unesp UNIVERSIDADE ESTADUAL PAULISTA

REDES DE COMPUTADORES

Slides e Notas de Aula

Capítulo 3
Camada de Transporte

---

http://www.acmesecurity.org/redes

Adriano Mauro Cansian
adriano@ieee.org

São José do Rio Preto
Bacharelado em Ciências da Computação

2003

Créditos

Este material é uma coleção de slides para notas de aula do **Curso de Redes de Computadores** para o primeiro semestre de 2003, sob responsabilidade do Professor Adriano Mauro Cansian, na UNESP – Universidade Estadual Paulista, Campus de São José do Rio Preto. Este material **não substitui o livro texto adotado no curso** (vide as informações sobre bibliografia em http://www.acmesecurity.org/redes), devendo ser usado de modo a complementa-lo, e em conjunto com outras referências recomendadas. A principal função destas notas de aula é facilitar a realização das anotações dos tópicos mais importantes discutidos em sala de aula, agilizando assim o andamento do curso para os alunos.

Estes slides para notas de aula são, em sua grande parte, baseadas nos livros textos adotados para o curso no ano de 2003: "***Computer Networking: A Top-Down Approach Featuring the Internet***" – James F. Kurose & Keith W. Ross (ISBN: 0201477114), Editora Addison Wesley Longman, e *"**Computer Networks**", 3th Edition*, de Andrew S. Tanenbaum – Prentice-Hall (ISBN 0-13-349945-6) e em sua tradução "Redes de Computadores" – Terceira Edição, da Editora Campus (ISBN 8535201572). Para informações mais detalhadas sobre a bibliografia do curso, bem como outras referências utilizadas, consulte http://www.acmesecurity.org/redes/bibliografia.html.

Contato:

Adriano Mauro Cansian
Professor Assistente Doutor

adriano@acmesecurity.org / adriano@unesp.br

UNESP - Universidade Estadual Paulista
Campus de São José do Rio Preto

Depto. de Ciência da Computação e Estatística
Laboratório ACME! de Pesquisa em Segurança de Computadores e Redes

Endereço:
R. Cristóvão Colombo, 2265 - Jd. Nazareth
15055-000 * São José do Rio Preto, SP.
Tel. (17) 221-2475 (laboratório) / 221-2201 (secretaria)
http://www.acmesecurity.org/~adriano

Chave PGP:

Adriano Mauro Cansian <adriano@unesp.br>
Key ID: **0x3893CD2B**
Key Type: DH/DSS
Key Fingerprint: **C499 85ED 355E 774E 1709 524A B834 B139 3893 CD2B**

“

Lakes that endlessly outspread
Their lone waters – lone and dead, –
Their still waters – still and chilly
With the snows of the lolling lily.

”

Trecho de “*Dreamland* ”, Edgar Alan Poe - 1844.

Curso de Redes de Computadores

Adriano Mauro Cansian
adriano@ieee.org

Capítulo 3
Camada de Transporte

Capítulo 3: Camada de Transporte

Metas do capítulo:

- Compreender os princípios por trás dos serviços da camada de transporte:
  - Multiplexação/ desmultiplexação.
  - Transferência confiável de dados.
  - Controle de fluxo.
  - Controle de congestionamento.
- Instanciação e implementação na Internet.

O que veremos:

- Serviços da camada de transporte.
- Multiplexação/desmultiplexação.
- Transporte sem conexão: UDP.
- Princípios de transferência confiável de dados.
- Transporte orientado a conexão: TCP
  - Transferência confiável
  - Controle de fluxo
  - Gerenciamento de conexões
- Princípios de controle de congestionamento.
- Controle de congestionamento em TCP.

Comparação entre as camadas

- Camada de transporte repousa exatamente a camada de rede.
- Protocolo da **Camada de Transporte** fornece **comunicação lógica entre processos**, rodando em *hosts* diferentes.
- Protocolo da **Camada de Rede** fornece **comunicação lógica entre *hosts***.

Esta distinção é sutil, mas MUITO importante!

Camada de Transporte X Camada de Rede (1)

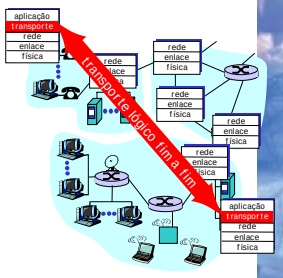
- Considere 12 irmãos numa casa em São Paulo.
- E outros 12 irmãos em outra casa em Boa Vista no Acre.
- Os de SP são primos daqueles em Boa Vista, e escrevem uns para os outros semanalmente.
- Em São Paulo, **João** recolhe as cartas dos irmãos e as entrega ao correio.
- No Acre, **Maria** recolhe as cartas dos irmãos, e as entrega ao correio.
- Ambos também fazem a distribuição local do que chega.

Camada de Transporte X Camada de Rede (2)

- *Hosts* (*end systems*) = Casas.
- Processos = os primos que trocam mensagens.
- Mensagens da aplicação = cartas em envelopes.
- **Protocolo da camada de rede** = serviço postal (correio).
- **Protocolo da Camada de Transporte** = João e Maria.

Serviços e protocolos de **TRANSPORTE (1)**

- Provê *comunicação lógica* entre processos de aplicação executando em hospedeiros diferentes.
- Protocolos de transporte **executam em sistemas finais.**
- Serviços das camadas de transporte versus rede:
  - *Camada de rede:* dados transferidos entre **sistemas**.
  - *Camada de transporte:* dados transferidos entre **processos**.



unesp - IBILCE - SJRP

Serviços e protocolos de TRANSPORTE (2)

- Do ponto de vista da **APLICAÇÃO** a camada de transporte permite enxergar os sistemas **como se eles estivessem fisicamente conectados**.
  - Mesmo que existam vários roteadores, links e outros equipamentos no caminho.
- A Camada de Aplicação não tem que se preocupar com os detalhes físicos da **infra-estrutura de interligação**, usada para carregar as mensagens.

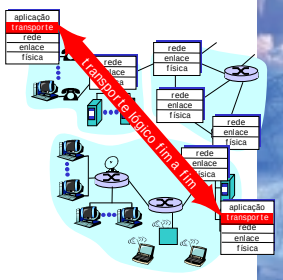
7

unesp - IBILCE - SJRP

Protocolos da camada de transporte (1)

Como já foi dito:

- Protocolos de transporte são implementados nos sistemas de extremidade (*end systems*) e **não** nos roteadores intermediários.
- TRANSPORTE: Camada 4, superior à camada de rede.



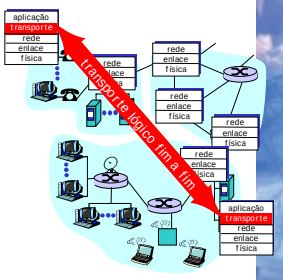
8

unesp - IBILCE - SJRP

Protocolos da camada de transporte (2)

Serviços de transporte na Internet:

- **Entrega confiável**, ordenada, ponto a ponto (TCP)
  - Congestionamento.
  - Controle de fluxo.
  - Estabelecimento de conexão (setup).
- **Entrega não confiável**, ("melhor esforço"), não ordenada, ponto a ponto ou multiponto: UDP.
- Serviços não disponíveis:
  - Tempo-real.
  - Garantias de banda.
  - Multiponto confiável.

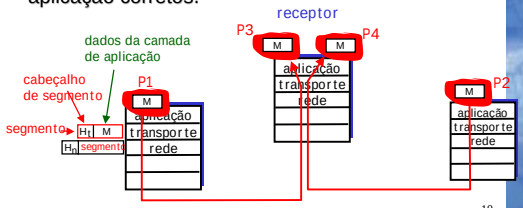


9

unesp - IBILCE - SJRP

Multiplexação/desmultiplexação (1)

- MULTIPLEXAÇÃO: Juntar dados de múltiplos processos de aplicações, **envelopando** dados com cabeçalho (usado depois para desmultiplexação).
- DESMULTIPLEXAÇÃO: entrega de segmentos recebidos para os processos da camada de aplicação corretos.



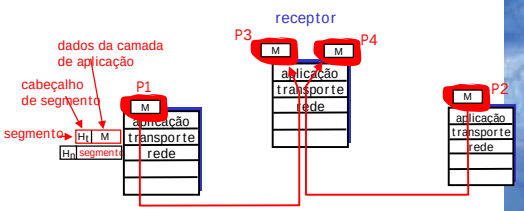
10

unesp - IBILCE - SJRP

Multiplexação/desmultiplexação (2)

Segmento - **Unidade de dados trocada entre entidades da camada de transporte.**

- Chamado de TPDU: (*Transport protocol Data Unit*) ou ***4-PDU*** (PDU da camada 4)



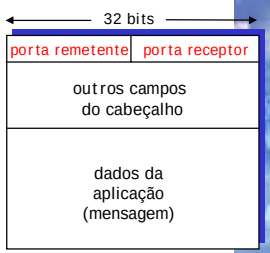
11

unesp - IBILCE - SJRP

Multiplexação/desmultiplexação (3)

Multiplexação/desmultiplexação:

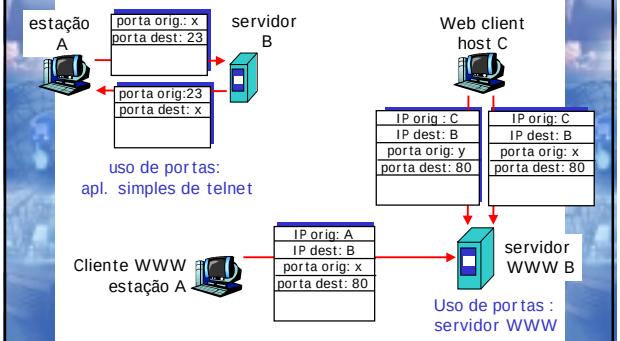
- Baseadas em **números de porta** e **endereços IP** de remetente e receptor:
  - Números de porta de remetente/receptor são enviados em cada segmento.
  - **Número de porta são bem conhecido para aplicações específicas (WKS): E-mail na porta 25, telnet na porta 23, http na porta 80, e assim por diante.**



formato de segmento TCP/UDP

12

Multiplexação/desmultiplexação: exemplos



UDP: *User Datagram Protocol* [RFC 768]

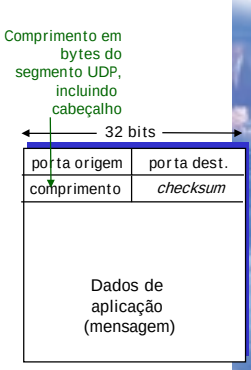
- Protocolo de transporte da Internet mínimo, "sem frescura" :-)
- Serviço "melhor esforço", resulta que segmentos UDP podem ser:
  - Perdidos.
  - Entregues à aplicação fora de ordem de envio.
- **Sem conexão:**
  - Não há "*setup*" UDP entre remetente e receptor.
  - Tratamento independente de cada segmento UDP.

Por quê existe um UDP?

- **Elimina** estabelecimento de **conexão** (o que pode causar retardo).
- **Simples**: não se mantém "estado" da conexão no remetente/receptor.
- **Pequeno cabeçalho** de segmento. Mais simples.
- **Sem controle de congestionamento**: UDP pode transmitir o mais rápido possível.

Mais sobre UDP

- Muito utilizado para aplicações de meios contínuos (voz, vídeo)
  - São tolerantes a perdas.
  - São sensíveis à taxa de transmissão.
- Outros usos de UDP (?):
  - DNS (servidor de nomes).
  - SNMP (gerenciamento).
- Transferência confiável com UDP: **deve incluir confiabilidade na camada de aplicação**.
  - Recuperação de erro específica à aplicação!



Formato de um sgmento UDP

Checksum UDP

Meta: detectar "erro" (e.g., bits invertidos) no segmento transmitido.

Remetente:

- Trata conteúdo do segmento como seqüência de inteiros de 16-bits.
- Campo *checksum* zerado.
- *Checksum*: soma (adição usando complemento de 1) do conteúdo do segmento.
- Remetente coloca *complemento do valor da soma* no campo *checksum* de UDP .

Receptor:

- Computa *checksum* do segmento recebido
- Verifica se *checksum* computado é zero:
  - NÃO - erro detectado.
  - SIM - nenhum erro detectado. *Mas ainda pode ter erros?* (Veremos mais adiante ….)

Exemplo de cálculo do *checksum* (1)

→ Ver RFC-1071 !

Considere 3 palavras de 16 bits sendo transmitidas:

```
0110011001100110
0101010101010101
0000111100001111
```

A soma das duas primeiras palavras é:

```
0110011001100110
0101010101010101
1011101110111011
```

Adicionando a terceira palavra, a soma acima fica:

```
1011101110111011
0000111100001111
1100101011001010
```

- Os complementos de 1 são obtidos convertendo todos os 0's para 1's , e todos os 1's para 0's.

Exemplo de cálculo do *checksum* (2)

Assim o complemento de 1's da soma `1100101011001010`
é `0011010100110101`

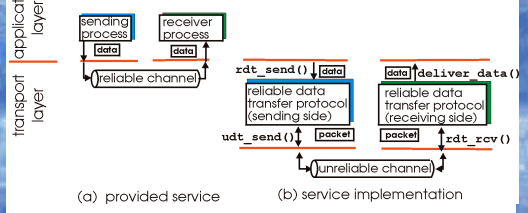
- Este valor se torna o *checksum*.
- No receptor, **todas as palavras de 16 bits são somadas**, incluindo o *checksum*.
- Se não foram introduzidos erros no pacote, a soma no receptor certamente deverá resultar em `1111111111111111`.
- Se um dos bits for zero, então algum erro foi introduzido no pacote.

Pergunta para casa: Por que o UDP usa *checksum*, se a maioria dos protocolos *data-link* (inferiores), incluindo o popular Ethernet, também fornece verificação de erro ??

unesp - IBILCE - SJRP

Princípios de Transferência confiável de dados *Reliable Data Transfer* (RDT)

- Importante nas camadas de transporte, enlace
- No topo da lista dos 10 tópicos mais importantes em redes!



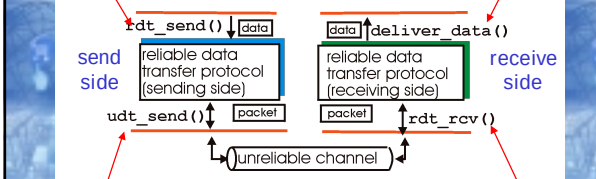
- Características do canal não confiável determinam a complexidade de um protocolo de transferência confiável de dados (RDT).

19

unesp - IBILCE - SJRP

Transferência confiável de dados (RDT): como começar

deliver_data(): chamada por **rdt** para entregar dados para camada superior.



udt_send(): chamada por ambos os lados para troca de pacotes de controle. UDT representa um *unreliable data transfer*.

rdt_rcv(): chamada quando pacote chega no lado receptor do canal.

20

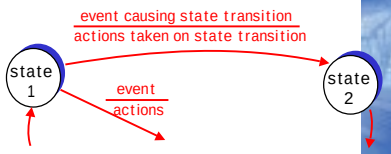
unesp - IBILCE - SJRP

Transferência confiável de dados (rdt): como começar

Iremos:

- Desenvolver incrementalmente os lados remetente e receptor do protocolo confiável RDT.
- Considerar apenas fluxo unidirecional de dados
  - Mas a informação de controle flui em ambos sentidos!
- Usar máquinas de estados finitos (FSM - *Finite State Machine*) para especificar remetente e receptor.

ESTADO: neste "estado", o próximo estado é determinado unicamente pelo próximo evento.



21

unesp - IBILCE - SJRP

Rdt1.0: transferência confiável usando um canal confiável

- Suposição: Canal subjacente perfeitamente confiável.
  - **Não apresenta erros de bits.**
  - **Não apresenta perda de pacotes.**
- FSMs separadas, para remetente e receptor:
  - Remetente envia dados **pelo** canal subjacente.
  - Receptor recebe dados **do** canal subjacente.

(a) rdt1.0: sending side (b) rdt1.0: receiving side

22

unesp - IBILCE - SJRP

rdt2.0 - Modelo um pouco mais realista:

- Bits num pacote podem ser **corrompidos**.
- Danos podem ocorrer nos componentes físicos da rede, quando um pacote é transmitido, propagado ou "buferizado".

Entretanto, continuamos supondo que **todos** os pacotes transmitidos são **recebidos na ordem em que são enviados** (ainda que seus bits possam estar corrompidos).

23

unesp - IBILCE - SJRP

Rdt2.0: canal com erros de bits

- Canal subjacente pode inverter bits no pacote
  - Lembre-se: *checksum* UDP pode detectar erros de bits.
- ***A*** questão é: como recuperar dos erros?
  - *Confirmação (ACKs):* receptor avisa explicitamente ao remetente que pacote chegou bem
  - *Confirmação negativos (NAKs):* receptor avisa explicitamente ao remetente que pacote tinha erros.
    - Remetente retransmite pacote ao receber um NAK

(> Cenários humanos usando ACKs, NAKs ? <)

24

Mensagens de controle

- Permitem o receptor **informar** ao emissor o que foi recebido corretamente e o que foi recebido com erro, exigindo repetição.
- Em redes de computadores, protocolo RDT baseados em retransmissão são chamados de Protocolos **ARQ** (***A**utomatic **R**epeat re**Q**uest*).

25

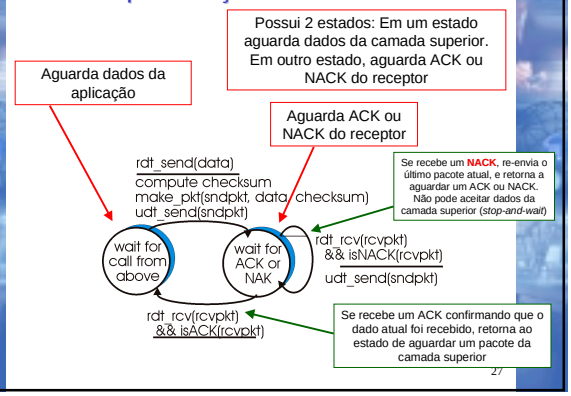
Novas capacidades em rdt2.0

- Três capacidades adicionais são exigidas em protocolos de ARQ para lidar com a presença de erros de bits:
  - **Detecção de erros:** mecanismo para permitir que o receptor identifique quando erros de bit ocorreram.
  - **Realimentação (*feedback*) pelo receptor**: mensagens de controle (ACK, NAK) trocadas entre receptor → remetente.
  - **Retransmissão**: para corrigir os erros detectados.

Estes novos mecanismos estão presentes em **`rdt2.0`** (além do **`rdt1.0`**)

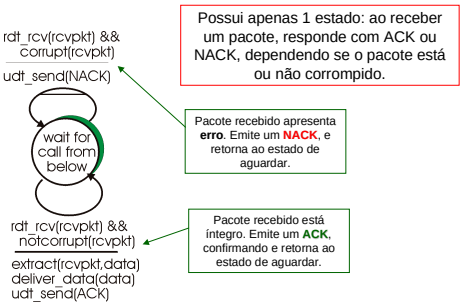
26

rdt2.0: especificação da FSM - **Emissor**



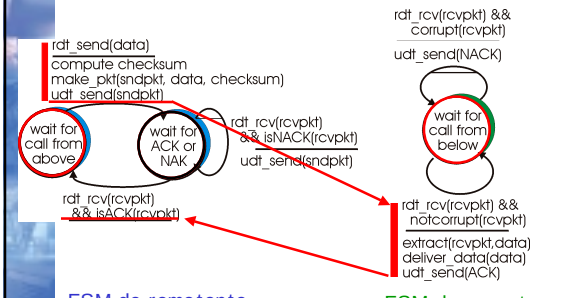
27

rdt2.0: especificação da FSM - **Receptor**



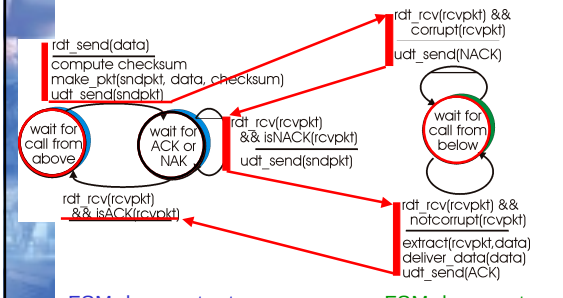
28

rdt2.0: em ação (sem erros)



29

rdt2.0: em ação (cenário de erro)



30

unesp - IBILCE - SJRP

rdt2.0 tem uma falha fatal!

O que acontece se ACK/NAK com erro ?

- Remetente não sabe o que aconteceu no receptor!
- Não se pode apenas retransmitir: possibilidade de pacotes duplicados.

O que fazer?

- Remetente usa ACKs/NAKs p/ ACK/NAK do receptor? E se perder ACK/NAK do remetente?
- Retransmitir, mas pode causar retransmissão de pacote recebido certo!

Lidando c/ duplicação:

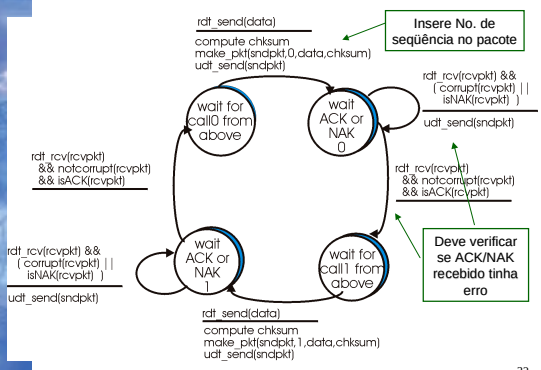
- Emissor inclui **número de sequência** p/ cada pacote.
- Remetente retransmite pacote atual se ACK/NAK recebido com erro.
- Receptor descarta (não entrega) pacote duplicado.

Stop and wait
Remetente envia um pacote, e então aguarda resposta do receptor.

31

unesp - IBILCE - SJRP

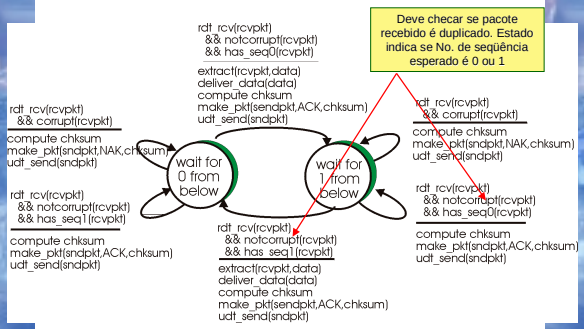
rdt2.1: **EMISSOR**, trata ACK/NAKs com erro



32

unesp - IBILCE - SJRP

rdt2.1: **receptor**, trata ACK/NAKs com erro



33

unesp - IBILCE - SJRP

rdt2.1: discussão (1)

Emissor:

- Insere No. de seqüência no pacote.
- Bastam dois Nos. de seqüência (0,1).
- Deve verificar se ACK/NAK recebido tinha erro.
- Duplicou o No. de estados
  - Estado deve "lembrar" se o pacote "corrente" tem No. de seqüência 0 ou 1.

34

unesp - IBILCE - SJRP

rdt2.1: discussão (2)

Receptor:

- Deve checar se pacote recebido é duplicado
  - Estado indica se No. de seqüência esperado é 0 ou 1.
- Receptor não tem como saber se **último ACK/NAK** foi recebido bem pelo **emissor**.

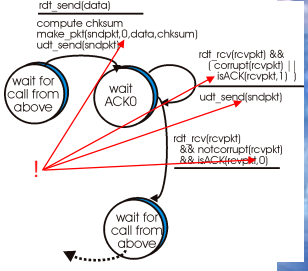
35

unesp - IBILCE - SJRP

rdt2.2: um protocolo sem **NAKs**

- Mesma funcionalidade que rdt2.1, **apenas com ACKs**.
- Ao invés de NAK, receptor envia ACK para o último pacote recebido bem.
  - **Receptor deve incluir explicitamente o No. de seqüência do pacote reconhecido.**
- ACK duplicado no remetente resulta na mesma ação que o NAK: **retransmite pacote atual.**

FSM do remetente



36

rdt3.0: canais **com erros e perdas** (1)

Nova suposição: além de corromper, o canal subjacente também pode **perder** pacotes (sejam dados ou ACKs).

Como lidar com perdas?

- Principalmente: **Como detectar perda de pacotes ?**
- O que fazer quando pacotes são perdidos?

Checksum, No. de seqüência, ACKs, ou retransmissões podem ajudar, mas não serão suficientes.

37

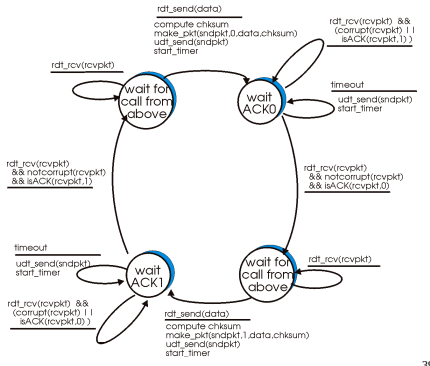
rdt3.0: canais com erros e perdas (2)

Abordagem: remetente aguarda um tempo "razoável" pelo ACK.

- Retransmite se nenhum ACK recebido neste intervalo
- Se pacote (ou ACK) apenas atrasado (e não perdido):
  - A retransmissão será **duplicada**, mas uso de número de seqüência já cuida disto.
  - Receptor deve **especificar número de seqüência** do pacote sendo reconhecido (**confirmado**).
- Exige uso de temporizadores.

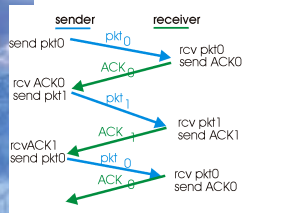
38

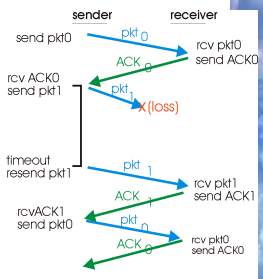
rdt3.0: remetente



39

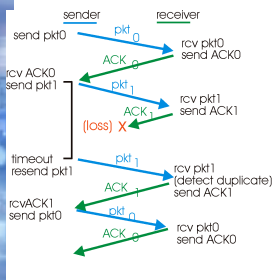
rdt3.0 em ação

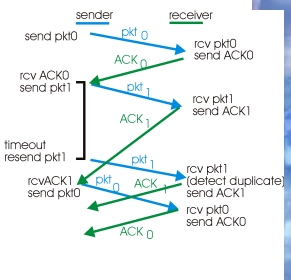




40

rdt3.0 em ação





41

Desempenho de rdt3.0

- rdt3.0 funciona, porém seu desempenho é muito ruim
- Exemplo: enlace de 1 Gbps, retardo fim a fim de 15 ms, pacote de 1KB:

$$T_{transmit} = \frac{8kb/pacote}{10^{**}9\ b/seg} = 8\ microseg$$

$$\text{Utilização} = U = \frac{\text{fração do tempo remetente ocupado}}{} = \frac{8\ microseg}{30.016\ mseg} = 0{,}00015$$

Pacote de 1KB a cada 30 mseg → vazão de **33kB/seg** num enlace de 1 Gbps !
Protocolo limita uso dos recursos físicos!

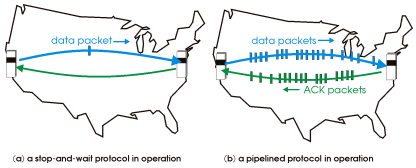
42

unesp - IBILCE - SJRP

Protocolos "dutados" (*pipelined*)

Dutagem (*pipelining*): remetente admite múltiplos pacotes "em trânsito", ainda não reconhecidos.

- Faixa de números de seqüência deve ser aumentada.
- *Buffers* no remetente e/ou no receptor.



(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

- Duas formas genéricas de protocolos "dutados": *volta-N, retransmissão seletiva.*

43

unesp - IBILCE - SJRP

GBN - Go Back N (1)

- Emissor: pode enviar múltiplos pacotes sem aguardar ACK do receptor.
- Entretanto, restringe-se a ter não mais do que um valor máximo N de pacotes não confirmados no duto.

44

unesp - IBILCE - SJRP

GBN - Go Back N (2)

send_base = Seq. # do pacote mais velho não confirmado no duto.
nextseqnum = Menor Seq. # não usado (Seq.# do próximo a enviar).

[0, send_base - 1] = Pacotes transmitidos e confirmados
[send_base, nextseqnum -1] = Pacotes enviados mas não confirmados
[nextseqn, send_base + N - 1] = Disponíveis para serem usados imediatamente, assim que dados chegarem da camada superior.

***Sequence number* ≥ (base + N) : não podem ser usados até que um pacote não confirmado que esteja atualmente no duto tenha sido confirmado.**

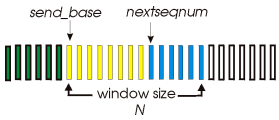
45

unesp - IBILCE - SJRP

GBN - Go Back N (3)

- Emissor: O intervalo de números de seqüência para pacotes transmitidos mas ainda não confirmados poder ser visto como uma **janela** (*window*) de tamanho *N* sobre o intervalo de números de seqüência.
- À medida que o protocolo opera, **a janela se desloca** sobre o espaço de números de seqüência: ***sliding-window protocol***

Pergunta: por que limitar o número de pacotes não confirmados ?



46

unesp - IBILCE - SJRP

GBN: FSM estendida do **emissor**

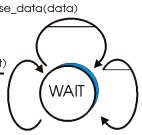
Recibo de um pacote. ACK(n): confirma que todos pacotes, até - e inclusive - o No. de seqüência n foram recebidos no receptor: "ACK cumulativo" pode receber ACKs duplicados.

Chamada superior: verifica se a janela está cheia. Se não está, forma o pacote e envia. Se está cheia, recusa.

Existe um temporizador para cada pacote em trânsito. *Timeout(n):* retransmite pacote *n* e todos os pacotes com No. de seqüência maiores na janela.

```
rdt_send(data)
if (nextseqnum < base+N) {
  compute chksum
  make_pkt(sndpkt(nextseqnum),nextseqnum,data,chksum)
  udt_send(sndpkt(nextseqnum))
  if (base == nextseqnum)
    start_timer
  nextseqnum = nextseqnum + 1
  }
else
  refuse_data(data)
```

```
rdt_rcv(rcv_pkt) && notcorrupt(rcvpkt)
base = getacknum(rvcpkt)+1
if (base == nextseqnum)
  stop_timer
else
  start_timer
```

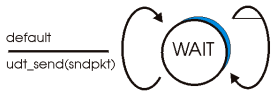



```
timeout
start_timer
udt_send(sndpkt(base))
udt_send(sndpkt(base+1)
.....
udt_send(sndpkt(nextseqnum-1))
```

47

unesp - IBILCE - SJRP

GBN: FSM estendida do **receptor**

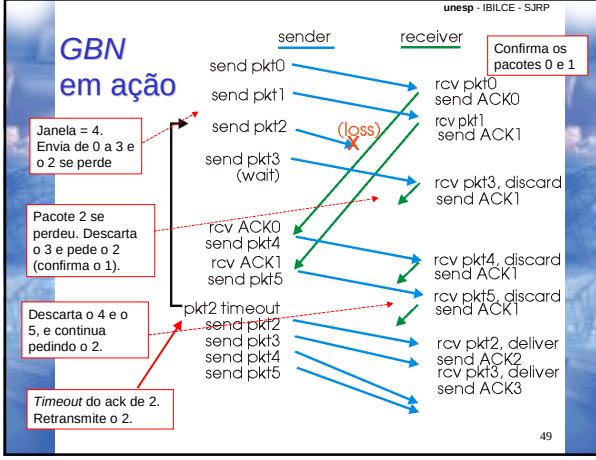


Receptor é muito simples:

- Usa apenas ACK: sempre envia ACK para pacote recebido o.k. com o maior No. de seq. em ordem
  - Pode gerar ACKs duplicados
  - Só precisa se lembrar do `expectedseqnum`
- Pacote fora de ordem:
  - Descarta (não armazena) → receptor não usa *buffers* !
  - Manda ACK de pacote com maior No. de seqüência em ordem.

48

GBN em ação



GBN não é o TCP

- GBN incorpora **quase** todas as técnicas que serão encontradas nos componentes do TCP (visto mais adiante):
  - Número de seqüência;
  - *Checksum*;
  - *ACK* cumulativos;
  - *Timeouts*;
  - Operação de retransmissão
- Entretanto existem diferenças entre o TCP e o *GBN*.
- Por exemplo: muitas implementações TCP fazem *buffering* de segmentos recebidos corretamente, mas fora de ordem.
- TCP é um híbrido de **GBN** e **Repetição Seletiva** (a seguir).

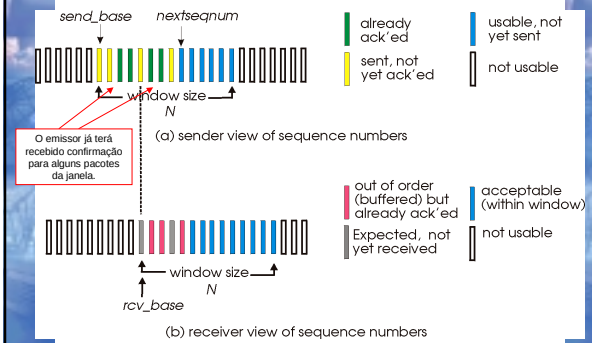
Problemas do GBN

- GBN tem problemas de performance.
  - Se o **tamanho da janela é grande** e o **atraso** da rede também é grande, **muitos pacotes podem estar no duto**.
  - Um único erro em um segmento resulta na **retransmissão de um grande número de segmentos**, a maioria desnecessários.
- À medida em que a probabilidade de **erro do canal cresce**, o duto fica lotado de **retransmissões desnecessárias**.

Repetição seletiva

- Evita retransmissões desnecessárias.
  - O emissor **retransmite apenas** os pacotes que ele suspeita terem sido **recebidos com erro** pelo receptor.
- **Receptor** confirma **individualmente** todos os pacotes **recebidos corretamente**.
  - Armazena pacotes no *buffer*, conforme necessário, para posterior entrega em ordem à camada superior.
- Emissor apenas re-envia pacotes para os quais o ACK não foi recebido.
  - Temporizador de remetente para cada pacote sem ACK.
- Janela do emissor
  - ***N*** números de seqüência consecutivos.
  - Outra vez limita nos. de seqüência de pacotes enviados, mas ainda não reconhecidos.

Repetição seletiva: janelas de remetente e receptor



Repetição seletiva no **emissor**

- **Dados recebidos de acima.** O emissor verifica o próximo número de seqüência disponível para o pacote.
  - Se o número de seqüência estiver dentro da janela do emissor, os dados são empacotados e enviados; caso contrário são *bufferizados* ou devolvidos à camada superior para transmissão posterior.
- ***Timeout***. Temporizadores são usados para proteger contra perda de pacotes.
  - Somente um único pacote será transmitido no caso de *timeout*.
- ***ACK* recebido.** Se um ACK for recebido, o emissor marca o pacote como recebido.
  - Se o número de seqüência do pacote for igual à **send-base,** então a base da janela avança até o pacote com o menor número de seqüência não-confirmado. Se houver pacotes não transmitidos, com números de seqüência que caem agora dentro da janela, estes pacotes são transmitidos.

unesp - IBILCE - SJRP

Repetição seletiva no **receptor** (1)

- Pacote com número de seqüência no intervalo **[rcv_base, rcv_base+N-1]** é recebido corretamente:

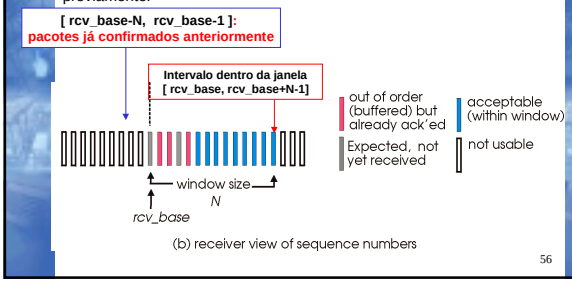
  Neste caso, o pacote recebido cai dentro da janela do receptor e um pacote seletivo de ACK é retornado ao remetente. Se o pacote não foi recebido previamente, ele é armazenado. Se este pacote tiver um número de seqüência igual à base da janela da recepção (rcv_base), então este pacote, e os quaisquer pacotes previamente armazenados e numerados em seqüência (começando com o rcv_base) são entregues à camada superior. A janela de recepção é movida então para a frente pelo número total dos pacotes entregues à camada superior.
- Pacote com número de seqüência é recebido dentro de **[rcv_base-N, rcv_base-1]:** Neste caso, um ACK deve ser gerado, mesmo que este seja um pacote que o receptor já tenha confirmado previamente.
- Se não: Ignora o pacote.

55

unesp - IBILCE - SJRP

Repetição seletiva no **receptor** (2)

Pacote com número de seqüência é recebido dentro de **[rcv_base-N, rcv_base-1]:** Neste caso, um ACK deve ser gerado, mesmo que este seja um pacote que o receptor já tenha confirmado previamente.



(b) receiver view of sequence numbers

56

unesp - IBILCE - SJRP

Repetição seletiva - resumo

emissor

Dados de cima:
- Se próximo No. de seqüência na janela, envia pacote.

Timeout(n):
- Reenvia pacote n, reiniciar temporizador

ACK(n) em [sendbase,sendbase+N]:
- Marca pacote n "recebido".
- Se N for menor pacote não reconhecido, avança base da janela ao próximo No. de seqüência não reconhecido

receptor

- Pacote n dentro de [rcvbase, rcvbase+N-1]

Envia ACK(n)

Fora de ordem: buffer

Em ordem: entrega (tb. entrega pacotes em ordem do buffer), avança janela p/ próximo pacote ainda não recebido.

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

ACK(n), mesmo que já tenha enviado antes.

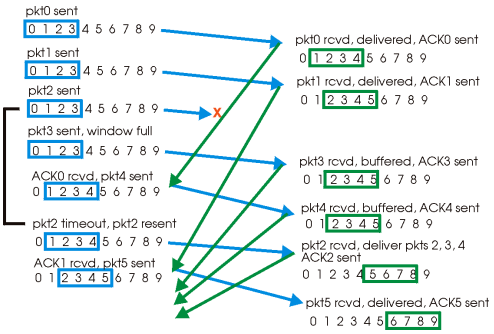
- Senão:

Ignora.

57

unesp - IBILCE - SJRP

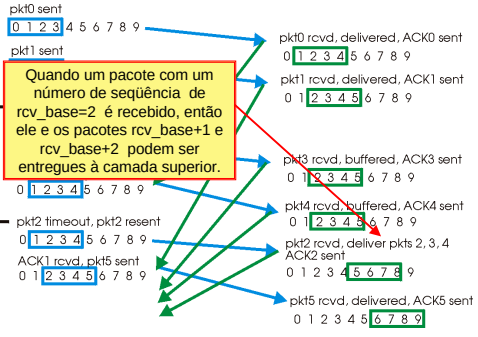
Retransmissão seletiva em ação



58

unesp - IBILCE - SJRP

Retransmissão seletiva em ação



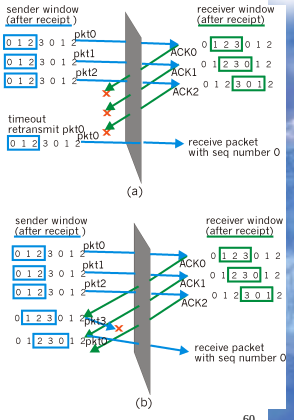
59

unesp - IBILCE - SJRP

Repetição seletiva: dilema

Exemplo:
- Nos. de seqüência : 0, 1, 2, 3
- Tamanho de janela =3
- Receptor não vê diferença entre os dois cenários (a) e (b) !
- Incorretamente passa dados duplicados como novos em (a)

Q: Qual a relação entre tamanho de No. de seqüência e tamanho de janela?



60

unesp - IBILCE - SJRP

TCP: Visão geral (1)

RFCs: 793, 1122, 1323, 2018, 2581

- Peet-to-Peer (*P2P*):
  - Único emissor transmite para um único receptor.
- Fluxo de bytes, ordenados, confiável:
  - Não estruturado em mensagens.
- Dutado (*pipelined*):
  - Tamanho da janela ajustado por controle de fluxo e congestionamento do TCP.
- *Buffers* de envio e recepção, e variáveis de estado para cada conexão.

61

unesp - IBILCE - SJRP

TCP: Visão geral (2)

RFCs: 793, 1122, 1323, 2018, 2581

1500 bytes, 536 bytes ou 512 bytes

- Transmissão *Full Duplex*:
  - Fluxo de dados **bi-direcional** na **mesma conexão**.
  - MSS: tamanho máximo de segmento **de dados**.
- Orientado a conexão:
  - *Handshaking* de **3 vias** (troca de MSGs de controle) inicia estado de remetente, receptor antes de trocar dados.
- Fluxo controlado: Receptor não será afogado.

62

unesp - IBILCE - SJRP

TCP: estrutura do segmento

PSH = push: envia dados já (pouco usado)

32 bits

URG: dados urgentes (pouco usado)

Contagem de dados por bytes (não segmentos!)

Tamanho do *header* em palavras de 32 bits

ACK: No. ACK é válido

RST, **SYN**, **FIN**: gestão de conexão (comandos de *estabelecimento*, *liberação*)

checksum Internet (como UDP)

No. bytes receptor quer aceitar: **Controle de Fluxo**.

20 **bytes** fixos no ***header***

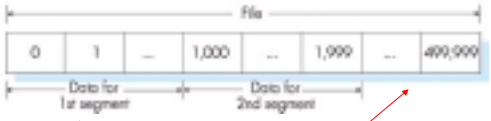
No. porta origem	No. porta dest
número de seqüência	
número de confirmação (ACK)	
tam. cab. / sem uso / U A P R S F	janela receptor
checksum	ptr dados urg.
Opções (tamanho variável)	
dados da aplicação (tamanho variável)	

63

unesp - IBILCE - SJRP

Numeração de segmentos

Transmissão de um arquivo de 500.000 bytes, com MSS = 1.000 bytes



1o. segmento

último segmento

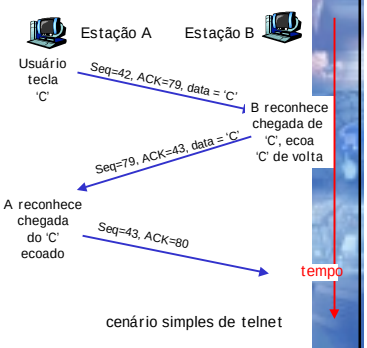
64

unesp - IBILCE - SJRP

TCP: Números de seqüência e ACKs

- Nos. de seqüência: "número" dentro do fluxo de bytes do primeiro byte de dados do segmento.
- ACKs: No. de seqüência do próximo byte esperado do outro lado.
  - ACK cumulativo.

P: Como receptor trata segmentos fora da ordem?
- R: especificação do TCP é omissa - deixado ao implementador.



cenário simples de telnet

65

unesp - IBILCE - SJRP

TCP: transferência confiável de dados

Passagem dos dados da aplicação ao TCP e o subseqüente enquadramento e transmissão de um segmento.

event: data received from application above

create, send segment

event: timer timeout for segment with seq # y

retransmit segment

Cada vez que o TCP entrega um segmento ao IP, ele inicia um **temporizador para aquele segmento**. Se este temporizador expirar, um evento de interrupção é gerado no host. O TCP responde ao evento de timeout, re-enviando o segmento que causou o ***timeout***.

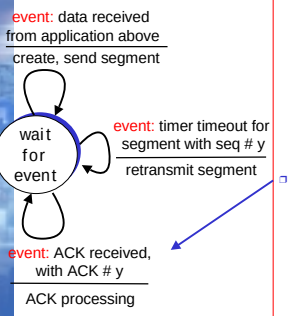
event: ACK received, with ACK # y

ACK processing

Chegada de um segmento de **reconhecimento** (**ACK**) vindo do receptor.

66

TCP: transferência confiável de dados



- Chegada de um segmento de reconhecimento (ACK) do receptor (mais especificamente, um segmento que contem um valor válido do campo do ACK). Aqui, o remetente TCP deve determinar se o ACK é um ***"first-time ACK"*** para um segmento o qual o remetente ainda está para receber um reconhecimento, ou uma **"duplicata de ACK"** que re-reconheça um segmento para qual o remetente já tenha recebido um reconhecimento anteriormente.
- No exemplo da **chegada de um ACK** *first-time*, o emissor sabe agora que todos os dados até o byte que está sendo reconhecido foram recebidos corretamente no receptor. O emissor pode assim atualizar sua variável do estado do TCP que segue o número de seqüência do último byte que tenha sido recebido corretamente, e em ordem, no receptor.

Emissor TCP simplificado

```
00  sendbase = número de seqüência inicial
01  nextseqnum = número de seqüência inicial
02
03  loop (forever) {
04    switch(event)
05    event: dados recebidos da aplicação acima
06        cria segmento TCP com número de seqüência nextseqnum
07        inicia temporizador para segmento nextseqnum
08        passa segmento para IP
09        nextseqnum = nextseqnum + comprimento(dados)
10    event: expirado temporizador de segmento c/ No. de seqüência y
11        retransmite segmento com número de seqüência y
12        calcula novo intervalo de temporização para segmento y
13        reinicia temporizador para número de seqüência y
14    event: ACK recebido, com valor de campo ACK de y
15        se (y > sendbase) { /* ACK cumulativo de todos dados até y */
16            cancela temporizadores p/ segmentos c/ Nos. de seqüência < y
17            sendbase = y
18            }
19        senão { /* é ACK duplicado para segmento já reconhecido */
20            incrementa número de ACKs duplicados recebidos para y
21            if (número de ACKs duplicados recebidos para y == 3) {
22              /* TCP:  retransmissão rápida */
23              reenvia segmento com número de seqüência y
24              reinicia temporizador para número de seqüência y
25            }
26    } /* fim de loop forever */
```


Retransmissão rápida

- Quando um receptor TCP recebe um segmento com um **número de seqüência que seja maior do que próximo número de seqüência em-ordem esperado**, ele detecta uma falha no fluxo de dados - isto é, um segmento faltante.
- Uma vez que o TCP não usa reconhecimentos negativos, o receptor não pode emitir uma negativa de ACK explícita de volta ao emissor. Ao invés disso, re-reconhece simplesmente (isto é, gera um ACK em duplicata para) o último byte em-ordem dos dados que ele recebeu.
- Se o **emissor receber três ACKs duplicados** para o **mesmo dado** (segmento), ele assume como uma indicação que foi perdido o segmento que vem em seguida ao segmento que foi confirmado ("*ACK*ado") três vezes . Neste caso, o TCP executa **uma retransmissão rápida** [RFC 2581], re-enviando o segmento faltante antes que o temporizador do segmento expire.

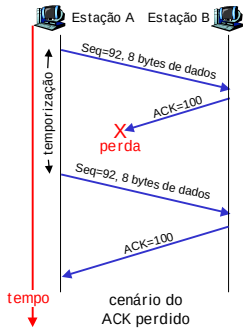
Geração de ACKs no TCP [RFCs 1122, 2581]

Evento	Ação do receptor TCP
Chegada de segmento em ordem sem lacunas, todos anteriores já reconhecidos.	ACK retardado. Espera até 500ms pelo próx. segmento. Se não chegar segmento, envia ACK
Chegada de segmento em ordem sem lacunas, um ACK retardado pendente.	Envia imediatamente um único ACK cumulativo.
Chegada de segmento fora de ordem, com No. de seq. maior que esperado → lacuna.	Envia ACK duplicado, indicando No. de seq. do próximo byte esperado.
Chegada de segmento que preenche a lacuna parcial ou completamente.	ACK imediato se segmento no início da lacuna.

Geração de ACKs no TCP [RFCs 1122, 2581]

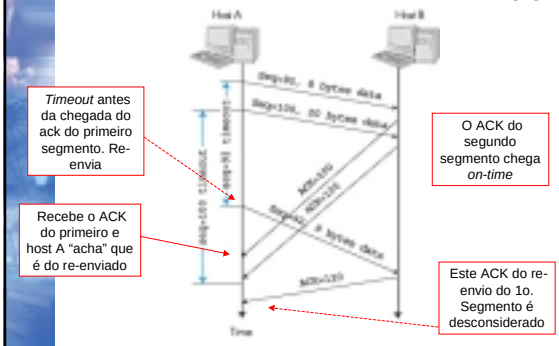
Evento	Ação do receptor TCP
Chegada do segmento em ordem, com número de seqüência previsto. **Todos os dados até o núm. de seq. previsto já reconhecidos**. Nenhum *gap* nos dados recebidos.	ACK atrasado. Espera até 500ms pela chegada de um outro segmento em ordem. Se o próximo segmento não chegar em ordem neste intervalo, **emita um ACK do anterior**.
Chegada de um segmento em ordem com número de seqüência previsto. Um outro segmento em ordem aguardando transmissão de ACK. **Nenhum gap nos dados recebidos.**	Emita imediatamente a único ACK cumulativo. **Confirmando ambos os segmentos em-ordem**.
Chegada do segmento fora de ordem com **número de seqüência mais alto do que esperado**. Detectado *gap*.	**Emita imediatamente o ACK duplicado**, indicando o número de seqüência do próximo byte esperado.
Chegada de segmento que completa parcial ou completamente um *gap* nos dados recebidos.	Emita imediatamente o ACK, contanto que o segmento comece no fim mais baixo da *gap*.

TCP: cenários de retransmissão (1)



unesp - IBILCE - SJRP

TCP: cenários de retransmissão (2)



Timeout antes da chegada do ack do primeiro segmento. Re-envia

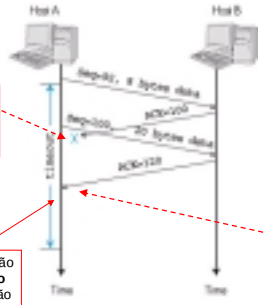
Recebe o ACK do primeiro e host A "acha" que é do re-enviado

O ACK do segundo segmento chega *on-time*

Este ACK do re-envio do 1o. Segmento é desconsiderado

unesp - IBILCE - SJRP

TCP: cenários de retransmissão (3)



Envia dois segmentos de uma vez e aguarda

O ACK do primeiro segmento se perde

(i) Indica que B **recebeu tudo certo, até o 100** (aguarda o 120)

(ii) Recebe a indicação que B **recebeu tudo certo, até o 100** e não re-envia nada

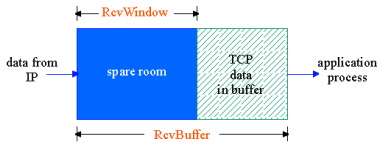
unesp - IBILCE - SJRP

TCP: Controle de Fluxo (1)

Para **emissor** não esgotar os *buffers* do **receptor** por transmitir demais, ou muito rapidamente.

RcvBuffer = tamanho do Buffer de recepção.

RcvWindow = informa ao emissor quantidade de espaço vazio no *buffer*. Este valor é mantido como variável em cada emissor (Lembrar *Full Duplex*).



buffering pelo receptor

unesp - IBILCE - SJRP

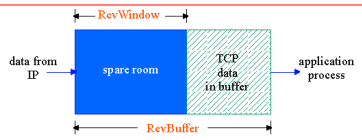
TCP: Controle de Fluxo (2)

RECEPTOR: explicitamente avisa o **emissor** da quantidade de **espaço livre disponível** (muda dinamicamente).

- Campo **RcvWindow** no segmento TCP .

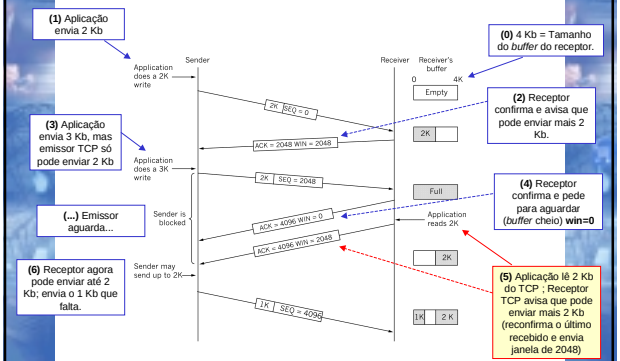
EMISSOR: mantém a quantidade de dados transmitidos, porém ainda não reconhecidos, **MENOR** que o **valor mais recente** de **RcvWindow.**

(para cada lado da conexão)



unesp - IBILCE - SJRP

Troca de informações sobre controle de fluxo TCP



unesp - IBILCE - SJRP

TCP: Tempo de Resposta (RTT) e temporização

P: Como escolher valor do temporizador TCP?

- Deve ser **maior** que o RTT.
  - Note que o RTT pode variar.
  - Muito curto: temporização prematura.
    - Retransmissões são desnecessárias.
  - Muito longo: reação demorada à perda de segmentos.

P: Como estimar RTT?

- `RTTamostra`: tempo medido entre a transmissão do segmento e o recebimento do ACK correspondente.
  - Ignora retransmissões, e segmentos com ACKs cumulativos
- `RTTamostra` vai variar, queremos "ponderador" de RTT estimado.
  - Usa várias medições recentes, não apenas o valor corrente (`RTTamostra`)

TCP: Tempo de Resposta (RTT) e Temporização (1)

`RTT_estimado = (1-x)* RTT_estimado + x*RTT_amostra`

- Trata-se de uma Média corrente exponencialmente ponderada.
- Influência de cada amostra diminui exponencialmente com o tempo
- Valor típico de x = 0.125 (1/8)

RTT_estimado = 0.875 **RTT_estimado** + 0.125 • **RTT_amostra**.

TCP: Tempo de Resposta (RTT) e Temporização (2)

Escolhendo o intervalo de temporização:

- **RTT_estimado** mais uma "**margem de segurança**"
- Se há variação grande em **RTT_estimado, exige** margem de segurança maior

Temporização = RTT_estimado + 4*Desvio

```
Desvio = (1-x)* Desvio +
          x * | RTT_amostra - RTT_estimado |
```

Exemplos interativos:
http://www.ce.chalmers.se/~fcela/tcp-tour.html

TCP: Gerenciamento de Conexões (1)

Lembrete: Remetente e receptor TCP estabelecem "conexão" **antes** de trocar segmentos de dados. Fluxo de dados vai nos dois sentidos.

- Inicializam variáveis TCP:
  - Números de seqüência.
  - *Buffers*, informações de controle de fluxo (por exemplo **RcvWindow**).
- **Cliente**: é aquele que inicia a conexão

```
Socket clientSocket = newSocket("hostname","port number");
```

- **Servidor**: é aquele contatado pelo cliente

```
Socket connectionSocket = welcomeSocket.accept();
```


TCP: Iniciando Conexão (1)

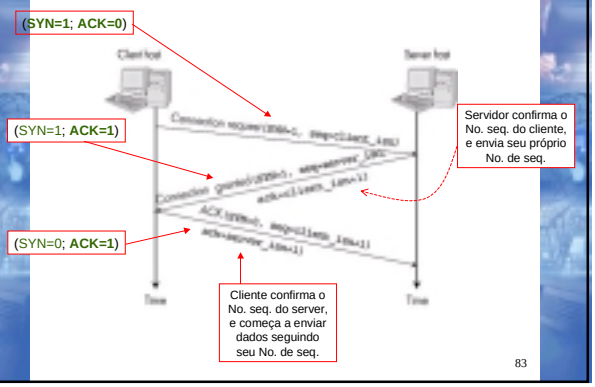
Inicialização em 3 passos (*3-way handshake*):

(1): **Cliente** envia segmento de controle **SYN** do TCP ao servidor.
- Bit **SYN** do TCP é ajustado como 1. (SYN=1; **ACK=0**)
- Cliente especifica No. de Seqüência inicial.

(2): **Servidor** recebe **SYN**, responde com segmento de controle **SYN-ACK**
- Ajusta SYN=1 E ACK=1 (**SYN=1**; **ACK=1**)
- Confirma SYN recebido.
- Aloca *buffers*, especifica No. seq. inicial de servidor para o receptor.

(3): **Cliente** recebe **SYN=1; ACK=1**, e responde com segmento de controle **ACK** e **começa a enviar dados**. (SYN=0; **ACK=1**)

TCP: Iniciando Conexão (2)



TCP: Fechando uma Conexão (1)

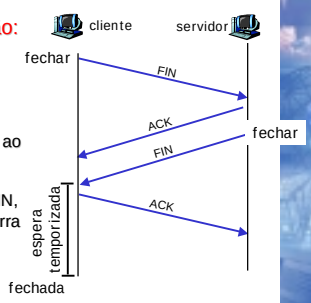
Encerrando uma conexão:

cliente fecha soquete:
clientSocket.close();

Passo 1: cliente envia segmento de controle **FIN** ao servidor.

Passo 2: servidor recebe FIN, responde com **ACK**. Encerra a conexão, enviando **FIN**.

(Segue.... →)



unesp - IBILCE - SJRP

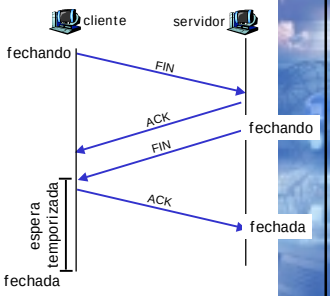
TCP: Fechando uma Conexão (2)

Passo 3: cliente recebe **FIN**, responde com **ACK**.

- Entre em "espera temporizada" - responderá com ACK a FINs recebidos

Step 4: servidor, recebe **ACK**. Conexão encerrada.

Note: com pequena modificação, consegue tratar FINs simultâneos.



85

unesp - IBILCE - SJRP

TCP: Ciclo de vida (1)

Ciclo de vida do servidor TCP

86

unesp - IBILCE - SJRP

TCP: Ciclo de vida (2)

Ciclo de vida do cliente TCP

87

unesp - IBILCE - SJRP

Princípios de Controle de Congestionamento (1)

Congestionamento:

- Informalmente: "trata-se de muitas fontes enviando muitos dados mais rapidamente do que a **REDE** pode tratar."
- Diferente de controle de fluxo!

Retransmissão de pacotes trata o **sintoma** do **congestionamento** da rede (**perda de segmentos da camada de transporte**) mas **não trata a causa do congestionamento**: muitas origens tentando enviar dados numa taxa muito alta.

88

unesp - IBILCE - SJRP

Princípios de Controle de Congestionamento (2)

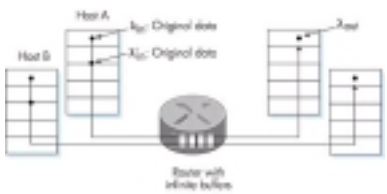
- Como se manifesta:
  - Perda (*drop*) de pacotes (esgotamento de *buffers* em roteadores).
  - Retransmissão de pacotes (devido aos *drops*).
  - Longos atrasos (grande enfileiramento nos *buffers* dos roteadores).
- Um dos 10 problemas mais importantes em redes!

89

unesp - IBILCE - SJRP

Causas e custos de congestionamento: cenário 1

- Dois emissores A e B e dois receptores.
- Um roteador, com *buffers* infinitos.
- Sem nenhum tipo retransmissão (sem controle de erros).
- Aplicação entrega dados para a camada de transporte a uma taxa λ_{in} .
- Capacidade do link é *R* bps



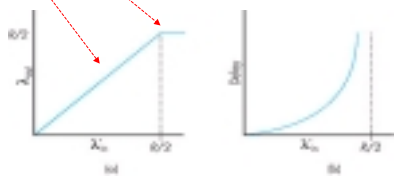
90

unesp - IBILCE - SJRP

Causas e custos de congestionamento: cenário 1 (1)

Tudo que é enviado pelo emissor é recebido no receptor a uma taxa finita.

Quando a taxa de emissão é **ACIMA** de R/2, a **recepção** é sempre R/2.



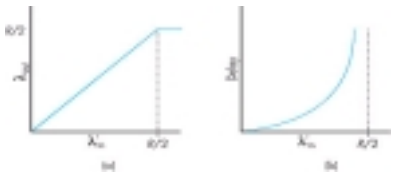
throughput **por conexão** (número de bytes/seg no receptor) em função da taxa de emissão da conexão

À medida que a taxa de emissão se aproxima de **R/2** o atraso em para cada emissor se aproxima de infinito

91

unesp - IBILCE - SJRP

Causas e custos de congestionamento: cenário 1 (2)



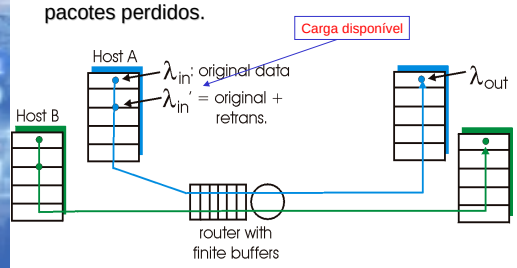
- O link não consegue enviar pacotes a uma taxa estacionária que ultrapasse R/2.
- Não importa quão alta seja a taxa que A e B ajustem para emissão, eles nunca obterão um *throughput* superior a R/2
- Existem grandes retardos quando a rede está congestionada.
- Mesmo neste cenário irreal, existe uma vazão máxima atingível.

92

unesp - IBILCE - SJRP

Causas e custos de congestionamento: cenário 2

- Um roteador, com *buffers finitos*.
- Conexão confiável: retransmissão pelo emissor de pacotes perdidos.

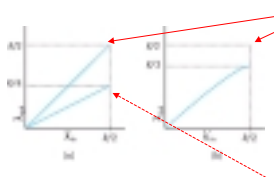


93

unesp - IBILCE - SJRP

Causas e custos de congestionamento: cenário 2

- (a) Considere que o host A só envia quando *buffer* do router está livre. Neste caso $\lambda_{in} = \lambda'_{in}$ = *throughput* da conexão.
- (b) Considere agora que $\lambda'_{in} = 0{,}5\ R$ e que apenas *0,333 R* chega ao receptor.
  - Isso significa que *0,333 R* bytes/seg é a carga original e *0,266 R* bytes/seg são retransmissões.



A taxa de transmissão nunca poderá ultrapassar R/2

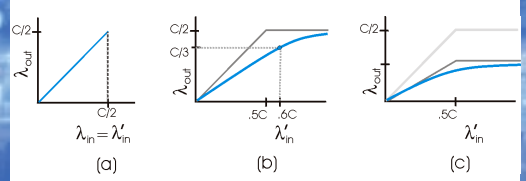
Imagine que cada pacote tem de ser enviado 2 vezes devido a atrasos

94

unesp - IBILCE - SJRP

Causas e custos de congestionamento: cenário 2

- Sempre: $\lambda_{in} = \lambda_{out}$ (Denominado de "carga disponível" ou "*goodput*")
- Retransmissão "perfeito" apenas quando perda: $\lambda'_{in} > \lambda_{out}$



95

unesp - IBILCE - SJRP

Causas e custos de congestionamento: cenário 2

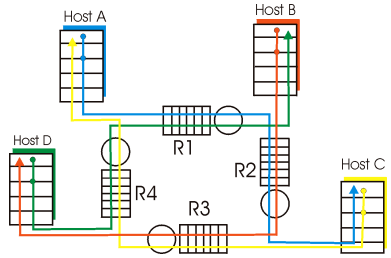
Outros "Custos" de congestionamento:

- Mais trabalho (retransmissão) para determinado "*goodput*".
- Retransmissões desnecessárias:
  - São enviadas múltiplas cópias do pacote.

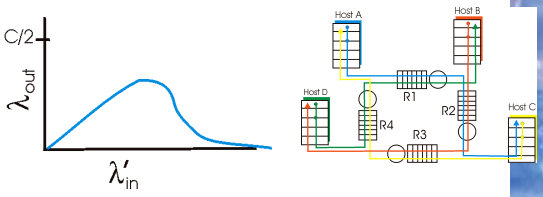
96

Causas e custos de congestionamento: cenário 3

- Quatro emissores.
- Caminhos com múltiplos enlaces.
- Temporização/retransmissão.



Causas e custos de congestionamento: cenário 3



Outro "custo" de congestionamento:

- Quando pacote é descartado, qualquer capacidade de transmissão já usada (antes do descarte) para esse pacote foi desperdiçada!

Abordagens de controle de congestionamento

Duas abordagens **amplas** para **controle de congestionamento**:

Controle de congestionamento fim-a-fim :

- Não possui realimentação explícita pela rede.
- Congestionamento inferido das perdas, e do retardo observados pelo sistema final.
- **É a abordagem usada pelo TCP.**

Controle de congestionamento com apoio da rede:

- Roteadores realimentam os emissores a respeito da situação da rede.
  - Por exemplo Bit único indicando congestionamento no link (SNA, DECbit, TCP/IP ECN. É usado pelo ATM).
  - Taxa explícita fixada para envio pelo emissor.

TCP: Controle de Congestionamento

- **Verdadeira solução para o congestionamento é diminuir a taxa de transmissão de dados.**
- A idéia é não incluir um novos pacotes na rede até que os antigos tenham deixado a rede (ou seja, até que os antigos tenham sido entregues).
- **O TCP tenta alcançar esse objetivo manipulando dinamicamente o tamanho da janela.**

TCP: Controle de Congestionamento

- Antes de discutirmos como o TCP reage aos congestionamentos:
  - → O que ele faz para tentar evitar sua ocorrência ?
- **Quando uma conexão é estabelecida, deve-se escolher um tamanho de janela adequado.**
- O **receptor** pode **especificar uma janela** a partir do **tamanho de seu *buffer***.
- Se o **transmissor** se mantiver **dentro do tamanho da janela, não haverá problemas causados pela sobrecarga** dos *buffers* no receptor.
  - Mas eles ainda podem ocorrer devido a congestionamentos internos na rede.

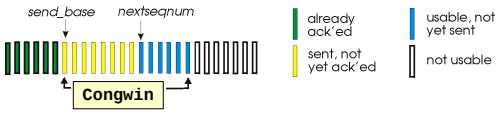
TCP: Controle de Congestionamento

- Solução Internet é entender que existem dois problemas potenciais: **capacidade da rede** e a **capacidade do receptor** → lidar com cada um em separado.
- Para isso, **cada transmissor mantém duas janelas**: a janela fornecida pelo receptor e a janela de congestionamento ***congestion window*** (ou ***congwin***).
- Cada uma mostra o número de bytes que o transmissor pode enviar → pode enviar o valor mínimo das janelas.

unesp - IBILCE - SJRP

TCP: Controle de Congestionamento

- **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 a cada RTT:

$$throughput = \frac{w * MSS}{RTT} \text{ Bytes/sec}$$

unesp - IBILCE - SJRP

TCP: Controle de Congestionamento

- Portanto, **a janela efetiva é o mínimo do que o transmissor e o receptor consideram viável**.
- Se o **receptor** disser: "envie 8KB", mas o **transmissor** souber que qualquer rajada com mais de 4 KB poderá congestionar a rede, **ele enviará apenas 4 KB**.
- Se o receptor disser: "envie 8KB", e o transmissor souber que rajadas de até 32 KB passam pela rede sem problemas, ele enviará os 8 KB solicitados.

unesp - IBILCE - SJRP

TCP: Controle de Congestionamento

- "Sondagem" para banda utilizável:
  - Idealmente: transmitir o mais rápido possível (ou seja, **`Congwin`** ajustado ao máximo possível) sem perder pacotes.
  - **Aumentar `Congwin`** até perder pacotes = congestionamento.
  - Quando houver perdas: **diminui `Congwin`**, depois volta a à sondagem (aumentando) novamente.
- Duas "fases"
  - Partida lenta.
  - Evitar congestionamento.
- Variáveis importantes:
  - **`Congwin`**
  - **`threshold:`** define limiar entre fases de partida lenta e controle de congestionamento.

Vejamos....

unesp - IBILCE - SJRP

Partida lenta e sondagem do congestionamento (1)

- **Conexão é estabelecida → transmissor ajusta a janela de congestionamento ao MSS em uso na conexão**.
  - Em seguida, ele envia este segmento máximo (***Maximum Segment Size*** - **MSS**).
  - Se esse **segmento for confirmado antes de ocorrer um *timeout***, o **transmissor adicionará o número de bytes de um segmento na janela de congestionamento** de modo que ela tenha capacidade **equivalente a dois segmento máximos**, e enviará os dois segmentos.

unesp - IBILCE - SJRP

Partida lenta e sondagem do congestionamento (2)

- À medida que **cada um desses segmentos for confirmado**, a **janela de congestionamento é aumentada em um tamanho destes segmento máximo, de tal forma que - quando ambos forem confirmados - a janela terá 4 vezes o valor inicial**.
- Quando a janela de congestionamento chegar a ***n*** **segmentos**, e se **todos os *n* segmentos forem confirmados** a tempo, **a janela de congestionamento será aumentada pelo número de bytes correspondentes aos *n* segmentos**.
- Na prática, cada **rajada confirmada duplica a janela de congestionamento**.

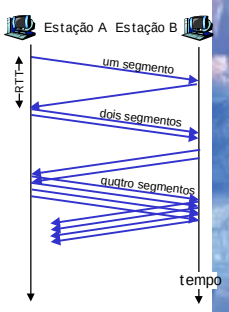
unesp - IBILCE - SJRP

TCP: Partida lenta

Algoritmo Partida Lenta

```
inicializa: Congwin = 1
for (cada segmento c/ ACK)
      Congwin++
until (evento de perda OR
       CongWin > threshold)
```

- Aumento exponencial (a cada RTT) no tamanho da janela (não muito lenta!).
- Evento de perda: temporizador (***Tahoe TCP***) e/ou três ACKs duplicados (***Reno TCP***).



unesp - IBILCE - SJRP

Threshold (1)

- **Algoritmo de controle de congestionamento da Internet → utiliza um terceiro parâmetro**, limitante (***threshold***).
- ***threshold* (limitante) →** em princípio é 64 KB.
- Quando há um *timeout*, o ***threshold*** é **atribuído à metade da janela de congestionamento atual**, e a **janela de congestionamento é reinicializada para um tamanho de segmento máximo**.

109

unesp - IBILCE - SJRP

Threshold (2)

- Em seguida, a inicialização lenta é usada para determinar o que a rede é capaz de gerenciar, só que agora **o crescimento exponencial é interrompido quando o limite é alcançado**.
- **A partir daí, as transmissões bem-sucedidas proporcionam um crescimento linear à janela de congestionamento** (o aumento é de um segmento máximo para cada rajada) em vez de um para cada segmento. **Na prática, esse algoritmo diminui o tamanho da janela de congestionamento à metade, e depois retoma seu crescimento a partir daí**.

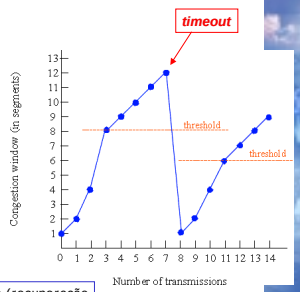
110

unesp - IBILCE - SJRP

TCP: Evitar Congestionamento (1)

Evitar congestionamento

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



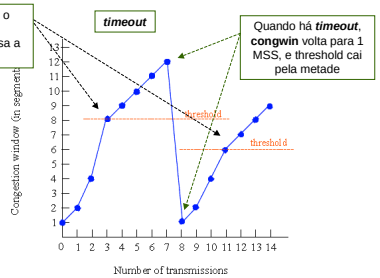
1: *TCP Reno* pula partida lenta (recuperação rápida) depois de três ACKs duplicados

111

unesp - IBILCE - SJRP

TCP: Evitar Congestionamento (2)

Quando atinge o *threshold* o crescimento passa a ser linear



Quando há ***timeout***, **congwin** volta para 1 MSS, e threshold cai pela metade

Quando atinge o *threshold* o aumento é de um segmento máximo para cada rajada em vez de um para cada segmento.

112

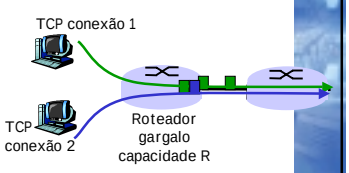
unesp - IBILCE - SJRP

AADM / *Additive-Increase, Multiplicative-Decrease (AIMD)*

- AADM: *Aumento Aditivo, Decremento Multiplicativo*
  - Aumenta janela em 1 a cada RTT.
  - Diminui janela por fator de 2 num evento de perda.

Justeza do TCP

Meta de justeza: se N sessões TCP compartilham o mesmo enlace de gargalo, cada uma deve ganhar 1/N da capacidade do enlace.



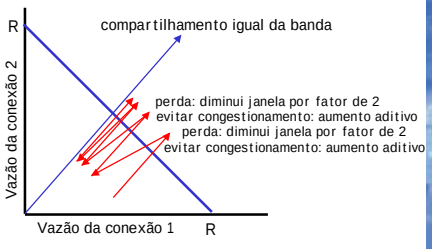
113

unesp - IBILCE - SJRP

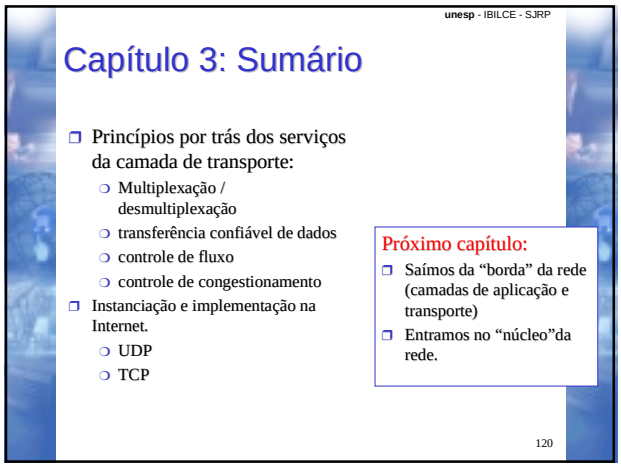
Por quê TCP é justo?

Duas sessões concorrentes:

- Aumento aditivo dá gradiente de 1, enquanto vazão aumenta.
- Decremento multiplicativa diminui vazão proporcionalmente.



114


unesp - IBILCE - SJRP
Capítulo 3: Sumário
Princípios por trás dos serviços da camada de transporte:
Multiplexação / desmultiplexação
transferência confiável de dados
controle de fluxo
controle de congestionamento
Instanciação e implementação na Internet.
UDP
TCP
Próximo capítulo:
Saímos da "borda" da rede (camadas de aplicação e transporte)
Entramos no "núcleo"da rede.
120