A Camada de Transporte

Objetivos :

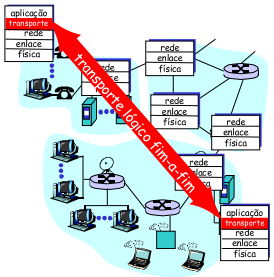
- entender os princípios por trás dos serviços da camada de transporte:
  - Multiplexação / demultiplexação
  - transferência de dados confiável
  - controle de fluxo
  - controle de congestionamento
- instanciação e implementação na Internet

Resumo:

- serviços da camada de transporte
- Multiplexação / demultiplexação
- transporte sem conexão: UDP
- princípios de transferência confiável de dados
- transporte orientado a conexão: TCP
  - transferência confiável
  - controle de fluxo
  - gerenciamento de conexão
- princípios de controle de congestionamento
- controle de congestionamento no TCP

Protocolos e Serviços de Transporte

- Fornecem comunicação lógica entre processos de aplicação rodando em hospedeiros diferentes
- Os protocolos de transporte são executados nos sistemas finais da rede



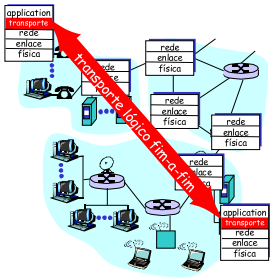
Serviço de Transporte vs Rede

- *camada de rede:*
  comunicação lógica entre hospedeiros
- *camada de transporte:*
  comunicação lógica entre processos
  - utiliza e aprimora os serviços oferecidos pela camada de rede

Protocolos da Camada de Transporte

Serviços de Transporte da Internet:

- entrega *unicast*, sequencial e confiável: TCP
  - estabelecimento de conexão
  - controle de fluxo
  - controle de congestionamento
- entrega *unicast* ou *multicast*, não confiável (*best-effort*) e não seqüencial: UDP
- não disponíveis:
  - garantia de retardo
  - garantia de banda
  - *multicast* confiável

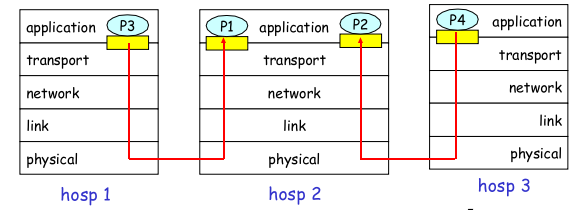


Multiplexação/demultiplexação

Demultiplex no receptor:
entrega dos segmentos recebidos ao soquete correto

Multiplex no emissor:
reunir dados de múltiplos soquetes; encapsular os dados com cabeçalho

= soquete = processo



Demultiplexação sem conexão

- Criar soquetes com números de portas:

```
DatagramSocket mySocket1 = new
   DatagramSocket(99111);
DatagramSocket mySocket2 = new
   DatagramSocket(99222);
```

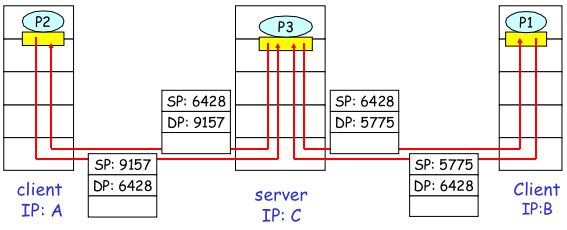
- Soquete UDP identificado pela 2-tupla:

(end IP dest, numero porta dest)

- Hospedeiro recebe um segmento UDP :
  - Verifica o número da porta de destino no segmento
  - Repassa o segmento para o soquete com esse número de porta
- Datagramas IP com endereços IP fonte e/ou números de portas fonte diferentes entregues ao mesmo soquete

Demultiplexação sem conexão (cont)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```

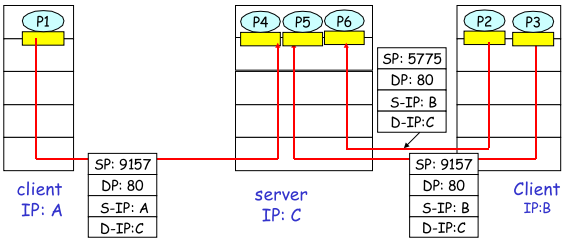


SP (Source Port) fornece o "endereço de retorno"

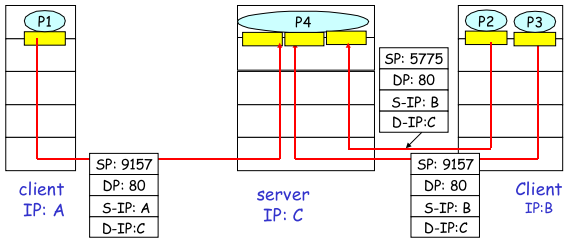
Demultiplexação orientada a conexão

- Soquete TCP identificado pela 4-tupla:
  - End. IP fonte
  - Num. porta fonte
  - End. IP destino
  - Num. porta destino
- Hospedeiro receptor usa os 4 valores para dirigir o segmento ao soquete apropriado
- Hospedeiros servidores podem ter vários soquetes TCP simultaneamente
  - Cada um identificado pela sua quadrupla
- Servidores Web usam soquetes diferentes para cada cliente conectado
  - HTTP não persistente terá um soquete diferente para cada requisição

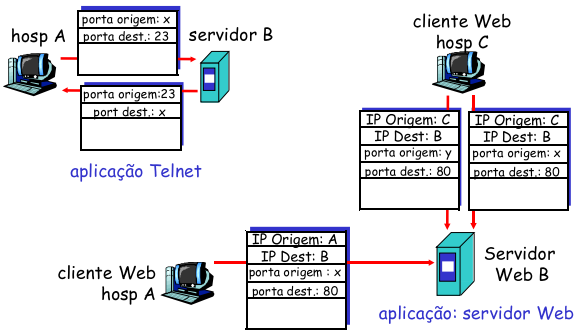
Demultiplexação orientada a conexão (cont)



Demultiplexação orientada a conexão : *Servidor com Threads*



Multiplexação /Demultiplexação : exemplos



UDP: *User Datagram Protocol* [RFC 768]

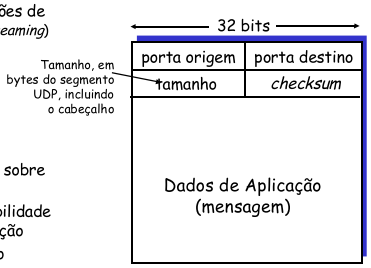
- protocolo de transporte da Internet "sem gorduras", "sem frescuras"
- serviço "melhor esforço", segmentos UDP podem ser:
  - perdidos
  - entregues fora de ordem
- *sem conexão:*
  - não há apresentação entre o UDP transmissor e o receptor
  - cada segmento UDP é tratado de forma independente dos outros

Porque existe um UDP?

- não há estabelecimento de conexão (que pode redundar em atrasos)
- simples: não há estado de conexão
- cabeçalho pequeno
- não há controle de congestionamento:
  - pode enviar segmentos tão rapidamente quanto desejado (e possível)

UDP: mais

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



formato do segmento UDP

UDP : *Checksum*

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

Transmissor:

- trata o conteúdo do segmento como uma seqüência de inteiros de 16 bits
- *checksum*: complemento de 1 da soma do conteúdo do segmento
  - colocado no campo de *checksum* do UDP

Receptor:

- computa o *checksum* do segmento recebido
- verifica se o *checksum* calculado é igual recebido:
  - NÃO - erro detectado
  - SIM - não há erros. *Mas, talvez haja erros apesar disto ...*

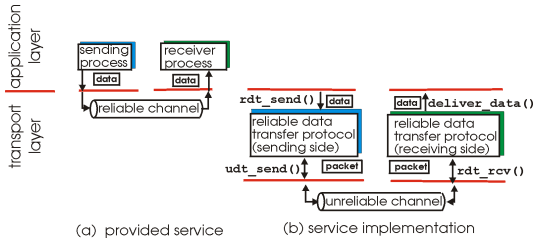
Checksum da Internet: Exemplo

- Nota
  - Ao adicionar números, um vai-um do bit mais significativo deve ser somando ao resultado
- Exemplo: soma de dois inteiros de 16-bits

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Vai-um	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
soma	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Transferência Confiável de Dados

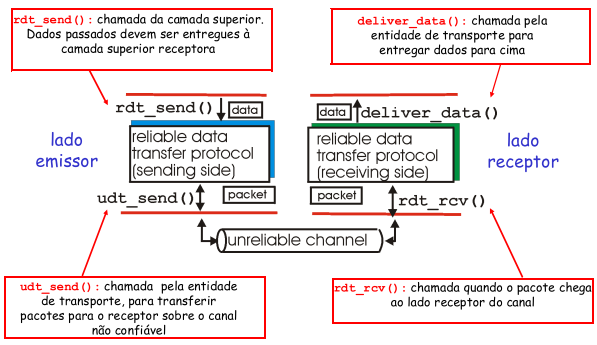
- importante nas camadas de aplicação, transporte e enlace
- um dos 10 tópicos mais importantes de redes!



(a) provided service (b) service implementation

as características dos canais não confiáveis determinarão a complexidade dos protocolos de transferência de dados confiável (*rdt*)

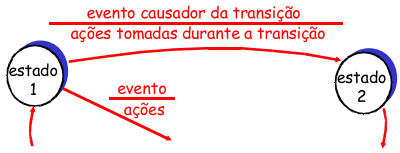
Transferência confiável: o ponto de partida



Transferência confiável: o ponto de partida

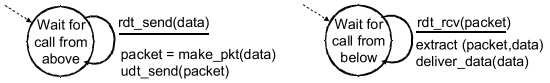
- desenvolvimento incremental dos lados transmissor e receptor de um protocolo de transferência de dados confiável (rdt)
- Considera-se apenas a transferência de dados unidirecional
  - entretanto informação de controle fluem em ambas direções!
- uso de máquinas de estados finitos (MEF) para especificar o transmissor e o receptor

estado: a partir de um "estado", o próximo estado será determinado unicamente pelo próximo evento



Rdt1.0: transferência confiável sobre canal confiável

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

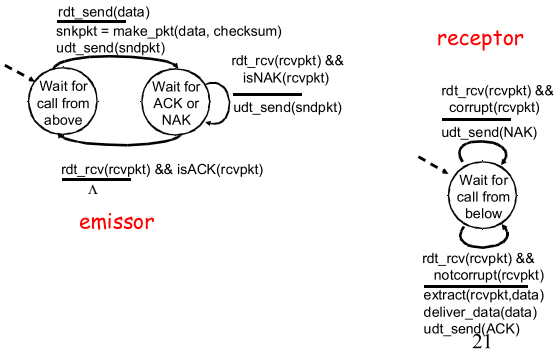


emissor receptor

Rdt2.0: canal com erros de bit

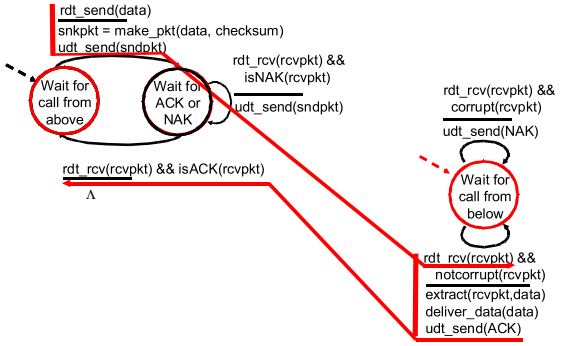
- canal subjacente pode trocar valores de bits de um pacote
  - *checksum* detecta erros de bits
- *a* questão: como se recuperar desses erros:
  - *reconhecimentos (ACKs):* receptor avisa explicitamente ao transmissor que o pacote foi recebido corretamente
  - *reconhecimentos negativos (NAKs):* receptor avisa explicitamente ao transmissor que o pacote tem erros
  - transmissor reenvia o pacote quando da recepção de um NAK
- novos mecanismos no `rdt2.0`:
  - detecção de erros
  - retorno do receptor: mensagens de controle (ACK,NAK) rcvr -> sender

rdt2.0: especificação da MEF

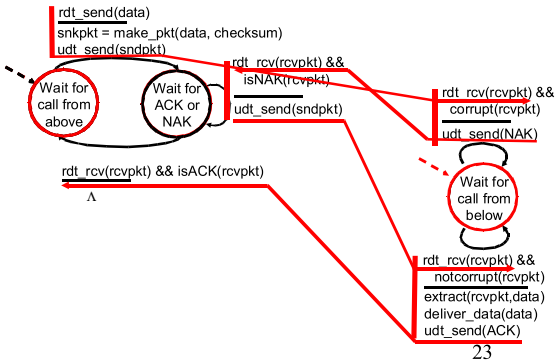


emissor

rdt2.0: operação sem de erros



rdt2.0: operação com erros



rdt2.0 tem um problema fatal!

O que acontece se o ACK/NAK é corrompido?

- emissor não sabe o que aconteceu no receptor!

O que fazer?

- retransmitir: pode gerar uma duplicata;

Tratamento de duplicatas:

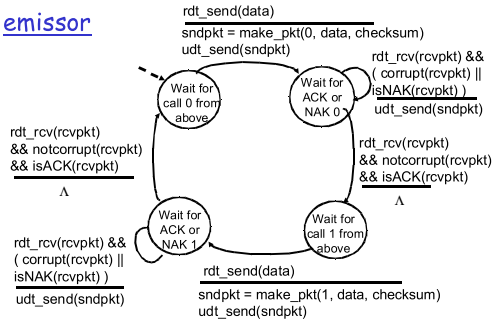
- emissor acrescenta *número de seqüência* em cada pacote
- emissor reenvia o pacote corrente se ACK/NAK for corrompido
- receptor descarta (não passa para a cima) pacotes duplicados

parar e esperar (stop and wait)

Emissor envia um pacote e então espera pela resposta do receptor

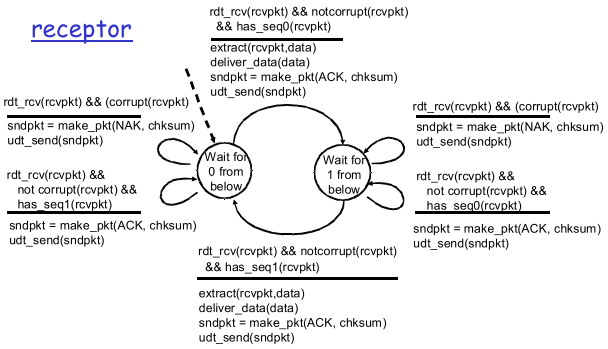
rdt2.1: trata ACK/NAKs corrompidos

emissor



rdt2.1: trata ACK/NAKs corrompidos

receptor



rdt2.1: discusssão

Emissor:

- adiciona número de seqüência ao pacote
- dois números de seqüência (0 e 1) bastam.
- deve verificar se os ACK/NAK recebidos estão corrompidos
- duplicação do número de estados (vs. rdt 2.0)
  - o estado deve "lembrar" se o pacote "corrente" tem número de seqüência 0 ou 1

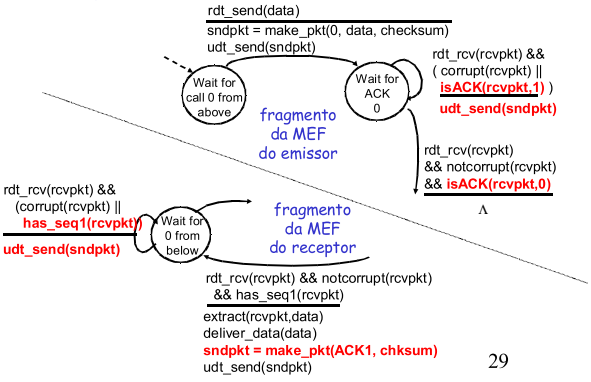
Receptor:

- deve verificar se o pacote recebido é duplicado
  - estado indica que pacote é esperado (0 ou 1)
- nota: receptor não tem como saber se seu último ACK/NAK foi recebido pelo emissor

rdt2.2: um protocolo sem NAK

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

rdt2.2: trechos do emissor e do receptor



rdt3.0: canais com erros e perdas

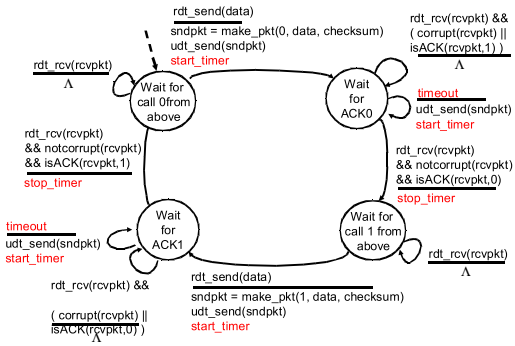
Nova Hipótese: canal de transmissão pode também perder pacotes (dados ou ACKs)

- *checksum*, números de seqüência, ACKs, retransmissões serão úteis, mas não suficientes.

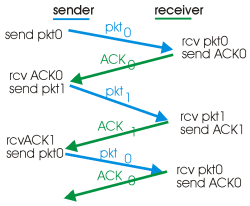
Abordagem: emissor espera um tempo "razoável" pelo ACK

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

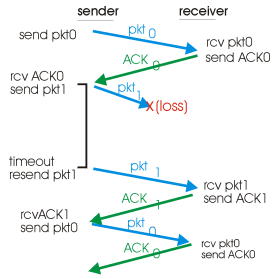
rdt3.0 emissor



rdt3.0 em ação

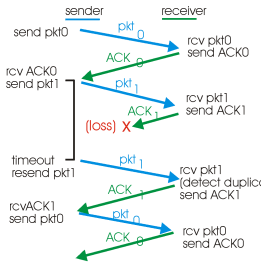


(a) operação sem perda

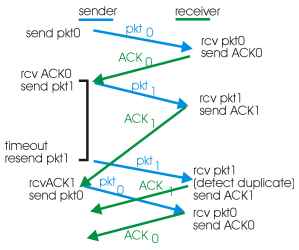


(b) pacote perdido

rdt3.0 em ação



(c) ACK perdido



(d) *timeout* prematuro

Desempenho do rdt3.0

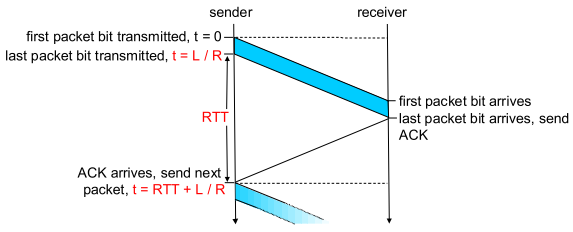
- rdt3.0 funciona, mas o seu desempenho é sofrível
- exemplo: enlace de 1 Gbps, 15 ms de atraso de propagação, pacotes de 1000 bytes:

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

$$U_{emissor} = \frac{L/R}{RTT+L/R} = \frac{0{,}008ms}{30{,}008ms} = 0{,}00027$$

  - Um pacote de 1000 bytes a cada 30,008 ms -> 267 kbps de vazão sobre um canal de 1 Gbps
  - o protocolo de rede limita o uso dos recursos físicos!

rdt3.0: funcionamento do "stop-and-wait"

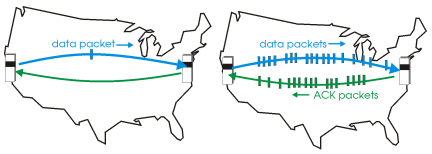


$$U_{emissor} = \frac{L/R}{RTT+L/R} = \frac{0{,}008ms}{30{,}008ms} = 0{,}00027$$

Protocolos com *Pipelining*/janela

Pipelining : o emissor permite vários pacotes "em trânsito" ainda não reconhecidos

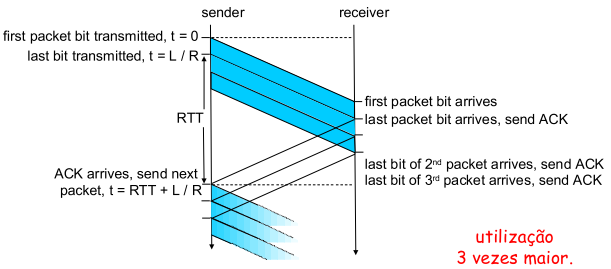
- faixa de números de seqüência deve ser aumentada
- armazenamento no transmissor e/ou no receptor



(a) operação do protocolo stop-and-wait **(b) operação do protocolo com *pipelining***

- Protocolos com *pipelining*: *go-back-N, retransmissão seletiva*

Pipelining: aumento da utilização



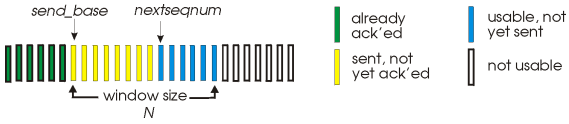
utilização 3 vezes maior.

$$U_{emissor} = \frac{3 * L/R}{RTT+L/R} = \frac{3 * 0{,}008ms}{30{,}008ms} = 0{,}0008$$

Go-Back-N (GBN)

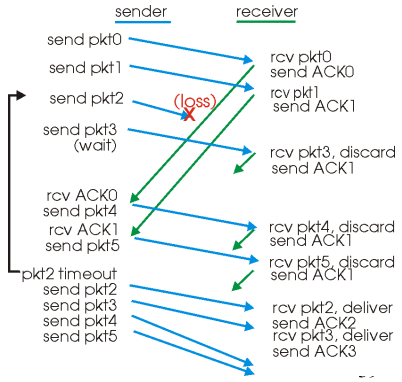
Emissor:

- Número de seqüência com k bits no cabeçalho do pacote
- "janela" de tamanho N:
  - N pacotes não reconhecidos consecutivos permitidos



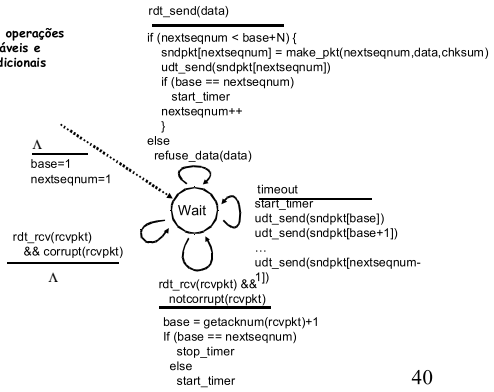
- ACK(n): reconhece todos os pacotes até o número de seqüência *n*, inclusive. "ACK cumulativo"
- *timeout(n):* retransmite o pacote *n* e todos os pacotes subsequentes que que estejam dentro da janela

GBN em ação

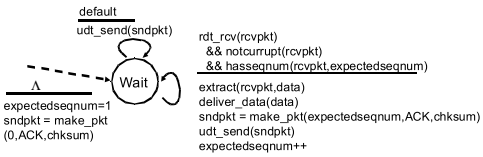


GBN: MEF estendida* do emissor

* variáveis, operações sobre variáveis e ações condicionais



GBN: MEF estendida do receptor

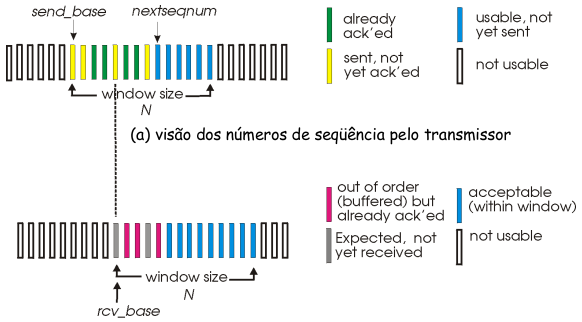


- sempre envia ACK para o pacote corretamente recebido com o maior número de seqüência *em ordem*
  - pode gerar ACKs duplicados
  - precisa lembrar apenas do número de seqüência esperado (`expectedseqnum`)
- pacotes fora de ordem:
  - descarta (não armazena) -> não há *buffer* de recepção!
  - Reconfirma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

Retransmissão Seletiva (RS)

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

Retransmissão seletiva: janelas do transmissor e do receptor



(a) visão dos números de seqüência pelo transmissor

(b) visão dos números de seqüência pelo receptor

Retransmissão seletiva

emissor

dados da camada superior :
- **se** próximo número de seqüência está na janela **então** envia o pacote **senão** armazena ou recusa os dados

timeout(n):
- reenvia pacote n
- reinicia o temporizador.

ACK(n) em [sendbase,sendbase+N]:
- marca pacote n como recebido
- **se** n = sendbase **então** avança a base da janela para o próximo número de seqüência não reconhecido

receptor

pacote n em [rcvbase, rcvbase+N-1]
- envia ACK(n)
- fora de ordem: armazena
- em ordem:
  - entrega todos os pacotes armazenados em ordem
  - avança base da janela para o próximo pacote ainda não recebido

pkt n em [rcvbase-N,rcvbase-1]
- ACK(n)

caso contrário:
- ignora

Retransmissão seletiva em ação

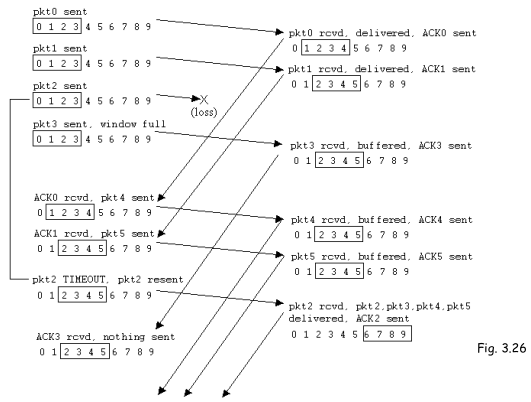


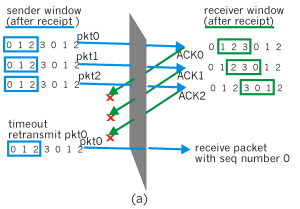
Fig. 3.26

Retransmissão seletiva: dilema

Exemplo:
- #s de seqüência: 0, 1, 2, 3
- tamanho da janela=3

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

Q: relação entre o tamanho do espaço de números de seqüência e o tamanho da janela?



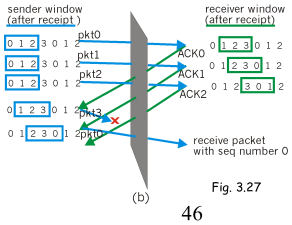


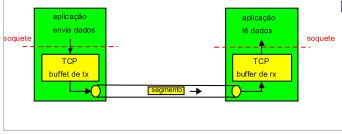
Fig. 3.27

Canal com memória

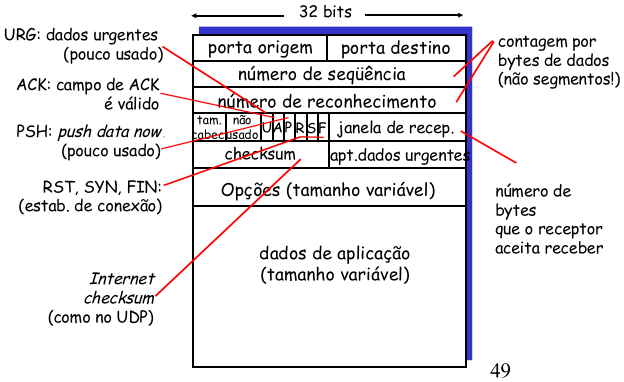
- Os protocolos apresentados pressupõem que o canal não reordena os pacotes, i.e., o canal não tem memória.
- Em canais com memória, um emissor não deve reutilizar um número de seqüência até ter certeza de que ele não está mais presente na rede.

TCP: Visão Geral RFCs: 793, 1122, 1323, 2018, 2581

- ponto-a-ponto:
  - um transmissor, um receptor
- fluxo de bytes, confiável, em ordem:
  - não há limites de mensagens
- *pipelined*:
  - Controle de congestão e de fluxo definem tamanho da janela
- *buffers de transmissão e de recepção*
- dados *full-duplex*:
  - transmissão bidirecional na mesma conexão
  - MSS: *maximum segment size*
    - quantidade máxima de dados da aplicação no segmento
- orientado a conexão:
  - *apresentação (handshaking)* - *troca de mensagens de controle:* inicia o estado do transmissor e do receptor antes da troca de dados
- controle de fluxo:
  - transmissor não esgota a capacidade do receptor



Estrutura do Segmento TCP

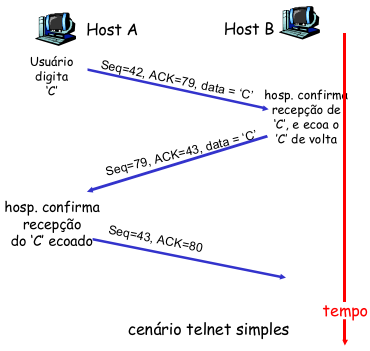


Números de Seqüência e ACKs do TCP

Números de seqüência:
- Número, no fluxo de dados, do primeiro byte no segmento de dados

ACKs:
- número do próximo byte esperado do outro lado
- ACK cumulativo

Q: como o receptor trata segmentos fora de ordem?
- A especificação do TCP não define, fica a critério do implementador



cenário telnet simples

Fig. 3.35

TCP: RTT e Temporização

Q: como escolher o valor de temporização do TCP?
- maior que o RTT
- muito curto: temporização prematura
  - retransmissões desnecessárias
- muito longo: reação lenta à perda de segmentos

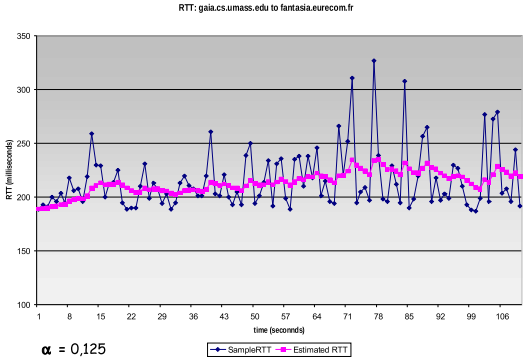
Q: Como estimar o RTT?
- `SampleRTT`: tempo medido desde a transmissão de um segmento até a recepção da respectiva confirmação
  - ignora retransmissões e segmentos reconhecidos de forma cumulativa
- `SampleRTT` varia de forma rápida, é desejável um amortecedor para a estimativa do RTT
  - usar várias medidas recentes, não apenas o último `SampleRTT`

TCP: RTT e Temporização

$$\texttt{EstimatedRTT} = (1-\alpha)*\texttt{EstimatedRTT} + \alpha*\texttt{SampleRTT}$$

- Média móvel com peso exponencial
- influência de uma amostra passada decresce de forma exponencial
- valor típico de α = 0,125 (isto é 1/8)

Amostras e Estimativas do RTT



α = 0,125

TCP: RTT e Temporização

Definindo o intervalo de temporização

- `EstimtedRTT` mais "margem de segurança"
- Primeiro estima-se de quanto `SampleRTT` se desvia de `EstimatedRTT`:

$$\texttt{DevRTT} = (1-\beta)*\texttt{DevRTT} + \beta*|\texttt{SampleRTT}-\texttt{EstimatedRTT}|$$

(tipicamente, β = 0,25)

- Então, define-se o intervalo de temporização:

$$\texttt{TimeoutInterval} = \texttt{EstimatedRTT} + 4*\texttt{DevRTT}$$

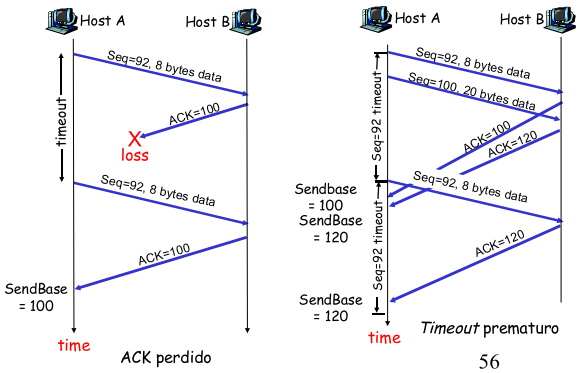
```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
   switch(event)
   event: data received from application above
        create TCP segment with sequence number NextSeqNum
        if (timer currently not running)
             start_timer()
        pass segment to IP
        NextSeqNum = NextSeqNum + length(data)
        break;
   event: timer timeout
        retransmit not-yet-acknowledged segment with
             smallest sequence number
        start_timer()
        break;
   event: ACK received, with ACK field value of y
        if (y > SendBase) {
             SendBase = y
             if (there are currently not-yet-acknowledged segments)
                  start_timer()
        }
        break;
} /* end of loop forever */
```

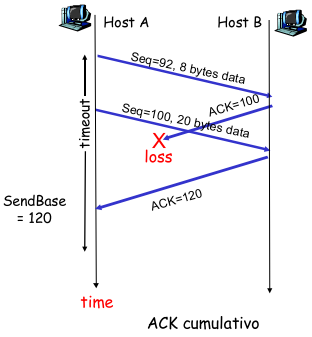
TCP: Transf. Confiável

Emissor (simplificado)

TCP: cenários de retransmissão



TCP: cenários de retransmissão (cont.)




```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
   switch(event)
   event: data received from application above
        create TCP segment with sequence number NextSeqNum
        if (timer currently not running)
             start_timer(to:=TimeouInterval)
        pass segment to IP
        NextSeqNum = NextSeqNum + length(data)
        break;
   event: timer timeout
        retransmit not-yet-acknowledged segment with
             smallest sequence number
        // forma limitada de controle de congestionamento
        start_timer(to := 2 * to)
        break;
   event: ACK received, with ACK field value of y
        if (y > SendBase) {
             SendBase = y
             if (there are currently not-yet-acknowledged segments)
                  start_timer(to := TimeoutInterval)
        }
        break;
} /* end of loop forever */
```

TCP: Transf. Confiável

Emissor (simplificado)

TCP: Geração de ACKs [RFC 1122, RFC 2581]

Evento no Receptor	Ação do TCP Receptor
segmento chega em ordem, não há lacunas, seg.s anteriores confirmados	ACK retardado. Espera até 500ms pelo próximo segmento. Se não chegar, envia ACK
segmento chega em ordem, não há lacunas, um ACK atrasado pendente	envia imediatamente um ACK cumulativo
segmento chega fora de ordem # de seq. maior que o esperado lacuna detectada	envia ACK repetido, indicando número de seqüência do próximo byte esperado
chegada de segmento que preenche uma lacuna parcial ou completamente	reconhece imediatamente, se o segmento começa na borda inferior da lacuna

Retransmissão Rápida [RFC 2581]

- Período de *timeout* em geral é relativamente longo:
  - Grande demora para o reenvio de pacotes perdidos
- Detecção de segmentos perdidos através de ACKs repetidos:
  - O emissor costuma mandar muitos segmentos um logo após o outro
  - Se um segmento é perdido, provavelmente cheguem muitos ACKs repetidos.
- SE o emissor receber 3 ACKs para o mesmo dado, ele supõe que o segmento após o dado confirmado foi perdido:
  - retransmissão rápida : reenvia o segmento antes que seu temporizador expire

Algoritmo de Retransmissão Rápida

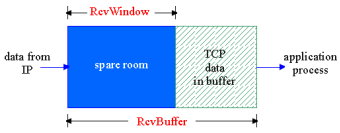
```
event: ACK received, with ACK field value of y
        if (y > SendBase) {
            SendBase = y
            if (there are currently not-yet-acknowledged segments)
                start timer
        }
        else {
            increment count of ACKs received for y
            if (count of ACKs received for y = 3) {
                resend segment with sequence number y
            }
```

ACK repetido para segmento já confirmado

Retransmissão rápida

TCP: Controle de Fluxo

- Lado receptor do TCP tem um *buffer* de recepção:



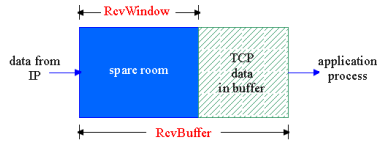
- Processos de aplicação podem ser lentos na leitura do *buffer*

controle de fluxo

transmissor não deve fazer transbordar o buffer de recepção enviando dados rápido demais

- Serviço de compatibilização das taxas de envio e recepção

TCP: Controle de Fluxo



LastByteRcvd – LastByteRead <= RcvBuffer

RcvWindow = RcvBuffer – (LastByteRcvd –LastByteRead)

receptor: informa dinamicamente o espaço livre incluindo **`RcvWindow`** nos segmentos

transmissor: limita a quantidade de dados transmitidos mas não reconhecidos a **`RcvWindow`**

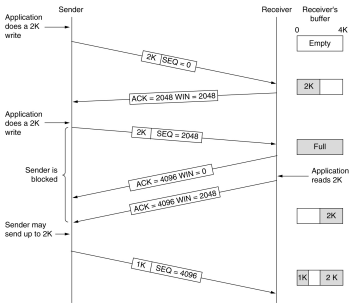
LastByteRcvd: número do último byte na cadeia de dados colocado no buffer

LastByteRead: número do último byte na cadeia de dados lido do buffer

TCP: Controle de Fluxo

Quando a janela for 0, o emissor pode mandar dados apenas nas seguintes situações

- dados urgentes
- segm/to de 1 byte de dados, para que o receptor anuncie a janela



TANENBAUM, A. S., Computer Networks, 4rd. Ed., Prentice-Hall, 2003. Capítulo 6.

TCP: Gerência de Conexão

Lembre-se: O TCP transmissor estabelece uma "conexão" com o receptor antes de trocar segmentos de dados

- inicia variáveis:
  - números de seqüência
  - *buffers*, info. de controle de fluxo (ex. **`RcvWindow`**)
- ***cliente:*** iniciador da conexão
- ***servidor:*** chamado pelo cliente

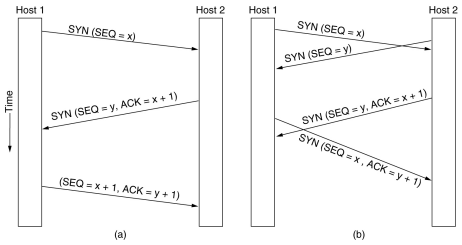
Apresentação em três vias

Passo 1: sistema final cliente envia um segmento TCP SYN ao servidor
- especifica número de seqüência inicial

Passo 2: o sistema final servidor recebe o SYN, responde com segmento SYNACK
- aloca *buffers*
- especifica o número de seqüência inicial do servidor

Passo 3: o cliente recebe SYNACK, responde com segmento ACK , que pode conter dados

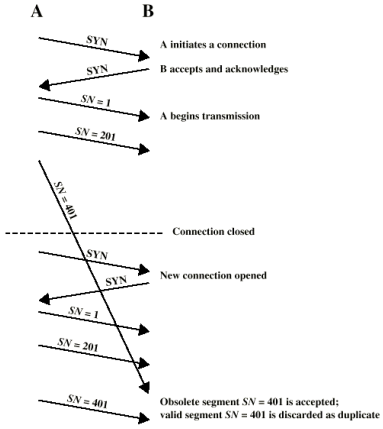
TCP: Estabelecimento de Conexão



(a) Situação normal; (b) Colisão de chamada

TANENBAUM, A. S., Computer Networks, 4rd. Ed., Prentice-Hall, 2003. Capítulo 6.

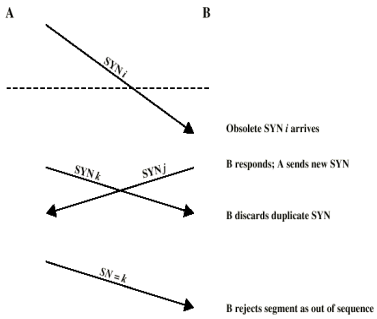
Apresentação em duas vias: Segmento de dados obsoleto



Obs. Numeração de segmentos não segue a convenção do TCP

STALLINGS, W. Data and Computer Communications. 6th ed. Prentice-Hall, 2000.

Apresentação em duas vias: Segmento SYN obsoleto

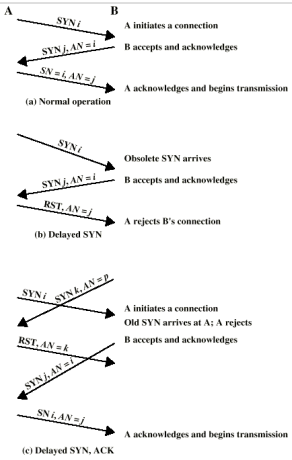


Obs. Numeração de segmentos não segue a convenção do TCP

STALLINGS, W. Data and Computer Communications. 6th ed. Prentice-Hall, 2000.

Apresentação em três vias:

Exemplos



Obs. Numeração de segmentos não segue a convenção do TCP

STALLINGS, W. Data and Computer Communications. 6th ed. Prentice-Hall, 2000.

TCP Término de Conexão

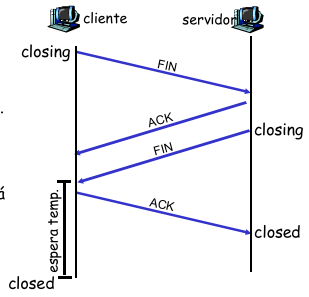
Passo 1: o cliente envia o segmento TCP FIN ao servidor

Passo 2: servidor recebe FIN, responde com ACK. Fecha a conexão, envia FIN.

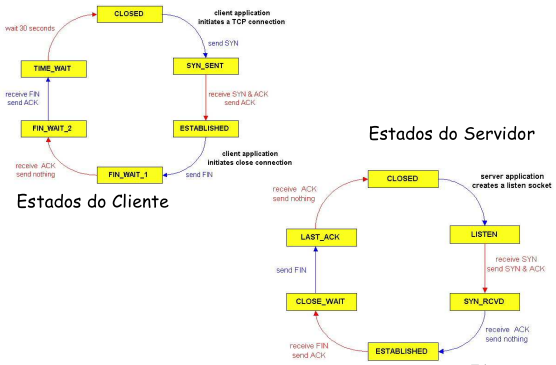
Passo 3: cliente recebe FIN, responde com ACK.

- Entra "espera temporizada" - responderá com ACK a FINs recebidos

Passo 4: servidor, recebe ACK. Conexão fechada.



TCP Controle de Conexão



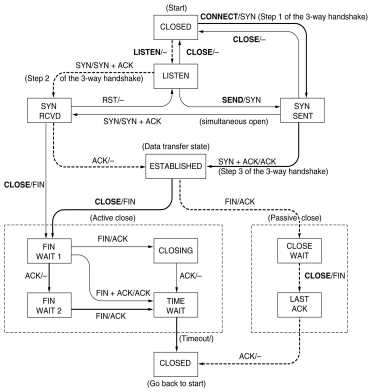
TCP: Modelando a Gerência de Conexão

State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIMED WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

TANENBAUM, A. S., Computer Networks, 4rd. Ed., Prentice-Hall, 2003. Capitulo 6.

TCP: Modelando a Gerência de Conexão

- Linha grossa contínua: caminho normal do cliente
- Linha grossa tracejada: caminho normal do servidor
- Linhas finas: eventos não usuais
- Transições são rotuladas pelos eventos que as causam e pela ação resultante, separada por uma barra



TANENBAUM, A. S., Computer Networks, 4rd. Ed., Prentice-Hall, 2003. Capítulo 6.

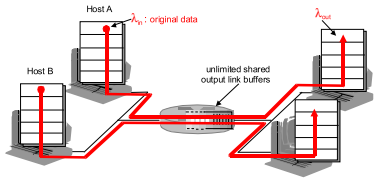
Princípios de Controle de Congestionamento

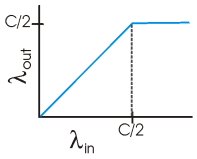
Congestionamento:

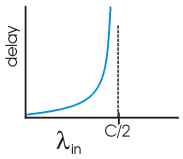
- informalmente: "muitas fontes enviando muitos dados muito rapidamente para a *rede* tratá-los"
- diferente de controle de fluxo!
- sintomas:
  - perda de pacotes (transbordamento de *buffers* nos roteadores)
  - atrasos elevados (filas nos *buffers* dos roteadores)
- um dos 10 problemas mais importantes em redes!

Causas/custos do congestionamento: cenário 1

- dois transmissores, dois receptores
- um roteador, *buffers* infinitos
- não há retransmissão



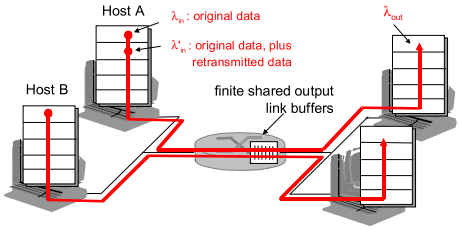




- Pode-se obter a vazão máxima
- Grandes atrasos quando congestionado

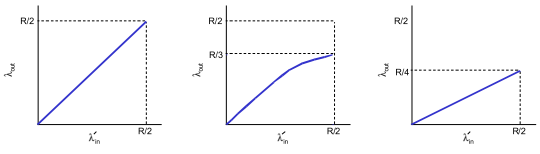
Causas/custos do congestionamento: cenário 2

- um roteador, buffers *finitos*
- transmissor reenvia pacotes perdidos



Causas/custos do congestionamento: cenário 2

- (a) A envia um pacote apenas quando há um *buffer* disponível (A é adivinho!)
- (b) retransmissão "perfeita" somente quando há perdas (*t.o.* grande o bastante):
- (c) retransmissão de pacotes atrasados (não perdidos)



a. b. c.

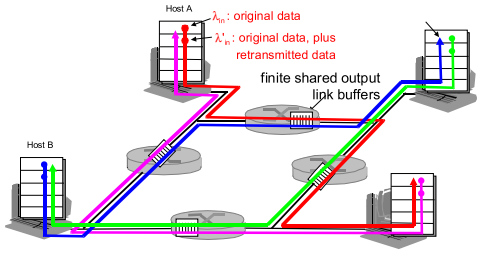
"custos" do congestionamento:

- mais trabalho (retransmissões) para um dado "tráfego bom"
- retransmissões desnecessárias: enlace transporta várias cópias do mesmo pacote

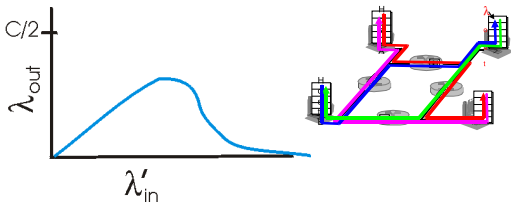
Causas/custos do congestionamento: cenário 3

- quatro transmissores
- caminhos com múltiplos saltos
- temporizações/retransmissões

Q: o que acontece quando λ_{in} e λ'_{in} aumentam ?



Causas/custos do congestionamento: cenário 3



Outro "custo" do congestionamento:

- quando um pacote é descartado, a capacidade de transmissão usada "corrente acima" para aquele pacote é desperdiçada!

Abordagens ao problema de controle de congestionamento

Duas abordagens gerais:

Controle de congestionamento fim-a-fim:

- não há realimentação explícita da rede
- congestionamento é inferido a partir das perdas e dos atrasos observados nos sistemas finais
- abordagem usada pelo TCP

Controle de congestionamento assistido pela rede:

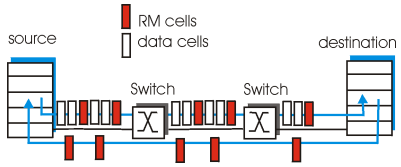
- roteadores fornecem informações aos sistemas finais
  - bit indicando o congestionamento (SNA, DECbit, TCP/IP ECN, ATM)
  - taxa explícita a que o transmissor pode enviar

Estudo de caso: controle de congestionamento no serviço ATM ABR

ABR: *available bit rate*:

- "serviço elástico"
- se o caminho do transmissor está pouco usado:
  - transmissor pode usar a banda disponível
- se o caminho do transmissor está congestionado:
  - transmissor é limitado a uma taxa mínima garantida

Controle de congestionamento no ATM ABR



Células RM (*resource management*):

- bits nas células RM são modificados pelos comutadores
  - NI bit: não aumentar a taxa (congestionamento leve)
  - CI bit: indicação de congestionamento
- células RM são devolvidos pelo receptor, com os bits de indicação intactos
- comutadores tb. podem gerar e enviar células RM diretamente a uma fonte
- campo ER (*explicit rate*) de dois bytes nas células RM
  - comutador congestionado pode reduzir o valor de ER nas células
  - o transmissor envia dados de acordo com a vazão mínima no caminho
- bit EFCI nas células de dados: marcado pelos comutadores congestionados
  - se a célula de dados que precede uma célula RM tem o bit EFCI marcado, o destino marca o bit CI da célula RM devolvida à fonte

TCP: Controle de Congestionamento

- Controle fim-a-fim (sem assistência da rede)
- Emissor limita a transmissão:
  `LastByteSent-LastByteAcked`
  `≤ min{CongWin, RcvWindow}`
- Aproximadamente,

$$\text{taxa} = \frac{\text{CongWin}}{\text{RTT}} \text{ bytes/s}$$

- `CongWin` é dinâmica, função do congestionamento de rede percebido

Como o emissor percebe o congestionamento?

- perda = *timeout ou* 3 acks duplicados (Reno)
- Emissor TCP reduz a taxa (`CongWin`) após uma perda

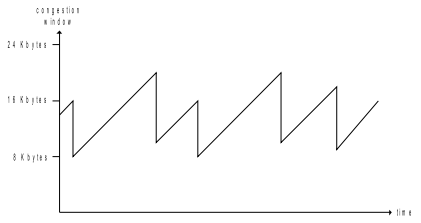
Três mecanismos:

- AIMD
- Partida lenta
- Conservadorismo após *timeouts*

TCP: Controle de Congestionamento AIMD

Aumento aditivo: incrementar `CongWin` de 1 MSS a cada RTT na ausência de perdas: *testando*

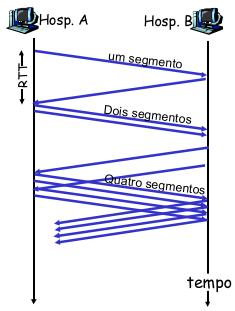
Diminuição multiplicativa: cortar `CongWin` pela metade após uma perda



Conexão TCP ao longo do tempo

TCP: Controle de Congestionamento Partida Lenta (*Slow Start*)

- Quando uma conexão começa, **CongWin** = 1 MSS
  - Exemplo: MSS = 500 bytes, RTT = 200 ms
  - Taxa inicial = 20 kbps
- Largura de banda disponível pode ser >> MSS/RTT
  - Desejável subir rapidamente para uma taxa respeitável
- Quando a conexão começa, incrementar a taxa exponencialmente até o primeiro evento de perda:
  - Dobrar **CongWin** a cada RTT
  - Feito incrementando **CongWin** para cada ACK recebido
- Resumo: taxa inicial é baixa, mas cresce exponencialmente



TCP: Controle de Congestionamento Conservadorismo após *timeouts*

- Após 3 ACKs duplicados :
  - **CongWin** é cortada pela metade
  - janela cresce então linearmente
- Mas, após um *timeout*:
  - **CongWin** = 1 MSS;
  - janela cresce então exponencialmente ...
  - ... até um patamar, então cresce linearmente

Motivos:

- 3 ACKs duplicados indicam que a rede é capaz de entregar alguns segmentos
- *timeout* antes de 3 ACKs duplicados é "mais alarmante"

TCP: Controle de Congestionamento Síntese

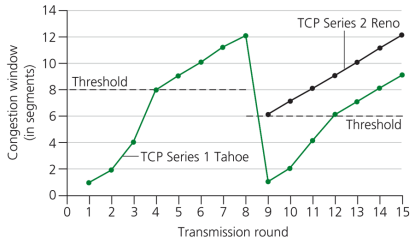


Figure 3.51 ♦ Evolution of TCP's congestion window (Tahoe and Reno)

Resumo: Controle de Congestionamentos no TCP

- Quando **CongWin** está abaixo de **Threshold**, o emissor está na fase de partida lenta, e a janela cresce exponencialmente.
- Quando **CongWin** está acima de **Threshold**, o emissor está na fase de prevenção de congestionamento, e a janela cresce linearmente.
- Quando ocorre um triplo ACK duplicado, **Threshold** = **CongWin**/2 **CongWin** = **Threshold**.
- Quando ocorre um *timeout*, **Threshold** = **CongWin**/2 **CongWin** = 1 MSS.

TCP: controle de congestionamento no emissor

Estado	Evento	Ação do TCP Emissor	Commentário
Slow Start (SS)	ACK para dados ainda não reconhecidos	CongWin = CongWin + MSS If (CongWin > Threshold) State = CA	Resulta em dobrar CongWin a cada RTT
Congestion Avoidance (CA)	ACK para dados ainda não reconhecidos	CongWin = CongWin+MSS * (MSS/CongWin)	Aumento aditivo, resultando no aumento de CongWin de 1 MSS por RTT
SS or CA	ACK triplicado	Threshold = CongWin/2, CongWin = Threshold, State = CA	Recuperação rápida, implementação da diminuição multiplicativa.
SS or CA	*Timeout*	Threshold = CongWin/2, CongWin = 1 MSS, State = SS	Entra na partida lenta
SS or CA	ACK duplicado	Incrementar o contador de duplicatas do segmento confirmado	CongWin e Threshold não são modificados

TCP: vazão (*throughput*)

- Qual é a vazão média do TCP como função do tamanho da janela e do RTT?
  - Ignore a partida lenta
- Seja W o tamanho da janela, em bytes, quando ocorre uma perda
  - vazão é W/RTT
- Logo após a perda, a janela cai para W/2
  - vazão passa aa W/(2 · RTT)
- Vazão aumenta linearmente entre W/(2 · RTT) e W/ RTT (o processo repete-se continuamente)
  - Vazão média de uma conexão = 0,75 W/RTT

TCP: O Futuro

- Exemplo:
  segmentos de 1500 bytes, RTT de 100ms, deseja-se vazão de 10 Gbps

  Vazão média = 0,75 W/RTT

  $10 \cdot 10^9 = 0{,}75\ [(1500 \cdot 8) \cdot S]/100 \cdot 10^{-3}$
  $S \sim 112.000$

  Precisamos de uma janela de ~112.000 segmentos (em trânsito)

- Vazão média em termos da taxa de perda:

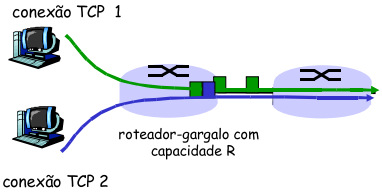
  Vazão média = $\frac{1{,}22 \cdot MSS}{RTT\sqrt{L}}$

- Para alcançar a vazão de 10 Gbps precisamos de $L = 2 \cdot 10^{-10}$
  (um evento de perda a cada 5.000.000.000 de segmentos)

- Necessárias novas versões do TCP para redes de alta velocidade!

TCP: Eqüidade

Objetivo : se K sessões TCP compartilham o mesmo enlace-gargalo com largura de banda R, cada uma deve receber uma taxa média de R/K

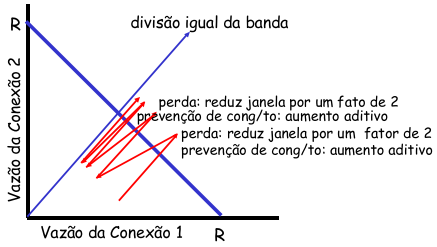


- Supomos que ambas conexões têm o mesmo MSS e RTT (mesma vazão para janelas iguais)
- Ignoramos a partida lenta.

Porque o TCP é justo?

Duas sessões competindo pela banda:

- O aumento aditivo fornece uma inclinação de 1, quando a vazão aumenta
- redução multiplicativa diminui a vazão proporcionalmente



TCP: Eqüidade (...)

Eqüidade e UDP

- Apl. multimídia geralmente não usam o TCP
  - não querem que a taxa seja reduzida pelo controle de congestionamento
- ... usam UDP:
  - Enviam áudio/vídeo a uma taxa constante
  - Toleram a perda de pacotes
- Área de pesquisa: controle de congestionamento X UDP

Eqüidade e Conexões TCP paralelas

- Nada impede as apl.s de abrirem conexões paralelas entre 2 hospedeiros
- Navegadores da Web fazem isso
- Exemplo: enlace de taxa R com 9 conexões;
  - Nova apl. abre 1 conexão TCP → recebe R/10
  - Nova apl. abre 9 conexões TCP → recebe R/2

A Camada de Transporte : Resumo

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