Infra-Estrutura de Comunicação (IF678)

Módulo III

Fonte: kurose Adaptações : Prof. Paulo Gonçalves pasg@cin.ufpe.br CIn/UFPE

Camada Transporte 3-1

Módulo 3: Camada Transporte

Nossos objetivos: ❒ Compreender os

princípios por trás dos serviços de transporte:

❍

❒ Aprender sobre os

protocolos da camada transporte na Internet:

❍

UDP: transporte não- multiplexação/demulti

orientado à conexão plexação

❍

❍

TCP: transporte orientado à Transferência

conexão confiável de dados

❍

❍

Controle de Controle de fluxo

congestionamento do TCP

❍

Controle de congestionamento

Camada Transporte 3-2

Resumo do Módulo 3

❒ 3.1 Serviços da camada

transporte ❒ 3.2 Multiplexação e

demultiplexão ❒ 3.3 transporte não-

orientado à conexão: UDP ❒ 3.4 Princípios da transferência confiável de dados

❒ 3.5 transporte orientado

à conexão: TCP

❍

Estrutura do segmento

❍

Transferência de dados confiável

❍

Controle de fluxo

❍

Gerenciamento de conexão ❒ 3.6 Princípios do

controle de congestionamento ❒ 3.7 controle de

congestionamento TCP

Camada Transporte 3-3

Serviços e Protocolos de Transporte

❒ provêem entre executando hosts

processos comunicação em diferente

aplicativos

lógica

❒ Protocolos “rodam” em de end transporte systems ❍

aplicação transporte rede enlace física

rede enlace rede física enlace física

Lado mensagens em passados rede

segmentos emissor: à da camada aplicação

quebra

que são de

rede enlace física

rede enlace física

rede enlace física

❍

Lado segmentos e aplicação

os passa receptor: à em camada

mensagens remonta

aplicação transporte rede enlace física ❒ mais transporte aplicações

de um disponível protocolo de

para as

❍

Internet: TCP e UDP

Camada Transporte 3-4

CamadaTransporte vs. Camada de Rede

❒ Camada de rede:

comunicação lógica entre hosts ❒ Camada transporte:

comunicação lógica entre processos

❍

analogia: 12 crianças enviando

cartas à 12 crianças ❒ processos = crianças ❒ mensagens da aplic.= cartas em envelopes

Conta com (e melhora) serviços da camada de

❒ hosts = casas ❒ Protocolo de rede

transporte = Ana e Bill ❒ Protocolo da camada de rede = serviço postal

Camada Transporte 3-5

Protocolos da camada transporte da Internet

❒ Entrega confiável, em ordem

(TCP)

❍

aplicação transporte rede Controle de

enlace física

rede enlace congestionamento

rede física enlace

❍

Controle de fluxo

física

rede ❍

Estabelecimento de conexão

enlace física

rede ❒ Entrega não-confiável, sem

enlace física garantias de ordenação: UDP

rede enlace ❍

Extensões “sem

física ornamentos” ao serviço de melhor esforço (best- effort) IP

aplicação transporte rede enlace física ❒ serviços indisponíveis:

❍

Garantias de atraso

❍

Garantias de banda passante

Camada Transporte 3-6

Resumo do Módulo 3

❒ 3.1 Serviços da camada

transporte ❒ 3.2 Multiplexação e

demultiplexão ❒ 3.3 transporte não-

orientado à conexão: UDP ❒ 3.4 Princípios da transferência confiável de dados

❒ 3.5 transporte orientado

à conexão: TCP

❍

Estrutura do segmento

❍

Transferência de dados confiável

❍

Controle de fluxo

❍

Gerenciamento de conexão ❒ 3.6 Princípios do

controle de congestionamento ❒ 3.7 controle de

congestionamento TCP

Camada Transporte 3-7

Multiplexação/demultiplexação

Demultiplexação no host receptor:

Multiplexação no host emissor:

Entrega dos segmentos recebidos aos sockets corretos

aplicação

transporte

rede

enlace

física

Coletar dados dos vários sockets, adiciona cabeçalho aos dados (mais tarde usado para demultiplexação) = socket

= processo P3

P1 P1

aplicação

P2 P4

aplicação

transporte

transporte

rede

rede

enlace

enlace

física

física

host 1 host 2

host 3

Camada Transporte 3-8

Como a demultiplexação funciona

❒ host recebe datagramas IP

❍

cada datagrama possui endereço IP fonte, endereço IP de destino

❍

32 bits

**# porta de origem # porta de destino**

cada datagrama carrega 1 segmento da camada

outros campos do cabeçalho transporte

❍

cada segmento possui número de porta de origem e de porta de destino

Dados da aplicação ❒ host usa os endereços IP &

número das portas para enviar

(mensagem)

segmento ao socket adequado

Formato do segmento TCP/UDP

Camada Transporte 3-9

demultiplexação com UDP

❒ Criar sockets com portas

respectivas: DatagramSocket mySocket1 = new

DatagramSocket(99111); DatagramSocket mySocket2 = new

DatagramSocket(99222); ❒ Socket UDP identificado

pela tupla: (

endereço IP de destino, número da

porta de destino)

❒ Quando um host recebe um

segmento UDP:

❍

Verifica o número da porta de destino no segmento

❍

direciona o segmento UDP para o socket com o número da porta especificado ❒ Datagramas IP com endereço

IP fonte diferentes e/ou números de porta de origem diferentes são direcionados ao mesmo socket

Camada Transporte 3-10

demux com UDP (cont)

DatagramSocket serverSocket = new DatagramSocket(6428);

P2

P3

P1 P1

SP: 6428 DP: 9157

SP: 9157 cliente

DP: 6428 IP: A

servidor IP: C

cliente IP:B SP: 6428 DP: 5775

SP: 5775 DP: 6428

SP provê uma espécie de “endereço de retorno”

Camada Transporte 3-11 SP : source port (porta de origem) DP: destination port (porta de destino)

Demux com TCP

❒ Socket TCP identificado pela

tupla quádrupla:

❍

Endereço IP de origem

❍

Número da porta de origem

❍

Endereço IP de destino

❍

Número da porta de destino ❒ Host receptor usa todos

esses quatro valores para enviar o segmento ao socket apropriado

❒ host servidor pode suportar

diversos sockets TCP simultaneamente:

❍

cada socket é identificado por sua proópria tupla quádrupla ❒ Servidores Web possuem

sockets diferentes para cada cliente conectado

❍

HTTP não-persistente terá diferentes sockets para cada requisição

Camada Transporte 3-12

demux (cont) com TCP

P1

P4

P5 P6 P2 P1 P3

SP: 5775 DP: 80 S-IP: B D-IP:C

SP: 9157

SP: 9157 cliente

DP: 80 IP: A

S-IP: A D-IP:C

servidor IP: C

DP: 80

cliente IP:B D-IP:C S-IP: B

Camada Transporte 3-13

Demux com TCP: Web Server com Threads

P1

P4 P2

P1 P3

SP: 5775 DP: 80 S-IP: B D-IP:C

SP: 9157

SP: 9157 client

DP: 80 IP: A

S-IP: A D-IP:C

server IP: C

DP: 80

Client S-IP: B

IP:B D-IP:C

Camada Transporte 3-14

Resumo do Módulo 3

❒ 3.1 Serviços da camada

transporte ❒ 3.2 Multiplexação e

demultiplexão ❒ 3.3 transporte não-

orientado à conexão: UDP ❒ 3.4 Princípios da transferência confiável de dados

❒ 3.5 transporte orientado

à conexão: TCP

❍

Estrutura do segmento

❍

Transferência de dados confiável

❍

Controle de fluxo

❍

Gerenciamento de conexão ❒ 3.6 Princípios do

controle de congestionamento ❒ 3.7 controle de

congestionamento TCP

Camada Transporte 3-15

UDP: User Datagram Protocol

[RFC 768]

❒ Protocolo Internet de

transporte “sem ornamentos” e com “elementos básicos” ❒ Serviço “best effort”,

segmentos UDP podem ser:

❍

Por que existe o UDP? ❒ Sem estabelecimento de

conexão (que pode adicionar atraso) perdidos

❒ simples: sem estado de

❍

entregues fora de ordem à aplicação

conexão no emissor nem no receptor

❒ não-orientado à conexão:

❍

sem handshaking entre o emissor e receptor UDP

❒ Cabeçalho do segmento

pequeno ❒ nenhum controle de

congestionamento

❍

Cada segmento UDP é tratado de forma independente dos outros

Camada Transporte 3-16

UDP: mais ...

❒ Freqüentemente aplicações streaming

multimídia usado de

para

❍

Tolerante à perdas ❍

Sensível dados

à taxa de

❒ Outros usos do UDP

❍

DNS ❍

SNMP ❒ Transferência sobre adicionada aplicação

UDP: na confiabilidade

camada

confiável

❍

Recuperação específica aplicação!

da de camada erros

tamanho, em

bytes do segmento UDP,

incluindo o cabeçalho

32 bits

**# porta de origem # porta de destino**

tamanho checksum

Dados da aplicação (mensagem)

Formato do segmento UDP

Camada Transporte 3-17

Checksum UDP

Objetivo: detectar “erros” (e.g., bits trocados) no

segmento transmitido

Emissor: ❒ Trata o conteúdo de

segmentos como uma seqüência de inteiros de 16 bits ❒ checksum: adição (soma

complemento 1) do conteúdo do segmento ❒ Emissor coloca o valor do

checksum no campo checksum do UDP

Receptor: ❒ computa o checksum do

segmento recebido ❒ Verifica se o checksum

computado bate com o valor informado no campo checksum:

❍

Não - erro detectado

❍

Sim - nenhum erro detectado. Mas pode haver erros? Mais em breve ...

Camada Transporte 3-18

Exemplo de Checksum Internet

❒ Nota

❍

Ao adicionar números, não esquecer do “vai um”

❒ Examplo: adição de dois inteiros de 16 bits

**1 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1**

wraparound

**1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1**

soma

**1 1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0 checksum**

**1 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1**

Camada Transporte 3-19

Resumo do Módulo 3

❒ 3.1 Serviços da camada

transporte ❒ 3.2 Multiplexação e

demultiplexão ❒ 3.3 transporte não-

orientado à conexão: UDP ❒ 3.4 Princípios da transferência confiável de dados

❒ 3.5 transporte orientado

à conexão: TCP

❍

Estrutura do segmento

❍

Transferência de dados confiável

❍

Controle de fluxo

❍

Gerenciamento de conexão ❒ 3.6 Princípios do

controle de congestionamento ❒ 3.7 controle de

congestionamento TCP

Camada Transporte 3-20

Princípios da transferência confiável de dados

❒ importante nas camadas aplicação, transporte e enlace ❒ Está na lista dos 10 tópicos mais importantes em redes!

❒ características do canal não confiável determinará a

complexidade do protocolo de transferência confiável (rdt)

Camada Transporte 3-21

Princípios da transferência confiável de dados

❒ importante nas camadas aplicação, transporte e enlace ❒ Está na lista dos 10 tópicos mais importantes em redes!

❒ características do canal não confiável determinará a

complexidade do protocolo de transferência confiável (rdt)

Camada Transporte 3-22

Princípios da transferência confiável de dados

❒ importante nas camadas aplicação, transporte e enlace ❒ Está na lista dos 10 tópicos mais importantes em redes!

❒ características do canal não confiável determinará a

complexidade do protocolo de transferência confiável (rdt)

Camada Transporte 3-23

rdt\_send(): chamado pela camada superior, (e.g., aplic.). Passagem de dados da camada superior

udt\_send(): chamado por rdt para transferir pacote pelo canal não-confiável ao receptor

Transferência confiável de dados: introdução ...

lado emissor

deliver\_data(): chamado pelo rdt para enviar dados à camada superior

rdt\_rcv(): chamado quando pacote chega no lado receptor do canal

Camada Transporte 3-24

lado receptor

Transferência confiável de dados: introdução ... Iremos: ❒ Desenvolver incrementalmente os lados receptor e emissor do protocolo de transferência confiável de dados (rdt) ❒ considere somente transferência de dados unidirecional

❍

mas informações de controle transitarão em ambos sentidos! ❒ usar máquinas de estado finitas (FSM) para especificar

o emissor e o receptor

estado 1

Evento causando transição de estado Ações tomadas na transição de estados

estado: quando estiver neste “estado”, próximo estado é unicamente determinado pelo próximo evento

evento ações

Camada Transporte 3-25

estado 2

Rdt1.0: transferência confiável sobre um canal confiável ❒ Canal de base perfeitamente confiável

❍

Nenhum erro nos bits

❍

Nenhuma perda de pacotes ❒ FSMs separadas para emissor e receptor:

❍

Emissor envia dados no canal

❍

Receptor lê dados do canal

Wait for call from above

rdt\_send(data)

Wait for

rdt\_rcv(packet)

packet = make\_pkt(data) udt\_send(packet)

call from below

extract (packet,data) deliver\_data(data)

emissor receptor

Camada Transporte 3-26

Rdt2.0: canal com erros em bits

❒ Canal de base pode trocar bits dos pacotes

❍

checksum para detectar erros nos bits ❒ a questão: como se recuperar de erros:

❍

acknowledgements (ACKs): receptor informa explicitamente ao emissor que pacote recebido está OK

❍

negative acknowledgements (NAKs): receptor informa explicitamente ao emissor que o pacote possui erros

❍

Emissor retransmite pacote ao receber um NAK ❒ novos mecanismos no rdt2.0 (além de rdt1.0):

❍

Detecção de erro

❍

Feedback do receptor: msgs de controle (ACK,NAK) do recptor para o emissor

Camada Transporte 3-27

rdt2.0: especificação FSM

rdt\_send(data) snkpkt = make\_pkt(data, checksum) udt\_send(sndpkt)

rdt\_rcv(rcvpkt) &&

Wait for call from above

receptor

Wait for

isNAK(rcvpkt)

ACK or NAK

rdt\_rcv(rcvpkt) && udt\_send(sndpkt)

corrupt(rcvpkt)

udt\_send(NAK)

rdt\_rcv(rcvpkt) && isACK(rcvpkt)

Wait for call from emissor

below rdt\_rcv(rcvpkt) &&

notcorrupt(rcvpkt) extract(rcvpkt,data) deliver\_data(data) udt\_send(ACK)

Camada Transporte 3-28

rdt2.0: operação na ausência de erros

rdt\_send(data) snkpkt = make\_pkt(data, checksum) udt\_send(sndpkt)

rdt\_rcv(rcvpkt) &&

Wait for

Wait for

isNAK(rcvpkt)

call from

ACK or above

NAK

rdt\_rcv(rcvpkt) && udt\_send(sndpkt)

corrupt(rcvpkt)

udt\_send(NAK)

rdt\_rcv(rcvpkt) && isACK(rcvpkt)

Wait for call from below

rdt\_rcv(rcvpkt) &&

notcorrupt(rcvpkt) extract(rcvpkt,data) deliver\_data(data) udt\_send(ACK)

Camada Transporte 3-29

rdt2.0: cenário na presença de erros

rdt\_send(data) snkpkt = make\_pkt(data, checksum) udt\_send(sndpkt)

rdt\_rcv(rcvpkt) &&

Wait for

Wait for

isNAK(rcvpkt)

call from

ACK or above

NAK

rdt\_rcv(rcvpkt) && udt\_send(sndpkt)

corrupt(rcvpkt)

udt\_send(NAK)

rdt\_rcv(rcvpkt) && isACK(rcvpkt)

Wait for call from below

rdt\_rcv(rcvpkt) &&

notcorrupt(rcvpkt) extract(rcvpkt,data) deliver\_data(data) udt\_send(ACK)

Camada Transporte 3-30

rdt2.0 tem um problema fatal!

O que acontece se um

ACK/NAK é corrompido? ❒ emissor não sabe o que aconteceu no receptor! ❒ Não pode somente

retransmitir: possibilidade de duplicação de pacotes

Tratando duplicações: ❒ Emissor retransmitepacote

atual se ACK/NAK é corrompido ❒ Emissor adiciona números de

seqüência a cada pacote ❒ Receptor descarta (não

entrega para a camada superior) pacotes duplicados

stop and wait Emissor envia 1 pacote e então aguarda pela Resposta do receptor

Camada Transporte 3-31

rdt2.1: emissor trata ACK/NAKs corrompidos

rdt\_send(data) sndpkt = make\_pkt(0, data, checksum) udt\_send(sndpkt)

rdt\_rcv(rcvpkt) &&

Wait for call 0 from

Wait for ACK or above

( corrupt(rcvpkt) || isNAK(rcvpkt) )

NAK 0

sndpkt = make\_pkt(1, data, checksum) udt\_send(sndpkt)

udt\_send(sndpkt)

rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)

Wait for call 1 from above

rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt) Λ

udt\_send(sndpkt)

Λ Wait for

rdt\_rcv(rcvpkt) &&

ACK or NAK 1 ( corrupt(rcvpkt) || isNAK(rcvpkt) )

rdt\_send(data)

Camada Transporte 3-32

rdt2.1: receptor trata ACK/NAKs corrompidos

rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt)

&& has\_seq0(rcvpkt)

extract(rcvpkt,data) deliver\_data(data) sndpkt = make\_pkt(ACK, chksum)

rdt\_rcv(rcvpkt) && (corrupt(rcvpkt)

udt\_send(sndpkt)

rdt\_rcv(rcvpkt) && (corrupt(rcvpkt) sndpkt = make\_pkt(NAK, chksum)

sndpkt = make\_pkt(NAK, chksum) udt\_send(sndpkt)

udt\_send(sndpkt) Wait for 0 from below

rdt\_rcv(rcvpkt) &&

not corrupt(rcvpkt) && has\_seq0(rcvpkt) sndpkt = make\_pkt(ACK, chksum) udt\_send(sndpkt)

rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt)

sndpkt = make\_pkt(ACK, chksum) udt\_send(sndpkt)

&& has\_seq1(rcvpkt)

extract(rcvpkt,data) deliver\_data(data) sndpkt = make\_pkt(ACK, chksum) udt\_send(sndpkt)

Wait for rdt\_rcv(rcvpkt) &&

1 from not corrupt(rcvpkt) &&

below has\_seq1(rcvpkt)

Camada Transporte 3-33

rdt2.1: discussão

Emissor: ❒ # de seq. adicionado a

pkt ❒ 2 #‟s de seq. (0,1) são suficientes. Por que? ❒ Deve verificar se

ACK/NAK recebido está corrompido ❒ 2x mais estados

❍

Receptor: ❒ Deve verificar se

pacote recebido é um duplicado

❍

estado indica se # de seq. do pacote esperado é 0 ou 1 ❒ nota: receptor não

pode saber se seu último ACK/NAK foi state deve “recordar” se pacote “atual” possui # de seq. 0 ou 1

recebido OK no emissor

Camada Transporte 3-34

rdt2.2: um protocolo sem NAKs

❒ Igual ao rdt2.1 mas usando ACKs somente ❒ Ao invés de NAK, receptor envia ACK do último

pacote recebido corretamente (OK)

❍

Receptor deve incluir explicitamente o # de seq. do pacote sendo confirmado ❒ ACK duplicado no emissor resulta na mesma ação

como para o NAK: retransmição do pacote atual

Camada Transporte 3-35

rdt2.2: fragmentos do emissor e receptor

rdt\_send(data) sndpkt = make\_pkt(0, data, checksum) udt\_send(sndpkt)

rdt\_rcv(rcvpkt) &&

Wait for call 0 from above

Wait for ACK

( corrupt(rcvpkt) ||

**isACK(rcvpkt,1) ) 0**

udt\_send(sndpkt) Fragmento da FSM do emissor

rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt) rdt\_rcv(rcvpkt) &&

&& isACK(rcvpkt,0) (corrupt(rcvpkt) ||

**has\_seq1(rcvpkt))**

**udt\_send(sndpkt)**

Wait for 0 from below

fragmento da FSM Do receptor

rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt)

&& has\_seq1(rcvpkt) extract(rcvpkt,data) deliver\_data(data) sndpkt = make\_pkt(ACK1, chksum) udt\_send(sndpkt)

Camada Transporte 3-36

rdt3.0: canais com erros e perdas

Nova hipótese: canal de base

pode agora perder pacotes (dados ou ACKs)

❍

Abordagem: emissor aguarda um

tempo “razoável” a recepção de ACKs checksum, # de seq.,

❒ retransmite se nenhum ACK é ACKs, retransmissões

recebido neste tempo ajudarão mas não serão suficientes

❒ se pkt (ou ACK) apenas atrasado

(não foi perdido):

❍

retransmissão causará duplicação, mas uso de # de seq. tratam isso

❍

receptor deve especificar # de seq. do pacote sendo confirmado ❒ requer temporizador

Camada Transporte 3-37

Emissor rdt3.0

rdt\_send(data) sndpkt = make\_pkt(0, data, checksum) udt\_send(sndpkt) start\_timer

Wait for ACK0

rdt\_rcv(rcvpkt) && ( corrupt(rcvpkt) || isACK(rcvpkt,1) ) rdt\_rcv(rcvpkt)

Λ Λ

Wait for call 0from above rdt\_rcv(rcvpkt)

Wait for call 1 from above

sndpkt = make\_pkt(1, data, checksum) udt\_send(sndpkt) start\_timer

timeout udt\_send(sndpkt) start\_timer

&& notcorrupt(rcvpkt)

rdt\_rcv(rcvpkt) && isACK(rcvpkt,1)

&& notcorrupt(rcvpkt) stop\_timer

&& isACK(rcvpkt,0) stop\_timer

timeout

Wait for udt\_send(sndpkt)

ACK1 start\_timer

Λ rdt\_rcv(rcvpkt) && ( corrupt(rcvpkt) || isACK(rcvpkt,0) ) Λ

rdt\_rcv(rcvpkt)

rdt\_send(data)

Camada Transporte 3-38

rdt3.0 em ação

Camada Transporte 3-39

rdt3.0 em ação

Camada Transporte 3-40

Desempenho do rdt3.0

❒ rdt3.0 funciona, mas desempenho é ruim ❒ examplo: enlace de 1 Gbps, 15 ms de atraso de propagação fim-a-fim, pacote de 1KB:

T

transmit

=

R L (tamanho (taxa de transmissão, do pacote bits)

bps)

=

8kb/pkt 10\*\*9 b/sec

❍

= 8 microsec

❍

U ocupado sender

: utilização enviando

: fração do tempo em que o emissor está

U

sender

=

L / R RTT + L / R

=

30.008 .008

= 0.00027 microsec Pacote de 1KB a cada 30 msec -> 33kB/sec de vazão em um enlace de 1 Gbps !

❍

O protocolo de rede limita o uso de recursos físicos!

Camada Transporte 3-41

= 0.027%

rdt3.0: funcionamento do stop-and- wait

emissor receptor

Primeiro bit do pacote transmitido, t = 0 Último bit do pkt transmitido, t = L / R

Primeiro bit do pacote chega RTT

Último bit do pacote chega, ACK envidado

ACK chega, envia próximo pacote, t = RTT + L / R

U

sender

=

L / R RTT + L / R

= 30.008

.008

= 0.00027

= 0.027% microsec

Camada Transporte 3-42

Protocolos com Pipeline

Pipelining: emissor permite múltiplos pacotes

enviados e que ainda não foram confirmados

❍

range do número de seqüência deve ser aumentado

❍

“bufferização” no emissor e/ou receptor

❒ 2 formas genéricas de protocolos pipeline: go-Back-N,

selective repeat (SR)

Camada Transporte 3-43

Pipelining: aumento da utilização

emissor receptor

Primeiro bit do pkt transmitido, t = 0 Último bit transmitido, t = L / R

Primeiro bit do pacote chega RTT

Último bit do pacote chega, envia ACK

ACK chega, envia próximo pacote, t = RTT + L / R

Último bit do 2o pacote chega, envia ACK Último bit do 3o pacote chega, envia ACK

U

sender

Utilização aumenta de um fator de 3 (neste exemplo) !

=

3 \* L / R RTT + L / R

=

30.008 .024

= 0.0008

microsecon

Camada Transporte 3-44

= 0.08%

Utilizáveis, mas não enviados

Não utilizáveis no momento

Já confirmados

Enviados mas não confirmados

❒ ACK(n): confirma todos os pacotes até o # n - “ACK cumulativo”

❍

Pode receber ACKs duplicados ACKs (veja receptor) ❒ Temporizador para cada pacote enviado ❒ timeout(n): retransmite pkt n e todos pacotes com #s de seq.

maiores que estão dentro da janela

Go-Back-N

Emissor: ❒ # de seq. de k bits no cabeçalho do pacote ❒ “janela” de até N consecutivos pacotes não confirmados permitida

Camada Transporte 3-45

GBN: FSM do emissor estendida

rdt\_send(data)

if (nextseqnum < base+N) {

sndpkt[nextseqnum] = make\_pkt(nextseqnum,data,chksum) udt\_send(sndpkt[nextseqnum]) if (base == nextseqnum)

start\_timer nextseqnum++ } Λ base=1

else

refuse\_data(data)

nextseqnum=1

Wait

rdt\_rcv(rcvpkt)

&& corrupt(rcvpkt)

base = getacknum(rcvpkt)+1 If (base == nextseqnum)

Somente ACKs são usados (sem NAKs)

stop\_timer else

start\_timer

timeout start\_timer udt\_send(sndpkt[base]) udt\_send(sndpkt[base+1]) ... udt\_send(sndpkt[nextseqnum-1])

rdt\_rcv(rcvpkt) &&

notcorrupt(rcvpkt)

Camada Transporte 3-46

GBN: FSM do receptor estendida

ACK-only: sempre envia ACK para pacote recebido “na ordem” corretamente com # de seq. mais elevado

❍

default udt\_send(sndpkt)

rdt\_rcv(rcvpkt)

&& notcurrupt(rcvpkt) Λ

&& hasseqnum(rcvpkt,expectedseqnum)

expectedseqnum=1

Wait

extract(rcvpkt,data) sndpkt =

deliver\_data(data) make\_pkt(expectedseqnum,ACK,chksum)

sndpkt = make\_pkt(expectedseqnum,ACK,chksum) udt\_send(sndpkt) expectedseqnum++

Pode gerar ACKs duplicados

❍

Precisa recordar somente o # de seq. esperado (expectedseqnum) ❒ Pkt fora de ordem:

❍

descarta (não armazena) -> receptor não possui buffer!

❍

Re-ACK pkt com o # de seq. mais elevado e em ordem

Camada Transporte 3-47

Tamanho da janela = 4

GBN em ação

Camada Transporte 3-48

SR - Selective Repeat

❒ Receptor confirma individualmente cada pkt

recebido corretamente

❍

armazena pkts, quando necessário, para eventual envio de pacotes ordenados à camada superior ❒ Emissor reenvia somente pkts para os quais o

ACK não foi recebido

❍

Emissor possui temporizador para cada pacote não confirmado (unACKed pkt) ❒ Janela do emissor

❍

N consecutivos #s de seq.

❍

Novamente limita #s de seq. de enviados, pkts não confirmados (unACKed pkts)

Camada Transporte 3-49

Selective repeat: janelas do emissor e receptor

Utilizáveis, Já confirmados

mas não enviados

Enviados mas não confirmados

Não utilizáveis no momento

Números de seq. do ponto de vista do emissor

Fora de ordem (armazenados), mas já confirmados

Sendo aguardados

Números de seq. do ponto de vista do receptor

Aceitáveis (dentro da janela)

Não utilizáveis no momento

Camada Transporte 3-50

Selective repeat

Dados superior da camada

: ❒ Se disponível envia próximo pkt

dentro # de seq.

da janela,

timeout(n): ❒ reenvia temporizador

pkt n, reinicia

ACK(n) no intervalo

[sendbase,sendbase+N]: ❒ marca pkt n como recebido ❒ se pacotes avança o confirmado

próximo n é o base menor não # confirmados, da de # janela seq. de não

seq. para

de emissor

receptor pkt n no intervalo

[rcvbase, rcvbase+N-1] ❒ envia ACK(n) ❒ For a de ordem: buffer ❒ Em ordem: entrega

(incluindo pkts ordenados no buffer), avança janela para o próximo pkt ainda não recebido pkt n no intervalo

[rcvbase- N,rcvbase-1] ❒ ACK(n) Caso contrário: ❒ ignore

Camada Transporte 3-51

Selective repeat em ação

Camada Transporte 3-52

Selective repeat:

dilema

Exemplo: ❒ #s de seq.: 0, 1, 2, 3 ❒ Tamanho da janela =3

❒ Receptor não vê

diferença nos dois cenários! ❒ Incorretamente passa dados duplicados como sendo novos no exemplo (a)

Q: qual a relação entre o

o # de seq. máximo e o tamanho da janela?

Camada Transporte 3-53

Resumo do Módulo 3

❒ 3.1 Serviços da camada

transporte ❒ 3.2 Multiplexação e

demultiplexão ❒ 3.3 transporte não-

orientado à conexão: UDP ❒ 3.4 Princípios da transferência confiável de dados

❒ 3.5 transporte orientado

à conexão: TCP

❍

Estrutura do segmento

❍

Transferência de dados confiável

❍

Controle de fluxo

❍

Gerenciamento de conexão ❒ 3.6 Princípios do

controle de congestionamento ❒ 3.7 controle de

congestionamento TCP

Camada Transporte 3-54

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581 ❒ ponto-a-ponto:

❍

Um emissor, um receptor ❒ Stream de bytes confiável e

em ordem:

❒ Dados full duplex:

❍

❍

Fluxo de dados bidirecional

“mensagens não possuem limites”

sobre a mesma conexão

❍

❒ pipelined:

❍

MSS: Maximum Segment Size (tamanho máximo de segmento) Controle de fluxo e congestionamento do TCP setam tamanho da janela

❒ Orientado ❍

à conexão:

❒ Buffers nos emissor e

receptor

socket door

TCP send buffer

handshaking (troca de msgs de controle) inicia emissor e estado do receptor antes da troca de dados ❒ Fluxo controlado:

❍

Emissor não vai enviar mais

application

application

dados que o receptor possa writes data

reads data

socket

receber door

segment

TCP receive buffer

Camada Transporte 3-55

Estrutura do segmento TCP

URG: dados urgentes

32 bits Contagem (geralmente não usado)

**# da porta do emissor # da porta do destino**

Dados da aplicação (tamanho variável)

por bytes de dados ACK: # de ACK

(não por válido

segmentos!) PSH: passar dados imediatamente para

a camada superior (geralmente não usado)

RST, SYN, FIN: estabelecimento

de conexão (comandos setup,

teardown)

Internet checksum (como no UDP)

número de seqüência número confirmado (ACK)

cabç. tam.

usado não

U

A P R S F Janela do receptor checksum Urg data pnter Opções (tamanho variável)

Camada Transporte 3-56

# de bytes receptor está desejando receber

# de seq. e ACKs noTCP

#s de seq:

❍

byte stream “number” of first byte in segment‟s data ACKs:

❍

Host A Host B

Usuário digita „C‟

host confirma (ACK) recepção do caracter „C‟ ecoado

host confirma

# de seq do

(ACK) recepção de próximo byte

„C‟, ecoa „C‟ aguardado pelo

de volta outro lado

❍

ACK cumulativo Q: como receptor trata segmentos fora de ordem?

❍

R: especificação TCP não diz –

tempo Cenário com telnet

deixada para o desenvolvedor

Camada Transporte 3-57

Round Trip Time (RTT) e Timeout no TCP

Q: como setar o valor do timeout do TCP? ❒ maior que o RTT

❍

Q: como estimar o RTT? ❒ SampleRTT: tempo medido a

partir da transmissão do segmento até a recepção do ACK mas o RTT varia

❍ ❒ Muito curto:

timeout prematuro

❍

ignora retransmissões ❒ SampleRTT variará, deseja-se

uma estimativa de RTT Retransmissões

“suavizada”

desnecessárias

❍

❒ Muito longo: reação lenta à perda de segmentos

Faz-se uma média de várias medidas recentes, não se usa somente a atual (SampleRTT)

Camada Transporte 3-58

Round Trip Time (RTT) e Timeout no TCP

**EstimatedRTT = (1- α)\*EstimatedRTT + α\*SampleRTT**

❒ Exponential Weighted Moving Average (EWMA)

❒ Influência de amostras do passado decresce

exponencialmente de forma rápida

❍

Peso para um dado SampleRTT decai exponencialmente rápido quando atualizações (novos SampleRTTs) são feitas

❍

Maior peso para amostras recentes

❒ Valor típico: α = 0.125

Camada Transporte 3-59

Exemplo de estimação do RTT :

**RTT: gaia.cs.umass.edu to fantasia.eurecom.fr**

350

300

250

200

150

100

1 8 15 22 29 36 43 50 57 64 71 78 85 92 99 106

**time (seconnds)**

**R**

**T**

**T**

**(**

**m**

**i**

**l**

**l**

**i**

**s**

**e**

**c**

**o**

**n**

**d**

**s**

**)**

SampleRTT Estimated RTT

Camada Transporte 3-60

Round Trip Time (RTT) e Timeout no TCP Setando o timeout

❒ EstimtedRTT + “margem segura”

❍

Quanto maior a variação no RTT estimado (EstimatedRTT) -> maior a margem de segurança ❒ Primeira estimativa de quanto a amostra do RTT

(SampleRTT) DevRTT = (1-β)\*DevRTT desvia do RTT estimado +

(EstimatedRTT):

**β\*|SampleRTT-EstimatedRTT|**

**(tipicamente, β = 0.25)**

Em seguida, o intervalo de timeout é setado para:

**TimeoutInterval = EstimatedRTT + 4\*DevRTT**

Camada Transporte 3-61

Resumo do Módulo 3

❒ 3.1 Serviços da camada

transporte ❒ 3.2 Multiplexação e

demultiplexão ❒ 3.3 transporte não-

orientado à conexão: UDP ❒ 3.4 Princípios da transferência confiável de dados

❒ 3.5 transporte orientado

à conexão: TCP

❍

Estrutura do segmento

❍

Transferência de dados confiável

❍

Controle de fluxo

❍

Gerenciamento de conexão ❒ 3.6 Princípios do

controle de congestionamento ❒ 3.7 controle de

congestionamento TCP

Camada Transporte 3-62

Transferência confiável de dados no TCP

❒ TCP cria serviço de transf. confiável de dados no topo do serviço não-confiável do IP ❒ Segmentos “Pipelined” ❒ Acks cumulativos ❒ Conceitualmente o TCP

usa múltiplos temporizadores de transmissão

❒ Retransmissões são

disparadas por:

❍

Eventos de expiração (timeout)

❍

Acks duplicados ❒ Inicialmente considere

um emissor TCP simplificado:

❍

ignore acks duplicados

❍

ignore controle de fluxo, ignore controle de congestionamento

Camada Transporte 3-63

TCP: transferência confiável de dados

evento: dado recebido da camada superior

emissor simplificado, supondo:

cria, envia segmento

wait for

evento: expiração do tempor. do seg. com # de seq. y

event

retransmite segmento

evento: ACK recebido, com ACK # = y

processamento do ACK

•fluxo de dados uni-direcional

•sem controle de fluxo, congestionamento wait for event

Camada Transporte 3-64

TCP: transfe- rência confiável de dados

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 \*/

Camada Transporte 3-65

Eventos no emissor TCP:

dados receb. da aplic.: ❒ Cria segmento com #

de seq. ❒ # de seq. é o número

do primeiro byte de dados no segmento ❒ Inicia temporizador se

já não estiver executando ❒ Intervalo de expiração: TimeOutInterval

timeout: ❒ retransmite segmento que

causou o timeout ❒ reinicia temporizador

Ack recebido: ❒ Se confirma segmentos

prévios não-confirmados

❍

Atualiza o que é conhecido a ser confirmado

❍

Inicia temporizador se existem segmentos devidos

Camada Transporte 3-66

TCP: cenários com retransmissão

Host B

Host A

Host B

t

X

S

t

SendBase = 100

Timeout prematuro tempo

Cenário com ACK perdido

tempo

t

i

m

e

o

u

Host A

perda

Sendbase = 100

SendBase = 120

SendBase = 120

e

q

=

9

2

t

i

m

e

o

u

t

S

e

q

=

9

2

t

i

m

e

o

u

Camada Transporte 3-67

TCP: + cenários com retransmissão

Host A

perda

Host B

X

SendBase = 120

tempo

t

Cenário com ACK cumulativo

t

i

m

e

o

u

Camada Transporte 3-68

Geração de ACK no TCP

[recomendações na RFC 1122 e na RFC 2581] Evento no Receptor

Chegada “na ordem” de segmento com esperado # de seq. Todos os dados até o # de seq. esperado já confirmados. Sem lacunas

Chegada “na ordem” de segmento com # de seq. esperado. Um outro seg. “em ordem” aguardando transmissão de ACK Chegada de segmento fora de ordem com # de seq. > que esperado. Lacuna detectada

Chegada de segmento que parcialmente ou totalmente preenche a lacuna

Ação do Receptor TCP

Atrasar ACK. Aguarde até 500ms pelo próximo segmento. Se não chegar, envie ACK

Enviar imediatamente ACK cumulativo único, confirmando ambos segmentos “na ordem”

Enviar imediatamente ACK (duplicado), indicando # de seq. do próximo byte esperado

Enviar imediatamente ACK, dado que segmento inicia no princípio da lacuna

Camada Transporte 3-69

Fast Retransmit – Retransmissão Rápida

❒ Período de expiração

relativamente longo:

❍

❒ Se emissor recebe 3 duplicados ACKs (4 Grande atraso antes de

ACKS reenviar pacote perdido

consecutivos/idênticos/ ❒ Detecta segmentos

sem outros pacotes no perdidos via ACKs

meio) para o mesmo duplicados.

dado, ele supõe que

❍

Emissor freqüentemente envia

segmento após dado confirmado foi perdido: muitos segmentos sucessivos

❍

❍

fast retransmit: reenvia

Se segmento é perdido, haverá provavelmente

segmento antes do temporizador expirar

muitos ACKs duplicados.

Camada Transporte 3-70

Algoritmo do Fast retransmit:

evento: ACK recebido, com campo ACK de valor igual a y

if (y > SendBase) {

SendBase = y if (há atualmente segmentos ainda não confirmados)

iniciar temporizador } else {

incrementar contador de ACKs duplicados recebidos para y if (contador de ACKs duplicados recebidos para y = 3) {

reenviar segmento com # de seq. y }

Um ACK duplicado para segmento já confirmado

fast retransmit

Camada Transporte 3-71

Resumo do Módulo 3

❒ 3.1 Serviços da camada

transporte ❒ 3.2 Multiplexação e

demultiplexão ❒ 3.3 transporte não-

orientado à conexão: UDP ❒ 3.4 Princípios da transferência confiável de dados

❒ 3.5 transporte orientado

à conexão: TCP

❍

Estrutura do segmento

❍

Transferência de dados confiável

❍

Controle de fluxo

❍

Gerenciamento de conexão ❒ 3.6 Princípios do

controle de congestionamento ❒ 3.7 controle de

congestionamento TCP

Camada Transporte 3-72

Controle de FluxoTCP

Controle de fluxo

❒ Lado receptor da

conexão TCP possui um

Emissor não estoura o buffer do receptor transmitindo demais buffer receptor:

ou muito rápido

❒ serviço de “casamento”

de taxas: “casar” a taxa de envio com a taxa de drenagem de

❒ Processo aplicação

pode ser lento ao ler

dados da aplicação receptora

dados do buffer

Camada Transporte 3-73

Controle de fluxoTCP: funcionamento

(Suponha que receptor TCP

descarte segmentos for a de ordem) ❒ espaço livre no buffer = RcvWindow = RcvBuffer-[LastByteRcvd -

**LastByteRead]**

❒ Recept. informa espaço

livre através da inclusão nos segmentos do valor da janela de recepção RcvWindow ❒ Emissor limita dados

não confirmados ao tamanho da janela RcvWindow

❍

guarante que buffer do receptor não “transborda”

Camada Transporte 3-74

Resumo do Módulo 3

❒ 3.1 Serviços da camada

transporte ❒ 3.2 Multiplexação e

demultiplexão ❒ 3.3 transporte não-

orientado à conexão: UDP ❒ 3.4 Princípios da transferência confiável de dados

❒ 3.5 transporte orientado

à conexão: TCP

❍

Estrutura do segmento

❍

Transferência de dados confiável

❍

Controle de fluxo

❍

Gerenciamento de conexão ❒ 3.6 Princípios do

controle de congestionamento ❒ 3.7 controle de

congestionamento TCP

Camada Transporte 3-75

Gerenciamento de conexão TCP Lembre-se: estabelecimento de conexão entre o emissor e receptor TCP antes da troca de segmentos de dados ❒ inicializar variáveis TCP :

❍

Three-way handshake:

Passo 1: host cliente envia

segmento SYN TCP para o servidor

#s de seq.

❍

buffers, informação de

❍

especifica # de seq. inicial

❍

controle de fluxo (e.g. RcvWindow) ❒ cliente: iniciador da

sem dados

Passo 2: host servidor recebe

SYN, responde com segmento SYNACK

conexão

**❍ Socket clientSocket = new Socket("hostname","port**

number"); ❒ servidor: contactado pelo

**cliente Socket connectionSocket = welcomeSocket.accept();**

servidor aloca buffers

❍

especifica # de seq. inicial do servidor Passo 3: cliente recebe SYNACK,

responde com segmento ACK que pode conter dados

Camada Transporte 3-76

Gerenciamento de ConexãoTCP (cont.)

Fechando uma conexão:

cliente fecha socket:

**clientSocket.close();**

Passo 1: cliente envia

segmento TCP de controle FIN ao servidor

Passo 2: servidor recebe

FIN, responde com ACK. Fecha conexão, envia FIN.

cliente servidor

fechar

fechar

fechado

E

s

p

e

r

a

t

e

m

p

o

r

i

z

a

d

a

Camada Transporte 3-77

Gerenciamento de Conexão TCP (cont.)

Passo 3: cliente recebe FIN,

responde com ACK.

❍

cliente servidor

fechar Entra na “espera temporizada” – responderá com ACK aos FINs recebidos

fechar

Passo 4: servidor, recebe

ACK. Conexão fechada.

fechado

E

s

p

e

r

a

t

e

m

p

o

r

i

z

a

d

a

Camada Transporte 3-78

Ciclo de vida do cliente TCP

Gerenciamento de Conexão TCP (cont.)

Camada Transporte 3-79

Ciclo de vida do servidor TCP

Resumo do Módulo 3

❒ 3.1 Serviços da camada

transporte ❒ 3.2 Multiplexação e

demultiplexão ❒ 3.3 transporte não-

orientado à conexão: UDP ❒ 3.4 Princípios da transferência confiável de dados

❒ 3.5 transporte orientado

à conexão: TCP

❍

Estrutura do segmento

❍

Transferência de dados confiável

❍

Controle de fluxo

❍

Gerenciamento de conexão ❒ 3.6 Princípios do

controle de congestionamento ❒ 3.7 controle de

congestionamento TCP

Camada Transporte 3-80

Princípios do Controle de Congestionamento

Congestionamento: ❒ informalmente: “fontes demais enviando dados

demais, muito rápido ultrapassando a capacidade da rede” ❒ diferente de controle de fluxo! ❒ manifestações:

❍

Pacotes perdidos (“estouro” de buffer nos roteadores)

❍

Atrasos elevados (“enfileiramento” em buffers nos roteadores) ❒ Problema na lista dos top-10!

Camada Transporte 3-81

Causas/custos do congestionamento: cenário 1

Enlace de capacidade C

❒ 2 receptores

emissores, 2

Host A

λ

in

❒ 1 infinitos

roteador, buffers

❒ sem retransmissões

❒ Atraso muito quando congestionada elevado rede (delay)

❒ Vazão

(throughput) máxima alcançável

λ

out

Em bytes/segundo

:

dado original

Host B

Buffers ilimitados do enlace de saída compartilhado

Camada Transporte 3-82

Causas/custos do congestionamento: cenário 2

❒ 1 roteador, buffers finitos ❒ emissor: retransmissão de pacote perdido

Host A

λ

in

λ

out

λ'

in

Buffers finitos de enlace de saída compartilhado : dado original

Host B

: dado original, mais dado retransmitido

Camada Transporte 3-83

λ

o

u

Causas/custos do congestionamento: cenário 2 ❒ ❒ ❒ (a) (b) (c) sempre: Retransmissão Retransmissões λ

in

=

λ de out “perfeitas” pacotes (goodput)

atrasados somente (não quando perdidos) perda:

faz λ

in

>

maior (que caso perfeito) para mesmo

λ

out

λ

in

R/2

R/2

λ

in

λ

in t

R/2

R/2

a. “custos” do congestionamento: ❒ (b) mais trabalho (retransmissões) para dada “goodput” ❒ (c) Retransmissões desnecessárias: enlace carrega múltiplas

cópias do mesmo pacote

λ

o

u

R/3

t

b.

λ

o

u

R/2

R/4

λ

in

R/2

Camada Transporte 3-84

t

c.

λ

out

Causas/custos do congestionamento: cenário 3

❒ 4 emissores ❒ Caminhos com múltiplos saltos ❒ timeout/retransmissão

Q: quando aumentam o que acontece

λ

in

?

e

λ

in

Host Host A A

λ

in

buffers finitos de enlace de saída compartilhado

R1

R2

R3

λ

out

λ'

in

: dado original

Host B

: dado original, mais dado retransmitido

Host Host B

D

Host C R4

Camada Transporte 3-85

Causas/custos do congestionamento: cenário 3

Um outro “custo” do congestionamento: ❒ Quando pacote descartado, a capacidade de

transmissão “upstream” usada para este pacote foi desperdiçada!

H o s t A

H o s t B

Camada Transporte 3-86

λ o u t

Abordagens para o controle de congestionamento

2 abordagens amplas para o controle de

congestionamento: Controle de

congestionamento fim- à-fim: ❒ Sem feedback explícito da

rede ❒ Congestionamento inferido pelos end-systems através das perdas e atrasos observados ❒ Abordagem usada pelo TCP

Controle de congestionamento

assistido pela rede

(Network- assisted congestion control)

: ❒ roteadores provêem feedback

para os end systems

❍

bit único indicando congestionamento (SNA, DECbit, TCP/IP ECN, ATM)

❍

Taxa explícita que emissor deve usar

Camada Transporte 3-87

Estudo de caso: Controle de congestionamento ABR da rede ATM

ABR: available bit rate: ❒ “serviço elástico” ❒ Se caminho emissor-

receptor estiver “sub- utilizado” (underloaded):

❍

Células RM (resource

management): ❒ Enviadas pelo emissor,

intercaladas com células de dados Emissor deve usar

❒ bits na célula RM cell setados banda disponível

pelos comutadores (“network- ❒ Se caminho emissor-

assisted”) receptor estiver

❍ congestionado:

❍

Bit NI: nenhum incremento na taxa (mild congestion) Emissor reduz taxa

❍ para o menor valor garantido

Bit CI: indicação de congestinamento ❒ Receptor retorna células RM

ao emissor com os bits inalterados

Camada Transporte 3-88

Estudo de caso: Controle de congestionamento ABR da rede ATM

❒ Campo ER (explicit rate) de 2 bytes no cabeçalho da célula RM

❍

Comutador congestionado pode reduzir valor ER na célula

❍

Taxa do emissor ajustada para a menor taxa suportada pelo caminho até o receptor ❒ Bit EFCI em células de dados: setado para 1 em comutadores

congestionados

❍

Se célula de dados precedendo célula RM possui o bit RFCI setado, emissor seta bit CI na célula RM retornada

Camada Transporte 3-89

Resumo do Módulo 3

❒ 3.1 Serviços da camada

transporte ❒ 3.2 Multiplexação e

demultiplexão ❒ 3.3 transporte não-

orientado à conexão: UDP ❒ 3.4 Princípios da transferência confiável de dados

❒ 3.5 transporte orientado

à conexão: TCP

❍

Estrutura do segmento

❍

Tranferência de dados confiável

❍

Controle de fluxo

❍

Gerenciamento de conexão ❒ 3.6 Princípios do

controle de congestionamento ❒ 3.7 controle de

congestionamento TCP

Camada Transporte 3-90

**Congwin**

❒ w segmentos, cada um c/ MSS bytes, enviados por RTT:

throughput =

w \* MSS

RTT

Bytes/sec

TCP: Controle de Congestionamento

❒ controle fim a fim (sem apoio da rede) ❒ taxa de transmissão limitada pela tamanho da janela

de congestionamento, Congwin:

Camada Transporte 3-91

TCP: Partida lenta (slow start)

❒ aumento exponencial (por

RTT) no tamanho da janela (não muito lenta!) ❒ evento de perda:

temporizador (Tahoe TCP) e/ou três ACKs duplicados (Reno TCP)

Estação A Algoritmo Partida Lenta

inicializa: Congwin = 1

R for (cada segmento c/ ACK)

Congwin++ until (evento de perda OR

CongWin > threshold)

T

T

Camada Transporte 3-92

Estação B

tempo

TCP: Evitar Congestionamento

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

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

Camada Transporte 3-93

Controle de congestionamentoTCP: additive increase, multiplicative decrease (AIMD) ❒ Abordagem: aumentar taxa de transmissão (tamanho da janela), sondar bw ainda utilizável, até ocorrência de perda de pacote

❍

Aumento aditivo: aumenta CongWin de 1 MSS a cada RTT até perda ser detectada

❍

redução multiplicativa: reduz CongWin pela metade após detectar perda

congestion Comportamento

24 Kbytes

window

dente-de-serra: Sondagem de

16 Kbytes bw

8 Kbytes

Camada tempo

time

Transporte 3-94

Refinamento

Q: Quando o aumento

exponencial deve parar para se tornar linear? A: Quando CongWin

assume 1/2 do seu valor antes do timeout.

Implementação: ❒ Threshold (limiar) Variável ❒ No evento de perda,

Threshold é setado para 1/2 da CongWin. Valor da janela antes do evento de perda

Camada Transporte 3-95

Refinamento: inferindo perdas

❒ Após 3 ACKs duplicados:

❍

CongWin é reduzida pela metade

❍

Filosofia: Janela cresce lineamente ❒ Mas após evento de

timeout:

❍

❑ 3 ACKs dup. indica rede capaz de entregar alguns segmentos

CongWin é setada para 1 MSS;

❑ timeout indica um cenário “mais alarmante” de congestionamento ❍

Janela então cresce exponencialmente

❍

Até um threshold (limiar), e então volta a crescer linearmente

Camada Transporte 3-96

Sumário: Controle de Congestionamento

❒ Quando CongWin está abaixo de Threshold,

emissor está na fase slow-start, janela cresce exponencialmente.

❒ Quando CongWin está acima de Threshold,

emissor está na fase congestion-avoidance, janela cresce linearmente.

❒ Quando três ACKs duplicados ocorrem,

**Threshold é setado para CongWin/2 e CongWin é setado para Threshold.**

❒ Quando timeout ocorre, Threshold é setado para

CongWin/2 e CongWin é setado para 1 MSS.

Camada Transporte 3-97

Controle de congestionamento TCP

**estado Evento Ação do emissor TCP Comentário**

Slow Start (SS)

ACK recebido para dado ainda não confirmado

CongWin = CongWin + MSS, If (CongWin > Threshold)

estado atual setado para “Congestion Avoidance”

Dobra CongWin a cada RTT

Congestion Avoidance (CA)

ACK recebido para dado ainda não confirmado

CongWin = CongWin+MSS \* (MSS/CongWin)

Aumento aditivo, CongWin aumenta de 1 MSS a cada RTT

SS ou CA Perda

detectada através de 3 ACKs duplicados

Threshold = CongWin/2, CongWin = Threshold, Estado atual setado para “Congestion Avoidance”

Recuperação rápida, implementando redução multiplicativa. CongWin não cairá abaixo de 1 MSS.

SS ou CA Timeout Threshold = CongWin/2,

CongWin = 1 MSS, Estado atual setado para “Slow Start”

Entra no slow start

SS ou CA ACK

duplicado

Incrementar contador de ACK duplicado para o segmento sendo confirmado

CongWin e Threshold não mudam

Camada Transporte 3-98

Vazão do TCP

❒ Qual a vazão média do TCP em função do

tamanho da janela e do RTT?

❍

Ignore slow start ❒ Seja W o tamanho da janela quando uma

perda ocorre. ❒ Quando janela é W, vazão é W/RTT ❒ Logo após perda, janela cai à W/2, e vazão

à W/2RTT. ❒ Vazão média: .75 W/RTT

Camada Transporte 3-99

Características do TCP

❒ Exemplo: segmentos de 1500 bytes, RTT = 100ms,

deseja-se 10 Gbps de vazão ❒ Requer janela W = 83,333 segmentos sendo

enviados ❒ Vazão em termos da taxa de perdas (L):

❒ ➜ L = 2·10-10 Uau! ❒ Novas versões do TCP para altas taxas de envio

necessárias!

22.1

*∙ MSS LRTT*

Camada Transporte 3-100

Justiça no TCP (TCP Fairness)

Objetivo da justiça: se K sessões TCP compartilham

um mesmo enlace (de gargalo) de banda passante R, cada uma deve ter uma taxa média de R/K

Conexão TCP 1

Roteador gargalo capacidade R conexãoTCP 2

Camada Transporte 3-101

Por que TCP é justo?

2 sessões competindo: ❒ Aumento aditivo - saltos de 1 (vazão aumenta) ❒ Redução multiplicativa dimuni vazão proporcionalmente

R

Compartilhamento igual da banda pasante

perda: congestion diminui avoidance: janela por aumento fator aditivo

de 2

perda: diminui janela por fator de 2 congestion avoidance: aumento aditivo

Vazão da conexão 1

R

Camada Transporte 3-102

Justiça (mais)

Justiça e o UDP ❒ Apps multimídia geralmente

não usam TCP

❍

Para não restringir taxa devido ao controle de congestionamento ❒ Em vez do TCP, as apps

multimídia usam UDP:

❍

áudio/vídeo injetados à taxas constantes, toleram perdas de pacotes ❒ Área de pesquisa: TCP

friendly (protocolos amigos do TCP)

Justiça e conexões TCP paralelas ❒ Nada previne apps de abrirem conexões paralelas entre dois hosts. ❒ Web browsers fazem isto ❒ Exemplo: enlace de taxa R suportando 9 conexões;

❍

Nova app requisita 1 conex.TCP, recece taxa R/10

❍

Nova app requisita 11 conex.TCPs, recebe R/2 !

Camada Transporte 3-103

Modelagem de Atraso

Q: Quanto tempo leva para

receber um objeto de um servidor Web após o envio da requisição ? Ignorando

congestionamento, atraso é influenciado por: ❒ Estabelecimento de conexão

TCP ❒ Atraso de transmissão de

dados ❒ slow start (partida lenta)

Notação, hipóteses: ❒ Assuma 1 enlace de taxa R entre cliente e servidor ❒ S: MSS (bits) ❒ O: tamanho do objeto

(bits) ❒ sem retransmissões (sem

perdas, não há pkts corrompidos) Tamanho da janela: ❒ Primeiro assuma: janela de

congestionamento fixa, W segmentos ❒ Em seguida, assuma janela

dinâmica com modelagem do slow start

Camada Transporte 3-104

Janela de congestionamento fixa (1)

Primeiro caso: WS/R > RTT + S/R: ACK para

primeiro segmento na janela retorna antes da transmissão de todos os segmentos da janela

delay = 2RTT + O/R

Camada Transporte 3-105

Janela de congestionamento fixa(2)

Segundo caso: ❒ WS/R < RTT + S/R:

aguarda por ACK após enviar todos os segmentos previstos na janela

delay = 2RTT + O/R + (K-1)[S/R + RTT - WS/R]

Camada Transporte 3-106

Modelagem do Atraso no TCP: Slow Start (1)

Agora suponha que janela cresce de acordo com o

slow start O atraso para um objeto será :

*Latency =*

*2 RTT + O R*

*+ RTTP ⌈ │ ⌊*

*+ R S*

⌉ │ ⌋

-

)12( P - S R

onde P é o número de vezes que o TCP fica esperando no servidor:

P = }1,{min KQ - -onde Q é o número de vezes que o servidor espera caso o objeto fosse de tamanho infinito.

- e K é o número necessário de janelas para transmitir o objeto.

Camada Transporte 3-107

Modelagem do Atraso no TCP: Slow Start (2)

Componentes do atraso:

• 2 RTT para estabelecimento de conex. e requisição

• O/R para transmitir objeo

• tempo que o servidor

RTT

espera devido ao slow start

Servidor espera:

P = min{K-1,Q} vezes

initiate TCP connection

request object

first window = S/R

second window = 2S/R

third window = 4S/R

Exemplo:

• O/S = 15 segmentos

fourth window = 8S/R

• K = 4 janelas

• Q = 2

• P = min{K-1,Q} = 2

complete transmission

Servidor espera P=2 vezes

object delivered

time at client

time at server

Camada Transporte 3-108

Modelagem do Atraso no TCP (3)

*R S*

*+ RTT*

= time until

from Tempo when desde server o starts início to do send envio segment de um server segmento receives até acknowledg o recebimento ement do ack

2 -

1 R

= totime Tempo transmit para ésima transmitir the janela

kth window a k-

Tempo de espera após k- ésima janela

atraso Tempo de espera

p

*R*

initiate TCP

*k*

*S*

connection

request object

first window = S/R ⌈ │ ⌊

*R S*

+

*RTT*

- 2 k -

*1 R S*

⌉ │ ⌋

+

=

idle time after the k

th window RTT

second window = 2S/R

third window = 4S/R

delay

=

*O R*

*+ 2 RTT*

+ ∑ p

*P*

=

1

*idleTime p*

fourth window = 8S/R

=

*O R*

+ 2

*RTT*

+ ∑

*P*

*k*

=

1

*[ R S*

*+ RTT*

- ]2 k -

1

*S R*

complete transmission =

+ 2

+ [ + )12(] - P

- object O R

*RTTPRTT*

*S R*

*S*

delivered

time at client

time at server

Camada Transporte 3-109

Modelagem do Atraso no TCP (4)

Lembre que K = número de janelas para transmitir objeto Como calculamos K ?

*K*

=

22:{min

*SSk 0*

+ 1 + + 2 k

- 1 OS ≥ } 

-

*k*

)1(log

2

*S*

│ │

 =

22:{min

*k*

0

+ 1 + + 2 k 1 ≥ SO }/ =

}12:{min

*k*

*- ≥ O S*

=

:{min

*kk*

≥ )}1(log 2

Cálculo de Q, número de “esperas” para objeto de tamanho infinito, é similar (veja o livro).

*O S + =*

⌈ │ │

*O*

+ ⌉

Camada Transporte 3-110

Modelagem do HTTP

❒ Assuma que uma página Web consista em:

❍

1 página-base em HTML (de tamanho O bits)

❍

M imagens (cada uma de tamanho O bits) ❒ HTTP não-persistente:

❍

M+1 conexões TCP em série

❍

Tempo de Resposta = (M+1)O/R + (M+1)2RTT + soma de tempos de espera ❒ HTTP Persistente:

❍

2 RTT para requição e recebimento da página-base HTML

❍

1 RTT para requisição e início de recebimento de M images

❍

Tempo de Resposta = (M+1)O/R + 3RTT + soma de tempos de espera ❒ HTTP Não-persistente com X conexões paralelas

❍

Suponha que M/X seja inteiro.

❍

1 conexão TCP para o aqrquivo de base

❍

M/X conjuntos de conexões paralelas para imagens.

❍

Tempo de Resposta = (M+1)O/R + (M/X + 1)2RTT + soma de tempos de espera

Camada Transporte 3-111

Tempo de resposta do HTTP (em seg.)

RTT = 100 ms, O = 5 Kbytes, M=10 e X=5

Para baixa BW, tempo de conexão e de resposta dominado pelo tempo de transmissão.

Conexões persistentes somente provêem pequena melhoria com relação ao uso de conexões paralelas.

**20 18 16 14 12 10 8 6 4 2 0**

**28 Kbps**

**100 Kbps**

**1 Mbps**

**10 Mbps**

**non-persistent**

**persistent**

**parallel non- persistent**

Camada Transporte 3-112

Tempo de resposta do HTTP (em seg.)

RTT =1 sec, O = 5 Kbytes, M=10 and X=5

Para altos RTTs, tempo de resposta é dominado pelos atrasos de estabelecimento de conexão & do slow start. Conexões persistentes proporcionam agora melhoria importante de desempenho: particularmente em redes com alto produto atraso•bw.

**70**

**60 50**

**40**

**30 20**

**10 0**

**28 Kbps**

**100 Kbps**

**1 Mbps**

**10 Mbps**

**non-persistent**

**persistent**

**parallel non- persistent**

Camada Transporte 3-113

Módulo 3: Sumário

❒ Princípios por trás dos serviços da camada transporte:

❍

multiplexação, demultiplexação

❍

Transferência confiável de dados

❍

Controle de fluxo

❍

Controle de congestionamento ❒ Instanciação e

implementação na Internet

❍

UDP

❍

TCP

A seguir: ❒ Deixando a

extremidade da rede (camadas aplicação, transporte) ❒ e entrando no núcleo da

rede

Camada Transporte 3-114