Camada de transporte

Objetivos:

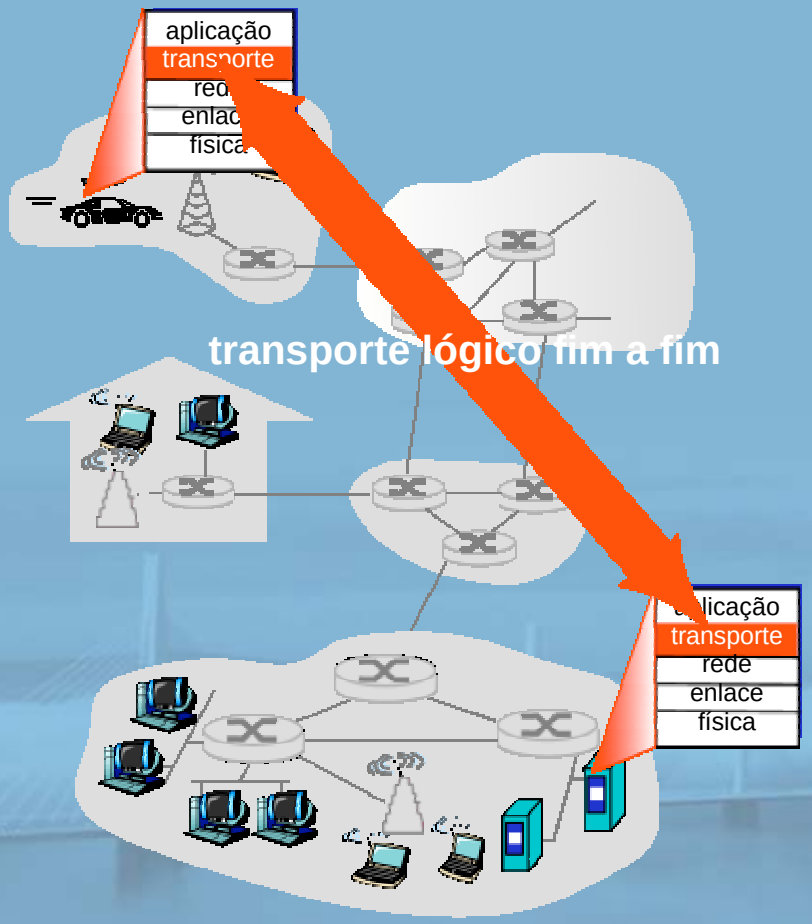
- entender princípios por trás dos serviços da camada de transporte:
  - multiplexação/demul-tiplexação
  - transferência de dados confiável
  - controle de fluxo
  - controle de congestionamento
- aprender sobre os protocolos da camada de transporte na Internet:
  - UDP: transporte sem conexão
  - TCP: transporte orientado a conexão
  - controle de congestionamento TCP

Esboço

- Serviços da camada de transporte
- Multiplexação e demultiplexação
- Transporte não orientado para conexão: UDP
- Princípios da transferência confiável de dados
- Transporte orientado para conexão: TCP
  - estrutura de segmento
  - transferência confiável de dados
  - controle de fluxo
  - gerenciamento da conexão
- Princípios de controle de congestionamento
- Congestionamento no TCP

Serviços e protocolos de transporte

- oferecem *comunicação lógica* entre processos de aplicação rodando em hospedeiros diferentes
- protocolos de transporte rodam em sistemas finais
  - lado remetente: divide as msgs da aplicação em segmentos, passa à camada de rede
  - lado destinatário: remonta os segmentos em msgs, passa à camada de aplicação
- mais de um protocolo de transporte disponível às aplicações
  - Internet: TCP e UDP



Camada de transporte *versus* rede

- *camada de rede:* comunicação lógica entre hospedeiros
- *camada de transporte:* comunicação lógica entre processos
  - conta com e amplia os serviços da camada de rede

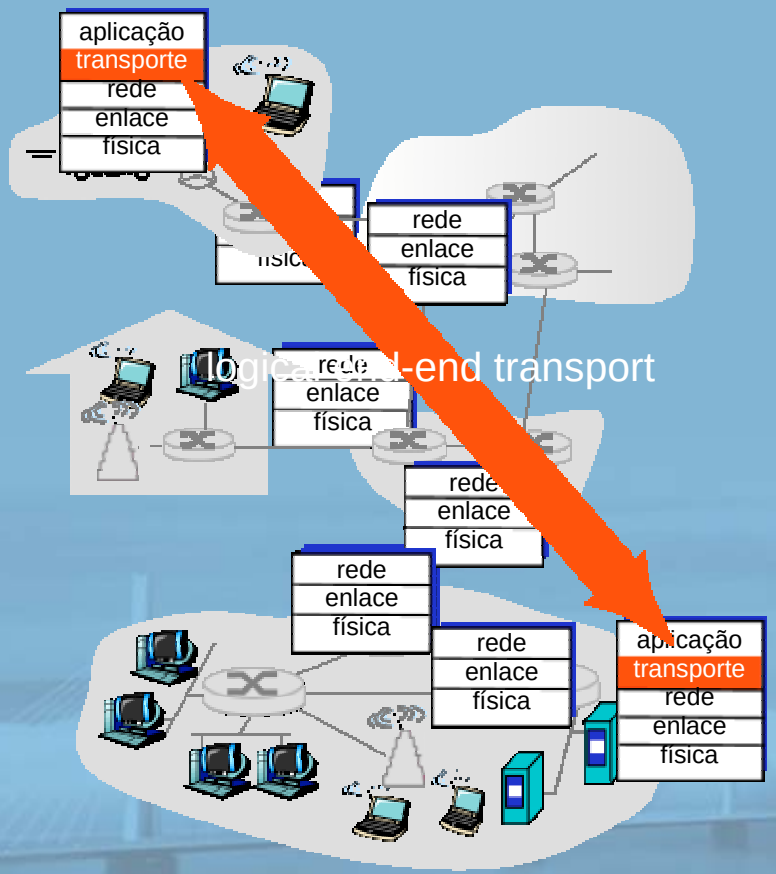
analogia com a família:

12 crianças mandando carta a 12 crianças

- processos = crianças
- msgs da aplicação = cartas nos envelopes
- hospedeiros = casas
- protocolo de transporte = Ana e Bill
- protocolo da camada de rede = serviço postal

Protocolos da camada de transporte da Internet

- remessa confiável e em ordem (TCP)
  - controle de congestionamento
  - controle de fluxo
  - estabelecimento da conexão
- remessa não confiável e desordenada: UDP
  - extensão sem luxo do IP pelo "melhor esforço"
- serviços não disponíveis:
  - garantias de atraso
  - garantias de largura de banda



Esboço

- Serviços da camada de transporte
- Multiplexação e demultiplexação
- Transporte não orientado para conexão: UDP
- Princípios da transferência confiável de dados
- Transporte orientado para conexão: TCP
  - estrutura de segmento
  - transferência confiável de dados
  - controle de fluxo
  - gerenciamento da conexão
- Princípios de controle de congestionamento
- Congestionamento no TCP

Multiplexação/ demultiplexação

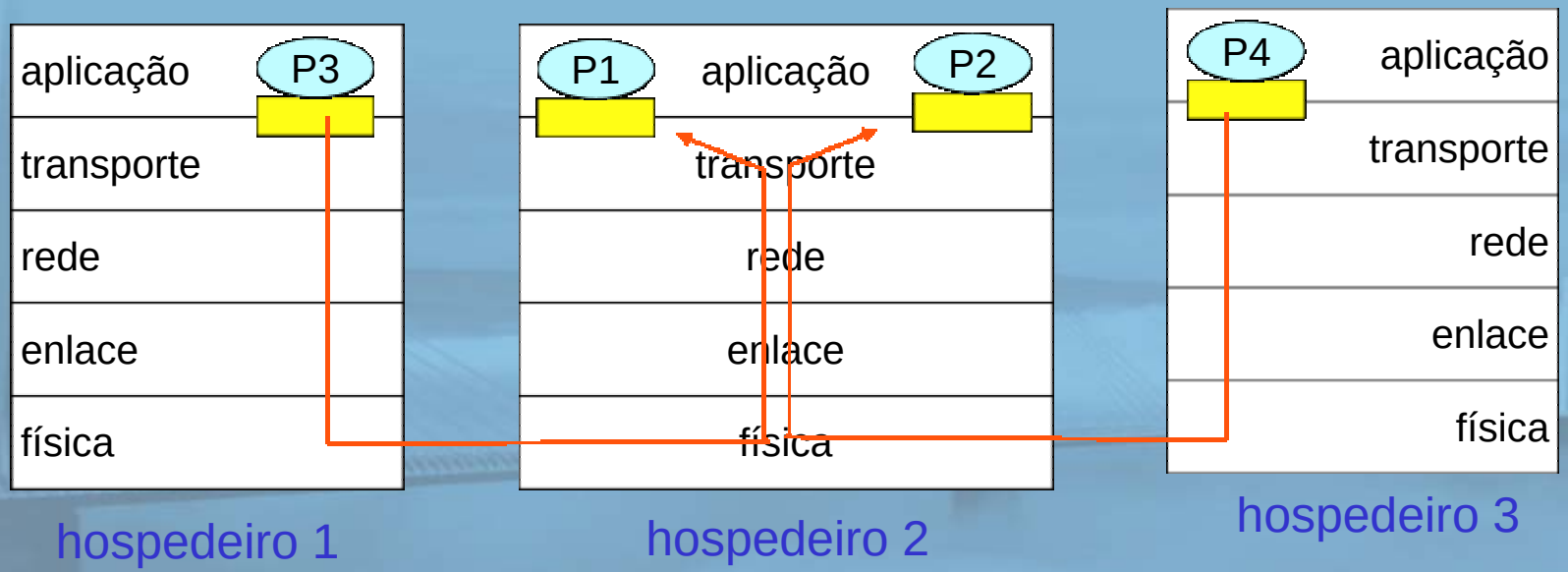
demultiplexação no destinatário:

entregando segmentos recebidos ao socket correto

multiplexação no remetente:

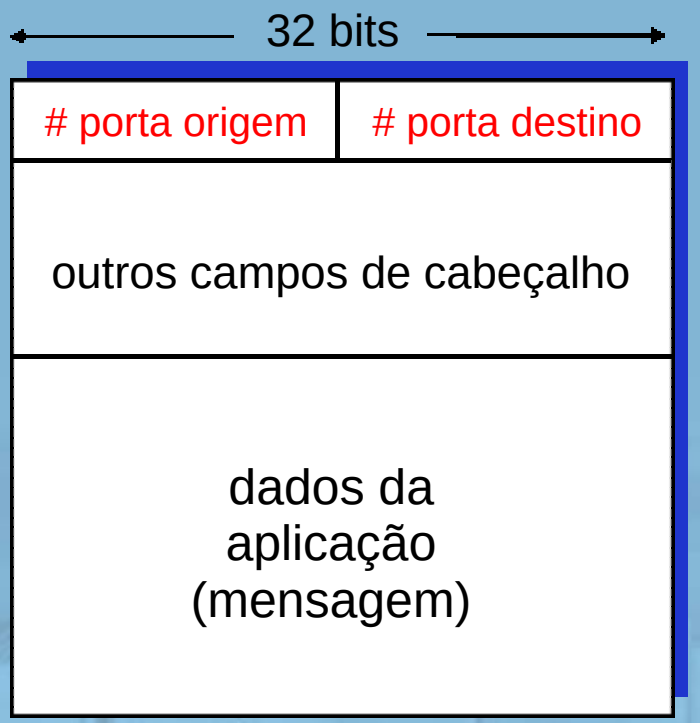
colhendo dados de múltiplos sockets, envelopando dados com cabeçalho (usados depois para demultiplexação)

= socket = processo



Como funciona a demultiplexação

- hospedeiro recebe datagramas IP
  - cada datagrama tem endereço IP de origem, endereço IP de destino
  - cada datagrama carrega 1 segmento da camada de transporte
  - cada segmento tem número de porta de origem, destino
- hospedeiro usa endereços IP & números de porta para direcionar segmento ao socket apropriado



formato do segmento TCP/UDP

Demultiplexação não orientada para conexão

- cria sockets com números de porta:

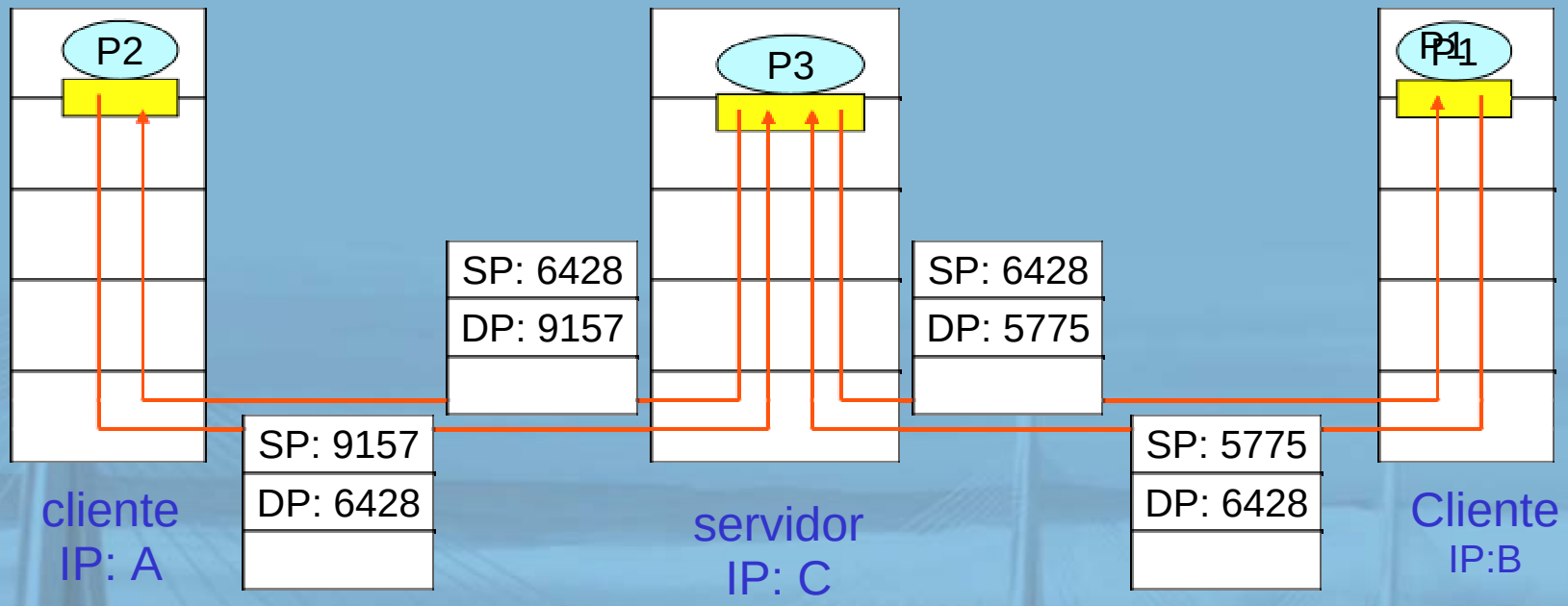
```
DatagramSocket mySocket1 = new
DatagramSocket(12534);
DatagramSocket mySocket2 = new
DatagramSocket(12535);
```

- socket UDP identificado por tupla de dois elementos:

(endereço IP destino, número porta destino)

- quando hospedeiro recebe segmento UDP:
  - verifica número de porta de destino no segmento
  - direciona segmento UDP para socket com esse número de porta
- datagramas IP com diferentes endereços IP de origem e/ou números de porta de origem direcionados para o mesmo socket

DatagramSocket serverSocket = new DatagramSocket(6428);



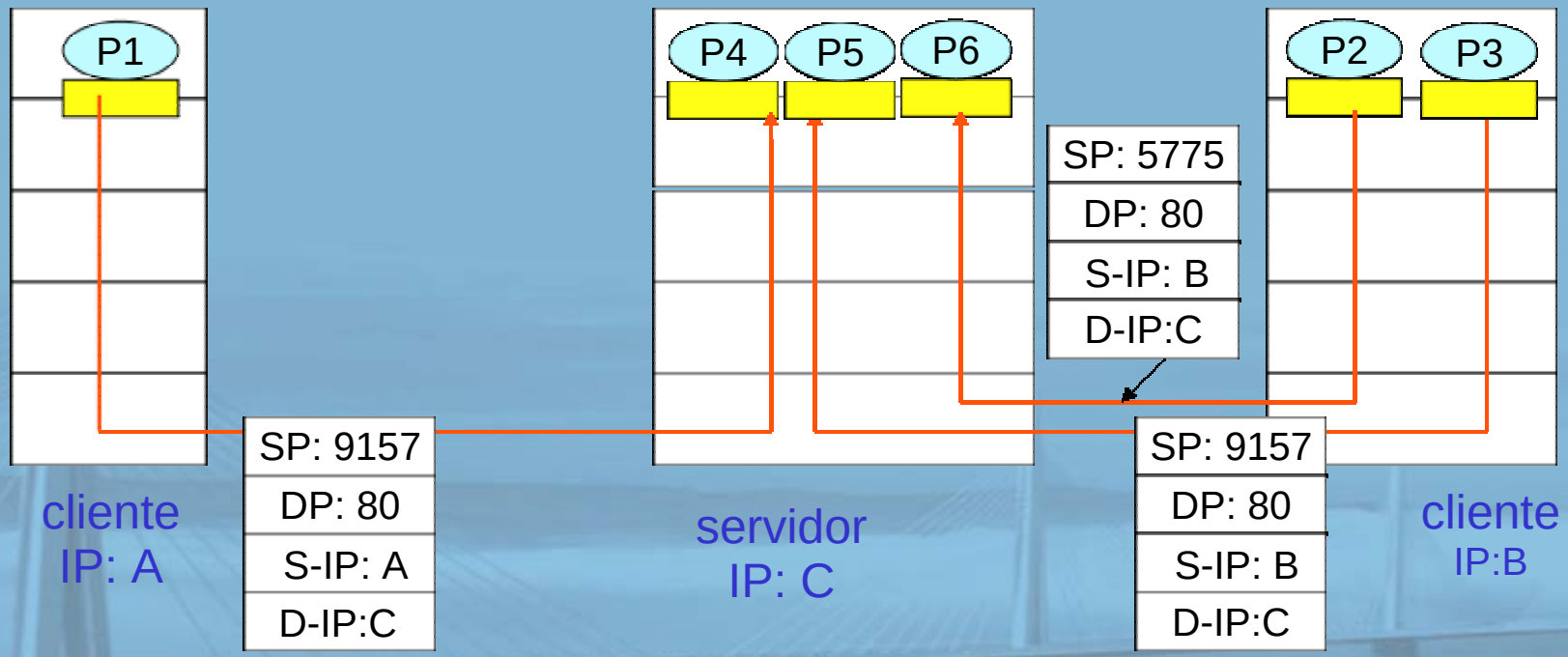
SP oferece “endereço de retorno”

Demultiplexação orientada para conexão

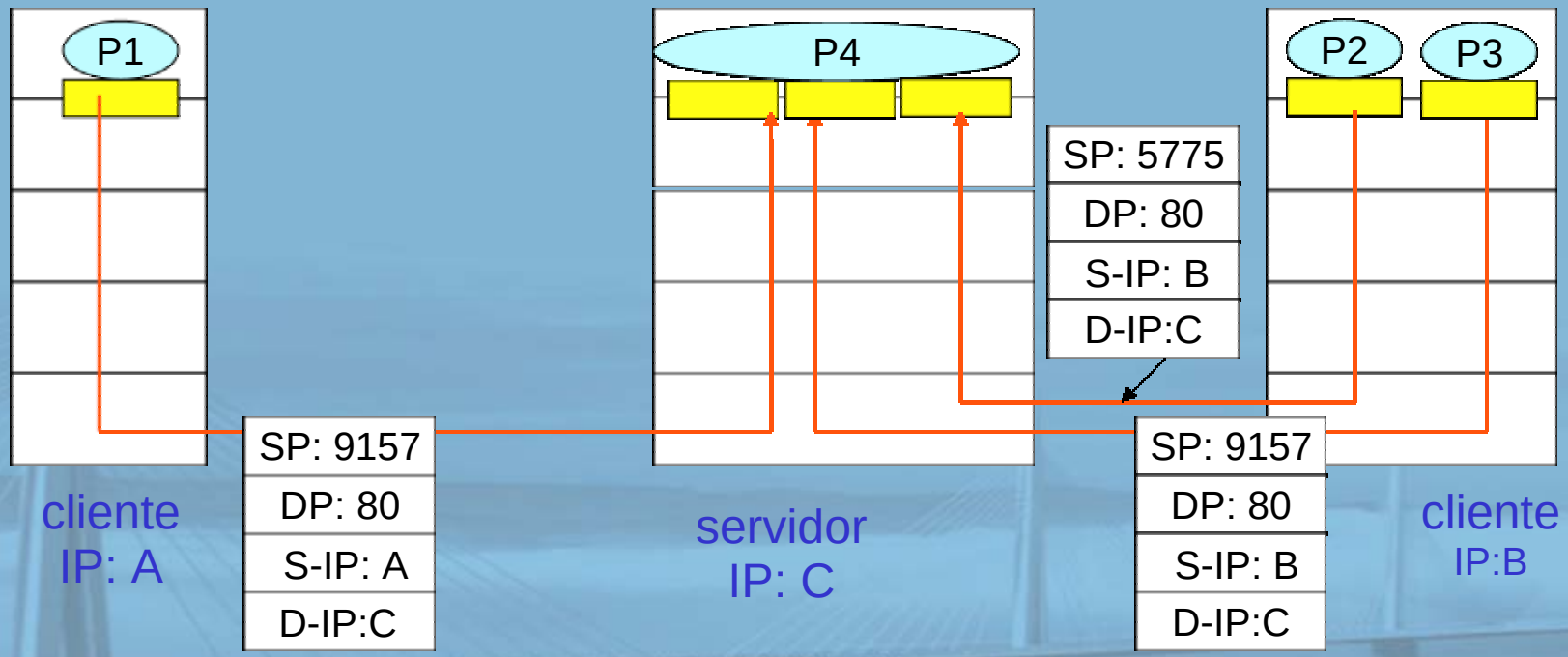
REDES DE COMPUTADORES E A INTERNET 5ª edição
Uma Abordagem Top-Down

- socket TCP identificado por tupla de 4 elementos:
  - endereço IP de origem
  - número de porta de origem
  - endereço IP de destino
  - número de porta de destino
- hospedeiro destinatário usa todos os quatro valores para direcionar segmento ao socket apropriado

- hospedeiro servidor pode admitir muitos sockets TCP simultâneos:
  - cada socket identificado por usa própria tupla de 4
- servidores Web têm diferentes sockets para cada cliente conectando
  - HTTP não persistente terá diferentes sockets para cada requisição



Demultiplexação orientada para conexão: servidor Web threaded



Esboço

- Serviços da camada de transporte
- Multiplexação e demultiplexação
- Transporte não orientado para conexão: UDP
- Princípios da transferência confiável de dados
- Transporte orientado para conexão: TCP
  - estrutura de segmento
  - transferência confiável de dados
  - controle de fluxo
  - gerenciamento da conexão
- Princípios de controle de congestionamento
- Congestionamento no TCP

UDP: User Datagram Protocol [RFC 768]

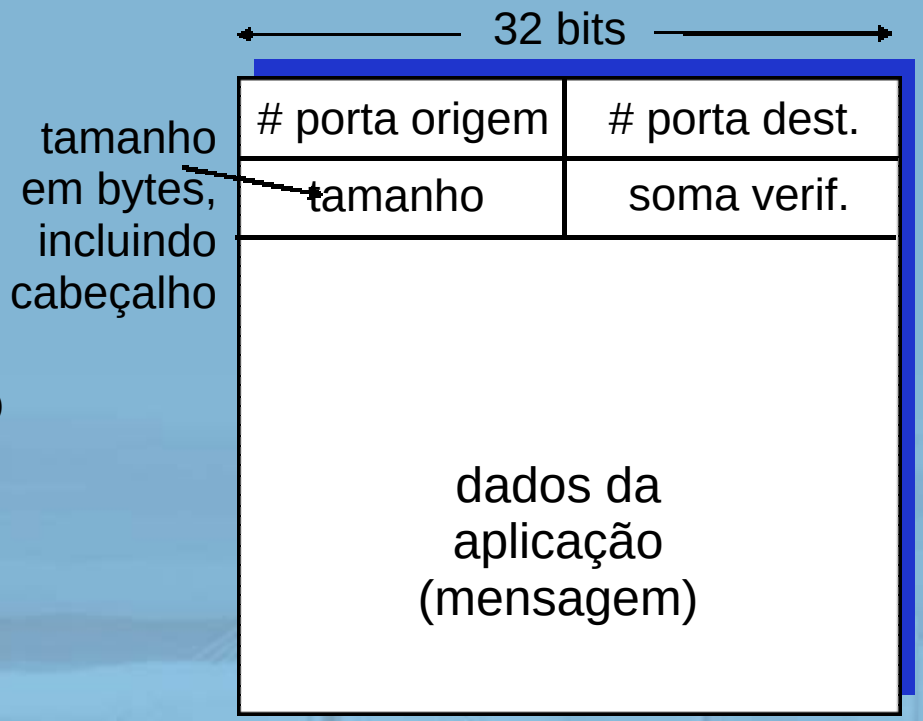
- protocolo de transporte da Internet “sem luxo”, básico
- serviço de “melhor esforço”, segmentos UDP podem ser:
  - perdidos
  - entregues à aplicação fora da ordem
- *sem conexão:*
  - sem handshaking entre remetente e destinatário UDP
  - cada segmento UDP tratado independente dos outros

Por que existe um UDP?

- sem estabelecimento de conexão (que pode gerar atraso)
- simples: sem estado de conexão no remetente, destinatário
- cabeçalho de segmento pequeno
- sem controle de congestionamento: UDP pode transmitir o mais rápido possível

UDP: mais

- normalmente usado para streaming de aplicações de multimídia
  - tolerante a perdas
  - sensível à taxa
- outros usos do UDP
  - DNS
  - SNMP (simple network management)
- transferência confiável por UDP: aumenta confiabilidade na camada de aplicação
  - recuperação de erro específica da aplicação!



formato de segmento UDP

Soma de verificação UDP

objetivo: detectar “erros” (p. e., bits invertidos) no segmento transmitido

remetente:

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

destinatário:

- calcula soma de verificação do segmento recebido
- verifica se soma de verificação calculada igual ao valor do campo de soma de verificação:
  - NÃO – erro detectado
  - SIM – nenhum erro detectado. *Mas pode haver erros mesmo assim*

Esboço

- Serviços da camada de transporte
- Multiplexação e demultiplexação
- Transporte não orientado para conexão: UDP
- Princípios da transferência confiável de dados
- Transporte orientado para conexão: TCP
  - estrutura de segmento
  - transferência confiável de dados
  - controle de fluxo
  - gerenciamento da conexão
- Princípios de controle de congestionamento
- Congestionamento no TCP

Princípios de transferência confiável de dados

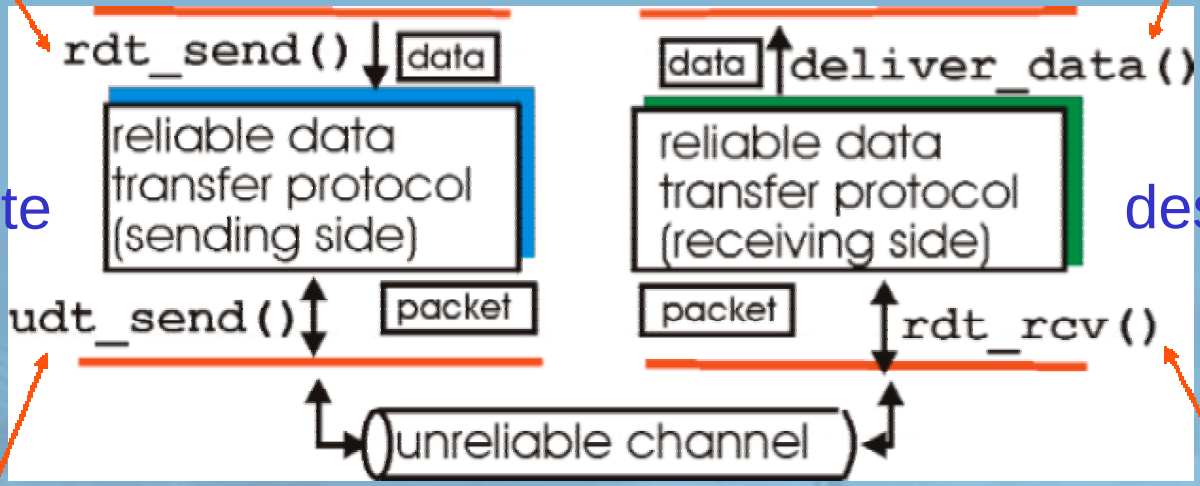
- importante nas camadas de aplicação, transporte e enlace
- entre os mais importantes tópicos de redes!
- características do canal confiável determinarão complexidade do protocolo de transferência confiável.

Transferência confiável de dados: introdução

rdt_send(): chamado de cima, (p. e., pela apl.). Dados passados para remeter à camada superior do destinatário

deliver_data(): chamado pela **rdt** para remeter dados para cima

lado remetente



lado destinatário

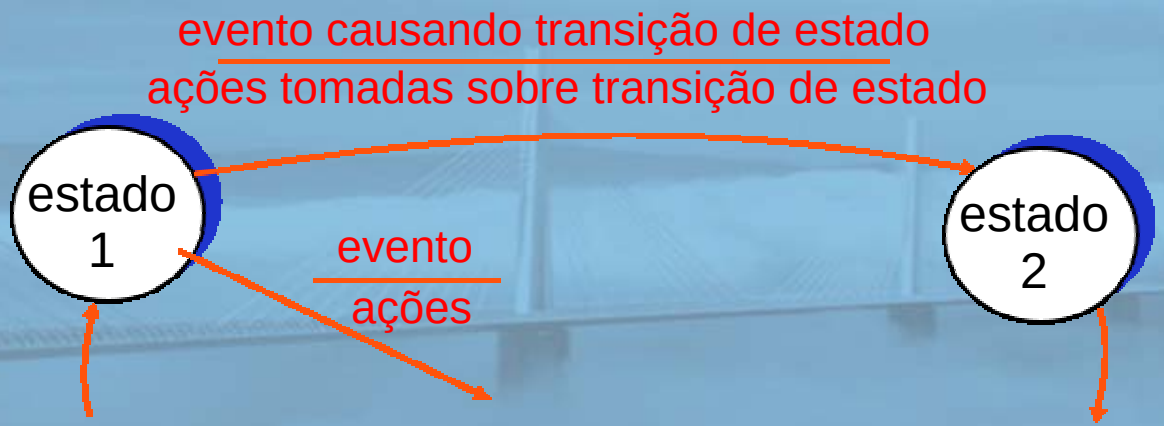
udt_send(): chamado pela rdt, para transferir pacote por canal não confiável ao destinatário

rdt_rcv(): chamado quando pacote chega no lado destinatário do canal

vamos:

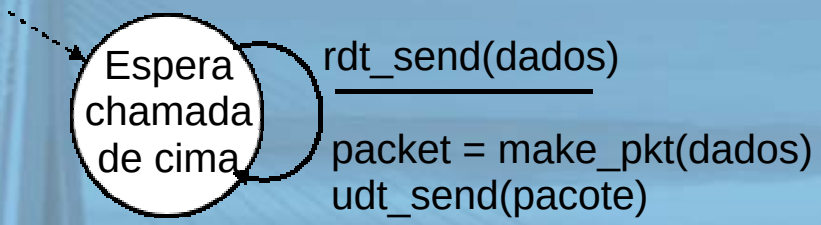
- desenvolver de forma incremental os lados remetente e destinatário do protocolo de transferência confiável de dados (rdt)
- considerar apenas a transf. de dados unidirecional
  - mas informações de controle fluirão nas duas direções!
- usar máquinas de estado finito (FSM) para especificar remetente, destinatário

estado: quando neste "estado", próximo estado determinado exclusivamente pelo próximo evento

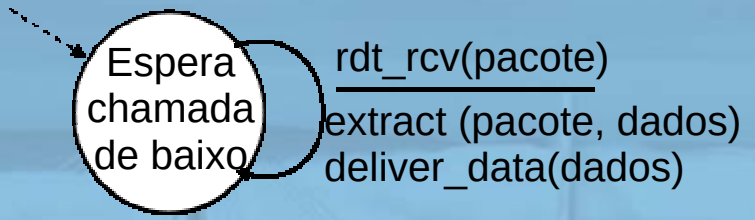


Rdt1.0: transferência confiável por canal confiável

- canal subjacente perfeitamente confiável
  - sem erros de bit
  - sem perda de pacotes
- FSMs separadas para remetente e destinatário:
  - remetente envia dados para canal subjacente
  - destinatário lê dados do canal subjacente



remetente

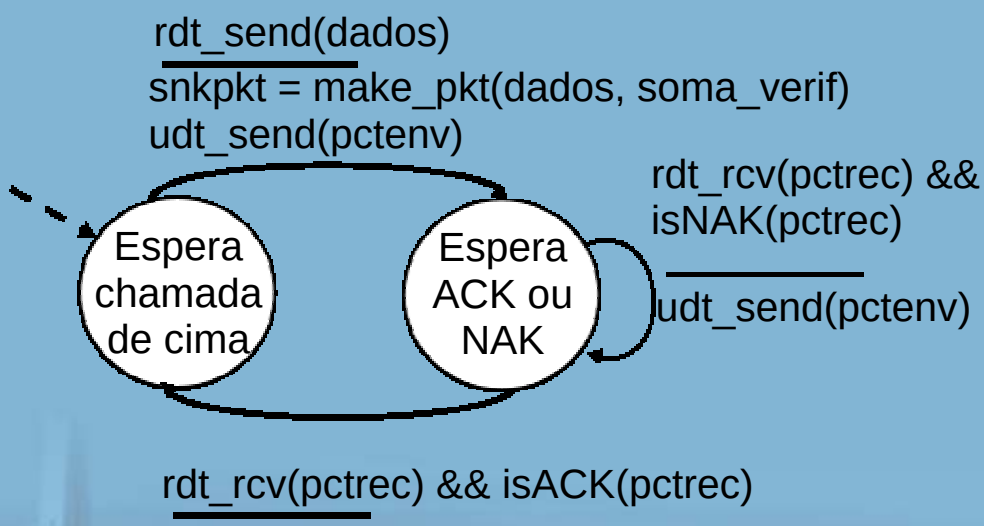


destinatário

Rdt2.0: canal com erros de bit

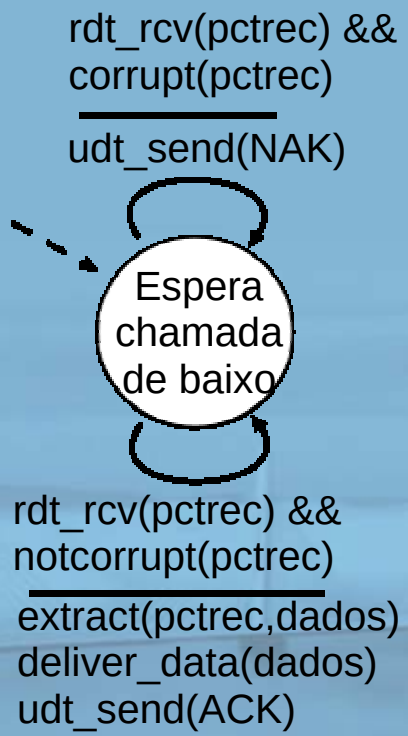
- canal subjacente pode inverter bits no pacote
  - soma de verificação para detectar erros de bit
- *a* questão: como recuperar-se dos erros:
  - *reconhecimentos (ACKs):* destinatário diz explicitamente ao remetente que o pacote foi recebido OK
  - *reconhecimentos negativas (NAKs):* destinatário diz explicitamente ao remetente que o pacote teve erros
  - remetente retransmite pacote ao receber NAK
- novos mecanismos no **rdt2.0** (além do **rdt1.0**):
  - detecção de erro
  - feedback do destinatário: msgs de controle (ACK,NAK) destinatário->remetente

rdt2.0: especificação da FSM

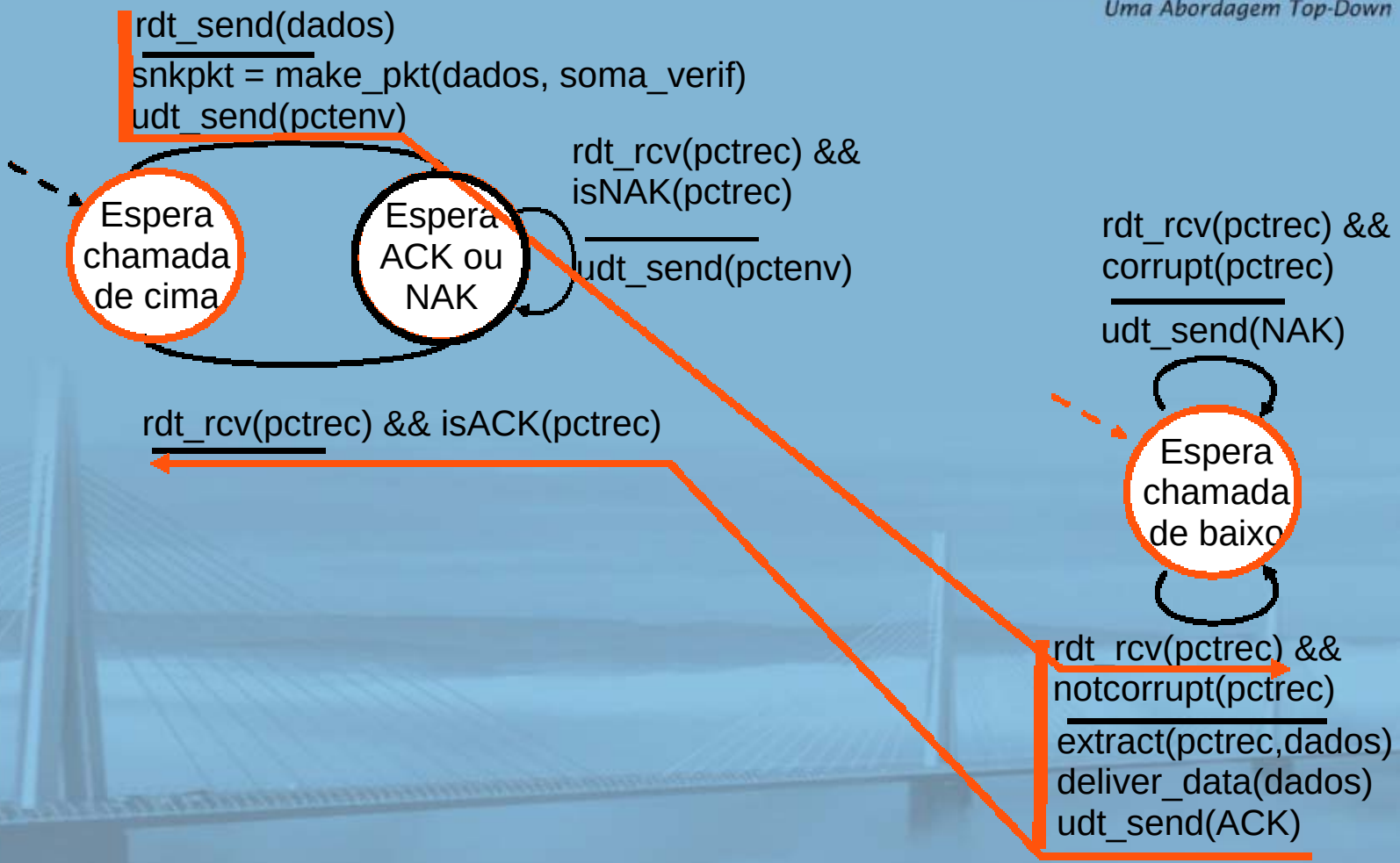


remetente

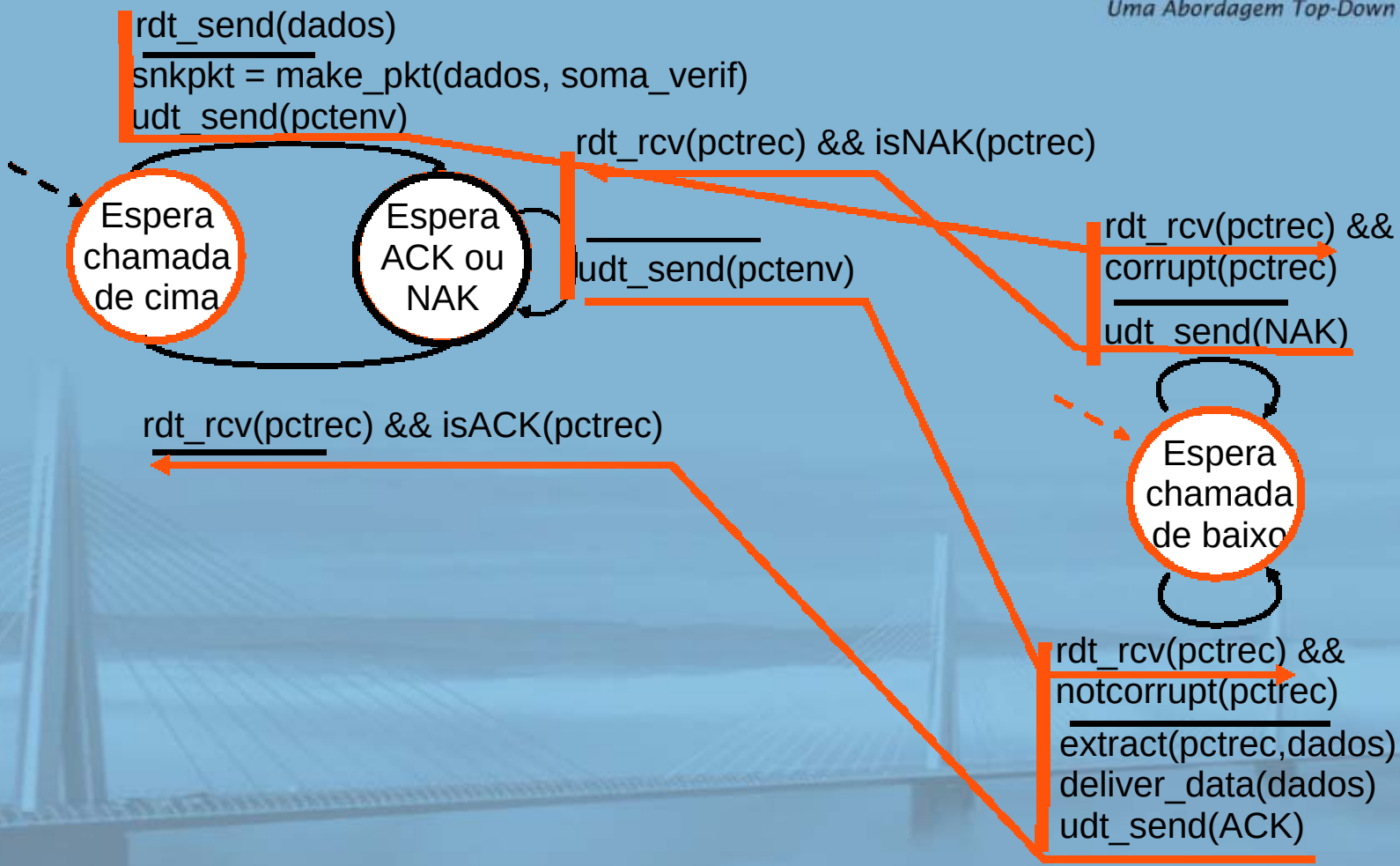
destinatário



rdt2.0: operação sem erros



rdt2.0: cenário de erro



rdt2.0 tem uma falha fatal!

O que acontece se ACK/NAK for corrompido?

- remetente não sabe o que aconteceu no destinatário!
- não pode simplesmente retransmitir: possível duplicação

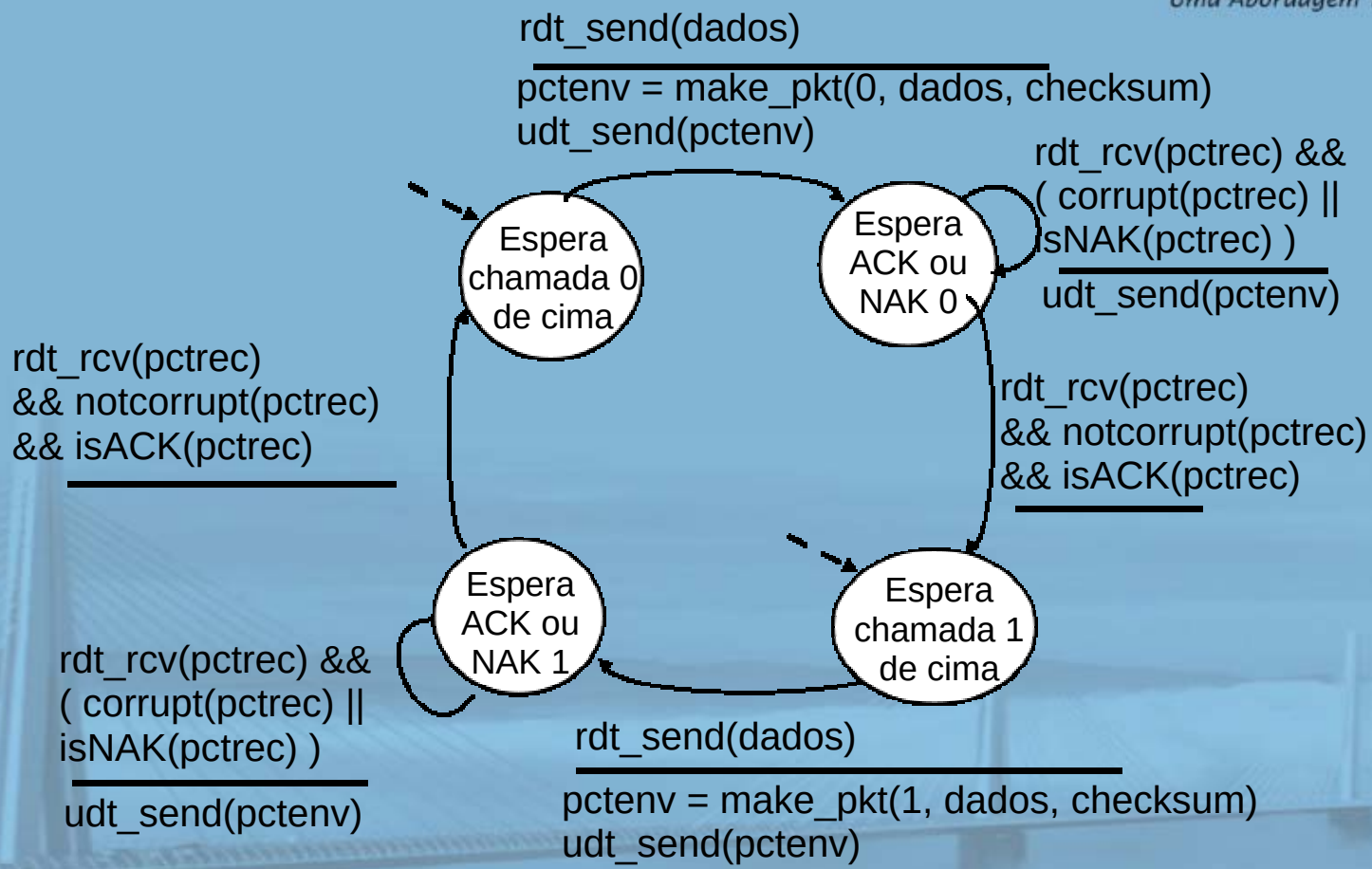
tratando de duplicatas:

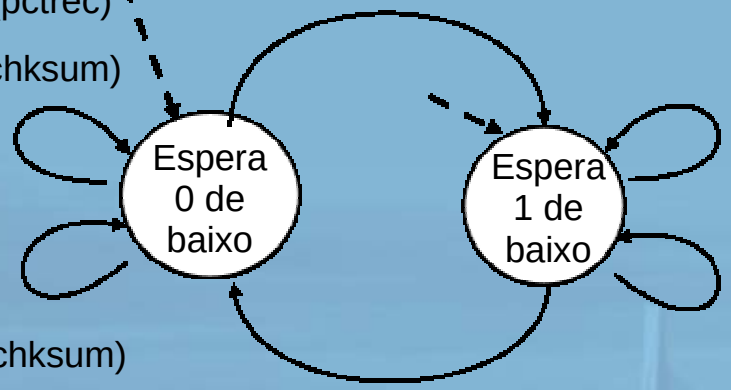
- remetente retransmite pacote atual se ACK/NAK corrompido
- remetente acrescenta *número de sequência* a cada pacote
- destinatário descarta (não sobe) pacote duplicado

pare e espere

remetente envia um pacote, depois espera resposta do destinatário

rdt2.1: remetente trata de ACK/NAKs corrompidos





rdt_rcv(pctrec) && notcorrupt(pctrec) && has_seq0(pctrec)

extract(pctrec,dados)
deliver_data(dados)
pctenv = make_pkt(ACK, chksum)
udt_send(pctenv)

rdt_rcv(pctrec) && (corrupt(pctrec)

pctenv = make_pkt(NAK, chksum)
udt_send(pctenv)

rdt_rcv(pctrec) && (corrupt(pctrec)

pctenv = make_pkt(NAK, chksum)
udt_send(pctenv)

rdt_rcv(pctrec) && not corrupt(pctrec) && has_seq1(pctrec)

pctenv = make_pkt(ACK, chksum)
udt_send(pctenv)

rdt_rcv(pctrec) && not corrupt(pctrec) && has_seq0(pctrec)

pctenv = make_pkt(ACK, chksum)
udt_send(pctenv)

rdt_rcv(pctrec) && notcorrupt(pctrec) && has_seq1(pctrec)

extract(pctrec,dados)
deliver_data(dados)
pctenv = make_pkt(ACK, chksum)
udt_send(pctenv)

rdt2.1: discussão

remetente:

- # seq acrescentado ao pkt
- dois #s seq. (0,1) bastarão. Repetido ou não.
- deve verificar se ACK/NAK recebido foi corrompido
- o dobro de estados
  - estado de “lembrar” se pacote “atual” tem # seq. 0 ou 1

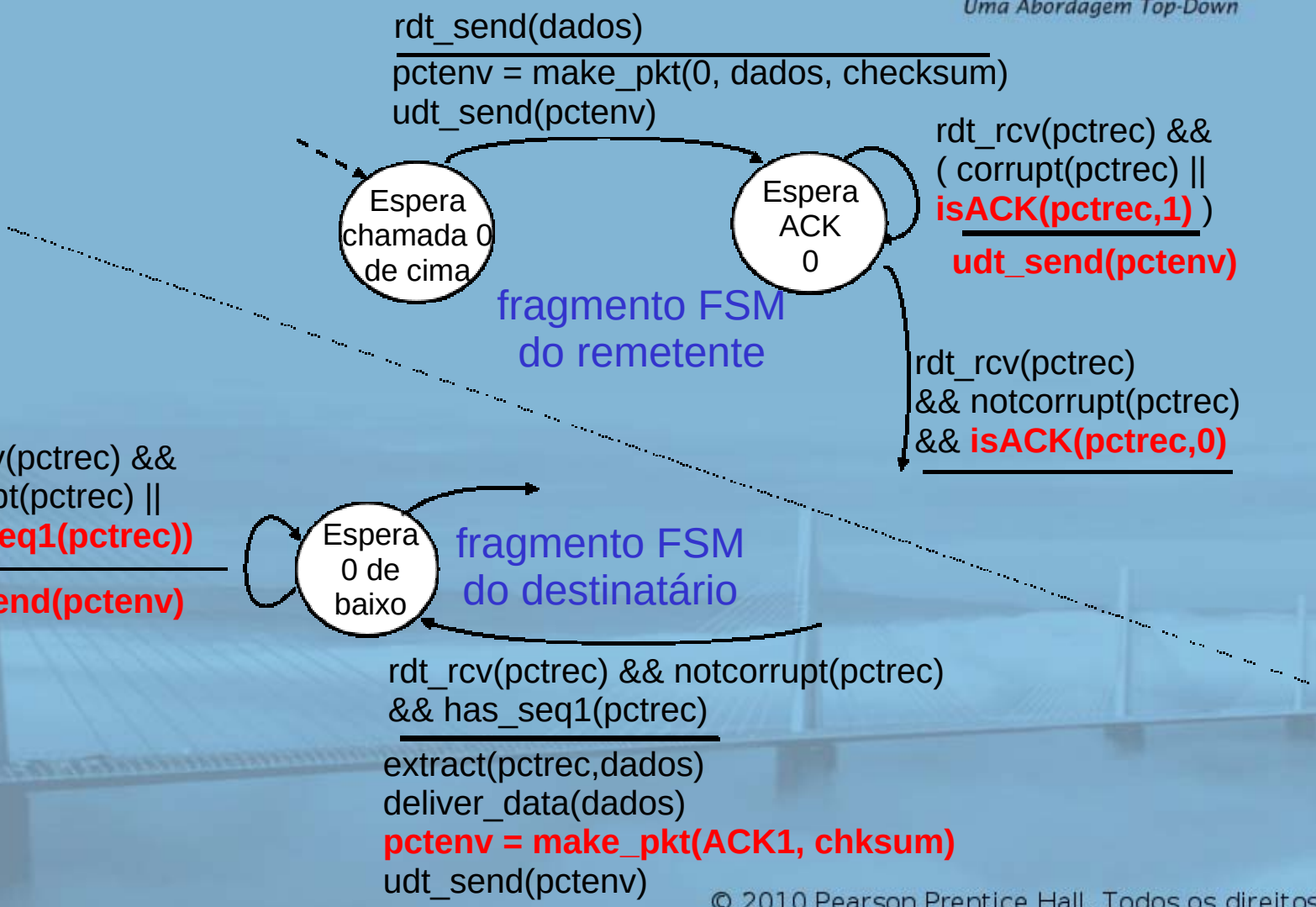
destinatário:

- deve verificar se pacote recebido está duplicado
  - estado indica se 0 ou 1 é # seq. esperado do pacote
- nota: destinatário *não* sabe se seu último ACK/NAK foi recebido OK no remetente
- ACK/NACK não precisam de #s seq.

rdt2.2: um protocolo sem NAK

- mesma funcionalidade de rdt2.1, usando apenas ACKs
- em vez de NAK, destinatário envia ACK para último pacote recebido OK
  - destinatário precisa incluir *explicitamente* # seq. do pacote sendo reconhecido com ACK
- ACK duplicado no remetente resulta na mesma ação de NAK: *retransmitir pacote atual*

rdt2.2: fragmentos do remetente, destinatário



rdt3.0: canais com erros e perda

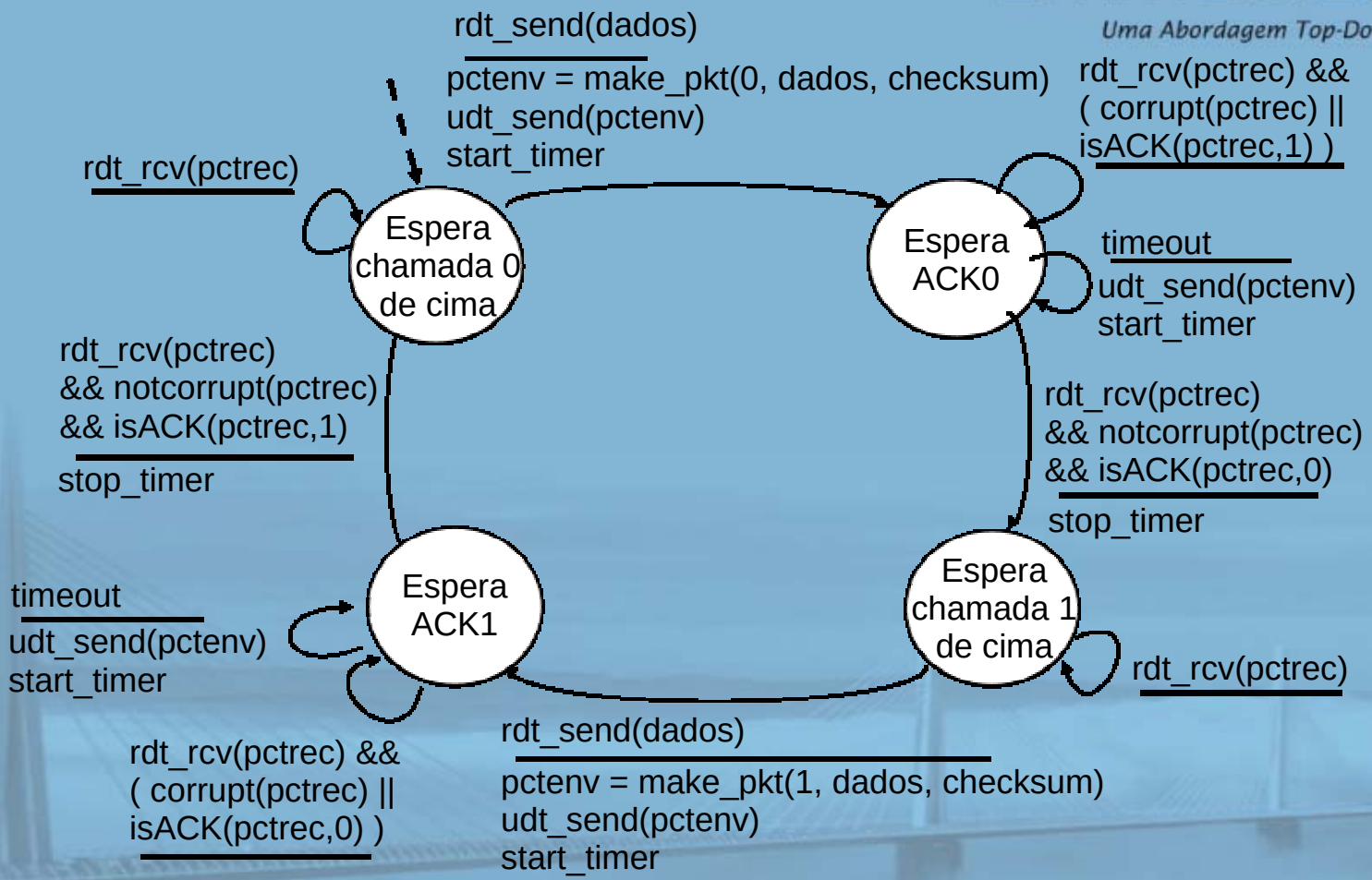
nova suposição: canal subjacente também pode perder pacotes (dados ou ACKs)

- soma de verificação, # seq., ACKs, retransmissões serão úteis, mas não suficientes

técnica: remetente espera quantidade "razoável" de tempo por ACK

- retransmite se não chegar ACK nesse tempo
- se pct (ou ACK) simplesmente atrasado (não perdido):
  - retransmissão será duplicada, mas os #s de seq. já cuidam disso
  - destinatário deve especificar # seq. do pacote sendo reconhecido com ACK
- requer contador regressivo

remetente rdt3.0



rdt3.0 em ação