Redes de Computadores I

Prof. Ronaldo T. Oikawa

Camada de Transporte

Capítulo 3: Camada de Transporte

Objetivos do Capítulo:

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

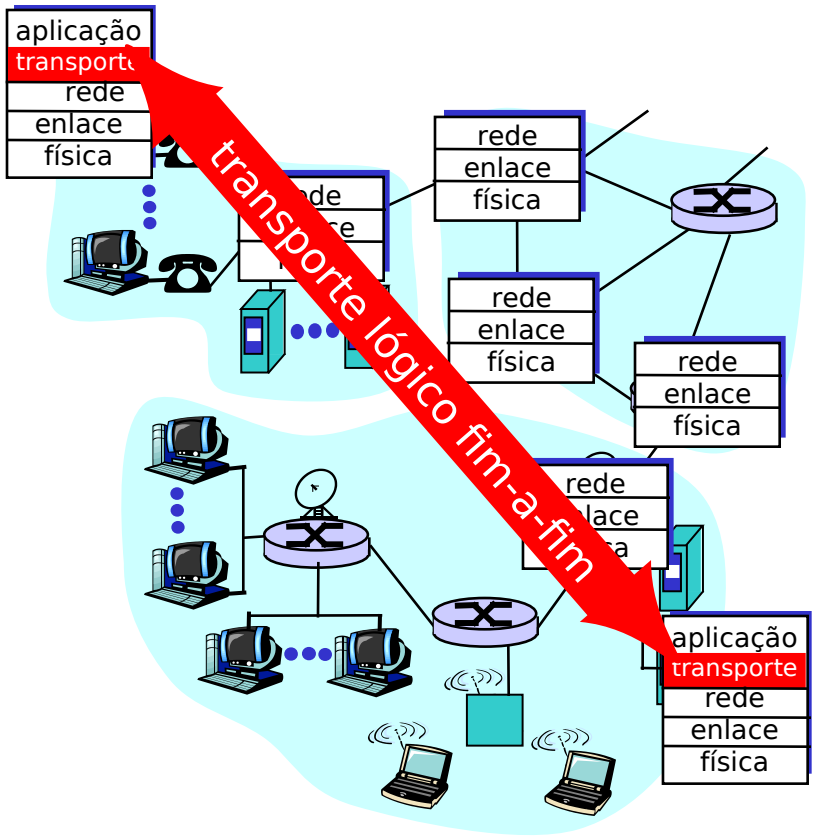
Resumo do Capítulo:

- serviços da camada de transporte
- multiplexação/demultiplexação
- transporte sem conexão: UDP
- princípios de transferência confiável de dados
- transporte orientado à conexão: TCP
  - transferência confiável
  - controle de fluxo
  - gerenciamento de conexão
- princípios de controle de congestionamento
- controle de congesetionamento do TCP

Protocolos e Serviços de Transporte

- Fornecem comunicação lógicas entre processos de aplicação em diferentes hosts
- Os protocolos de transporte são executados nos sistemas finais da rede

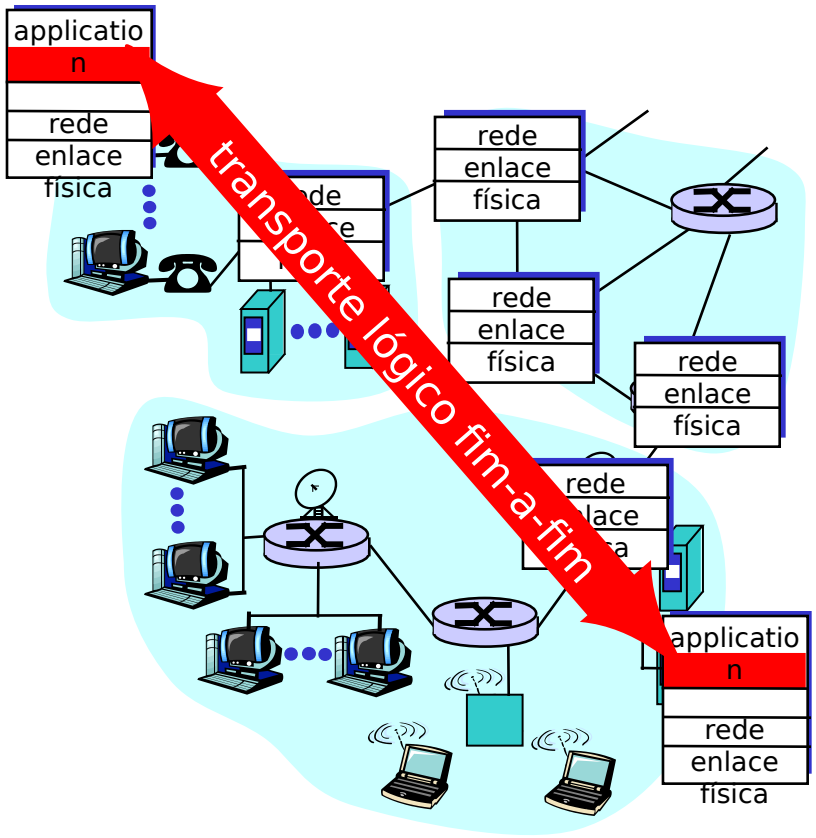
- serviço de transporte vs serviços de rede :
- *camada de rede:* transferência de dados entre computadores (end systems)
- *camada de transporte:* transferência de dados entre processos
  - utiliza e aprimora os serviços oferecidos pela camada de rede



Protocolos da Camada de Transporte

Serviços de Transporte da Internet:

- confiável, seqüencial e unicast (TCP)
  - congestão
  - controle de fluxo
  - orientado à conexão
- não confiável ("best-effort"), não seqüencial, entrega unicast or multicast : UDP
- serviços não disponíveis:
  - tempo-real
  - garantia de banda
  - multicast confiável

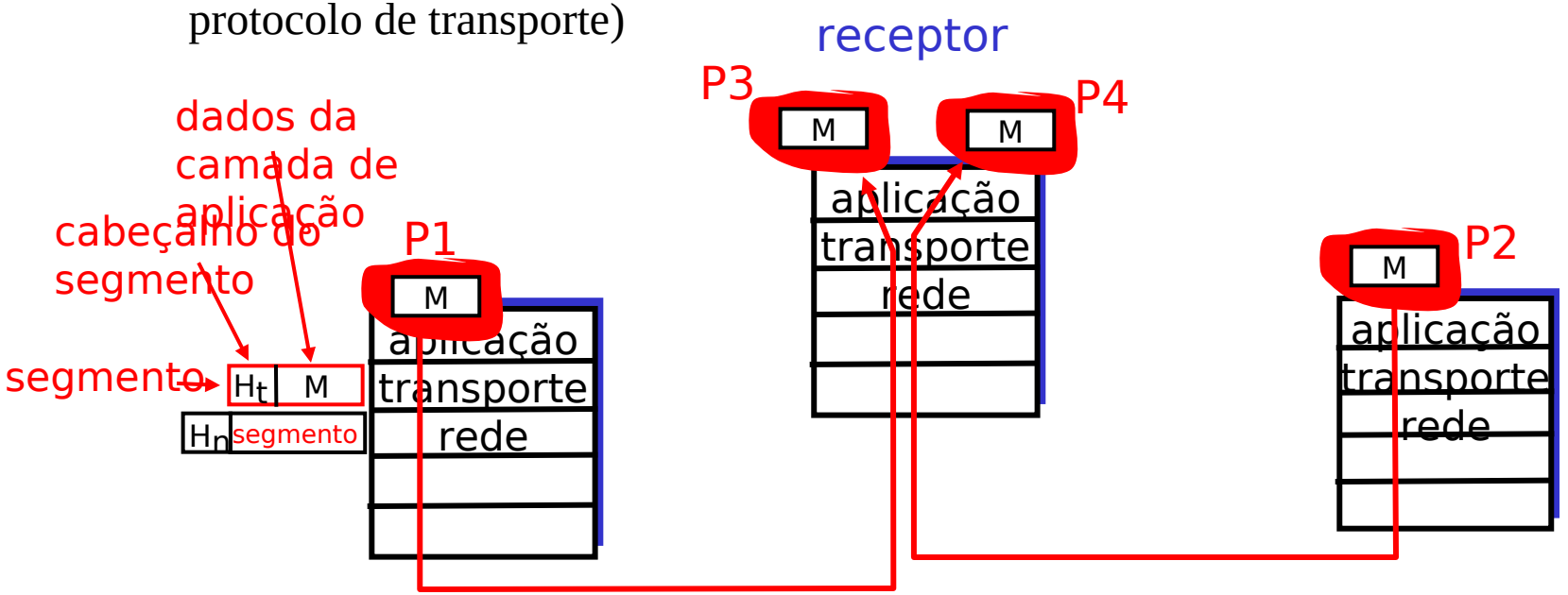


Multiplexação de Aplicações

Segmento - unidade de dados trocada entre entidades da camada de transporte

- TPDU: transport protocol data unit (unidade de dados do protocolo de transporte)

Demultiplexação: entrega de segmentos recebidos aos processos de aplicação corretos

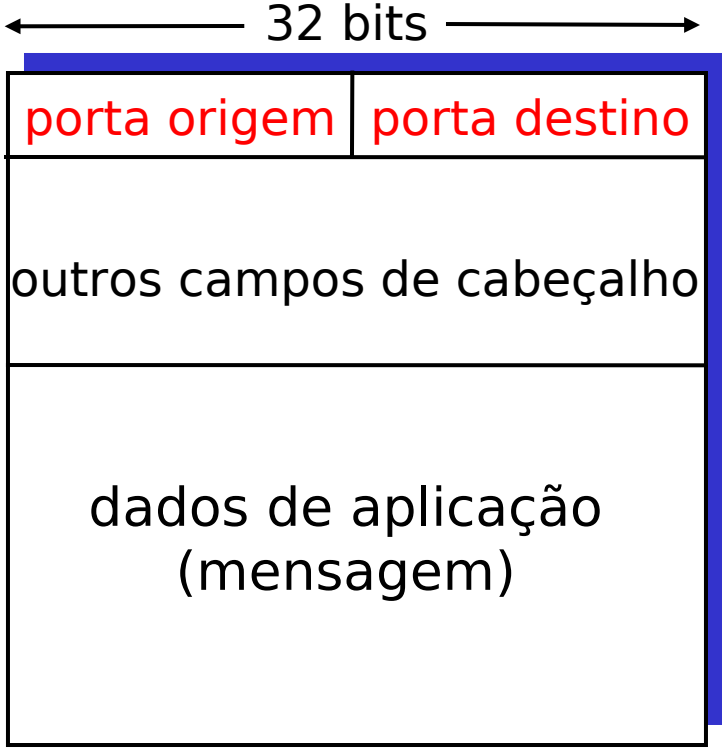


Multiplexação de Aplicações

Multiplexação:
reunir dados de múltiplos processo de aplicação, juntar cabeçalhos com informações para demultiplexação

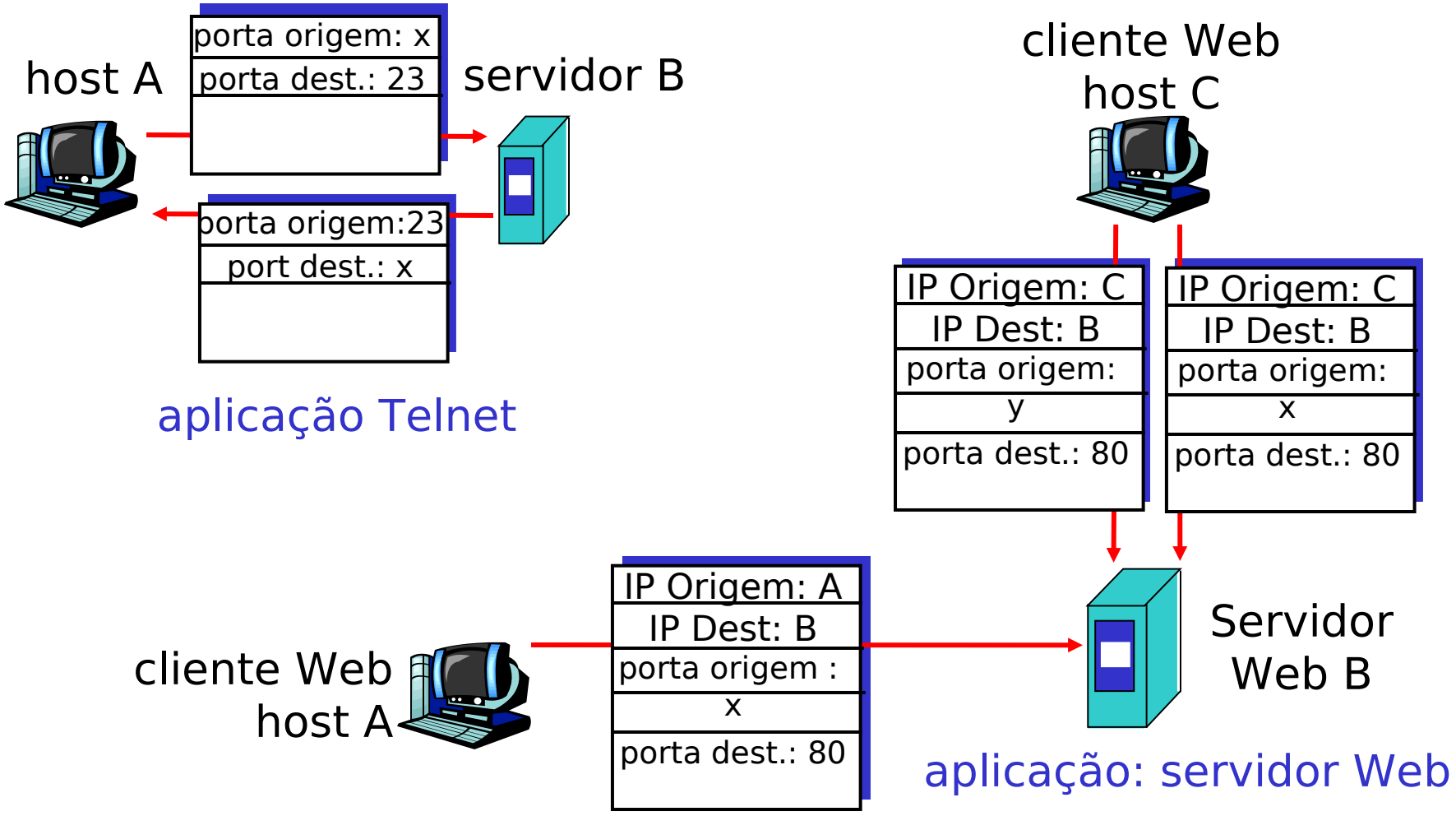
multiplexação/demultiplexação:

- baseada nos número de porta do transmissor, número de porta do receptor e endereços IP
  - números de porta origem e destino em cada segmento
  - lembre: portas com números bem-conhecidos são usadas para aplicações específicas



formato do segmento TCP/UDP

Multiplexação: exemplos



UDP: User Datagram Protocol [RFC 768]

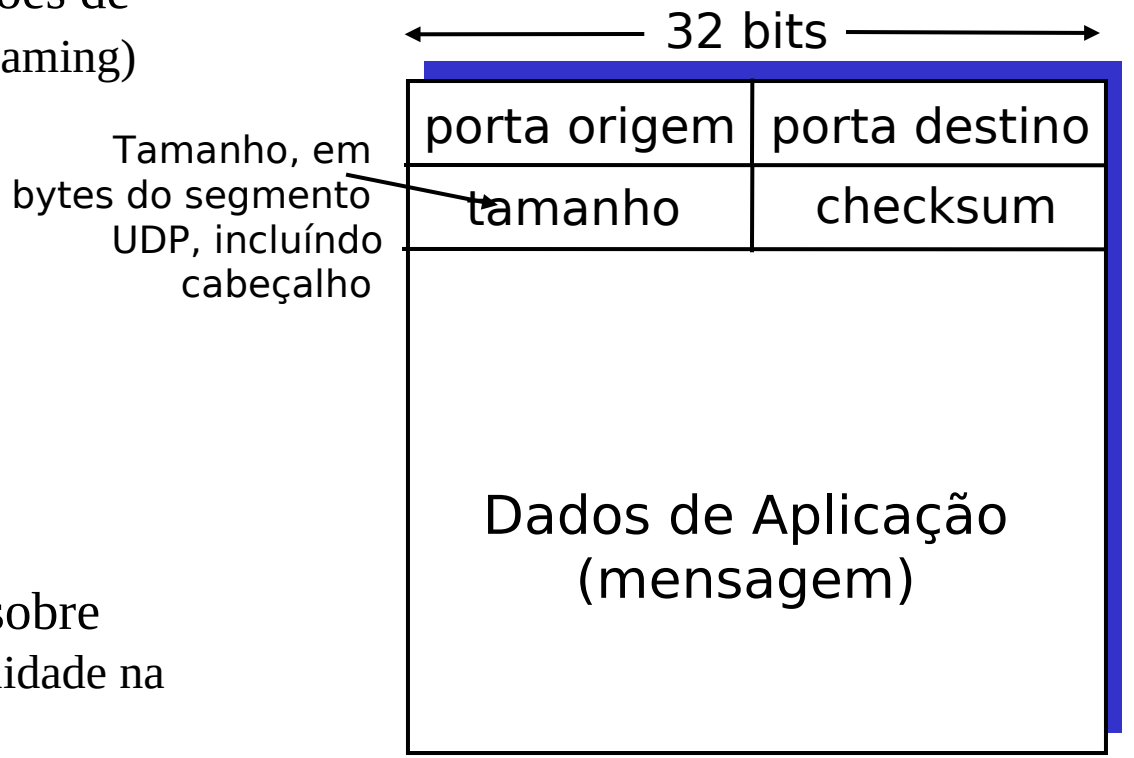
- protocolo de transporte da Internet “sem gorduras” “sem frescuras”
- serviço “best effort” , segmentos UDP podem ser:
  - perdidos
  - entregues fora de ordem para a aplicação
- *sem conexão:*
  - não há apresentação entre o UDP transmissor e o receptor
  - cada segmento UDP é tratado de forma independente dos outros

Porque existe um UDP?

- não há estabelecimento de conexão (que pode redundar em atrasos)
- simples: não há estado de conexão nem no transmissor, nem no receptor
- cabeçalho de segmento reduzido
- não há controle de congestionamento: UDP pode enviar segmentos tão rápido quanto desejado (e possível)

Mais sobre UDP

- muito usado por aplicações de mutimídia contínua (streaming)
  - tolerantes à perda
  - sensíveis à taxa
- outros usos do UDP (porque?):
  - DNS
  - SNMP
- transferência confiável sobre UDP: acrescentar confiabilidade na camada de aplicação
  - recuperação de erro específica de cada aplicação



formato do segmento UDP

UDP checksum

Objetivo: detectar “erros” (ex.,bits trocados) no segmento transmitido

Transmissor:

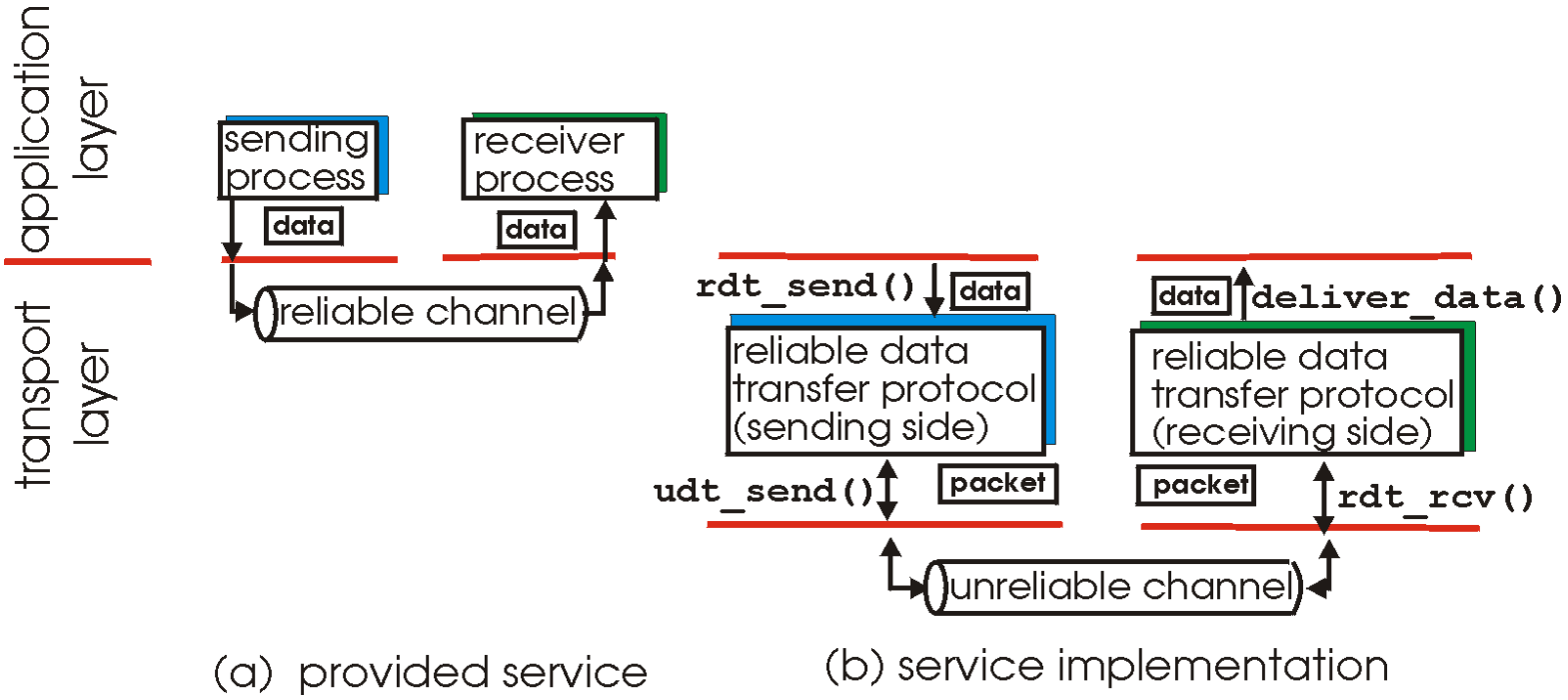
- trata o conteúdo do segmento como seqüencia de inteiros de 16 bits
- checksum: soma (complemento de 1 da soma) do conteúdo do segmento
- transmissor coloca o valor do checksum no campo de checksum do UDP

Receptor:

- computa o checksum do segmento recebido
- verifica se o checksum calculado é igual ao valor do campo checksum:
  - NÃO - error detectado
  - SIM - não há erros. *Mas, talvez haja erros apesar disto?* Mais depois ….

Princípios de Transferência Confiável de Dados

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

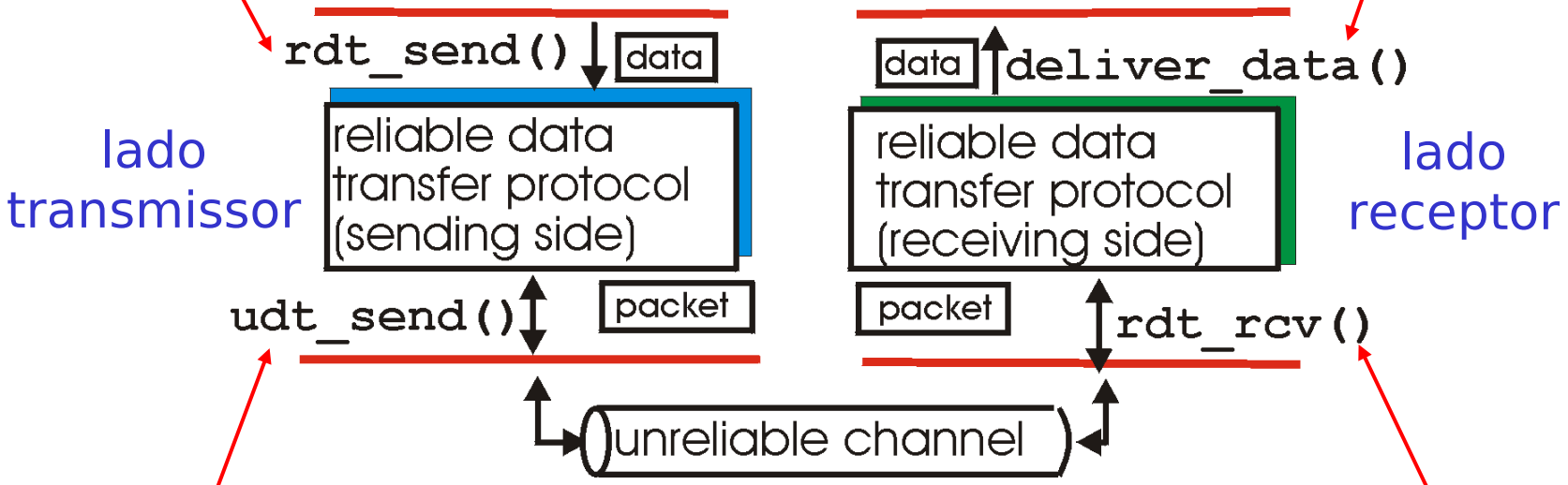


- caracteristicas dos canais não confiáveis determinarão a complexidade dos protocolos confiáveis de transferência de dados (rdt)

Transferência confiável: o ponto de partida

`rdt_send()`: chamada da camada superior, (ex., pela aplicação). Passa dados para entregar à camada superior receptora

`deliver_data()`: chamada pela entidade de transporte para entregar dados para cima



`udt_send()`: chamada pela entidade de transporte, para transferir pacotes para o receptor sobre o canal não confiável

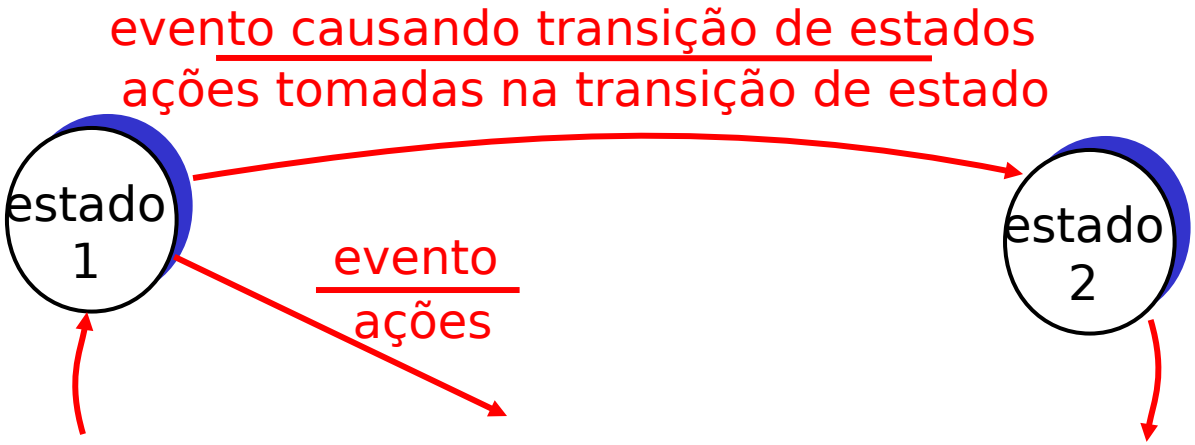
`rdt_rcv()`: chamada quando o pacote chega ao lado receptor do canal

Transferência confiável: o ponto de partida

Etapas:

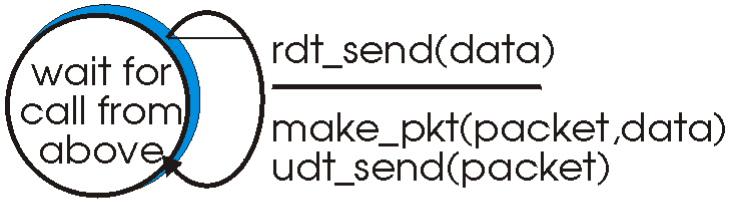
- desenvolver incrementalmente o transmissor e o receptor de um protocolo confiável de transferência de dados (rdt)
- considerar apenas transferências de dados unidirecionais
  - mas informação de controle deve fluir em ambas as direções!
- usar máquinas de estados finitos (FSM) para especificar o protocolo transmissor e o receptor

estado: quando neste "estado" o próximo estado fica unicamente determinado pelo próximo evento



Rdt1.0: transferência confiável sobre canais confiáveis

- canal de transmissão perfeitamente confiável
  - não há erros de bits
  - não há perdas de pacotes
- FSMs separadas para transmissor e receptor:
  - transmissor envia dados para o canal subjacente
  - receptor lê os dados do canal subjacente



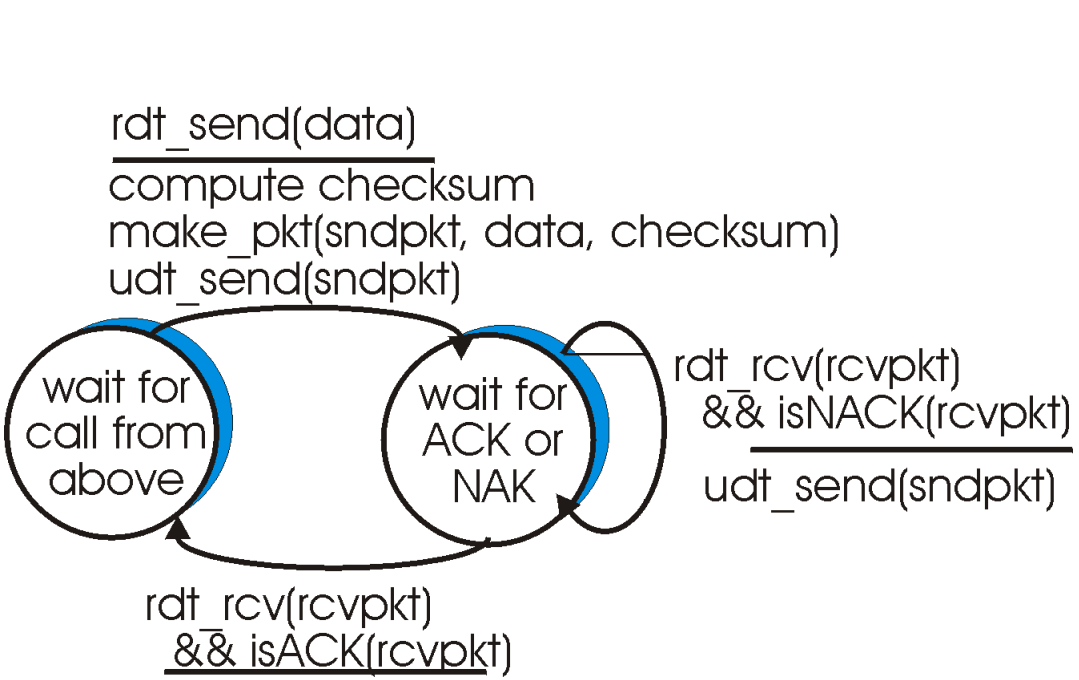
(a) rdt1.0: sending side

(b) rdt1.0: receiving side

Rdt2.0: canal com erros de bit

- canal subjacente pode trocar valores dos bits num pacote
  - lembrete: checksum do UDP pode detectar erros de bits
- *a* questão: como recuperar esses erros:
  - *reconhecimentos (ACKs):* receptor avisa explicitamente ao transmissor que o pacote foi recebido corretamente
  - *reconhecimentos negativos (NAKs):* receptor avisa explicitamente ao transmissor que o pacote tem erros
  - transmissor reenvia o pacote quando da recepção de um NAK
  - cenários humanos usando ACKs, NAKs?
- novos mecanismos no **`rdt2.0`** (além do **`rdt1.0`**):
  - deteção de erros
  - retorno do receptor: mensagens de controle (ACK,NAK) rcvr->sender

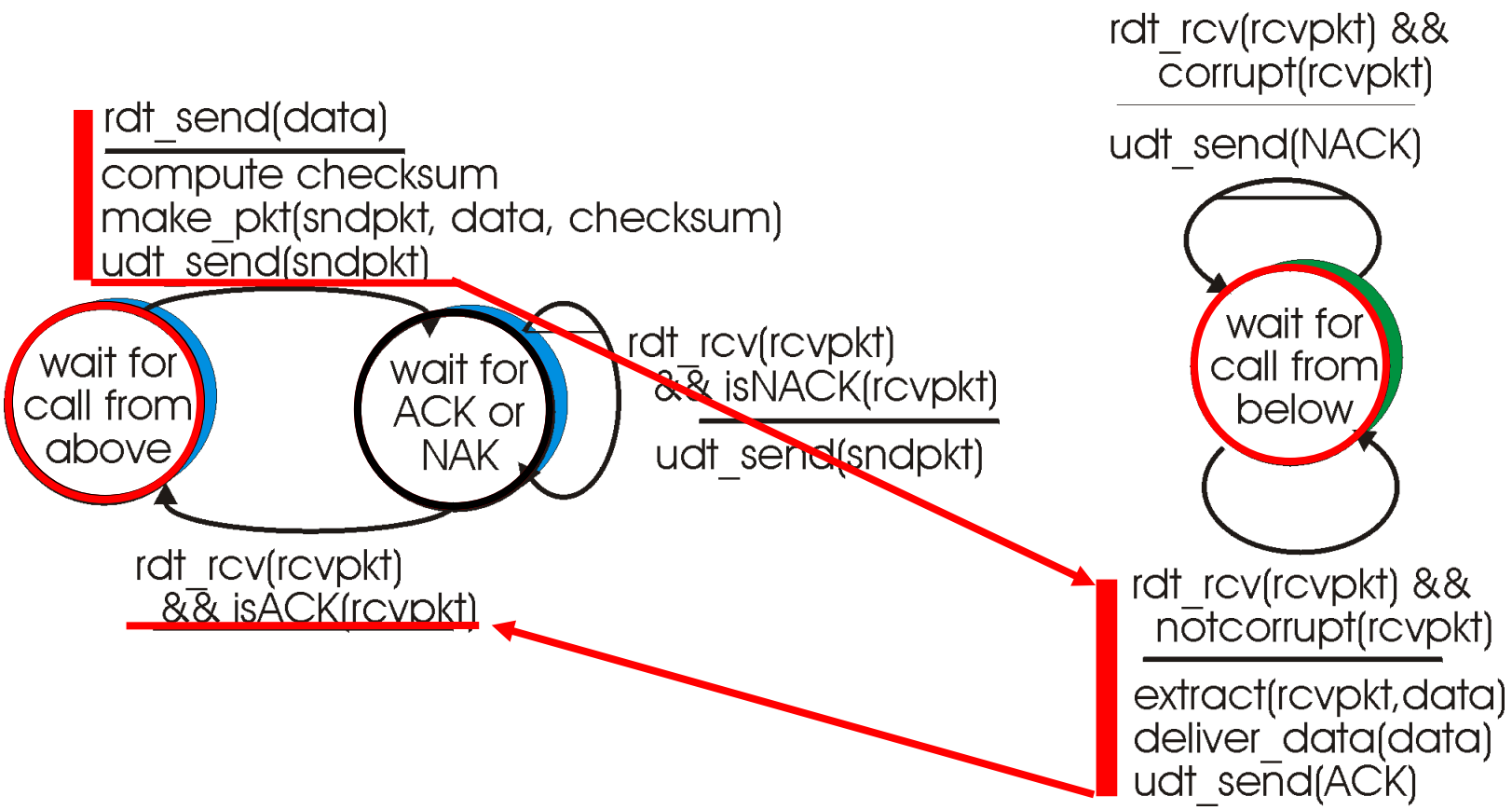
rdt2.0: especificação da FSM



FSM do transmissor

FSM do receptor

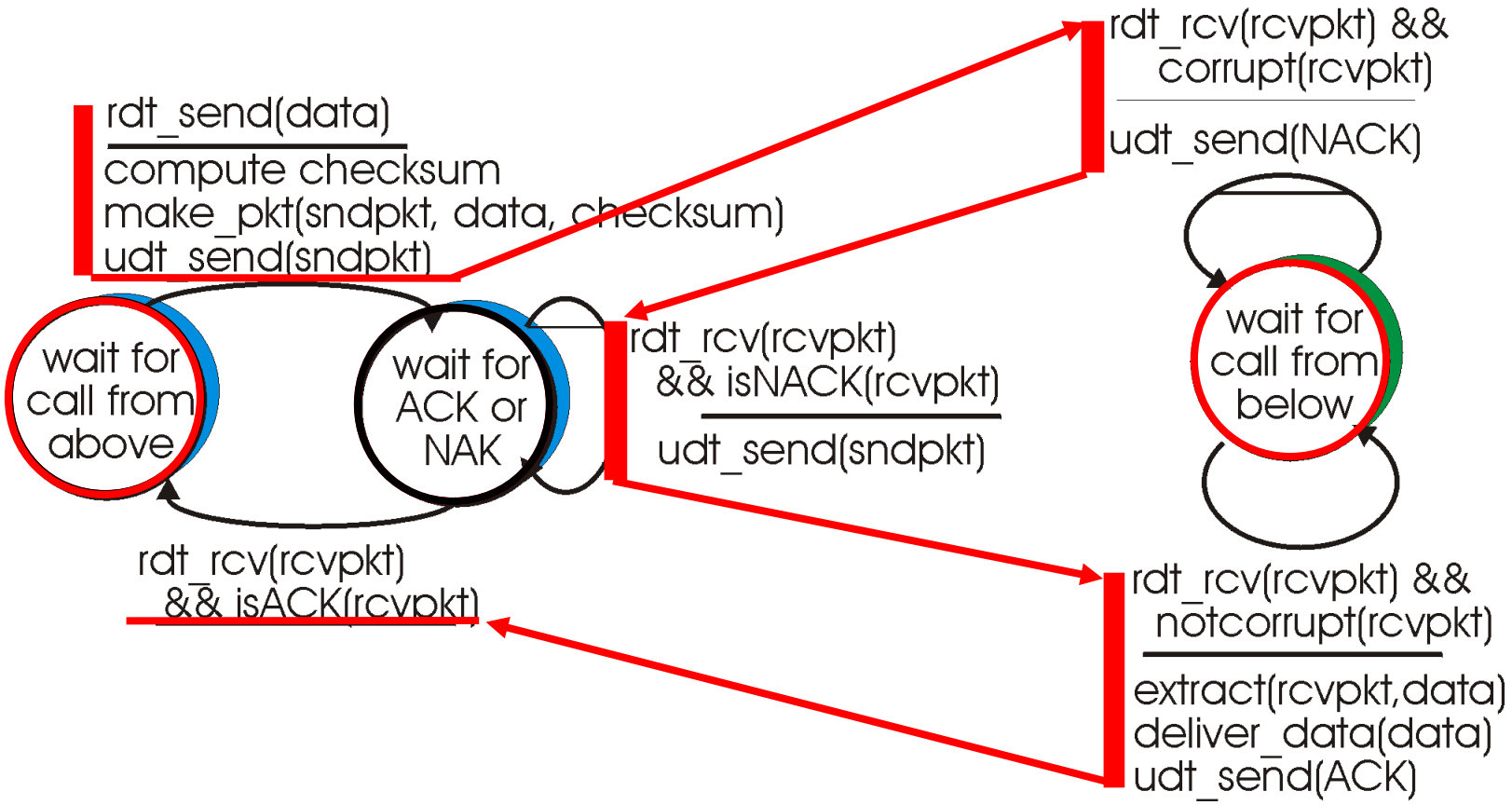
rdt2.0: em ação (ausência de erros)



FSM do transmissor

FSM do receptor

rdt2.0: em ação (cenário com erros)



FSM do transmissor

FSM do receptor

rdt2.0 tem um problema fatal!

O que acontece se o ACK/NAK é corrompido?

- transmissor não sabe o que aconteceu no receptor!
- não pode apenas retransmitir: possível duplicata

O que fazer?

- Transmissor envia ACKs/NAKs para reconhecer os ACK/NAK do receptor? O que acontece se estes ACK/NAK se perdem?
- retransmitir os ACK/NAK, mas isto poderia causar a retransmissão de um pacote recebido corretamente!

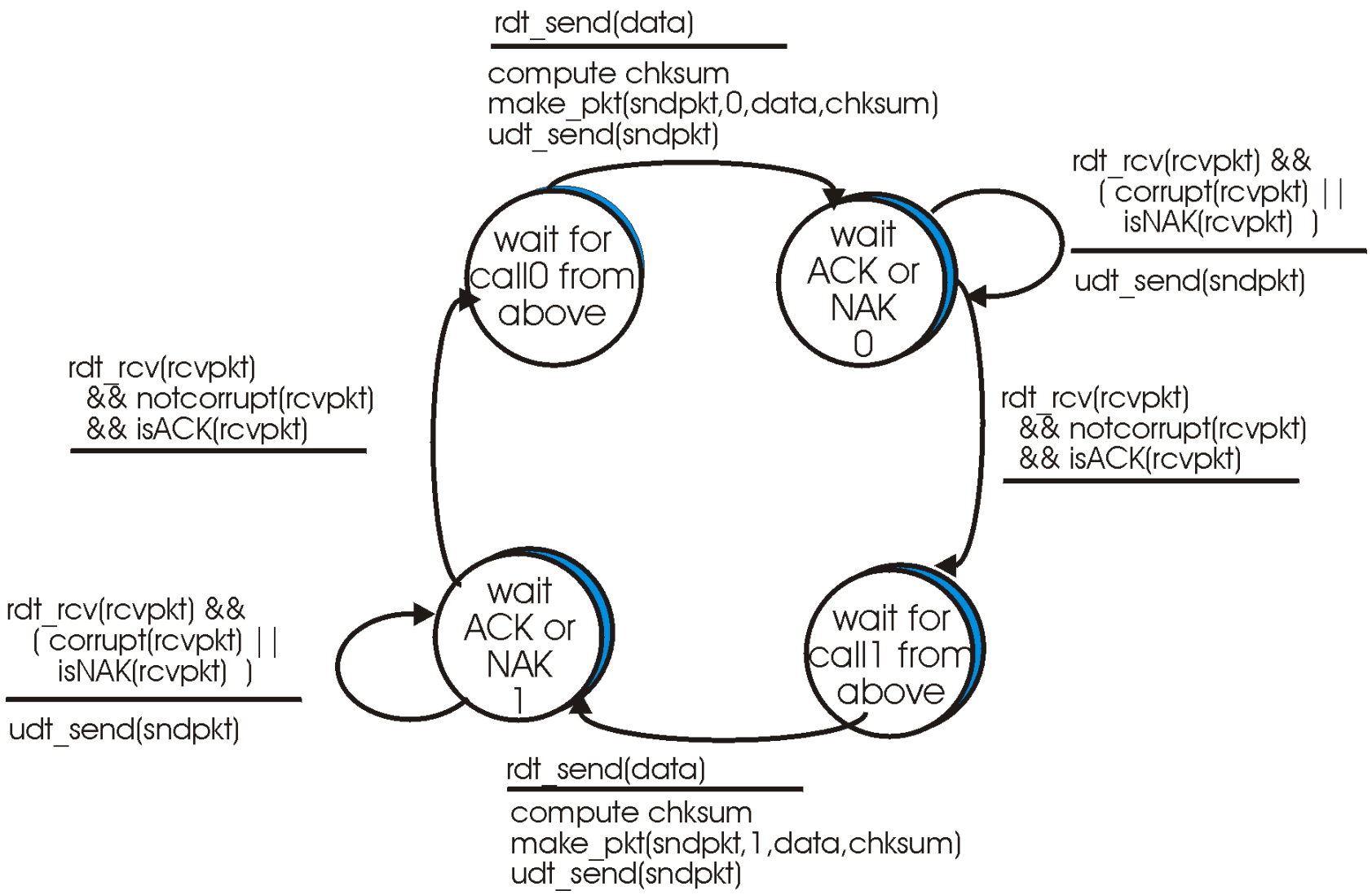
Tratando duplicatas:

- transmissor acrescenta *número de seqüência* em cada pacote
- Transmissor reenvia o último pacote se f ACK/NAK for perdido
- receptor descarta (não passa para a aplicação) pacotes duplicados

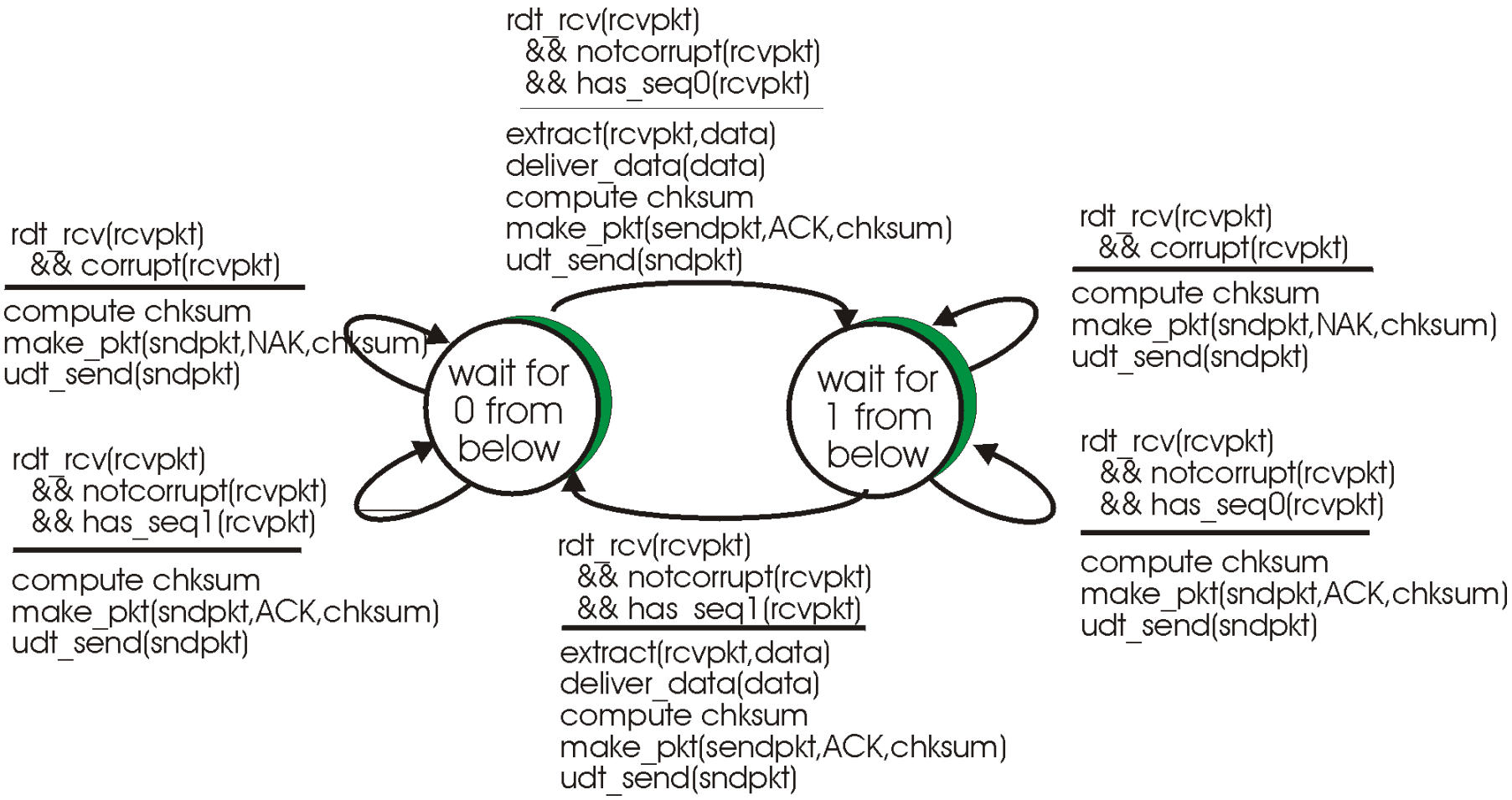
stop and wait

Transmissor envia um pacote e então espera pela resposta do receptor

rdt2.1: transmissor, trata ACK/NAKs perdidos



rdt2.1: receptor, trata ACK/NAKs perdidos



rdt2.1: discusssão

Transmissor:

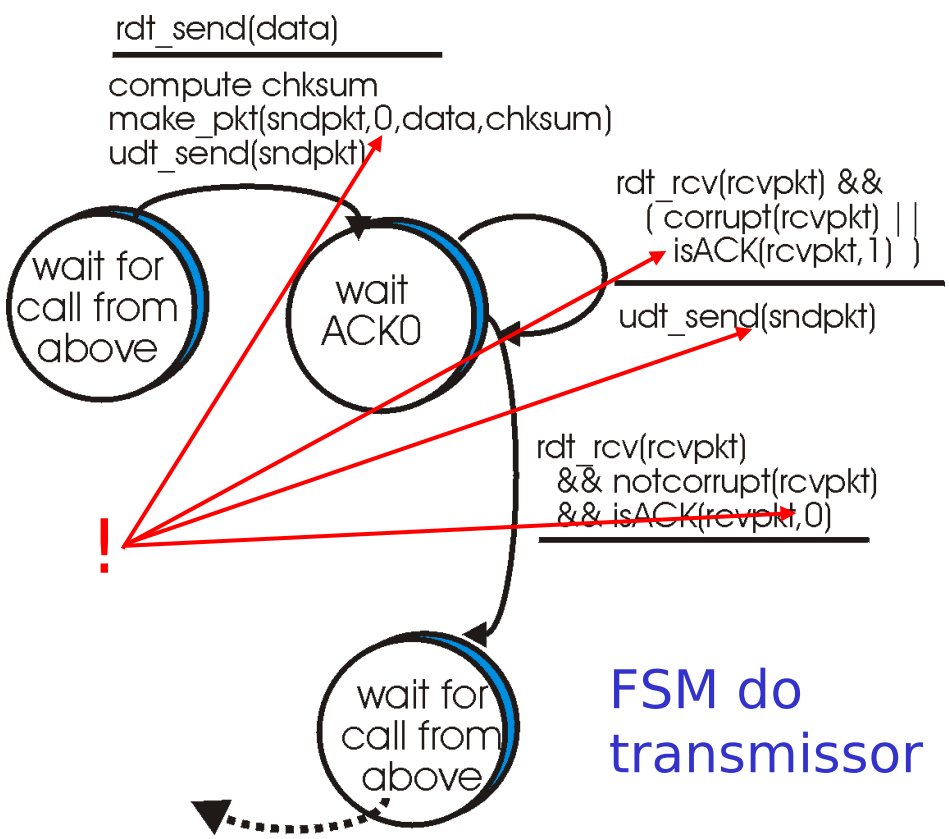
- adiciona número de seqüência ao pacote
- Dois números (0 e 1) bastam. Porque?
- deve verificar se os ACK/NAK recebidos estão corrompidos
- duas vezes o número de estados
  - o estado deve “lembrar” se o pacote “corrente” tem número de seqüência 0 ou 1

Receptor:

- deve verificar se o pacote recebido é duplicado
  - estado indica se o pacote 0 ou 1 é esperado
- nota: receptor pode não saber se seu últino ACK/ NAK foi recebido pelo transmissor

rdt2.2: um protocolo sem NAK

- mesma funcionalidade do rdt2.1, usando somente ACKs
- ao invés de enviar NAK, o receptor envia ACK para o último pacote recebido sem erro
  - receptor deve incluir explicitamente o número de seqüência do pacote sendo reconhecido
- ACKs duplicados no transmissor resultam na mesma ação do NAK: *retransmição do pacote corrente*



rdt3.0: canais com erros e perdas

Nova Hipótese: canal de transmissão pode também perder pacotes (dados oo ACKs)

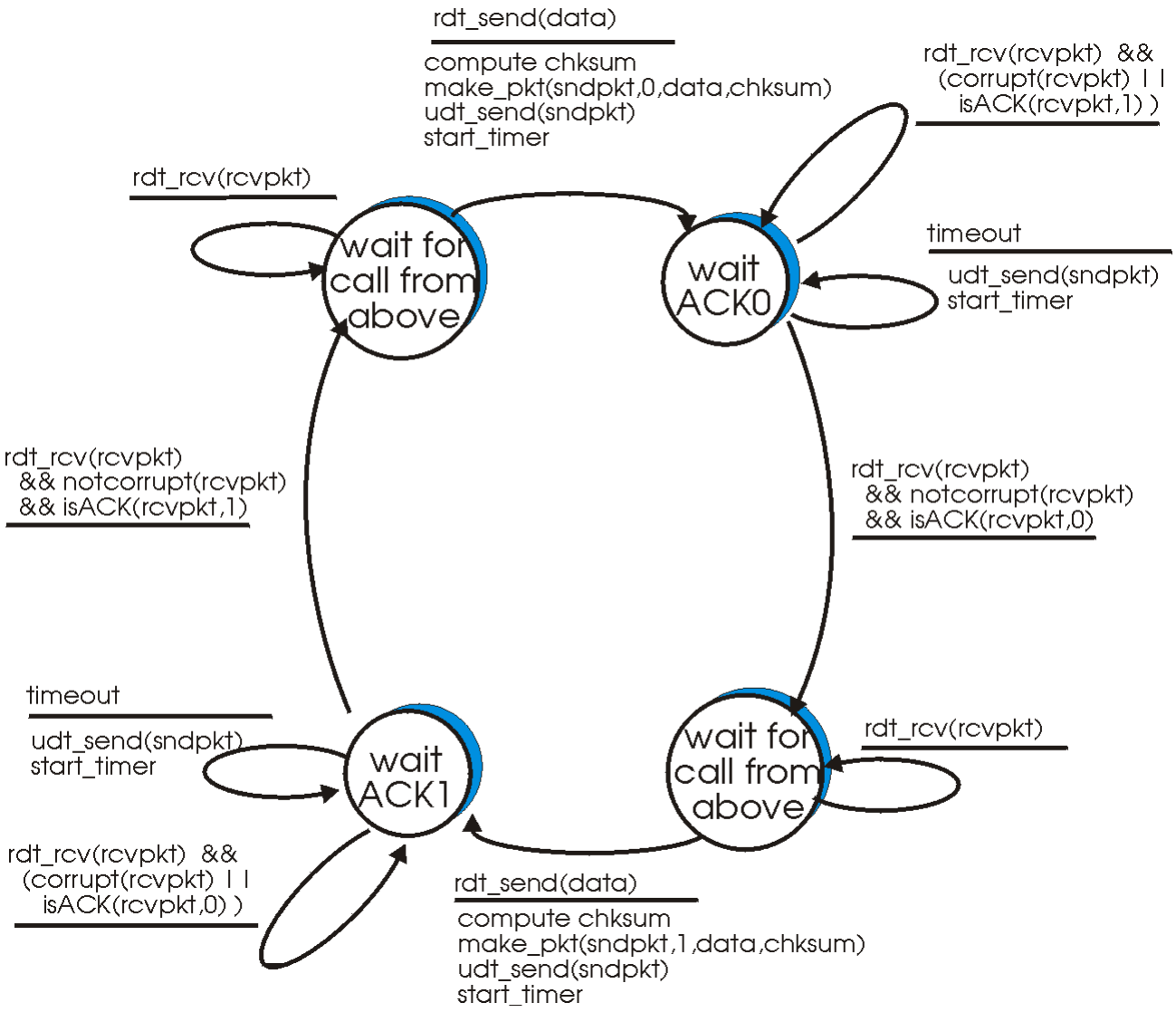
- checksum, números de seqüência, ACKs, retransmissões serão de ajuda, mas não o bastante

Q: como tratar com perdas?

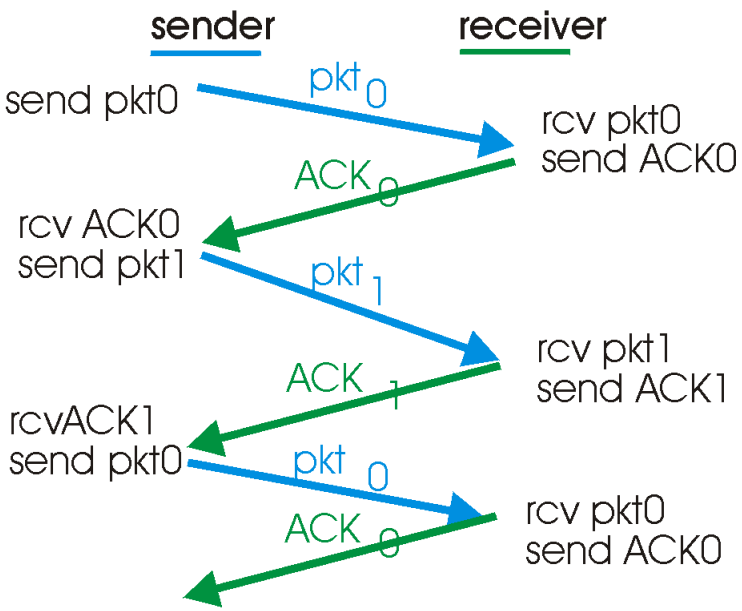
- transmissor espera até que certos dados ou ACKs sejam perdidos, então retransmite
- problemas?

Abordagem: transmissor espera um tempo “razoável” pelo ACK

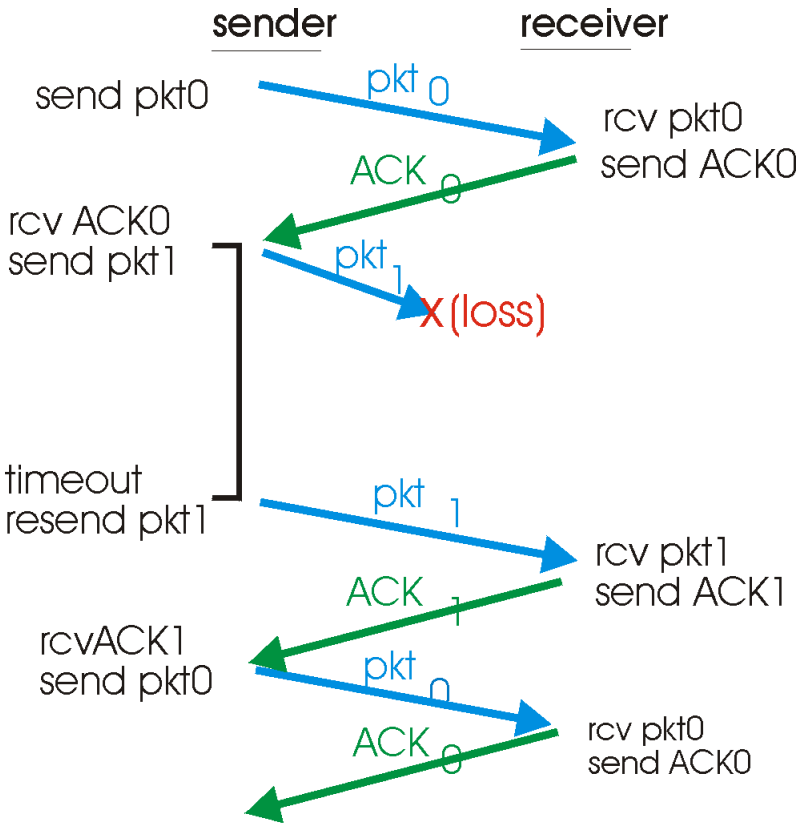
- retransmite se nenhum ACK for recebido neste tempo
- se o pacote (ou ACK) estiver apenas atrasado (não perdido):
  - retransmissão será duplicata, mas os números de seqüência já tratam com isso
  - receptor deve especificar o número de seqüência do pacote sendo reconhecido
- exige um temporizador decrescente


rdt_send(data)
compute chksum
make_pkt(sndpkt,0,data,chksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))
rdt_rcv(rcvpkt)
wait for
call from
above
wait
ACK0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)
timeout
udt_send(sndpkt)
start_timer
wait
ACK1
wait for
call from
above
rdt_rcv(rcvpkt)
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,0))
rdt_send(data)
compute chksum
make_pkt(sndpkt,1,data,chksum)
udt_send(sndpkt)
start_timer

rdt3.0 em ação

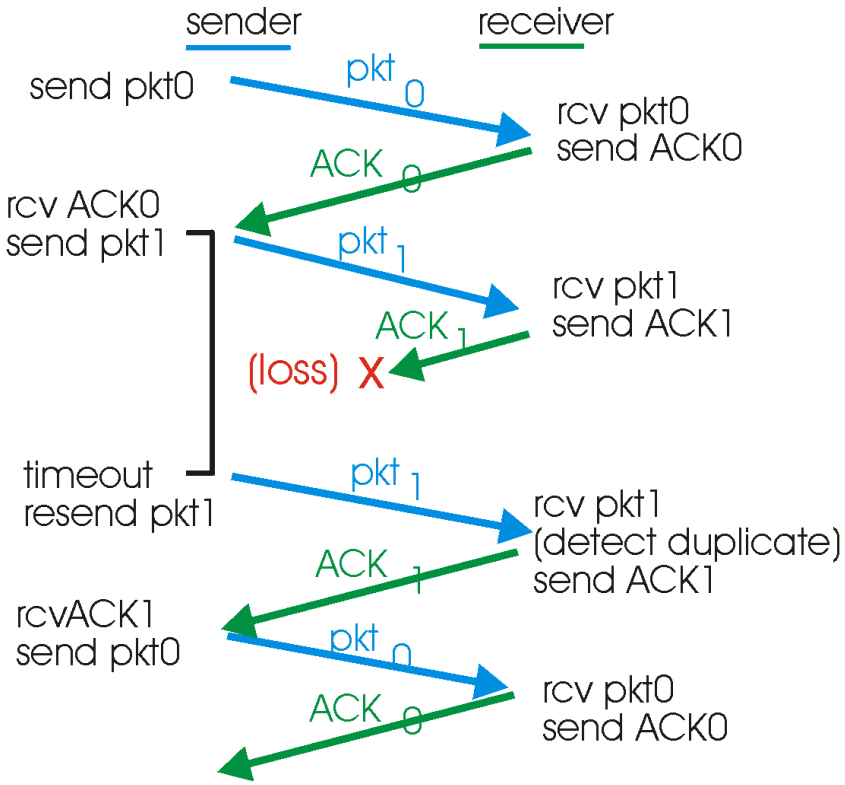


(a) operação sem perda

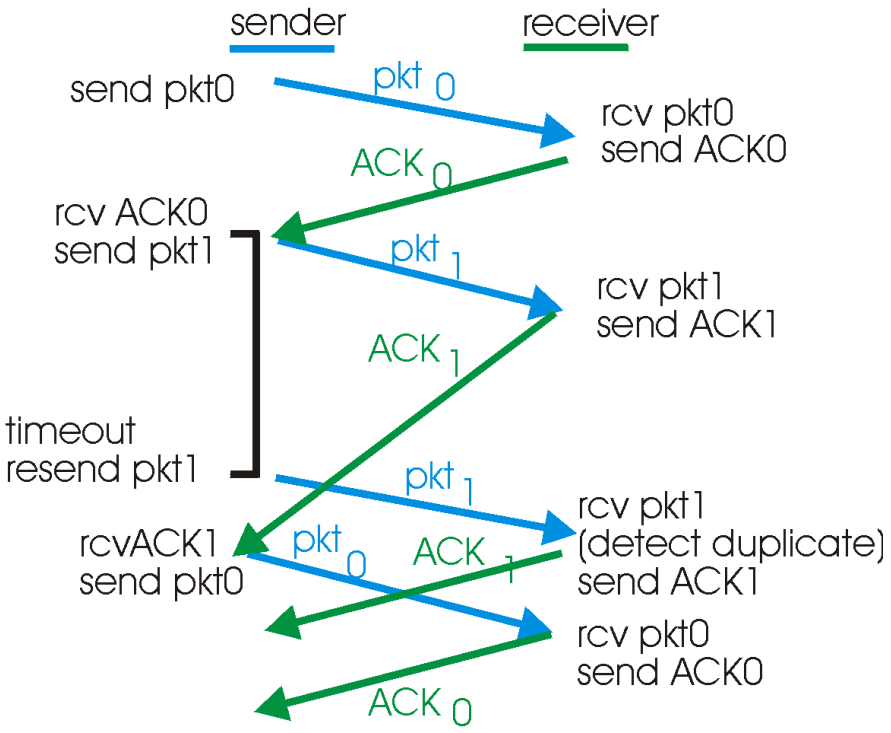


(b) pacote perdido

rdt3.0 em ação



(c) ACK perdido



(d) timeout prematuro

Desempenho do rdt3.0

- rdt3.0 funciona, mas o desempenho é sofrível
- exemplo: enlace de 1 Gbps, 15 ms de atraso de propagação, pacotes de 1KB:

$$\text{transmissão} = \frac{8\text{kb/pct}}{10^{**}9\ \text{b/seg}} = 8\ \mu\text{s}$$

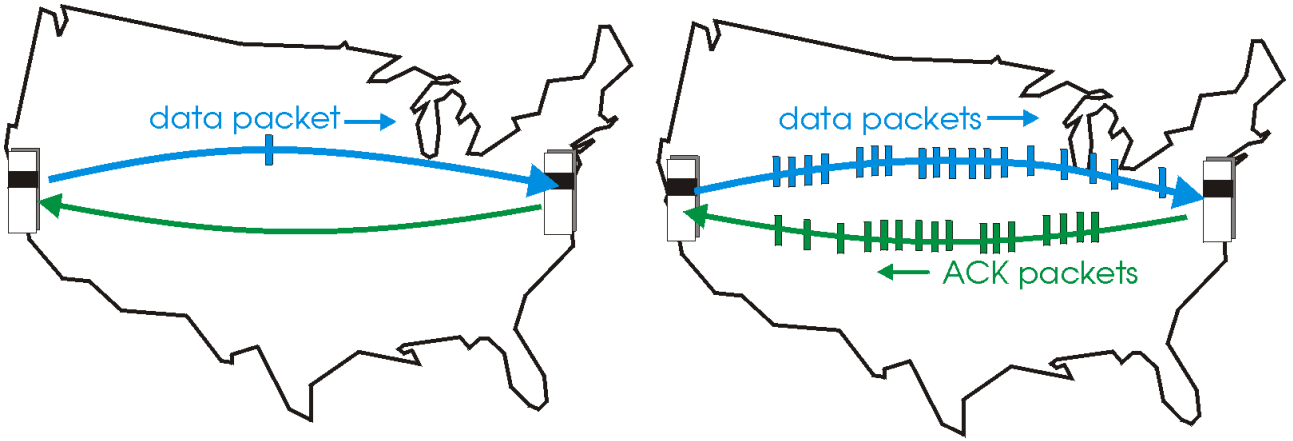
$$\text{Utilização} = U = \frac{\text{fração do tempo transmissor ocupado}}{} = \frac{8\ \mu\text{s}}{30{,}016\ \text{ms}} = 0.00015$$

  - Um pacote de 1KB cada 30 ms -> 33kB/seg de vazão sobre um canal de 1 Gbps
  - o protocolo de rede limita o uso dos recursos físicos!

Protocolos com Paralelismo (pipelining)

Paralelismo: transmissor envia vários pacotes ao mesmo tempo, todos esperando para serem reconhecidos

- faixa de números de seqüência deve ser aumentada
- armazenamento no transmissor e/ou no receptor



(a) operação do protocolo stop-and-wait

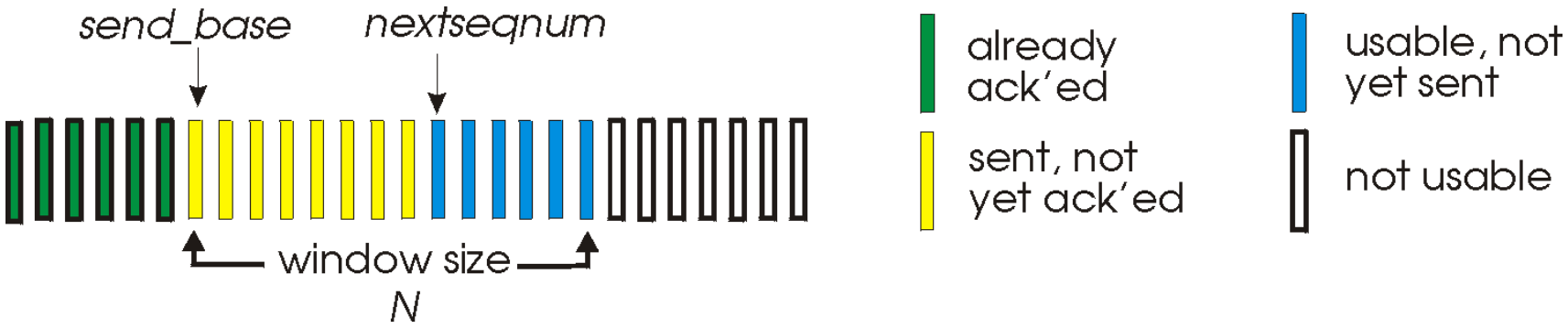
(a) operação do protocolo com paralelismo

- Duas formas genéricas de protocolos com paralelismo: *go-Back-N, retransmissão seletiva*

Go-Back-N

Transmissor:

- Número de seqüência com k bits no cabeçalho do pacote
- “janela” de até N, pacotes não reconhecidos, consecutivos, são permitidos

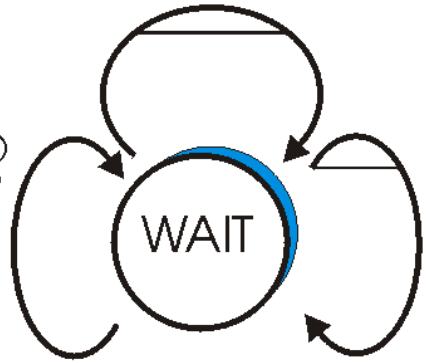


- ACK(n): reconhece todos os pacotes até o número de seqüência N (incluindo este limite). “ACK cumulativo”
  - pode receber ACKS duplicados (veja receptor)
- temporizador para cada pacote enviado e não confirmado
- *timeout(n):* retransmite pacote n e todos os pacotes com número de seqüência maior que estejam dentro da janela

GBN: FSM estendida para o transmissor

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)
```



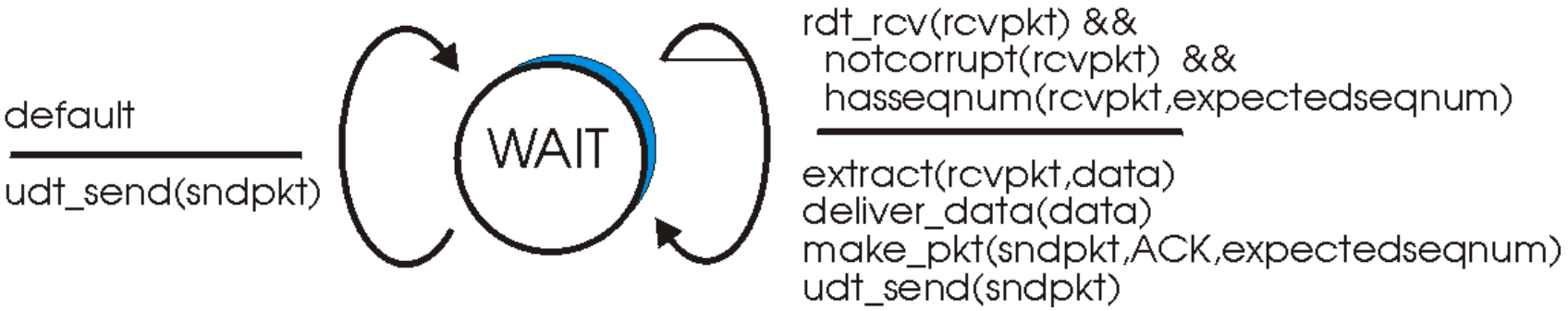
rdt_rcv(rcv_pkt) && notcorrupt(rcvpkt)

```
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))
```

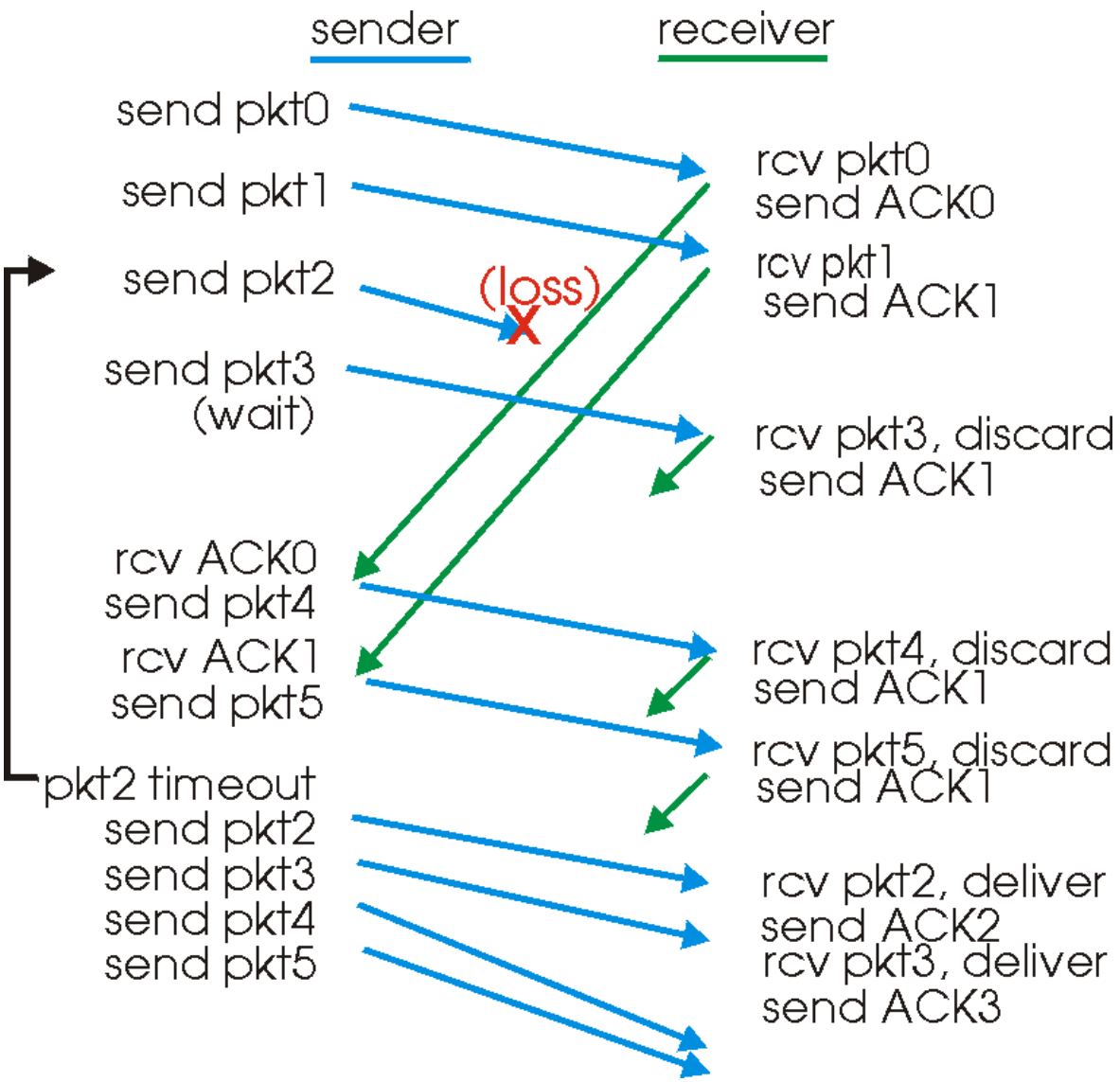
GBN: FSM estendida para o receptor



receptor simples:

- somente ACK: sempre envia ACK para pacotes corretamente recebidos com o mais alto número de seqüência *em ordem*
  - pode gerar ACKs duplicados
  - precisa lembrar apenas do número de seqüência esperado (**`expectedseqnum`**)
- pacotes fora de ordem:
  - descarte (não armazena) -> não há buffer de recepção!
  - reconhece pacote com o mais alto número de seqüência em ordem

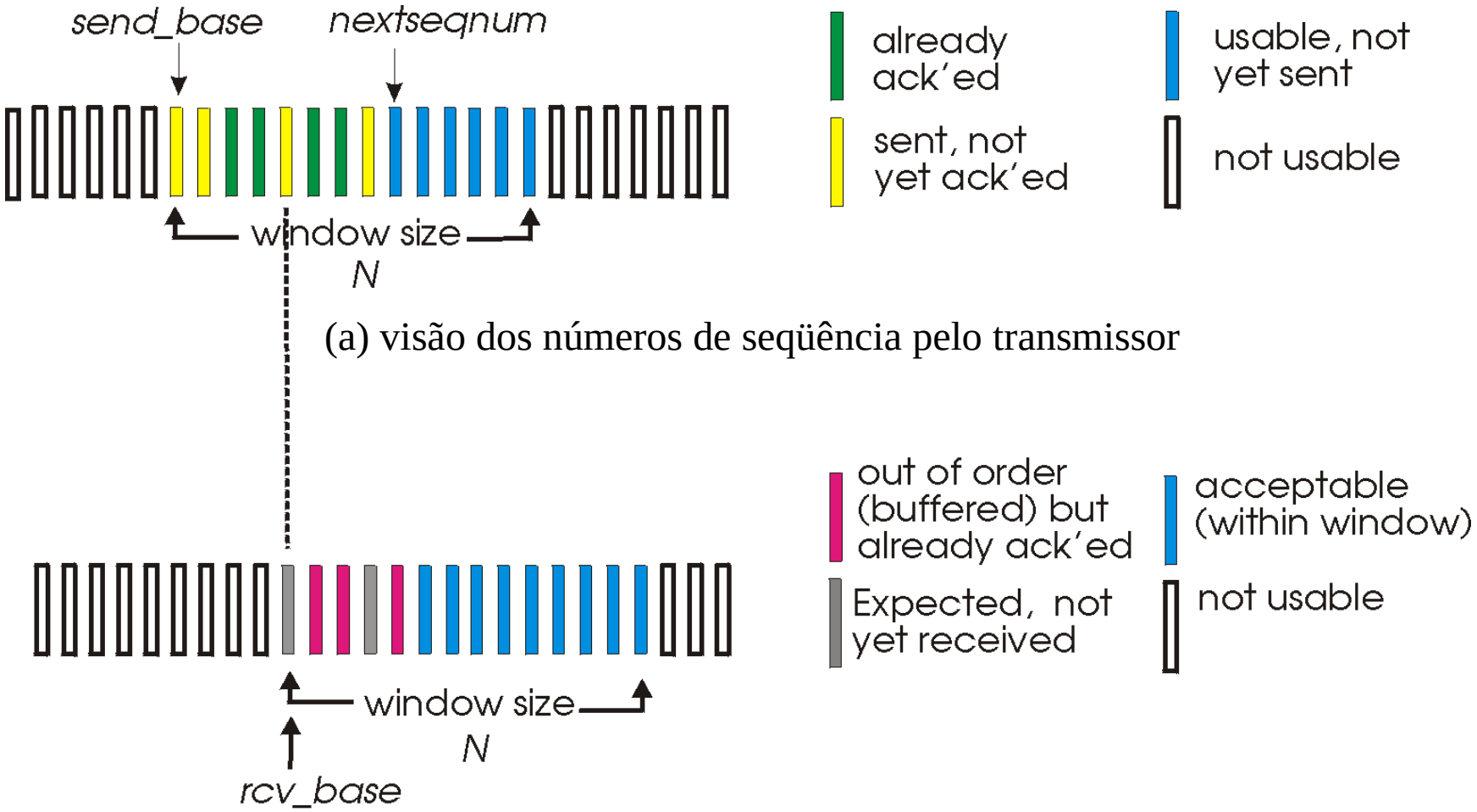
GBN em ação


sender
receiver
send pkt0
send pkt1
send pkt2
(loss)
send pkt3
(wait)
rcv ACK0
send pkt4
rcv ACK1
send pkt5
pkt2 timeout
send pkt2
send pkt3
send pkt4
send pkt5
rcv pkt0
send ACK0
rcv pkt1
send ACK1
rcv pkt3, discard
send ACK1
rcv pkt4, discard
send ACK1
rcv pkt5, discard
send ACK1
rcv pkt2, deliver
send ACK2
rcv pkt3, deliver
send ACK3

Retransmissão Seletiva

- receptor reconhece *individualmente* todos os pacotes recebidos corretamente
  - armazena pacotes, quando necessário, para eventual entrega em ordem para a camada superior
- transmissor somente reenvia os pacotes para os quais um ACK não foi recebido
  - transmissor temporiza cada pacote não reconhecido
- janela de transmissão
  - N números de seqüência consecutivos
  - novamente limita a quantidade de pacotes enviados, mas não reconhecidos

Retransmissão seletiva: janelas do transmissor e do receptor



(a) visão dos números de seqüência pelo transmissor

(b) visão dos números de seqüência pelo receptor

Retransmissão seletiva

transmissor

dados da camada superior :

- se o próximo número de seqüência disponível está na janela, envia o pacote

timeout(n):

- reenvia pacote n,
  restart timer

ACK(n) em [sendbase,sendbase+N]:

- marca pacote n como recebido
- se n é o menor pacote não reconhecido, avança a base da janela para o próximo número de seqüência não reconhecido

receptor

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

- envia ACK(n)
- fora de ordem: armazena
- em ordem: entrega (também entrega pacotes armazenados em ordem), avança janela para o próximo pacote ainda não recebido

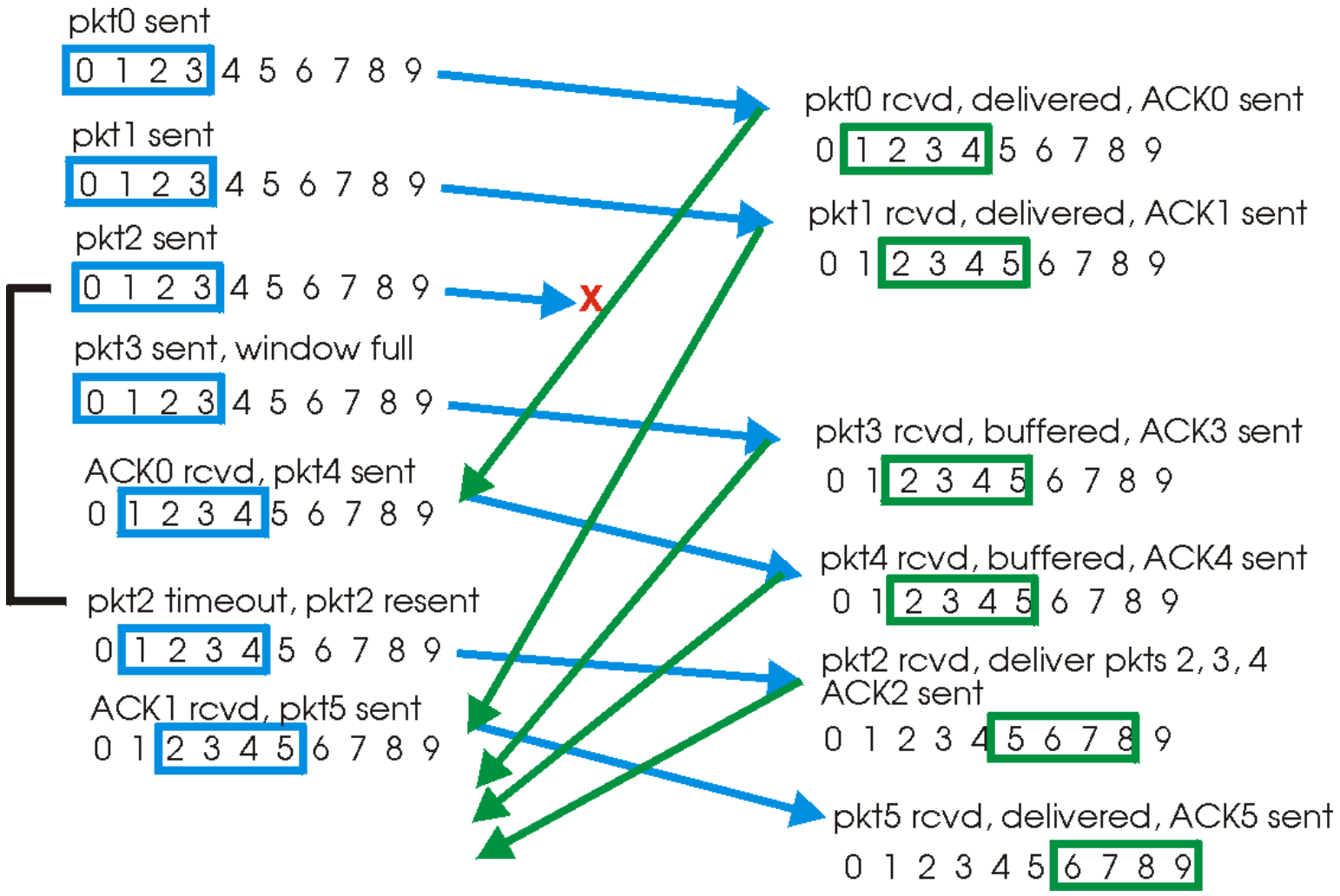
pkt n em [rcvbase-N,rcvbase-1]

- ACK(n)

caso contrário:

- ignora

Retransmissão seletiva em ação



Retransmissão seletiva: dilema

Exemplo:

- seqüências: 0, 1, 2, 3
- tamanho da janela=3

- receptor não vê diferença nos dois cenários!
- incorretamente passa dados duplicados como novos (figura a)

Q: qual a relação entre o espaço de numeração seqüencial e o tamanho da janela?