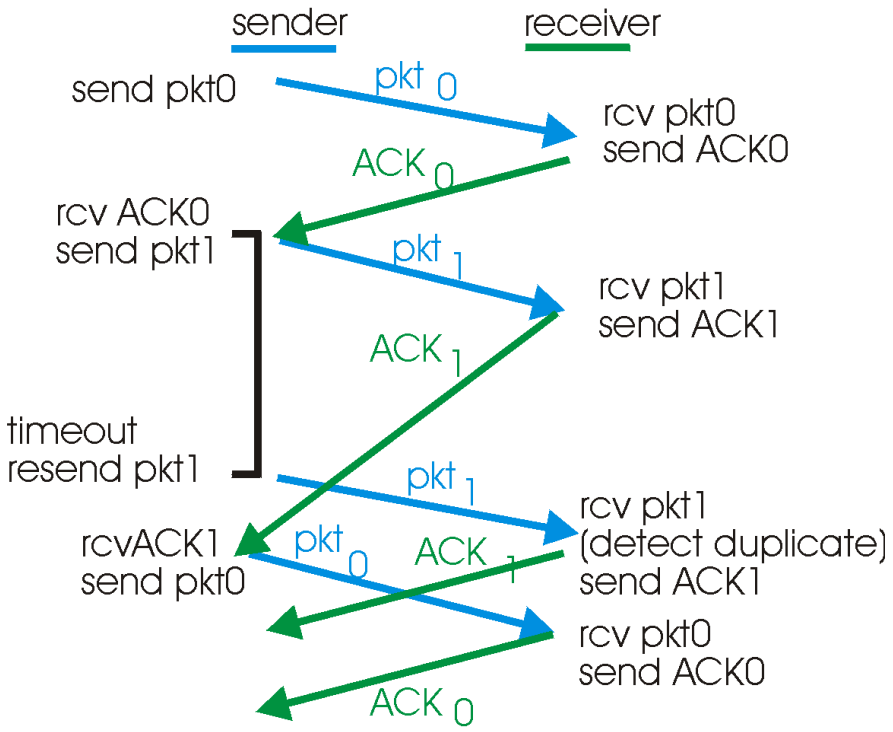


(b) lost packet

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:

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

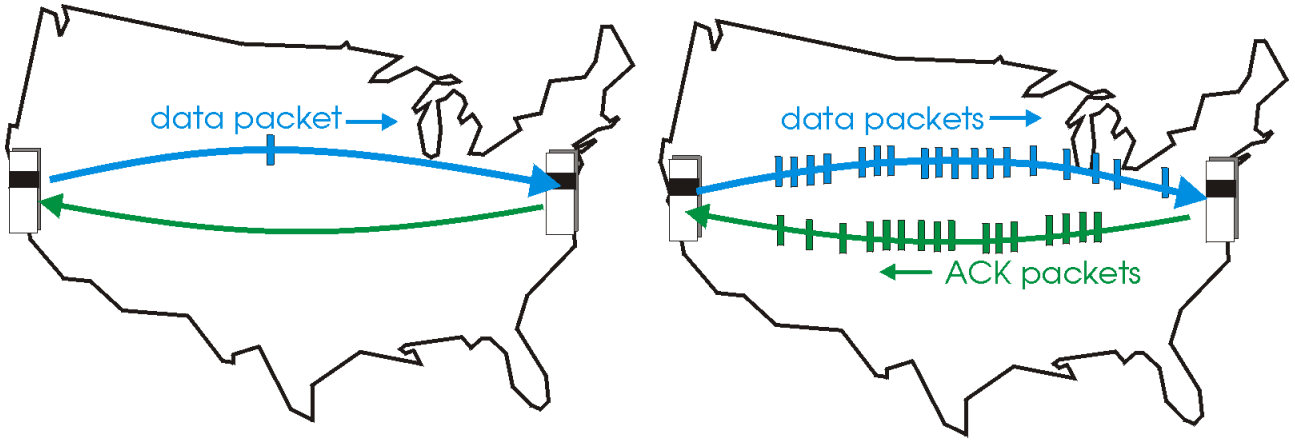
$$\text{Utilização} = U = \frac{\text{fração do tempo remetente ocupado}}{} = \frac{8\ microseg}{30.016\ mseg} = 0{,}00015$$

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

Protocolos "dutados" (*pipelined*)

Dutagem (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

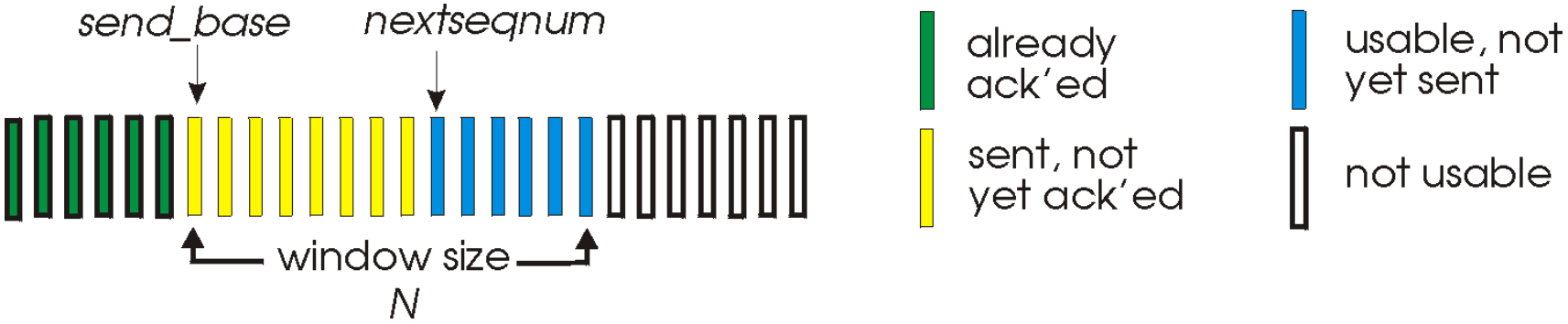
(b) a pipelined protocol in operation

- ➢ Duas formas genéricas de protocolos dutados: *volta-N, retransmissão seletiva*

Volta-N

Remetente:

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

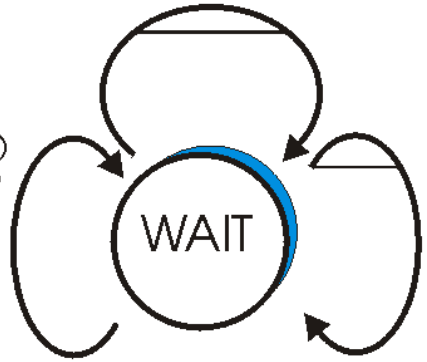


- ACK(n): reconhece todos pacotes, até e inclusive no. de seq n - "ACK cumulativo"
  - ✓ pode receber ACKs duplicados (veja receptor)
- temporizador para cada pacote em trânsito
- *timeout(n):* retransmite pacote n e todos os pacotes com no. de seq maiores na janela

Volta-N: 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)
```



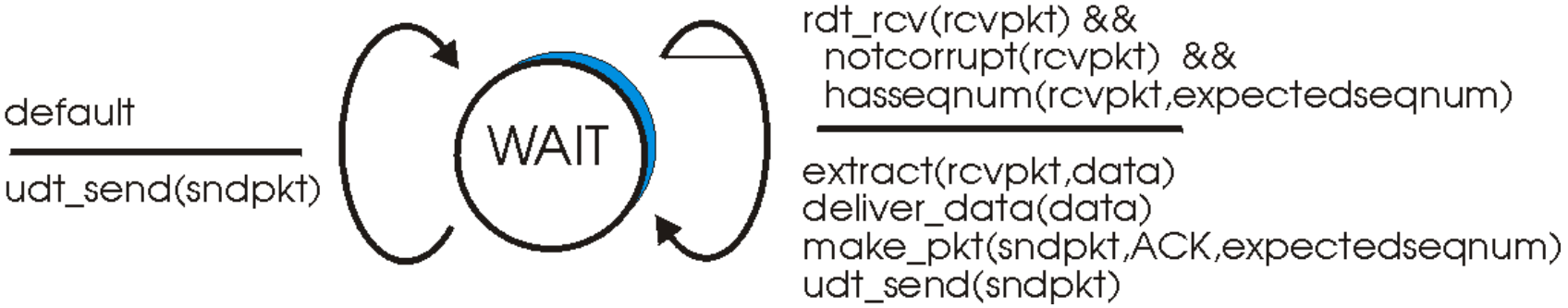
rdt_rcv(rcv_pkt) && notcorrupt(rcvpkt)

```
base = getacknum(rvcpkt)+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))
```

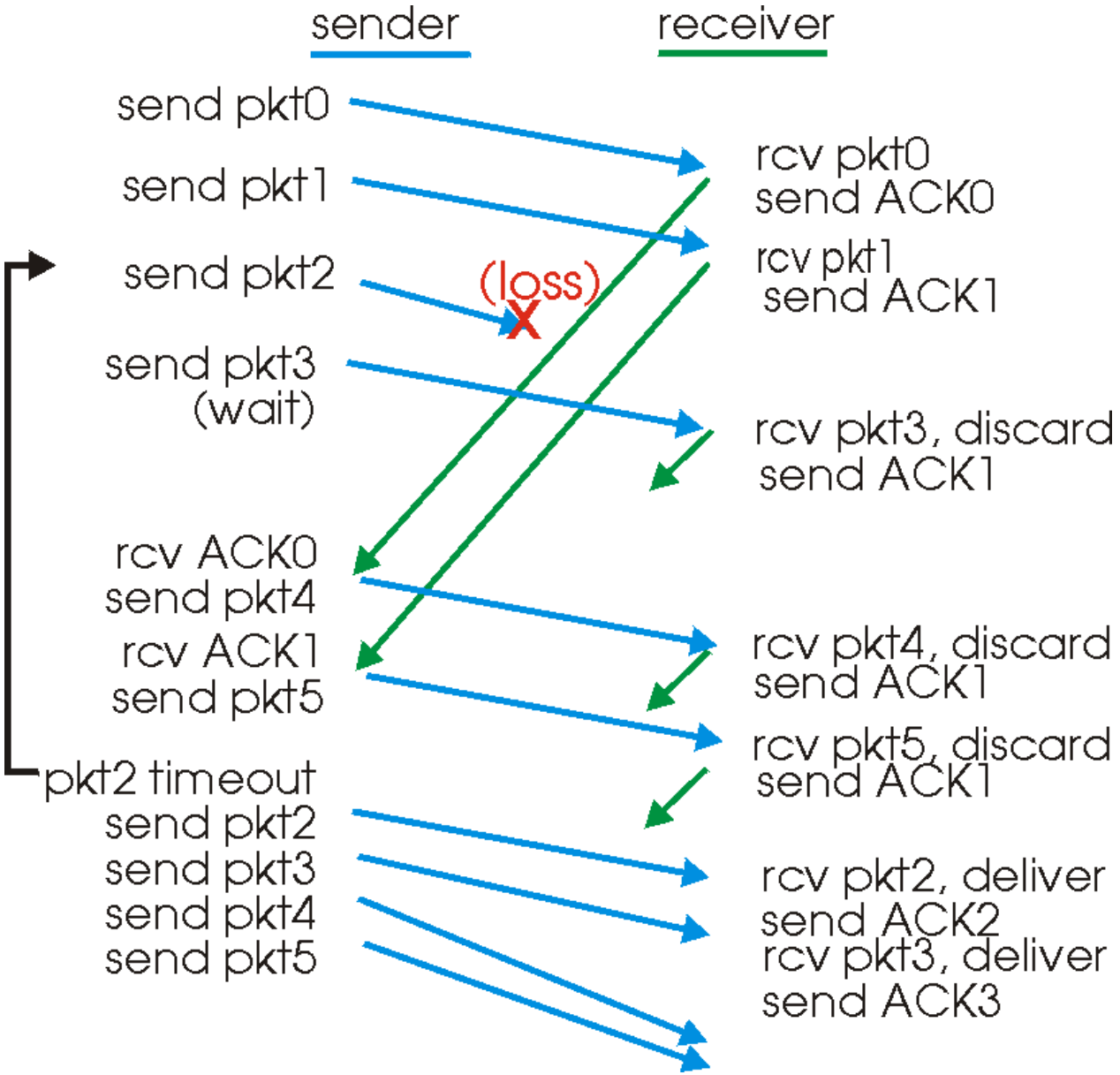
Volta-N: FSM estendida do receptor



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

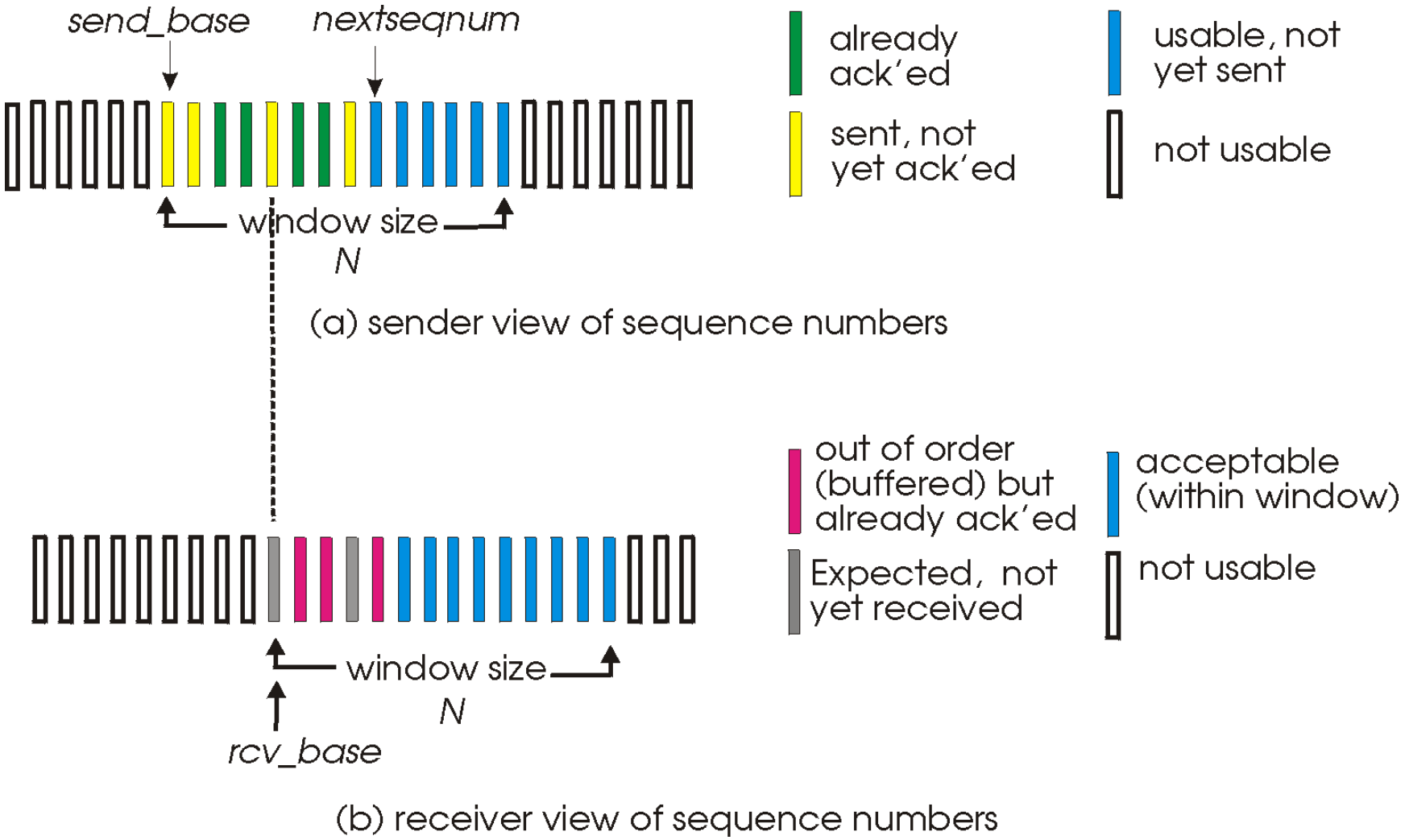
Volta-N em ação



Retransmissão seletiva

- receptor reconhece *individualmente* todos os pacotes recebidos corretamente
  - armazena pacotes no buffer, conforme precisa, 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

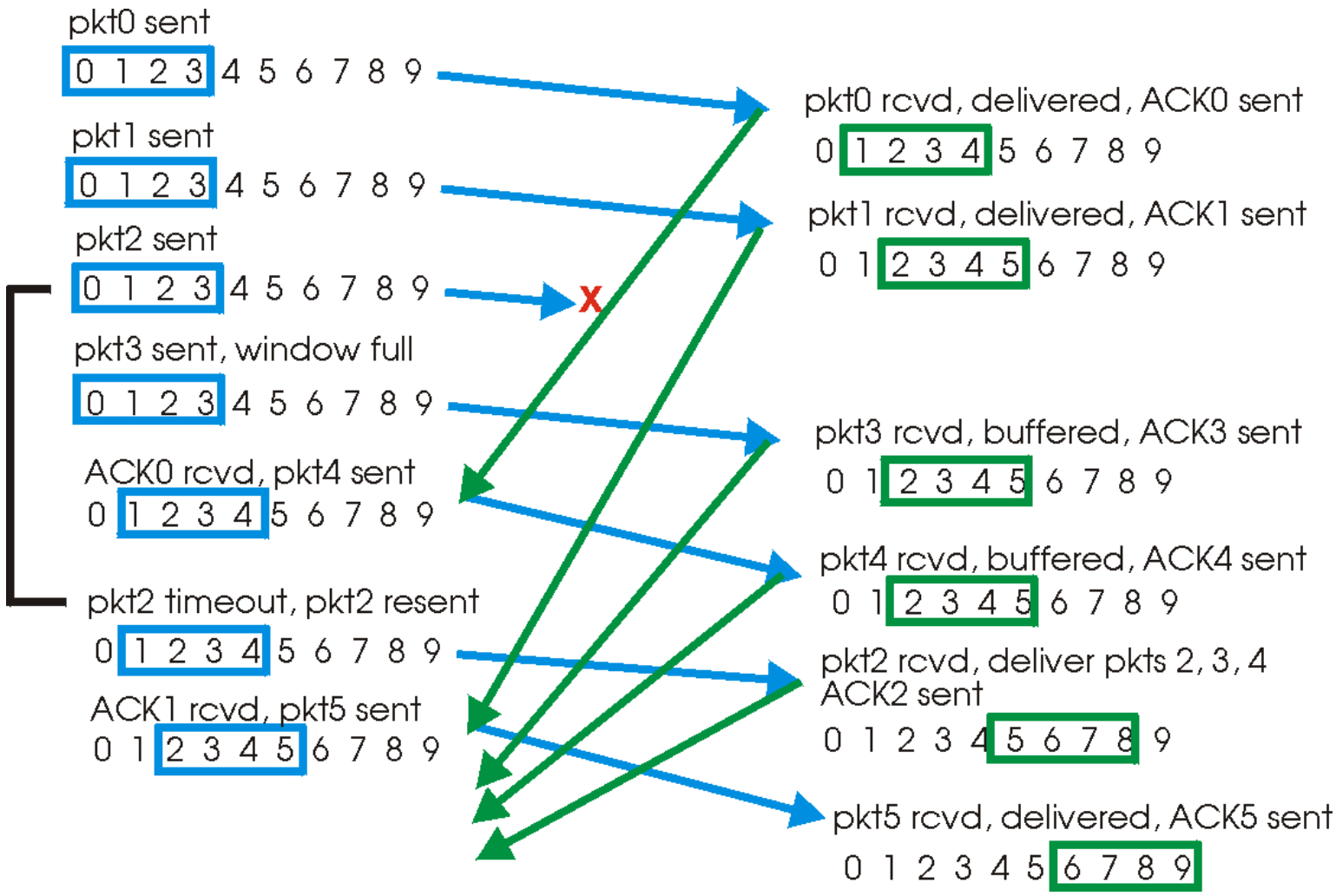
pacote n em [rcvbase-N,rcvbase-1]

- 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?