BM 402 Bilgisayar Ağları (Computer Networks)

M.Ali Akcayol
Gazi Üniversitesi
Bilgisayar Mühendisliği Bölümü

Not: Bu dersin sunumları, ders kitabının yazarları James F. Kurose ve Keith W. Ross tarafından sağlanan sunumlar üzerinde değişiklik yapılarak hazırlanmıştır.

Transport Layer

Amaçlar :

- Transport layer hizmetlerinin anlaşılması:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- Internette trnasport layer protokolleri:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control

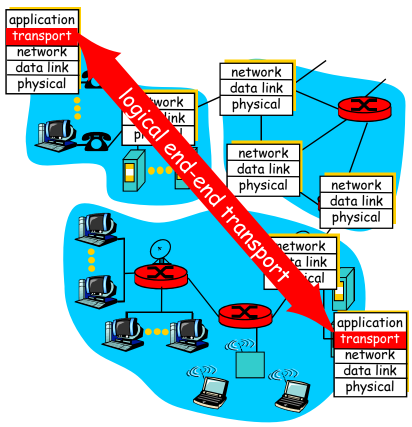
2/101

Ders konuları

- Transport-layer hizmetleri
- Multiplexing ve demultiplexing
- Connectionless transport: UDP
- Reliable data transferin prensipleri
- Connection-oriented transport: TCP
  - segment yapısı
  - reliable data transfer
  - flow control
  - connection management
- Tıkanıklık denetimi prensipleri
- TCP tıkanıklık denetimi

Transport hizmetleri ve protokoller

- Farklı hostlardaki prosesler arasında *mantıksal bağlantı (logical communication)* oluşturur
- transport protokolleri uç sistemlerde çalışır
  - Gönderen taraf: uygulama mesajlarını segment'lere böler ve network layer'a gönderir
  - Alıcı taraf: segmentleri birleştirir ve applicatio layer'a gönderir
- Uygulamalar için birden fazla transport protokol bulunmaktadır
  - Internet: TCP ve UDP



Transport ve network layer

- *network layer:* hostlar arasında mantıksal bağlantı
- *transport layer:* prosesler arasında mantıksal bağlantı
  - Network layer hizmetlerine güvenir

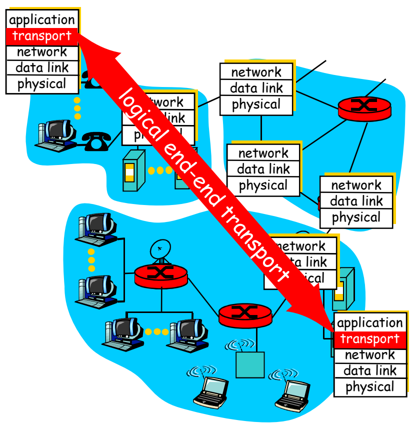
Benzetim:

12 çocuk farklı yerde aynı evdeki 12 çocuğa mektup gönderiyor

- prosesler = çocuklar
- Uygulama mesajları = zarflardaki mektuplar
- hostlar = evler
- transport protokol = her evde mektup toplama ve dağıtma yapan kişiler
- network-layer protokol = posta hizmeti

Internet transport layer protokolleri

- Güvenilir, sıralı gönderim: TCP
  - congestion control
  - flow control
  - connection setup
- Güvenilir olmayan, sırasız gönderim: UDP
  - IP protokolünün best-effort özelliği kullanılır
- Sunulamayan hizmetler:
  - delay garanti
  - bandwidth garanti



Ders konuları

- Transport-layer hizmetleri
- Multiplexing ve demultiplexing
- Connectionless transport: UDP
- Reliable data transferin prensipleri
- Connection-oriented transport: TCP
  - segment yapısı
  - reliable data transfer
  - flow control
  - connection management
- Tıkanıklık denetimi prensipleri
- TCP tıkanıklık denetimi

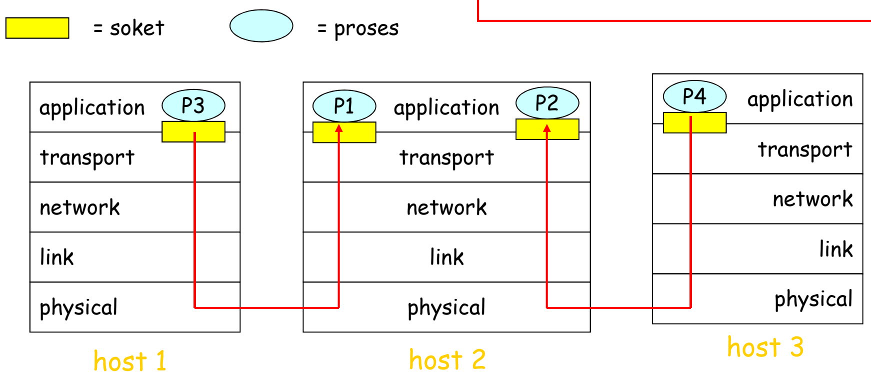
Multiplexing/demultiplexing

Demultiplexing alıcı host'ta

Doğru soketten gelen segmentler alınır

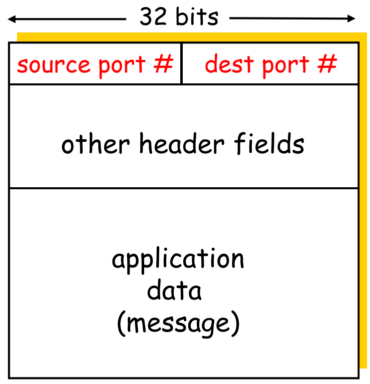
Multiplexing gönderen host'ta

Birden çok soketten gelen veri toplanır, başlık eklenir (demultiplex için kullanılır)



Demultiplexing işlemi

- host IP datagramlarını alır
  - her datagram kaynak IP adresine ve hedef IP adresine sahiptir
  - her datagram 1 transport-layer segment taşır
  - ehr segment kaynak ve hedef port numarasına sahiptir (bazı uygulamalar için bilinen port numaraları atanır)
- host IP adreslerini ve port numaralarını kullanarak segmenti uygun sokete yönlendirir.



TCP/UDP segment format

9/101

Connectionless demultiplexing

- Port numarlarıyla soketler oluşturulur:

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

- UDP soket bir ikiliyle tanımlanır:

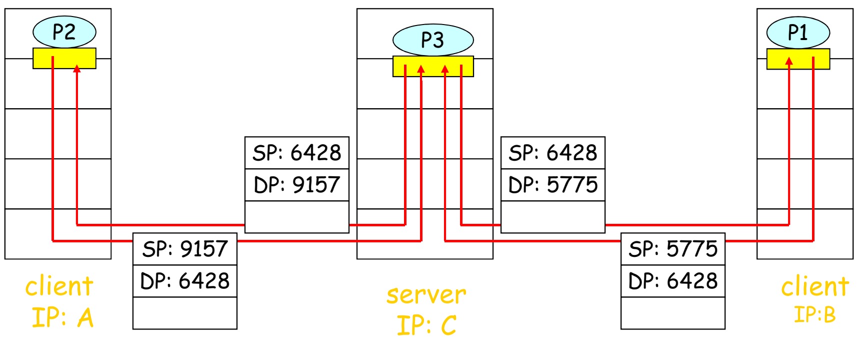
(dest IP address, dest port number)

- Host UDP segment aldığında:
  - Segmentteki hedef port numarası kontrol
  - UDP segment port numarasıyla birlikte sokete yönlendirilir
- IP datagramlar aynı sokete kaynak ve hedef IP ve port numaralarıyla yönlendirilir

10/101

Connectionless demultiplexing - devam

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



SP dönüş adresini (return address) sağlar

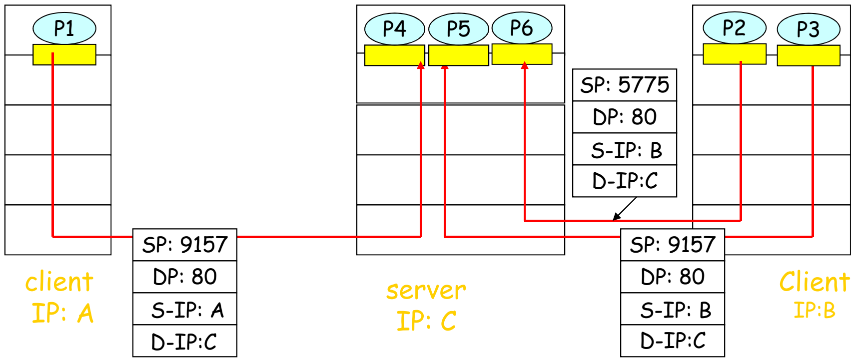
11/101

Connection-oriented demultiplexing

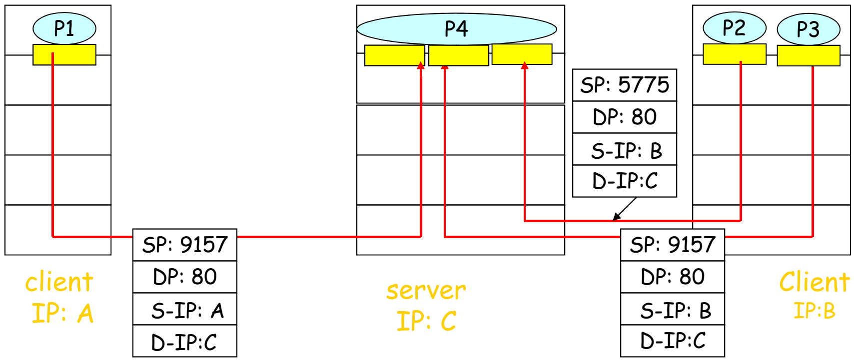
- TCP socket bir dörtlüyle tanımlanır:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- Alıcı host bu 4 değeri segmenti uygun sokete yönlendirmek için kullanır
- Server host eşzamanlı çok sayıda TCP soket destekler:
  - her soket kendi dörtlüsüyle tanımlanır
- Web sunucular her bağlanan client için farklı bir sokete sahiptir
  - non-persistent HTTP her istek için farklı bir sokete sahiptir

12/101

Connection-oriented demultiplexing - devam



Connection-oriented demux: Threaded Web Server



Ders konuları

- Transport-layer hizmetleri
- Multiplexing ve demultiplexing
- Connectionless transport: UDP
- Reliable data transferin prensipleri
- Connection-oriented transport: TCP
  - segment yapısı
  - reliable data transfer
  - flow control
  - connection management
- Tıkanıklık denetimi prensipleri
- TCP tıkanıklık denetimi

UDP: User Datagram Protocol [RFC 768]

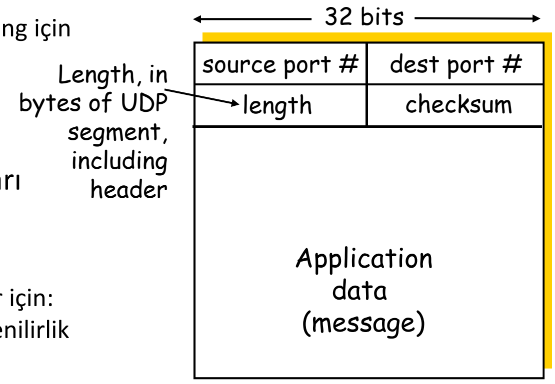
- Güvenilir olmayan Internet transport protokolü
- "best effort" servis, UDP segmentleri:
  - Lost (kayıp olabilir)
  - Delivered out of order to app (uygulamaya sırasız gidebilir)
- *connectionless:*
  - UDP alıcı ve gönderici arasında handshaking yapılmaz
  - Her UDP segment diğerlerinden ayrı değerlendirilir

UDP niçin vardır?

- Bağlantı oluşturulmaz (gecikme olur)
- basittir: alıcı ve gönderici arasında bağlantı durumu yoktur
- Küçük segment başlığı
- Tıkanıklık denetimi yoktur: UDP istediği hızda veri gönderebilir.

UDP - devam

- Sıklıkla multimedia uygulamalarında streaming için kullanılır
  - loss tolerant
  - rate sensitive
- diğer UDP kullanımları
  - DNS
  - SNMP
- UDP ile güvenilir transfer için: application layer'da güvenilirlik eklenmelidir.



UDP segment format

17/101

UDP checksum

Amaç: iletilen segmentte hata algılama

Gönderici:

- segment içeriklerine 16-bit integer dizisi olarak bakılır
- checksum: segment içeriğinin 1 tümleyeni toplamı alınır
- Gönderici checksum değeri UDP checksum alanına yerleştirir

Alıcı:

- Alınan segmentte checksum hesaplanır
- hesaplanan checksum gelen checksum değeriyle karşılaştırılır:
  - HAYIR – hata var
  - EVET – hata yok. *Yinede hiçbir şekilde hata yok denilebilirmi?*

18/101

Internet checksum örnek

- Sayıları toplarken en soldaki bitlerdeki taşma sonuca eklenir
- Örnek: iki 16-bit integer toplanırsa

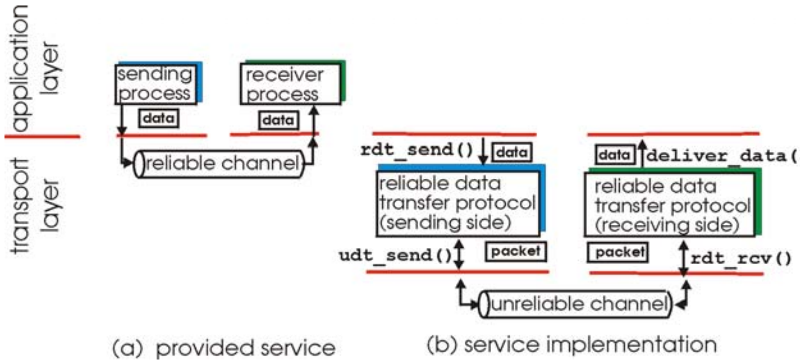
		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Ders konuları

- Transport-layer hizmetleri
- Multiplexing ve demultiplexing
- Connectionless transport: UDP
- Reliable data transferin prensipleri
- Connection-oriented transport: TCP
  - segment yapısı
  - reliable data transfer
  - flow control
  - connection management
- Tıkanıklık denetimi prensipleri
- TCP tıkanıklık denetimi

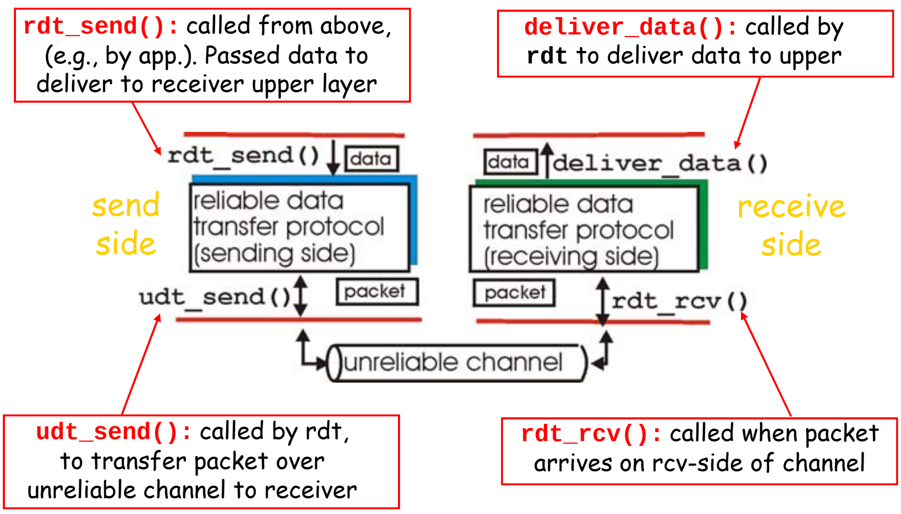
Reliable data transferin prensipleri

- Application, transport ve data link layer'da önemlidir
- Ağlarda en öncelikli 10 konunun içindedir.



- Reliable data transfer (rdt) protokol daha karmaşıktır.

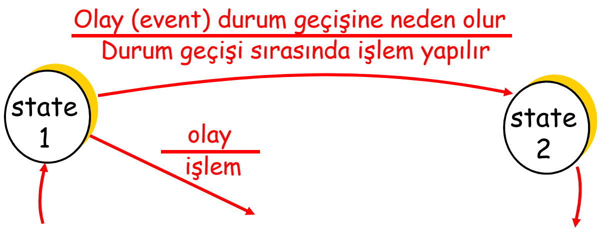
Reliable data transfer



Reliable data transfer

- reliable data transfer (rdt) protokolün gönderici ve alıcı taraflarını geliştirelim
- Data transferin tek yönlü (unidirectional) olduğunu düşünürsek
  - Ancak kontrol bilgisi iki yönlü gitmektedir
- Sonlu durum makineleriyle (finite state machines-FSM) modellenebilir

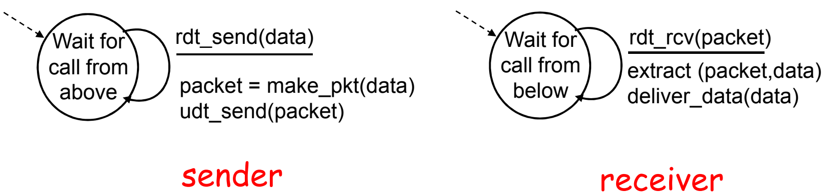
state: bir sonraki durum bu durumdayken oluşan sonraki olayla belirlenir



23/101

Rdt 1.0: reliable kanal kullanarak reliable transfer

- Altyapıdaki kanal tümüyle güvenilirdir
  - Bit hatası yoktur
  - Kayıp paket yoktur
- Gönderici ve alıcı için FSM:
  - Gönderici kanala veriyi gönderir
  - Alıcı kanaldan gelen veriyi okur



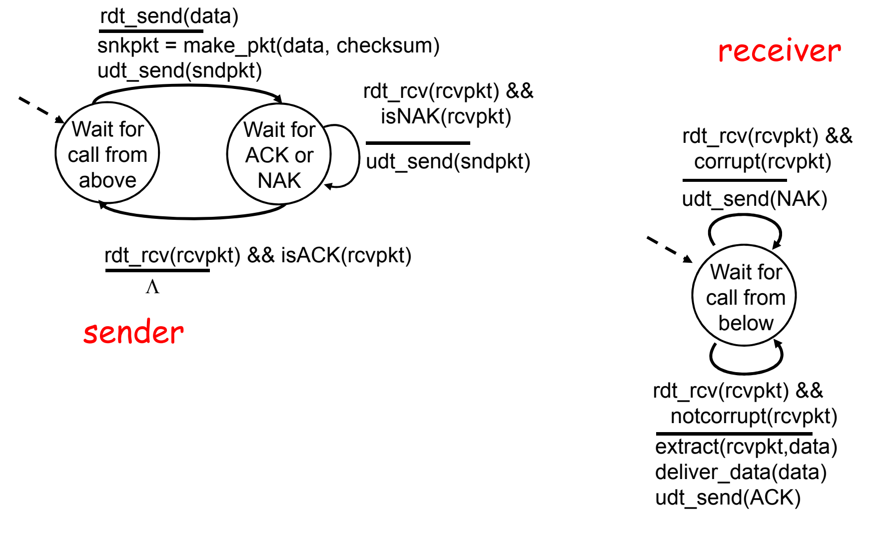
24/101

rdt 2.0: bit hatası olan kanal ile çalışma

- Kanalda paket içindeki bitlerde bozulma olabilir
  - Bit hatalarını kontrol etmek için checksum kullanılır
- Hatalar nasıl düzeltilir ?
  - *acknowledgements (ACKs):* alıcı göndericiye aldığı paketin hatasız olduğunu iletir.
  - *negative acknowledgements (NAKs):* alıcı göndericiye aldığı paketin hatalı olduğunu bildirir
  - Gönderici NAK ile bildirilen paketi tekar gönderir
- **`rdt2.0`** daki yenilikler(**`rdt1.0 a göre`**):
  - Hata denetimi
  - Alıcı geri bildirimi: kontrol mesajları (ACK, NAK) alıcı->gönderici

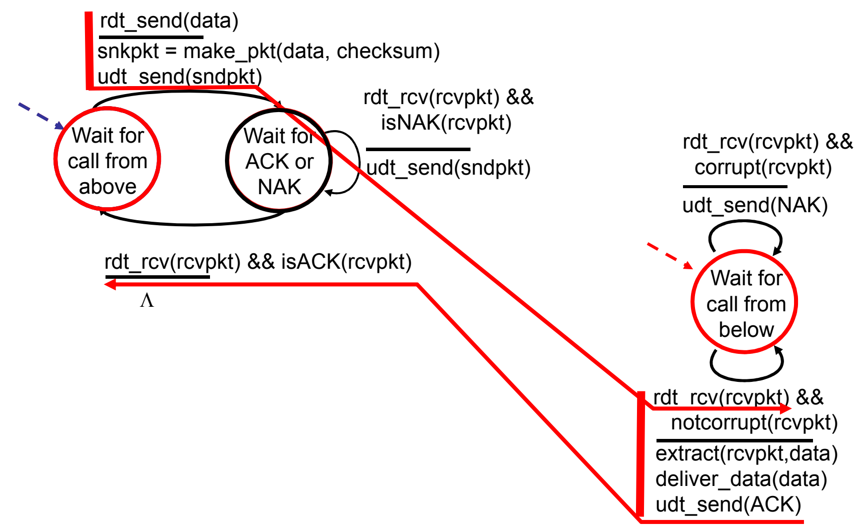
25/101

rdt2.0: FSM özellikleri



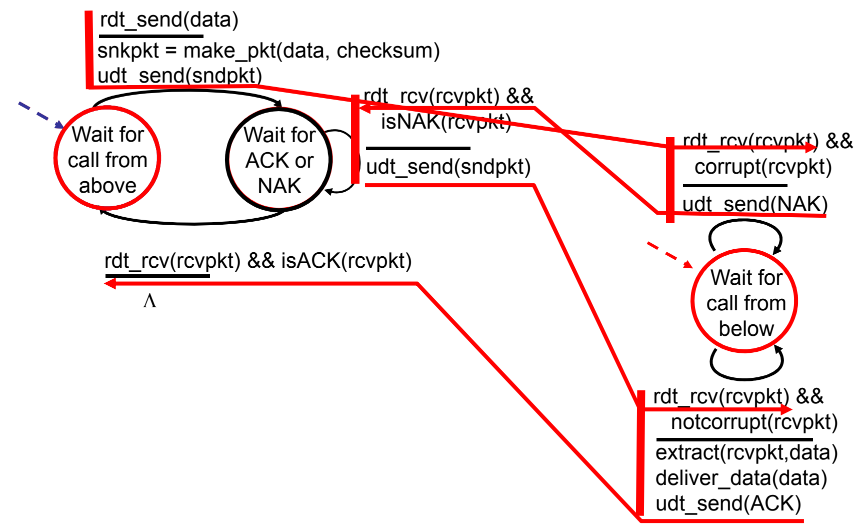
26/101

rdt2.0: hata olmadığı zaman çalışma



27/101

rdt2.0: hata durumu



28/101

rdt2.0 da karşılaşılan problemler

ACK/NAK bozulursa ?

- Gönderici alıcıda ne olduğunu bilemez
- Retransmit yapılmaz: duplicate olabilir

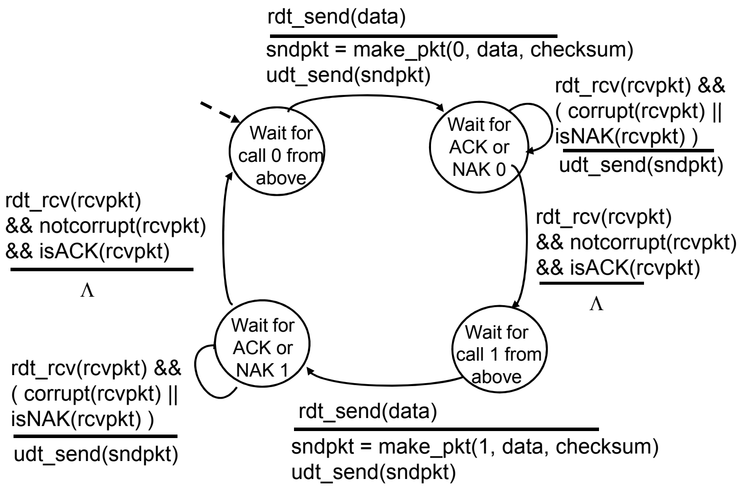
Duplicate'lerin seçilmesi:

- Gönderici her pakete *sequence number* ekler
- Gönderici mevcut paketi retransmit yapar ACK/NAK bozulursa
- alıcı duplicate paketleri atar

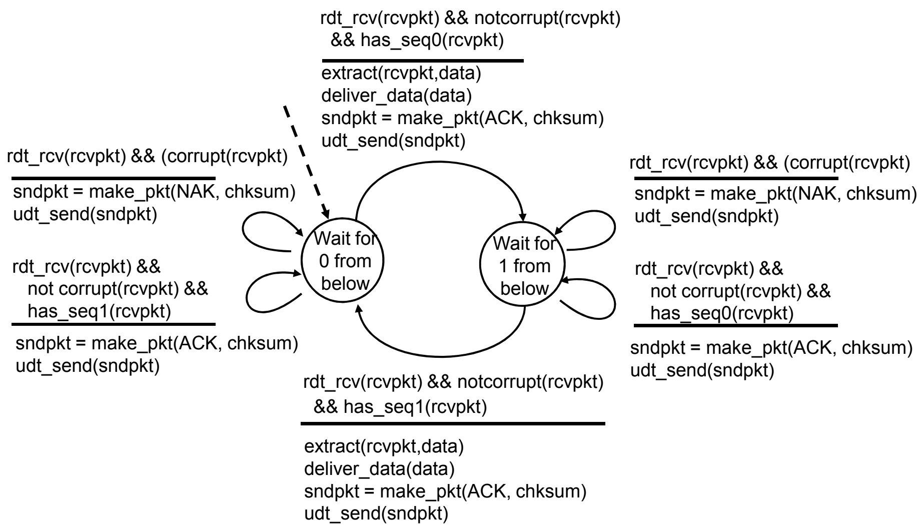
stop and wait

Gönderici bir paket gönderir, Alıcıdan cevap bekler

rdt2.1: gönderici bozulan ACK/NAK ları belirler



rdt2.1: gönderici bozulan ACK/NAK ları belirler



rdt2.1: değerlendirme

Gönderici:

- seq # pakete eklenir
- İki seq. no (0,1) yeterlidir.
- alınan ACK/NAK paketin bozuk olup olmadığı kontrol edilir

Alıcı:

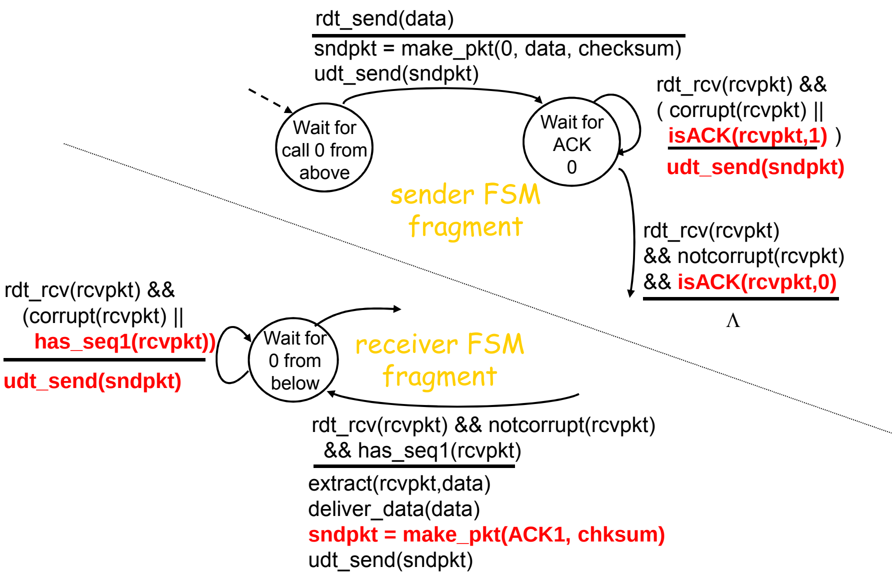
- Gelen paket çiftmi kontrol edilir
  - Bulunulan durum gelen paket için seq. no, 0 veya 1 olaraak bekler

rdt2.2: NAK kullanılmayan protokol

- ACK kullanarak rdt2.1 ile aynı fonkisyonu görür
- NAK yerine, alıcı en son doğru alınan paket için ACK paket gönderir
  - Alıcı paketin seq numarasını bilmelidir
- Alıcıdaki duplicate ACK : paketin *retransmit edilmesini sağlar*

33/101

rdt2.2: gönderici ve alıcı kısımları



34/101

rdt3.0: channels with errors *and* loss

New assumption: underlying channel can also lose packets (data or ACKs)

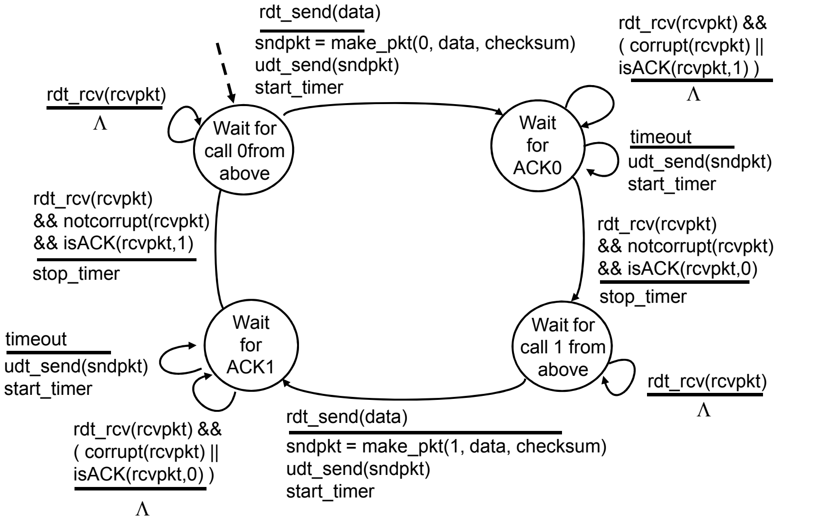
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Approach: sender waits "reasonable" amount of time for ACK

- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but use of seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

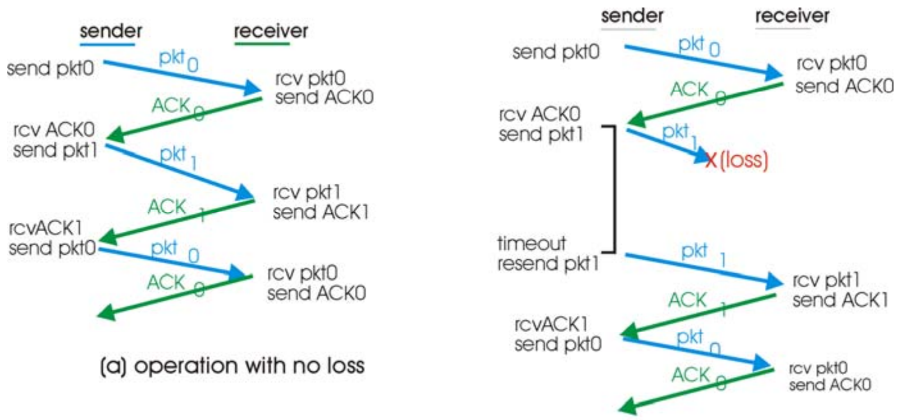
35/101

rdt3.0 sender



36/101

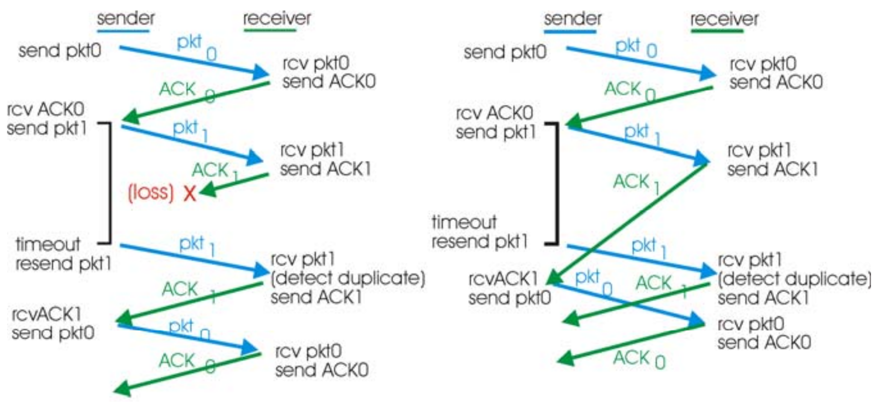
rdt3.0 in action



(a) operation with no loss

(b) lost packet

rdt3.0 in action



(c) lost ACK

(d) premature timeout

Performance of rdt3.0

- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

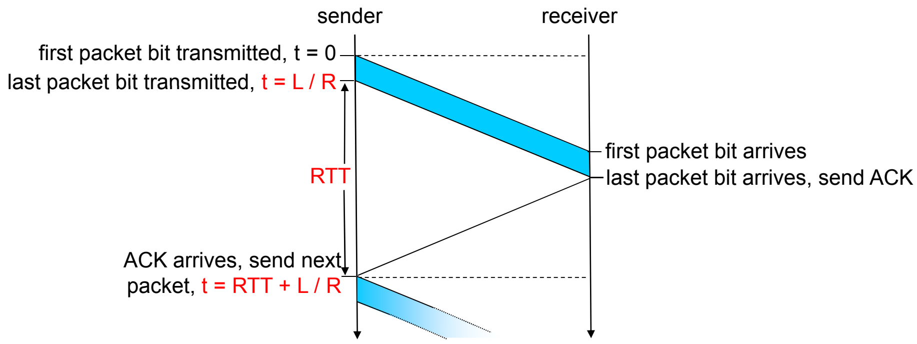
$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb/pkt}}{10^{**}9 \text{ b/sec}} = 8 \text{ microsec}$$

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

- U_{sender}: utilization – fraction of time sender busy sending
- 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!

39/101

rdt3.0: stop-and-wait operation



$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

40/101

Pipelined protocols

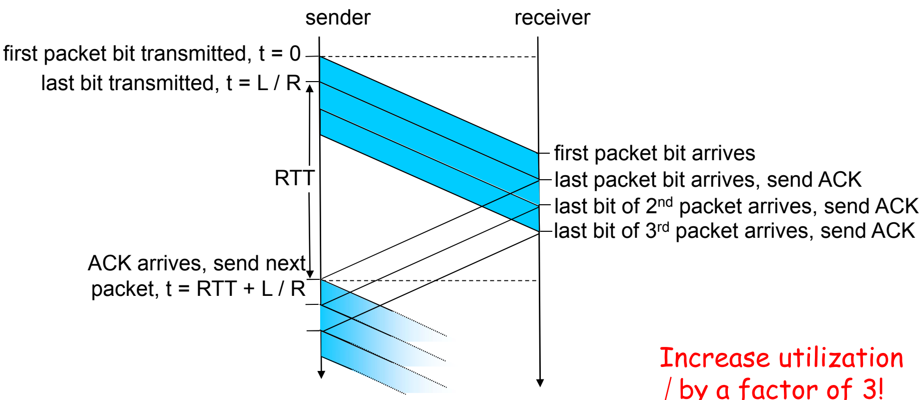
Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver

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

- Two generic forms of pipelined protocols: *go-Back-N, selective repeat*

Pipelining: increased utilization



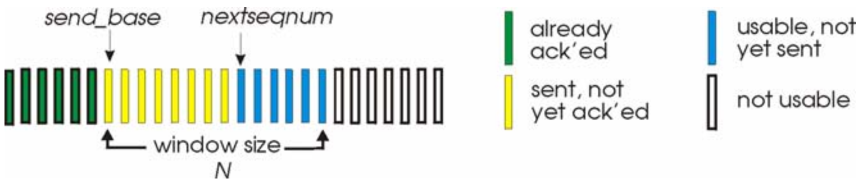
Increase utilization by a factor of 3!

$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Go-Back-N

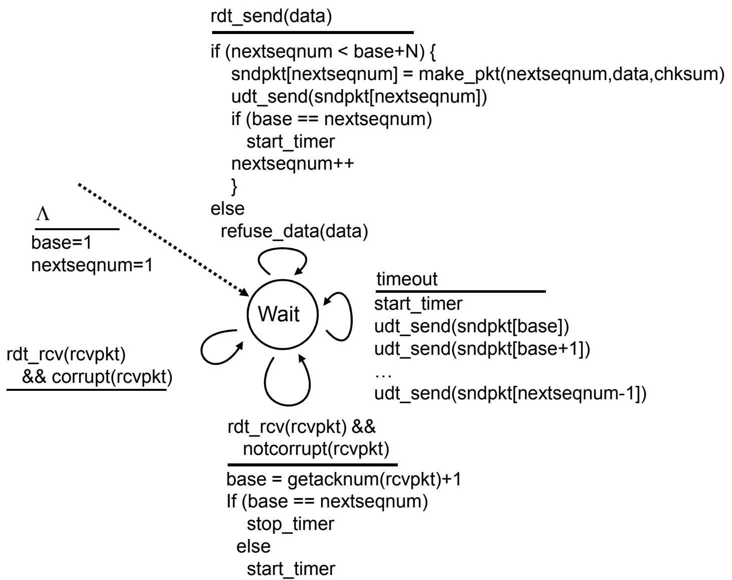
Sender:

- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed

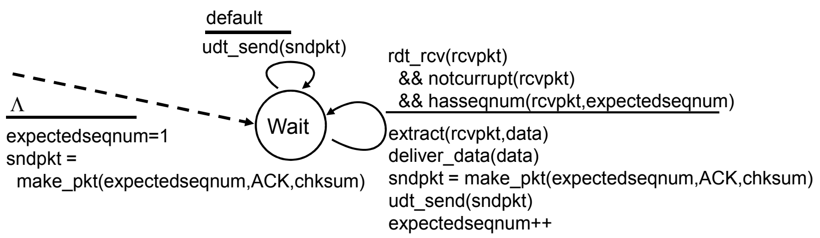


- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may deceive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- *timeout(n):* retransmit pkt n and all higher seq # pkts in window

GBN: sender extended FSM



GBN: receiver extended FSM

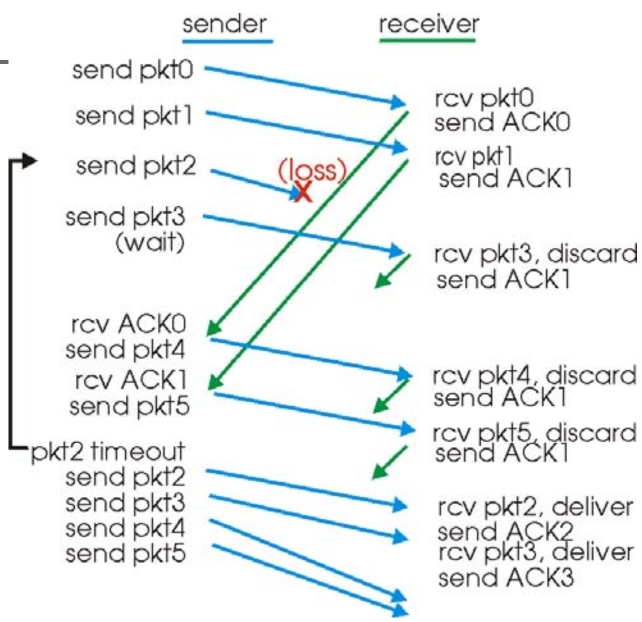


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember **expectedseqnum**

- out-of-order pkt:
  - discard (don't buffer) -> no receiver buffering!
  - Re-ACK pkt with highest in-order seq #

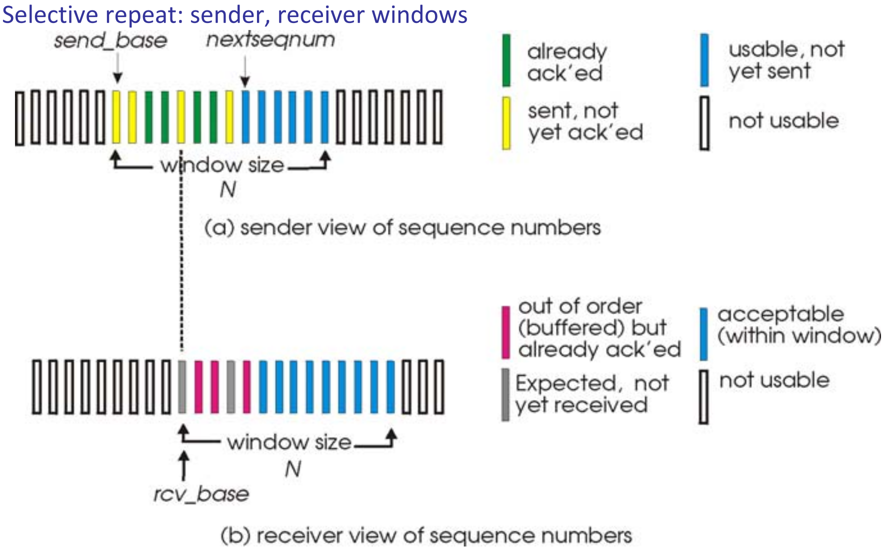
GBN in action



Selective Repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - N consecutive seq #'s
  - again limits seq #s of sent, unACKed pkts

Selective repeat: sender, receiver windows



Selective repeat

sender

data from above :
- if next available seq # in window, send pkt

timeout(n):
- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N]:
- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq #

receiver

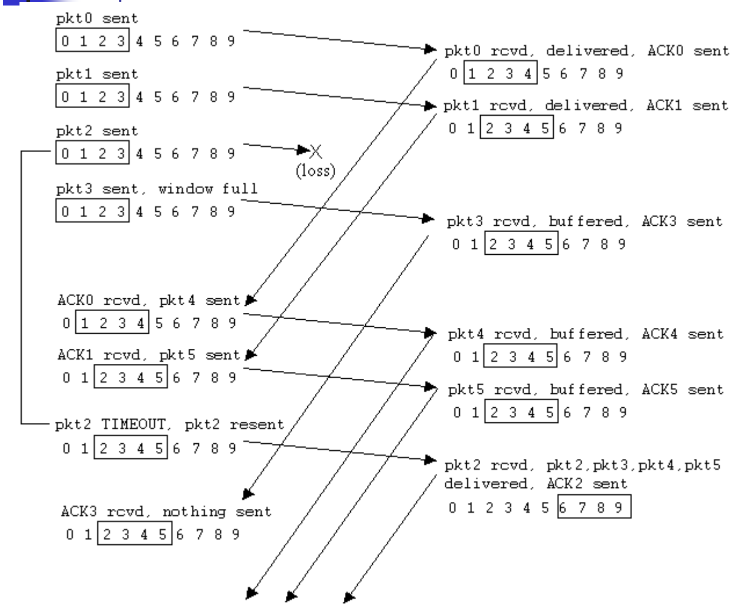
pkt n in [rcvbase, rcvbase+N-1]
- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

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

otherwise:
- ignore

49/101

Selective repeat in action

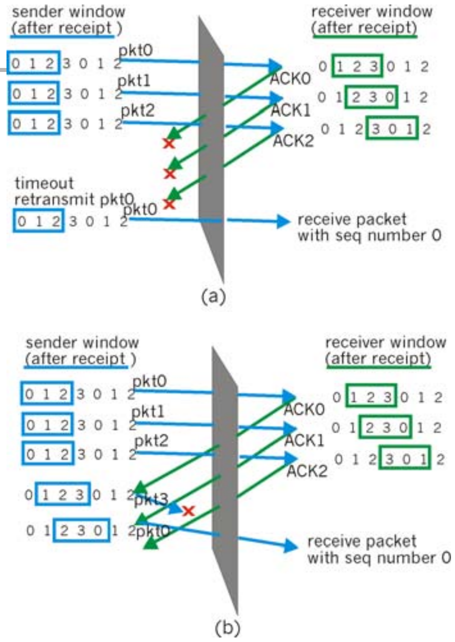


50/101

Selective repeat: dilemma

Example:

- seq #'s: 0, 1, 2, 3
- window size=3



- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size?

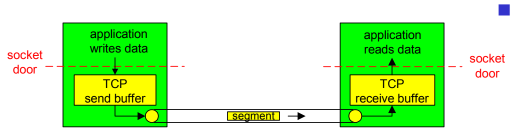
Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP: Overview

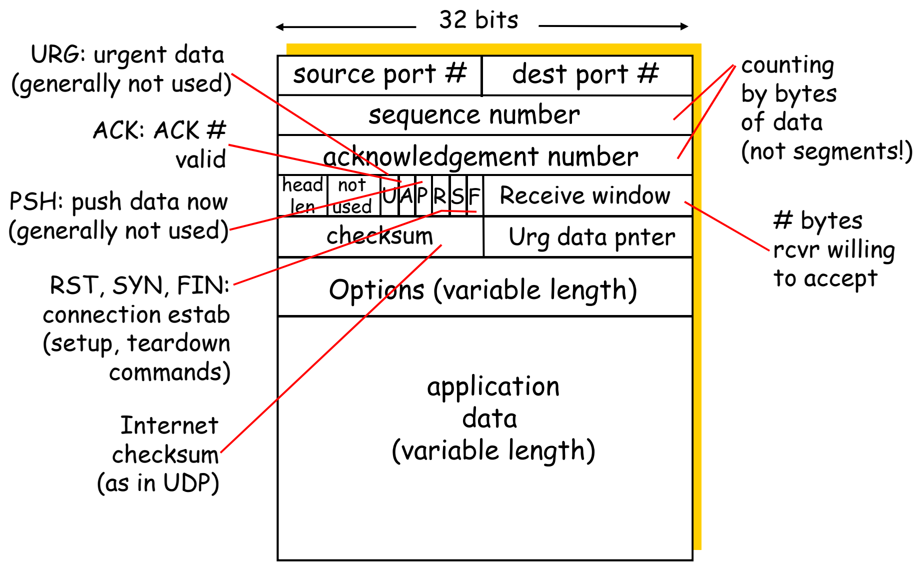
RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- *send & receive buffers*



- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure



TCP seq. #'s and ACKs

Seq. #'s:

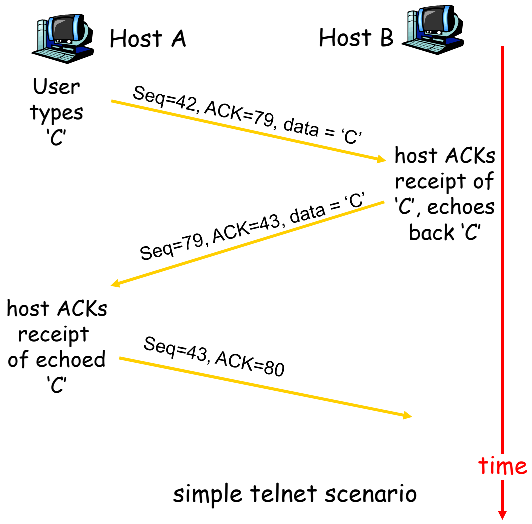
- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



simple telnet scenario

55/101

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

- longer than RTT
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?

- `SampleRTT`: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current **`SampleRTT`**

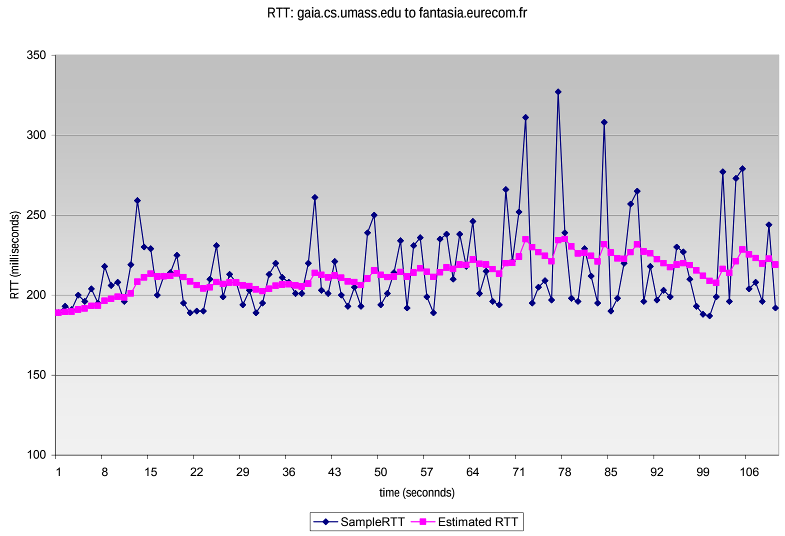
56/101

TCP Round Trip Time and Timeout

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

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: α = 0.125

Example RTT estimation:



TCP Round Trip Time and Timeout

Setting the timeout

- **EstimtedRTT** plus "safety margin"
  - large variation in **EstimatedRTT ->** larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

```
DevRTT = (1-β)*DevRTT +
             β*|SampleRTT-EstimatedRTT|

(typically, β = 0.25)
```

Then set timeout interval:

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```


Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - timeout events
  - duplicate acks
- Initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control

TCP sender events:

data rcvd from app:

- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running (think of timer as for oldest unacked segment)
- expiration interval: `TimeOutInterval`

timeout:

- retransmit segment that caused timeout
- restart timer

Ack rcvd:

- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments

TCP sender (simplified)

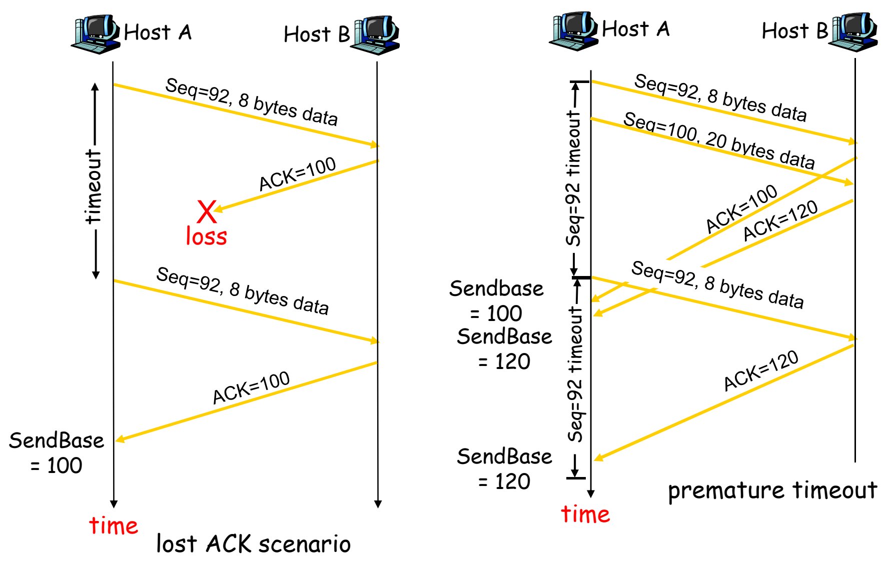
```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: data received from application above
      create TCP segment with sequence number NextSeqNum
      if (timer currently not running)
           start timer
      pass segment to IP
      NextSeqNum = NextSeqNum + length(data)

   event: timer timeout
      retransmit not-yet-acknowledged segment with
           smallest sequence number
      start timer

   event: ACK received, with ACK field value of y
      if (y > SendBase) {
           SendBase = y
          if (there are currently not-yet-acknowledged segments)
               start timer
          }

 }  /* end of loop forever */
```

Comment:
- SendBase-1: last cumulatively ack'ed byte

Example:
- SendBase-1 = 71; y= 73, so the rcvr wants 73+ ; y > SendBase, so that new data is acked

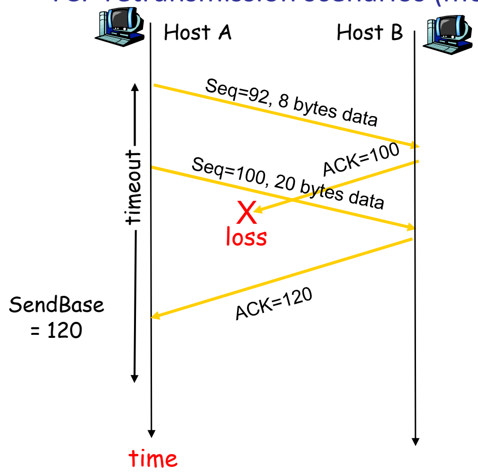
63/101

TCP: retransmission scenarios



64/101

TCP retransmission scenarios (more)



Cumulative ACK scenario

TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send duplicate ACK, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment startsat lower end of gap

Fast Retransmit

- Time-out period often relatively long:
  - long delay before resending lost packet
- Detect lost segments via duplicate ACKs.
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

67/101

Fast retransmit algorithm:

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

a duplicate ACK for already ACKed segment

fast retransmit

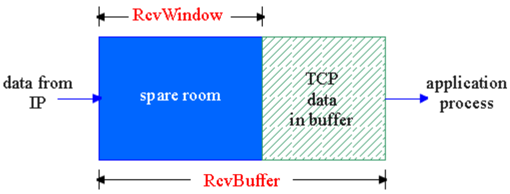
68/101

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP Flow Control

- receive side of TCP connection has a receive buffer:

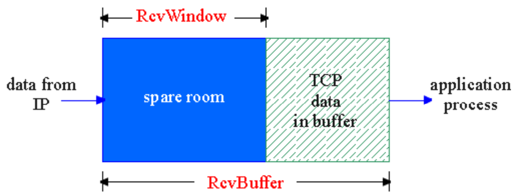


- app process may be slow at reading from buffer

flow control
sender won't overflow receiver's buffer by transmitting too much, too fast

- speed-matching service: matching the send rate to the receiving app's drain rate

TCP Flow control: how it works



(Suppose TCP receiver discards out-of-order segments)

- spare room in buffer
- = **RcvWindow**
- = **RcvBuffer-[LastByteRcvd - LastByteRead]**

- Rcvr advertises spare room by including value of **RcvWindow** in segments
- Sender limits unACKed data to **RcvWindow**
  - guarantees receive buffer doesn't overflow

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. **`RcvWindow`**)
- *client:* connection initiator

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

- *server:* contacted by client

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

Three way handshake:

Step 1: client host sends TCP SYN segment to server

- specifies initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment

- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data

TCP Connection Management (cont.)

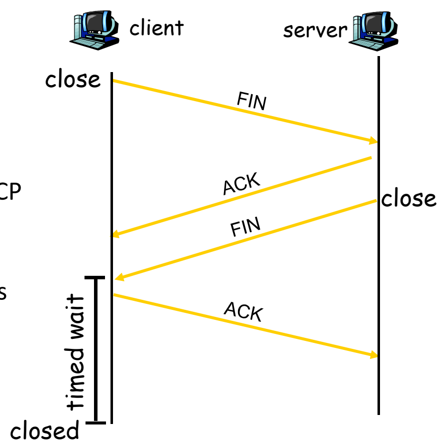
Closing a connection:

client closes socket:

```
clientSocket.close();
```

Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.



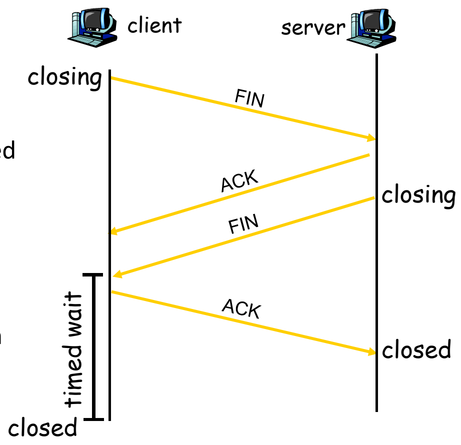
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

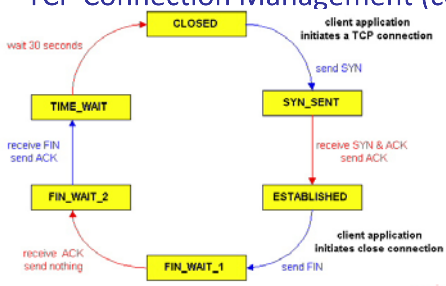
- Enters “timed wait” - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.

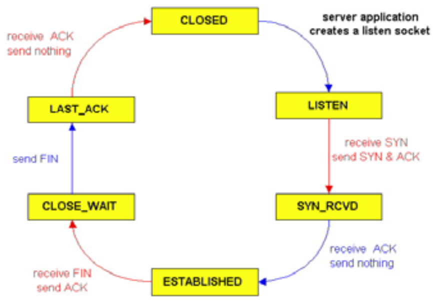


TCP Connection Management (cont)



TCP client lifecycle

TCP server lifecycle



Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

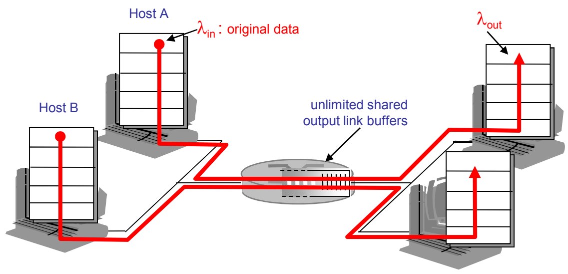
Principles of Congestion Control

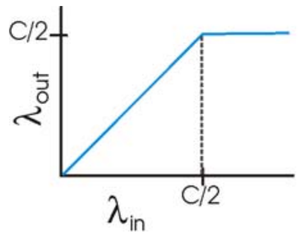
Congestion:

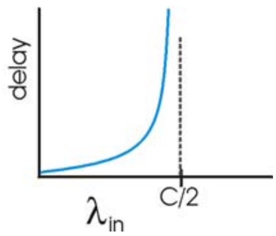
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission



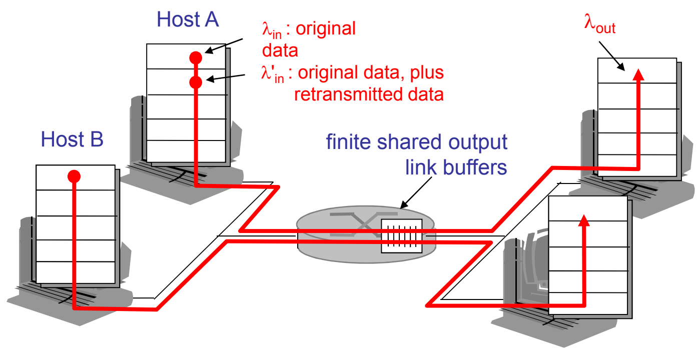




- large delays when congested
- maximum achievable throughput

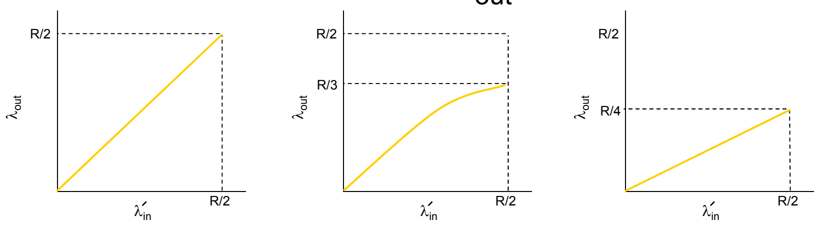
Causes/costs of congestion: scenario 2

- one router, *finite* buffers
- sender retransmission of lost packet



Causes/costs of congestion: scenario 2

- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- "perfect" retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}



a. b. c.

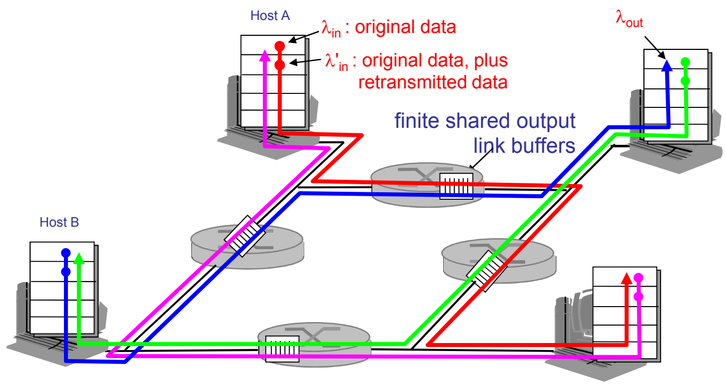
"costs" of congestion:

- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt

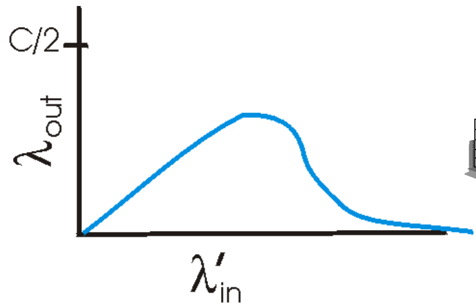
Causes/costs of congestion: scenario 3

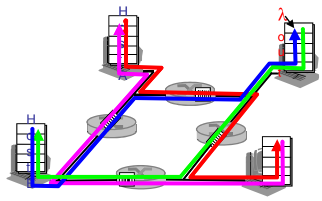
- four senders
- multihop paths
- timeout/retransmit

Q: what happens as λ_{in} and λ'_{in} increase ?



Causes/costs of congestion: scenario 3





Another "cost" of congestion:

- when packet dropped, any "upstream transmission capacity used for that packet was wasted!

Approaches towards congestion control

Two broad approaches towards congestion control:

End-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

Network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should send at

Case study: ATM ABR congestion control

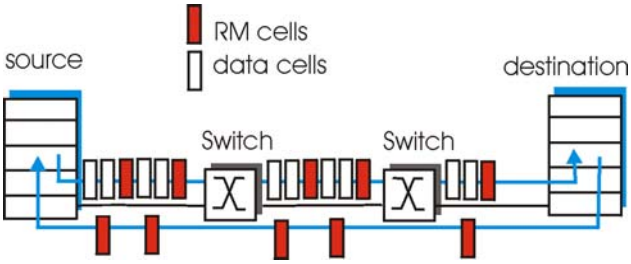
ABR: available bit rate:

- "elastic service"
- if sender's path "underloaded":
  - sender should use available bandwidth
- if sender's path congested:
  - sender throttled to minimum guaranteed rate

RM (resource management) cells:

- sent by sender, interspersed with data cells
- bits in RM cell set by switches ("*network-assisted*")
  - NI bit: no increase in rate (mild congestion)
  - CI bit: congestion indication
- RM cells returned to sender by receiver, with bits intact

Case study: ATM ABR congestion control



- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - sender' send rate thus minimum supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, sender sets CI bit in returned RM cell

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

87/101

TCP Congestion Control

- end-end control (no network assistance)
- sender limits transmission:
  `LastByteSent-LastByteAcked ≤ CongWin`
- Roughly,

$$\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

- **`CongWin`** is dynamic, function of perceived network congestion

How does sender perceive congestion?

- loss event = timeout *or* 3 duplicate acks
- TCP sender reduces rate (**`CongWin`**) after loss event

three mechanisms:

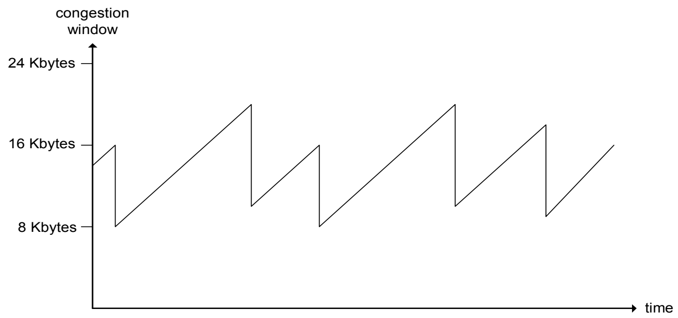
- AIMD
- slow start
- conservative after timeout events

88/101

TCP AIMD

multiplicative decrease: cut **CongWin** in half after loss event

additive increase: increase **CongWin** by 1 MSS every RTT in the absence of loss events: *probing*



Long-lived TCP connection

89/101

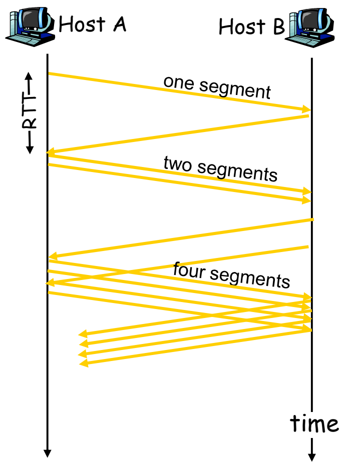
TCP Slow Start

- When connection begins, **CongWin** = 1 MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate
- When connection begins, increase rate exponentially fast until first loss event

90/101

TCP Slow Start (more)

- When connection begins, increase rate exponentially until first loss event:
  - double **`CongWin`** every RTT
  - done by incrementing **`CongWin`** for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast



91/101

Refinement

- After 3 dup ACKs:
  - **`CongWin`** is cut in half
  - window then grows linearly
- But after timeout event:
  - **`CongWin`** instead set to 1 MSS;
  - window then grows exponentially
  - to a threshold, then grows linearly

Philosophy:

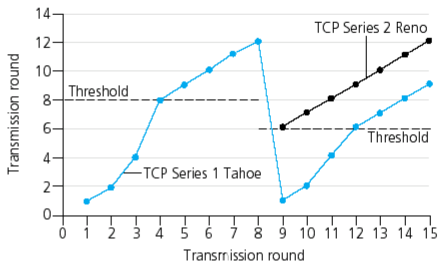
- 3 dup ACKs indicates network capable of delivering some segments
- timeout before 3 dup ACKs is "more alarming"

92/101

Refinement (more)

Q: When should the exponential increase switch to linear?

A: When **CongWin** gets to 1/2 of its value before timeout.



Implementation:

- Variable Threshold
- At loss event, Threshold is set to 1/2 of CongWin just before loss event

93/101

Summary: TCP Congestion Control

- When **CongWin** is below **Threshold**, sender in slow-start phase, window grows exponentially.
- When **CongWin** is above **Threshold**, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, **Threshold** set to **CongWin/2** and **CongWin** set to **Threshold**.
- When timeout occurs, **Threshold** set to **CongWin/2** and **CongWin** is set to 1 MSS.

94/101

TCP sender congestion control

Event	State	TCP Sender Action	Commentary
ACK receipt for previously unacked data	Slow Start (SS)	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to "Congestion Avoidance"	Resulting in a doubling of CongWin every RTT
ACK receipt for previously unacked data	Congestion Avoidance (CA)	CongWin = CongWin+MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
Loss event detected by triple duplicate ACK	SS or CA	Threshold = CongWin/2, CongWin = Threshold, Set state to "Congestion Avoidance"	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
Timeout	SS or CA	Threshold = CongWin/2, CongWin = 1 MSS, Set state to "Slow Start"	Enter slow start
Duplicate ACK	SS or CA	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

TCP throughput

- What's the average throughout ot TCP as a function of window size and RTT?
  - Ignore slow start
- Let W be the window size when loss occurs.
- When window is W, throughput is W/RTT
- Just after loss, window drops to W/2, throughput to W/2RTT.
- Average throughout: .75 W/RTT

TCP Futures

- Example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- Requires window size W = 83,333 in-flight segments
- Throughput in terms of loss rate:

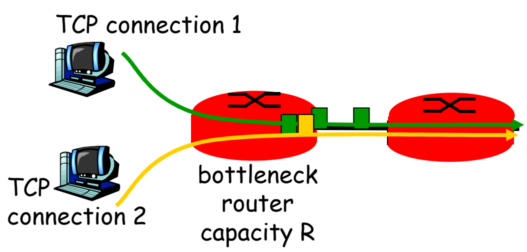
$$\frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

- ➜ $L = 2 \cdot 10^{-10}$ *Wow*
- New versions of TCP for high-speed needed!

97/101

TCP Fairness

Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K

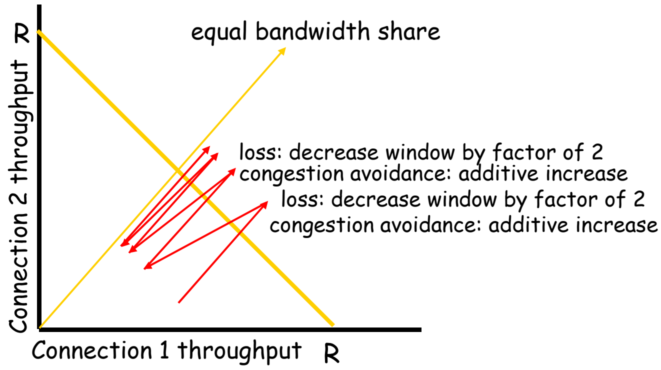


98/101

Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



Fairness (more)

Fairness and UDP

- Multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- Instead use UDP:
  - pump audio/video at constant rate, tolerate packet loss
- Research area: TCP friendly

Fairness and parallel TCP connections

- nothing prevents app from opening parallel cnctions between 2 hosts.
- Web browsers do this
- Example: link of rate R supporting 9 cnctions;
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2 !

Delay modeling

Q: How long does it take to receive an object from a Web server after sending a request?

Ignoring congestion, delay is influenced by:

- TCP connection establishment
- data transmission delay
- slow start

Notation, assumptions:

- Assume one link between client and server of rate R
- S: MSS (bits)
- O: object size (bits)
- no retransmissions (no loss, no corruption)

Window size:

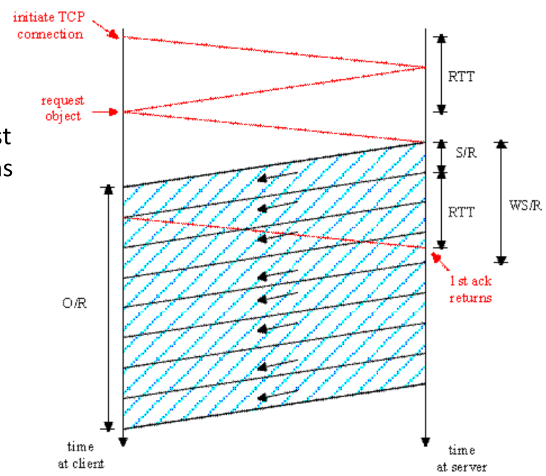
- First assume: fixed congestion window, W segments
- Then dynamic window, modeling slow start

Fixed congestion window (1)

First case:

WS/R > RTT + S/R: ACK for first segment in window returns before window's worth of data sent

delay = 2RTT + O/R

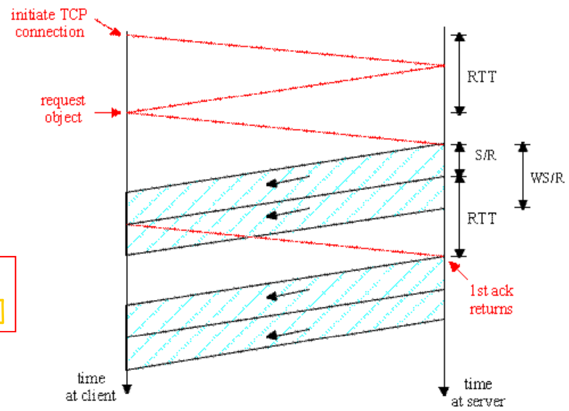


Fixed congestion window (2)

Second case:

- WS/R < RTT + S/R: wait for ACK after sending window's worth of data sent

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



TCP Delay Modeling: Slow Start (1)

Now suppose window grows according to slow start

Will show that the delay for one object is:

$$Latency = 2RTT + \frac{O}{R} + P\left[RTT + \frac{S}{R}\right] - (2^P - 1)\frac{S}{R}$$

where *P* is the number of times TCP idles at server:

$$P = \min\{Q, K-1\}$$

- where Q is the number of times the server idles if the object were of infinite size.

- and K is the number of windows that cover the object.

TCP Delay Modeling: Slow Start (2)

Delay components:

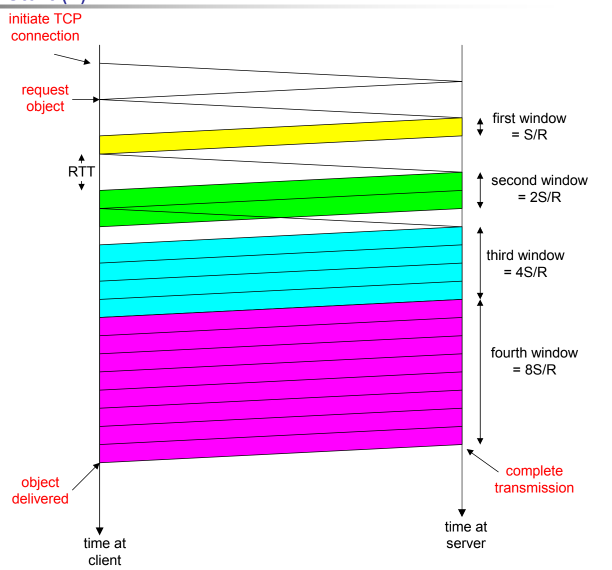
- 2 RTT for connection estab and request
- O/R to transmit object
- time server idles due to slow start

Server idles:
P = min{K-1,Q} times

Example:

- O/S = 15 segments
- K = 4 windows
- Q = 2
- P = min{K-1,Q} = 2

Server idles P=2 times



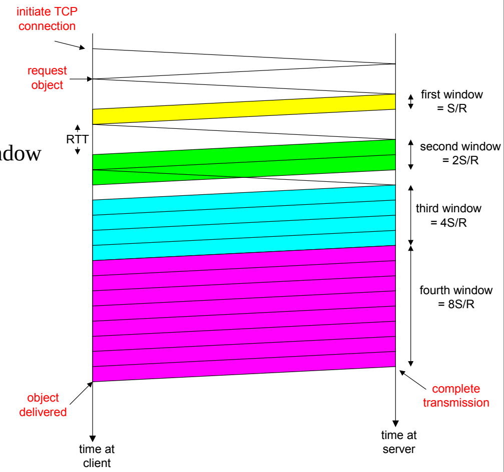
TCP Delay Modeling (3)

$$\frac{S}{R} + RTT = \text{time from when server starts to send segment until server receives acknowledgement}$$

$$2^{k-1}\frac{S}{R} = \text{time to transmit the kth window}$$

$$\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]^{+} = \text{idle time after the } k\text{th window}$$

$$\begin{aligned}\text{delay} &= \frac{O}{R} + 2RTT + \sum_{p=1}^{P} idleTime_p \\ &= \frac{O}{R} + 2RTT + \sum_{k=1}^{P}\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right] \\ &= \frac{O}{R} + 2RTT + P\left[RTT + \frac{S}{R}\right] - (2^{P} - 1)\frac{S}{R}\end{aligned}$$



TCP Delay Modeling (4)

Recall K = number of windows that cover object

How do we calculate K ?

$$
\begin{aligned}
K &= \min\{k : 2^0 S + 2^1 S + \cdots + 2^{k-1} S \geq O\} \\
&= \min\{k : 2^0 + 2^1 + \cdots + 2^{k-1} \geq O / S\} \\
&= \min\{k : 2^k - 1 \geq \frac{O}{S}\} \\
&= \min\{k : k \geq \log_2(\frac{O}{S} + 1)\} \\
&= \left\lceil \log_2(\frac{O}{S} + 1) \right\rceil
\end{aligned}
$$

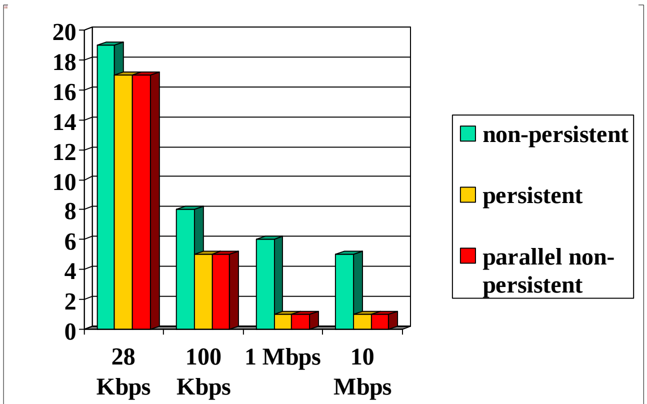
Calculation of Q, number of idles for infinite-size object, is similar (see HW).

HTTP Modeling

- Assume Web page consists of:
  - *1* base HTML page (of size *O* bits)
  - *M* images (each of size *O* bits)
- Non-persistent HTTP:
  - *M+1* TCP connections in series
  - *Response time = (M+1)O/R + (M+1)2RTT + sum of idle times*
- Persistent HTTP:
  - *2 RTT* to request and receive base HTML file
  - *1 RTT* to request and receive M images
  - *Response time = (M+1)O/R + 3RTT + sum of idle times*
- Non-persistent HTTP with X parallel connections
  - Suppose M/X integer.
  - 1 TCP connection for base file
  - M/X sets of parallel connections for images.
  - *Response time = (M+1)O/R + (M/X + 1)2RTT + sum of idle times*

HTTP Response time (in seconds)

RTT = 100 msec, O = 5 Kbytes, M=10 and X=5

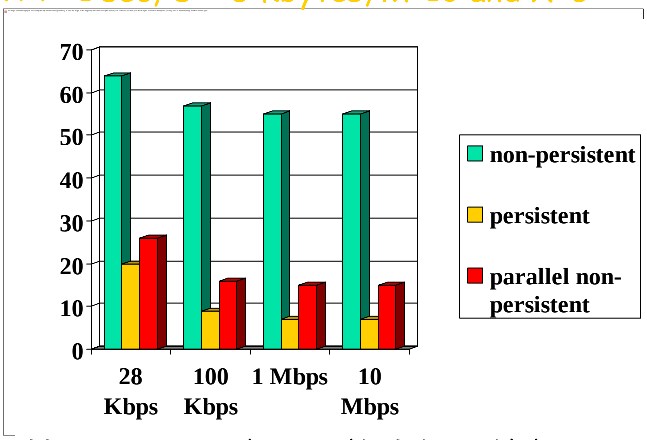


For low bandwidth, connection & response time dominated by transmission time.

Persistent connections only give minor improvement over parallel connections.

HTTP Response time (in seconds)

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



For larger RTT, response time dominated by TCP establishment & slow start delays. Persistent connections now give important improvement: particularly in high delay•bandwidth networks.

Chapter 3: Summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation and implementation in the Internet
  - UDP
  - TCP

Next:

- leaving the network "edge" (application, transport layers)
- into the network "core"

111/101