Fundamentos de Redes de Computadores

Tiago Alves

Faculdade UnB Gama
Universidade de Brasília

UnB Gama
O novo endereço da tecnologia

Sumário

Camada de Transporte

UnB Gama
O novo endereço da tecnologia

Camada de Transporte

Provê **comunicação lógica** entre **processos da camada de aplicação** executando em diferentes hosts.

- **Comunicação lógica**: processos executando em diferentes plataformas como se estivessem conectados diretamente.
- **Processos de aplicação** usam a comunicação lógica provida pela camada de transporte para enviar mensagens uns aos outros, sem se preocupar com a infraestrutura física usada no suporte à ligação.

Camada de Transporte

Os protocolos de camada de transporte são implementados nos end systems, mas não em comutadores de pacotes (roteadores e switches).

No lado emissor, a camada de transporte converte mensagens da camada de aplicação em pacotes da camada de transporte: **segmentos**.

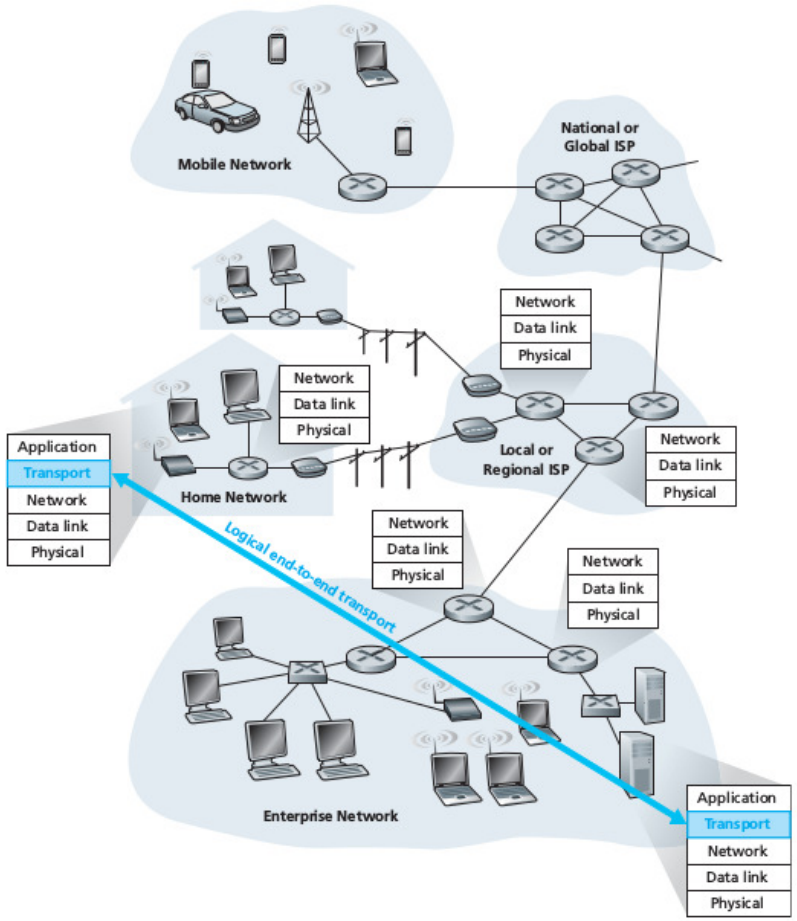
- quebra das mensagens em pedaços;
- adição de cabeçalho da camada de transporte em cada pedaço, o que cria **segmentos** da camada de transporte.
- encaminhamento dos segmentos para a camada de rede
- segmento é encapsulado em um pacote da camada de rede (**datagrama**) e encaminhado ao destino pelo enlace.

Quando há a interação com roteadores, esses agem apenas nos campos de cabeçalho da camada de rede.

Camada de Transporte

No receptor:

- a camada de rede extrai o segmento da camada de transporte do datagrama e passa o segmento para a camada de transporte.
- A camada de transporte disponibiliza o segmento para a camada de aplicação.


Mobile Network
National or Global ISP
Network
Data link
Physical
Application
Transport
Network
Data link
Physical
Home Network
Local or Regional ISP
Logical end-to-end transport
Enterprise Network

Camada de Transporte

Mais de um protocolo de camada de transporte pode estar disponíveis para aplicações: **TCP** e **UDP**.

2 casas, cada uma com 12 crianças

As crianças de cada casa são irmãs. Qualquer criança de uma das casas é prima de qualquer criança da outra casa.

- mensagens da camada de aplicação = cartas nos envelopes
- processos = primos
- hosts = casas
- protocolo de camada de transporte = Ann and Bill (Susan and Harvey - UDP)
- protocolo de camada de rede = serviço postal

Os serviços da camada de transporte dependem, de certa forma, das limitações impostas pelos serviços da camada de rede (melhor esforço).

Visão geral da camada de transporte na Internet

UDP: serviços de transmissão não confiáveis, sem estabelecimento de conexão.

TCP: serviços de transmissão confiáveis, orientado a conexão

Pacote da camada de transporte: **segmento**.

ATENÇÃO: os pacotes UDP são chamados de **datagramas**. Mas isso não é ambíguo, pois os pacotes de camada de rede são chamados, também, de datagramas?

Protocolo da camada de rede na Internet: Internet Protocol (IP)

- Provê a ligação lógica entre hosts. Serviço de entrega de melhor esforço: não há garantias. Serviço não confiável.
- Cada host possui, pelo menos, um endereço IP (endereço de camada de rede).

Relação entre protocolos da camada de transporte e o protocolo IP

Extensão dos serviços IP: estende serviço de entrega entre dois **end systems** para serviço de entrega entre **dois processos** executando nos dois end systems.

- multiplexação e demultiplexação do canal lógico.
- checagem de integridade: campos para detecção de erros nos cabeçalhos de segmentos

Os dois serviços acima são providos pelo **UDP**.

Protocolos da camada de transporte e IP

TCP vai além:

- **transmissão confiável de dados**: controle de fluxo, numeração de sequência, acknowledgments, temporizadores garantem dados entregues ao processo receptor de forma correta (íntegra) e ordenada. É o TCP que provê serviço de entrega de dados confiáveis sobre IP.
- **controle de congestionamento**:
  - previne que uma conexão TCP atole (em tráfego) os enlaces e roteadores que ligam os hosts comunicantes.
  - tenta garantir que cada conexão TCP atravessando um enlace congestionado receba uma parcela igual da largura de banda do enlace.

Comparando com UDP, o TCP é um protocolo **mais complexo**!

Introdução

No host de destino, a camada de transporte recebe segmentos da camada de rede.

A camada de transporte encaminha segmentos ao processo de aplicação.

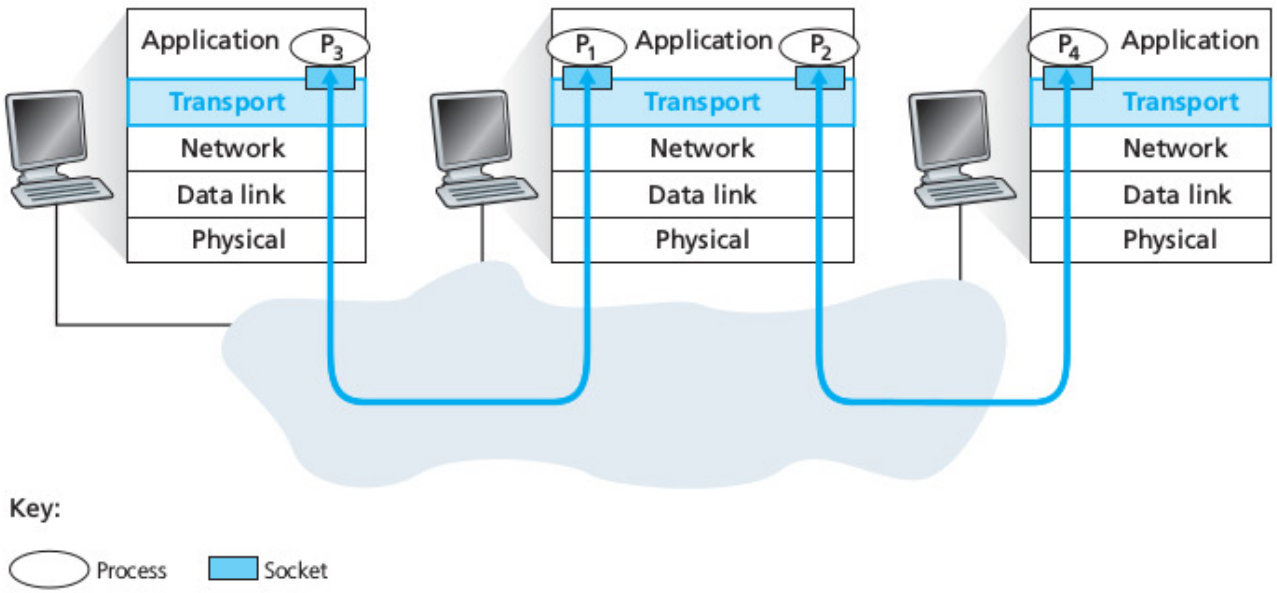
- A camada de transporte entrega dados através de **sockets** aos processos. Sockets possuem identificadores: tipo (TCP, UDP), porta...

Demultiplexação (receptor): a camada de transporte examina cabeçalho do segmento para identificar qual é o socket receptor. Uma vez identificado, o segmento é direcionado ao socket.

Multiplexação (emissor): transporte coleta pedaços de dados na fonte, a partir de sockets diferentes, encapsula cada pedaço de dado em segmentos. Passa segmentos à camada de rede. Datagramas são marcados com o mesmo IP.

De/Multiplexação acontecerá sempre que vários protocolos de camadas superiores usem os serviços de uma camada inferior

Multiplexação e Demultiplexação

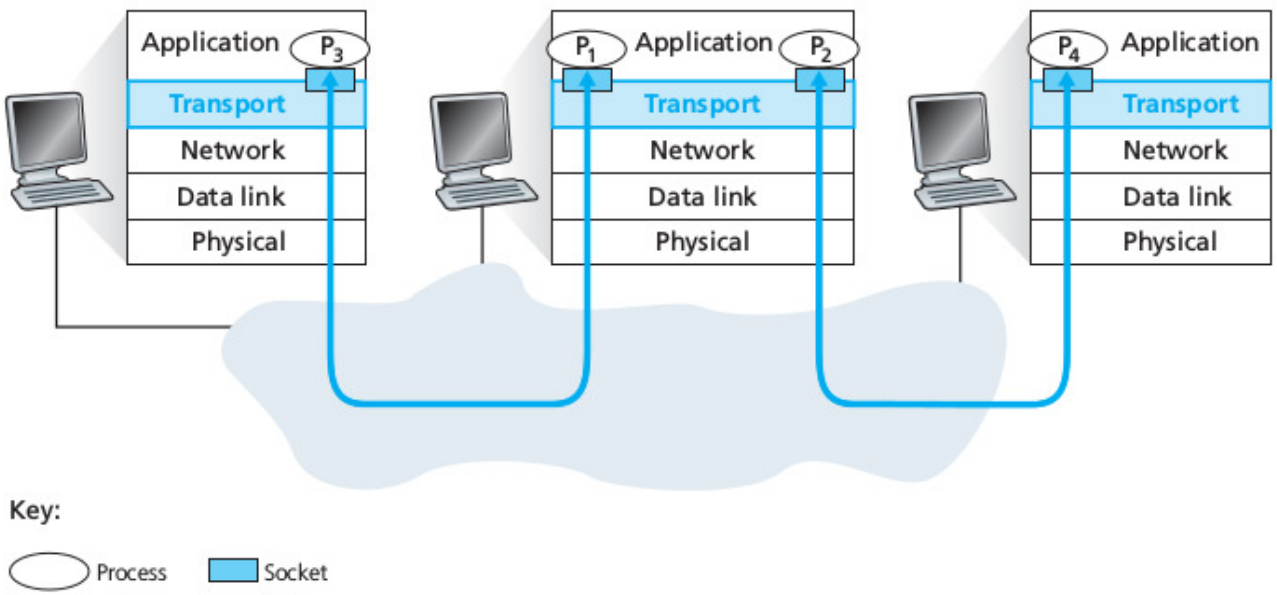


Introdução

Porta de origem e de destino: número de 16 bits, de 0 a 65535.

- de 0 a 1023 são portas restritas. RFCs 1700 e 3232. (/etc/services)
- cada aplicação de rede deve possuir seu conjunto de portas determinado.

As portas são dispositivos que viabilizam o serviço de demultiplexação.



MuxDemux sem conexão

Primitiva (cliente):

```
clientSocket = socket(socket.AF_INET, socket.SOCK_DGRAM)
```

Porta alta (> 1024) associada ao socket cliente: escolhe-se uma porta que não esteja em uso.

É possível vincular (explicitamente) uma porta de saída do cliente.

```
clientSocket.bind(('', 19157))
```

Servidor (elemento "passivo") costuma possuir porta **fixa**!

MuxDemux sem conexão

Cliente (python):

```
from socket import *
# importa todas as funções providas pela biblioteca socket
serverName = 'hostname' # ajuste de variáveis
serverPort = 12000
clientSocket = socket(socket.AF_INET, socket.SOCK_DGRAM)
# cria o socket
# AF_INET: IPv4
# SOCK_DGRAM: UDP
message = raw_input('Input lowercase sentence:')
# coleta a mensagem digitada pelo usuário
clientSocket.sendto(message,(serverName, serverPort))
# envia a mensagem (nao se estabelece conexao!)
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
# espera resposta do servidor
print modifiedMessage
# imprime mensagem
clientSocket.close()
# encerra o socket
```

MuxDemux sem conexão

Servidor (python):

```
from socket import *
# importa todas as funções providas pela biblioteca socket
serverPort = 12000 # ajuste de variáveis
serverSocket = socket(AF_INET, SOCK_DGRAM)
# cria o socket
# AF_INET: IPv4
# SOCK_DGRAM: UDP
serverSocket.bind(('', serverPort))
# vincula a porta ao processo do servidor
print "The server is ready to receive"
while 1:
  message, clientAddress = serverSocket.recvfrom(2048)
  # espera por requisicao (nao se estabelece conexao!)
  modifiedMessage = message.upper()
  # converte a mensagem recebida por socket UDP em caixa ALTA
  serverSocket.sendto(modifiedMessage, clientAddress)
  # devolve para o cliente a mensagem em caixa ALTA
```

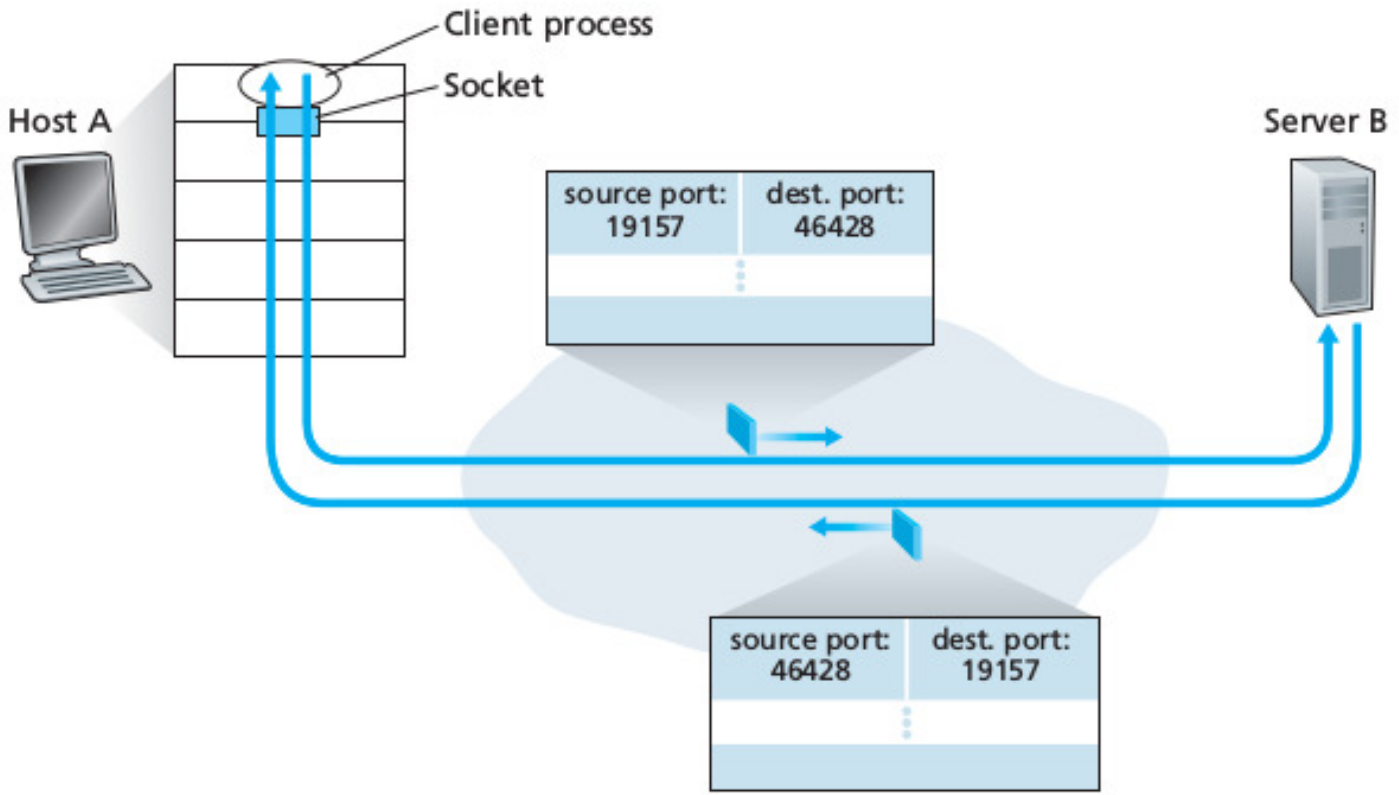
MuxDemux sem conexão

Cenário: Host A, UDP 19157 que enviar dados de aplicação para UDP 46428 em Host B.

- camada de transporte de Host A cria um segmento que inclui dados de aplicação (**payload**), porta de origem (19157), porta de destino (46428) e mais dois outros valores (IPs, cenas dos próximos capítulos!).
- camada de transporte passa o segmento para a camada de rede. O segmento é encapsulado em um datagrama e a camada de rede **faz o melhor esforço** para tentar entregar o segmento ao host de destino
- **se o segmento chegar** ao Host B, a camada de transporte examinará a porta de destino (46428) e entregará o segmento ao socket vinculado a essa porta. Quando segmentos UDP chegam pela rede, Host B demultiplexa cada segmento ao socket apropriado examinando a porta de destino do segmento

Atenção: não há estabelecimento de conexão!

Multiplexação e Demultiplexação

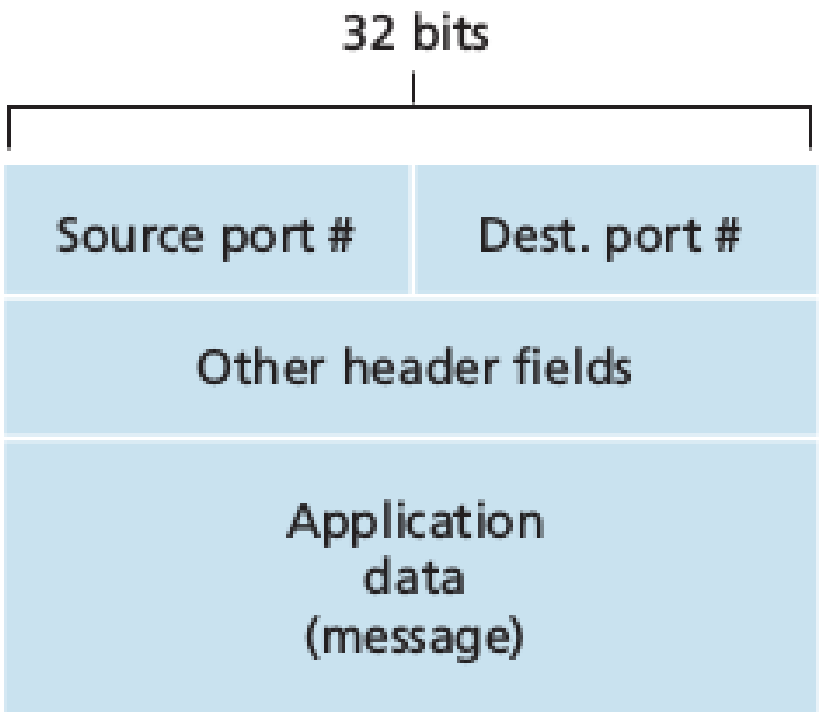


MuxDemux sem conexão

Um socket UDP é completamente identificado por **duplas** consistindo em porta de origem e porta de destino!

O número de porta de origem serve como endereço de retorno.

Como o destino saberá **para quem (para qual IP)** deverá responder?



MuxDemux orientado a conexão

Diferença sutil entre sockets TCP e UDP: socket TCP é identificado por uma **quádrupla**: IP e porta de origem e IP e porta de destino!

O host usa todas as quatro informações para demultiplexar o segmento para o socket apropriado.

MuxDemux orientado a conexão

Primitiva (cliente):

```
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,12000)
```

Primitiva (servidor): a criação de socket de conexão no servidor (depende da quádrupla)

```
connectionSocket, addr = serverSocket.accept()
```

MuxDemux orientado a conexão

Cliente:

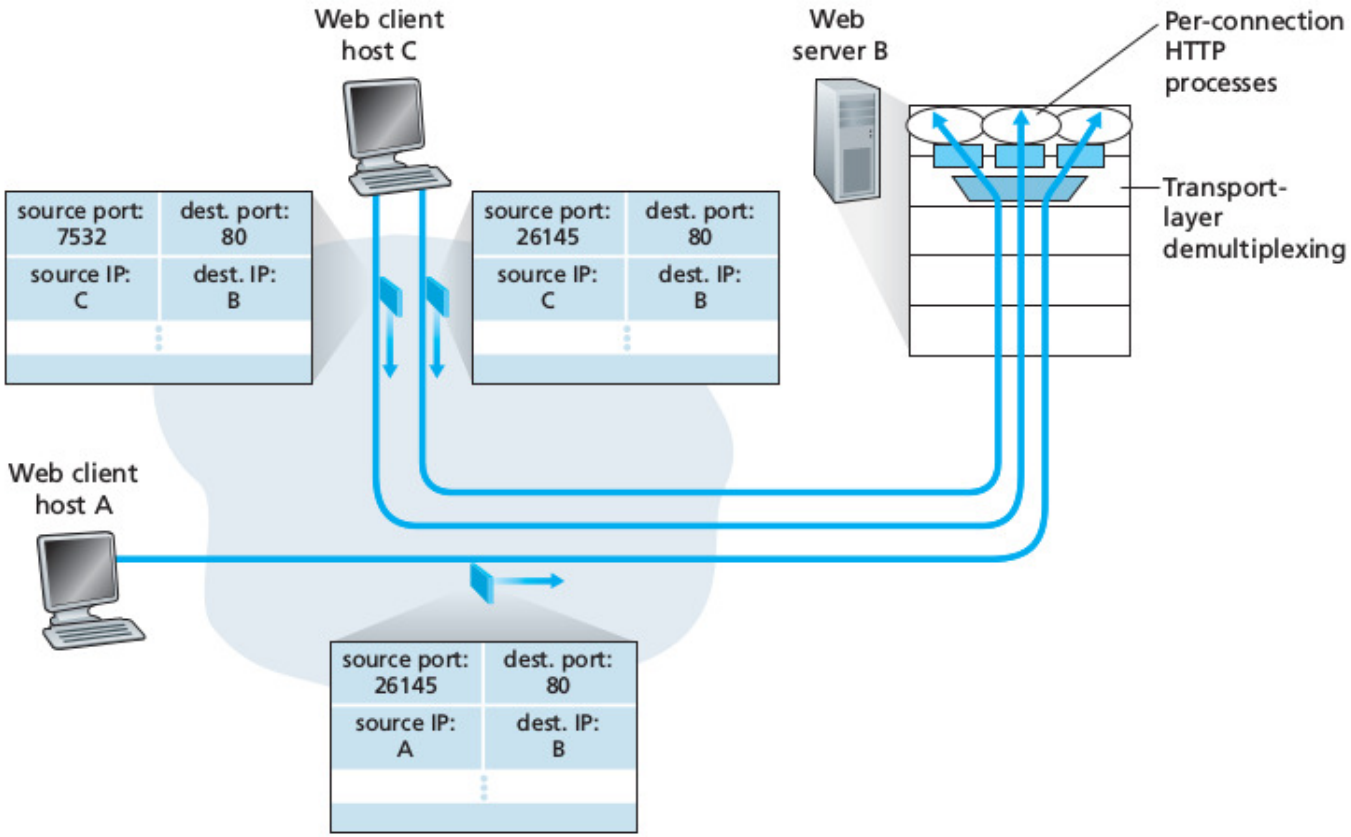
```
from socket import *
# importa todas as funções providas pela biblioteca socket
serverName = 'servername'
serverPort = 12000
# ajuste de variáveis
clientSocket = socket(AF_INET, SOCK_STREAM)# cria o socket
# AF_INET: IPv4
# SOCK_STREAM: TCP
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase sentence:')
# coleta a mensagem digitada pelo usuário
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1024)
# espera resposta do servidor
print 'From Server:', modifiedSentence
# imprime mensagem
clientSocket.close()
# encerra o socket
```

MuxDemux orientado a conexão

Servidor:

```
from socket import *
# importa todas as funções providas pela biblioteca socket
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
# ajuste de variáveis
serverSocket.bind(('',serverPort))
# vincula a porta ao processo do servidor
serverSocket.listen(1)
print 'The server is ready to receive'
while 1:
  connectionSocket, addr = serverSocket.accept()
  # espera por requisicao
  sentence = connectionSocket.recv(1024)
  # converte a mensagem recebida por socket TCP em caixa ALTA
  capitalizedSentence = sentence.upper()
  connectionSocket.send(capitalizedSentence)
  # devolve para o cliente a mensagem em caixa ALTA
  connectionSocket.close()
```

Multiplexação e Demultiplexação



MuxDemux orientado a conexão: Hands On

Nmap

Servidores Web e TCP

Servidores com conexões não persistentes: nova conexão TCP (e novo socket) é criada e fechada para cada requisição/resposta!

Efeitos colaterais: 2 RTTs para cada estabelecimento de conexão e recursos alocados no servidor para atendimento de cada conexão.

Introdução

Definido na RFC 768.

- Serviços básicos: multiplexação/demultiplexação e checagem de erros. Adicionados ao básico provido pelo protocolo IP.
- Interfaceamento direto com o protocolo IP

Introdução

Etapas numa transmissão UDP:

1. UDP coleta as mensagens do processo de aplicação, adiciona porta de origem e destino, adiciona mais dois outros campos menores e passa o segmento resultante para a camada de rede
2. Camada de rede encapsula o segmento em datagrama IP e usa do melhor esforço para entregar o segmento ao destino
3. No host de destino, é usada a informação de porta de destino para entregar os dados do segmento ao devido processo de aplicação.

Não há handshaking: sem estabelecimento de conexão!

Introdução

Aplicação que depende de UDP: DNS.

Em caso de perda: reenvio de requisição!

Introdução

Quando UDP é mais conveniente?

- Controle fino, em camada de aplicação, sobre **que dado** é enviado e **quando o dado** é enviado
  - Evita a aplicação de mecanismos de controle de congestionamento sobre os dados (o que é tipicamente aplicado quando se opera sobre TCP)
  - Evita retransmissões típicas das camada de transporte.
  - Convenientemente aplicado no provimento de aplicações tempo-real: implementador da aplicação deverá implementar os serviços extras (além do UDP) em nível de aplicação para atendimento de requisitos.
- Não há estabelecimento de conexão: não há handshaking (3-way handshaking). **Não há atraso devido ao estabelecimento de conexão**.

Introdução

Quando UDP é mais conveniente?

- Não há estado de conexão
  - TCP mantém estado de conexão em ambos os end systems: buffers de saída e de entrada, parâmetros de controle de congestionamento e número de sequência e de acknowledgment.
  - Convenientemente aplicado no provimento de aplicações tempo-real
- Pequeno overhead em função de cabeçalho de pacote: apenas 8 bytes além do payload.

Introdução

Aplicações que usam UDP:

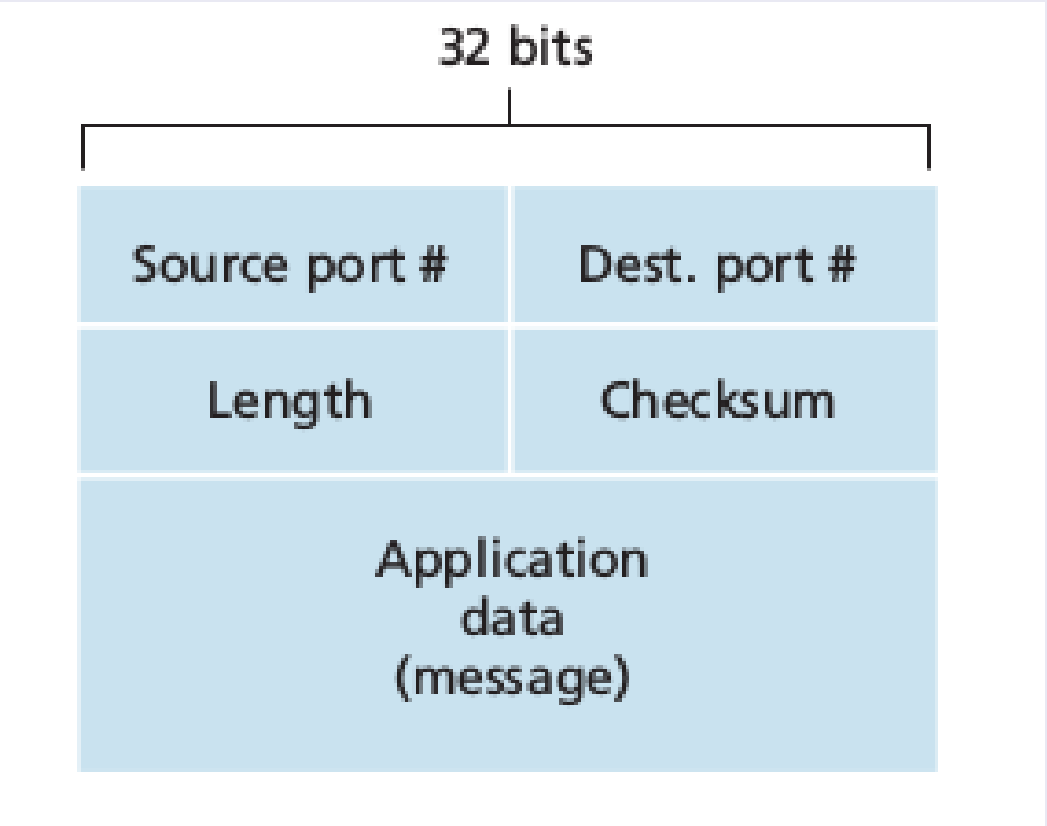
- RIP (protocolo de roteamento)
- SNMP
- DNS

Aplicações multimídia usam mecanismos híbridos de transporte: TCP e UDP.

Tráfego UDP é bloqueado por algumas organizações: atualmente, há mais serviços de aplicação multimídia operando sobre TCP.

Estrutura do segmento UDP

Definido na RFC 768.

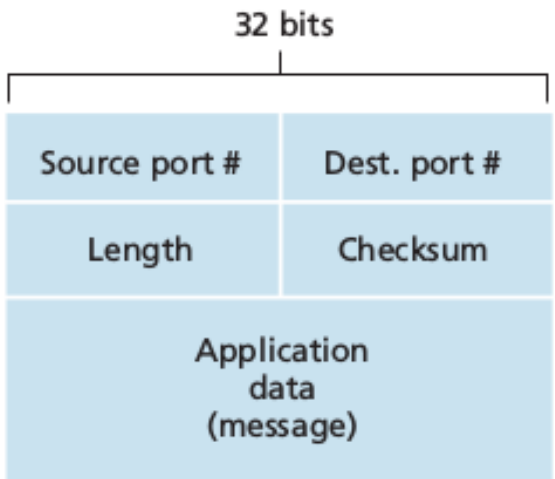


Estrutura do segmento UDP

Dado de aplicação ocupa o campo de dados (payload) do segmento UDP

Cabeçalho UDP possui apenas 4 campos, cada um de 2 bytes:

- porta de origem;
- porta de destino;
- length (header + dados);
- checksum: controle do payload.



UnB Gama
O novo endereço da tecnologia

Estrutura do segmento UDP

Checksum UDP: RFC 1071

- Soma em **complemento de 1** de todas as palavras de 16 bits com overflow realimentado como bit menos significativo.
- Complemento de 1 irá inverter todos os bits do resultado
- O resultado da soma no receptor deverá gerar FFFF!

O checksum UDP existe porque não há garantias de que os protocolos mais baixos proverão checagem de erro.

Embora UDP realize a checagem de erros, não há mecanismos para recuperação do erro: alguma versões simplesmente descartam o segmento corrompido; outras, passam o segmento para a aplicação, mas com um aviso.

Soma de conferência do UDP

0110011001100000
0101010101010101
1000111100001100

The sum of first two of these 16-bit words is

0110011001100000
0101010101010101
1011101110110101

Adding the third word to the above sum gives

1011101110110101
1000111100001100
0100101011000010

Introdução

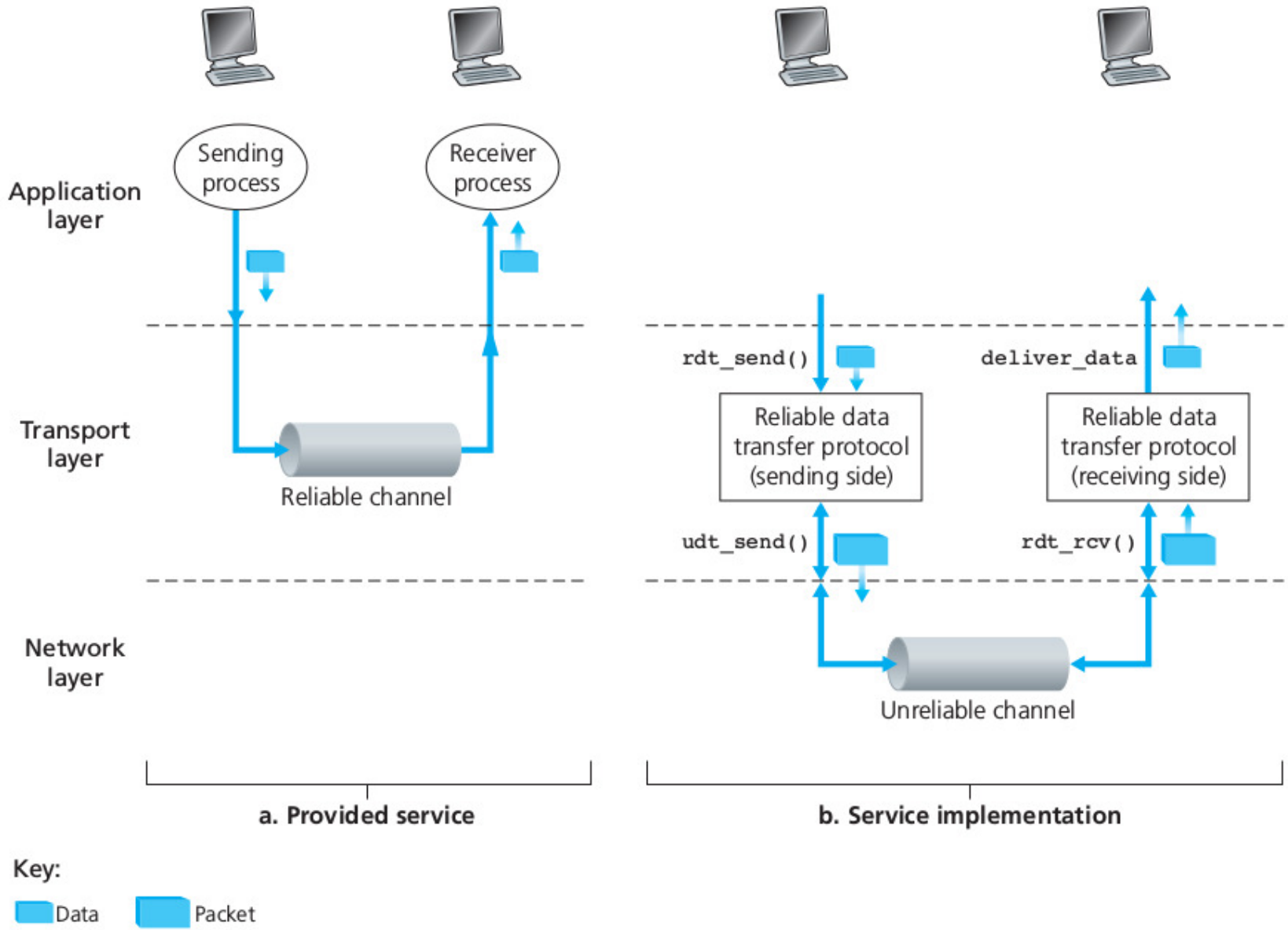
Transferência confiável de dados: um dos 10 problemas mais importantes em comunicações

Arcabouço:

- A abstração de serviço provida por entidades de camadas superiores é de um **canal confiável** através do qual os dados podem ser transferidos.
- **Nenhum bit** dos dados é **corrompido** (trocado de 0 para 1 ou de 1 para 0) ou **perdido**.
- Todos os bits são entregues **na ordem** em que foram enviados.

Protocolo de transporte implementa essa abstração de serviço sobre um agregado de outros protocolos que podem ser não-confiáveis: TCP sobre IP.

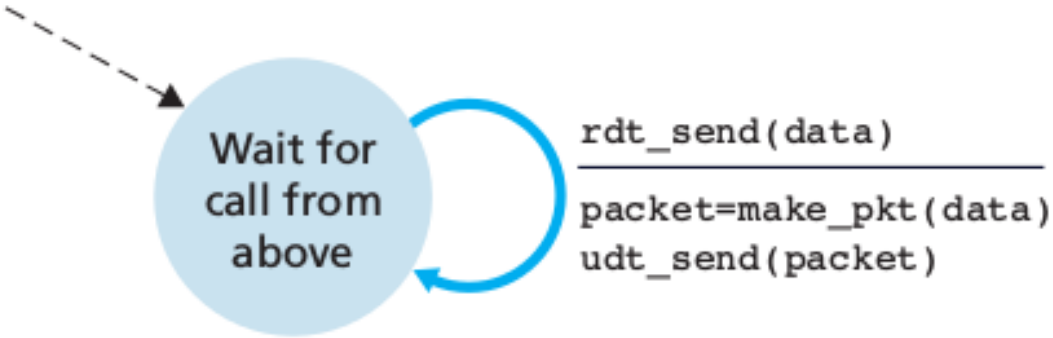
Princípios de Transmissão Confiável de Dados



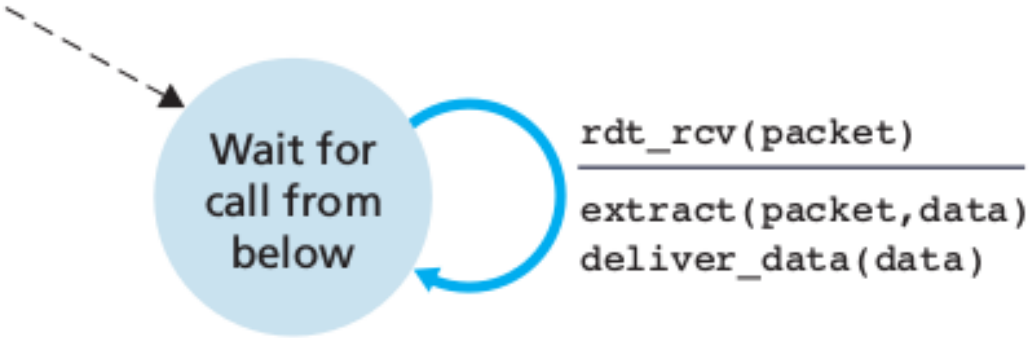
Construindo um protocolo de transmissão de dados confiável

rdt1.0: modelo de canal **perfeitamente confiável**. Não há perda nem corrupção dos pacotes.

- esquema de análise/representação: máquina de estados finitos.
- evento ou comando rdt_send(data) cria pacote contendo dados (ação make_pkt(data)) e envia pacotes pelo canal. Isso é resultado de uma chamada a rdt_send() na camada de aplicação
- evento rdt_rvc(packet) é sinalizado pelo canal. Por sua vez, há uma remoção extract(packet, data) do dado do pacote e entrega à camada superior através de deliver_data(data)
- Não há diferença entre uma unidade de dados e um pacote.



a. rdt1.0: sending side



b. rdt1.0: receiving side

Construindo um protocolo de transmissão de dados confiável

rdt2.0: canal com erros!

- Modelo mais realista: o canal pode provocar erros nos bits dos pacotes. Restrição: a ordem de recepção é a mesma que a ordem de transmissão.
- Chamada telefônica: acknowledgments positivos e acknowledgments negativos são sinalizados pelo interlocutor
- ARQ: automatic repeat request (protocols)

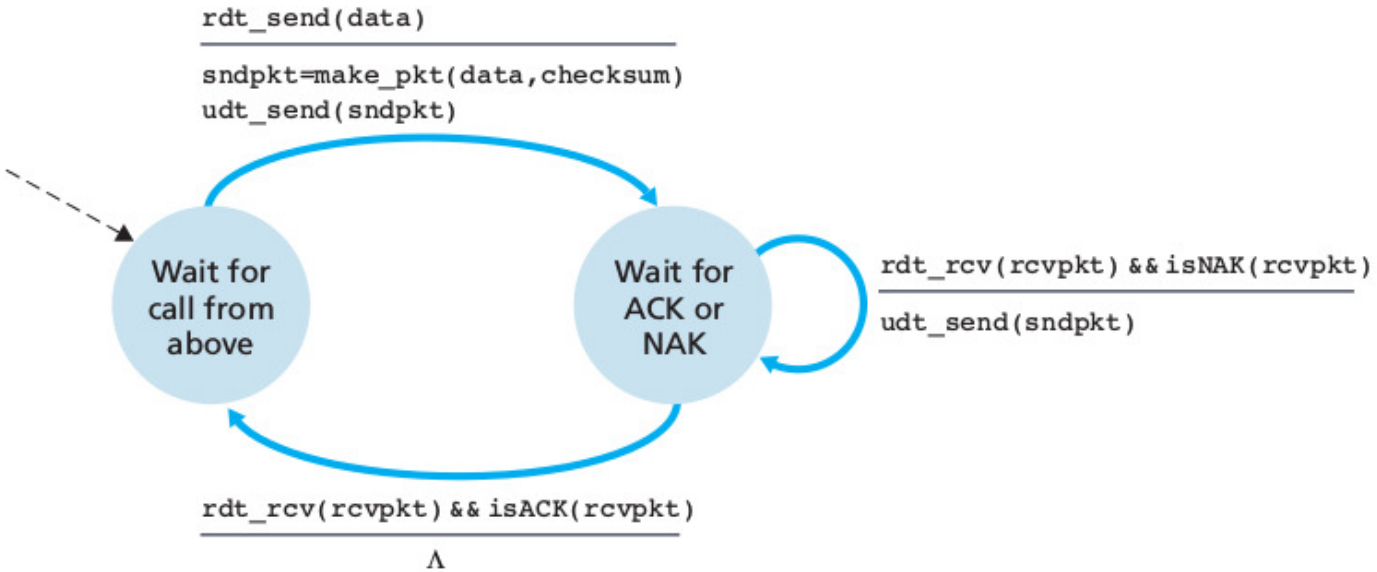
Construindo um protocolo de transmissão de dados confiável

rdt2.0: canal com erros

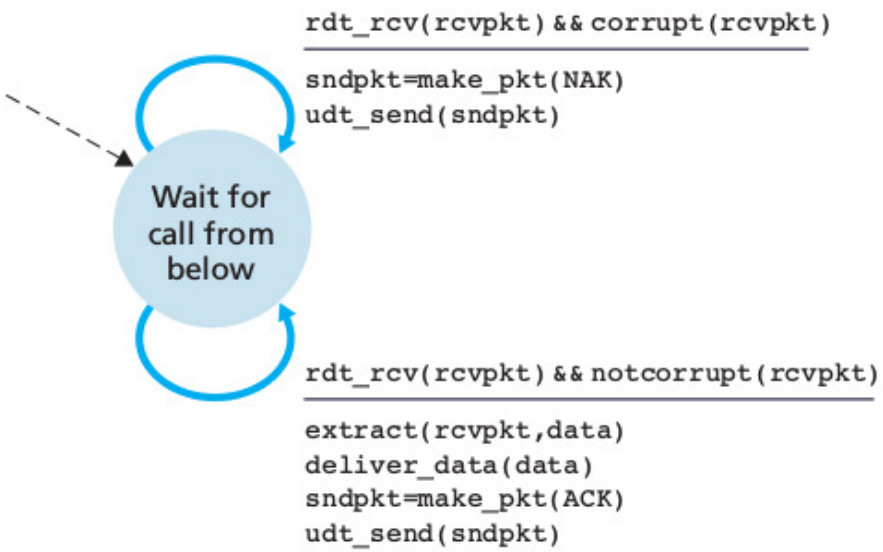
Capacidades (funcionalidades) necessárias à implementação de ARQ:

- **detecção de erros**: mecanismo para permitir o receptor detectar erros na transmissão. Bits extra adicionados e emitidos junto com a mensagem; esses bits serão agregados em um campo de checksum do pacote para rdt2.0
- **retorno/confirmação do receptor**: como os hosts estão separados fisicamente, a única forma de o transmissor tomar conhecimento do estado do receptor é através de uma resposta explícita: ACK e NACK. Pacotes de 1 bit de payload são suficientes para essa sinalização.
- **retransmissão**: se um pacote for recebido com erro no receptor, deverá ser retransmitido pelo emissor.

Protocolo do tipo **pára-e-espera**.



a. rdt2.0: sending side



b. rdt2.0: receiving side

Construindo um protocolo de transmissão de dados confiável

rdt2.0: canal com erros

Problema: e se houver um problema com a transmissão de ACK e NACK?

1. O que você falou? (por parte do emissor) (ACK - Okay; NACK, Por favor, repita). Mais uma mensagem no protocolo (que pode estar corrompida!).
2. Adicionar mecanismos de detecção e correção de erros.
3. Reenvio de pacote até uma correta recepção de ACK ou NACK. Problema: o ACK ou NACK recebido é refere-se a qual das transmissões?

Construindo um protocolo de transmissão de dados confiável

rdt2.0: canal com erros

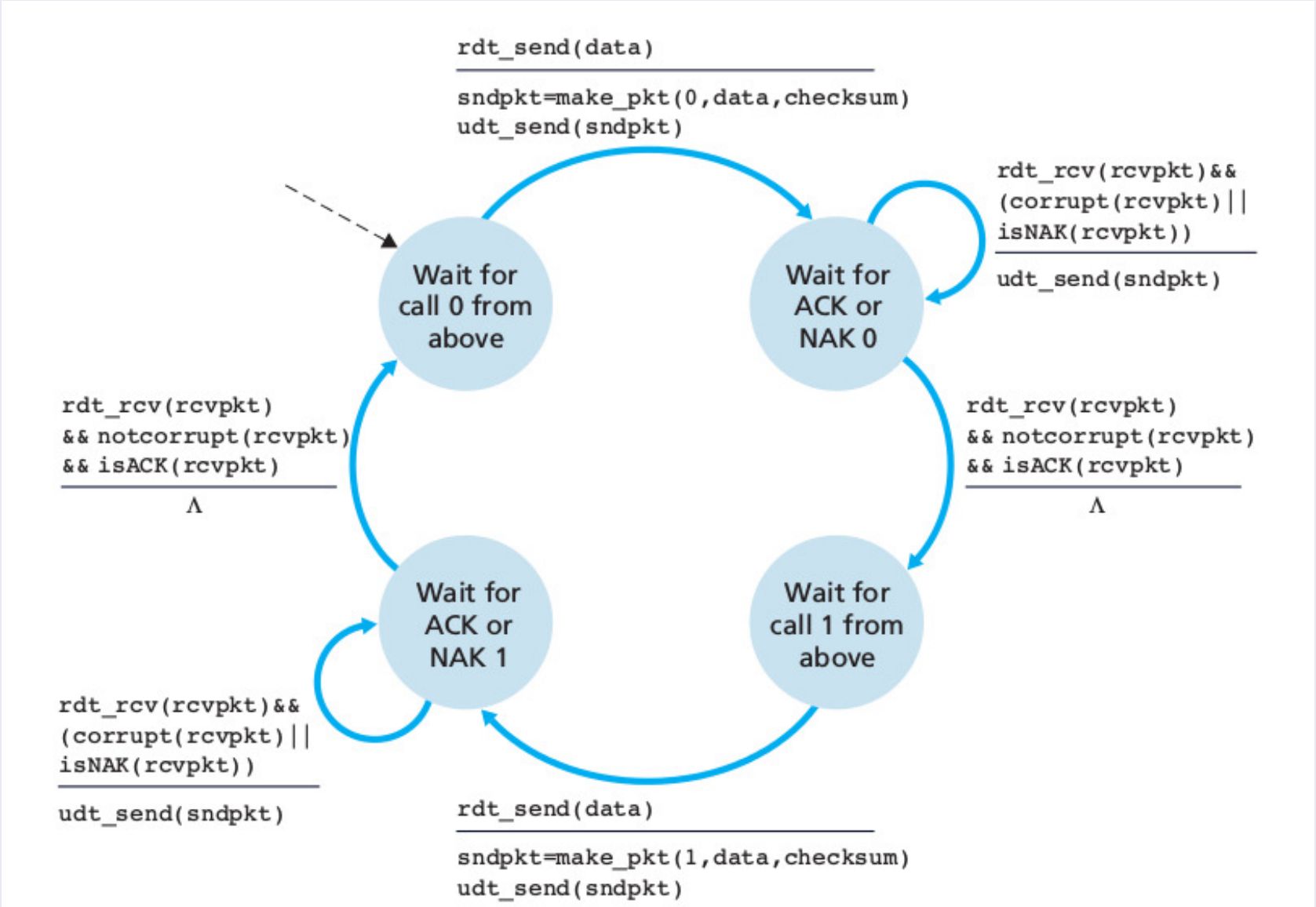
Solução:

- Novo campo no pacote: **número de sequência**. O receptor precisará apenas checar esse campo (número de sequência) para determinar se o pacote recebido é uma retransmissão.

Nova versão: rdt2.1!

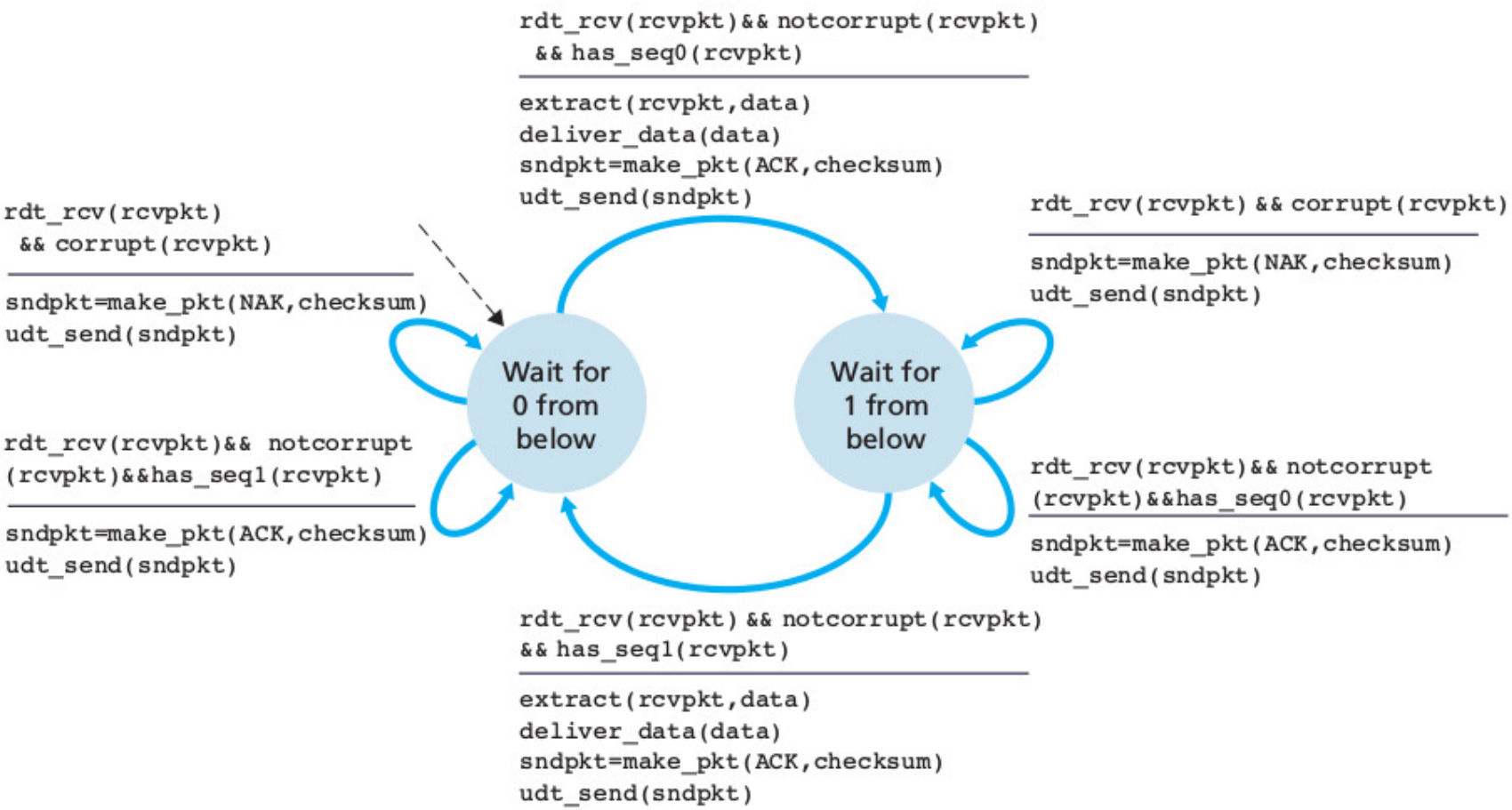
Construindo um protocolo de transmissão de dados confiável

rdt2.1: número de sequência, NACK e ACK. Visão da máquina pelo emissor.



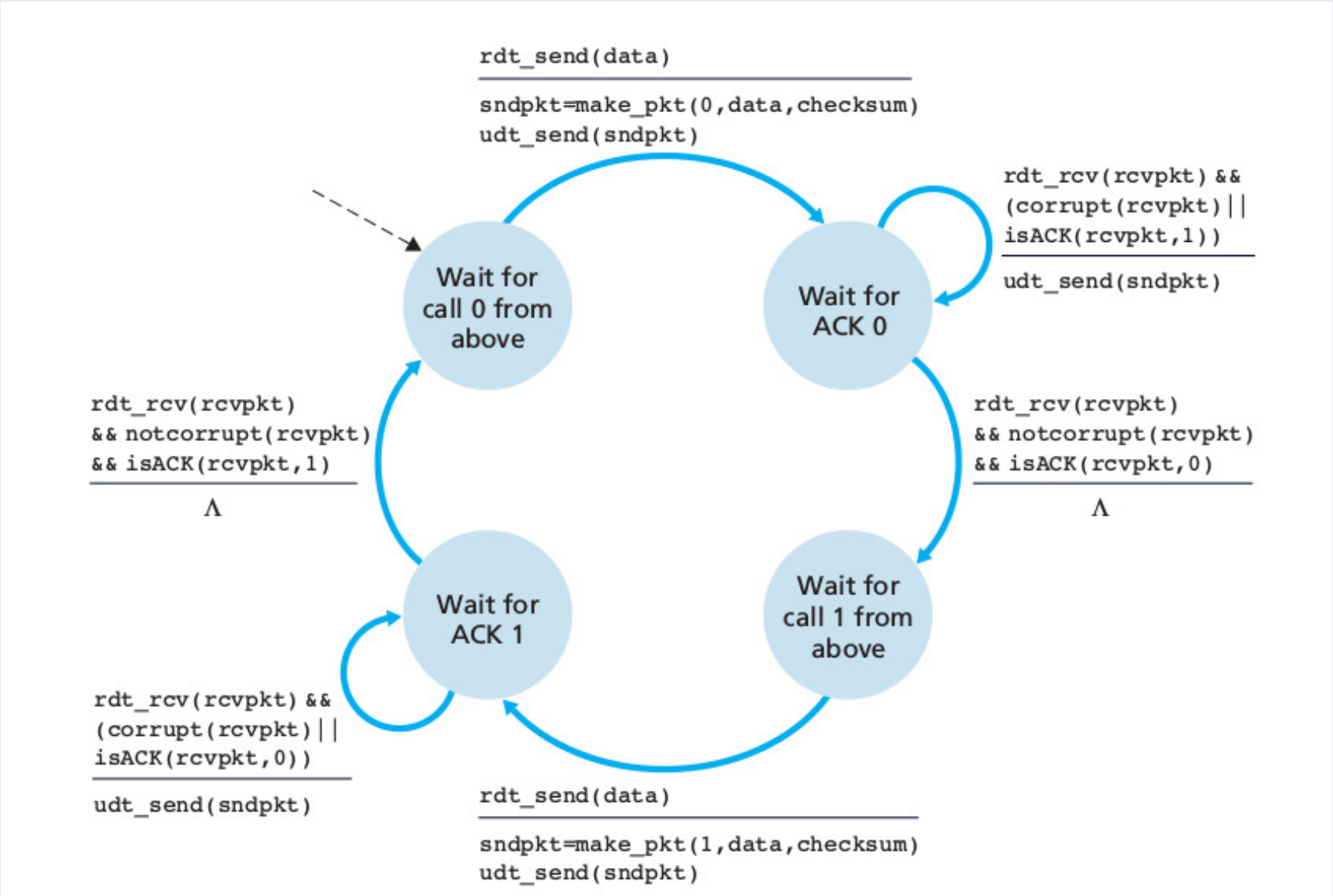
Construindo um protocolo de transmissão de dados confiável

rdt2.1: número de sequência, NACK e ACK. Visão da máquina pelo receptor.



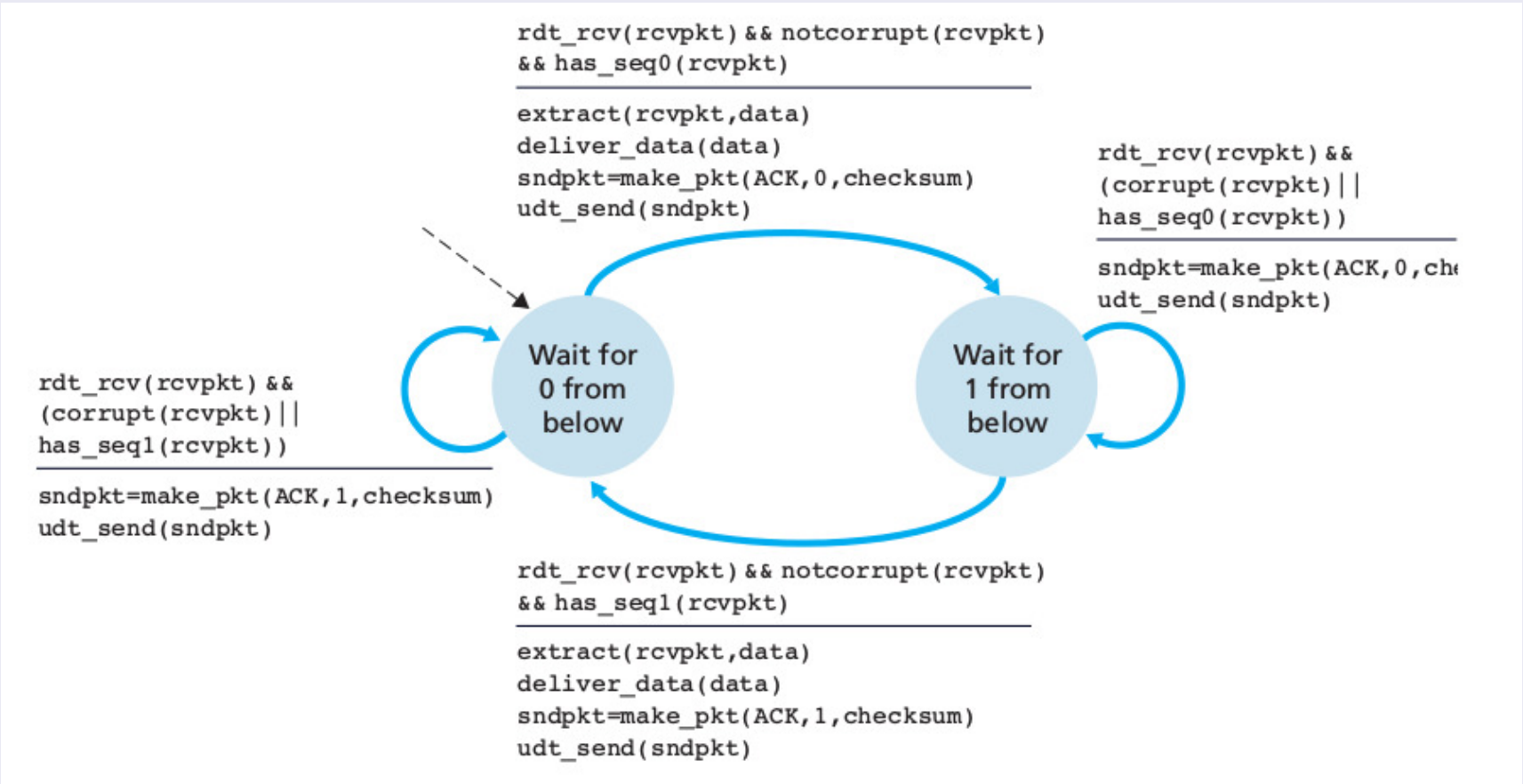
Construindo um protocolo de transmissão de dados confiável

rdt2.2: número de sequência e ACK (NACK implícito). Visão da máquina pelo emissor.



Construindo um protocolo de transmissão de dados confiável

rdt2.2: número de sequência e ACK (NACK implícito). Visão da máquina pelo receptor.



Construindo um protocolo de transmissão de dados confiável

rdt3.0: canal com **perdas** e **erros**!

- O modelo de canal é um pouco mais realista: ambiente agressivo provoca perda de pacotes.
  - Como detectar perda de pacotes?
  - O que fazer quando ocorre a perda de pacotes?
- Que tal arbitrar um **timeout** que, se alcançado, indica que houve perdas na transmissão?
  - Okay. **Q**: Mas qual o valor conveniente para o **timeout**?
  - **A**: A duração de uma round-trip entre o emissor e o receptor mais uma parcela de tempo necessária para processar o pacote no receptor.
- Um valor adequado deve **prevenir** esperas muito longas e **evitar** a retransmissão no caso em que o pacote gasta muito tempo para ser enviado (duplicação de pacote).

Construindo um protocolo de transmissão de dados confiável

rdt3.0: canal com perdas e erros

Novo mecanismo: cronômetro regressivo (temporizador), que avisa ao emissor sempre que o tempo esperado ultrapassou o valor arbitrado para timeout.

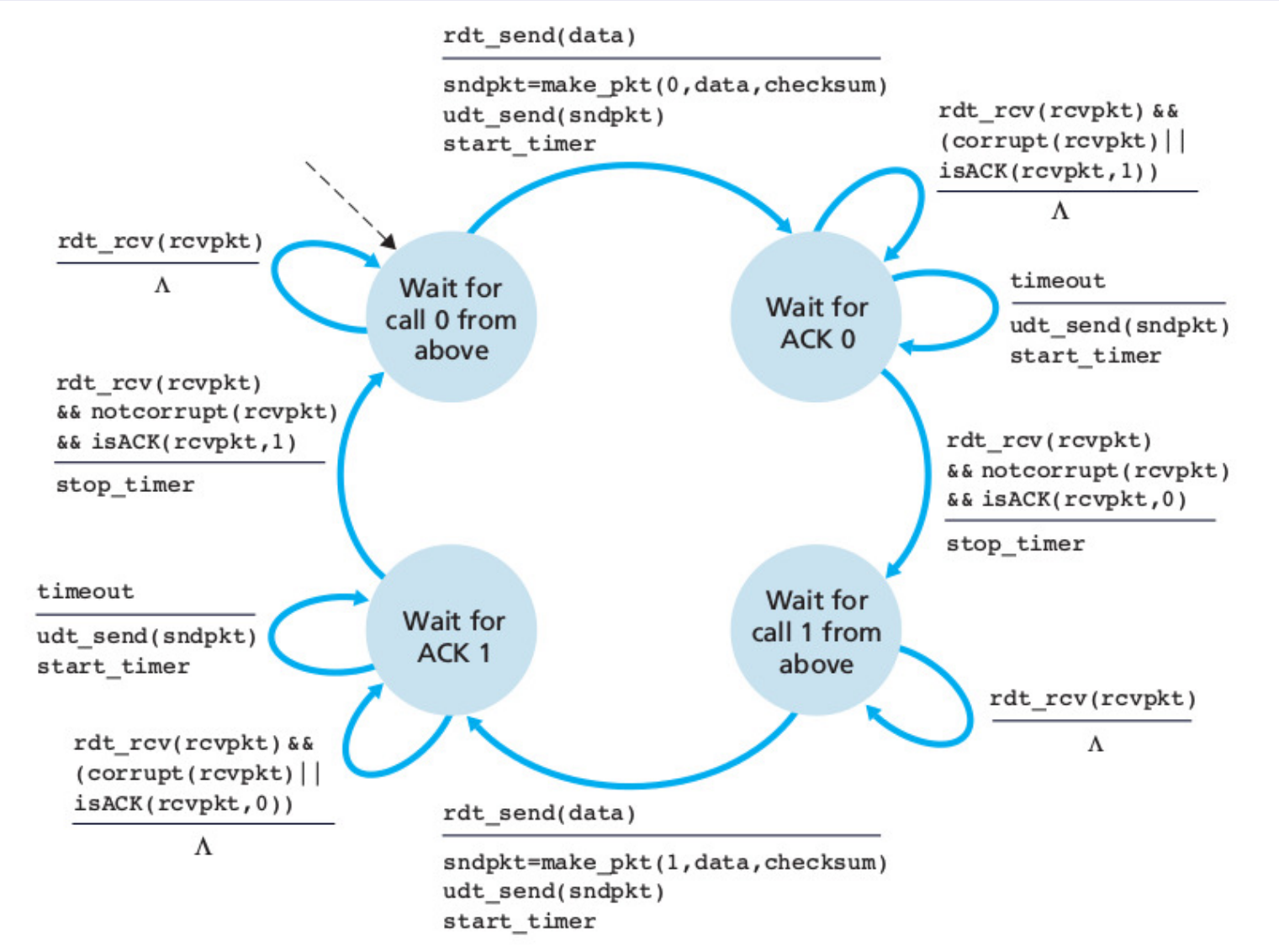
Heurística:

1. Inicializar um temporizador para cada pacote;
2. Responder à avisos de timeout;
3. Parar o cronômetro.

Protocolo do tipo de **alternância de bit**.

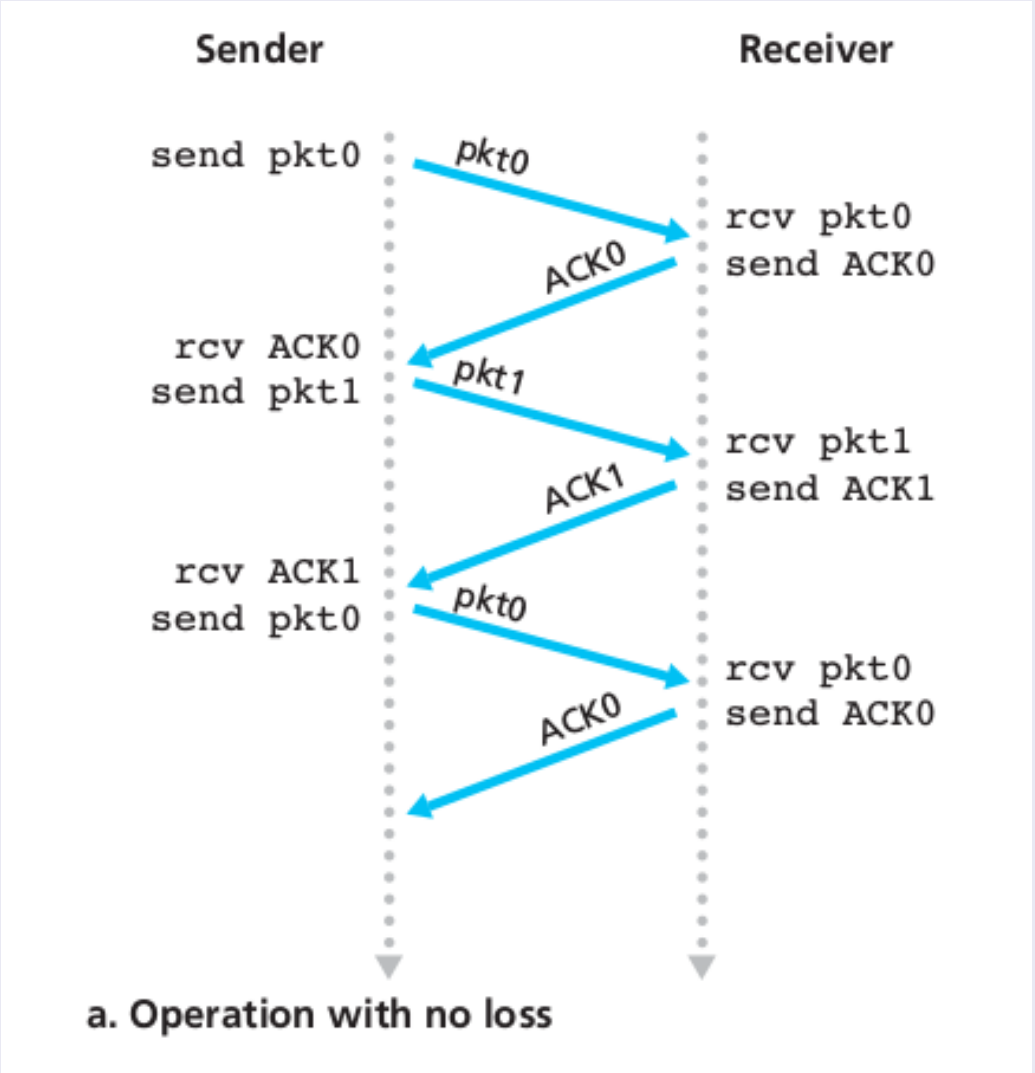
Construindo um protocolo de transmissão de dados confiável

rdt3.0: canal com perdas e erros. Visão da máquina pelo emissor.



Construindo um protocolo de transmissão de dados confiável

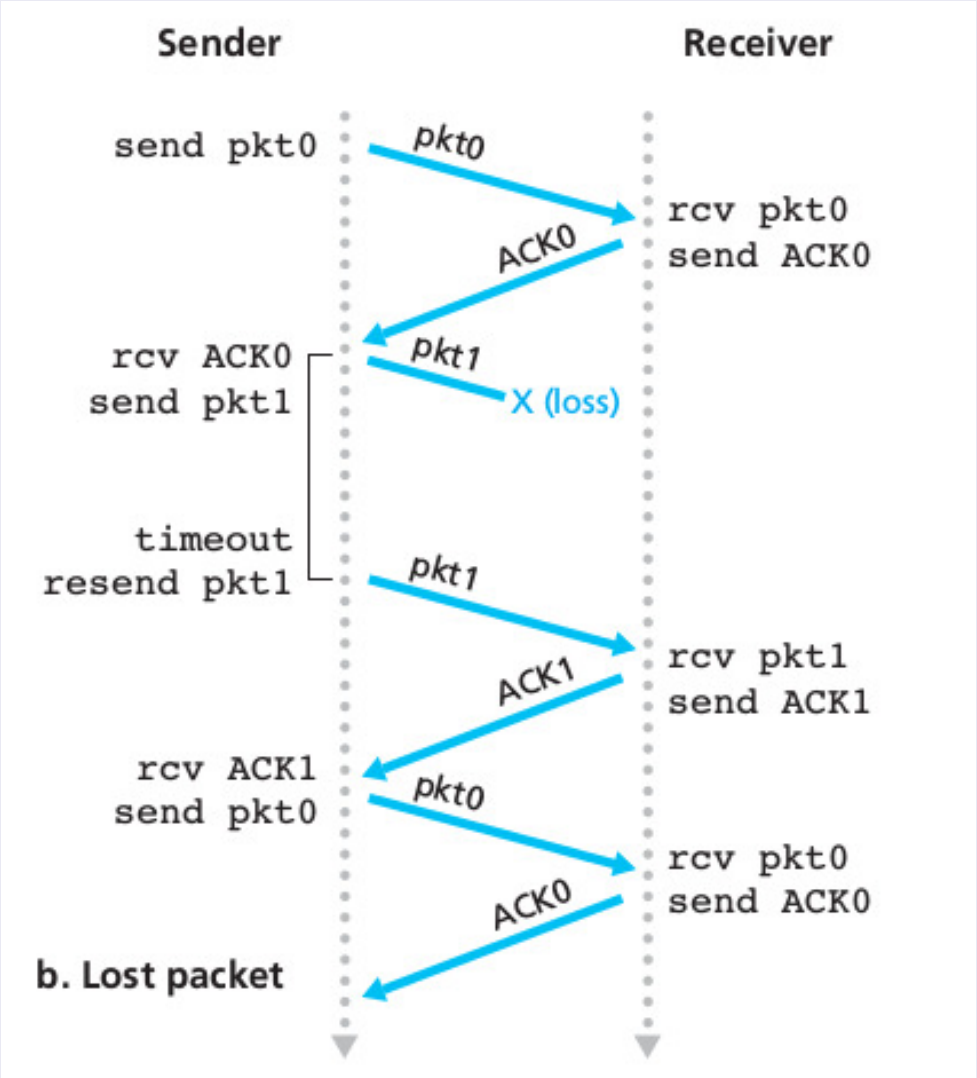
rdt3.0: canal com perdas e erros.



a. Operation with no loss

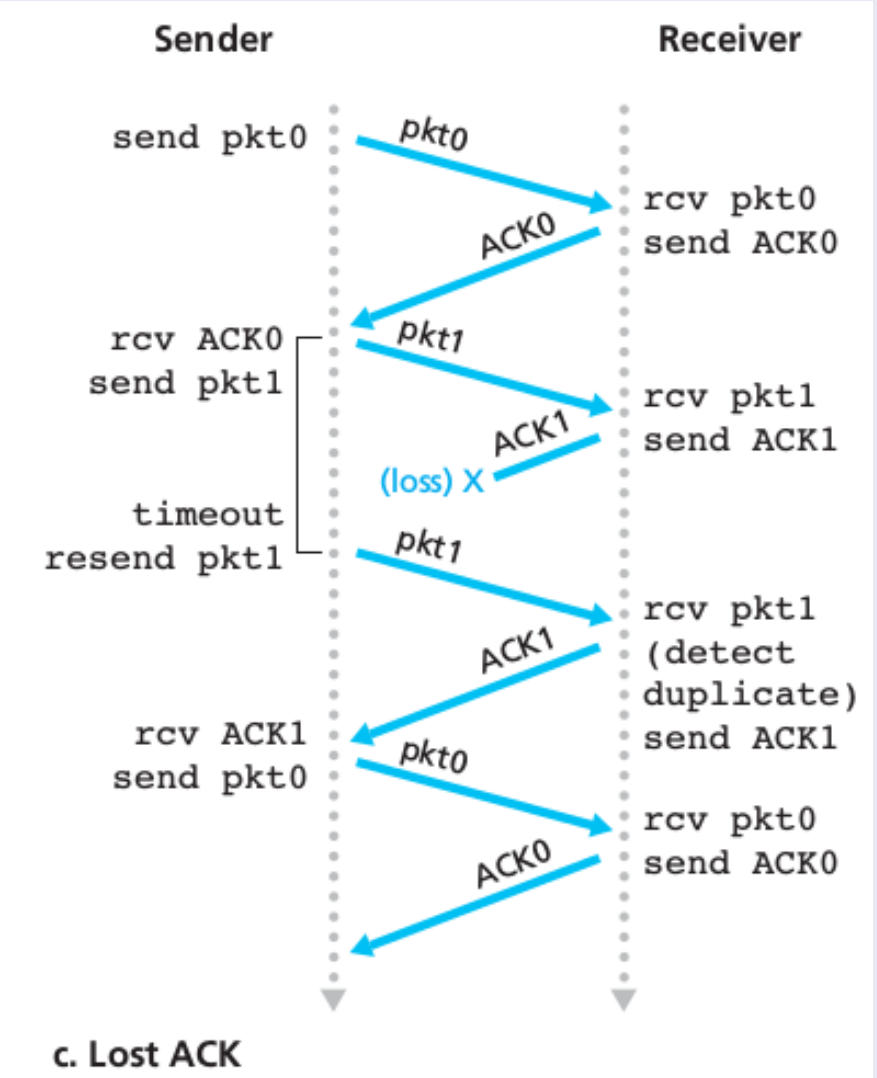
Construindo um protocolo de transmissão de dados confiável

rdt3.0: canal com perdas e erros



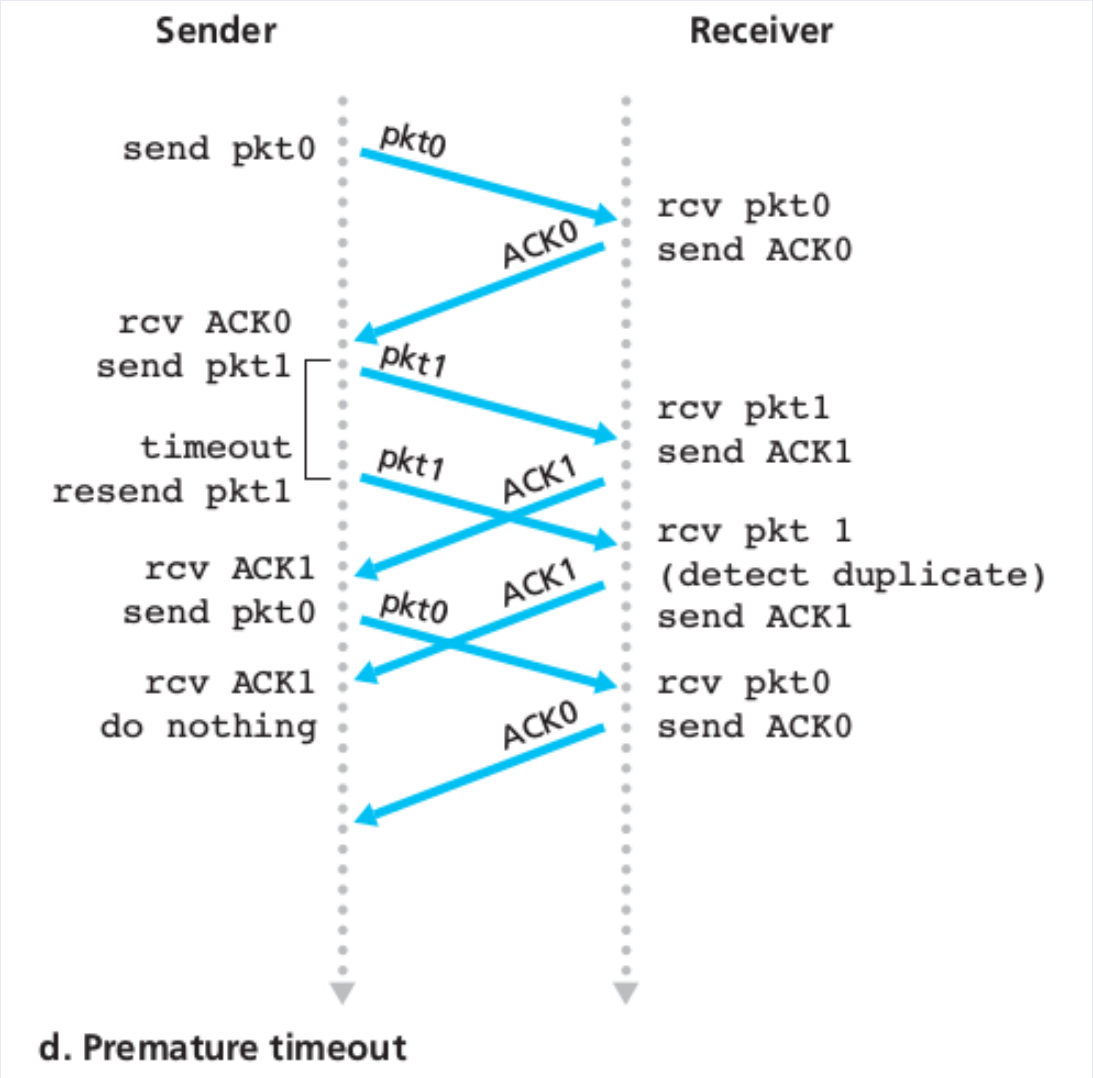
Construindo um protocolo de transmissão de dados confiável

rdt3.0: canal com perdas e erros



Construindo um protocolo de transmissão de dados confiável

rdt3.0: canal com perdas e erros



d. Premature timeout

Protocolo de transmissão confiável de dados com Pipeline

rdt3.0: **pára e espera**. Isso é ruim para desempenho!

métrica:

$$\text{utilização} = \frac{\text{tempo em que transmissor está } \textbf{ocupado} \text{ transmitindo bits pelo canal}}{\text{tempo de transmissão total do dado}}$$

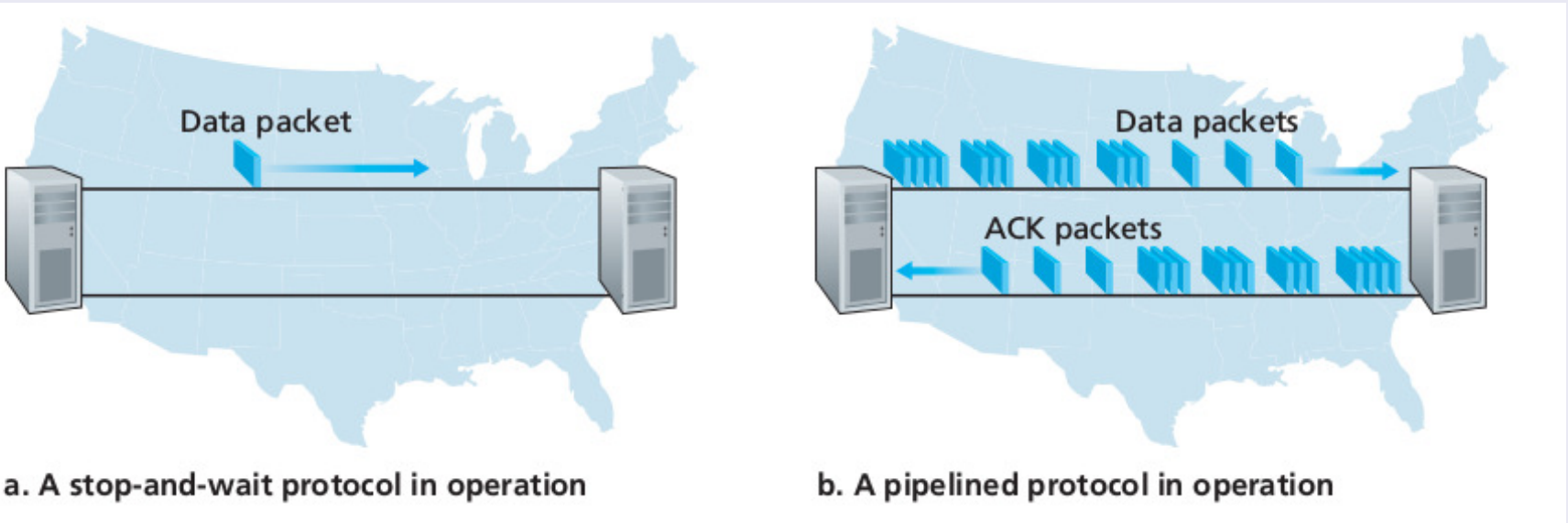
Solução? **Pipelining**: enviar múltiplos pacotes sem esperar por acknowledgments

- **aumentar** o tamanho dos números de sequência. Até então, havia 1 bit (por isso, **alternância de bit**) de número de sequência
- emissor e receptor devem possuir **bufferes** para armazenar mais de um pacote cuja recepção ainda não foi confirmada

Abordagens: **Go-Back-N** e **repetição seletiva**

Protocolo de transmissão confiável de dados com Pipeline

Pára e espera *vs.* Pipelined

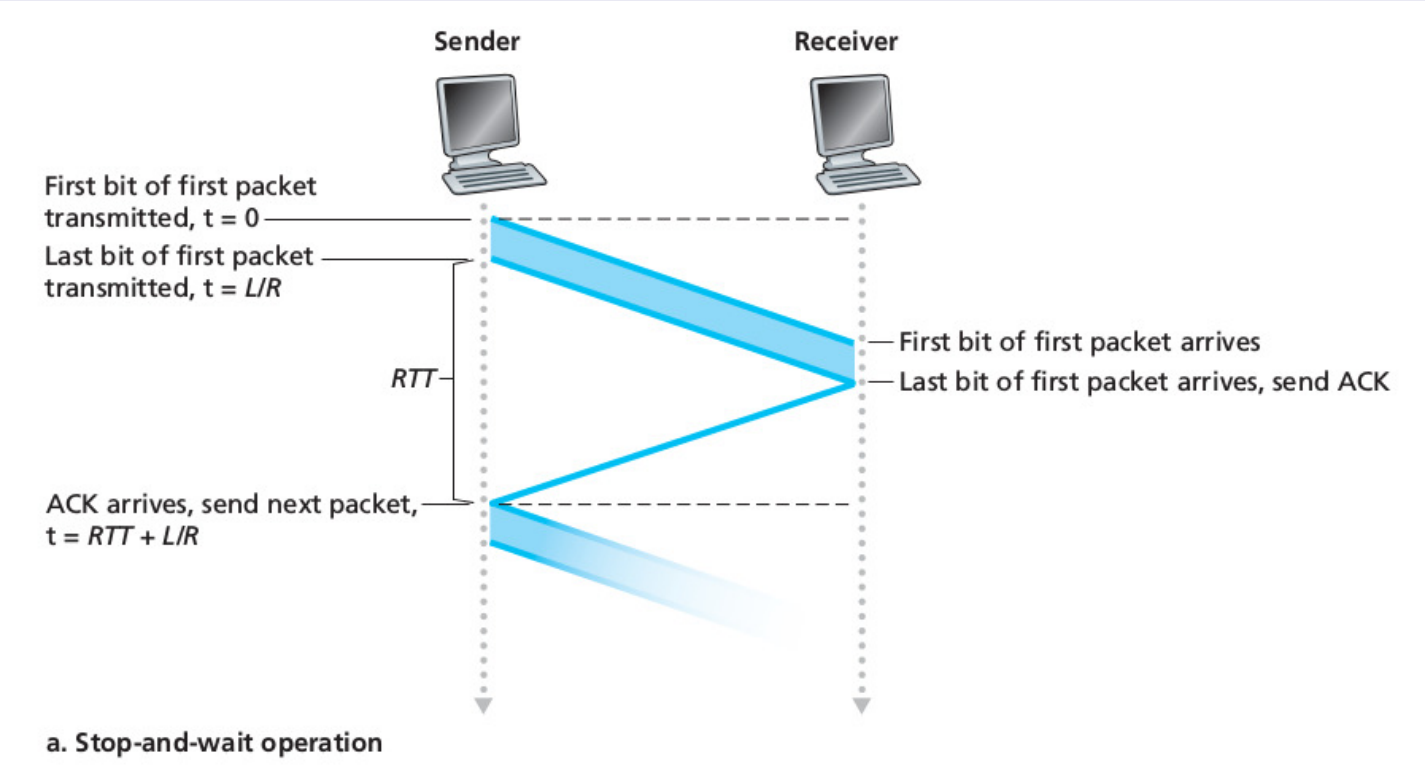


a. A stop-and-wait protocol in operation

b. A pipelined protocol in operation

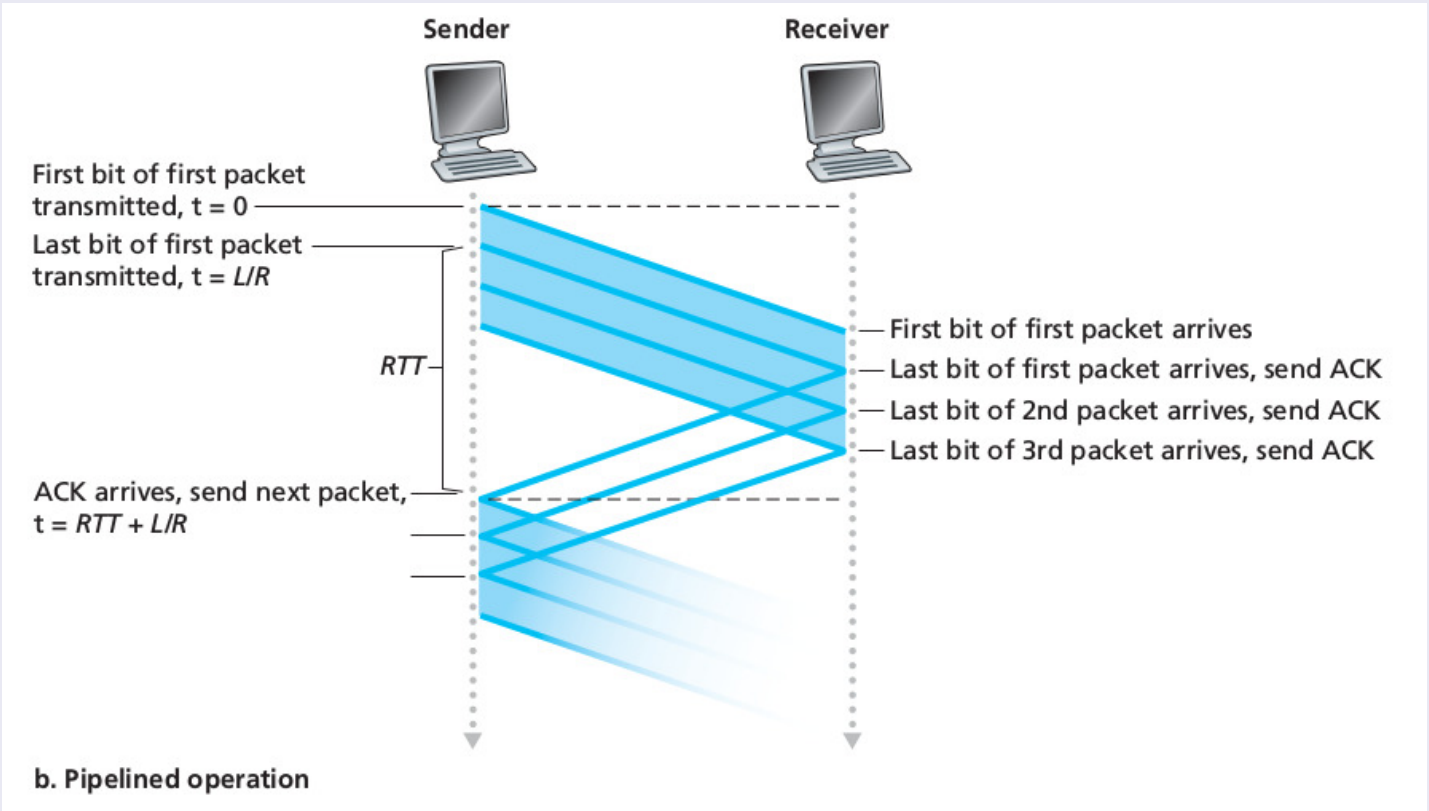
Protocolo de transmissão confiável de dados com Pipeline

Pára e espera: utilização



Protocolo de transmissão confiável de dados com Pipeline

Pipelined: utilização



b. Pipelined operation

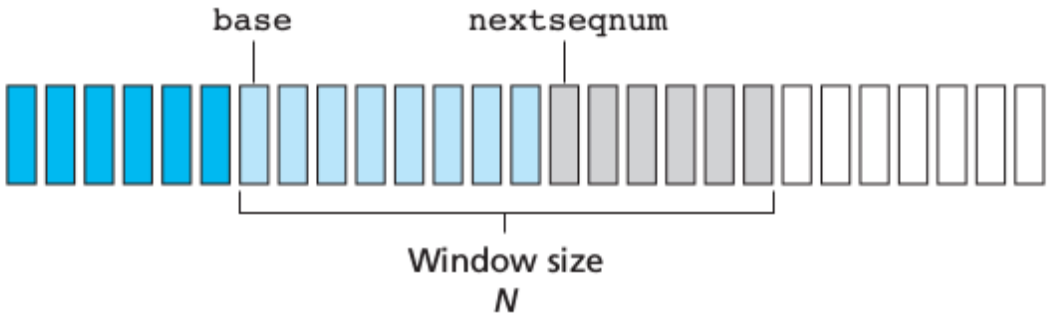
GBN: Go-Back-N

Emissor pode transmitir vários pacotes **sem esperar** por acknowlegdements. Não se admite ter mais que o limite de **N** pacotes não confirmados no pipeline.

- janela de tamanho **N**: protocolo de **janela deslizante**.
- o limite N ajuda no controle de tráfego e congestionamentos

GBN

Visão do emissor da utilização de números de sequência em GBN.



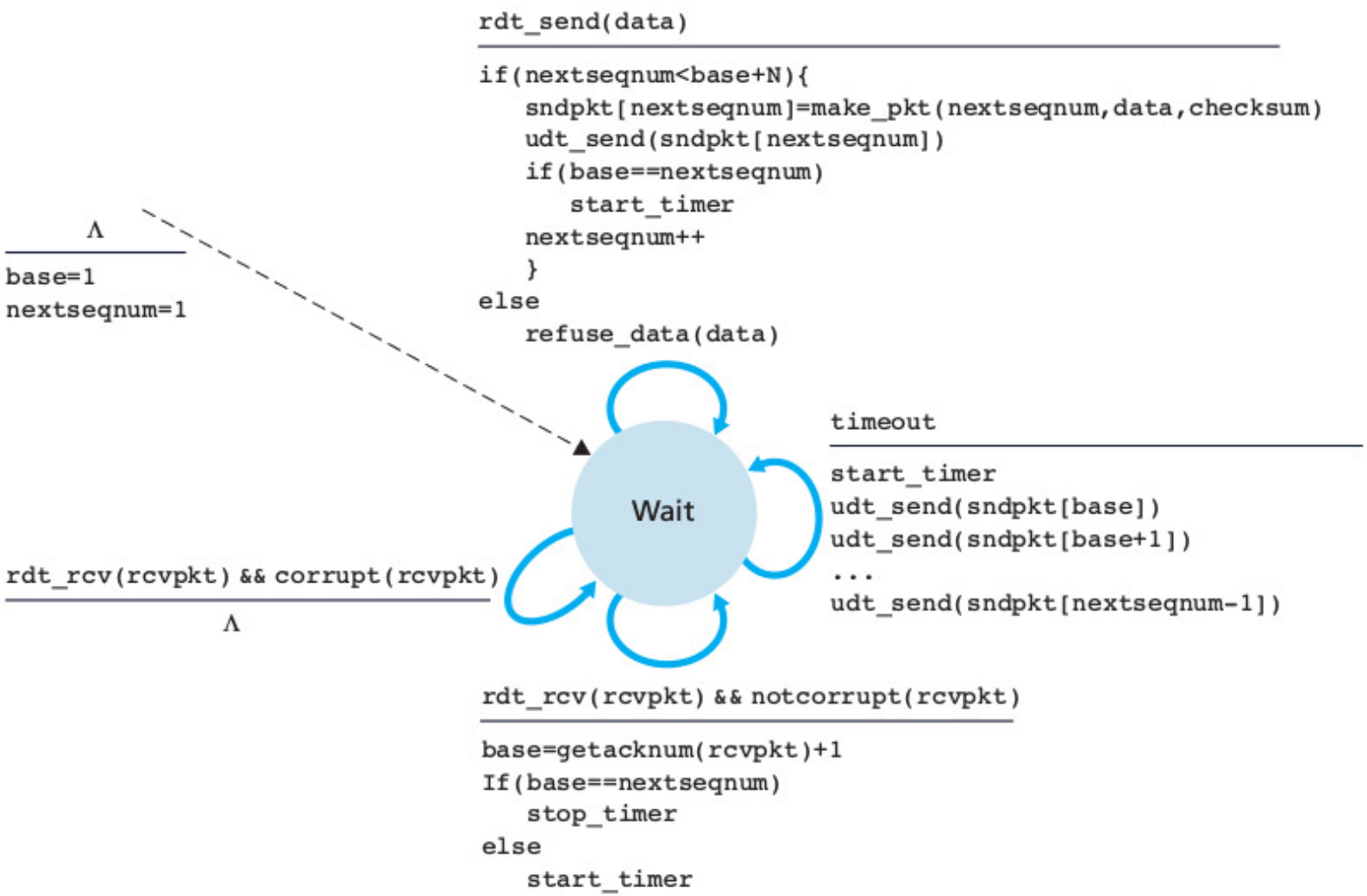
GBN

Emissor deve responder a **três tipos** de eventos

- **Chamada superior: rdt_send()**: Checa a janela. Se há espaço, pacote é criado e enviado; variáveis atualizadas de acordo. Se está cheia, retorna aviso de que janela está cheia para a camada de aplicação. Aplicação deverá tentar novamente depois.
- **Recebe ACK**: Confirmação acumulativa de que todos os pacotes até o número de sequência **n** indicado no ACK foram devidamente recebidos.
- **Evento timeout**: tratamento de perdas ou atrasos muito longos. Retransmissão de todos os pacotes não confirmados.

GBN

Emissor: máquina de estados. Visão da máquina pelo emissor.



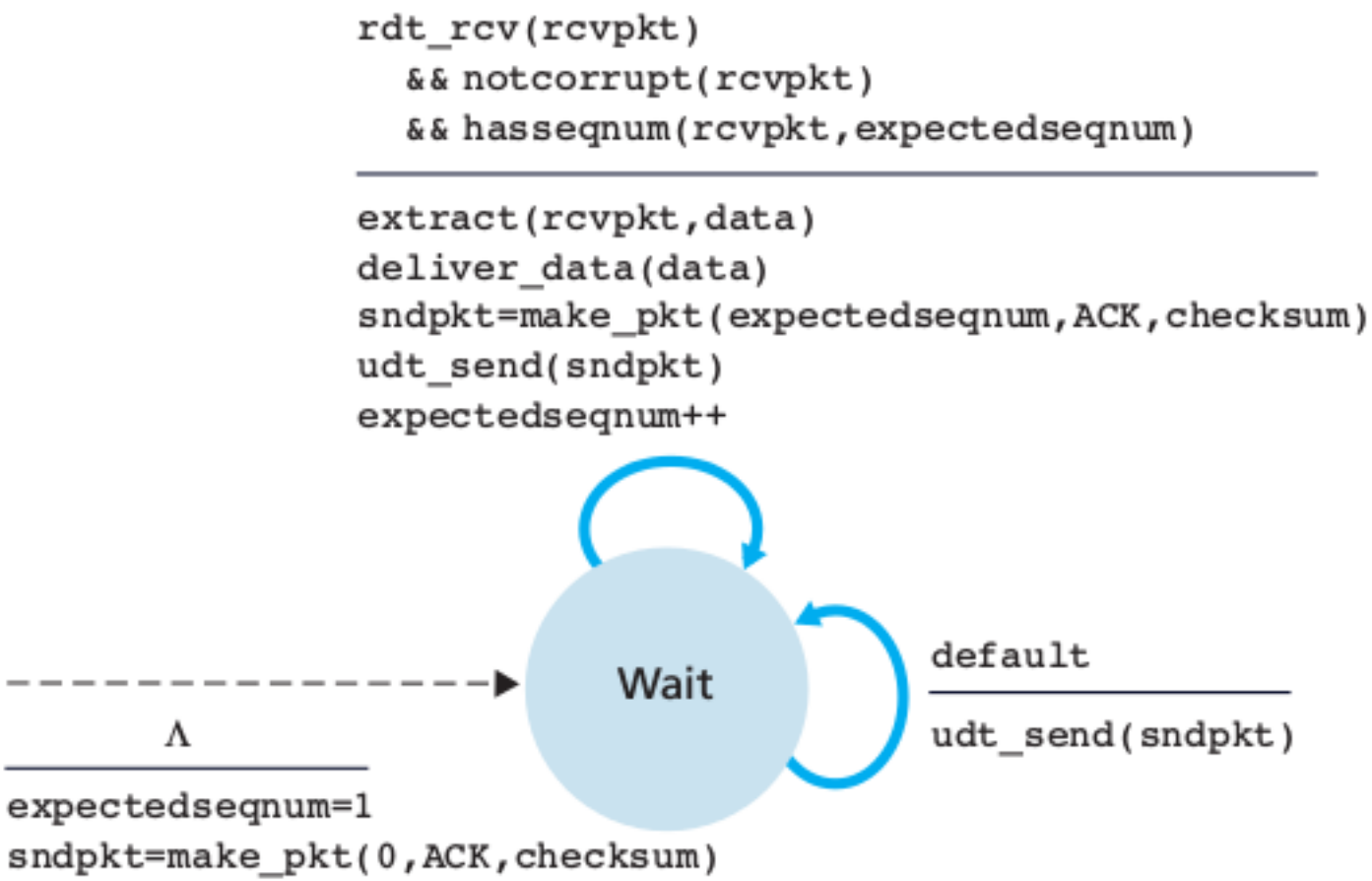
GBN

Receptor:

- Envia ACK do pacote mais recente recebido adequadamente.
  - Pacotes recebidos corretamente são encaminhados para a aplicação.
  - Pacotes com problemas são sumariamente descartados.
- Descartam-se pacotes fora de ordem: evita a bufferização de pacotes fora de ordem no receptor. Simplificação da implementação. Envio do ACK para o pacote mais recente recebido em ordem.

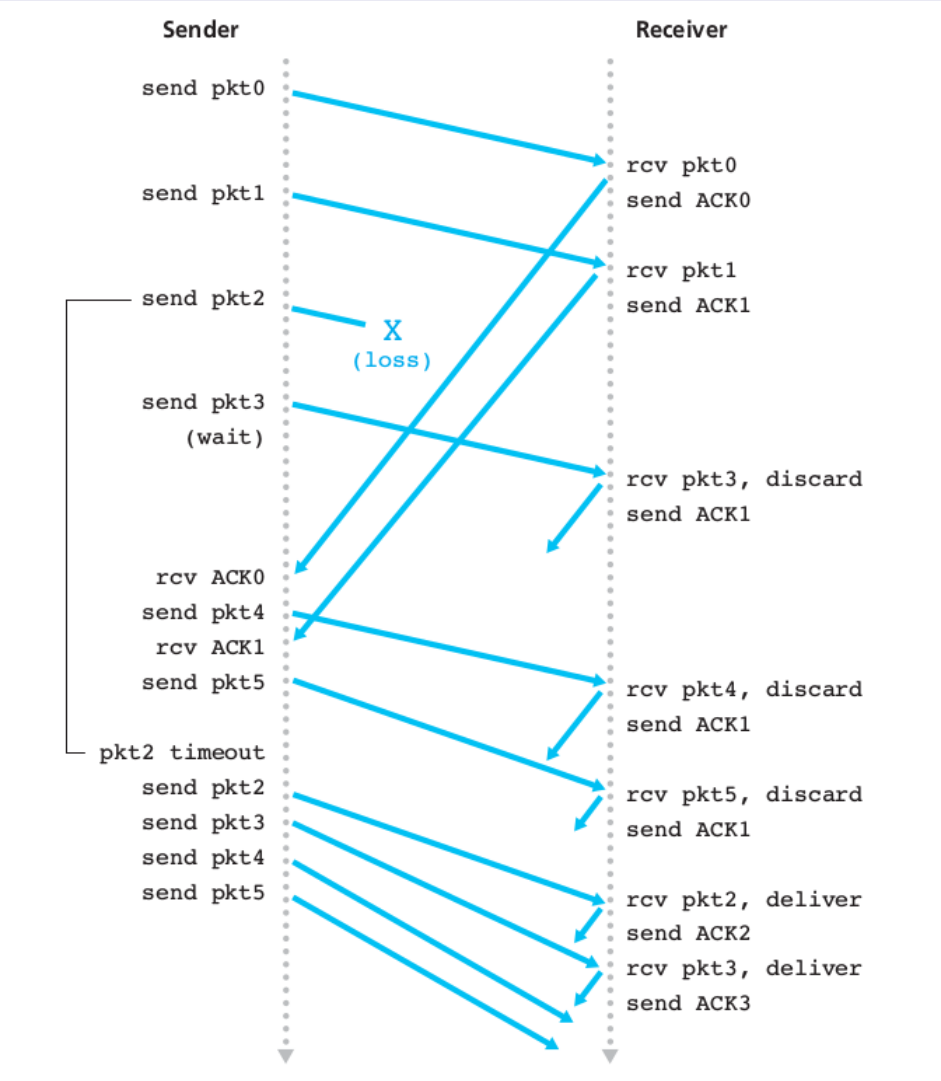
GBN

Receptor: máquina de estados. Visão da máquina pelo receptor.



GBN

Evolução com Go-Back-N. Janela de 4 pacotes.



Repetição Seletiva

A recepção de pacotes fora de ordem provoca envio duplicado de mensagens.

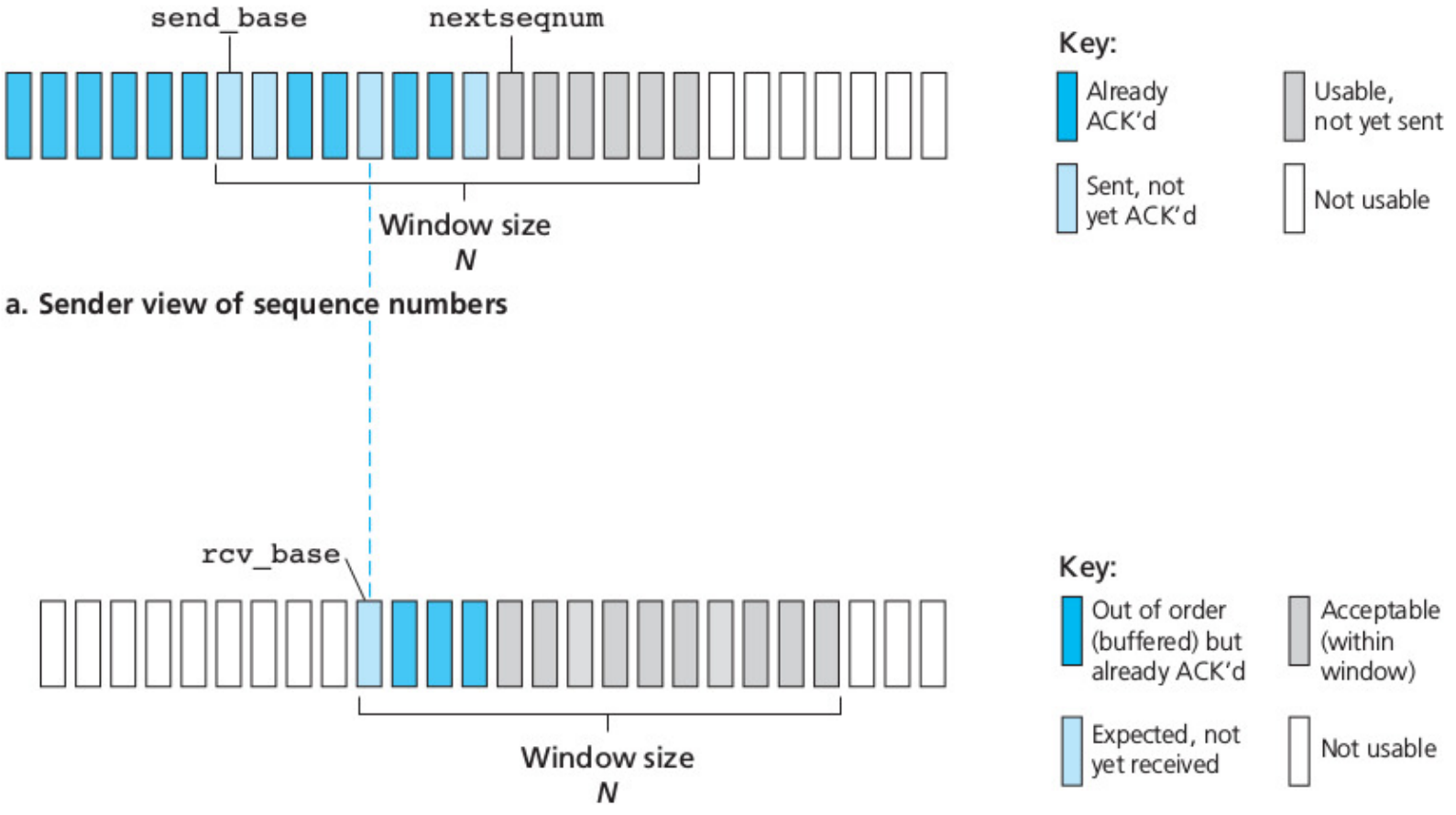
No caso na perda de um pacote, janela muito larga ou atraso muito grande proveniente da largura de banda **degradam** desempenho.

Repetição Seletiva

Ações e eventos do SR **emissor**:

1. **Recepção de dados da camada superior**: o emissor SR checa o próximo número de sequência disponível para o pacote. Se o número de sequência está dentro da janela do emissor, o dado é empacotado e enviado; caso contrário, é bufferado ou retornado para a camada de aplicação, para futura tentativa de transmissão.
2. **Timeouts proteção para perda de pacotes**. **Cada pacote** deve ter, agora, seu próprio **temporizador**.
3. **ACK recebido**: Se um ACK for recebido, o emissor SR marca o pacote como recebido, desde que pertença à janela. Se o número de sequência do pacote for igual à send_base, a janela é movida para o pacote com o menor número de sequência e ainda não confirmado. Se a janela mover e houver pacotes pendentes de transmissão, esse pacotes serão transmitidos.

Repetição Seletiva



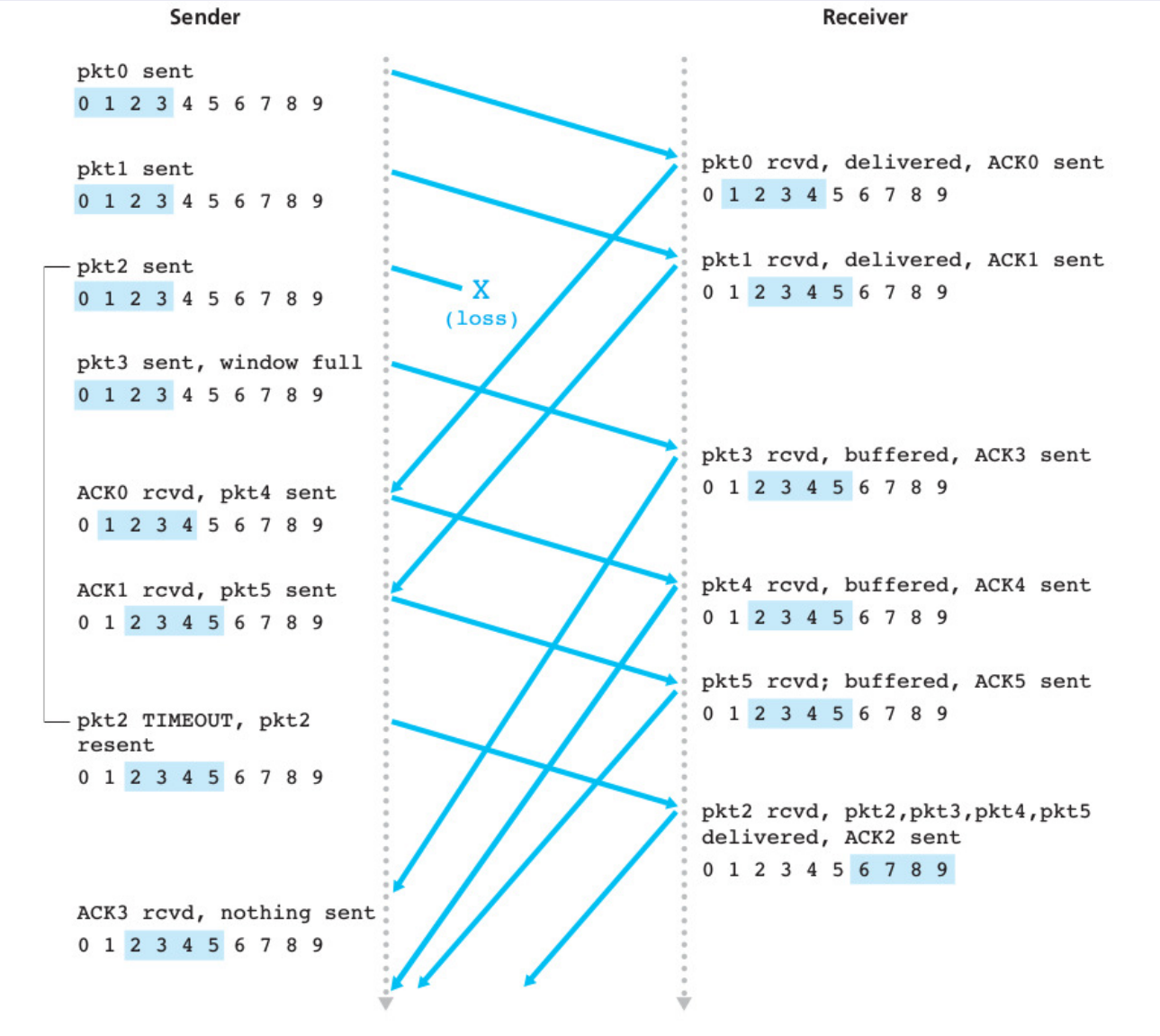
Repetição Seletiva

Ações e eventos do SR **receptor**

1. **Pacote com número de sequência entre [rcv_base, rcv_base+N-1] recebido corretamente**. Pacote pertence à janela de recepção do receptor; responde-se um ACK seletivo. Se o pacote não tiver sido recebido anteriormente, é bufferado. Se o pacote tem um número de sequência igual à da base de recepção, esse pacote e todos os pacotes bufferados anteriormente e numerados consecutivamente (começando em rcv_base) são entregues à camada superior: **flush**. A janela de recepção é movida adiante pelo número de pacotes entregue à camada de aplicação.
2. **Pacote com número de sequência entre [rcv_base-N, rcv_base-1]**. ACK deverá ser gerado, mesmo que ACK para esse pacote tenha sido emitido anteriormente.
3. Caso contrário, **ignorar o pacote**.

Repetição Seletiva

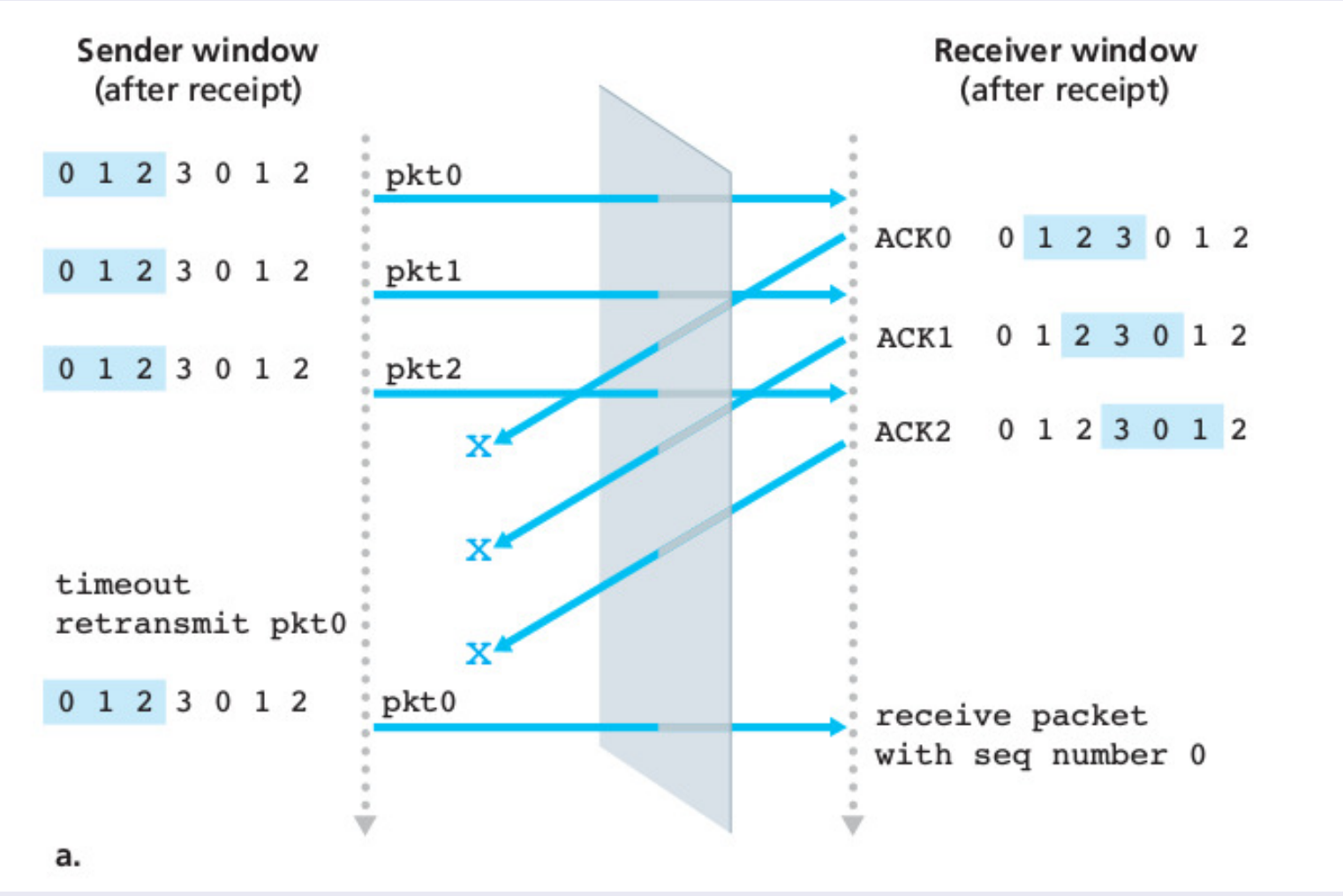
Evolução com repetição seletiva.



Repetição Seletiva

Dilema: tamanho de janelas muito grandes.

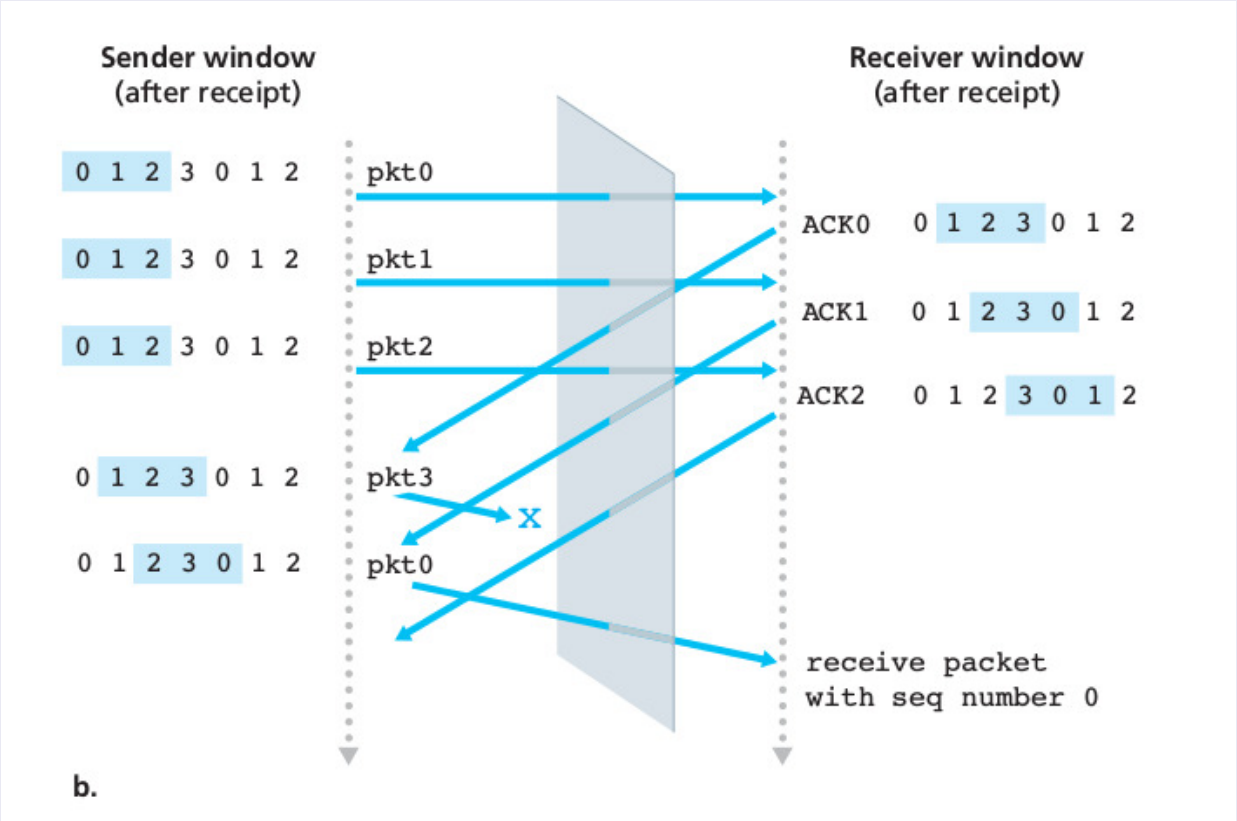
Novo pacote ou retransmissão? Cenário (a).



Repetição Seletiva

Dilema: tamanho de janelas muito grandes.

Novo pacote ou retransmissão? Cenário (b).



Qual dos dois pkt 0 foram perdidos? De fato, dessa maneira o receptor não tem como discernir o cenário (a) do cenário (b)!

Repetição Seletiva

O emissor e o receptor nem sempre terão uma visão idêntica do que foi recebido corretamente e do que não foi.

Problema com os mecanismos de tratamento de perda de pacote e aritmética modular usada nos números de sequência

Dilema tamanho da janela deslizante e anel de número de sequência $\Longleftrightarrow 1/2$

Tempo máximo de vida de um pacote TCP: **RFC 1323 (3 minutos)**.

Introdução

Definido nas RFCs 793, 1122, 1323, 2018 e 2581.

Conexão TCP

Antes dos processos iniciarem a troca de mensagens, há uma estapa de **handshaking**.

- ambos os lados da conexão inicializam várias variáveis de estado do TCP vinculadas a conexões TCP.
- Roteadores intermediários não tem conhecimento do estado da conexão TCP
- É do tipo **full-duplex**, ponto a ponto (apenas um emissor e um receptor)

Multicasting não é possível com TCP!

Conexão TCP

Estabelecimento de conexão:

clientSocket.connect((serverName,serverPort))

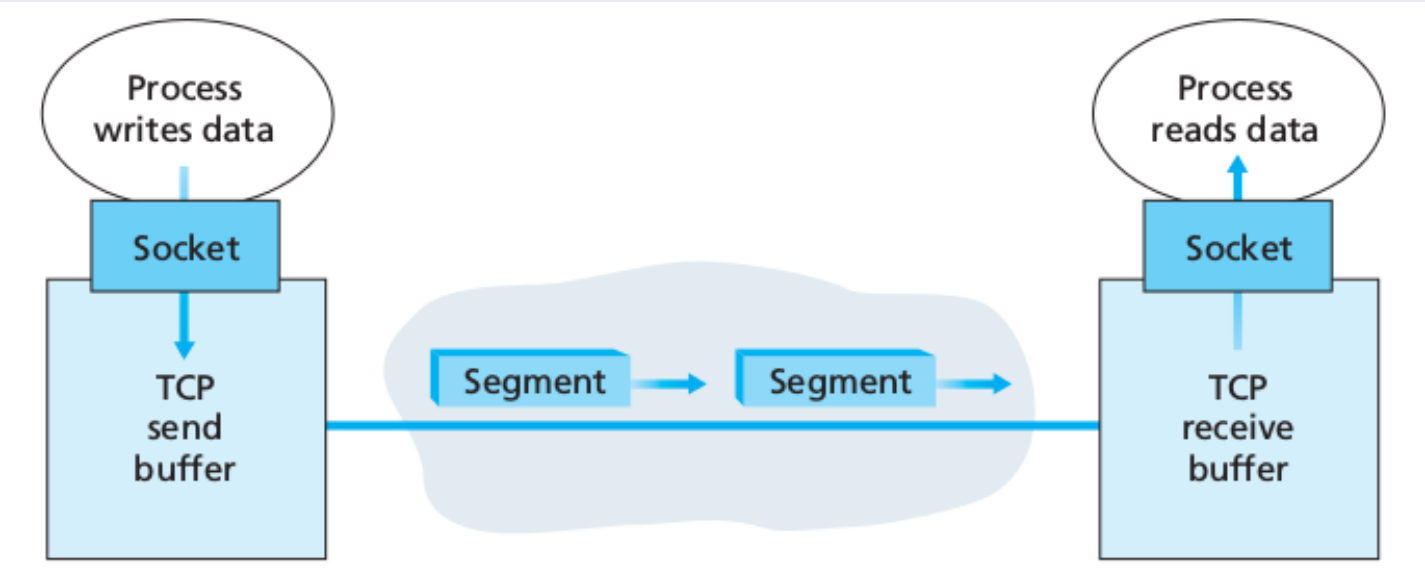
- 3-way handshake
- uma vez estabelecida a conexão, dois processos de aplicação podem trocar dados entre si
- dados são inseridos no buffer de envio. De tempo em tempo, TCP coleta porções de dados e os encaminha para a camada de aplicação. (RFC 793)

MSS: Maximum Segment Size (apenas payload). Tipicamente 1500 octetos.

RFC 1191: MSS dependendo do valor máximo da MTU (Maximum Transport Unit) encontrada no caminho.

Conexão TCP

Processo de encapsulamento: payload+headers = segmento, datagramas IP, quadros Ethernet



Quando o segmento TCP é recebido no destino, é inserido no buffer de chegada. Cada aplicação possui um buffer de envio e um buffer de chegada

Não se mantêm estados entre os hosts, apenas na origem e no destino (E os firewalls e filtros de conteúdo?).

Estrutura de um segmento TCP

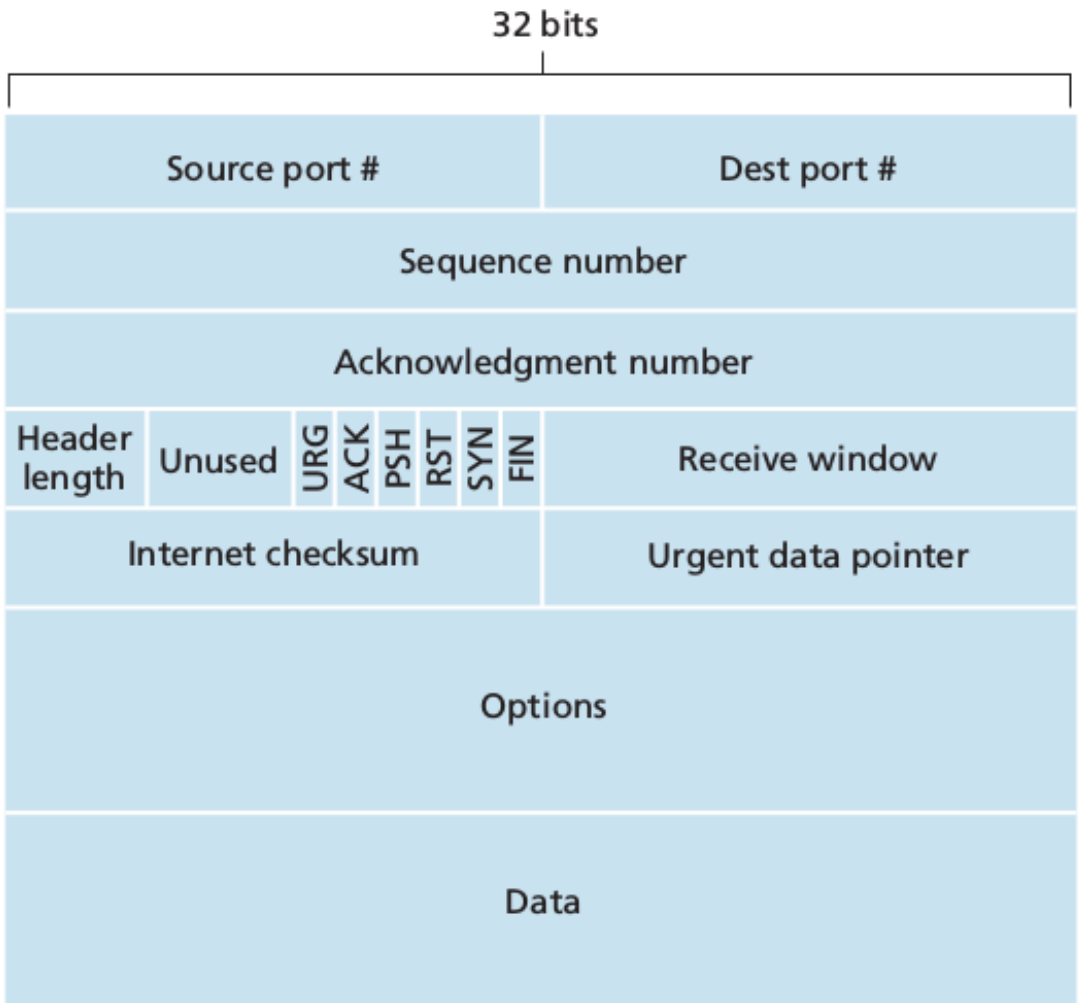
Dividido em campos de cabeçalho e de dados. Campo de cabeçalho embala os dados de aplicação.

MSS: Maximum Segment Size $\Longrightarrow$ apenas comprimento do payload!

Cabeçalho TCP: 20 octetos (ou bytes). São 12 bytes a mais que o cabeçalho UDP!

Segmento Telnet: 21 bytes de comprimento

Estrutura de um segmento TCP



Estrutura de um segmento TCP

Campos:

- **portas** de origem e destino; **checksum**.
- campo de **número de sequência** (32 bits), 32 bits de **número de acknowledgement** (transferência confiável de dados)
- campo de **comprimento de janela de recepção** (16 bits): controle de fluxo. Número de bytes que o receptor deseja receber
- campo de **comprimento de cabeçalho** (4 bits): o comprimento do cabeçalho. Codificado em **número de palavras de 32 bits**
- campo de **opções**: negociação do MSS; fator escalamento de janela usado em redes de alta velocidade; time-stamping. (RFCs 854 e 1323)

Estrutura de um segmento TCP

Campos:

- campo de **flags** (6 bits).
  - ACK: bit usado para indicar que o valor carregado no campo acknowledgment para o segmento foi devidamente recebido.
  - RST: setup e teardown
  - SYN: setup e teardown
  - FIN: setup e teardown
  - PSH: receptor deve repassar dados à camada de aplicação imediatamente
  - URG: aviso da camada superior de que o envio do dado deve ser imediato. Implica posterior conferência e interpretação do campo de **ponteiro de dados urgentes**: 16 bits que apontam para os último bytes (do bloco) de dados urgentes.

Estrutura de um segmento TCP

Números de sequência e números de acknowledgment: dois campos mais importantes, pois são críticos para o serviço de transmissão confiável de dados

- **Números de sequência** são derivados do fluxo de bytes transmitidos. É o número que **indica a posição que o primeiro byte de um segmento ocupa**
- O **número acknowledgment** que um Host A insere em seu segmento é o **número de sequência do próximo byte** que o Host A **espera que seja enviado** pelo Host B (num cenário em que B envia dados para A).

Estrutura de um segmento TCP

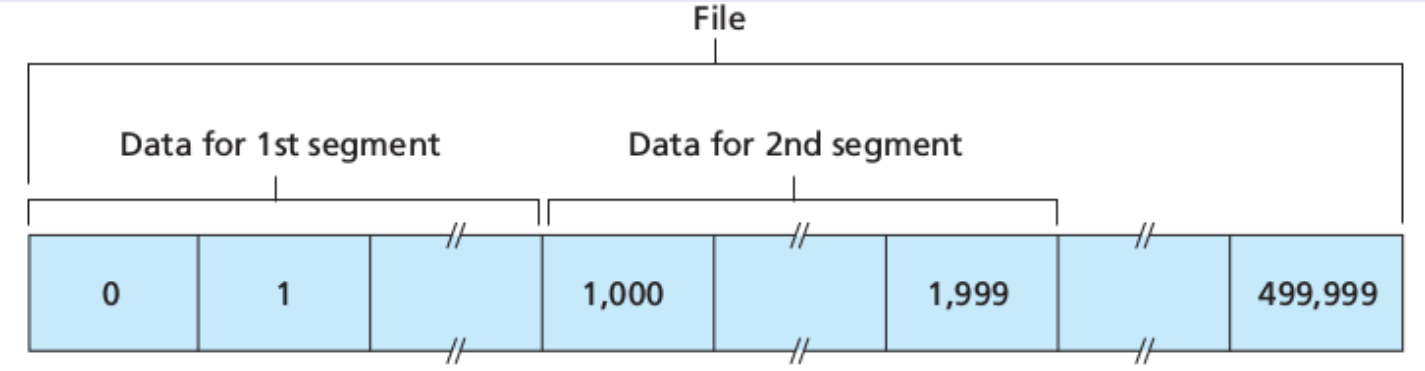
Acknowledgment acumulativo

No caso em que há perda de segmentos, as RFCs do TCP não indicam estratégia mandatória

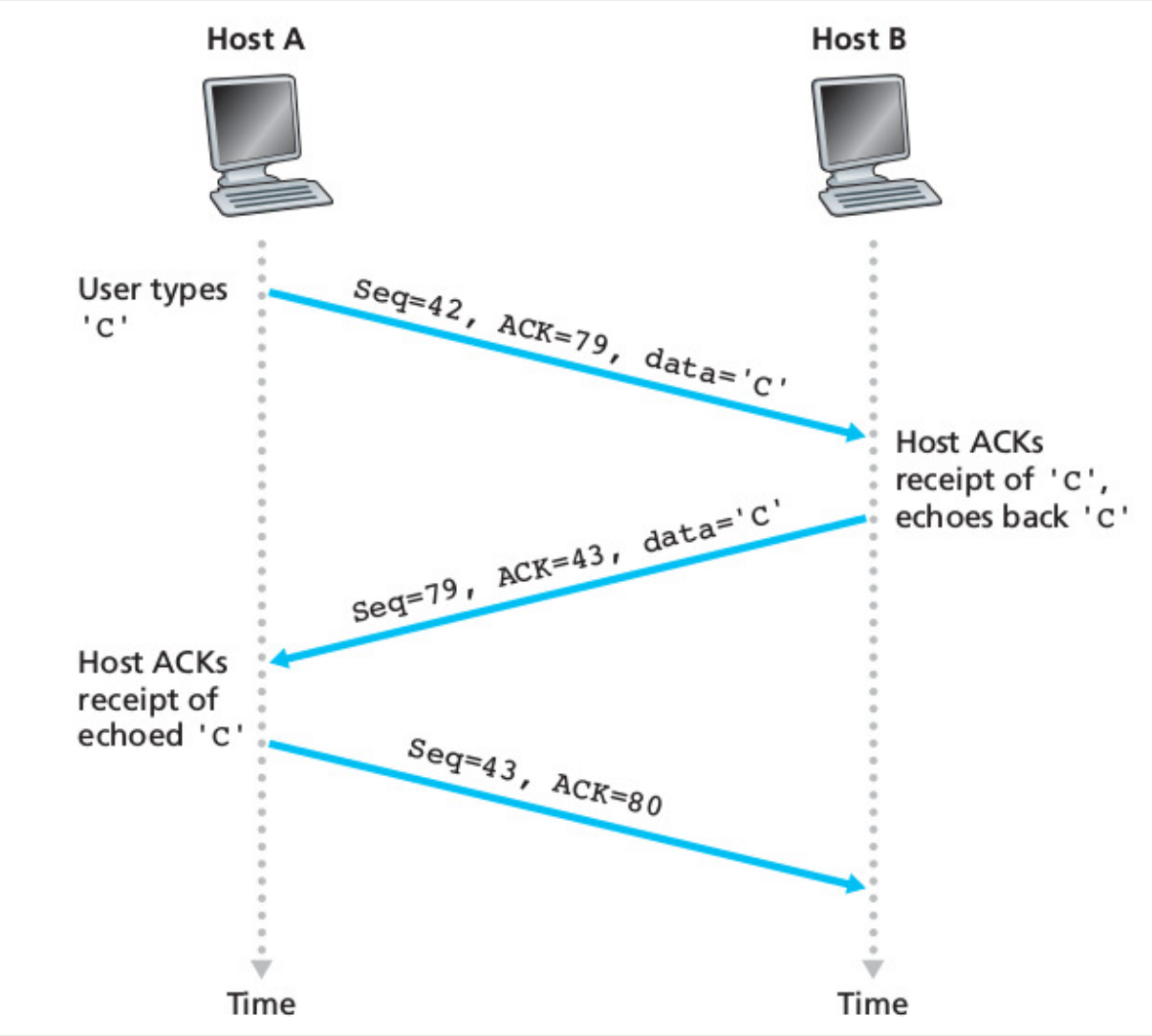
- É possível descartar segmentos fora da ordem (receptor simples)
- Ou o receptor mantém armazenados os bytes desordenados e espera os bytes faltantes para preencher as lacunas (receptor mais complexo, porém mais eficiente em termos de largura de banda)

O número de sequência inicial é aleatório: evitar "colisões" de números de sequência.

Estrutura de um segmento TCP



Telnet


Host A
Host B
User types 'C'
Seq=42, ACK=79, data='C'
Host ACKs receipt of 'C', echoes back 'C'
Seq=79, ACK=43, data='C'
Host ACKs receipt of echoed 'C'
Seq=43, ACK=80
Time
Time

Estimação de RTT e Timeout

$SampleRTT$: quantidade de tempo entre o momento em que o segmento é enviado (encaminhado para IP) e quando um acknowledgment para o segmento é recebido.

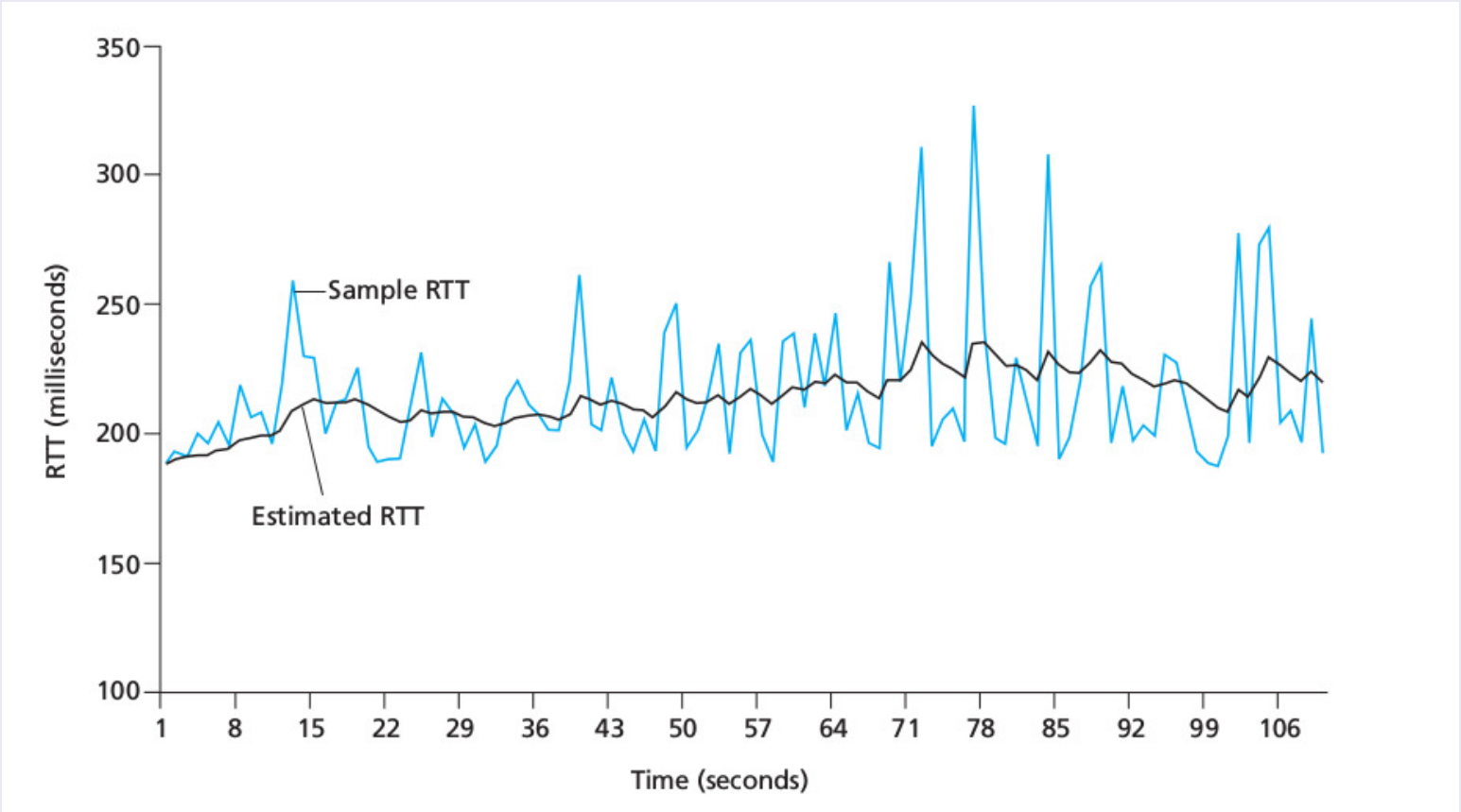
O TCP mede $SampleRTT$ apenas para segmentos que foram transmitidos sem demandarem retransmissões.

$EstimatedRTT$ é uma combinação ponderada dos valores anteriores de $EstimatedRTT$ e de $SampleRTT$

$$EstimatedRTT = (1 - \alpha) \times EstimatedRTT + \alpha \times SampleRTT$$

RFC 6298: $\alpha = 0,125$

Estimação de RTT e Timeout



Estimação de RTT e Timeout

Exponential Weighted Moving Average (EWMA)

$$DevRTT = (1 - \beta) \times DevRTT + \beta \times abs(SampleRTT - EstimatedRTT)$$

RFC 6298: $\beta = 0,25$

$$TimeoutInterval = EstimatedRTT + 4DevRTT$$

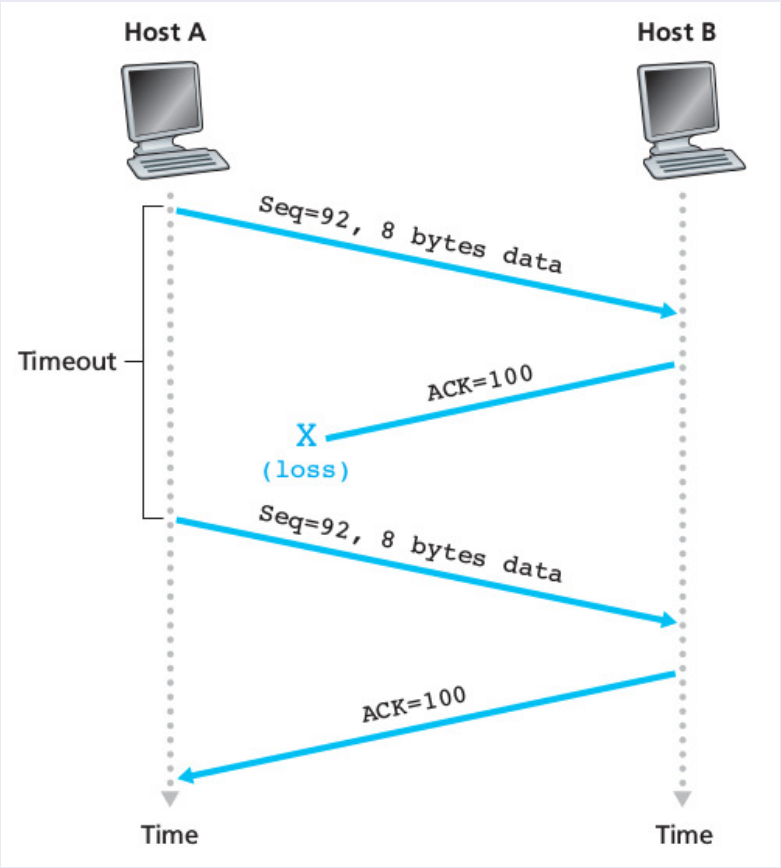
RFC 6298: inicialmente, 1 s

Transmissão Confiável de Dados

RFC 6298: temporizador de retransmissão único.

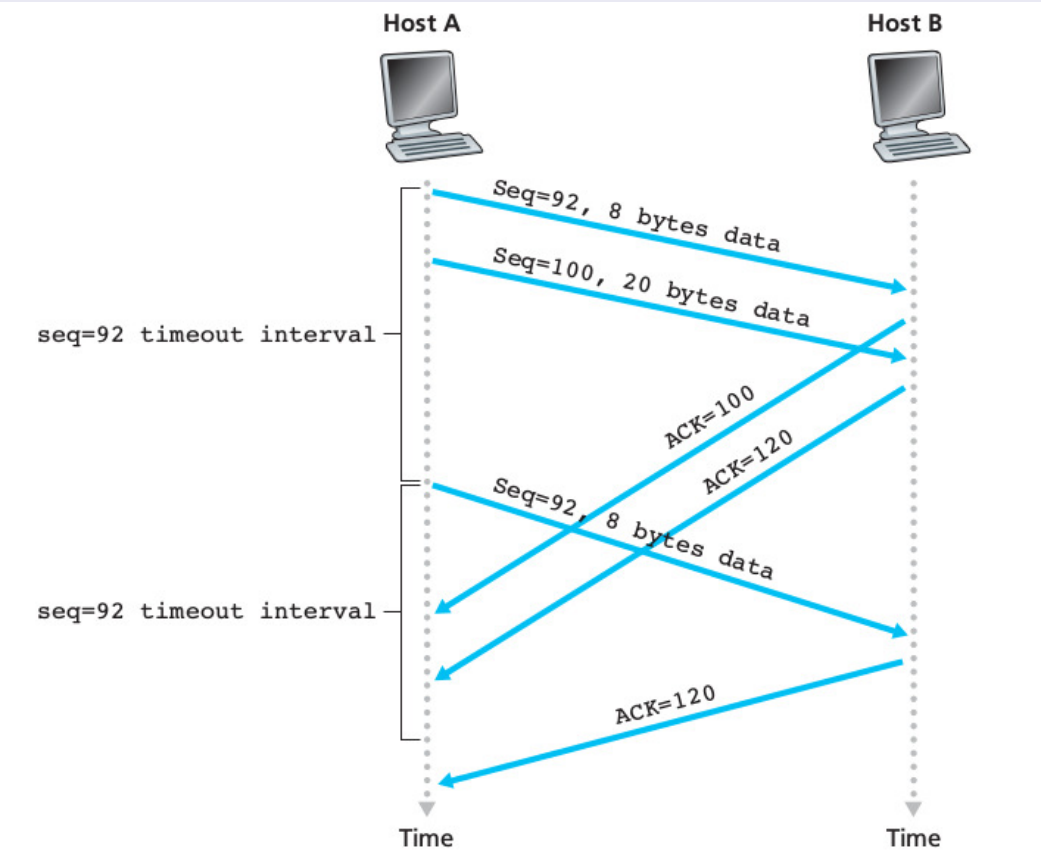
Associado com o segmento mais antigo não confirmado.

Cenário: Perda de ACK.



Transmissão Confiável de Dados

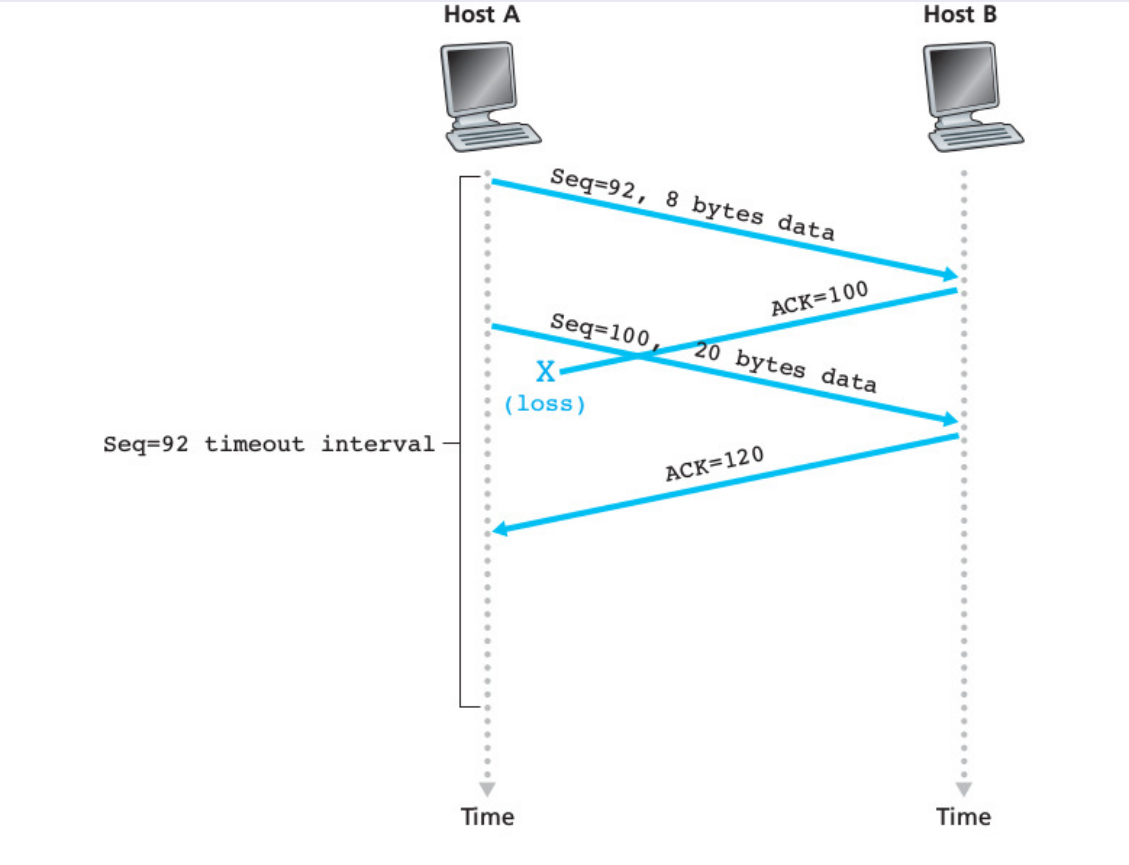
Cenário: inibição de uma retransmissão desnecessária. Segmento 100 não é retransmitido.



Observe que o ACK do segundo segmento chegou antes do novo intervalo de timeout (segunda transmissão de Seq=92) expirar.

Transmissão Confiável de Dados

Cenário: acknowledgement acumulativo. Evita retransmissão do primeiro segmento.



Observe que, embora tenha se perdido o ACK do primeiro segmento, o ACK acumulativo do segundo segmento confirma todos os segmentos até então recebidos adequadamente.

Transmissão Confiável de Dados

Duplicação de intervalo de Timeout: regra em caso de não confirmação. Constitui-se de uma forma limitada de controle de congestionamento!

Fast Restransmit

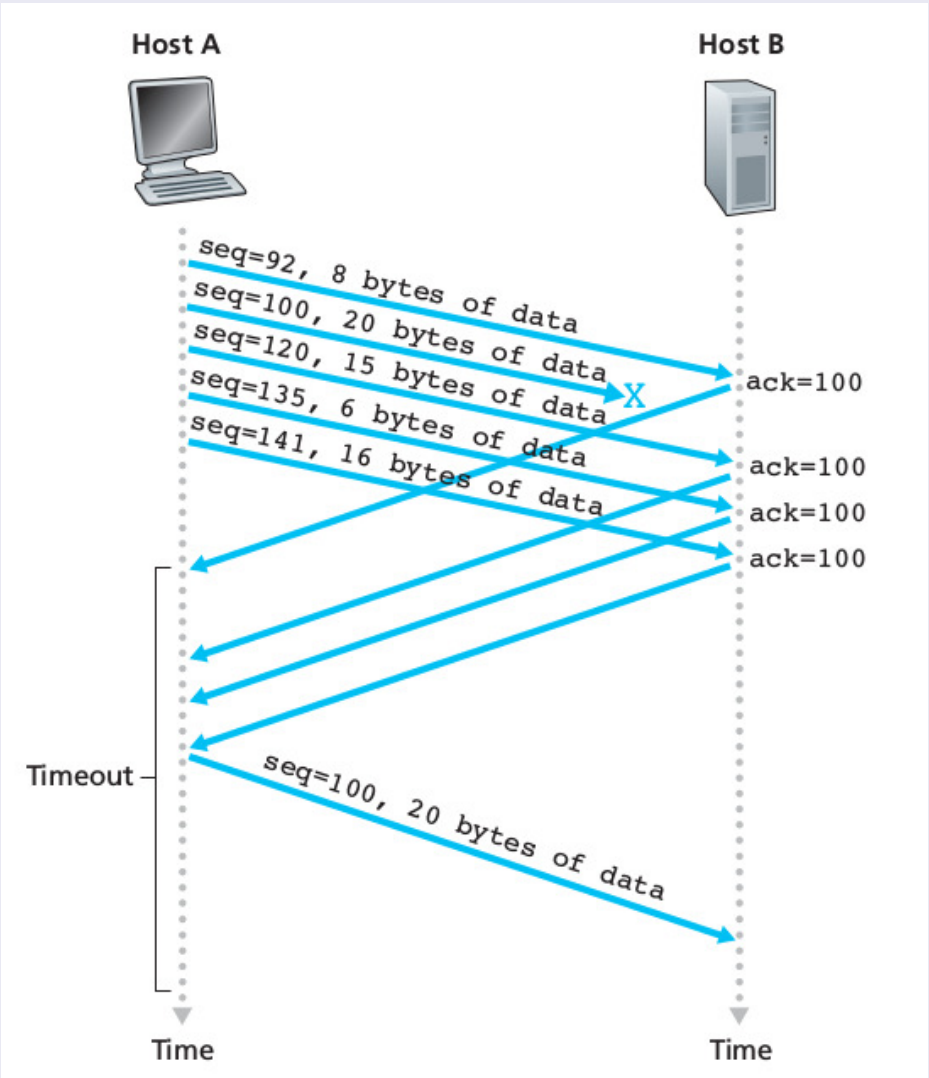
Transmissão Confiável de Dados

Situações em que se recomenda geração de ACK: ACK acumulativo.

Event	TCP Receiver Action
Arrival of in-order segment with expected sequence number. All data up to expected sequence number already acknowledged.	Delayed ACK. Wait up to 500 msec for arrival of another in-order segment. If next in-order segment does not arrive in this interval, send an ACK.
Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK transmission.	Immediately send single cumulative ACK, ACKing both in-order segments.
Arrival of out-of-order segment with higher-than-expected sequence number. Gap detected.	Immediately send duplicate ACK, indicating sequence number of next expected byte (which is the lower end of the gap).
Arrival of segment that partially or completely fills in gap in received data.	Immediately send ACK, provided that segment starts at the lower end of gap.

Transmissão Confiável de Dados

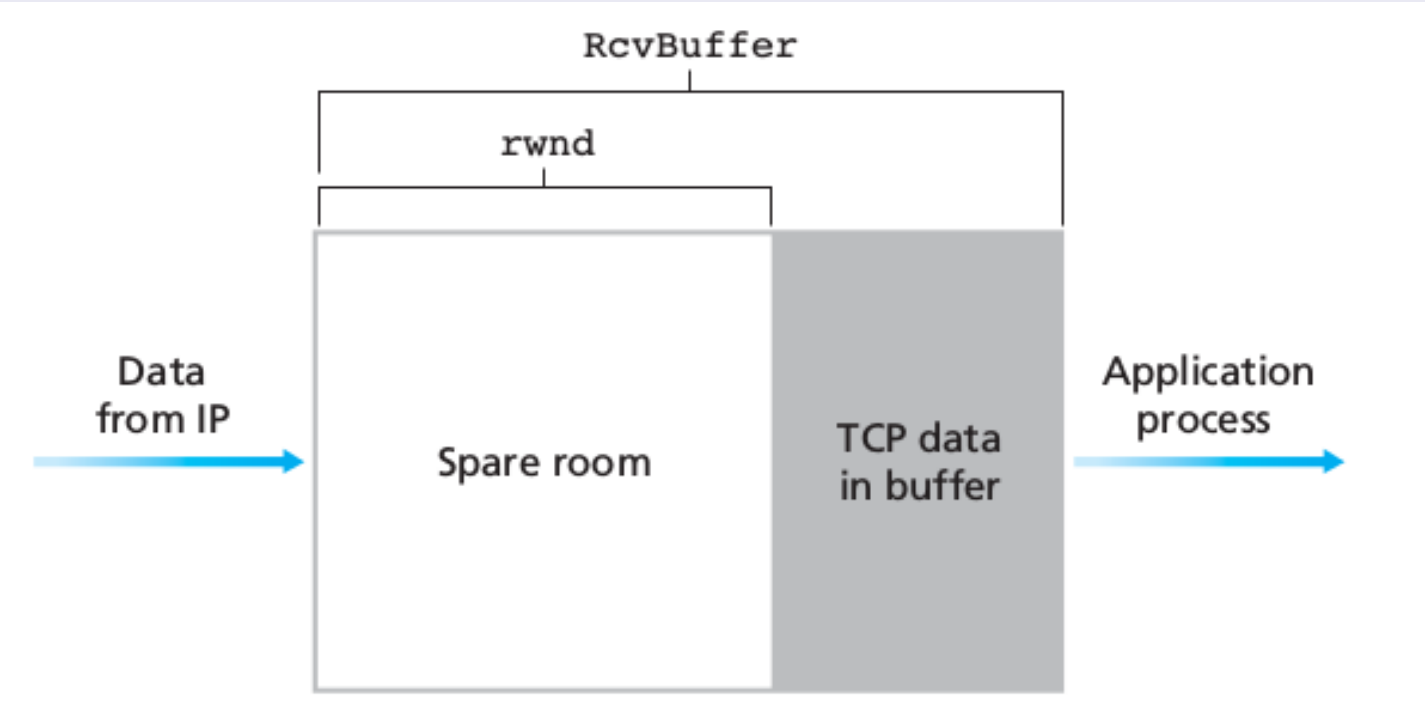
Fast retransmit: retransmissão antes do timeout do segmento perdido.



Controle de Fluxo

TCP provê um serviço de **controle de fluxo** que tenta compatibilizar as velocidades de transmissão e de consumo de dados

Não confundir com **controle de congestionamento**, provido pela camada de rede



Controle de Fluxo

$$LastByteRcvd - LastByteRead \leq RcvBuffer$$

$$rwnd = RcvBuffer - [LastByteRcvd - LastByteRead]$$

$$LastByteSent - LastByteAcked \leq rwnd$$

Controle de Fluxo

Quando a janela de recepção TCP for declarada nula, o emissor pode continuar a enviar segmento de um byte de dados.

Ao receber acknowledgments, haverá o informe do verdadeiro tamanho de $rwnd$

Isso é necessário para **prevenir** que o emissor **bloqueie** por não ter conhecimento do verdadeiro valor da janela depois que houver um consumo de dados por parte da aplicação receptora.

Note que as únicas situações em que um receptor deve enviar dados ao transmissor são: quando deve enviar dados ao emissor ou quando possui um ACK a ser enviado.

Gerenciamento de conexão TCP

Evolução típica de uma conexão TCP:

- um cliente quer iniciar uma conexão com outro processo em outro host (servidor).
- A aplicação cliente informa, inicialmente, ao cliente TCP que deseja estabelecer uma conexão com o processo no servidor.
- O TCP no cliente tenta estabelecer uma conexão TCP com o servidor seguindo uma sequência de etapas.

Etapas:

1. O cliente TCP envia um segmento especial para o servidor TCP; esse segmento não contém dados: SYN. O cliente escolhe aleatoriamente um número de sequência inicial (client_isn) e põe esse número no campo de número de sequência do segmento TCP inicial com SYN. O segmento, por sua vez, é encapsulado em um datagrama IP e enviado ao servidor.

Gerenciamento de conexão TCP

Etapas:

2. Quando o datagrama IP contendo o segmento TCP SYN chega ao servidor, o servidor extrai o TCP SYN do datagrama, aloca buffers e variáveis para a conexão TCP e encaminha o segmento com "conexão aceita/garantida" ao cliente TCP. Esse aviso de "conexão aceita/garantida" não carrega informação alguma da camada de aplicação.
   - Porém, contém 3 importantes peças de informação no cabeçalho do segmento: SYN bit é setado em 1; o campo de acknowledgment do cabeçalho do segmento TCP é ajustado client_isn+1; finalmente, o servidor escolhe seu próprio número de sequência inicial (server_isn) e põe esse valor no campo do número de sequência do cabeçalho do segmento TCP.
   - Esse segmento de "conexão aceita/garantida" representa a seguinte mensagem: "Eu recebi seu pacote SYN para iniciar uma conexão com seu número de sequência inicial, client_isn. Eu concordo em estabelecer essa conexão. Meu número de sequência inicial é server_isn."
   - Segmento **SYNACK**.

Gerenciamento de conexão TCP

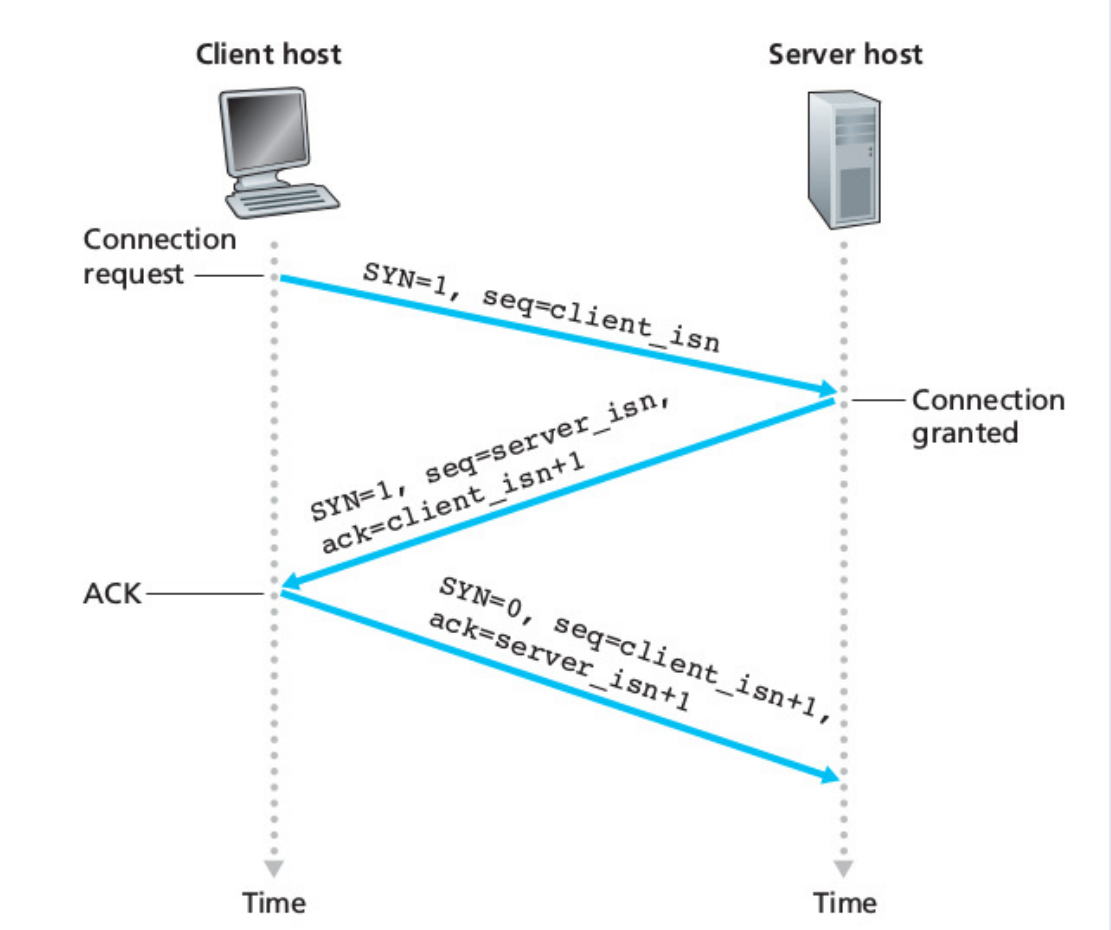
Etapas:

3. Depois de receber um segmento SYNACK, o cliente aloca buffers e variáveis para a conexão. O host cliente então envia ao servidor outro segmento; esse último segmento confirma a recepção do segmento "conexão aceita/garantida" (o cliente faz isso colocando o valor server_isn+1 no campo acknowledgement do cabeçalho do segmento TCP). O bit SYN é zerado, pois a conexão é estabelecida. Nessa terceira etapa, já é possível encaminhar dados cliente-servidor no payload do segmento TCP.

Gerenciamento de conexão TCP

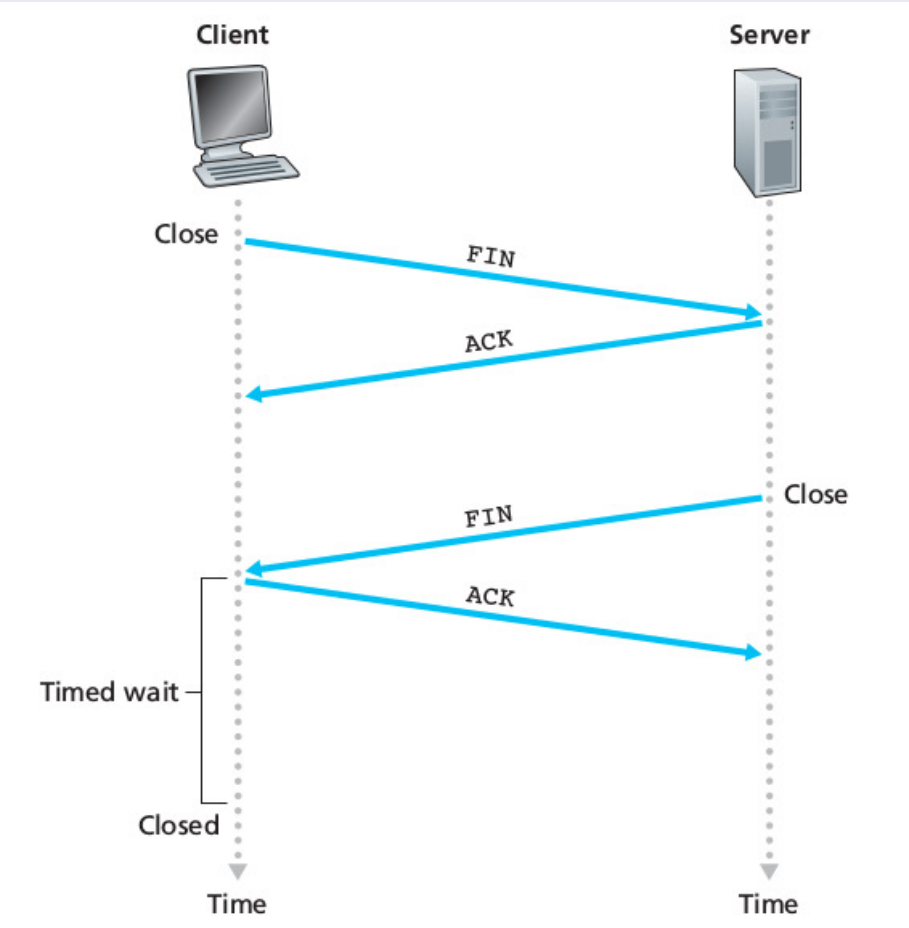
Uma vez que os três passos acima são completados, os hosts cliente e servidor podem enviar segmentos contendo dados entre si.

- Em cada um desses segmentos futuros, o bit SYN é ajustado para zero.
- 3-way handshaking: 3 pacotes são trocados entre os hosts.



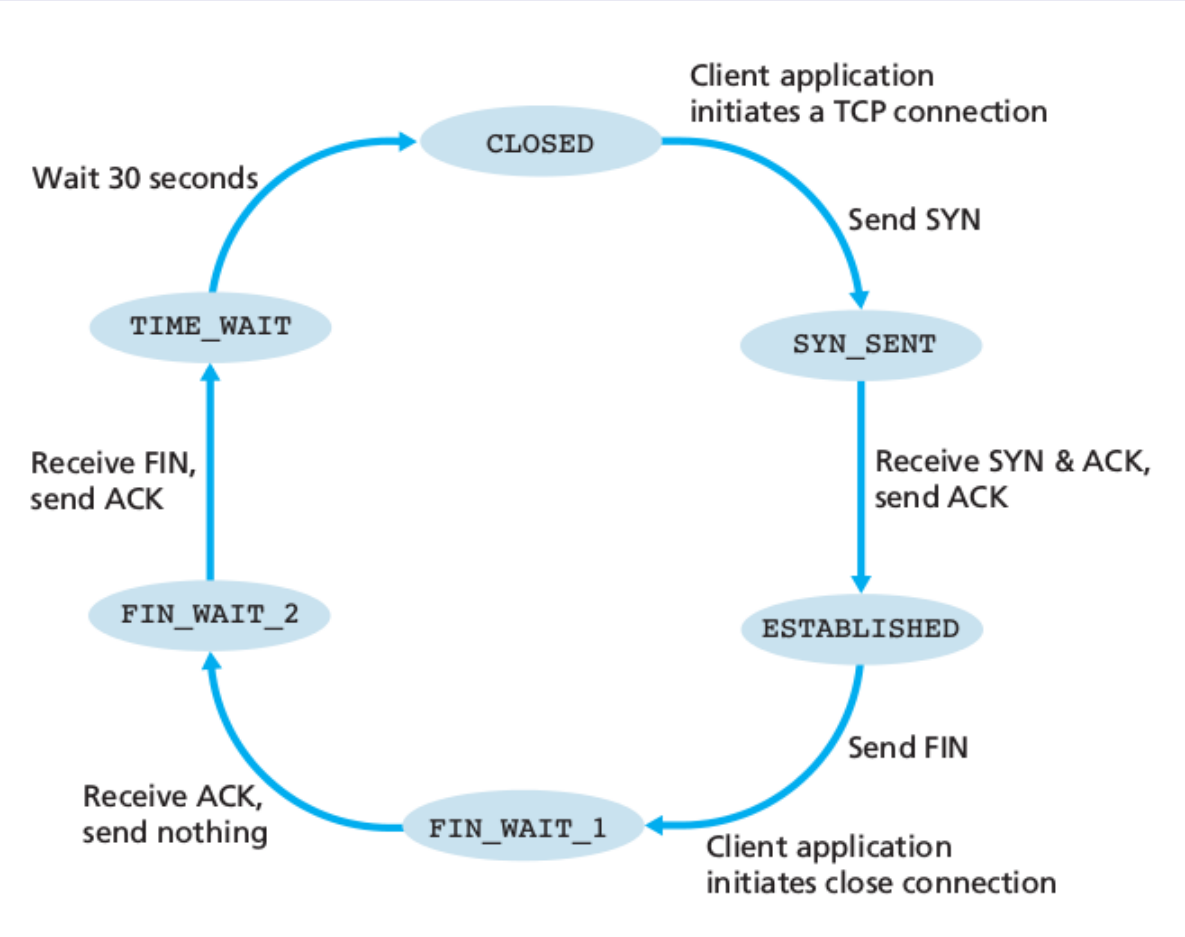
Gerenciamento de conexão TCP

Fechamento de conexão: FIN bit setado em 1 pelo cliente é encaminhado por um segmento ao servidor. Servidor confirma a recepção do segmento e envia, também, um segmento com bit FIN setado. O cliente, por sua vez, confirma o recebimento do FIN transmitido pelo segmento enviado pelo servidor. Conexão encerra.



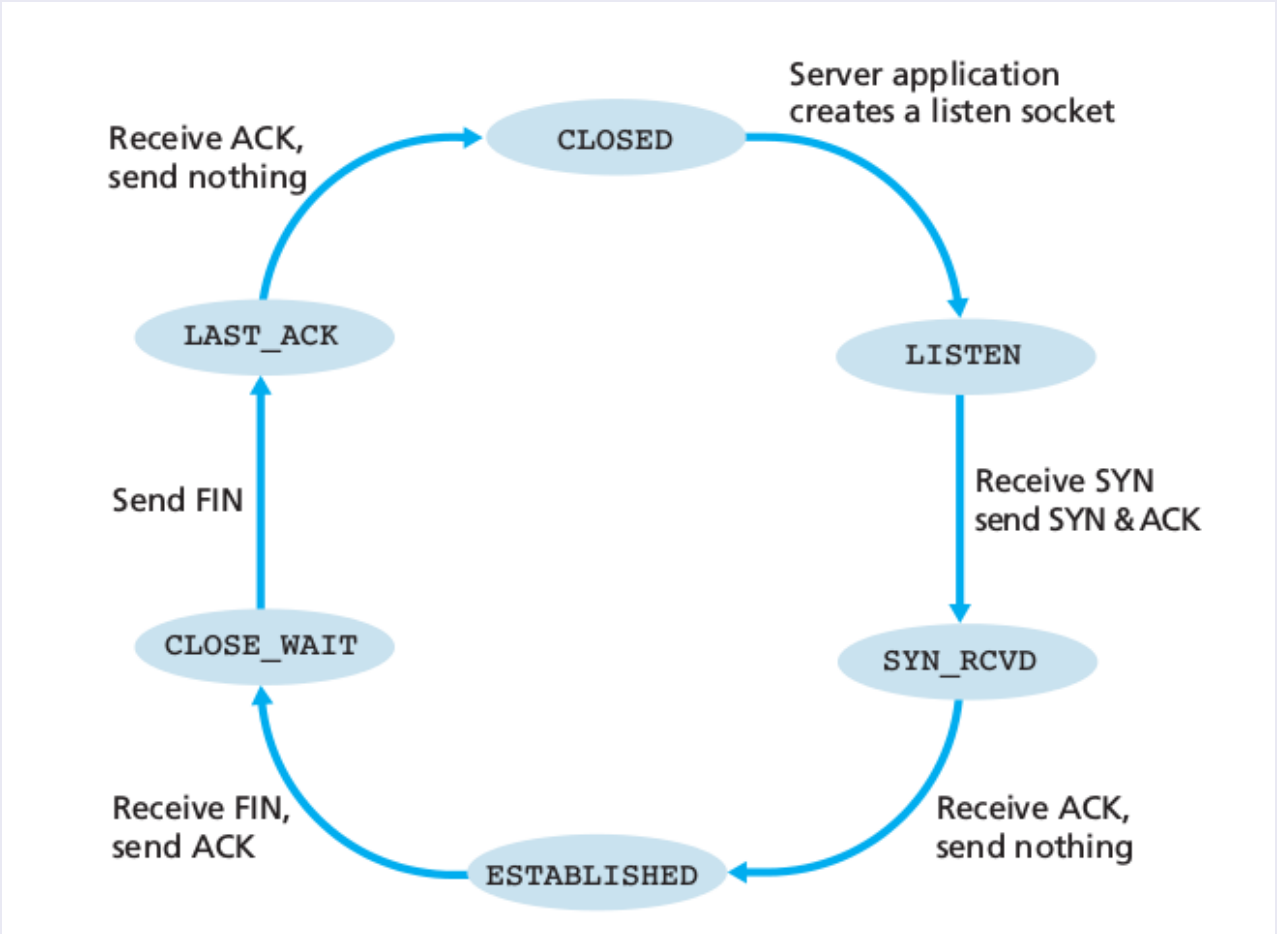
Gerenciamento de conexão TCP

Estados de um cliente TCP:



Gerenciamento de conexão TCP

Estados de um servidor TCP:



Gerenciamento de conexão TCP

No caso em que não há processo sendo servido no servidor: TCP retorna um segmento com o bit RST setado; UDP, por sua vez, retorna um datagrama ICMP especial.

SYN flood: primeiro dos ataques DoS

SYN Cookies: RFC 4987 $\Longleftarrow$ Contramedida

Nmap em 3 cenários: TCP SYNACK recebido do host alvo, TCP RST recebido do host alvo e nenhuma resposta (**firewall**)

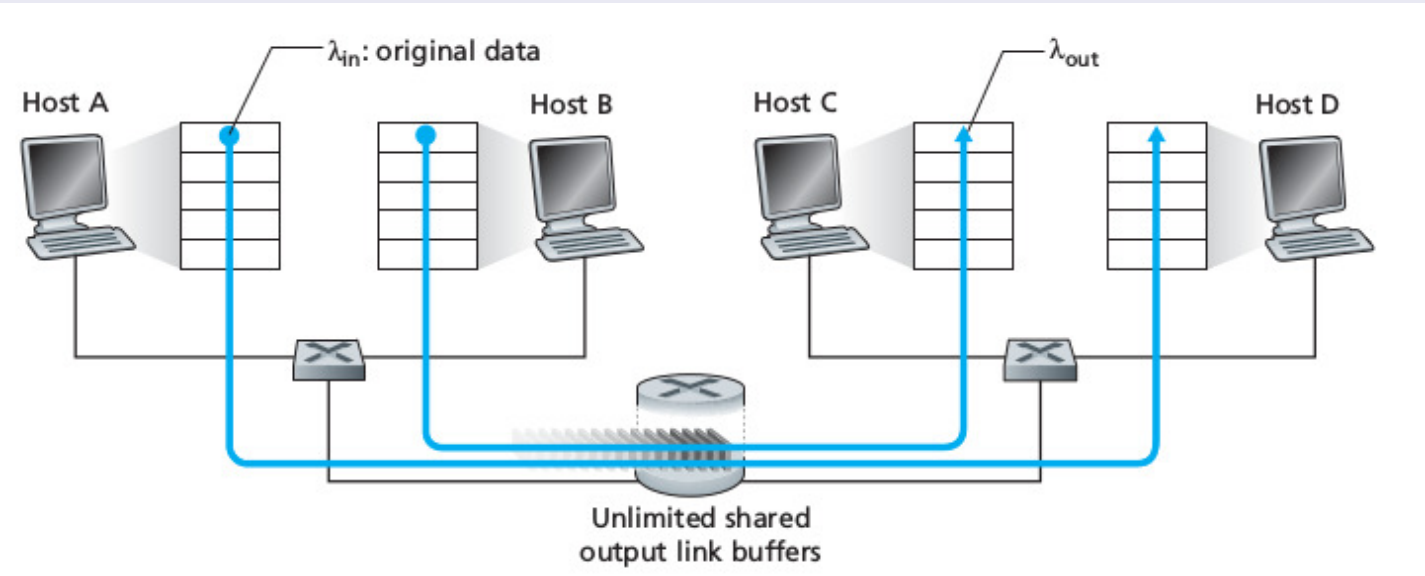
Introdução

Perda de pacotes:

- Provocadas pelo overflow nos buffers dos roteadores submetidos a elevadíssimas cargas de tráfego.
- Contramedida do TCP: retransmissão dos pacotes.
  - Trata, portanto, de um sintoma do congestionamento da rede.
  - Qual é a causa? Muitos emissores tentando enviar pacotes à elevada taxa.
  - Como mitigar: controle de velocidade de emissão nos remetentes

Causas e custos do congestionamento

Cenário 1: 2 emissores, 1 roteador (1 **hop**) com bufferes infinitos



Causas e custos do congestionamento

Cenário 1: 2 emissores, 1 roteador com bufferes infinitos

- A e B conectados em single hop por um roteador.
- Aplicação em A envia dados à taxa de λ_{in} bytes/s.
  - Dado encapsulado e enviado. Não há recuperação de erro (retransmissão), controle de fluxo ou controle de congestionamento.
  - A injeta tráfego à taxa de λ_{in} bytes/s, desconsiderando-se as informações de cabeçalho de camadas de transporte e inferiores.

Causas e custos do congestionamento

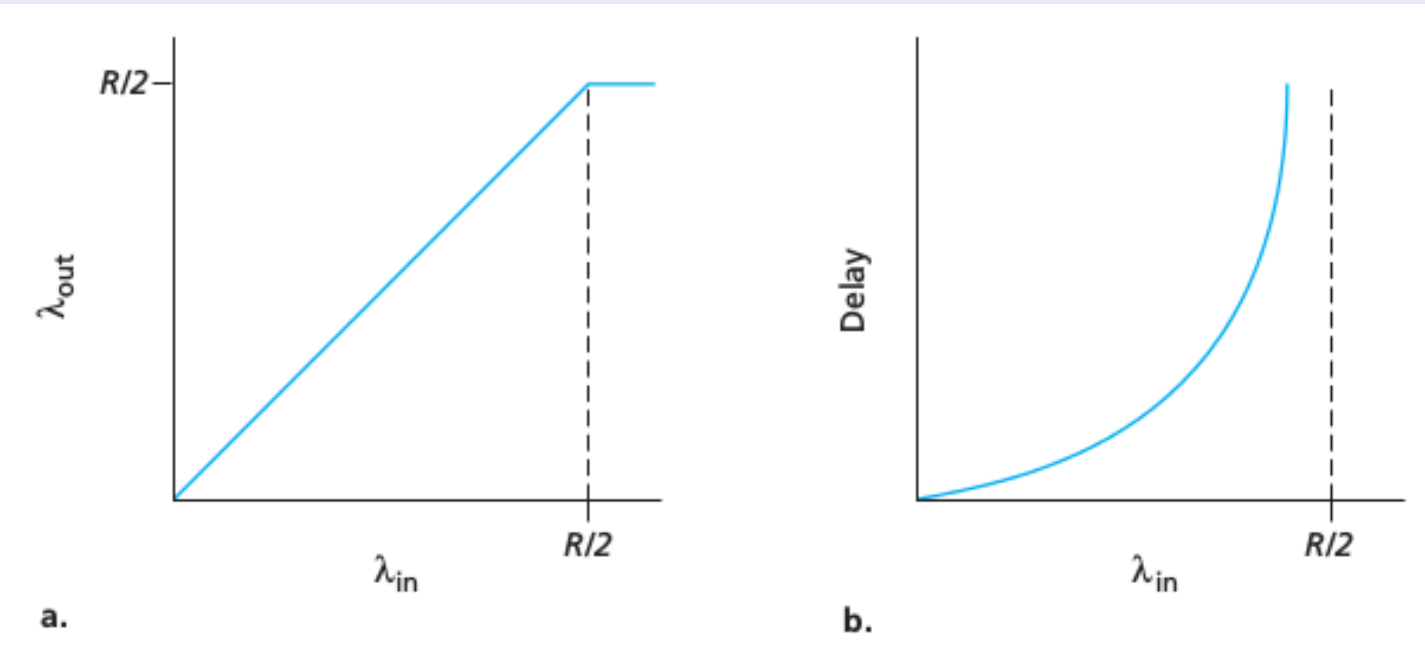
Cenário 1: 2 emissores, 1 roteador com bufferes infinitos

- Aplicação em B opera da mesma forma: injeta tráfego à taxa de λ_{in} bytes/s.
- O roteador de interligação possui capacidade de enlace igual à R bytes/s.
  - Se a velocidade com que chegam os pacotes é maior que a capacidade de saída do enlace, os pacotes são bufferados.
  - O roteador possui buffers infinitos!

Causas e custos do congestionamento

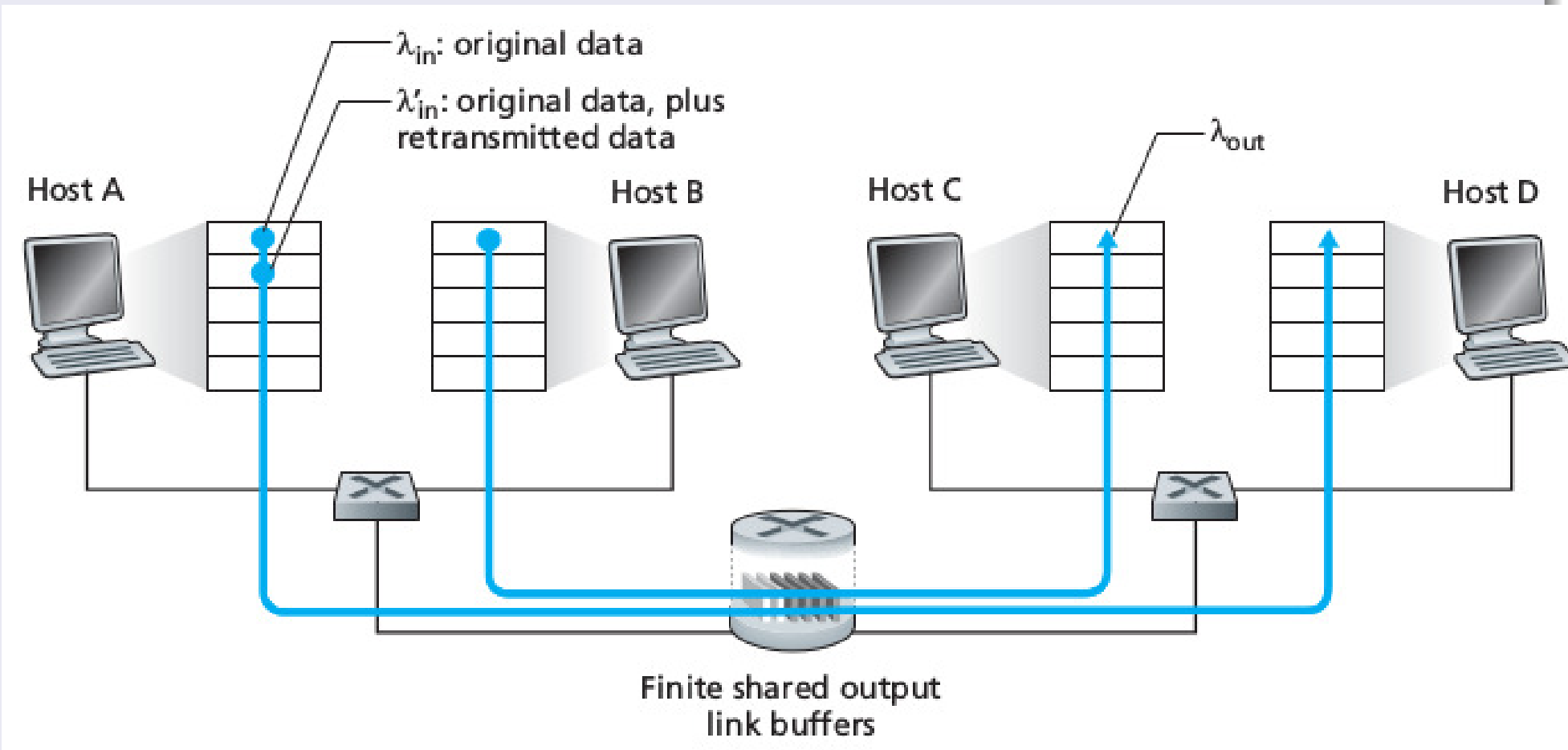
Cenário 1: 2 emissores, 1 roteador com bufferes infinitos

Pontos de inflexão: não há como injetar no canal um volume de informação maior que o mesmo é capaz de conduzir sem enfrentar enfileiramento!



Causas e custos do congestionamento

Cenário 2: 2 emissores e um roteador com bufferes finitos



Causas e custos do congestionamento

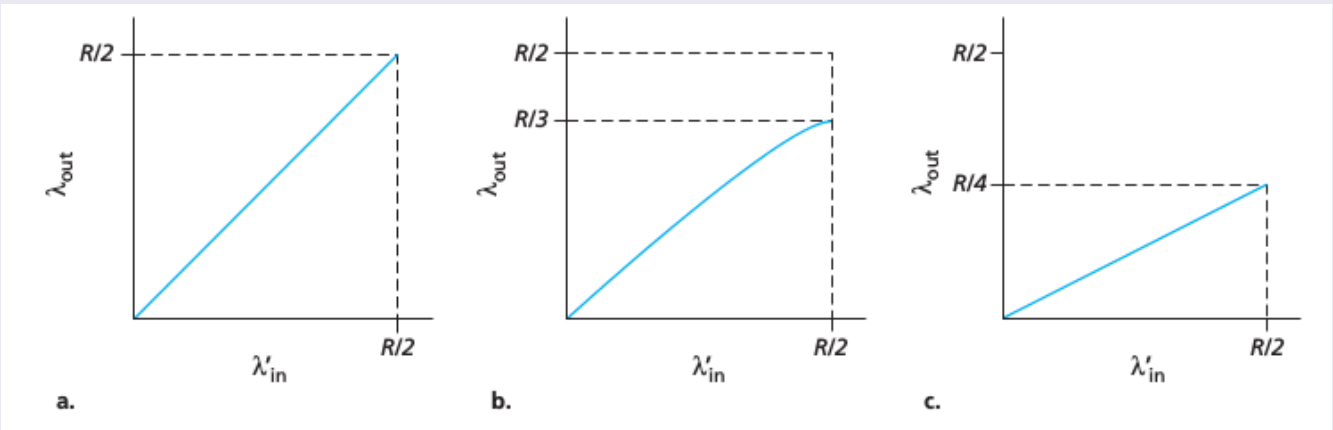
Cenário 2: 2 emissores e um roteador com bufferes finitos

- Pacotes serão descartados quando os bufferes estiverem completamente preenchidos.
- Assume-se, agora, que as conexões são confiáveis $\Longrightarrow$ Retransmissão está presente nesse cenário.
- taxa de envio: λ_{in} bytes/s
- taxa com que a camada de transporte envia segmentos: λ'_{in} bytes/s (não esqueça das retransmissões!).

Causas e custos do congestionamento

Cenário 2: 2 emissores e um roteador com bufferes finitos

$$\lambda'_{\mathbf{in}} \Longrightarrow \textbf{Carga oferecida à rede}$$



a Sem perdas

b Apenas $R/3$ alcança o receptor adequadamente: receptor deverá retransmitir de forma a compensar pacotes perdidos devido a buffer overflow.

c Perdas de pacotes e retransmissão: o tráfego que alcança o receptor é ainda menor!

Causas e custos do congestionamento

Cenário 2: 2 emissores e um roteador com bufferes finitos

- Caso em que A sabe da ocupação dos bufferes do roteador: $\lambda'_{\text{in}} = \lambda_{\text{in}} \Longrightarrow$ Operação no limite, com $\lambda_{\text{out}} = \lambda_{\text{in}}$
- Caso em que somente retransmite quando tem certeza de que o pacote foi perdido:
  - R/3 é entregue ao destino. De R/2 injetados, R/3 são originais e R/6 são provenientes de retransmissões.
  - Custo da rede congestionada: o emissor deverá realizar retransmissões de forma a compensar a perda de pacotes devido a buffer overflow.

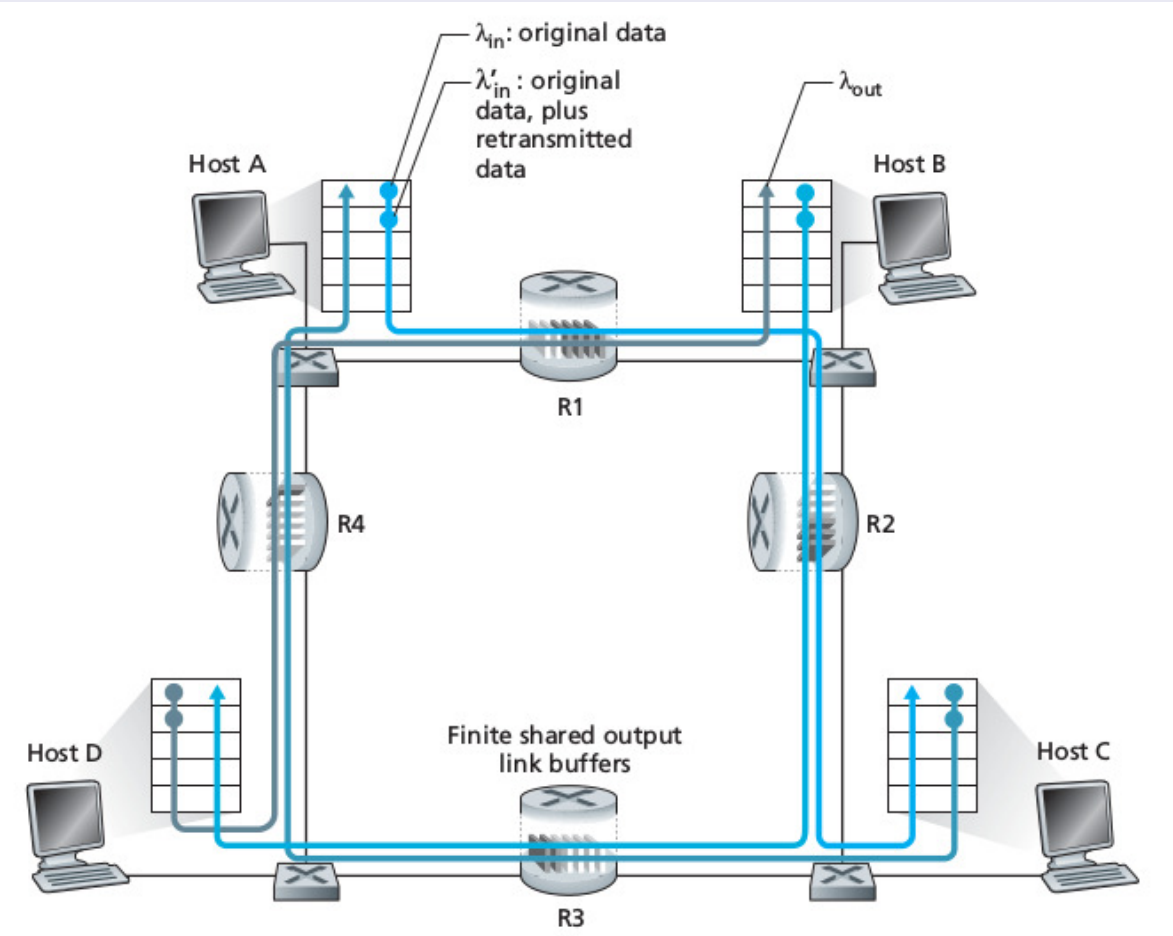
Causas e custos do congestionamento

Cenário 2: 2 emissores e um roteador com bufferes finitos

- Caso em que A retransmite prematuramente um pacote que ficou preso na fila de roteamento, porém não foi descartado.
  - Cópia devido a retransmissão desperdiçou banda.
  - Outro custo de rede congestionada: retransmissões desnecessárias pelo emissor provocadas por grandes atrasos no roteador podem consumir banda passante com o reenvio de cópias desnecessárias de pacotes.
  - Curva assintótica

Causas e custos do congestionamento

Cenário 3: 4 emissores, roteadores com bufferes finitos e rotas com **múltiplos hops**.



Causas e custos do congestionamento

Cenário 3: 4 emissores, roteadores com bufferes finitos e rotas com múltiplos hops

- Cada host usa mecanismo de timeout/retransmissão para implementar transmissão confiável de dados
  - taxa de envio: λ_{in} bytes/s
  - Roteadores de interligação possui capacidade de enlace igual à R.
- Caso
  - A comunica com C através de R1 e R2
  - A-C compartilha R1 com D-B e compartilha R2 com B-D.
  - Para pequenos valores de λ_{in}, buffer overflow é raro e o throughput é aproximadamente igual à carga oferecida à rede.

Causas e custos do congestionamento

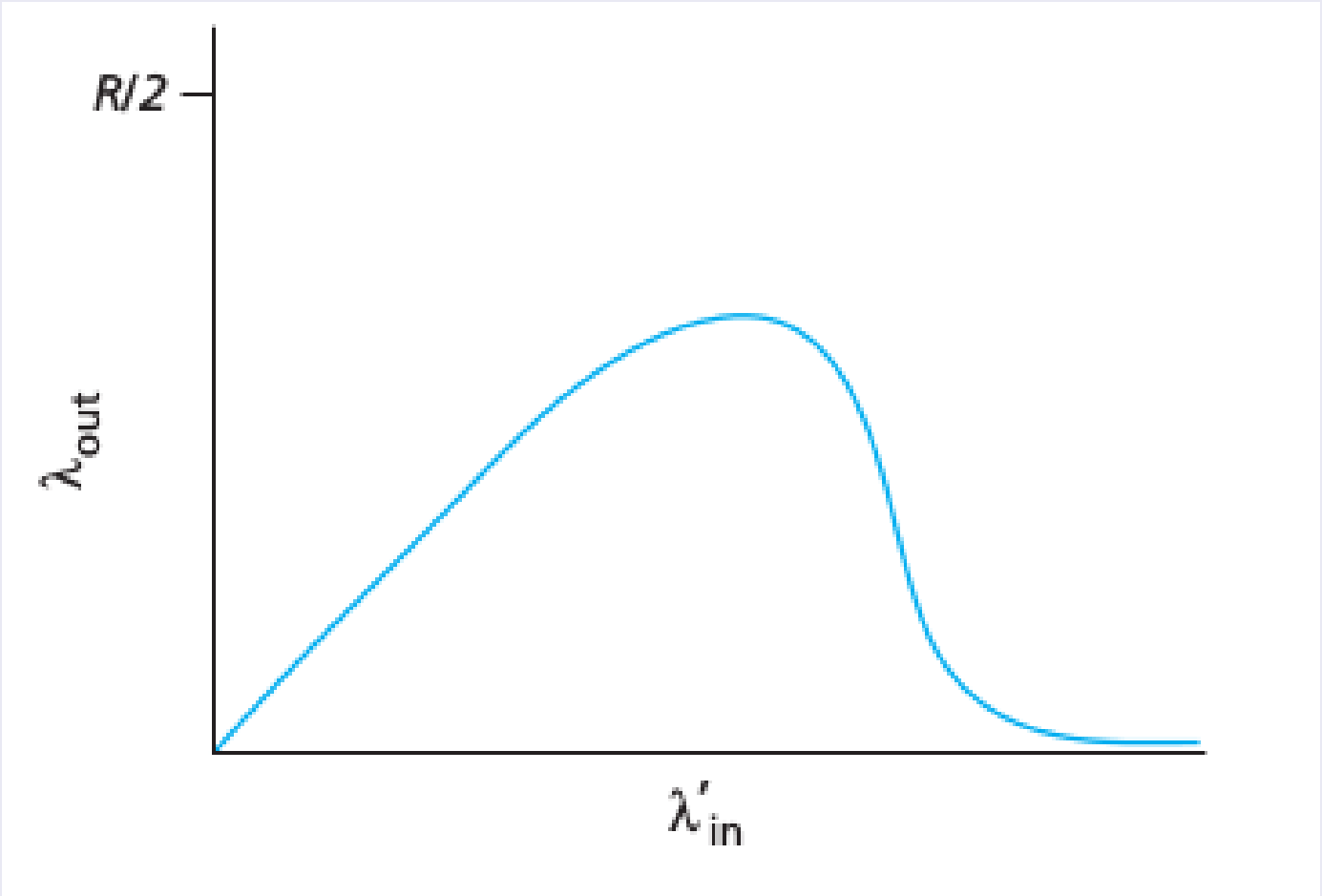
Cenário 3: 4 emissores, roteadores com bufferes finitos e rotas com múltiplos hops

- Caso
  - Para valores maiores (aumento controlado) de λ_{in}, throughput aumenta, e overflows continuam raros. $\Longrightarrow$ Pequenos aumentos em λ_{in} refletem em λ_{out}.
  - Para λ_{in} extremamente elevado, observemos R2.
    - O tráfego A-C em R2 pode consumir R integralmente, a capacidade do enlace R1-R2, independentemente do valor de λ_{in}.
    - Se λ'_{in} for muito elevado para todas as conexões através de R1-R2, a taxa de chegada de B-D poderá ser maior que a do tráfego A-C. Como os dois tráfegos atravessam R2, A-C ficará cada vez menor na medida que B-D consome a banda disponível.
    - No limite, a carga oferecida será infinita, R2 terá buffer completo apenas com tráfego B-D e a conexão A-C **será desligada**. Throughput ZERO devido a tráfego intenso.

Custo de se descartar um pacote: quando um pacote é descartado ao longo da rota, a capacidade de transmissão que foi usada em cada enlace de saída para encaminhar o pacote até o ponto em que foi descartado também é desperdiçada.

Causas e custos do congestionamento

Cenário 3: 4 emissores, roteadores com bufferes finitos e rotas com múltiplos hops



Abordagens para o controle de congestionamento

Controle de congestionamento ponto-a-ponto:

- Camada de rede não provê suporte explícito para a camada de transporte para controle de congestionamento
- O estado de congestionamento é inferido pelos end systems
- Abordagem usada no TCP:
  - Sintomas: perdas de segmentos TCP
  - Reação: diminuição de janelas de transmissão.

Abordagens para o controle de congestionamento

Controle de congestionamento com auxílio de informações da rede

- Roteadores provêem informações explícitas para o emissor quanto ao estado de congestionamento da rede $\Longrightarrow$ Bit de estado.
- ATM

Introdução

Do tipo ponto-a-ponto

Cada emissor limitará a taxa na qual enviará tráfego pela conexão como função do congestionamento percebido.

- Pouco congestionamento, aumento da taxa de envio.
- Muito congestionamento, redução da taxa de envio.
- Variável de estado no emissor: congestion window: $cwnd$

Introdução

Condições

- quantidade de dados não confirmados (sem recepção de acknowledgement) que um emissor não deve exceder

$$LastByteSent - LastByteAcked \leq min(cwnd, rwnd)$$

- Assumir que a janela de recepção do cliente é ilimitada: a quantidade de informação que o servidor não confirmou é limitada por $cwnd$
- O cliente sempre possui dado a ser enviado
- A restrição acima controla a quantidade de dados não confirmados e, indiretamente, limita a taxa de transmissão do emissor.

Introdução

Cenário 1: Conexão sem perda de pacotes e sem atraso de transmissão

- O transmissor pode inserir na rede $cwnd$ bytes (início de RTT)
- As confirmações, teoricamente, deveriam chegar no final de um RTT
- Taxa de transmissão: $cwnd/RTT$ bytes/s
- Ajustando o valor de $cwnd$,o emissor poderá modular sua velocidade.

Introdução

Cenário 2: congestionamento

- evento de perda do ponto de vista do emissor: timeout ou recepção de 3 ACKs duplicados do receptor.
- Congestionamento grave: enchimento dos bufferes dos roteadores e eventuais perdas de pacotes
- perda de pacote $\Longrightarrow$ evento de perda detectado pelo emissor

Introdução

Cenário 3: rede não congestionável

- Não ocorrem eventos de perdas
- Emissor sente o nível de congestionamento da rede através dos ACKs recebidos e modulará cwnd de acordo com a chegada dos ACKs
- Se ACKs chegam a taxas pequenas, a janela de congestionamento crescerá lentamente
- Se ACKs chegam a taxas altas, a janela de congestionamento poderá crescer rapidamente

Introdução

TCP: protocolo auto-sincronizado (self-clocking)

Como o TCP deverá determinar a taxa adequada para transmissão?

- Emissores enviando simultaneamente muito rapidamente: congestionamento da rede.
- Emissores muito cautelosos: capacidade instalada subtilizada!

Princípios seguido pelo TCP para controle de congestionamento

Um segmento perdido indica provável congestionamento. Dessa forma, a taxa de transmissão do emissor deverá ser reduzida no caso de perda de segmentos.

- Qual será a sistemática adequada para redução da janela de congestionamento quando é identificado/inferido um evento de perda?

Um segmento **confirmado** indica que o segmento chegou ao receptor. Dessa forma, a taxa de transmissão do emissor poderá ser aumentada com o recebimento de um ACK. A rede não está congestionada.

Princípios seguido pelo TCP para controle de congestionamento

Teste de largura de banda

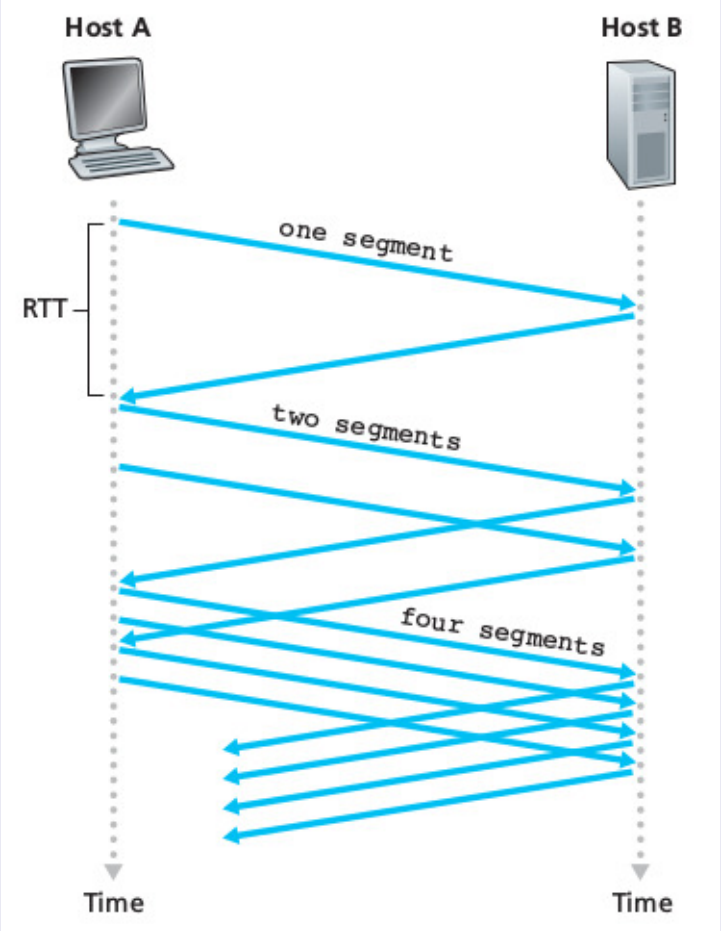
- Aumento da taxa de transmissão sempre que confirmação de recebimento for sinalizada.
- Processo deve ser interrompido sempre que uma perda for identificada: a taxa de transmissão deverá ser reduzida.
- Mecanismo de sensibilização:
  - Inicia com um processo em que a velocidade tende a crescer.
  - Quando uma perda ou ACK duplicado for identificado, o processo é interrompido. A taxa é reduzida e as confirmações são monitoradas.
  - Oscilação para convergência

Atenção: não há sinalização explícita de congestionamento!

Algoritmo de controle de congestionamento do TCP [RFC 5681]

Slow Start

- cwnd = 1 MSS $\Longrightarrow$ rate = MSS/RTT
- incrementa cwnd de 1 MSS a cada ACK recebido



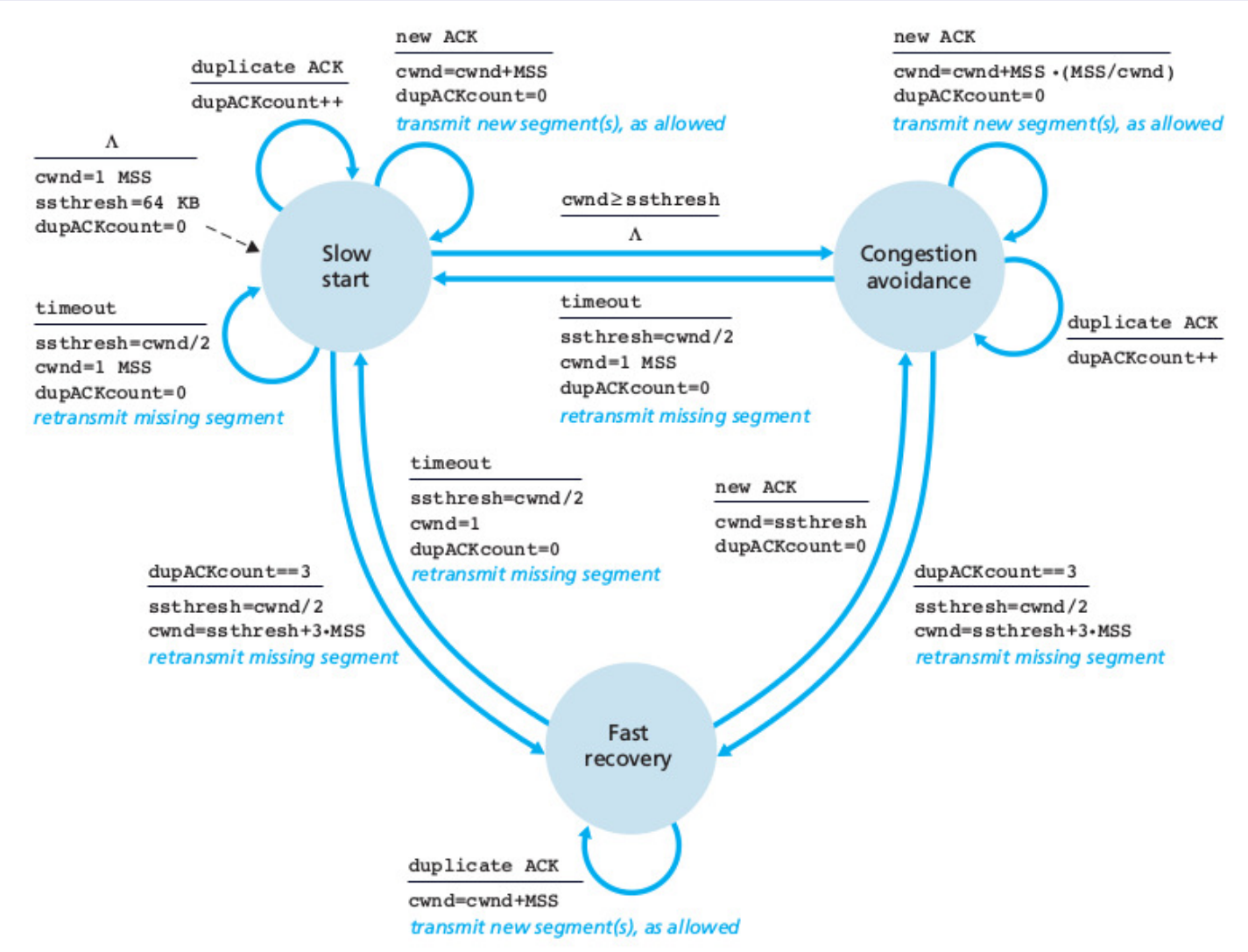
Algoritmo de controle de congestionamento do TCP [RFC 5681]

Slow Start

- duplicação da cwnd a cada RTT confirmado: crescimento exponencial
- Ocorrência de um timeout
  - variável de estado ssthresh = cwnd/2, onde cwnd é o valor do tamanho da janela de congestionamento quando da detecção do congestionamento
  - cwnd = 1 MSS e processo de início disparado novamente
  - quando cwnd crescer e alcançar ssthresh, o processo de alargamento da janela é interrompido
- Ocorrência de 3 ACKs duplicados $\Longrightarrow$ fast restransmit e fast recovery

Algoritmo de controle de congestionamento do TCP [RFC 5681]

Slow Start: máquina de estados



Algoritmo de controle de congestionamento do TCP [RFC 5681]

Mitigação de Congestionamento (Congestion Avoidance)

- cwnd = sshtresh
- aumento de cwnd por 1 a cada RTT $\Longrightarrow$ aumentar cwnd por MSS bytes a cada novo ACK
- ocorrência de timeout:
  - variável de estado ssthresh = cwnd/2, onde cwnd é o valor do tamanho da janela de congestionamento quando da detecção do congestionamento
  - cwnd = 1 MSS
- ocorrência de 3 ACKs duplicados
  - variável de estado ssthresh = cwnd/2, onde cwnd é o valor do tamanho da janela de congestionamento quando da detecção do congestionamento
  - cwnd = (cwnd+3MSS)/2

Algoritmo de controle de congestionamento do TCP [RFC 5681]

Fast recovery

- cwnd é aumentado de 1 MSS a cada ACK duplicado recebido para o segmento perdido
- ocorrência de timeout:
  - variável de estado ssthresh = cwnd/2, onde cwnd é o valor do tamanho da janela de congestionamento quando da detecção do congestionamento
  - cwnd = 1 MSS

Algoritmo de controle de congestionamento do TCP [RFC 5681]

Fast recovery