# **Anotações dos slides de InfraCom**

## **◖Módulo 3◗**

|  |
| --- |
| **Objetivos :**   * Compreender os princípios por trás dos serviços de transporte:   + Multiplexação e Demultiplexação   + Transferência confiável de dados   + Controle de fluxo   + Controle de congestionamento * Aprender sobre os protocolos da camada de transporte da Internet:   + UDP (Transporte não-orientado à conexão)   + TCP (Transporte orientado à conexão)   + Controle de congestionamento do TCP |

## **3.1 Serviços da camada de transporte**

* Serviços e Protocolos de transporte provêem **comunicação lógica** entre processos de aplicação executando em diferentes hosts.
* Protocolos de transporte “rodam” em **end systems** (sistemas finais)
  + **Lado emissor** : quebra as mensagens da aplicação em segmentos, que são passados a camada de rede. (Multiplexação)
  + **Lado receptor** : remonta os segmentos em mensagens e os passa à camada de aplicação. (Demultiplexação)
* Há mais de um protocolo de transporte disponível para as aplicações de rede, a internet possui dois : **TCP** e **UDP**.

### **3.1.1** **Camada de Transporte vs. Camada de Rede**

* **Camada de Rede** : fornece uma comunicação lógica entre hospedeiros.
* **Camada de Transporte** : fornece uma comunicação lógica entre processos.

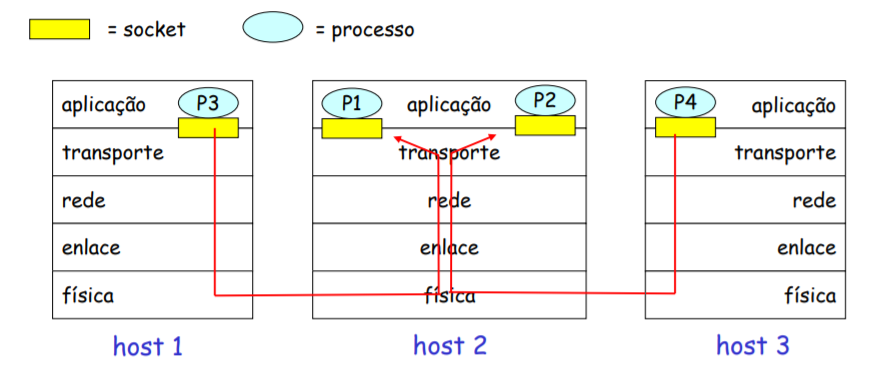
|  |
| --- |
| O livro usa a seguinte analogia :  Considere duas casas, uma na Costa Leste e outra na Costa Oeste dos Estados Unidos, cada qual com uma dúzia de crianças. As crianças da Costa Leste são primas das crianças da Costa Oeste e todas adoram escrever cartas umas para as outras — cada criança escreve a cada primo uma vez por semana e cada carta é entregue pelo serviço de correio tradicional dentro de um envelope separado. Assim, uma casa envia 144 cartas por semana para a outra. (Essas crianças economizariam muito dinheiro se tivessem email!) Em cada moradia há uma criança responsável pela coleta e distribuição da correspondência — Ana, na casa da Costa Oeste, e Bill, na da Costa Leste. Toda semana, Ana coleta a correspondência de seus irmãos e irmãs e a coloca no correio. Quando as cartas chegam à casa da Costa Oeste, também é Ana quem tem a tarefa de distribuir a correspondência, trazida pelo carteiro, a seus irmãos e irmãs. Bill realiza o mesmo trabalho na casa da Costa Leste.   * mensagens de aplicação = cartas em envelopes * processos = primos * hospedeiros (também denominados sistemas finais) = casas * protocolo de camada de transporte = Ana e Bill * protocolo de camada de rede = serviço postal (incluindo os carteiros) |

### **3.1.2 Protocolos da camada de transporte da Internet**

* **TCP**
  + Entrega confiável, em ordem
  + Estabelecimento de conexão
  + Controle de congestionamento
  + Controle de fluxo
* **UDP**
  + Entrega não-confiável, sem garantias de ordenação
  + Extensões “sem ornamentos” ao serviço de melhor esforço (best effort) IP
* Serviços **indisponíveis**:
  + Garantias de atraso
  + Garantias de banda passante

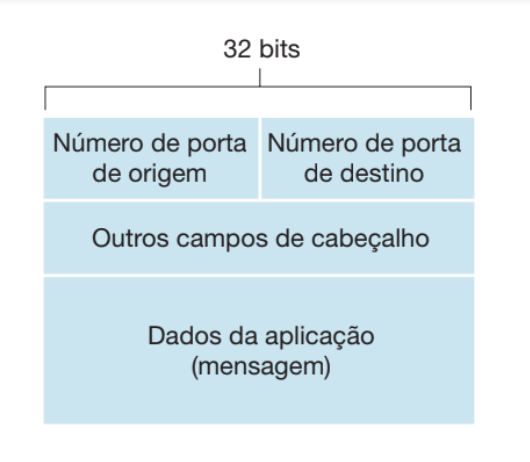
## **3.2 Multiplexação e Demultiplexação**

|  |  |
| --- | --- |
| * **Demultiplexação no host receptor :**   **Usa as informações do cabeçalho para entregar os segmentos recebidos aos sockets corretos.** | * **Multiplexação no host emissor :**   **Coletar dados dos vários sockets, adicionar cabeçalho aos dados (mais tarde usado para demultiplexação)** |



### ♦ **Como a demultiplexação funciona**

* Host recebe **datagramas IP**
  + Cada datagrama possui endereço IP fonte e endereço IP destino.
  + Cada datagrama carrega **1** segmento da camada de transporte
  + Cada segmento possui número de porta de origem e porta de destino
* Host usa os **endereços IP** e **número das portas** para enviar segmento ao socket adequado.



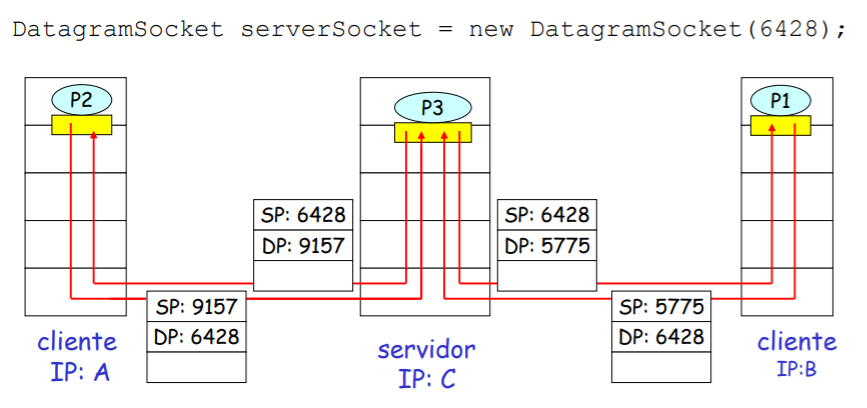
Formato do segmento **TCP**/**UDP**

### ♦ Demultiplexação com UDP

|  |
| --- |
| Criar sockets com portas respectivas :  DatagramSocket mySocket1 = new DatagramSocket(99111);  DatagramSocket mySocket2 = new DatagramSocket(99222); |

* Um socket **UDP** é identificado pela tupla :
  + Endereço IP de destino
  + Número da porta de destino
* Datagramas IP com endereços IP fonte e/ou números de porta de origem diferentes, mas com endereços IP de destino e números de porta de destino iguais, são direcionados ao mesmo socket.

Quando um host recebe um segmento UDP, ele verifica o **número da porta de destino** do segmento e direciona o segmento **UDP** para o socket com o número de porta especificado.



**SP: Source Port (porta de origem) e DP: Destination Port (porta de destino)**

### ♦ Demultiplexação com TCP

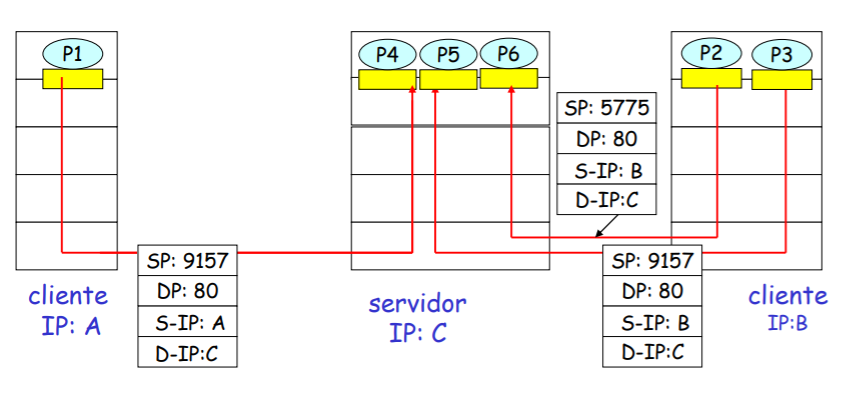
* Um socket **TCP** é identificado pela tupla quádrupla :
  + Endereço IP de origem
  + Número da porta de origem
  + Endereço IP de destino
  + Número da porta de destino

O host receptor usa todos esses quatro valores para enviar o segmento ao socket apropriado.

* Um host servidor pode suportar diversos sockets TCP simultaneamente.
* Servidores Web possuem sockets diferentes para cada cliente conectado
  + HTTP não-persistente terá diferentes sockets para cada requisição

Quando um host recebe um segmento TCP, ele verifica o **número da porta de origem e destino** do segmento e direciona o segmento **TCP** para o socket com o número de porta especificado.

⇓ Demultiplexação com TCP ⇓



⇓ Demultiplexação com TCP: Web Server com threads ⇓



## **3.3 UDP : Transporte não-orientado à conexão**

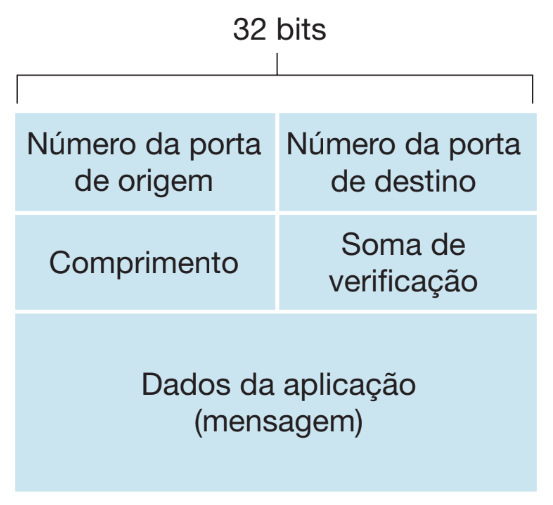
### ♦ UDP : User Datagram Protocol

* Protocolo Internet de transporte “sem ornamentos” e com “elementos básicos”
* Não-orientado à conexão :
  + Sem **handshaking** entre emissor e receptor
  + Cada segmento UDP é tratado de forma independente dos outros.
* Serviço “best effort”, os segmentos UDP podem ser :
  + **perdidos**
  + entregues **fora de ordem** à aplicação
* Por que o **UDP** existe?
  + Não há estabelecimento de conexão (que pode adicionar atraso).
  + Simples : sem estado de conexão no emissor, nem no receptor.
  + O cabeçalho do segmento é pequeno. (O **UDP** tem exatamente ***8 bytes*** de excesso/overhead de cabeçalho, enquanto o **TCP** tem **PELO MENOS** ***20 bytes***)
  + Nenhum controle de congestionamento (não há limite de velocidade)

|  |
| --- |
| * O **UDP** é frequentemente usado para :   + Aplicações multimídia de **streaming**     - Tolerante à perdas     - Sensível à taxa de dados   + **DNS**   + **SNMP** |

* Transferência confiável com **UDP** :
  + Confiabilidade adicionada na camada de aplicação (recuperação de erros específica da camada de aplicação)

### 3.3.1 Estrutura do segmento UDP



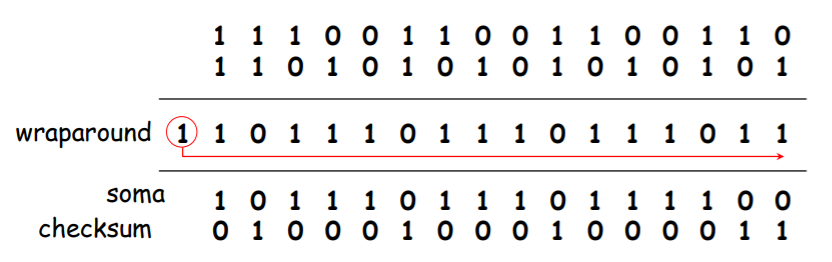
* Cabeçalho UDP **(8 bytes)** :

4 campos cada um com 2 bytes (16 bits)

* + Número de porta de origem
  + Número de porta de destino
  + Comprimento
  + Soma de verificação (checksum)
* Os dados da aplicação ocupam o campo de dados do segmento UDP.

### 3.3.2 Soma de verificação UDP

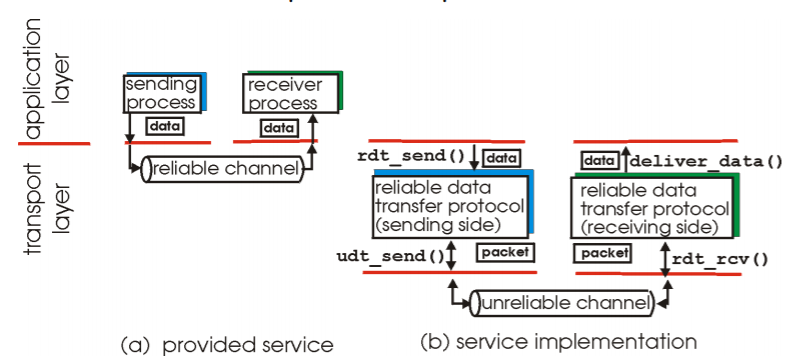
* Tem como objetivo detectar “erros” (por exemplo, bits trocados) no segmento transmitido.
* Emissor :
  + Trata o conteúdo de segmentos como uma sequência de inteiros de 16 bits.
  + Checksum: adição (Soma complemento 1) do conteúdo do segmento.
  + Emissor coloca o valor do checksum no campo checksum do UDP.
* Receptor :
  + Computa o checksum do segmento recebido.
  + Verifica se o checksum computado bate com o valor informado no campo checksum.
    - Não - erro detectado
    - Sim - nenhum erro detectado. (Mas pode haver erros)



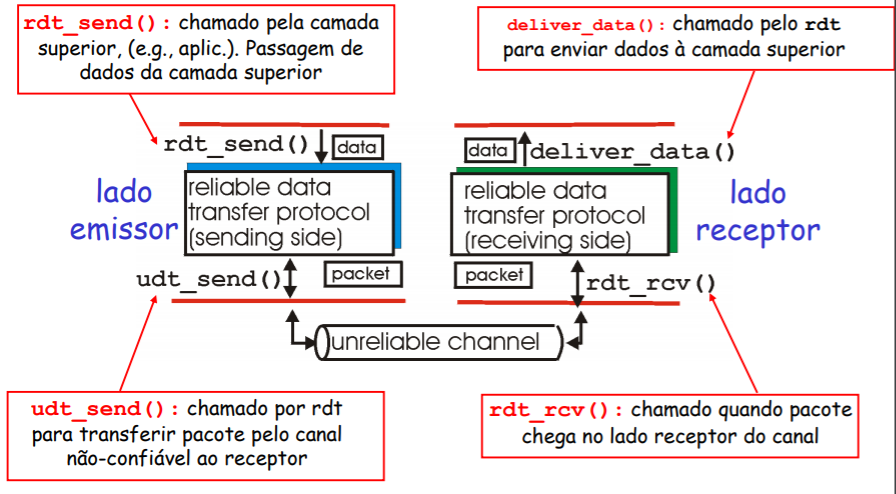
Exemplo de checksum Internet

## **3.4 Princípios da transferência confiável de dados**

* É um fator importante nas camadas de aplicação, transporte e enlace.
* Está na lista dos 10 tópicos mais importantes para o trabalho em redes.



* Características do canal não confiável determinará a complexidade do protocolo de transferência confiável de dados (rdt).



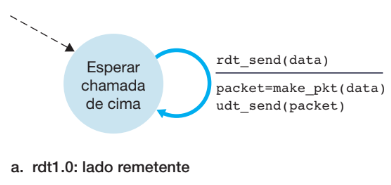
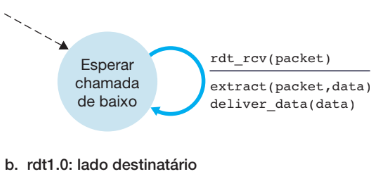
### 3.4.1 Construindo um protocolo de transferência confiável de dados

* Iremos desenvolver incrementalmente os lados receptor e emissor do protocolo de **transferência confiável de dados (rdt)**.
* Considere somente transferência de dados unidirecional, mas as informações de controle transitarão em ambos sentidos.
* Usaremos máquinas de estado finitas (FSM) para especificar o emissor e o receptor.



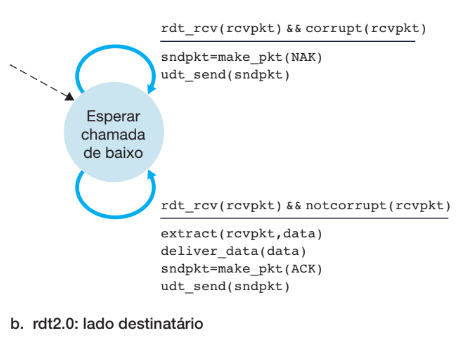
### ♦ Transferência confiável de dados por um canal perfeitamente confiável : rdt 1.0

* Canal base **perfeitamente confiável**
  + Nenhum erro nos bits.
  + Nenhuma perda de pacotes.
* FSMs separadas para emissor e receptor
  + **Emissor** envia dados no canal.
  + **Receptor** lê dados do canal.

### ♦ Transferência confiável de dados por um canal com erros de bits : rdt 2.0

* Canal de base pode **trocar bits dos pacotes**
  + Nenhuma perda de pacotes
* Usa-se o checksum para detectar erros nos bits
* Como se recuperar de erros:
  + Acknowledgements **(ACKs)** : Receptor informa explicitamente ao emissor que o pacote recebido está OK.
  + Negative acknowledgements **(NAKs)** : Receptor informa explicitamente ao emissor que o pacote recebido possui erros.
  + Emissor **retransmite** pacote ao receber um **NAK**.
* Mecanismos novos no rdt 2.0 (além de rdt1.0)
  + **Detecção de erro**
  + **Feedback do receptor** : msgs de controle (ACK, NAK) do receptor para o emissor

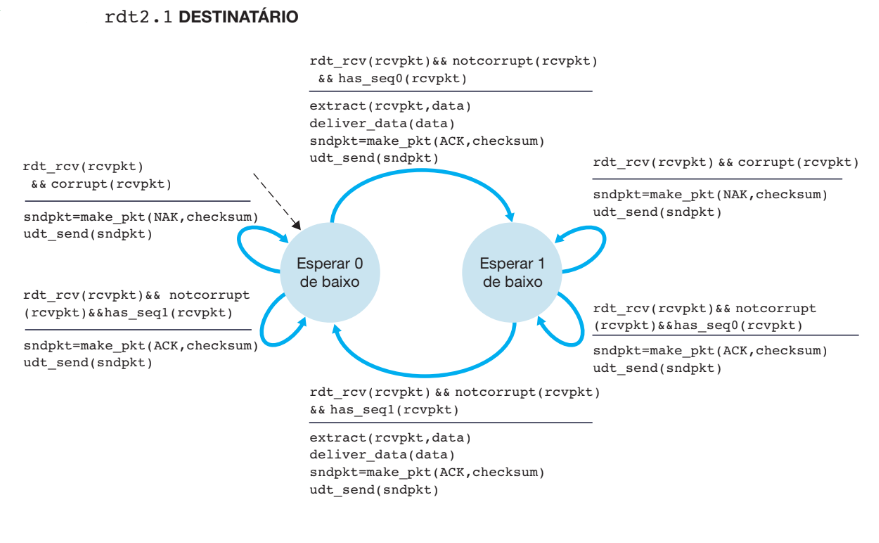


O rdt2.0 tem um **problema fatal**!

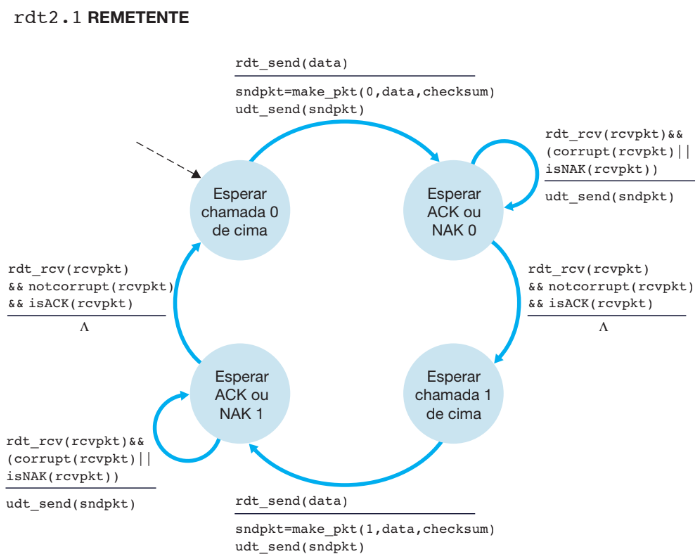
* E se o ACK/NAK é **corrompido**?
  + O emissor *não sabe* o que aconteceu com o receptor e *não pode* simplesmente retransmitir (possibilidade de **duplicação de pacotes**)
* Tratando de duplicações :
  + Emissor retransmite o pacote atual se o ACK/NAK é corrompido.
  + Emissor adiciona **números de sequência** a cada pacote.
  + Receptor descarta (não entrega para a camada superior) pacotes duplicados.
  + **Stop and wait** : Emissor envia 1 pacote e então aguarda resposta do receptor

#### ♦ Emissor trata de **ACK/NAKs corrompidos** : rdt 2.1

* Emissor :
  + Número de sequência adicionado ao pacote, 2 números (0, 1) são suficientes.
  + Deve **verificar** se ACK/NAK recebido está *corrompido*.
  + 2 vezes mais estados
    - Estado deve saber se o pacote “atual” possui número de sequência 0 ou 1.

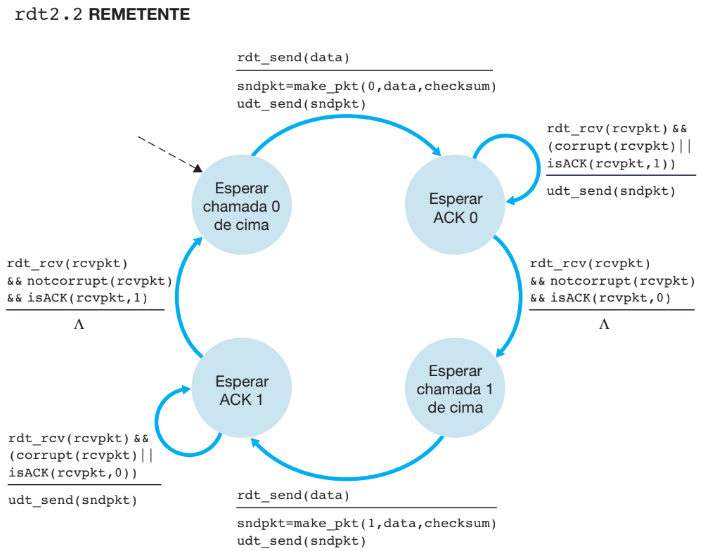


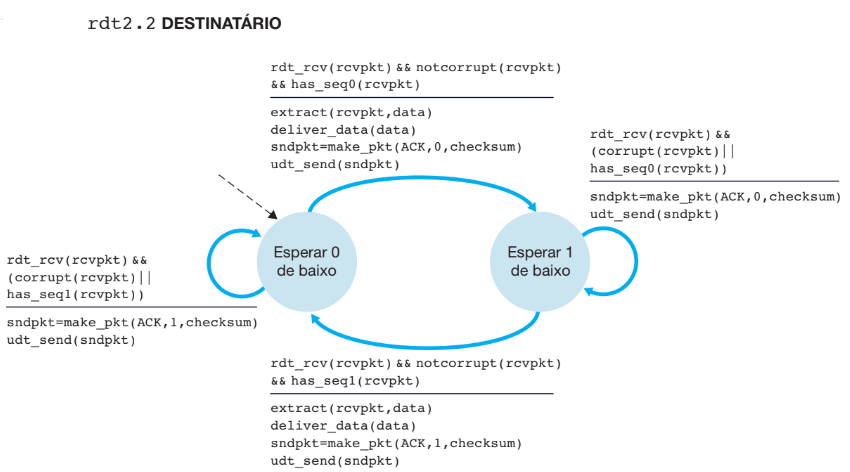
* Receptor :
  + Deve **verificar** se o pacote recebido é um *pacote duplicado*
    - Estado indica se o número de sequência do pacote esperado é 0 ou 1
  + Receptor não pode saber se seu último ACK/NAK foi recebido OK no emissor.



#### ♦ Protocolo **sem NAKs** : rdt 2.2

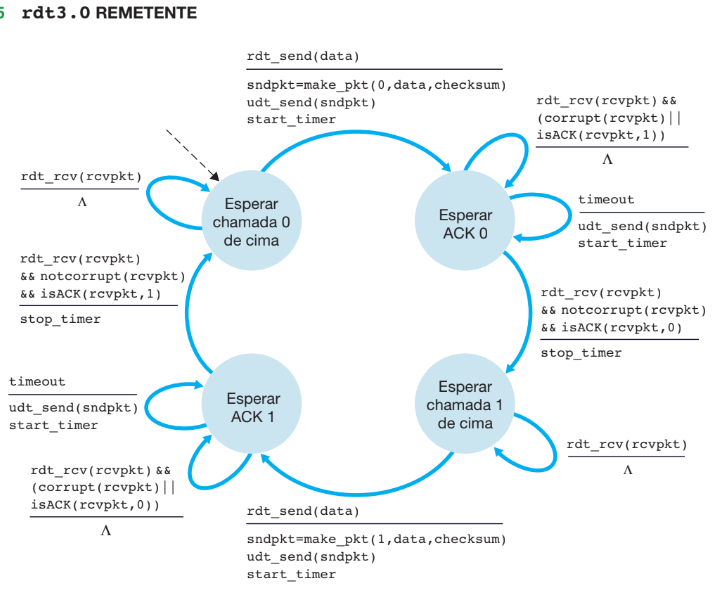
* Igual ao rdt2.1 mas usando **somente ACKs**.
* Ao invés de NAK, receptor envia ACK do último pacote recebido corretamente (deve incluir explicitamente o número de sequência do pacote sendo confirmado)
* **ACK duplicado** no emissor resulta na mesma ação como para o NAK : retransmissão do pacote atual.

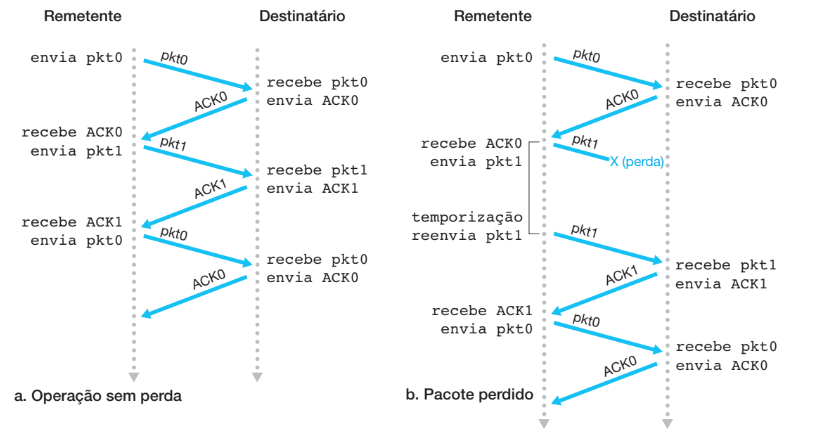


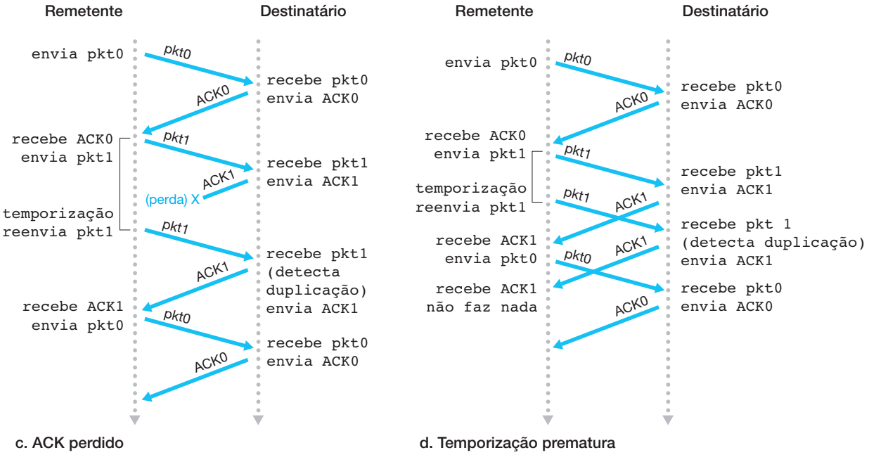


### ♦ Transferência confiável de dados por um canal com perda e com erros de bits : rdt 3.0

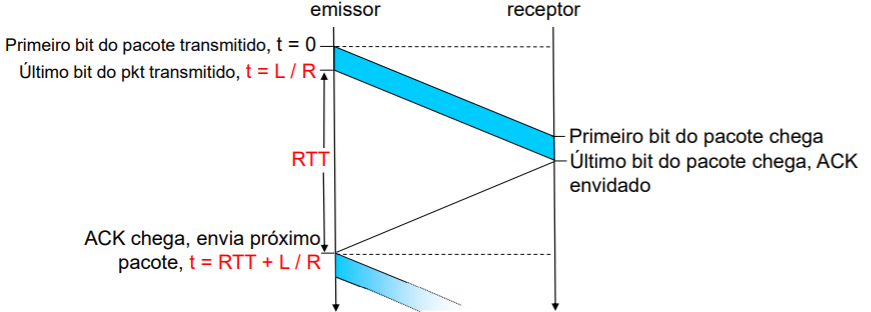
* Canal de base pode agora perder pacotes (dados ou ACKs)
  + Checksum, número de sequência, ACKs, retransmissões ajudarão mas não serão suficientes.
* **Abordagem** : Emissor aguarda um tempo “razoável” a recepção de ACKs
  + Retransmite se nenhum ACK é recebido neste tempo.
  + Se o pacote ou ACK estiver apenas atrasado (não foi perdido)
    - Retransmissão causará duplicação, mas o uso de número de sequência trata isso (receptor deve especificar o número de sequência do pacote sendo confirmado)
* Requer **temporizador**

****

****

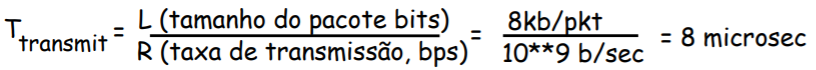
****

* Desempenho do rdt3.0

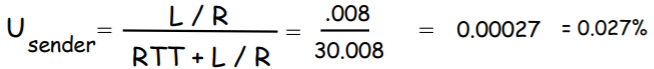


* + O rdt3.0 funciona, mas o desempenho é ruim

Exemplo : enlace de 1 Gbps, 15 ms de atraso de propagação fim-a-fim, pacote de 1KB.



Utilização: Fração do tempo em que o emissor está ocupado enviando.

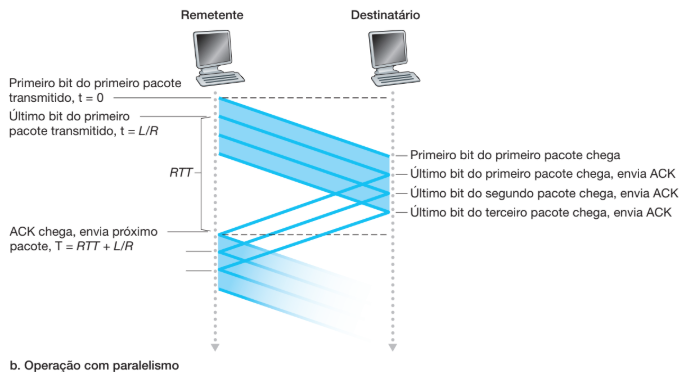


1 pacote de 1 KB a cada 30 msec ⇒ 33 kB/sec de vazão em um enlace de 1 Gbps.

* + O protocolo de rede **limita** o uso de recursos físicos!

### 3.4.2 Protocolos de transferência confiável de dados com paralelismo (Pipeline)

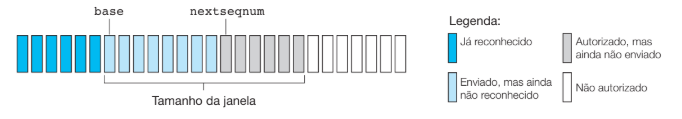
* **Pipelining** : emissor permite múltiplos pacotes enviados e que ainda não foram confirmados.
  + Alcance do **número de sequência** deve ser aumentado.
  + “**Bufferização**” no emissor e/ou receptor.
  + Aumento da utilização.



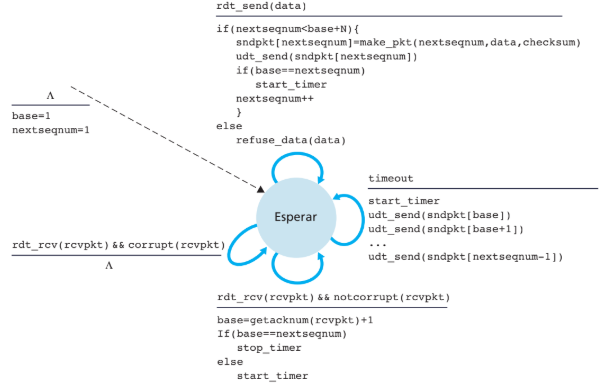
* Duas formas genéricas de protocolos pipeline : **go-Back-N (GBN)** e **Selective Repeat (SR)**.

### 3.4.3 Go-Back-N (GBN)

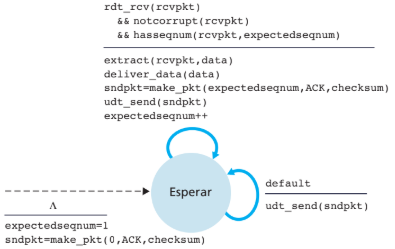
* Emissor :
  + Número de sequência de k bits no cabeçalho do pacote
  + “**Janela**” de até N pacotes consecutivos não confirmados permitida.



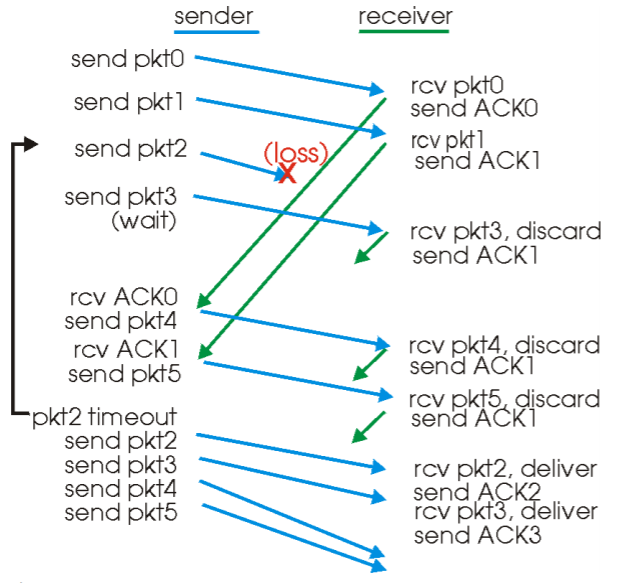
* + ACK(n) : confirma todos os pacotes até o n-ésimo (**ACK cumulativo**)
    - Pode receber ACKs duplicados.
  + Temporizador para **cada pacote** enviado.
  + **TimeOut(n)** : Retransmite **pacote n** e **todos os pacotes com números de sequência maiores** que estão dentro da janela.



* Receptor :
  + **ACK-only** : sempre envia ACK para pacote recebido “na ordem” corretamente com número de sequência mais elevado.
    - Pode gerar ACKs duplicados.
    - Precisa recordar somente o número de sequência esperado(expectedseqnum).
  + **Pacote fora de ordem** :
    - **Descarta** (não armazena), o receptor não possui **buffer**!
    - Manda **ACK** do pacote recebido de número de sequência mais elevado e em ordem.

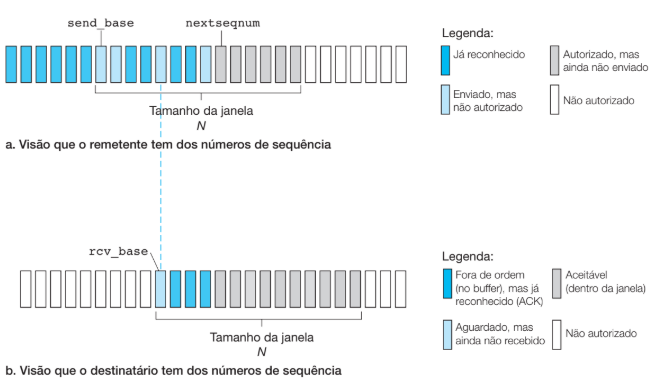
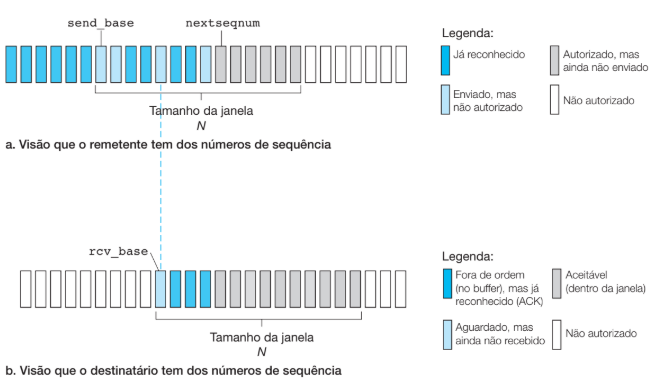


* Go-Back-N em operação

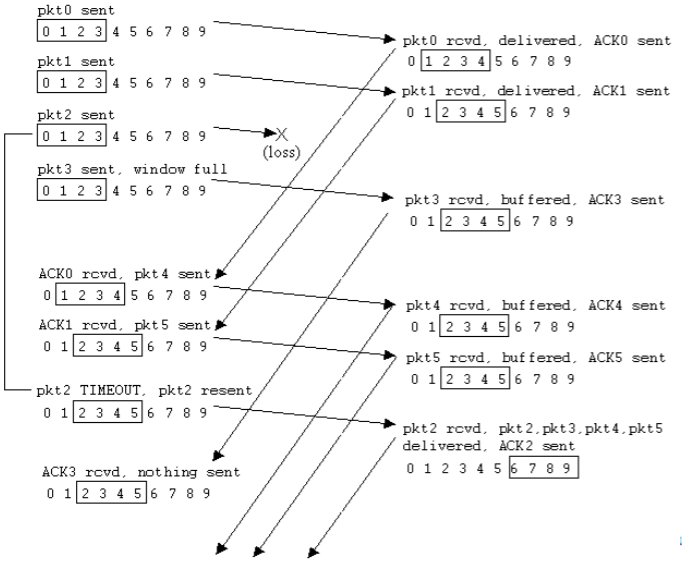


### 3.4.4 Repetição seletiva (SR)

* Receptor confirma **individualmente** cada pacote recebido corretamente.
  + **Armazena pacotes** quando necessário, para eventual envio de pacotes ordenados à camada superior.
* Emissor reenvia **somente** pacotes cujo ACK não foi recebido.
  + Emissor possui **temporizador** para cada pacote não confirmado.
* Janela do emissor :
  + N consecutivos números de sequência.
  + Novamente, limita números de sequência enviados, pacotes não confirmados.



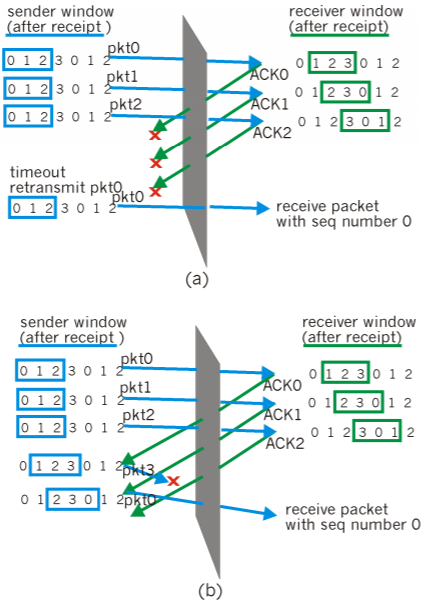
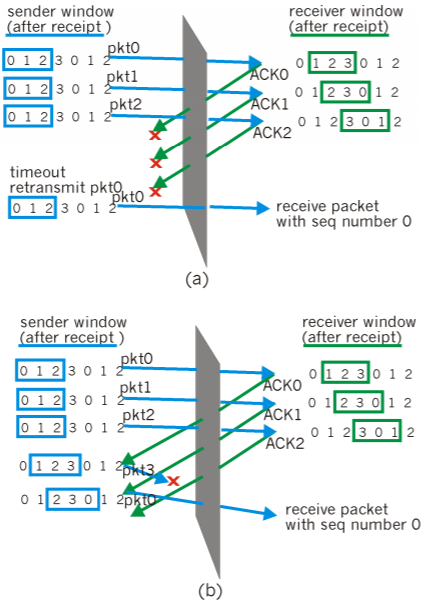
* Emissor :
  + **Dados da camada superior** :
    - Se o próximo número de sequência disponível estiver dentro da janela, envia pacote.
  + **TimeOut(n)** :
    - Reenvia pacote n e reinicia temporizador.
  + **ACK(n) no intervalo [sendbase, sendbase + n]** :
    - Marca pacote n como recebido.
    - Se n é o menor número de sequência de pacotes não confirmados, avança base da janela para o próximo número de sequência **não confirmado**.
* Receptor :
  + **Pacote n no intervalo[rcvbase, rcvbase + n - 1]** :
    - Envia ACK(n).
    - Fora de ordem : **Buffer**
    - Em ordem: **Entrega a camada superior** (incluindo pacotes ordenados no buffer), avança janela para o próximo pacote ainda não recebido.
  + **Pacote n no intervalo[rcvbase - n, rcvbase - 1]** :
    - Envia ACK(n).
  + **Caso contrário** : ignore.



* **O dilema do Selective repeat**

Exemplo :

* + Números de sequência : 0, 1, 2, 3
  + Janela de tamanho 3

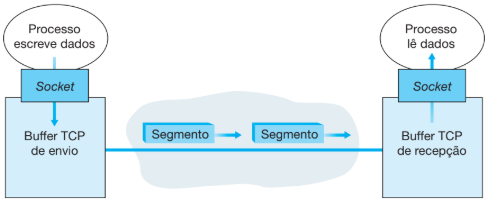


* + Receptor não vê diferença nos dois cenários!
  + Incorretamente passa dados duplicados como sendo novos dados, demonstrado no exemplo (a)
  + Para que esse problema não aconteça, a quantidade de números de sequência deve ser o dobro do tamanho da janela.

## **3.5 TCP : Transporte orientado à conexão**

### 3.5.1 A conexão TCP

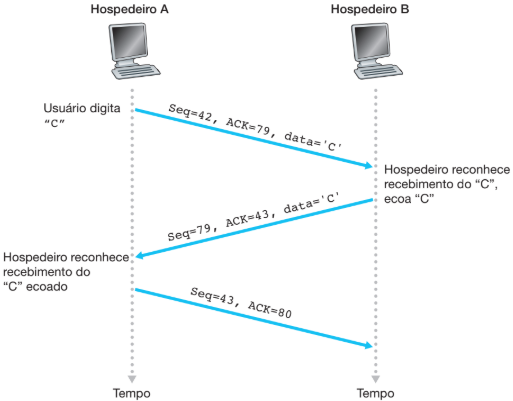
* **Ponto-a- Ponto :**
  + Apenas um emissor e um receptor
* **Stream de bytes confiável e em ordem :**
  + “Mensagens não possuem limites”
* **Pipelined :**
  + Controle de fluxo e congestionamento do TCP setam tamanho da janela.
* **Buffers no emissor e no receptor :**

****

* **Dados Full-Duplex :**
  + Fluxo de dados bidirecional sobre a mesma conexão
  + MSS: Maximum Segment Size
* **Orientado à conexão :**
  + **Handshaking** (troca de mensagens de controle) inicia emissor e estado do receptor antes da troca de dados
* **Fluxo controlado :**
  + Emissor não vai enviar mais dados que o receptor possa receber

### 3.5.2 Estrutura do segmento TCP

* **ACK**: Número de ACK válido
* **RST, SYN, FIN**: São usados para estabelecer e encerrar a conexão (comandos setup, teardown)
* **Soma de verificação**: CheckSum
* **Número de sequência, Número de reconhecimento**: Contagem por **bytes** de dados
* **Janela de recepção**: número de bytes o receptor está desejando receber
* **Campo de opções**: opcional e tem tamanho variável
* URG: Dados urgentes
* PSH: Passar dados imediatamente para a camada superior \*



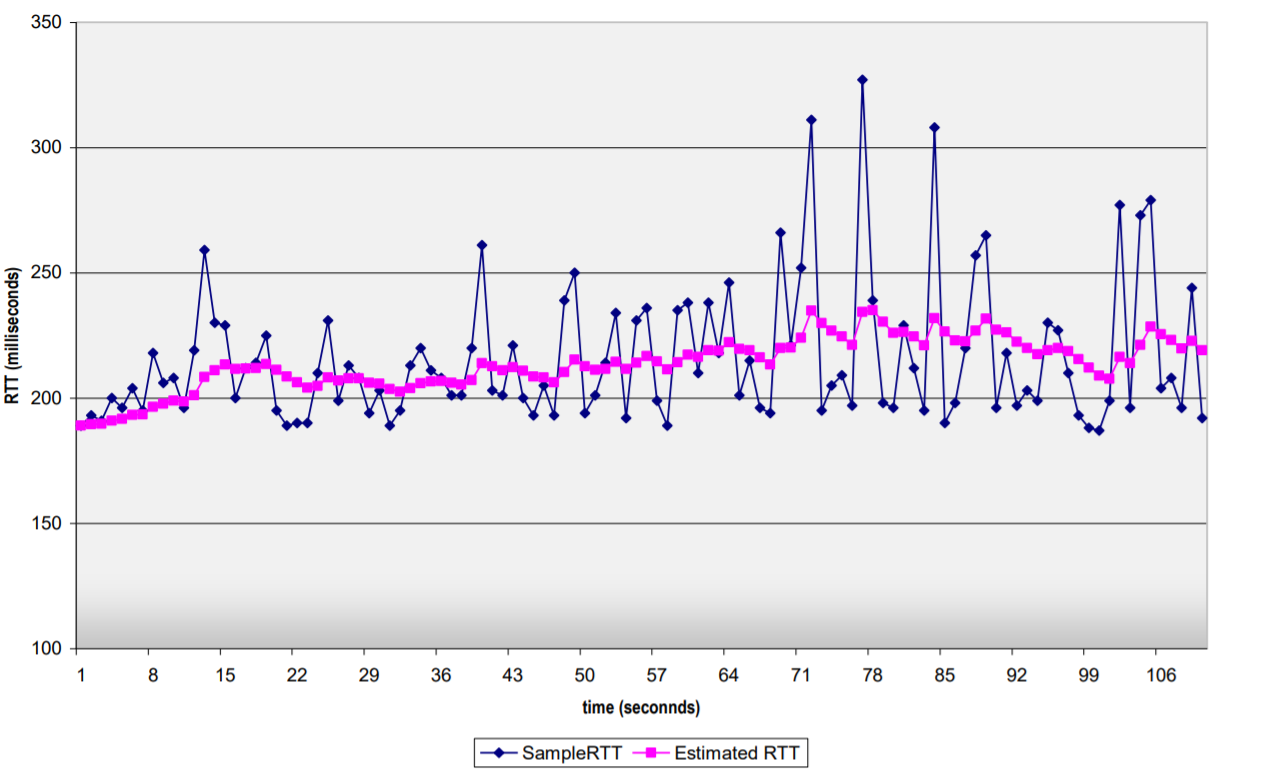
* Números de sequência:
  + Número da byte stream do primeiro byte nos dados do segmento
* ACKs:
  + Número de sequência do próximo byte aguardado pelo outro lado
  + **ACK cumulativo**
* A especificação TCP não diz como o receptor deve tratar de segmentos fora de ordem, é deixado para o desenvolvedor.

### 3.5.3 Estimativa de Round Trip Time (RTT) e TimeOut

* Como setar o valor do **TimeOut** do TCP ?
  + Deve ser **maior que o RTT**
    - Mas o RTT varia
  + Se for **muito longo**:
    - Reação lenta à perda de segmentos
  + Se for **muito curto**: TimeOut prematuro
    - Retransmissões desnecessárias
* Como estimar o RTT?
  + **SampleRTT** : Tempo medido a partir da transmissão do segmento até a recepção do ACK.
    - Ignora retransmissões
    - Variará, deseja-se fazer uma **estimativa** de RTT “suavizada”
      * Faz-se uma **média** de várias medidas recentes, não se usa somente a atual (SampleRTT)
* Exponential Weighted Moving Average (EWMA)

EstimatedRTT = (1-⍺)×EstimatedRTT + ⍺×SampleRTT

* + Valor típico : ⍺ = 0,125
* Influência de amostras do passado decresce exponencialmente
  + Peso para um dado SampleRTT decai exponencialmente rápido quando atualizações (novos SampleRTTs) são feitas
  + Maior peso para **amostras recentes**



##### **♦ Setando o TimeOut**

* EstimatedRTT + “margem segura”
  + Quanto maior a variação no EstimatedRTT, maior a margem de segurança
* Primeira estimativa de quanto o SampleRTT desvia do EstimatedRTT

DevRTT = (1-𝛽)×DevRTT + 𝛽×|SampleRTT-EstimatedRTT|

* + Tipicamente, 𝛽 = 0,25
* Em seguida, o intervalo de timeout é setado para:

TimeOutInterval = EstimatedRTT + **4**×DevRTT

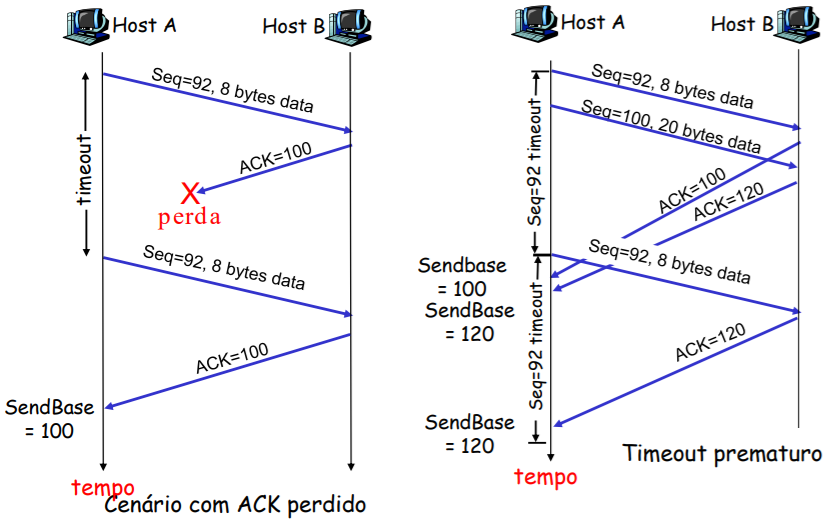
### 3.5.4 Transferência confiável de dados

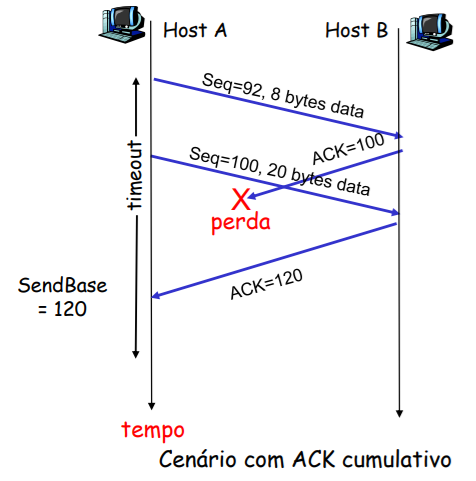
* TCP cria serviço de transferência confiável de dados no topo do serviço não-confiável do IP
* Segmentos “**Pipelined**”
* ACKs cumulativos
* *Conceitualmente* o TCP usa múltiplos temporizadores de transmissão
* **Retransmissões** são disparadas por:
  + Eventos de expiração (timeout)
  + ACKs duplicados
* Inicialmente considere um emissor TCP simplificado:
  + fluxo de dados unidirecional
  + ignore ACKs duplicados
  + ignore controle de fluxo
  + ignore controle de congestionamento

##### **♦ Eventos no emissor TCP**

* **Dados recebidos da aplicação** :
  + Cria segmento com número de sequência
  + Número de sequência é o número do primeiro byte de dados no segmento
  + Inicia temporizador se já não estiver executando
  + Intervalo de tempo de expiração : TimeOutInterval
* **TimeOut** :
  + Retransmite segmento que causou timeout
  + Reinicia temporizador
* **ACK recebido** :
  + Se confirma segmentos prévios não confirmados
    - Atualiza o que é conhecido as ser confirmado
    - Inicia temporizador se existem segmentos devidos

##### **♦ Cenários com retransmissão**



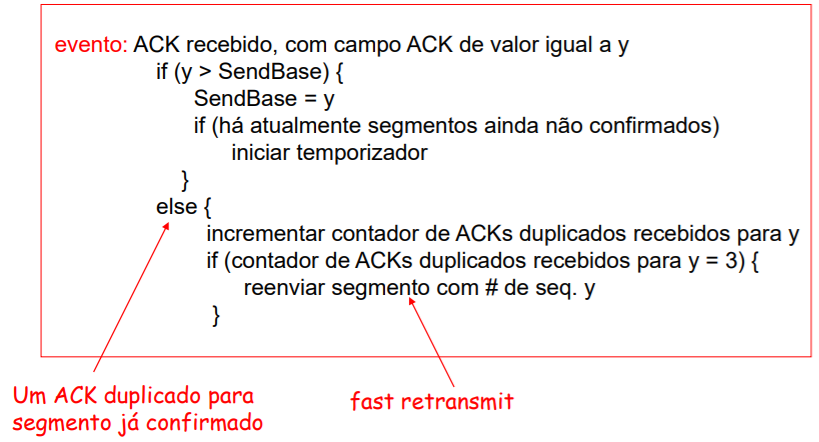


##### **♦ Geração de ACK no TCP** [recomendações na RFC 1122 e na RFC 2581]

|  |  |
| --- | --- |
| Evento no Receptor | Ação no Receptor TCP |
| * Chegada de segmento **na ordem** com número de sequência esperado. * Todos os dados até o número de sequência esperado já confirmados. * **Sem lacunas**. | * Atrasar ACK. * Aguarde até 500ms pelo próximo segmento. * Se não chegar nesse intervalo, envie ACK. |
| * Chegada de segmento **na ordem** com número de sequência esperado. * Um outro segmento em ordem aguardando transmissão do ACK | * Enviar imediatamente ACK único cumulativo, confirmando ambos segmentos na ordem |
| * Chegada de segmento **fora de ordem** com número de sequência maior que esperado. * **Lacuna detectada**. | * Enviar imediatamente **ACK (duplicado)**, indicando número de sequência do próximo bytes esperado (o começo da lacuna) |
| * Chegada de segmento que parcialmente ou totalmente preenche a lacuna | * Enviar imediatamente ACK, contanto que segmento o inicie no princípio da lacuna |

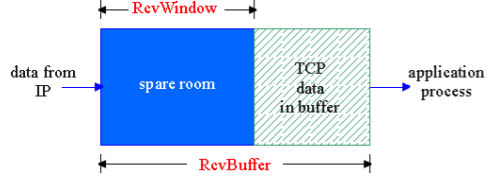
##### **♦ Fast Retransmit - Retransmissão rápida**

* Problema : Período de expiração **relativamente longo**:
  + Grande atraso antes de reenviar pacote perdido
* Solução : Detecta **segmentos perdidos** via ACKs duplicados.
  + Emissor frequentemente envia muitos segmentos sucessivos
  + Se o segmento é perdido, haverá provavelmente muitos ACKs duplicados
* Se o emissor recebe 3 ACKs duplicados (4 ACKs consecutivos, idênticos e sem outros pacotes no meio) para o mesmo dado, ele supõe que o segmento após dado confirmado foi perdido:
  + **Fast retransmit** : Reenvia segmento **antes** do temporizador expirar.



### 3.5.5 Controle de fluxo TCP

* Lado receptor da conexão TCP possui um buffer receptor:



* Processo aplicação pode ser lento ao ler dados do buffer
* Serviço de “casamento” de taxas:
  + “casar” a taxa de envio com a taxa de drenagem de dados da aplicação receptora

*(Suponha que o receptor TCP descarte segmentos fora de ordem)*

* Espaço livre no buffer
  + RcvWindow = RcvBuffer - [LastByteRcvd - LastByteRead]
* Receptor informa espaço livre através da inclusão do valor da janela de recepção RcvWindow nos segmentos.
* Emissor limita dados não confirmados ao tamanho da janela RcvWindow
  + garante que buffer do receptor não “transborde”

### 3.5.6 Gerenciamento da conexão TCP

***Lembre-se****: é estabelecida uma conexão entre o emissor e receptor TCP* ***antes*** *da troca de segmentos de dados*

* Inicializar variáveis TCP :
  + números de sequência
  + buffers, informação de controle de fluxo (por exemplo, RcvWindow)
* **Cliente**: iniciador da conexão

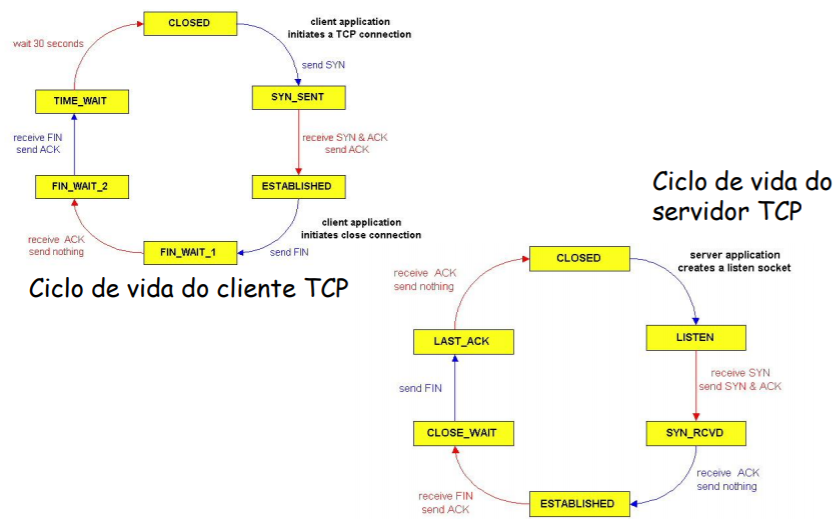
Socket clientSocket = new Socket("hostname","port number");

* **Servidor**: contactado pelo cliente

Socket connectionSocket = welcomeSocket.accept();

* **Three-way handshake**:
  + Passo 1: host cliente envia segmento **SYN** TCP para o servidor
    - específica número de sequência inicial
    - sem dados
  + Passo 2: host servidor recebe SYN, responde com segmento **SYNACK**
    - servidor aloca buffers
    - específica número de sequência inicial do servidor
  + Passo 3: cliente recebe SYNACK, responde com segmento **ACK** que pode conter dados
* **Fechando uma conexão**:
  + Passo 1: Host Cliente envia segmento **FIN** TCP para o servidor
  + Passo 2: Host Servidor recebe FIN, responde com segmento **ACK**
    - Fecha a conexão
    - Envia FIN
  + Passo 3: Cliente recebe FIN, responde com **ACK**
    - Entra na espera temporizada - responderá com ACK aos FINs recebidos
  + Passo 4: Servidor, recebe ACK. Conexão fechada.





## **3.6 Princípios de controle de congestionamento**

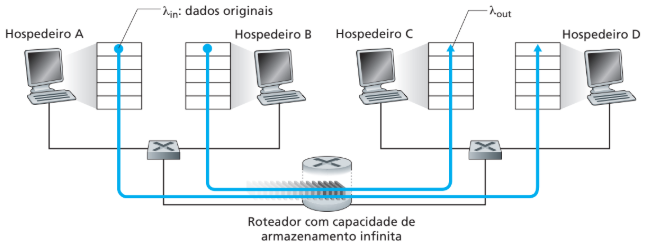
* Diferente de controle de fluxo!
* Informalmente:

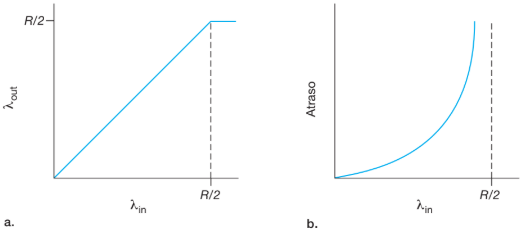
*“Fontes demais enviando dados demais e muito rápido, ultrapassando a capacidade da rede”*

* **Sintomas**:
  + Pacotes perdidos (“estouro” de buffer nos roteadores)
  + Atrasos elevados (“enfileiramento” em buffers nos roteadores)
* Na lista dos Top 10 problemas fundamentalmente importantes no trabalho em rede

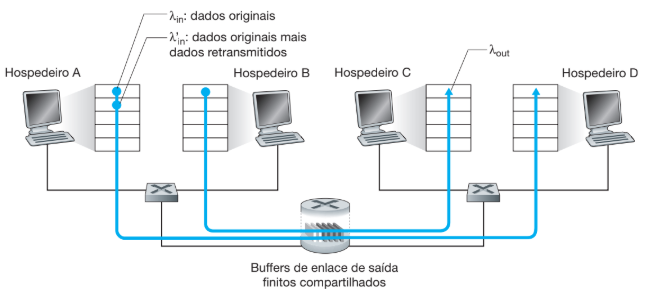
### 3.6.1 As causas e os custos do congestionamento

* Cenário 1
  + 2 emissores, 2 receptores
  + 1 roteador, buffers infinitos
  + Sem retransmissões
  + Enlace de capacidade R

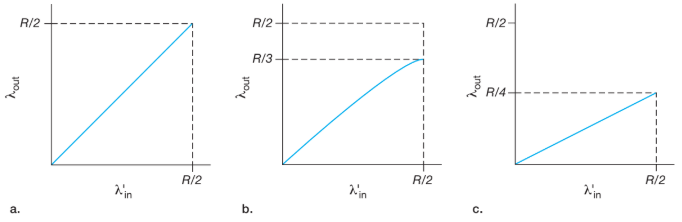




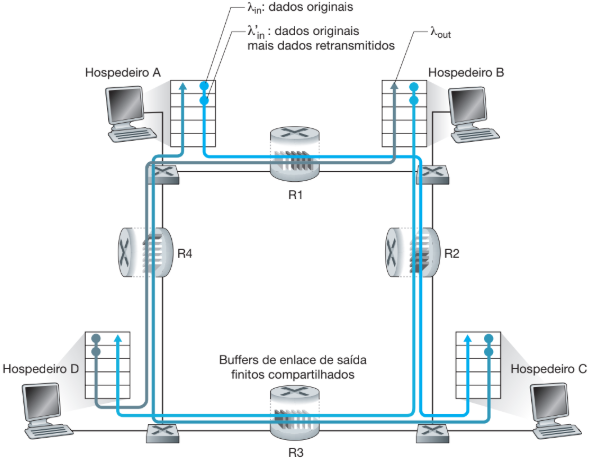
* Atraso muito elevado quando rede está congestionada
* Vazão (throughput) máxima alcançável
* Cenário 2
  + 1 roteador, buffers finitos
  + Emissor: retransmissão de pacote perdido
  + Enlace de capacidade R



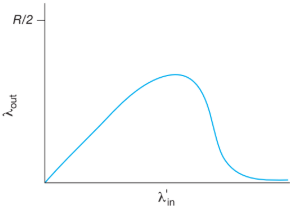
1. Ideal: Sempre ⇒ **λin** = **λ’in = λout** (goodput)
2. Mais realista: Retransmissões “perfeitas”, feitas somente no caso de perda: **λ’in > λout** \*
3. Mais+ realista: Retransmissão de pacotes atrasados (não perdidos) faz λ’in maior (que caso perfeito) para mesmo λout



* Custos do congestionamento:
  + (b) mais trabalho (retransmissões) para dada “goodput”
  + (c) Retransmissões desnecessárias: enlace carrega múltiplas cópias do mesmo pacote
* Cenário 3
  + 4 emissores
  + Caminhos com múltiplos saltos
  + TimeOut/Retransmissão
  + Enlace de capacidade R



* Um outro custo do congestionamento:
  + Quando o pacote é descartado, a capacidade de transmissão “upstream” usada para este pacote foi desperdiçada!



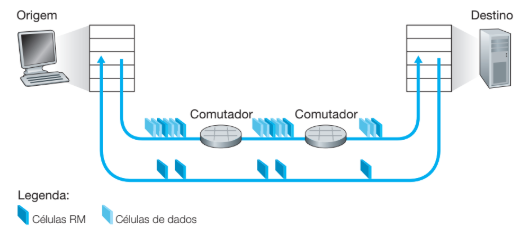
### 3.6.2 Mecanismos de controle de congestionamento

2 abordagens amplas para o controle de congestionamento

|  |  |
| --- | --- |
| **Controle de congestionamento fim-a-fim**: | **Controle de congestionamento assistido pela rede**  (Network-assisted congestion control): |
| * Sem feedback explícito da rede * Congestionamento inferido pelos end-systems através das perdas e atrasos observados * Abordagem usada pelo TCP | * Roteadores provêem feedback para os end-systems   + bit único indicando congestionamento (SNA, DECbit, TCP/IP ECN, ATM)   + Taxa explícita que emissor deve usar |

### 3.6.3 Exemplo de controle de congestionamento assistido pela rede: controle de congestionamento ABR da rede ATM

* ABR: available bit rate:
  + “serviço elástico”
  + Se caminho emissor-receptor estiver **“subutilizado”** (underloaded):
    - Emissor deve usar banda disponível
  + Se caminho emissor-receptor estiver **congestionado**:
    - Emissor reduz taxa para o menor valor garantido
* Células RM (resource management):
  + Enviadas pelo emissor, intercaladas com células de dados
  + bits na célula RM cell setados pelos comutadores (“network-assisted”)
    - **Bit NI**: nenhum incremento na taxa (mild congestion)
    - **Bit CI**: indicação de congestionamento
  + Receptor retorna células RM ao emissor com os bits inalterados
  + Campo **ER** *(explicit rate)* de 2 bytes no cabeçalho da célula RM
    - Comutador congestionado pode reduzir valor **ER** na célula
    - Taxa do emissor ajustada para a menor taxa suportada pelo caminho até o receptor
  + Bit **EFCI** em células de dados: setado para 1 em comutadores congestionados
    - Se célula de dados precedendo célula RM possui o bit **EFCI** setado, emissor seta bit CI na célula RM retornada



## **3.7 Controle de congestionamento no TCP**

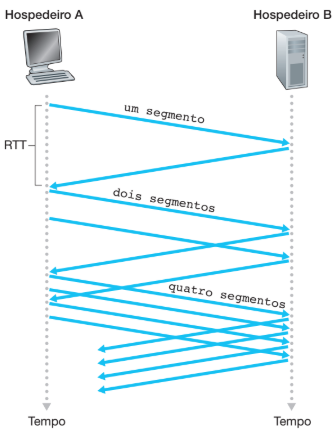
* Controle fim a fim (sem apoio da rede)
* Taxa de transmissão limitada pela tamanho da janela de congestionamento, Congwin:



* w segmentos, cada um com MSS bytes, enviados por RTT:

Bytes/sec

##### **♦ Partida lenta TCP**



* Aumento exponencial (por RTT) no tamanho da janela (não muito lenta)
* Evento de perda:
  + Temporizador (Tahoe TCP)
  + Três ACKs duplicados (Reno TCP)

|  |
| --- |
| Algoritmo Partida Lenta  inicializa: Congwin = 1  for (cada segmento com ACK)  Congwin++  until (evento de perda OR  CongWin > threshold) |

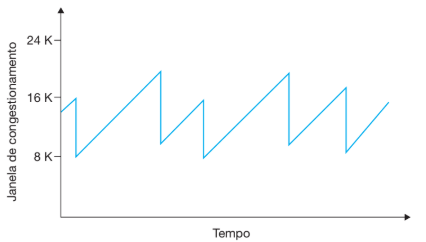
##### **♦ Prevenção de congestionamento**

|  |
| --- |
| /\* partida lenta acabou \*/  /\* Congwin > threshold \*/  Until (evento de perda) {  cada w segmentos reconhecidos:  Congwin++  }  threshold = Congwin/2  Congwin = 1  faça partida lenta |

##### **♦ Additive increase, multiplicative decrease (AIMD)**

Abordagem: aumentar taxa de transmissão (tamanho da janela), sondar banda passante ainda utilizável, até ocorrência de perda de pacote

* **Aumento aditivo**:
  + aumenta CongWin de 1 MSS a cada RTT até perda ser detectada
* **Redução multiplicativa**:
  + reduz CongWin pela metade após perda ser detectada
* Comportamento dente-de-serra: Sondagem de banda passante



##### 

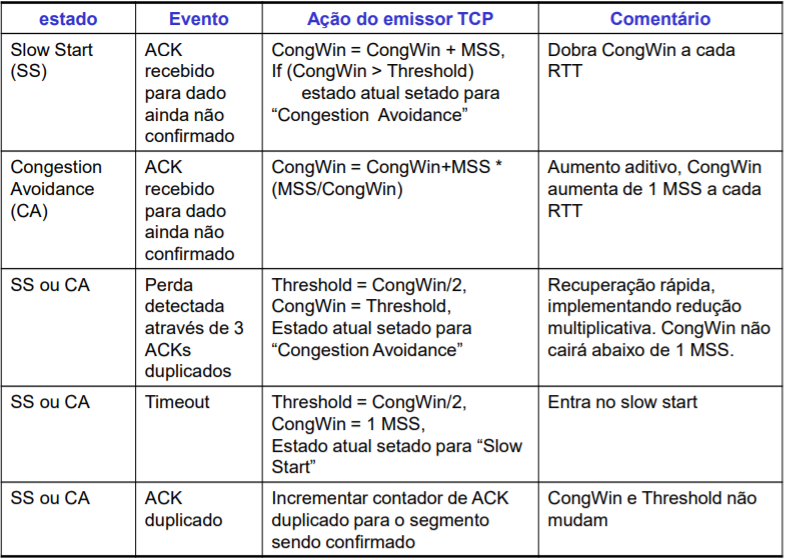
##### **♦ Refinamento**

O aumento exponencial deve parar de se tornar linear quando CongWin assume 0,5 do seu valor antes do TimeOut

* Implementação:
  + Threshold (limiar) variável
  + No evento de perda, Threshold é setado para 0,5 CongWin, valor da janela antes do evento de perda
* Inferindo perdas:
  + Após 3 ACKs duplicados:
    - CongWin é reduzida pela **metade**
    - Janela cresce **linearmente**
  + Mas após evento de timeout:
    - CongWin é setada para **1 MSS**
    - Janela então cresce **exponencialmente** até um threshold (limiar), e então volta a crescer **linearmente**

|  |
| --- |
| **Filosofia:**   * **3 ACKs duplicados** indica que a rede é capaz de entregar alguns segmentos * **TimeOut** indica um cenário “mais alarmante” de congestionamento |

|  |
| --- |
| Sumário : controle de congestionamento |
| * Quando CongWin está abaixo de Threshold, emissor está na fase **slow-start**, janela cresce **exponencialmente**. * Quando CongWin está acima de Threshold, emissor está na fase **congestion-avoidance**, janela cresce **linearmente**. * Quando três ACKs duplicados ocorrem, Threshold é setado para CongWin/2 e CongWin é setado para **Threshold**. * Quando TimeOut ocorre, Threshold é setado para CongWin/2 e CongWin é setado para **1 MSS**. |



##### **♦ Vazão do TCP**

*“Qual a vazão média do TCP em função do tamanho da janela e do RTT?”*

ignore partida lenta

* Seja W o tamanho da janela quando uma perda ocorre
* Quando janela é W, vazão é W/RTT
* Logo após a perda, janela cai à W/2 e a vazão à W/(2\*RTT)
* Vazão média : 0,75 W/RTT

##### **♦ Características do TCP**

Exemplo:

Segmentos de 1500 bytes, RTT = 100ms, deseja-se 10 Gbps de vazão

* Requer janela W = 83,333 segmentos sendo enviados
* Vazão em termos da taxa de perdas (L):

(1,22MSS)/RTT√L

* L = 210-10
* Necessárias novas versões do TCP para altas taxas de envio!

### 3.7.1 Equidade

ZZZzzzzzzzzz...😪😴