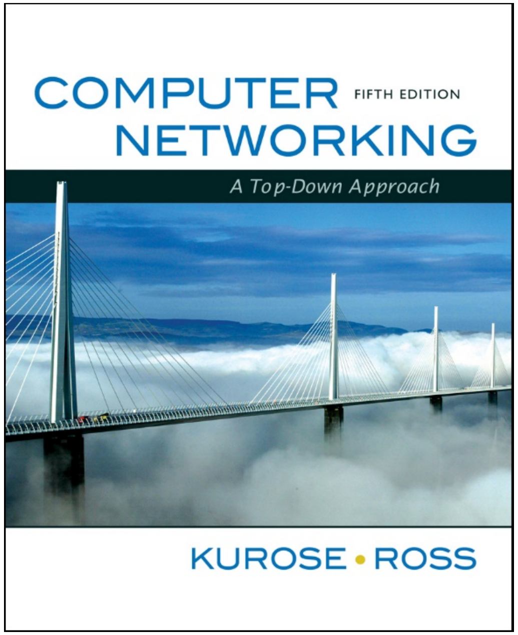
Chapter 3
Transport Layer



A note on the use of these ppt slides:

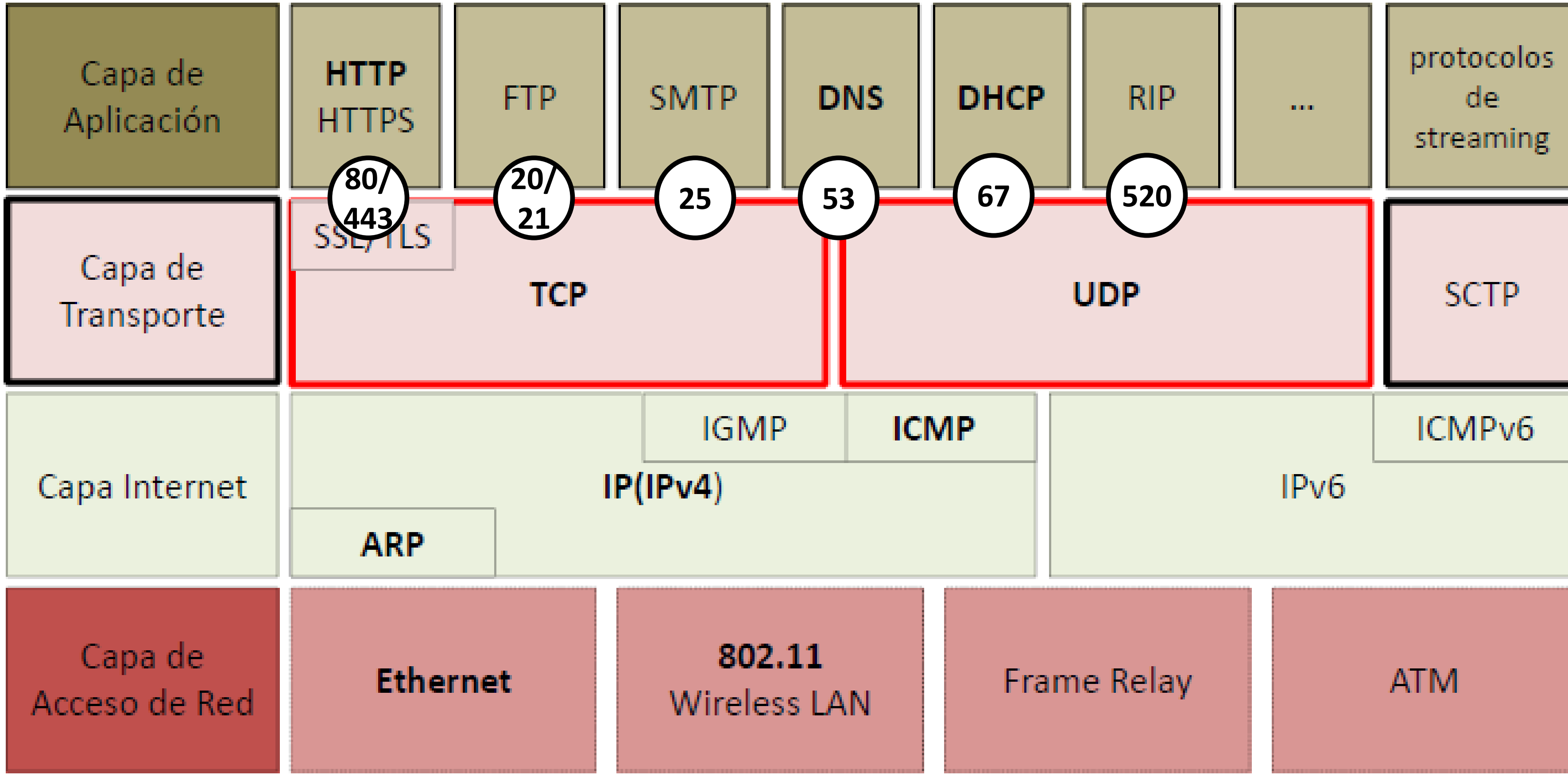
We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we'd like people to use our book!)
- If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

Computer Networking: A Top Down Approach
5th edition.
Jim Kurose, Keith Ross
Addison-Wesley, April 2009.

3 Gaia: Garraio geruza



3 Gaia: Garraio geruza

Helburua:

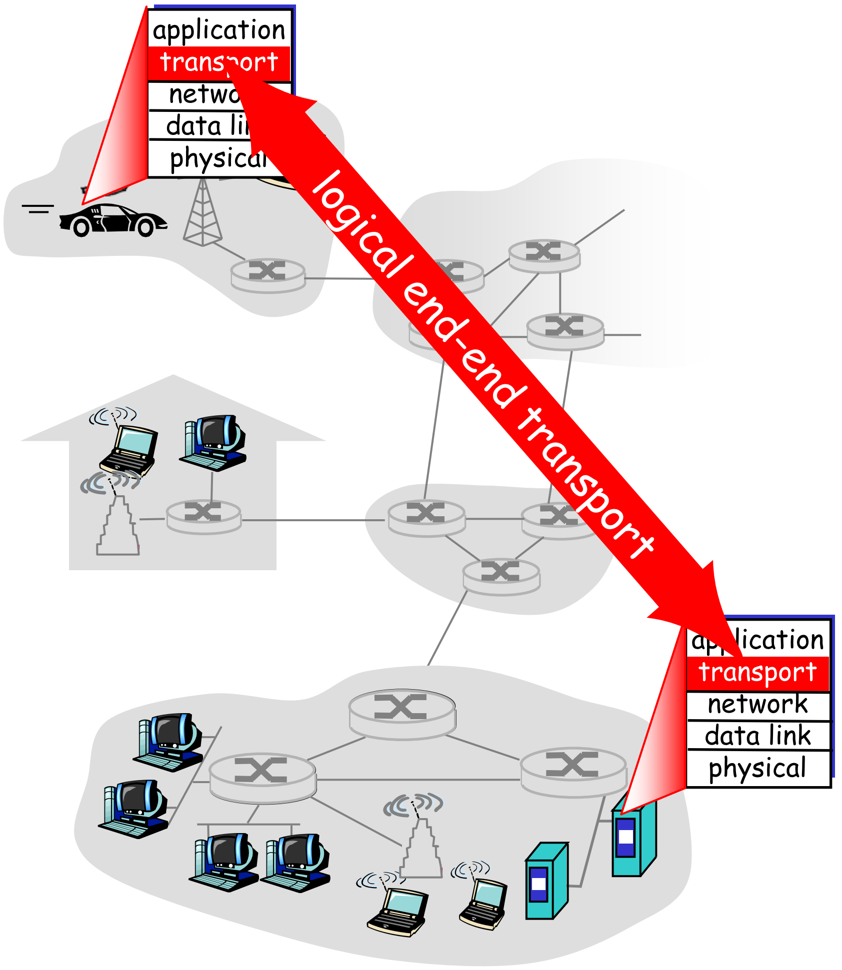
- Garraio geruzak ematen dituen zerbitzuak ulermena:
  - multiplexing/demultiplexing
  - Informazioaren garraio fidagarria
  - Fluxu kontrola
  - Pilaketen kontrola
- Garraio geruza Interneten:
  - UDP: konexiorik gabe
  - TCP: konexiora orientatuta
  - TCP pilaketen kontrola

3. Gaia:

- 3.1 Garraio geruzaren zerbitzuak
- 3.2 Multiplexing and demultiplexing
- 3.3 Konexiorik gabeko garraioa: UDP
- 3.4 Informazio garraio fidagarriaren oinarriak
- 3.5 Konexiorako bideratutako transportea: TCP
  - Segmentuen estruktura
  - Informazio transferentzia fidagarria
  - Fluxuaren kontrola
  - Konexioaren kudeaketa
- 3.6 Pilaketen kontrolaren oinarriak
- 3.7 TCP-ren pilaketen kontrola

Garraio zerbitzu eta protokoloak

- *Komunikazio logikoa* gauzatzen du host desberdinetan dauden prozesuen artean
- Garraio protokoloak terminaletan lan egiten dute
  - Bidaltzaileak: Aplikazioaren mezuak segmentuetan zatikatzen ditu eta sare geruzara pasatzen ditu
  - Jasotzaileak: jasotako segmentuekin mezuak berrosatzen ditu eta aplikazio geruzara pasatzen ditu
- Protokolo desberdinak daude
  - Interneten: TCP eta UDP



Garraio geruza vs sare geruza

- *Sare geruza:* Komunikazio logikoa host artean
- *Garraio geruza:* Komunikazio logikoa prozesuen artean
  - Sare geruzak dituen zerbitzuaz fido da hauek hobetzen

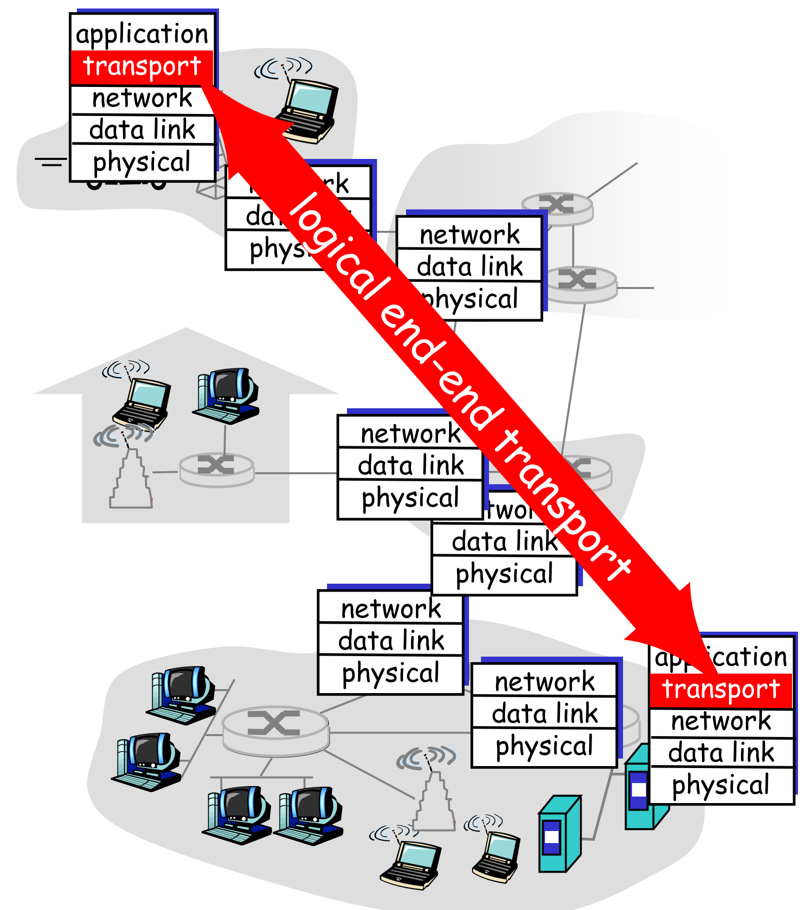
Household analogy (liburuan):

12 kids sending letters to 12 kids

- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill
- network-layer protocol = postal service

Internetaren garraio geruzaren protokoloak

- Bideraketa ordenatua, fidagarria (TCP)
  - Pilaketen kontrola
  - Fluxuaren kontrola
  - Konexioa
- Bideraketa ez-ordenatua, ez-fidagarria : UDP
  - no-frills extension of “best-effort” IP



3. Gaia:

- 3.1 Garraio geruzaren zerbitzuak
- 3.2 Multiplexing and demultiplexing
- 3.3 Konexiorik gabeko garraioa: UDP
- 3.4 Informazio garraio fidagarriaren oinarriak
- 3.5 Konexiorako bideratutako transportea: TCP
  - Segmentuen estruktura
  - Informazio transferentzia fidagarria
  - Fluxuaren kontrola
  - Konexioaren kudeaketa
- 3.6 Pilaketen kontrolaren oinarriak
- 3.7 TCP-ren pilaketen kontrola

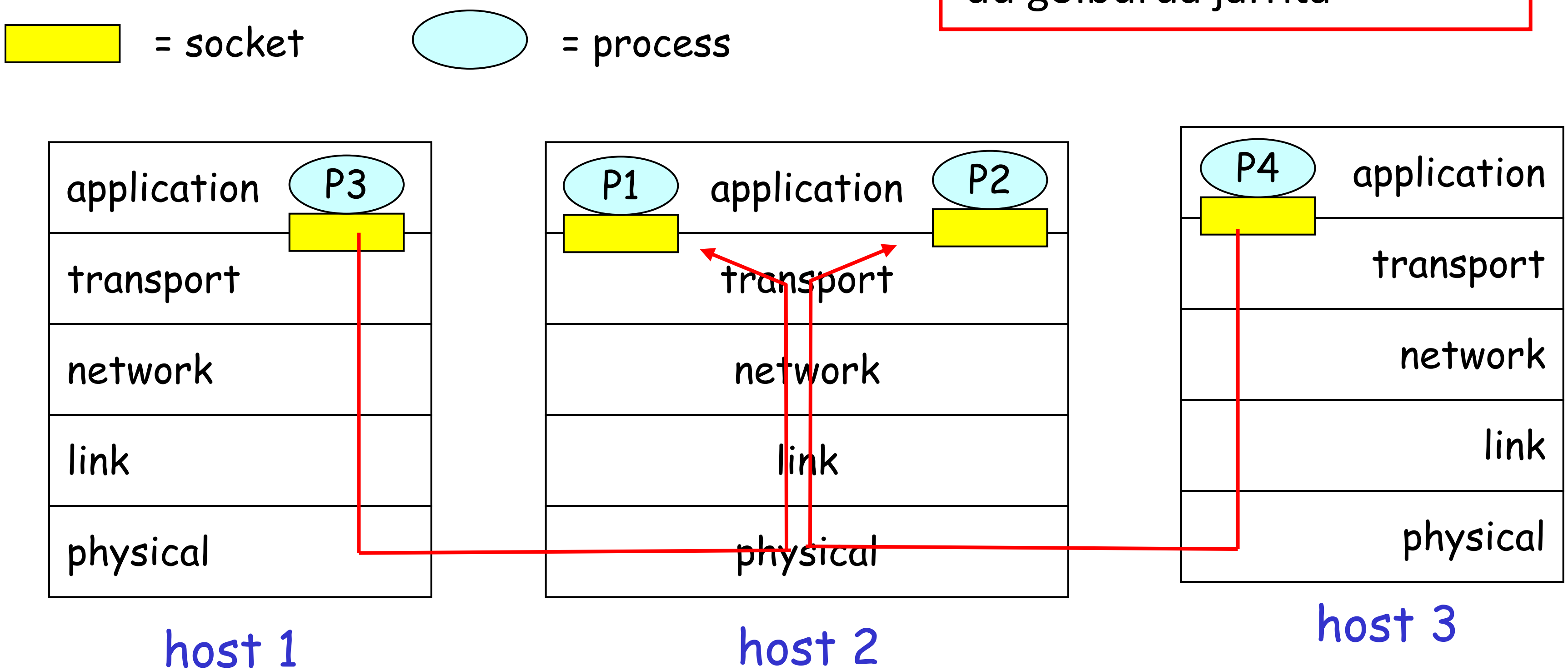
Multiplexing/demultiplexing

Demultiplexazioa jasotzailean

Dagokion socket-i bidaltzen dizkio jasotako segmentuak

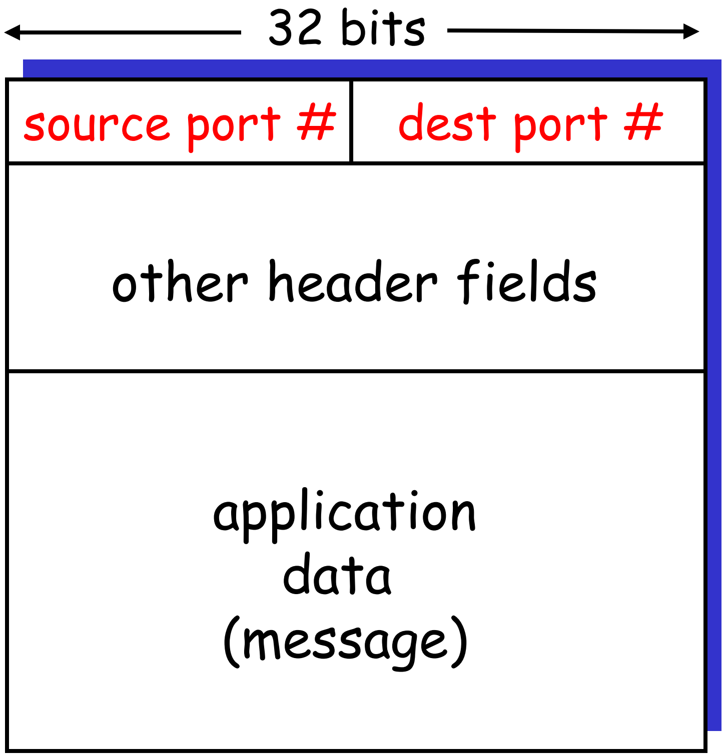
Multiplexazioa igorlean

Informazioa socket desberdinetatik jasotzen du eta segmentuetan banatzen du goiburua jarrita



Nola demultiplexatu

- Host-ak IP datagramak jasotzen ditu
  - Datagrama bakoitzak igorlearen IP helbidea dauka, baita jasotzailearen IP ere
  - Datagrama bakoitzak garraio geruzaren segmentu bat dakar
  - Segmentu bakoitzak igorle eta jasotzailearen portuen zenbakiak dakarzki
- Host-ak IP helbideak eta atakak erabiltzen ditu segmentuak dagokien socket-era bidaltzeko



TCP/UDP segment format

Konexiorik gabeko demultiplexazioa

- Socketak ataka zenbakiarekin sortzen dira:

```
DatagramSocket mySocket1 = new
   DatagramSocket(12534);
DatagramSocket mySocket2 = new
   DatagramSocket(12535);
```

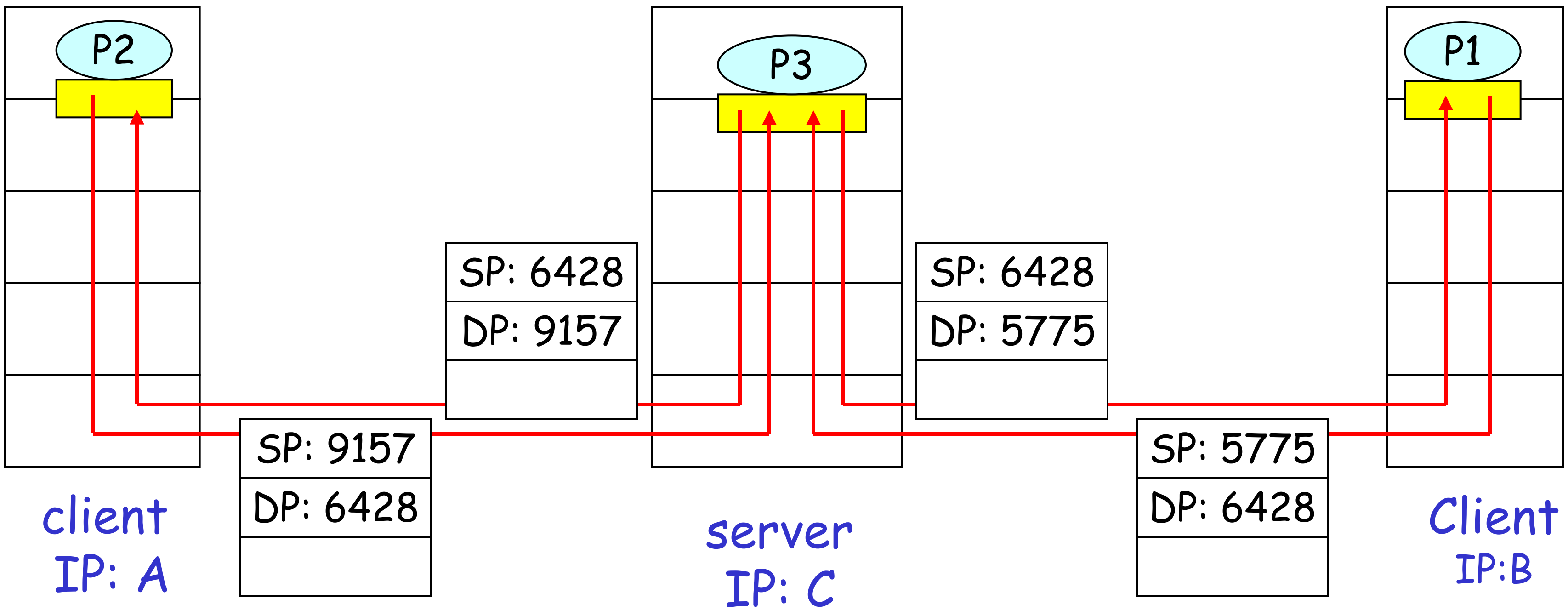
- UDP socketa tupla bikotzaz identifikatzen da:

  (dest IP address, dest port number)

- Hostak UDP segmentua jasotzen duenean:
  - Segmentuaren ataka zenbakia begiratzen du
  - UDP segmentua ataka hori duen socket-era bideratzen du
- IP datagrama igorle desberdinek (IP helbide/ataka desbedinek) bidalitako segmentuak socket berera bidera daitezke

Konexiorik gabeko demux (jarraipena)

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



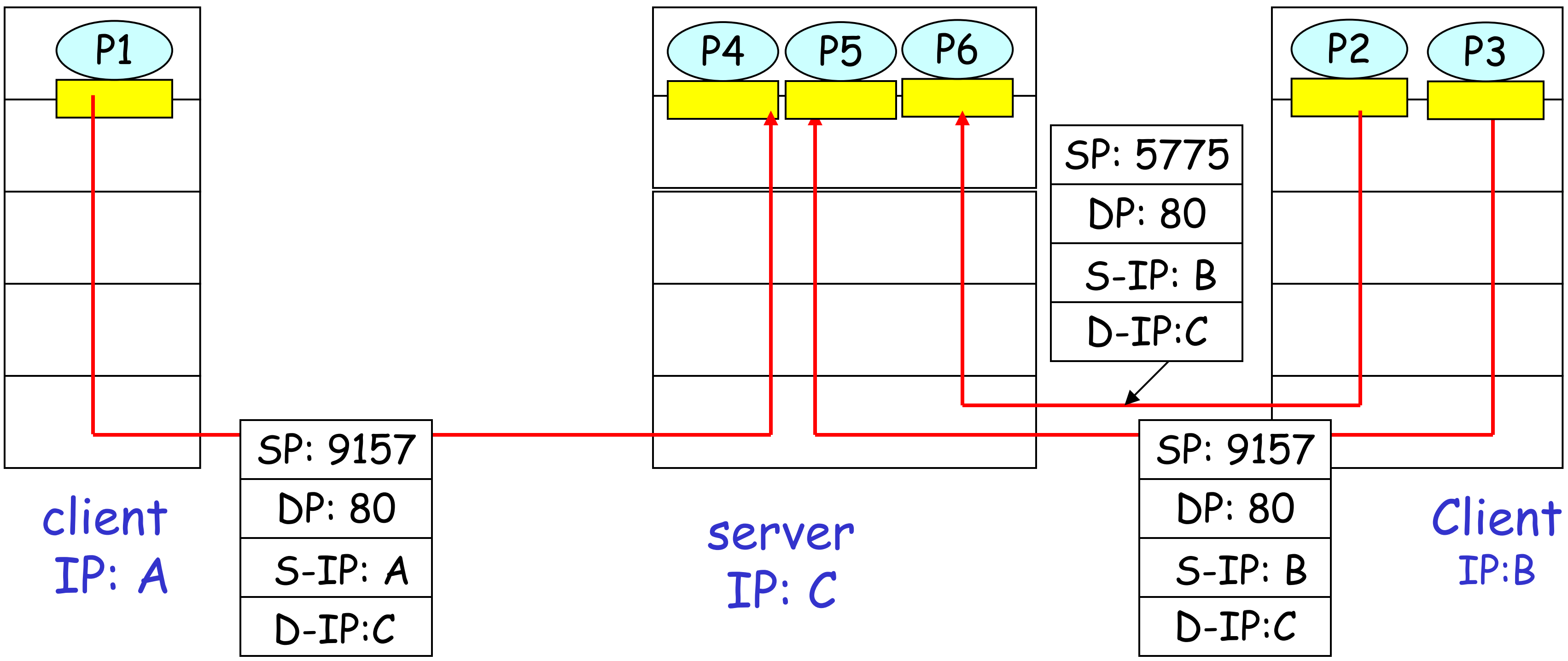
SP (Source Port) “return address” ematen du

Konexiora bideratutako demux

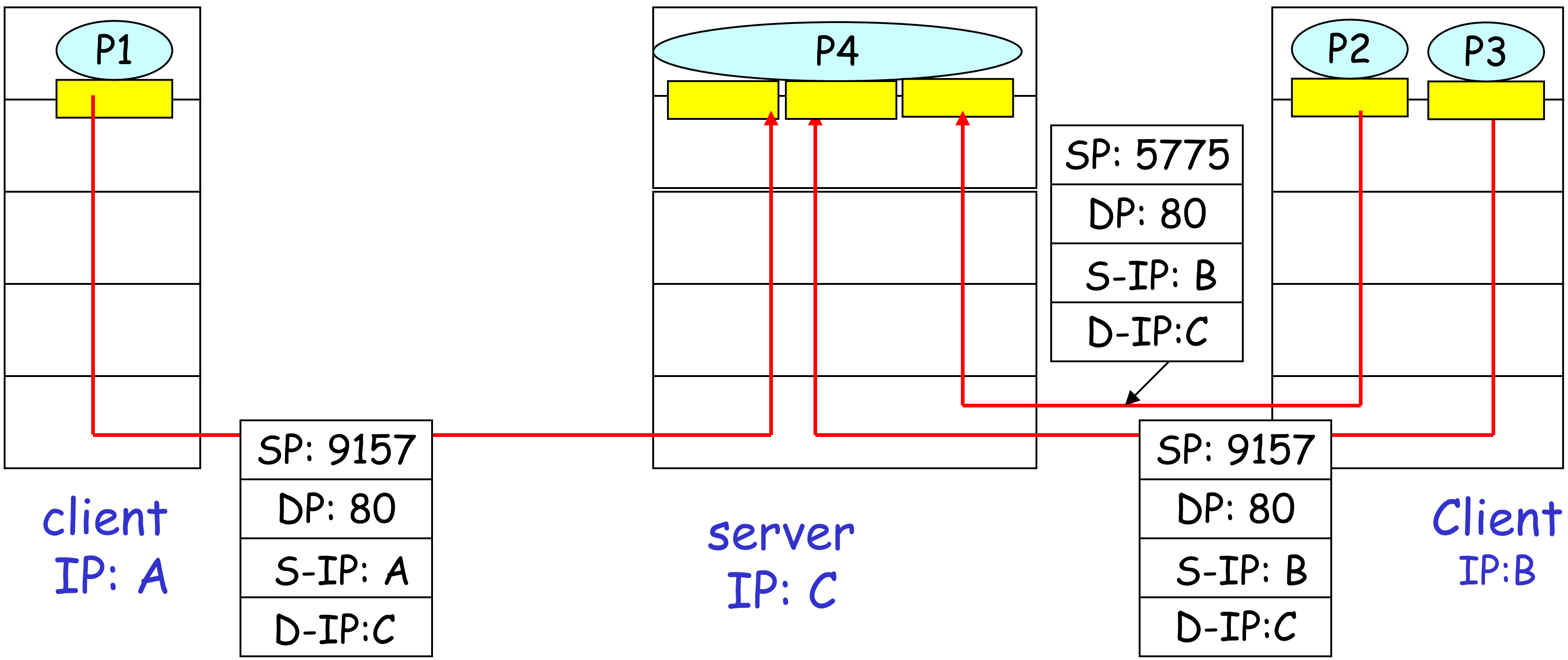
- TCP socketa 4-tupla bidez identifikatuta:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- Host jasotzaileak lau balioak erabiltzen ditu segmentua dagokion socket-ari bideratzeko

- Zerbitzariek, TCP socket desberdinak jaso dezakete aldi berean:
  - Socket bakoitza horren 4-tupla bidez identifikatzen da
- Web zerbitzariek socket desberdinak dituzte konektatzen den kliente bakoitzeko
  - non-persistent HTTP will have different socket for each request

Konexiora bideratutako demux (jarraipena)



Konexiora bideratutako demux : Threads erabiltzen dituen Web Server-a



3. Gaia:

- 3.1 Garraio geruzaren zerbitzuak
- 3.2 Multiplexing and demultiplexing
- 3.3 Konexiorik gabeko garraioa: UDP
- 3.4 Informazio garraio fidagarriaren oinarriak
- 3.5 Konexiorako bideratutako transportea: TCP
  - Segmentuen estruktura
  - Informazio transferentzia fidagarria
  - Fluxuaren kontrola
  - Konexioaren kudeaketa
- 3.6 Pilaketen kontrolaren oinarriak
- 3.7 TCP-ren pilaketen kontrola

TCP eta UDP Protokoloak: Alderaketa

- Garraio geruzaren protokolorik garrantzitsuenak:
  - TCP: Transmision Control Protocol
  - UDP: User Datagram Protocol
- Aplikazio desberdinen komunikazioak kudeatzen dituzte
- Funtzio desberdinak inplementatzen dituzte:

TCP-ren betekizunak	UDP-ren betekizunak
• Aplikazioen multiplexazioa • Segmentazioa • Akatsen kontrola • Fluxu kontrola • Pilaketen kontrola • Galdutako datuen berbidalketa • Konexio/deskonexioa	• Aplikazioen multiplexazioa • Segmentazioa • Akatsen kontrola (aukeran)

UDP: User Datagram Protocol [RFC 768]

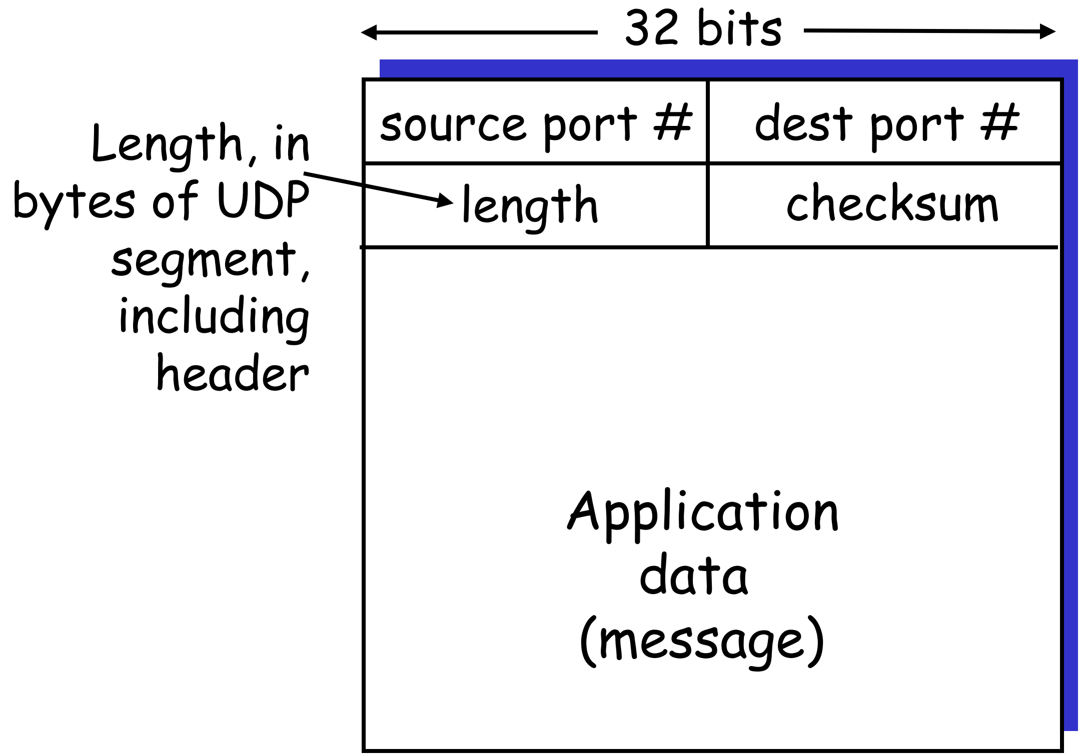
- Interneterako oinarrizko garraio protokoloa
- "best effort" service, UDP segmentuak:
  - Gal daitezke
  - Desordenatuta hel daitezke
- *Konexiorik gabe:*
  - Ez dago konexio protokolorik igorle eta jasotzaile artean
  - UDP segment bakoitzak indibidualki tratatzen da

Zergatik UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

UDP: gehiago

- Streaming aplikazioetan erabilita
  - loss tolerant
  - rate sensitive
- Beste erabilerak
  - DNS
  - SNMP
- Garraio fidagarria UDP gainean: fidagarritasuna aplikazio mailan gehitzen zaio
  - Aplikazio bakoitzak beraren errore kudeaketa!



UDP segment format

bitak	0 – 7	8 – 15	16 – 23	24 – 31
0	Jatorriko helbidea			
32	Helburuko helbidea			
64	Zeroak	Protokoloa	UDP luzera	
96	Jatorriko portua		Helburuko portua	
128	Luzera		Checksum	
160+	Data			

Checksum egiteko kontutan hartzen den informazioa IPv4an

UDP checksum: RFC 768-etan

Helburua: Akatzen datekzioa (e.g., flipped bits) igorritako segmentuetan

Igorleak:

- Segmentuaren informazioa 16-bit-eko integer bezala tratatzen ditu
- checksum: addition (1's complement sum) of segment contents
- Igorleak checksumaren balioa jartzen du UDP segmentuare checksum esparruan

Jasotzaileak:

- Jasotako segmentuen checksum egiten du
- Alderatzen du segmentuak dakarren checksum-arekin:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?* More later ....

Internet Checksum Example

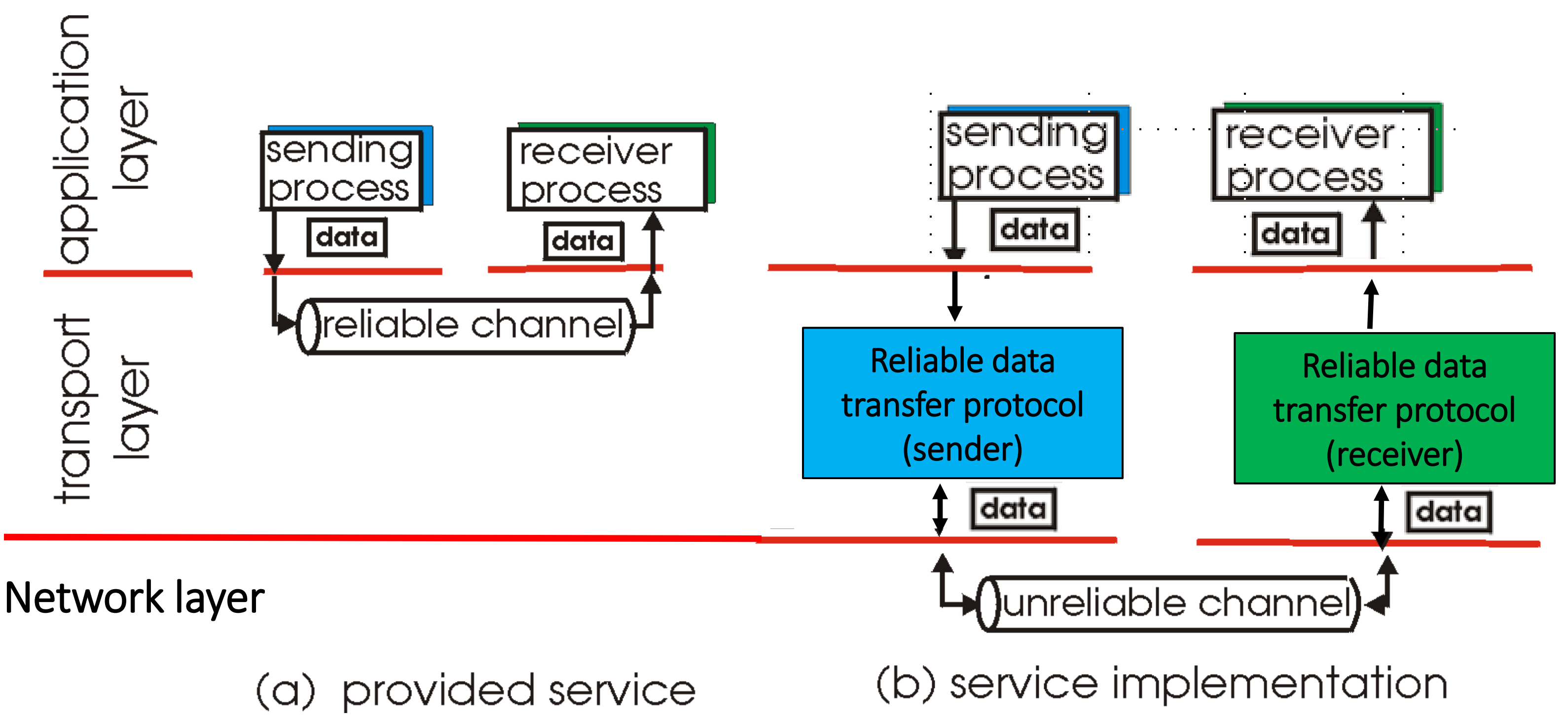
- Note
  - When adding numbers, a carryout from the most significant bit needs to be added to the result
- Example: add two 16-bit integers

		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

3. Gaia:

- 3.1 Garraio geruzaren zerbitzuak
- 3.2 Multiplexing and demultiplexing
- 3.3 Konexiorik gabeko garraioa: UDP
- 3.4 Informazio garraio fidagarriaren oinarriak (bukaeran informazio bezala)
- 3.5 Konexiorako bideratutako transportea: TCP
  - Segmentuen estruktura
  - Informazio transferentzia fidagarria
  - Fluxuaren kontrola
  - Konexioaren kudeaketa
- 3.6 Pilaketen kontrolaren oinarriak
- 3.7 TCP-ren pilaketen kontrola

Informazio garraio fidagarriaren oinarriak



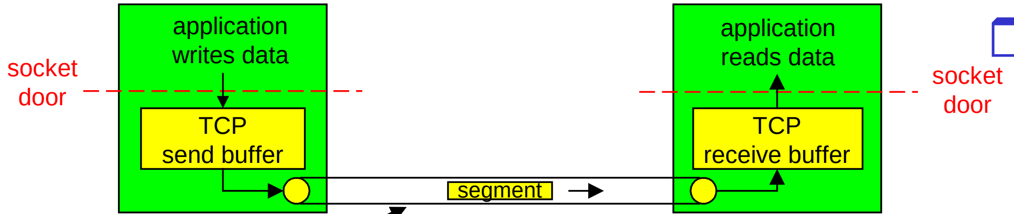
3. Gaia:

- 3.1 Garraio geruzaren zerbitzuak
- 3.2 Multiplexing and demultiplexing
- 3.3 Konexiorik gabeko garraioa: UDP
- 3.4 Informazio garraio fidagarriaren oinarriak
- 3.5 Konexiorako bideratutako transportea: TCP
  - Segmentuen estruktura
  - Informazio transferentzia fidagarria
  - Fluxuaren kontrola
  - Konexioaren kudeaketa
- 3.6 Pilaketen kontrolaren oinarriak
- 3.7 TCP-ren pilaketen kontrola

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

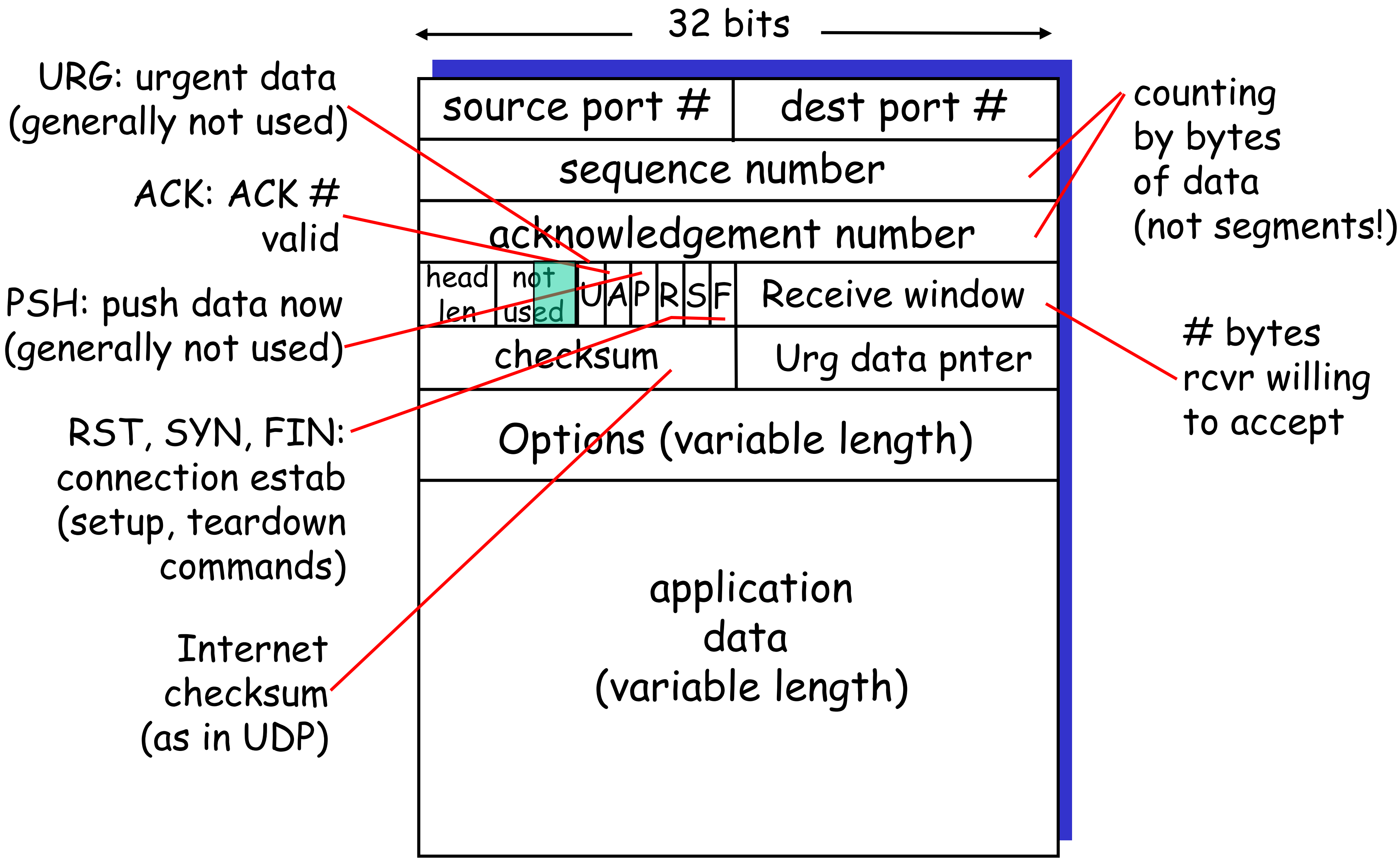
- point-to-point:
  - Igorle bat, jasotzaile bat
- fidagarria, ordenatutako *byte steam:*
- Pipelined, bideratuta:
  - TCPren pilaketa eta fluxu kontrolak lehioaren tamaina finkatzen du
- *igorle & jasotzailearen buffers*



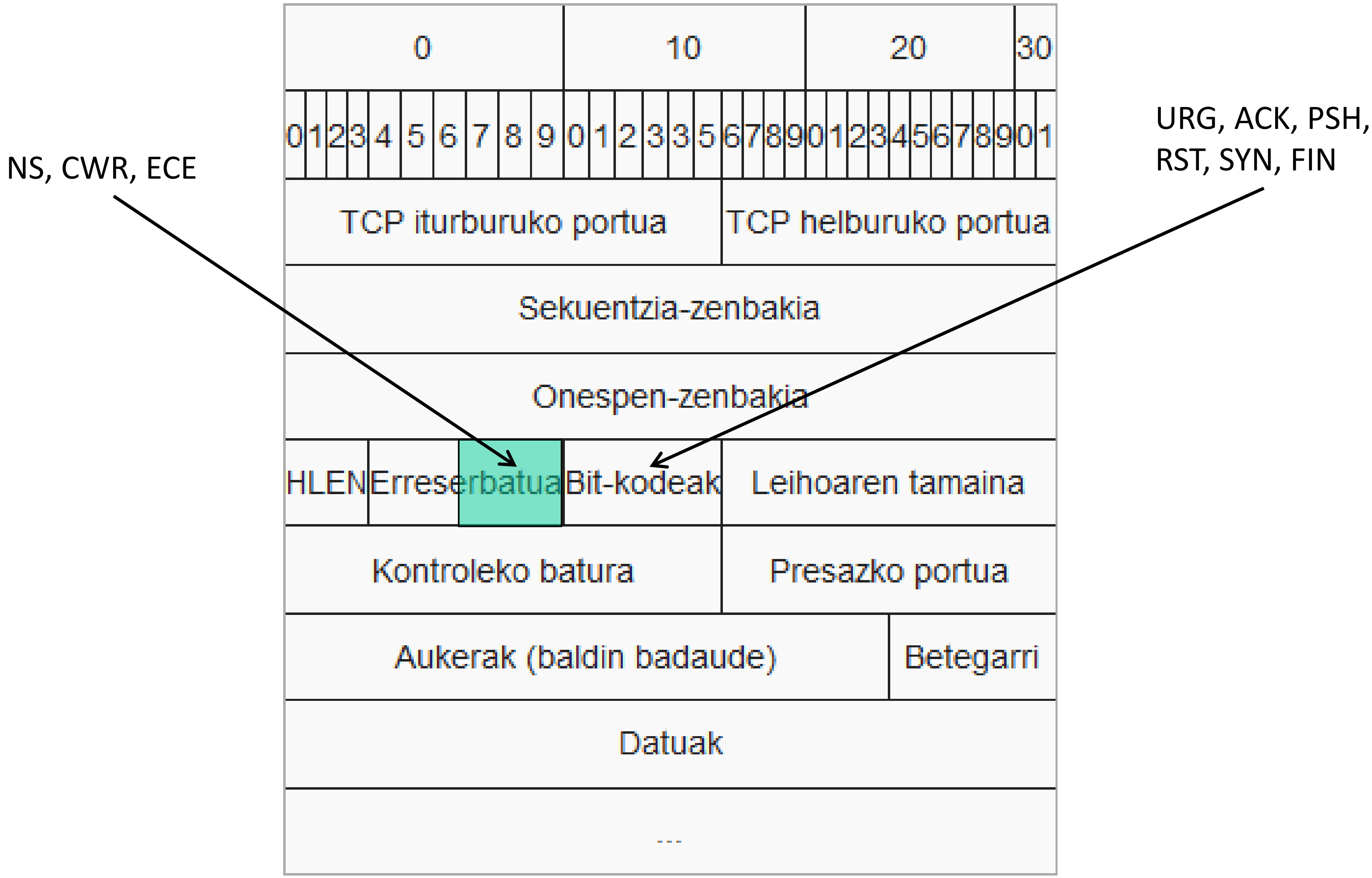
Baina sareko elementuetan ez du ezer inplementatzen

- full duplex data:
  - Informazioaren fluxu bi-directional konexio berean
  - MSS: maximum segment size
- Konexiora orientatuta:
  - **handshaking** (kontrol mezuen trukaketa) igorle eta jasotzailearen egoerarekin informazio transferentzia aurretik
- Fluxuaren kontrola:
  - Igorleak ez du hartzailea "itoko"

TCP segment structure



TCP segment structure (Wikipedia)



TCP segment structure (Wikipedia)

- **TCP iturburuko portua** (16 bit). Iturburuko makinaren portu-zenbakia.
- **TCP helburuko portua** (16 bit). Helburuko makinaren portu-zenbakia.
- **Sekuentzia-zenbakia** (32 bit). **TCP** informazioaren korrontearen barruan uneko datuen posizioa adierazten du.
- **Onespen-zenbakia** (32 bit). Eremu honekin aurretik jasotako byteak zuzen jaso direla adierazteko erabiltzen du helburuko makinak.
- **HLEN** (4 bit). Goiburuko luzera adierazten du. 32 bitetako multiploetan adierazita, segmentuaren tamaina. Gutxienekoa 5 (1001 bitarrean) litzateke, 20 bytetako datu gabeko segmentuari dagokiona.
- **Erreserbatua** (6 bit). Etorkizun batean erabiltzeko.
- **Bit-kodeak** (6 bit). Segmentuaren asmoa zehazten dute.
    - **NS.** Hautazkoa, jasotzaileak erabiltzen du datuak jasotzen dituela jakinarazteko
    - **CWR.** ECE jaso dela jakinarazten du. Pilaketen kontrako tresna erabili du
    - **ECE.** Pilaketa gertatu dela jakinarazten du
    - **URG**. Presazko erakuslearen datuak bilatu eta prozesatu behar dira.
    - **ACK**. Onespen-segmentua da, beraz onespen-zenbakia kontuan hartzen da. SYN eta FIN kodeekin konexioaren ezarpena eta askapena adierazten dute. Segmentu berberak noranzko baten datuak bidal ditzake eta komunikazioaren beste noranzkoaren onespenak.
    - **PSH**. Buffer osoa betetzen den arte segmentua buferretan ez gordetzeko erakuslea, honela iritsi ahala aplikazioari bidaliko zaio. Push eragiketa.
    - **RST**. Uneko konexioa etetea.
    - **SYN**. Konexio eskaera. Sekuentzia-zenbakia nondik hasiko den adierazten du. ACK kodearekin batera (ACK=1;SYN=1) konexio-eskaerari onespena ematen zaio.
    - **FIN**. Konexio amaitu nahi dela adierazteko, ez baitaude datu gehiagorik bidaltzeko.
- **Leihoaren tamaina** (16 bit). Konexio batentzako helburuko makinak onartuko duen buferren tamaina zehazten du.
- **Kontroleko batura** (24 bit). Uneko segmentuaren erroren kontroleko batura. Iturburuko eta helburuko IPren helbideak ere sartzen dira (sasigoiburua).
- **Presazko erakuslea** (8 bit). Presazko datuak non hasten diren zehazten du.
- **Aukerak** (aldakorrak). Aukera hau zehaztuta badago, onartuko den segmenturen gehienezko tamaina zehazten du.
- **Betegarri**. Eremu honekin segmentua 32 bitetako multiplokoa izatea behartuko litzateke.
- **Datuak**. Aplikazioari bidaltzen zaion informazioa.

TCP seq. #'s and ACKs

Seq. #'s:

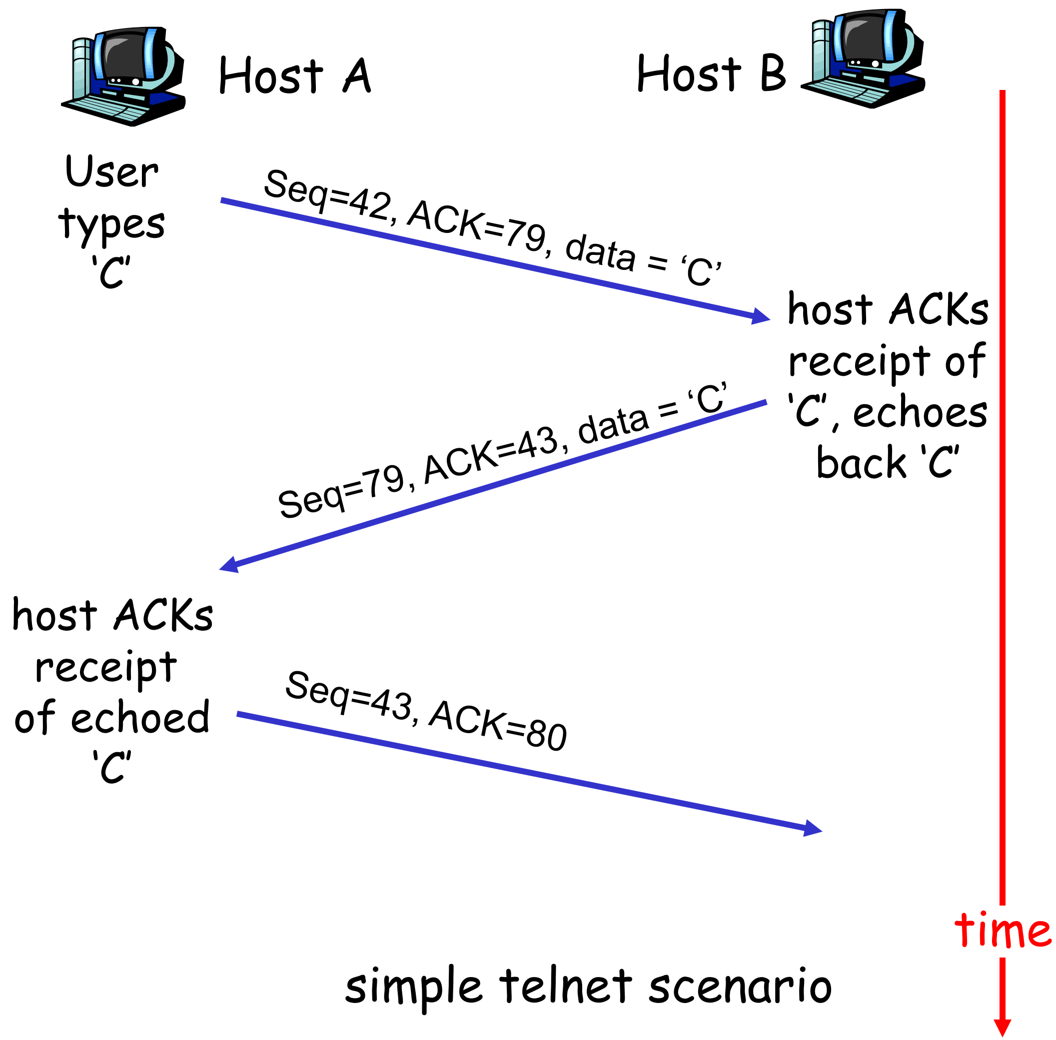
- byte stream "number" of first byte in segment's data

ACKs:

- Beste aldetik espero den hurrengo byte-aren **Seq #**
- cumulative ACK

Q: Nola kudeatzen dira ordenatu gabeko segmentuak?

- A: TCP espezifikazioek ez dute azaltzen, - inplementatzaileak definitu behar



simple telnet scenario

Joan-etorri denbora

TCP Round Trip Time and Timeout

Q: Nola ezarri Timeout balioa?

- RTT baino luzeago
  - baina RTT aldakorra da
- txikiegia: premature timeout
  - Behar ez diren birbidalketak
- luzeegia: segmentuen galeren aurreko erantzun geldoa

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**

3. Gaia:

- 3.1 Garraio geruzaren zerbitzuak
- 3.2 Multiplexing and demultiplexing
- 3.3 Konexiorik gabeko garraioa: UDP
- 3.4 Informazio garraio fidagarriaren oinarriak
- 3.5 Konexiorako bideratutako transportea: TCP
  - Segmentuen estruktura
  - Informazio transferentzia fidagarria
  - Fluxuaren kontrola
  - Konexioaren kudeaketa
- 3.6 Pilaketen kontrolaren oinarriak
- 3.7 TCP-ren pilaketen kontrola

Informazio transferentzia fidagarria

- TCP zerbitzu fidagarria sortzen du IP sareko zerbitzu ez fidagarriaren gainean
- Pipelined segmentuak
- Cumulative acks
- TCP-k erretransmisio tenporizadore sinplea erabiltzen du
- Retransmissions are triggered by:
  - timeout events
  - duplicate acks
- Hasteko, demagun TCP sinplifikatu bat:
  - Bikoiztutako ACK-ak arbuiatzen dira
  - Fluxu eta pilaketa kontrolak arbuiatzen dira

TCP igorlearen gertakariak:

Aplikaziotik jasotako informazioa:

- Segmentua sortu sekuentzia zenbakiarekin **seq #**
- **seq #**: Bidaltzen den segmentuari dagokion informazio fluxuaren lehen byte zenbakia
- Abiarazi erlojua (timer) martxan ez badago (think of timer as for oldest unacked segment)
- expiration interval: `TimeOutInterval`

timeout:

- Berbidali timeout sortarazi duen segmentua
- Erlojua berabiatu

Jasotako ACK:

- Oraindik onartu ez den segmentu baten ACK jasotzen bada
  - Eguneratu onartu (ACK) behar direnen zerrenda
  - Erlojua berabiazi falta diren segmentuentzat

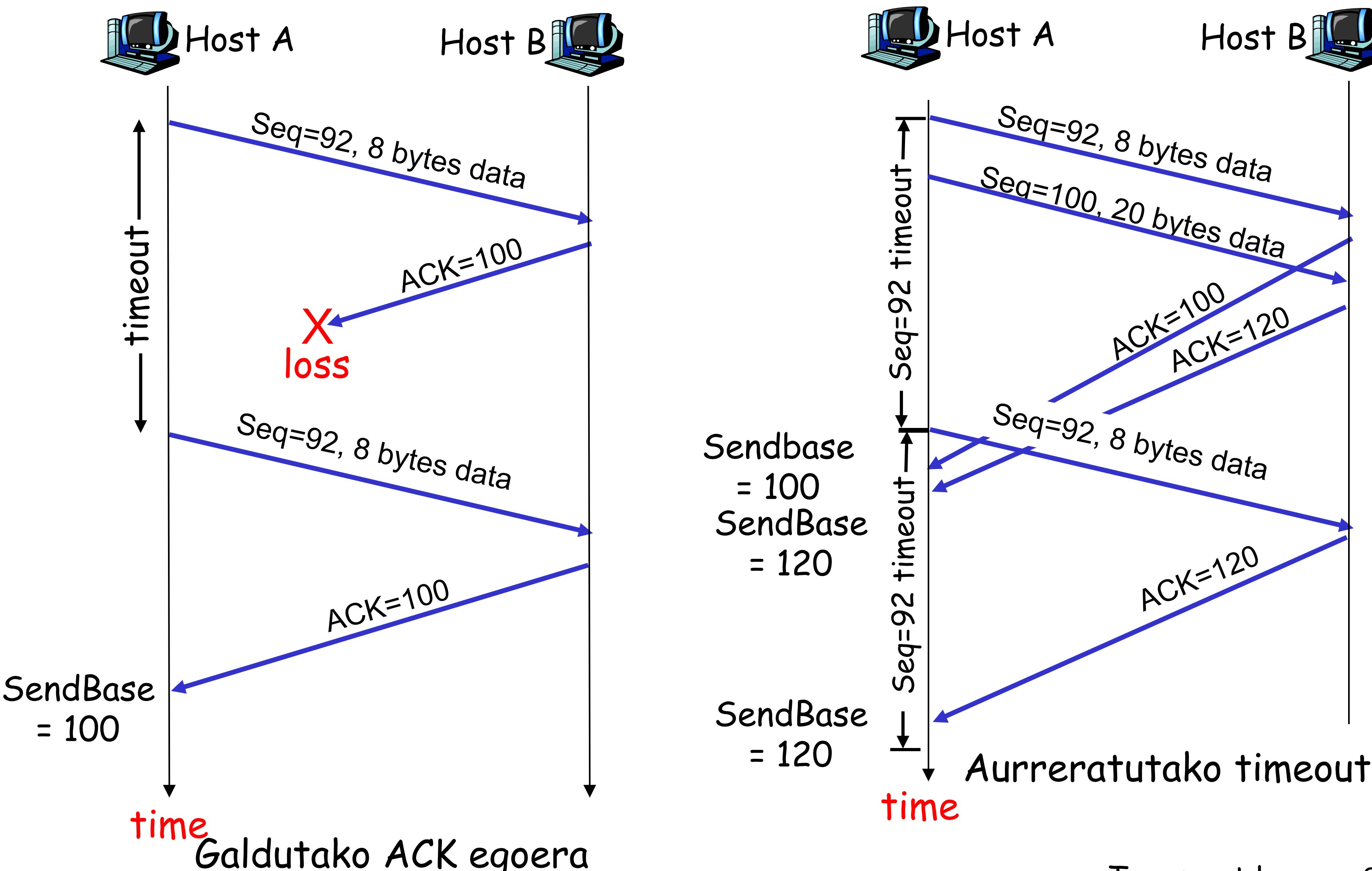
TCP igorle (sinplifikatuta)

```
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 */
```

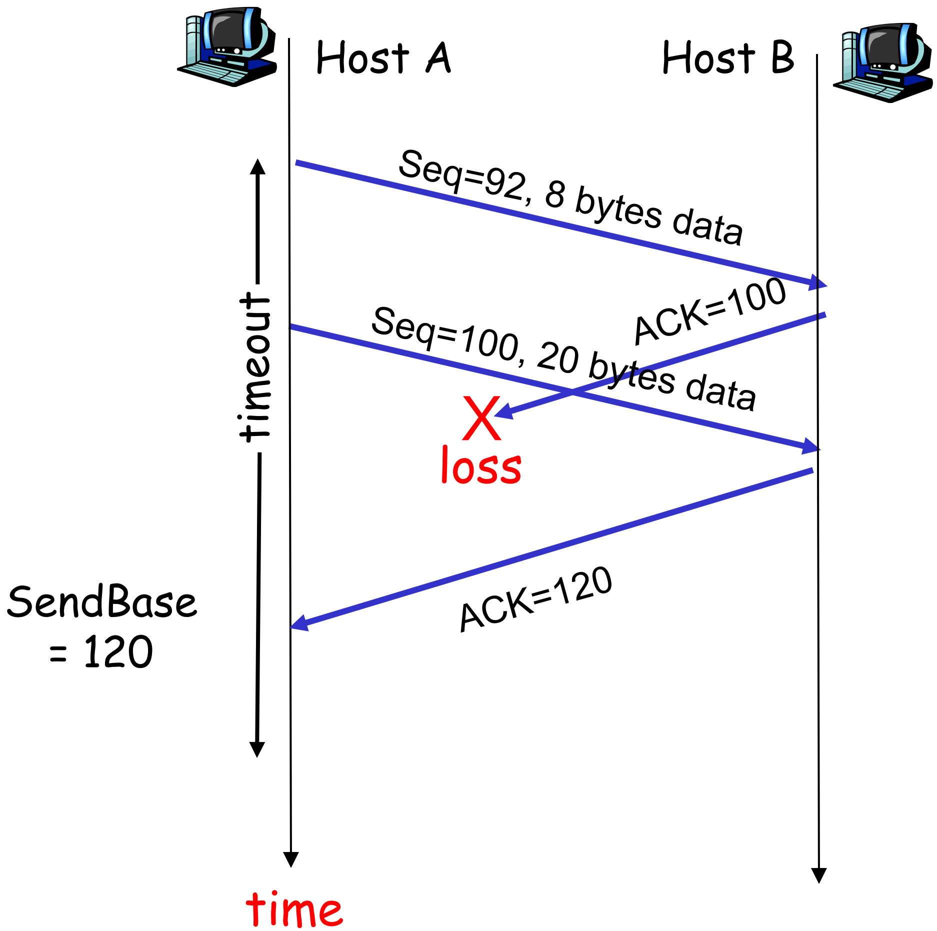
Comment:
- SendBase-1: last cumulatively ack'ed byte

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

TCP: erretransmisio egoerak



TCP: erretransmisio egoerak (gehiago)



Cumulative ACK scenario

TCP: ACK sorrera [RFC 1122, RFC 2581]

Gertaerak jasotzailean	TCP jasotzailearen ekintzak
Segmentuak ordenean heltzen dira espero den **seq** #. Aurreko informazio guztiaren ACK-k bidalita	Atzeratutako ACK. Itzaron beste segmentua 500ms arte. Beste segmenturik ez badago ACK bidali
Segmentuak ordenatuta heltzen dira espero den **seq** #. Beste segmentu Batzuen ACK bidali gabe	Berehala bidali ACK metagarria Honekin errekonozitzen dira horraino helduta ko segmentu guztiak
Heltzen den segmentua desodenatuta Espero den baino **seq** # handiago. Zuloa	Berehala bidali *duplicate ACK*, espero den hurrengo byte-aren **seq** # adieraziz
Zuloa erabat edo zatika betetzen duen segmentu baten iristera	Berehala bidali ACK, espero den segmentuaren **seq** # zuloaren goimuga da

Erretransmisio azkarra

- Time-out denbora, nahiko luzea sarritan:
  - Itzaron-denbora handia segmentua birbidali arte
- Errepikatutako ACK-en bidez galdutako segmentuak detektatu
  - Igorleak batzutan segmentuak bidalttzen ditu bata bestearen atzean
  - Segmentu bat galtzen bada, ACK errepikatuak egon daitezke

TCP fast retransmit

if sender receives 3 ACKs for same data ("triple duplicate ACKs"), resend unacked segment with smallest seq #

- likely that unacked segment lost, so **don't wait for timeout**

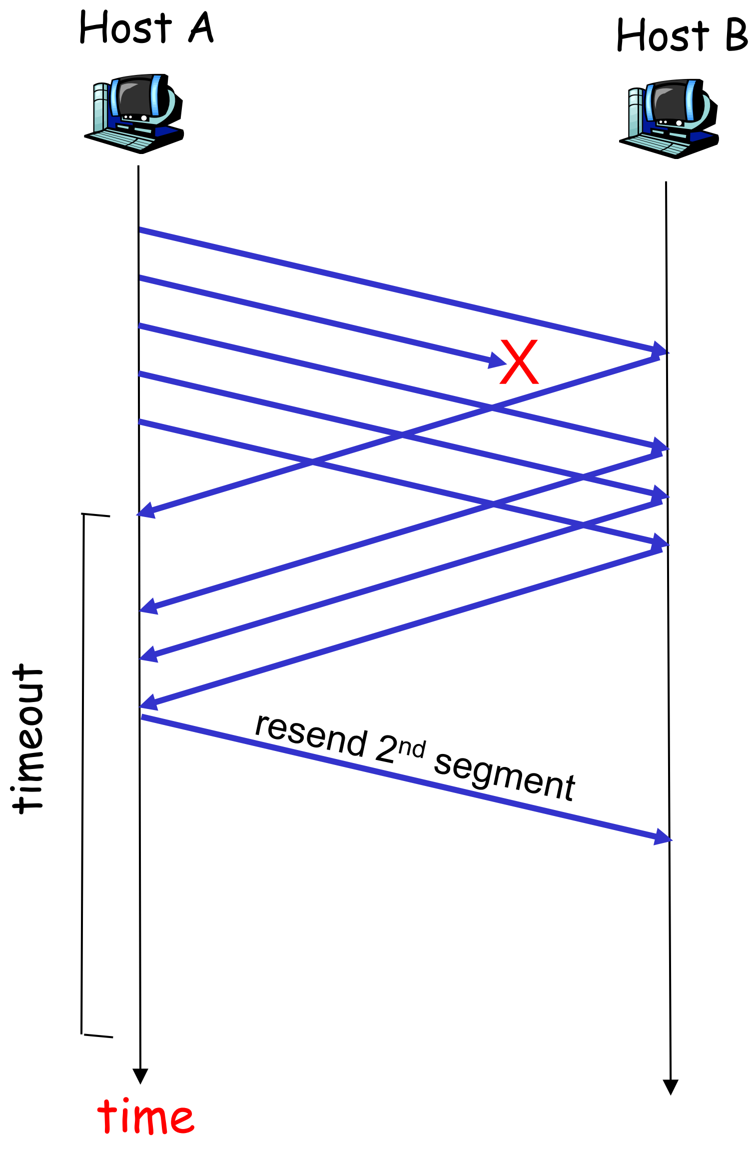


Figure 3.37 Resending a segment after triple duplicate ACK

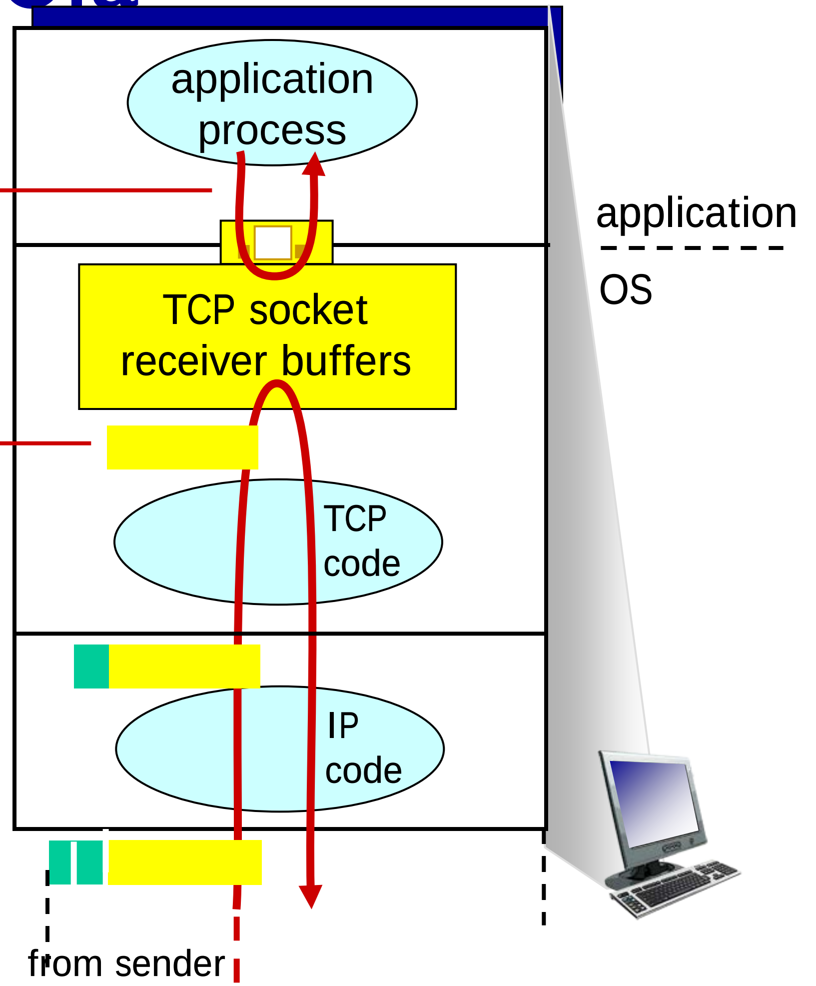
3. Gaia:

- 3.1 Garraio geruzaren zerbitzuak
- 3.2 Multiplexing and demultiplexing
- 3.3 Konexiorik gabeko garraioa: UDP
- 3.4 Informazio garraio fidagarriaren oinarriak
- 3.5 Konexiorako bideratutako transportea: TCP
  - Segmentuen estruktura
  - Informazio transferentzia fidagarria
  - Fluxuaren kontrola
  - Konexioaren kudeaketa
- 3.6 Pilaketen kontrolaren oinarriak
- 3.7 TCP-ren pilaketen kontrola

TCP Fluxuaren kontrola

application may
remove data from
TCP socket buffers ….

… slower than TCP
receiver is delivering
(sender is sending)



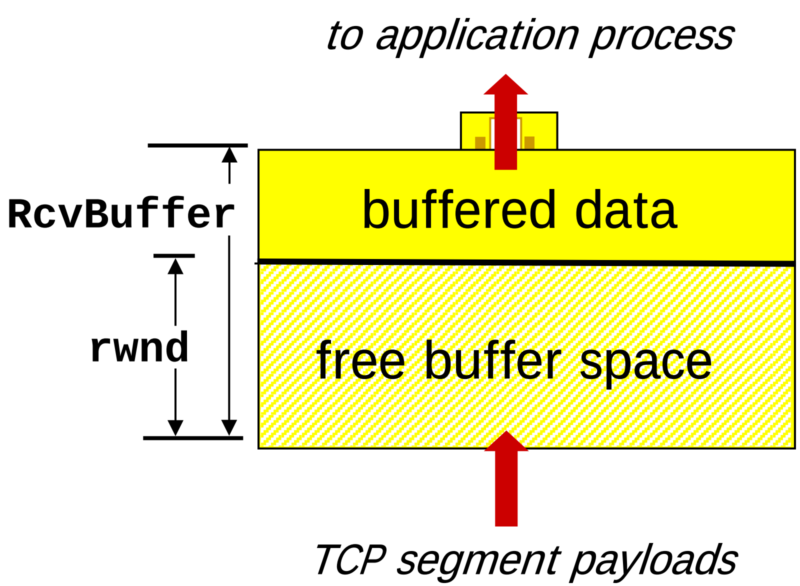
receiver protocol stack

flow control

Jasotzaileak igorlea kontrolatzen du, igorleak ez du jasotzailearen bufferra beteko arinegi transmitituz

TCP flow control

- Jasotzaileak "jakinarazten" du buffer-ean duen tokia, TCP goiburuan **`rwnd`** balioa sartuta *receiver-to-sender* segmentuetan
  - **`RcvBuffer`** balioa socketen aukeren bidez ezartzen da (balio tipikoa 4096 bytes da)
  - Sistema eragile batzuek **`RcvBuffer`**-en balioa ajustatzen du
- Igorleak ez-ACK informazioa mugatzen du jasotzailearen **`rwnd`** balioaren bidez
- Jasotzailearen buferra ez dela gaineratuko bermatzen du



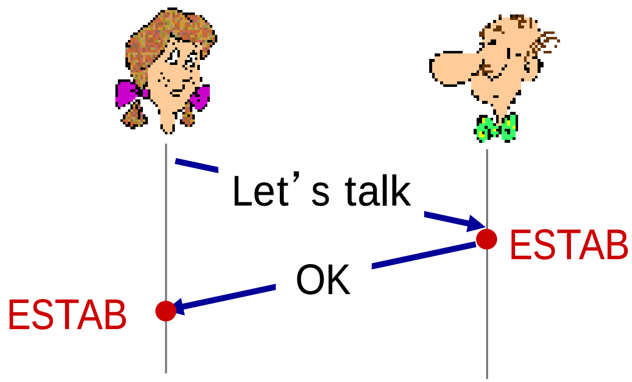
receiver-side buffering

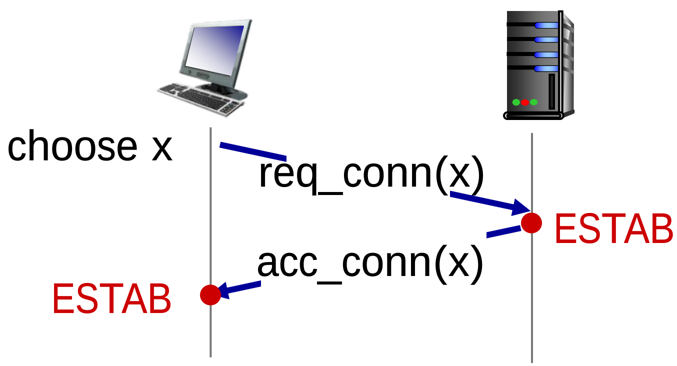
3. Gaia:

- 3.1 Garraio geruzaren zerbitzuak
- 3.2 Multiplexing and demultiplexing
- 3.3 Konexiorik gabeko garraioa: UDP
- 3.4 Informazio garraio fidagarriaren oinarriak
- 3.5 Konexiorako bideratutako transportea: TCP
  - Segmentuen estruktura
  - Informazio transferentzia fidagarria
  - Fluxuaren kontrola
  - Konexioaren kudeaketa
- 3.6 Pilaketen kontrolaren oinarriak
- 3.7 TCP-ren pilaketen kontrola

Konexio bat onartzen

2-way handshake:





Q: sareetan funtzionatuko du beti?

- Atzerapen aldakorrak
- Birbidalitako mezuak (e.g. req_conn(x)) mezuak galtzen direnean
- Mezuen berantolaketa
- Beste aldea ezin ikusi

TCP Konexioaren kudeaketa

Recall: TCP igorleak eta jasotzaileak konexioa ezartzen dute informazio segmentuak trukatzen hasi aurretik

- TCP aldagaien hasieratze:
  - seq. #s
  - buffers, fluxu kontrolaren informaziorako (ad. **RcvWindow**)
- *client:* konexioa abiaraztendu

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

- *server:* erantzuten du

Socket connectionSocket = welcomeSocket.accept();

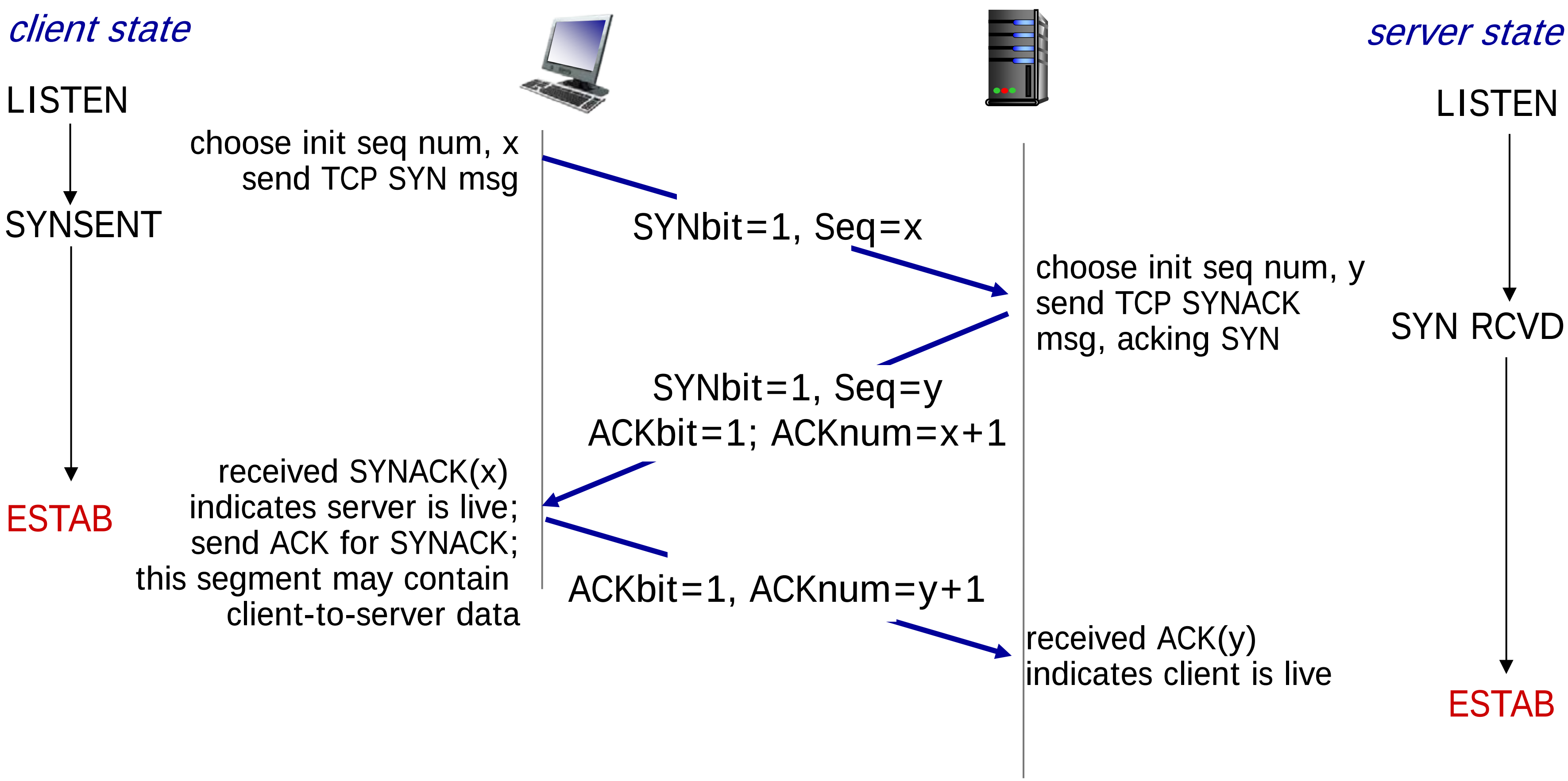
Three way handshake:

1. Pausua: bezeroak TCP SYN segmentua bidaltzen dio zerbitzariari
   - Hasierako **seq #**-ekin
   - Informazioarik gabe

2. Pausua: Zerbitzariak SYN jasotzen du, SYNACK segmentuarekin erantzuten du:
   - Zerbitzariak gordetako bufferekin
   - Zerbitzariaren hasierako **seq #**

3. Pausua: bezeroak SYNACK jasotzen du, ACK segmentuarekin erantzuten du, honek informazioa eduki dezake

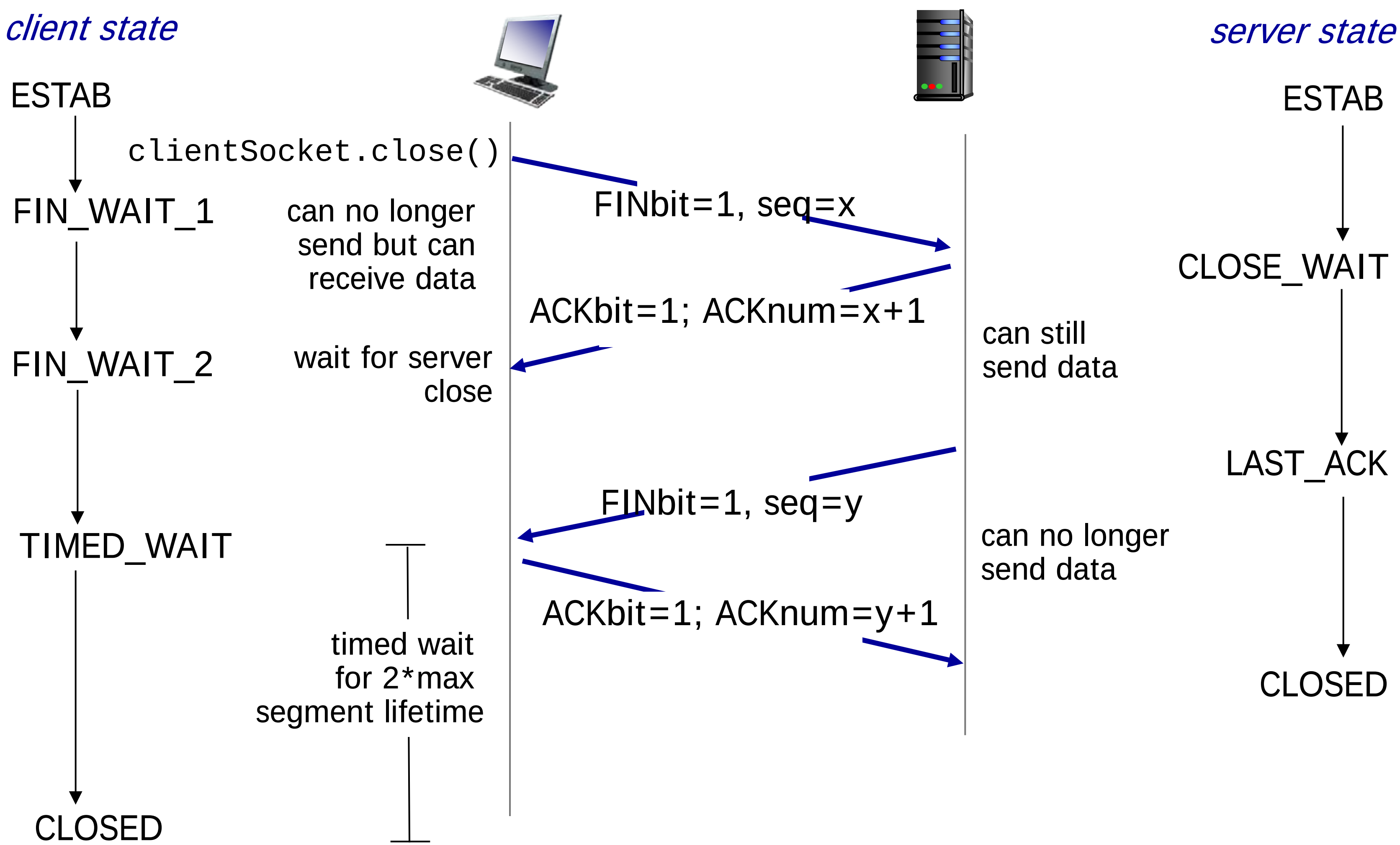
TCP 3-way handshake



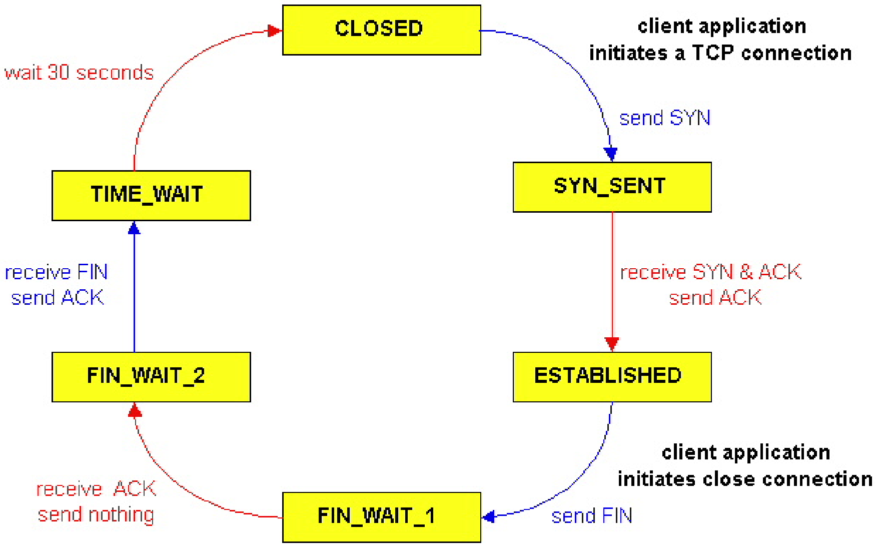
TCP: closing a connection

- Bezeroak eta zerbitzariak, biak izten dute konexioa beraren aldean
  - send TCP segment with FIN bit = 1
- Erantzun jasotako FIN, ACK-rekin
  - FIN jasotzen denean, ACK beste aldeko FIN-ekin nahas daiteke
- Batera ematen diren FIN trukaketak kudea daitezke

TCP: closing a connection

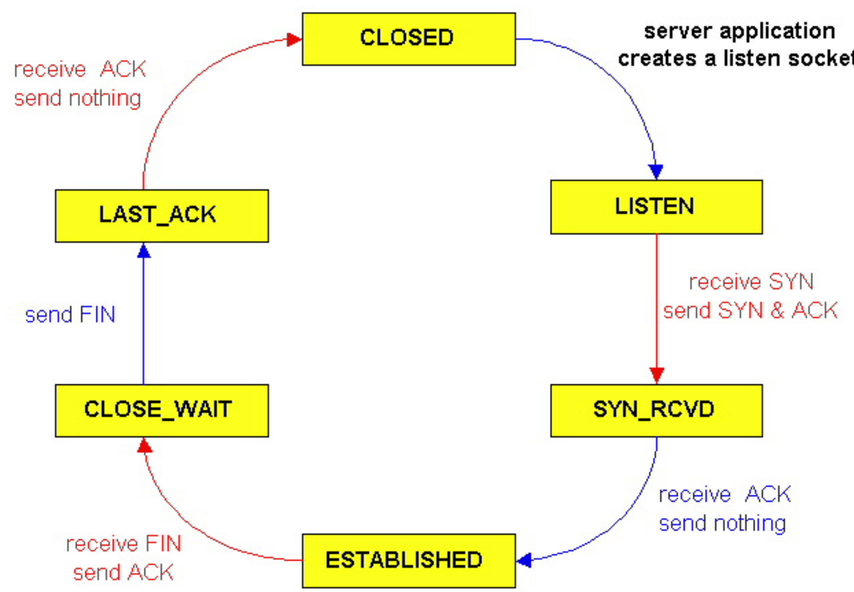


TCP Konexioaren kudeaketa (jarraipena)



TCP client lifecycle

TCP server lifecycle



3. Gaia:

- 3.1 Garraio geruzaren zerbitzuak
- 3.2 Multiplexing and demultiplexing
- 3.3 Konexiorik gabeko garraioa: UDP
- 3.4 Informazio garraio fidagarriaren oinarriak
- 3.5 Konexiorako bideratutako transportea: TCP
  - Segmentuen estruktura
  - Informazio transferentzia fidagarria
  - Fluxuaren kontrola
  - Konexioaren kudeaketa
- 3.6 Pilaketen kontrolaren oinarriak (Bukaeran)
- 3.7 TCP-ren pilaketen kontrola (Bukaeran)
- 3.8 Bukaerako kontzeptuak

3. Gaia:

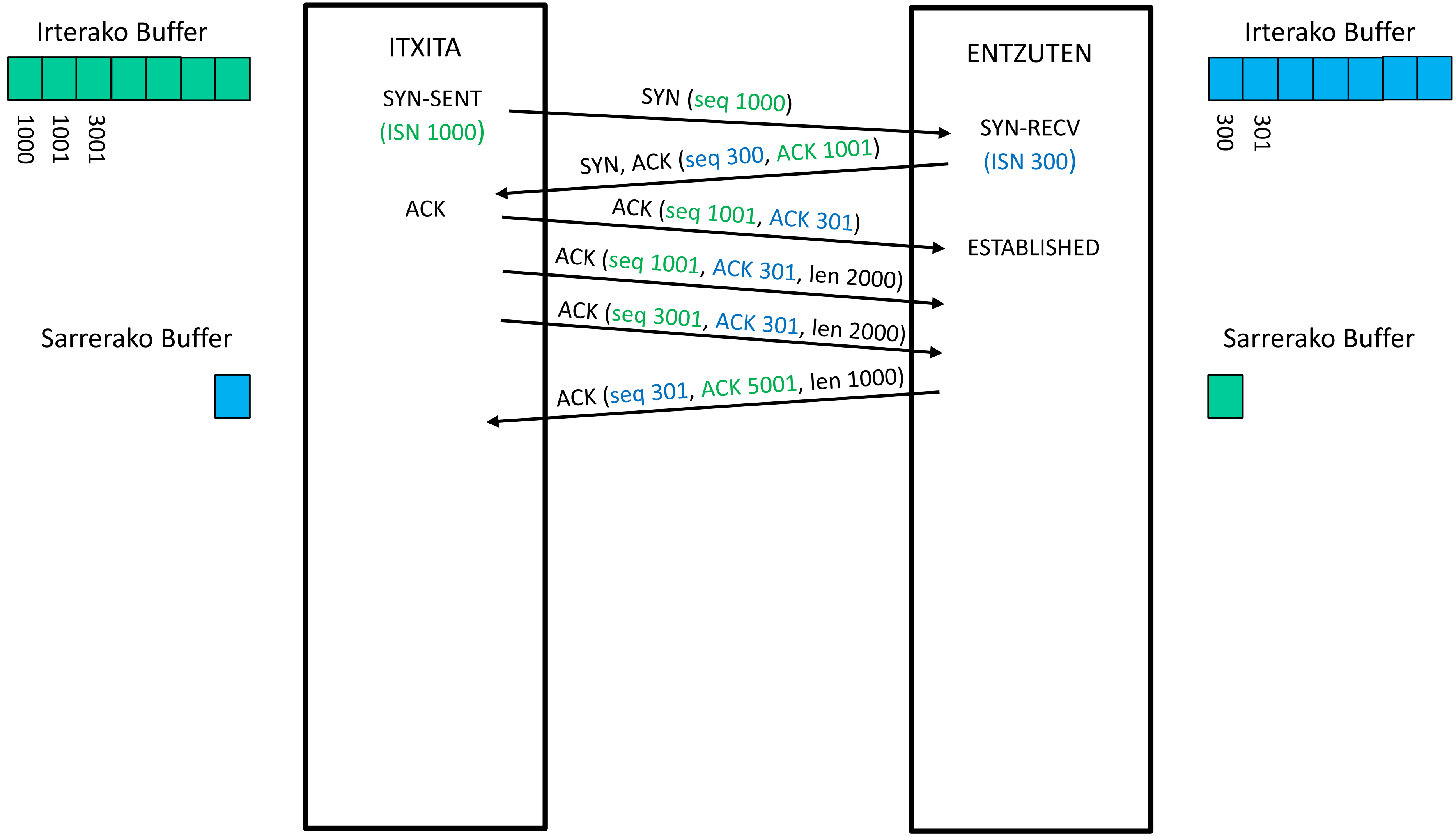
- 3.1 Garraio geruzaren zerbitzuak
- 3.2 Multiplexing and demultiplexing
- 3.3 Konexiorik gabeko garraioa: UDP
- 3.4 Informazio garraio fidagarriaren oinarriak
- 3.5 Konexiorako bideratutako transportea: TCP
- 3.6 Pilaketen kontrolaren oinarriak (Bukaeran)
- 3.7 TCP-ren pilaketen kontrola (Bukaeran)
- 3.8 Bukaerako kontzeptuak

TCP/UDP Portuak

- **Portu zenbakia**: 2 byte (0-65535)
- Hiru esparru:
  - **0-1023: Well Know Ports**, Normalean erabiltzen diren zerbitsuetan erabiltzen dira
  - **1024-49151: Portu erregistratuak**, IANAk asignatzen ditu aplikazio desberdinetan erabiltzeko garatzailearen eskariz
  - **49152-65535: Pribatuak edo dinamikoak**, Bezeroetan erabilitak. Era dinamikoan (ausazkoan) esleitzen dira konexia sortzerakoan

Zerbitzua	Portua	TCP	UDP
ECHO	7	X	X
Day time	13	X	X
FTP	20-21	X	
SSH	22	X	
TelNet	23	X	
SMPT	24	X	
DNS	53	X	X
BOOTP	67		X
TFPT	69		X
HTTP	80	X	
POP3	110	X	
NTP	123		X
SNMP	189		X
LDAP	389	X	
HTTPS	443		X

TCP: Datu trukaketa



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:

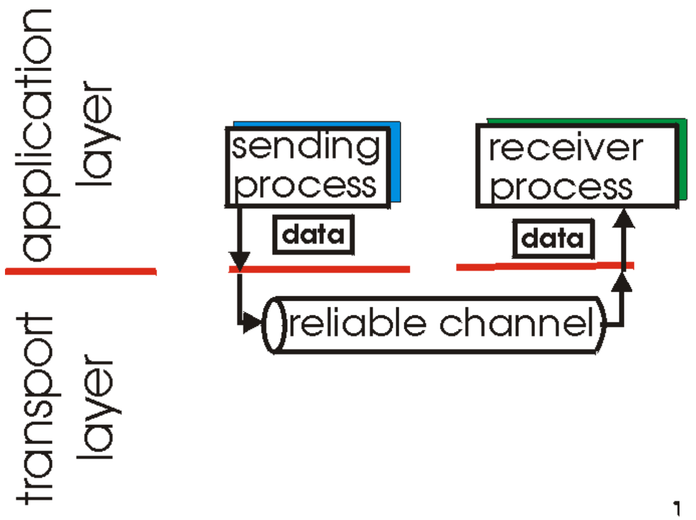
- leaving the network “edge” (application, transport layers)
- into the network “core”

3. Gaia:

- 3.1 Garraio geruzaren zerbitzuak
- 3.2 Multiplexing and demultiplexing
- 3.3 Konexiorik gabeko garraioa: UDP
- 3.4 Informazio garraio fidagarriaren oinarriak
- 3.5 Konexiorako bideratutako transportea: TCP
  - Segmentuen estruktura
  - Informazio transferentzia fidagarria
  - Fluxuaren kontrola
  - Konexioaren kudeaketa
- 3.6 Pilaketen kontrolaren oinarriak
- 3.7 TCP-ren pilaketen kontrola

Informazio garraio fidagarriaren oinarriak

- Garrantzitsua aplikazioetan, garraioan, lotura geruzetan
- top-10 list of important networking topics!

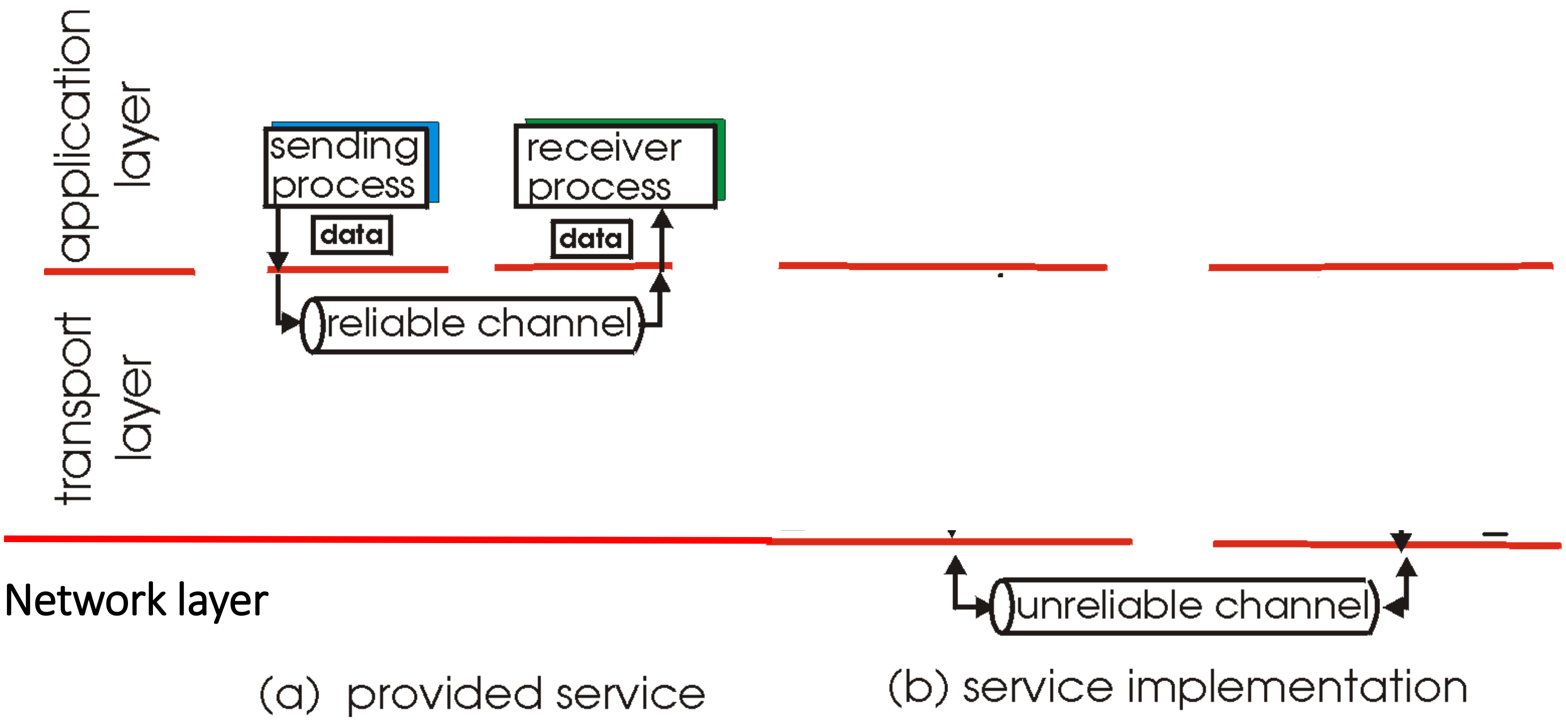


(a) provided service

- Kanal ez fidagarriaren ezaugarriek, informazio fidagarriaren garraio protokoloaren (rdt) konplexutasuna definitzen dute

Informazio garraio fidagarriaren oinarriak

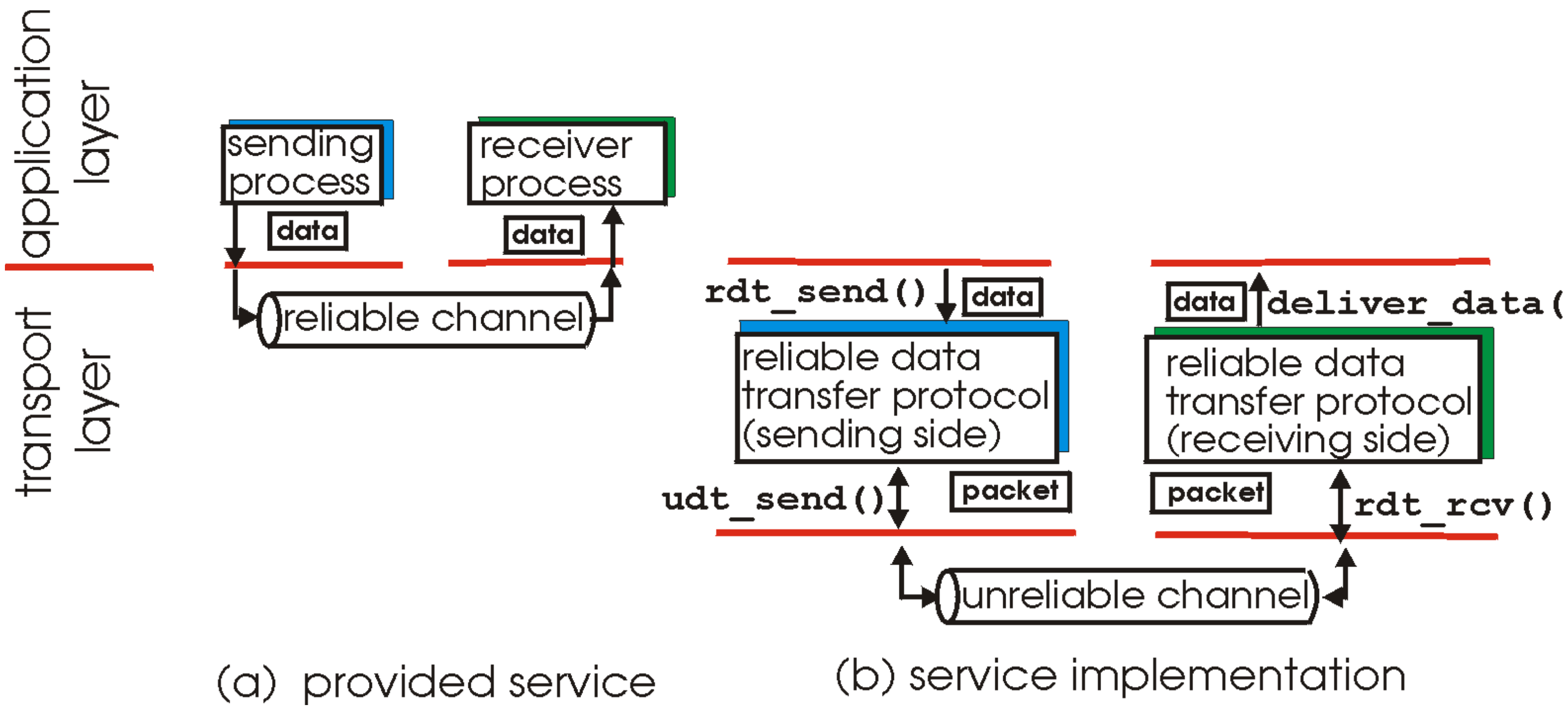
- Garrantzitsua aplikazioetan, garraioan, lotura geruzetan
- top-10 list of important networking topics!



- Kanal ez fidagarriaren ezaugarriek, informazio fidagarriaren garraio protokoloaren (rdt) konplexutasuna definitzen dute

Informazio garraio fidagarriaren oinarriak

- Garrantzitsua aplikazioetan, garraioan, lotura geruzetan
- top-10 list of important networking topics!

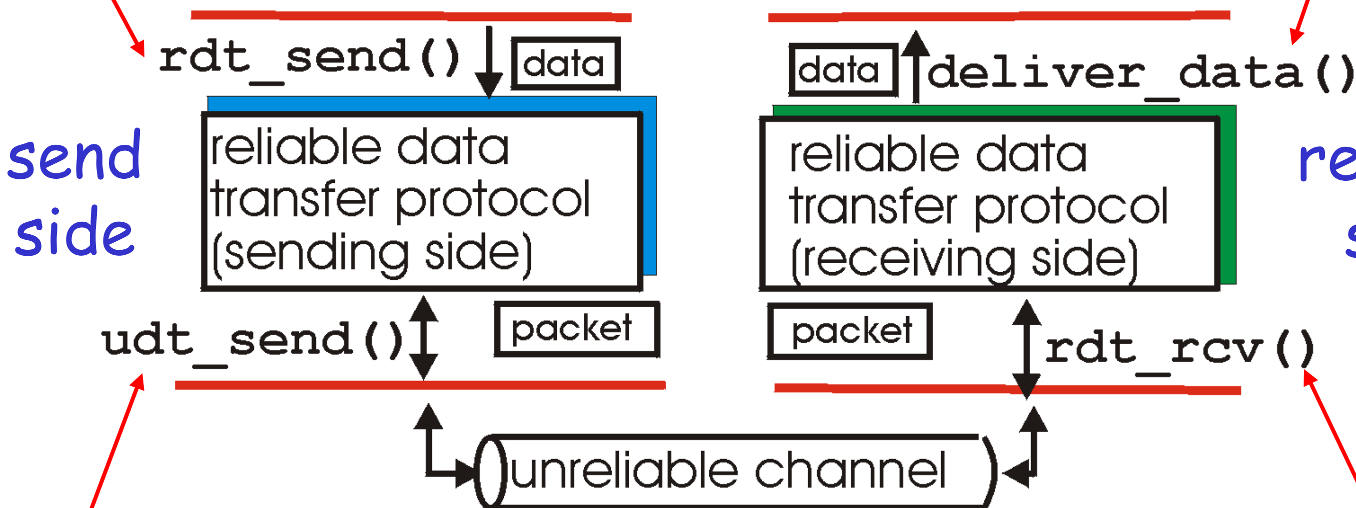


- Kanal ez fidagarriaren ezaugarriek, informazio fidagarriaren garraio protokoloaren (rdt) konplexutasuna definitzen dute

Informazio garraio fidagarria: hasiera

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by **rdt** to deliver data to upper



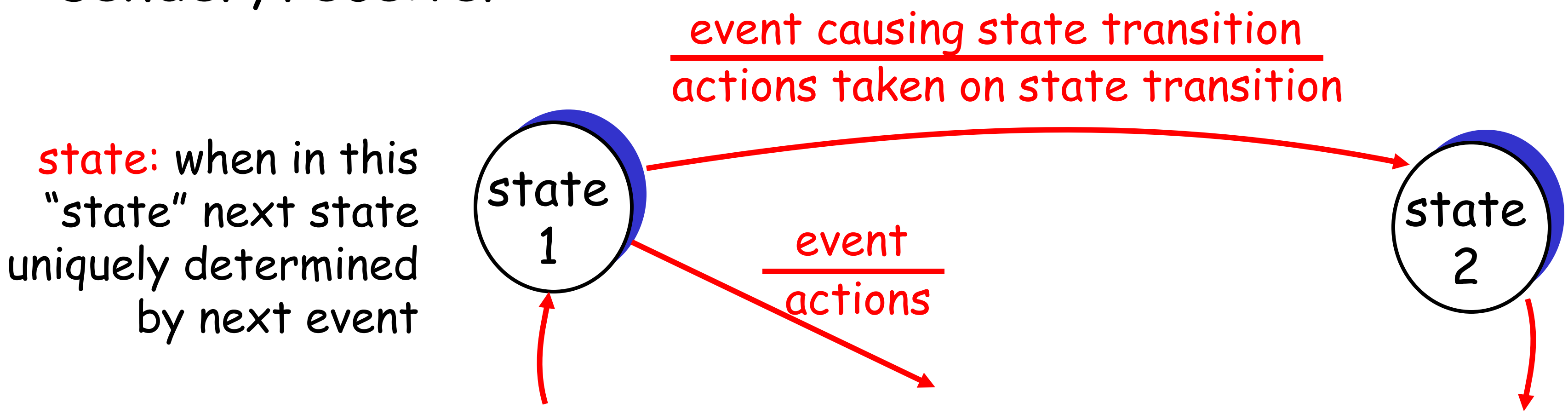
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

rdt_rcv(): called when packet arrives on rcv-side of channel

Informazio garraio fidagarria: hasiera

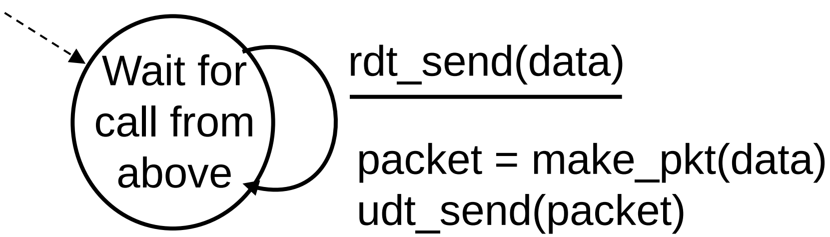
We'll:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

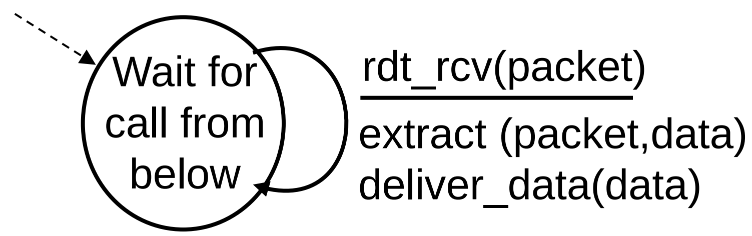


Rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel



sender

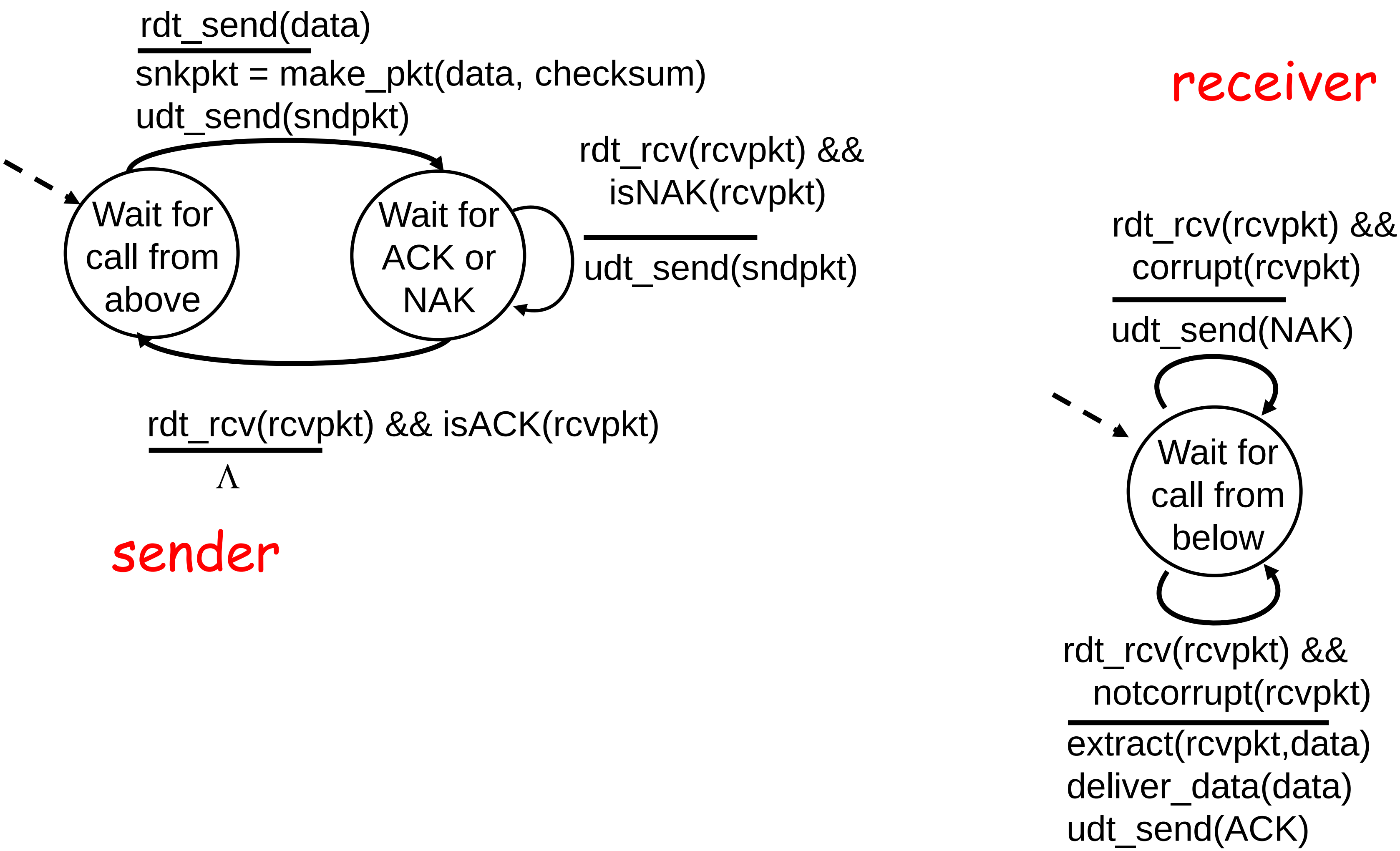


receiver

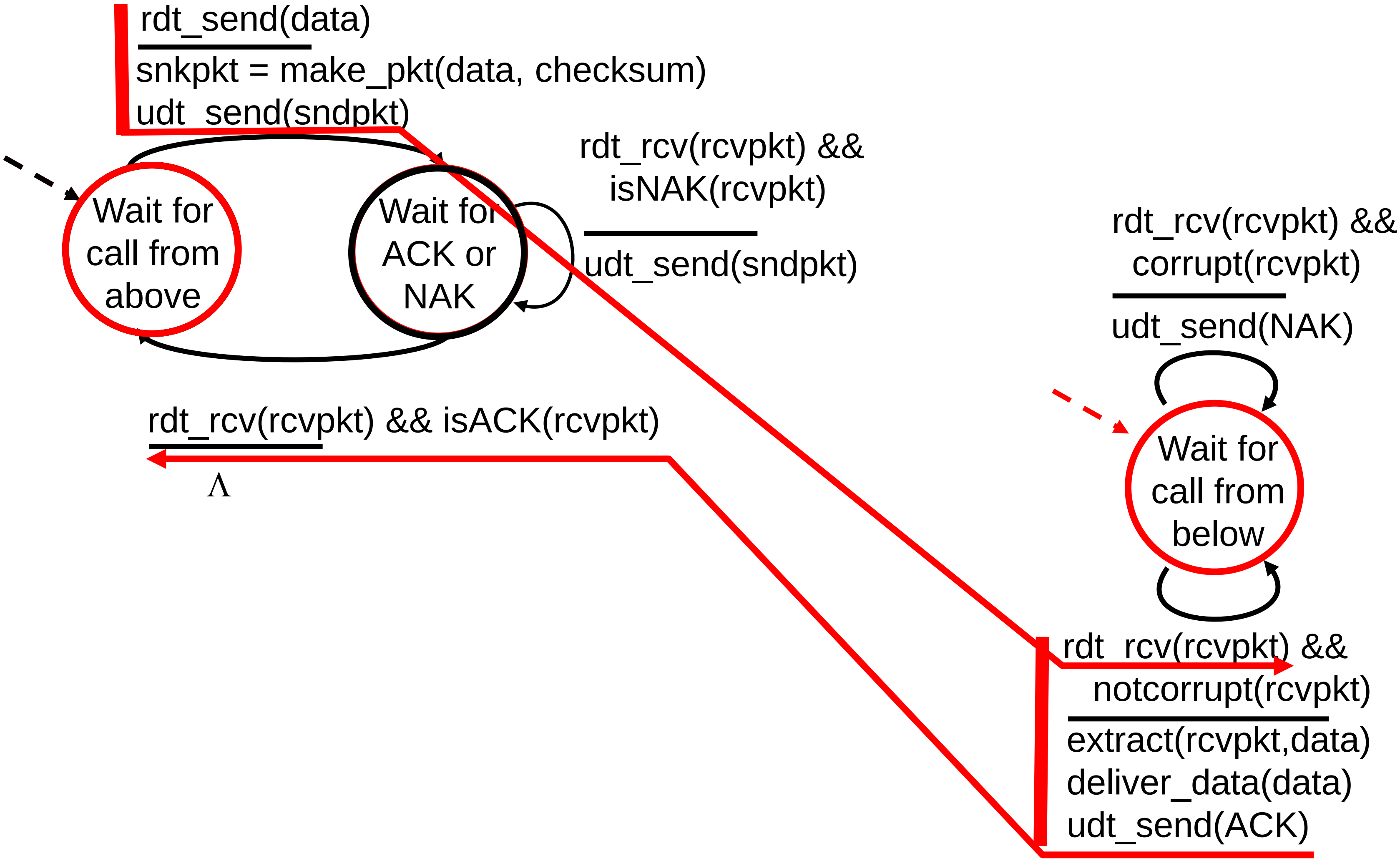
Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- new mechanisms in **`rdt2.0`** (beyond **`rdt1.0`**):
  - error detection
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender

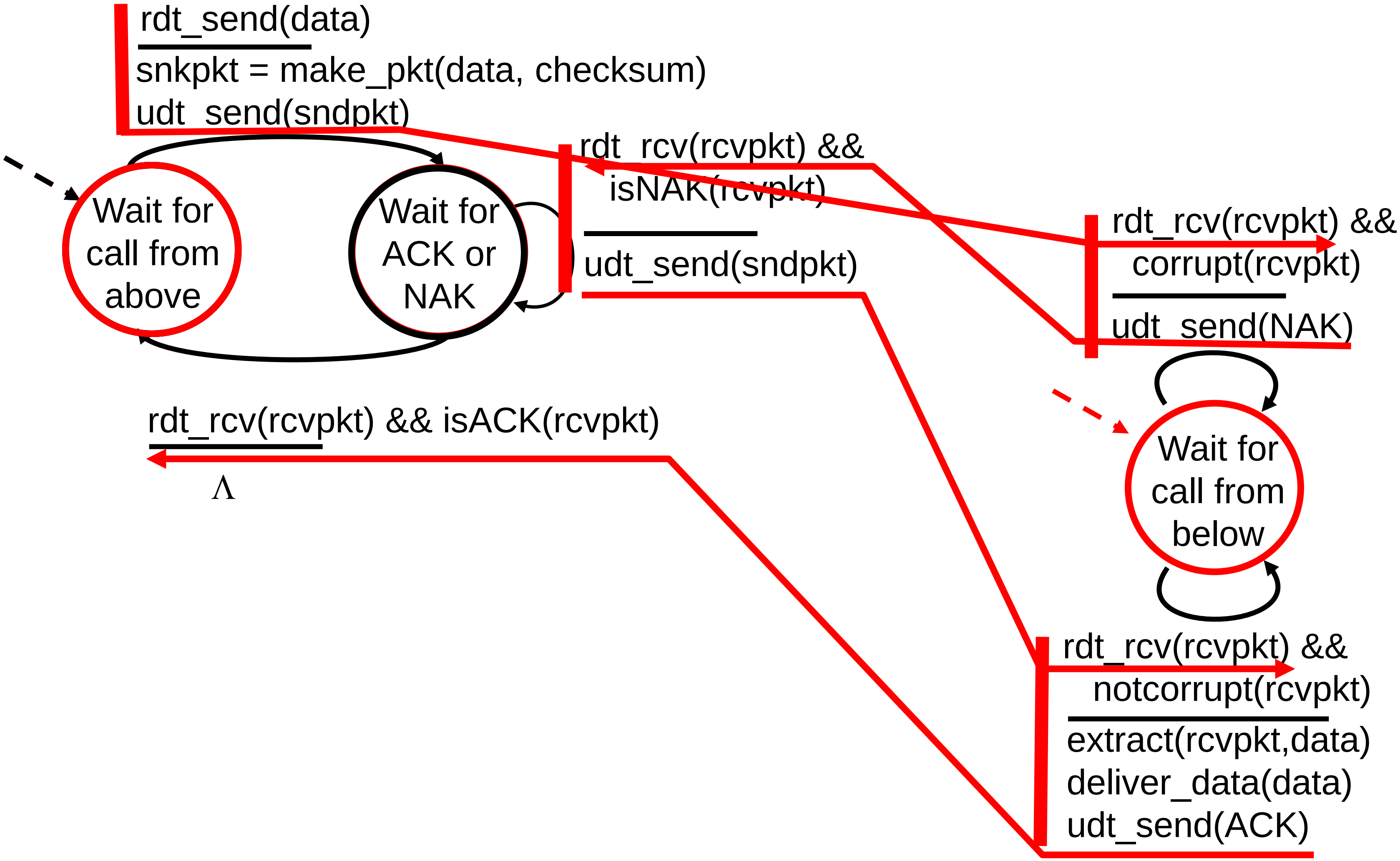
rdt2.0: FSM specification



rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

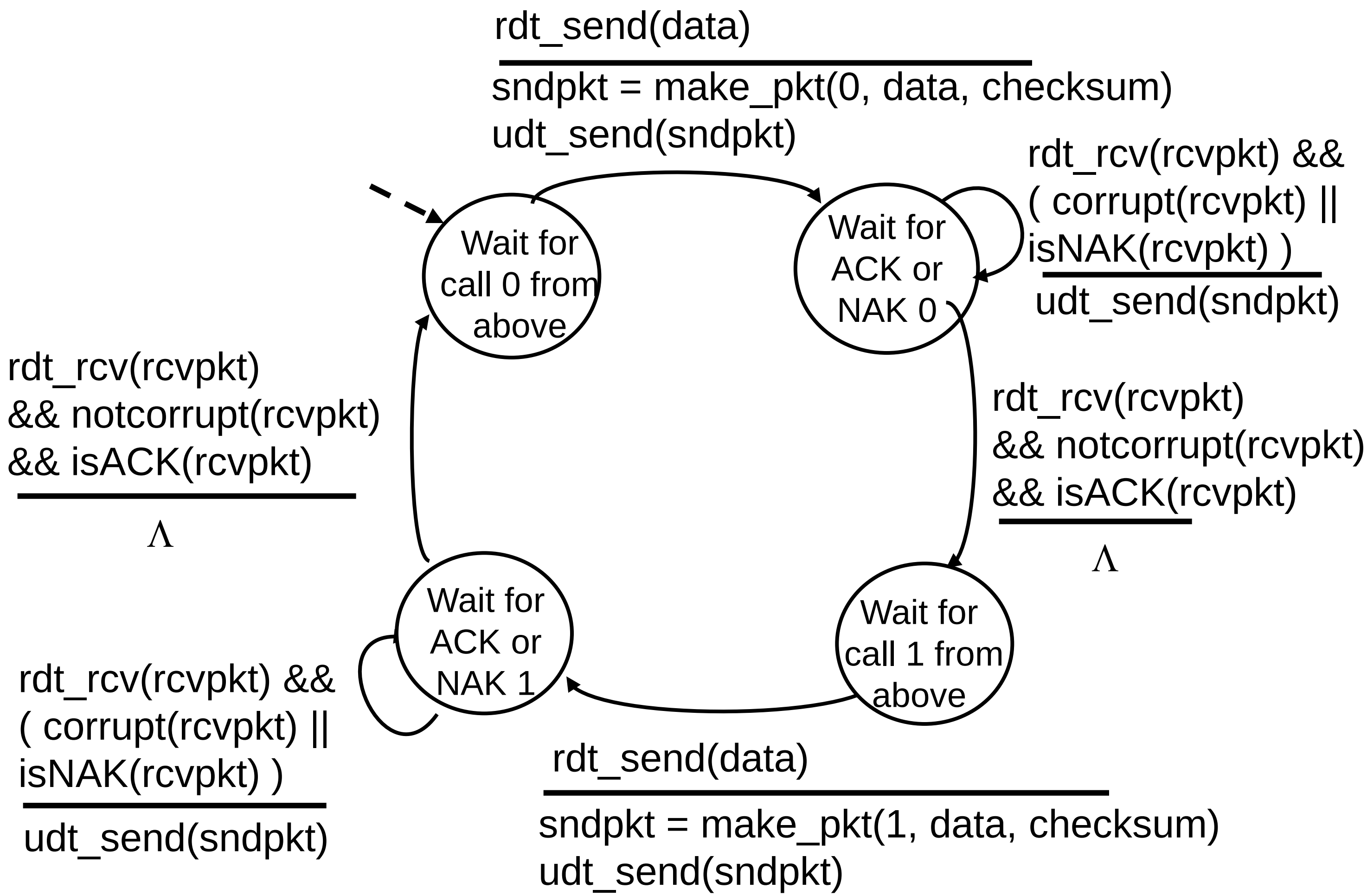
Handling duplicates:

- sender retransmits current pkt if ACK/NAK garbled
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

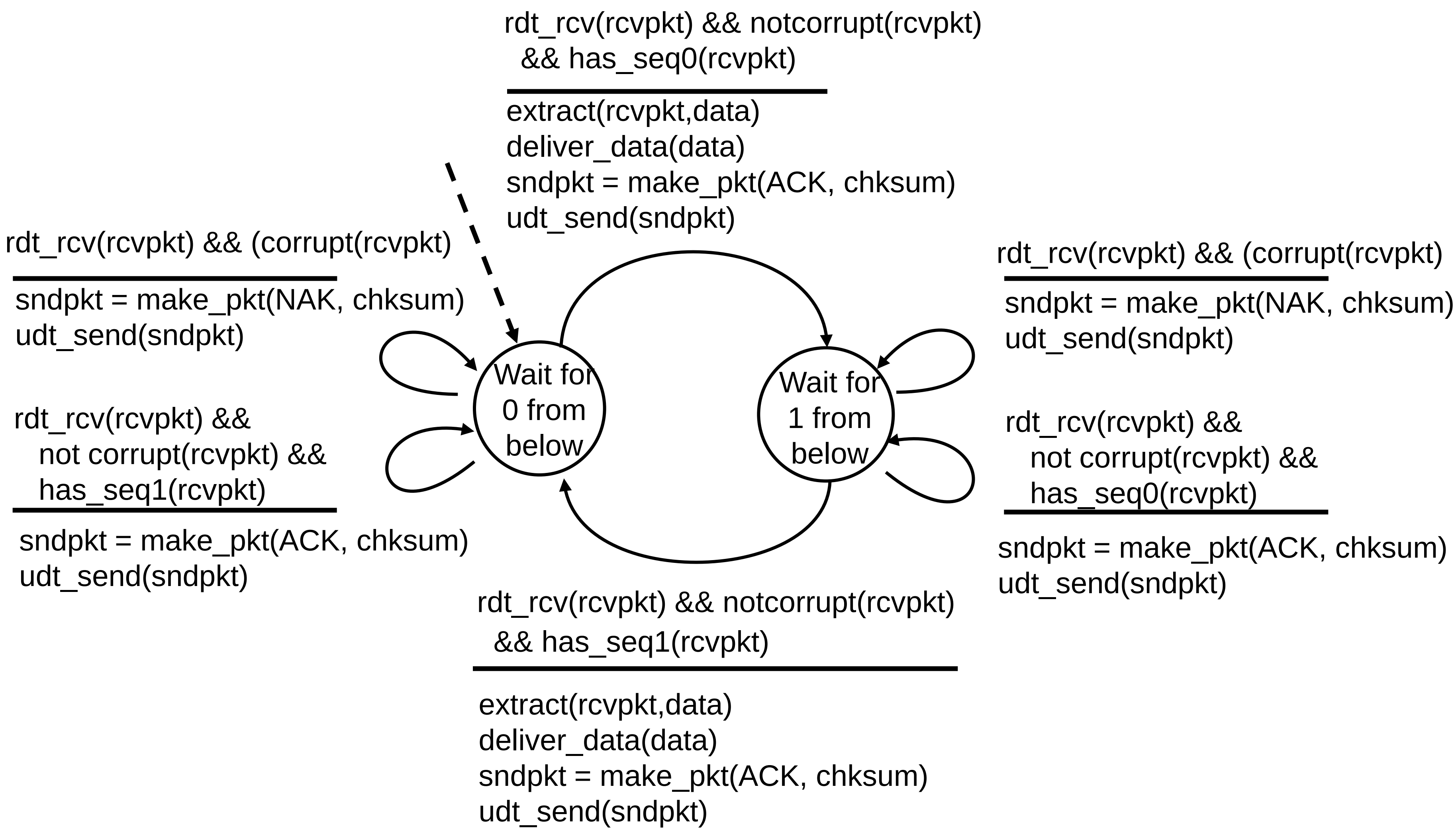
stop and wait

Sender sends one packet, then waits for receiver response

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



rdt2.1: discussion

Sender:

- seq # added to pkt
- two seq. #'s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "current" pkt has 0 or 1 seq. #

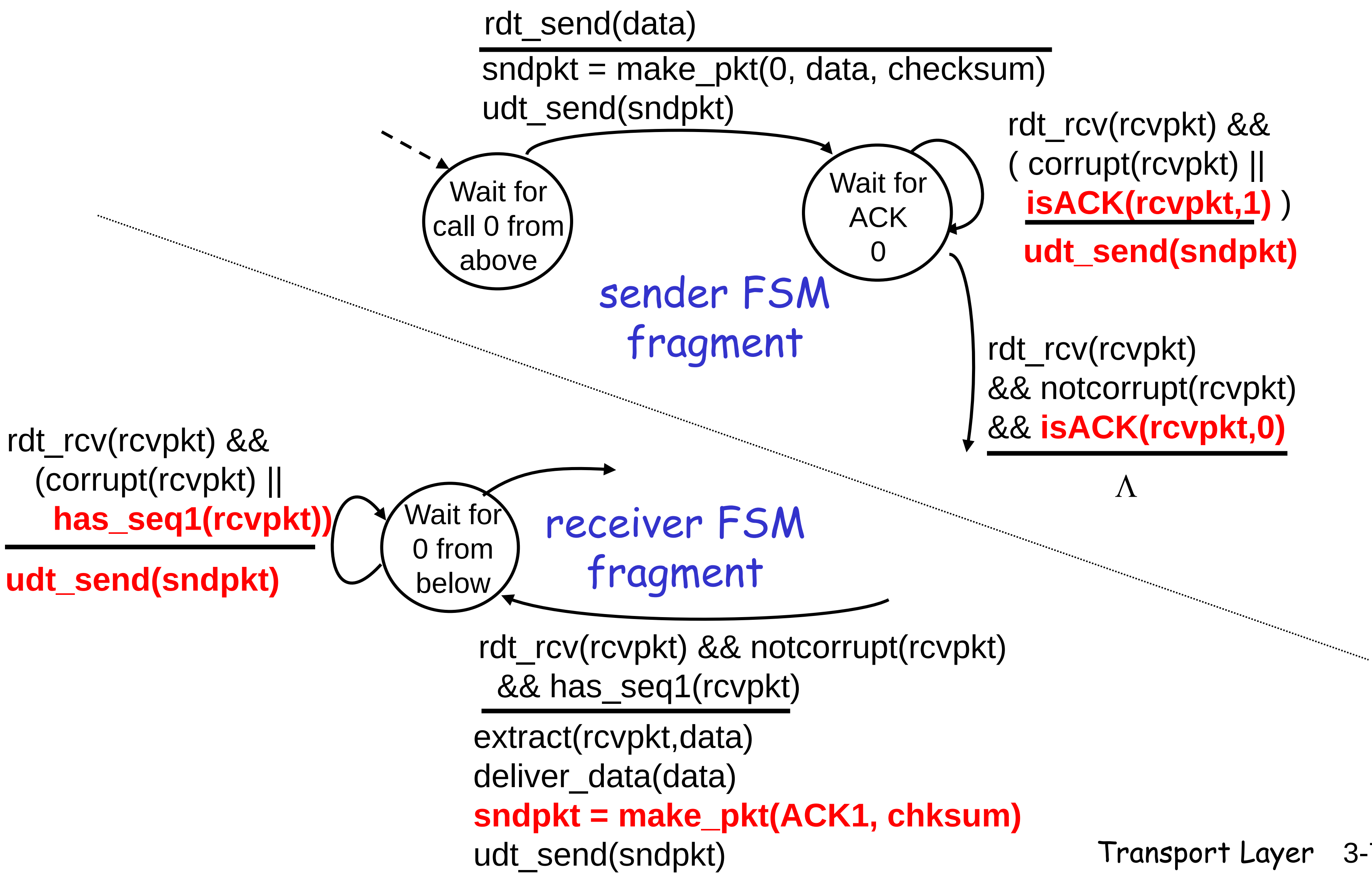
Receiver:

- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

rdt2.2: sender, receiver fragments



rdt3.0: channels with errors *and* loss

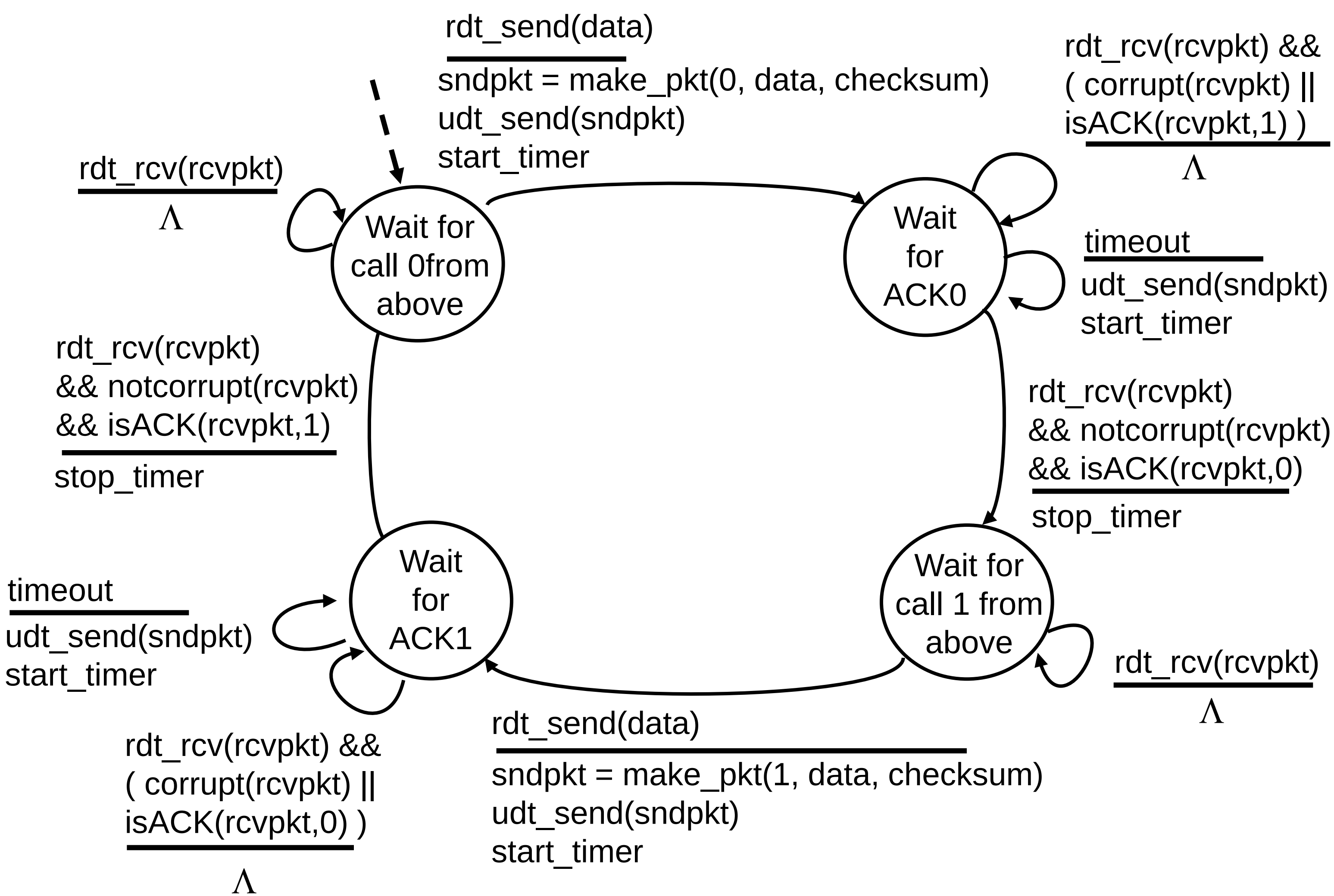
New assumption: underlying channel can also lose packets (data or ACKs)

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

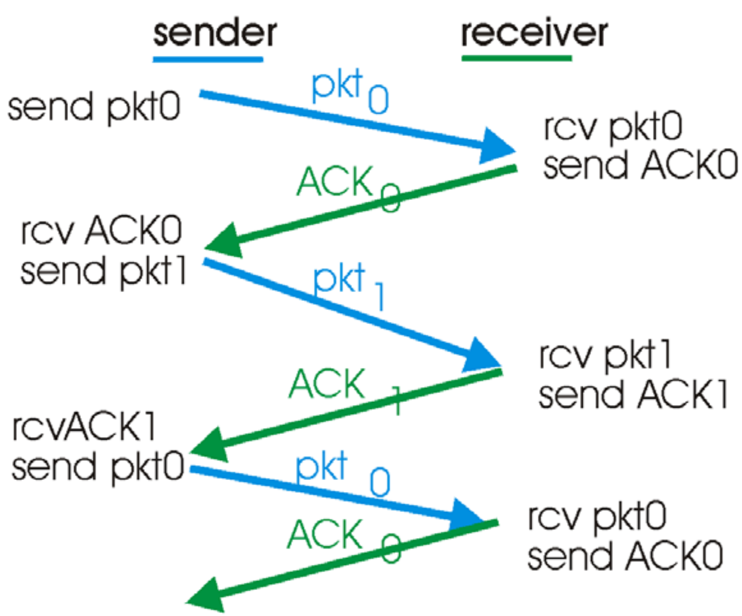
Approach: sender waits "reasonable" amount of time for ACK

- 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

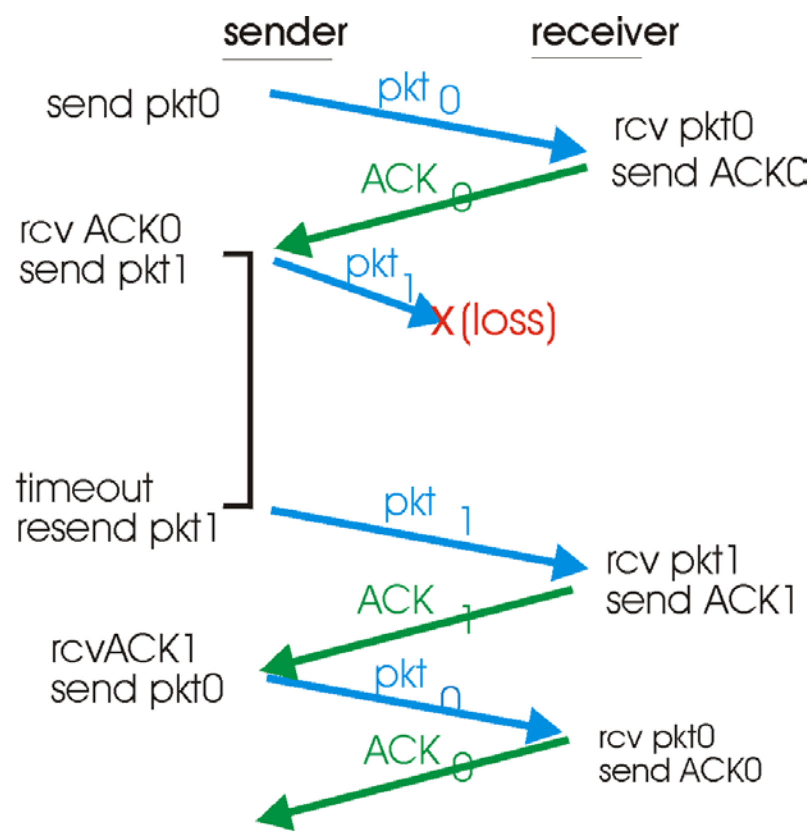
rdt3.0 sender



rdt3.0 in action

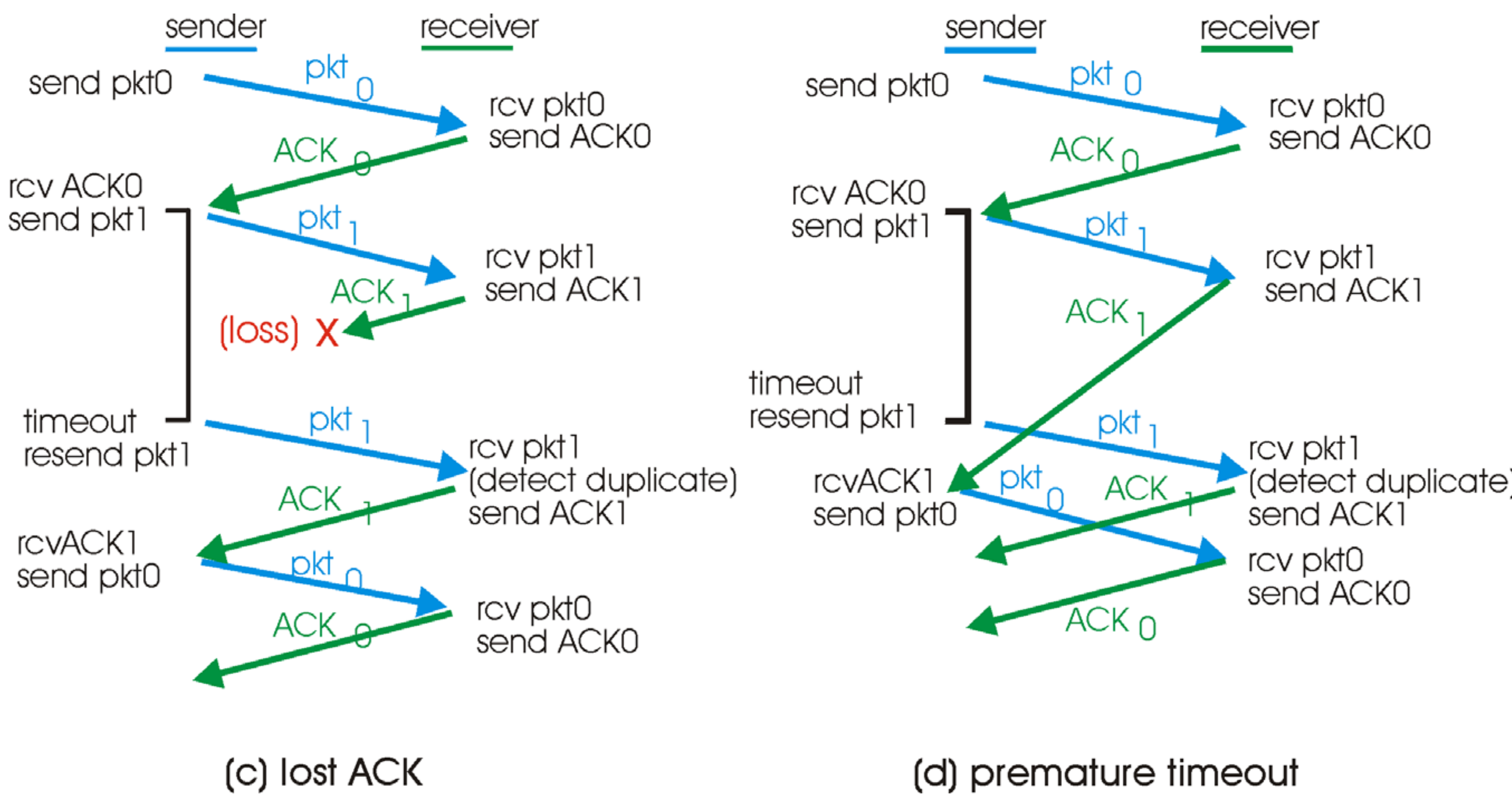


(a) operation with no loss



(b) lost packet

rdt3.0 in action



Performance of rdt3.0

- rdt3.0 works, but performance stinks
- ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

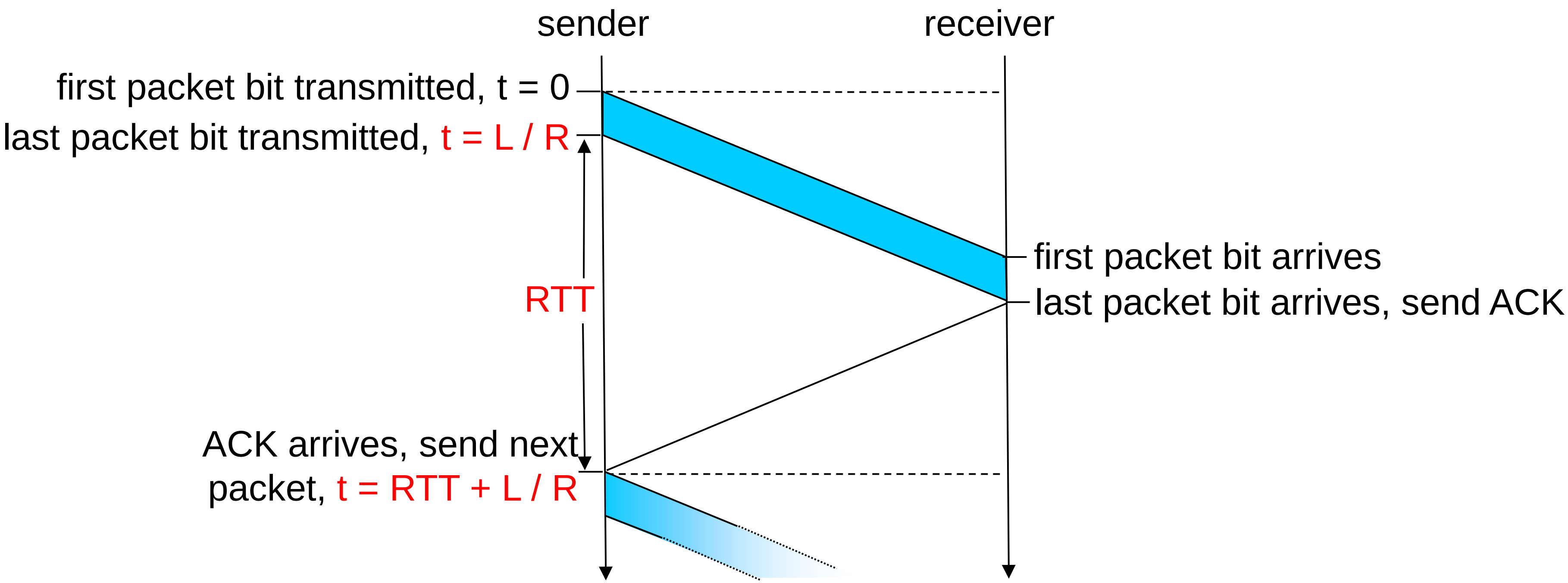
$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\,\text{bps}} = 8\,\text{microseconds}$$

  - U_{sender}: utilization – fraction of time sender busy sending

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

  - 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
  - network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation

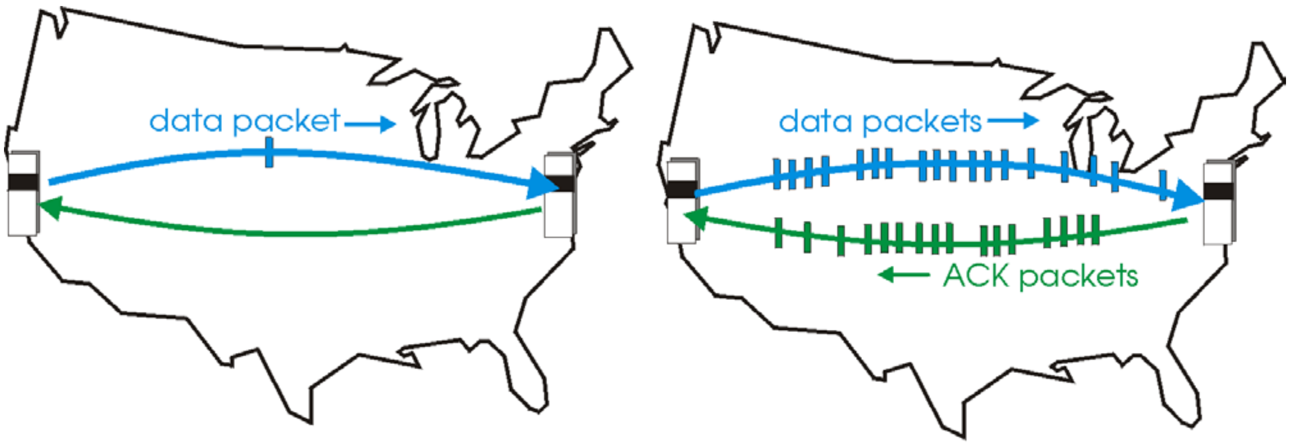


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Pipelined protocols

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

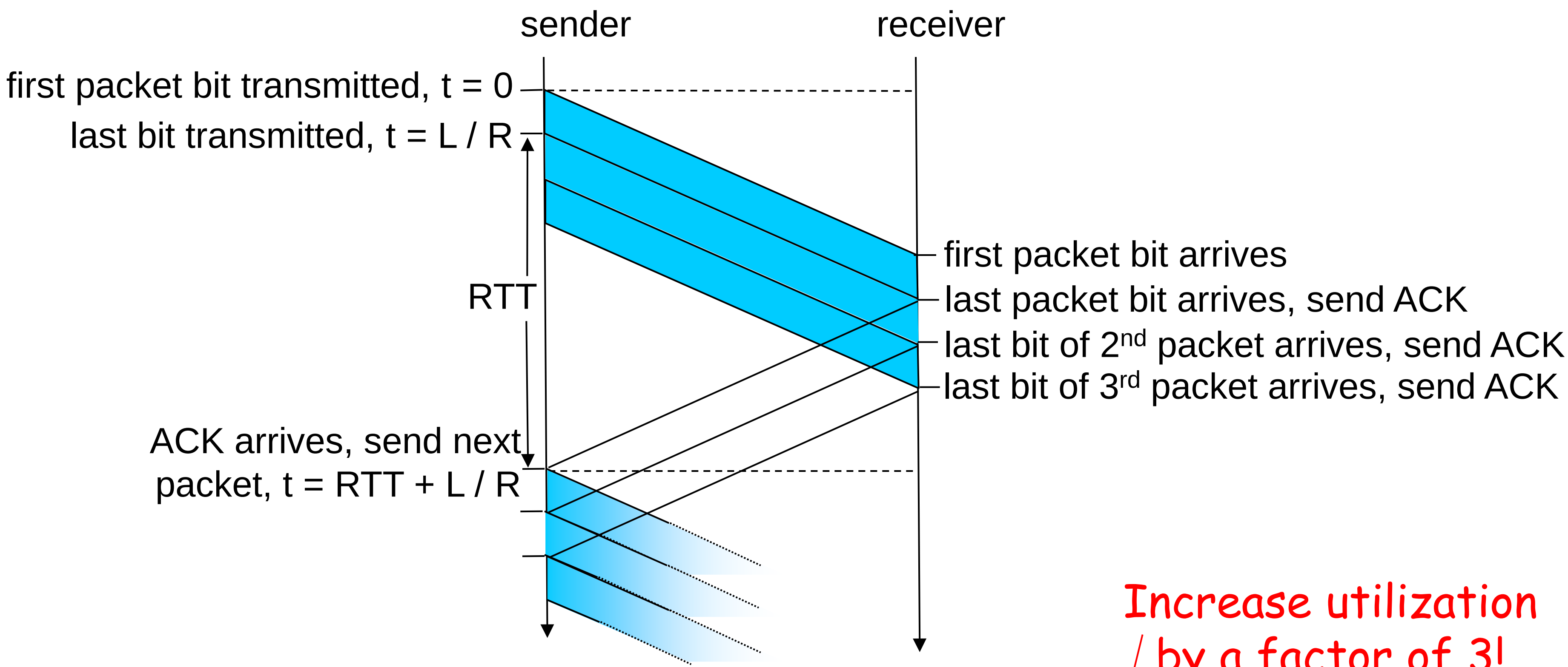
- range of sequence numbers must be increased
- buffering at sender and/or receiver



(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!

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

Pipelining Protocols

Go-back-N: big picture:

- Sender can have up to N unacked packets in pipeline
- Rcvr only sends cumulative acks
  - Doesn't ack packet if there's a gap
- Sender has timer for oldest unacked packet
  - If timer expires, retransmit all unacked packets

Selective Repeat: big pic

- Sender can have up to N unacked packets in pipeline
- Rcvr acks individual packets
- Sender maintains timer for each unacked packet
  - When timer expires, retransmit only unack packet

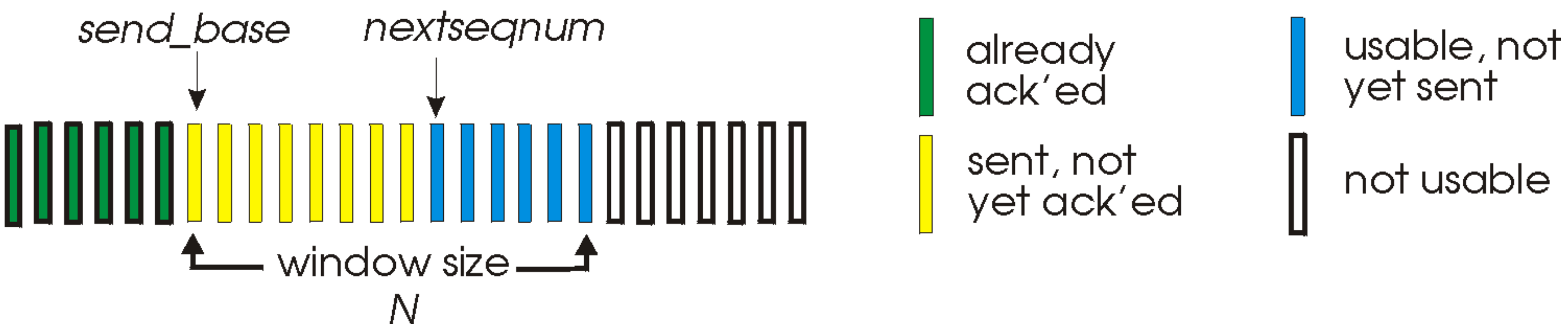
Selective repeat: big picture

- Sender can have up to N unacked packets in pipeline
- Rcvr acks individual packets
- Sender maintains timer for each unacked packet
  - When timer expires, retransmit only unack packet

Go-Back-N

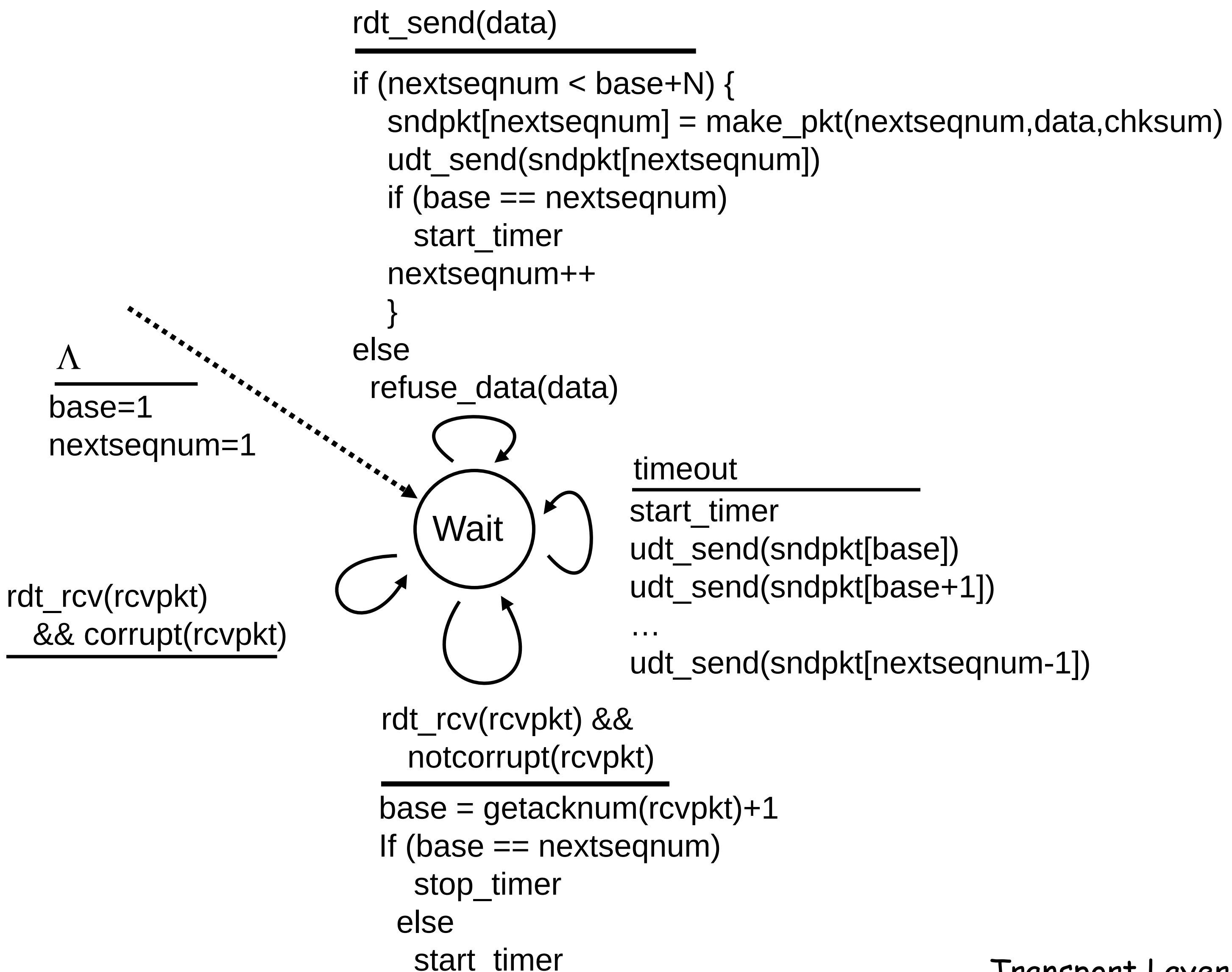
Sender:

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

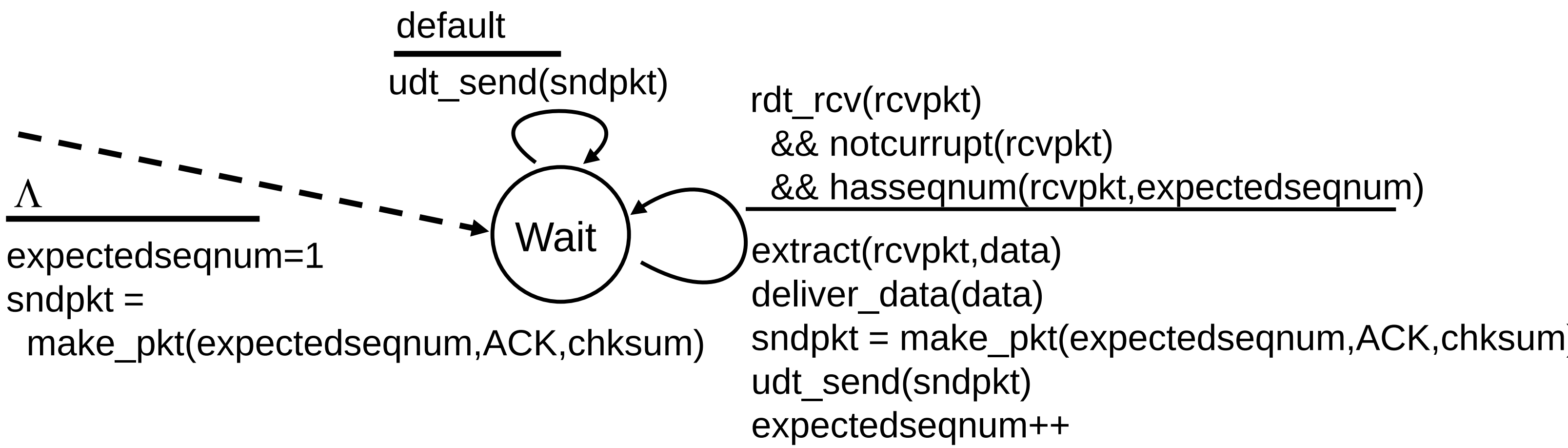


- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- *timeout(n):* retransmit pkt n and all higher seq # pkts in window

GBN: sender extended FSM



GBN: receiver extended FSM

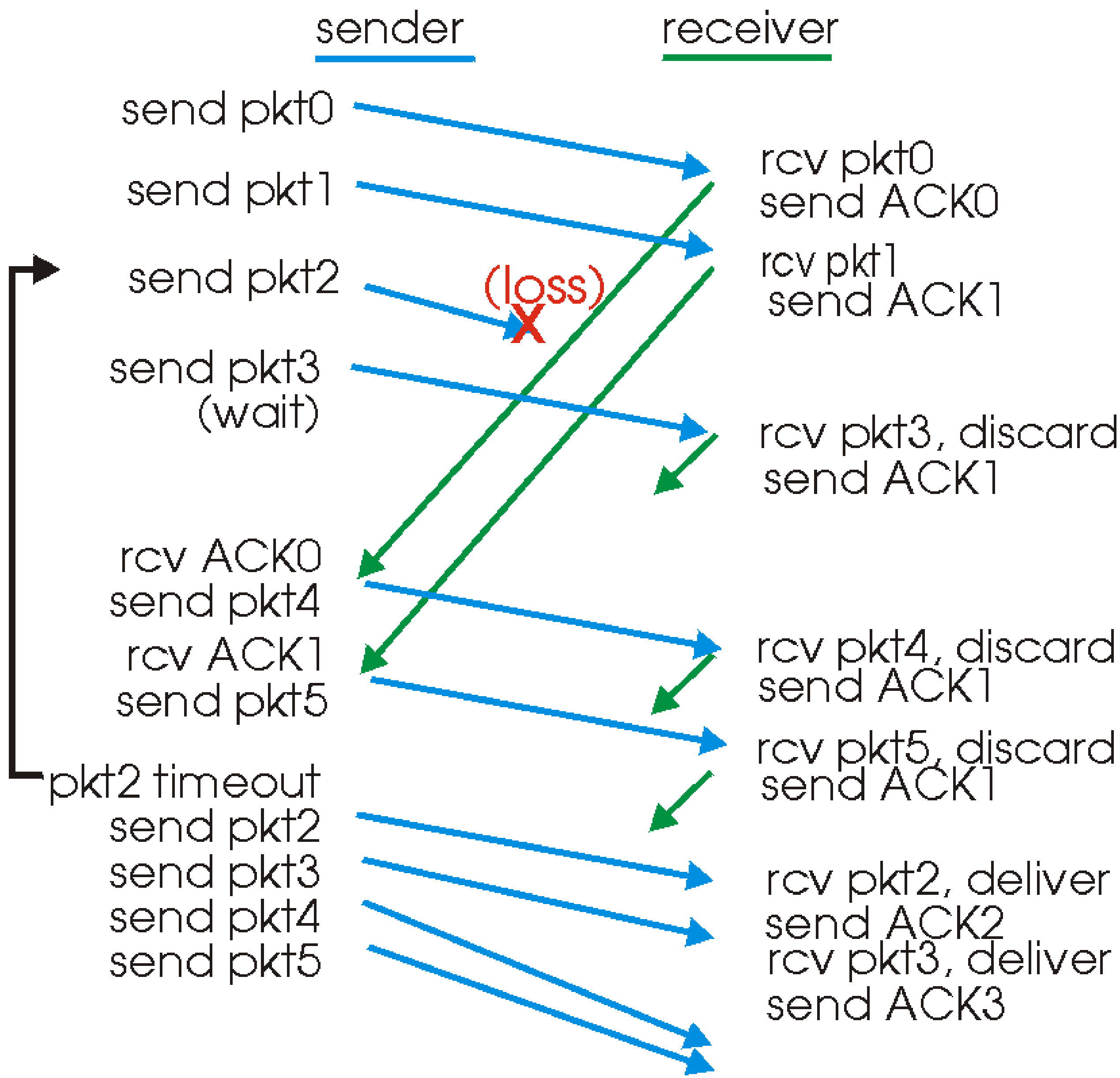


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- 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 #

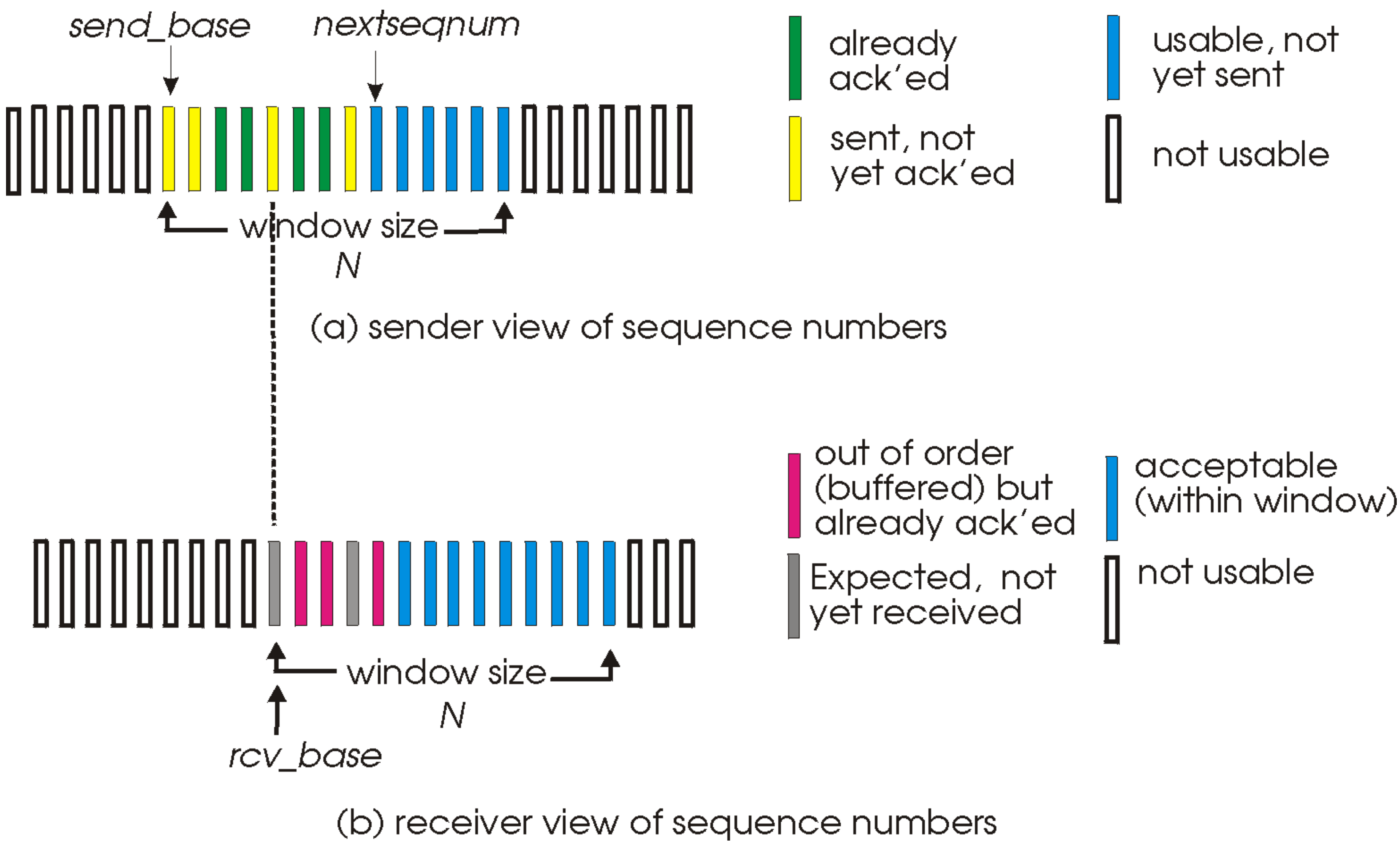
GBN in action



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

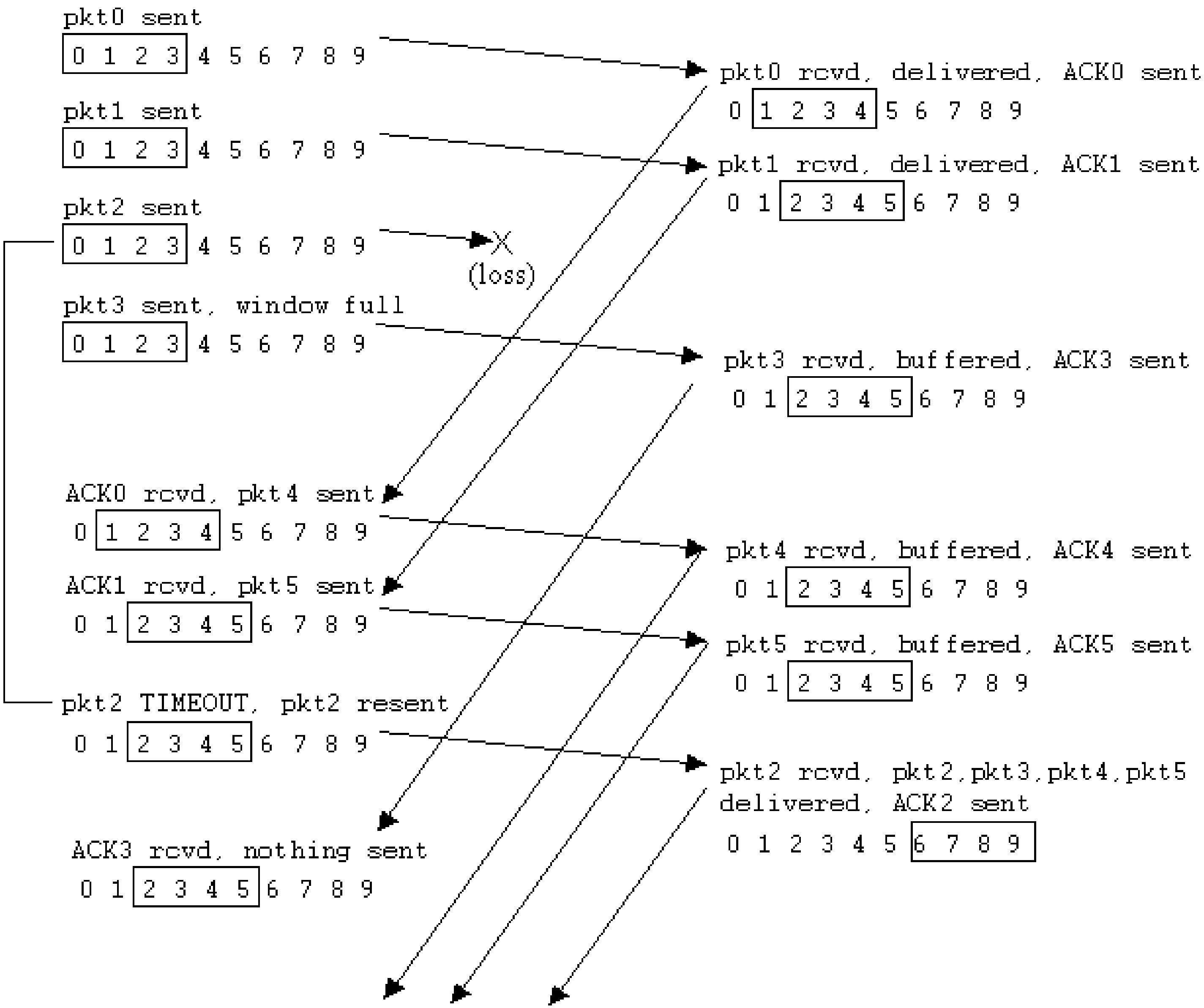
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

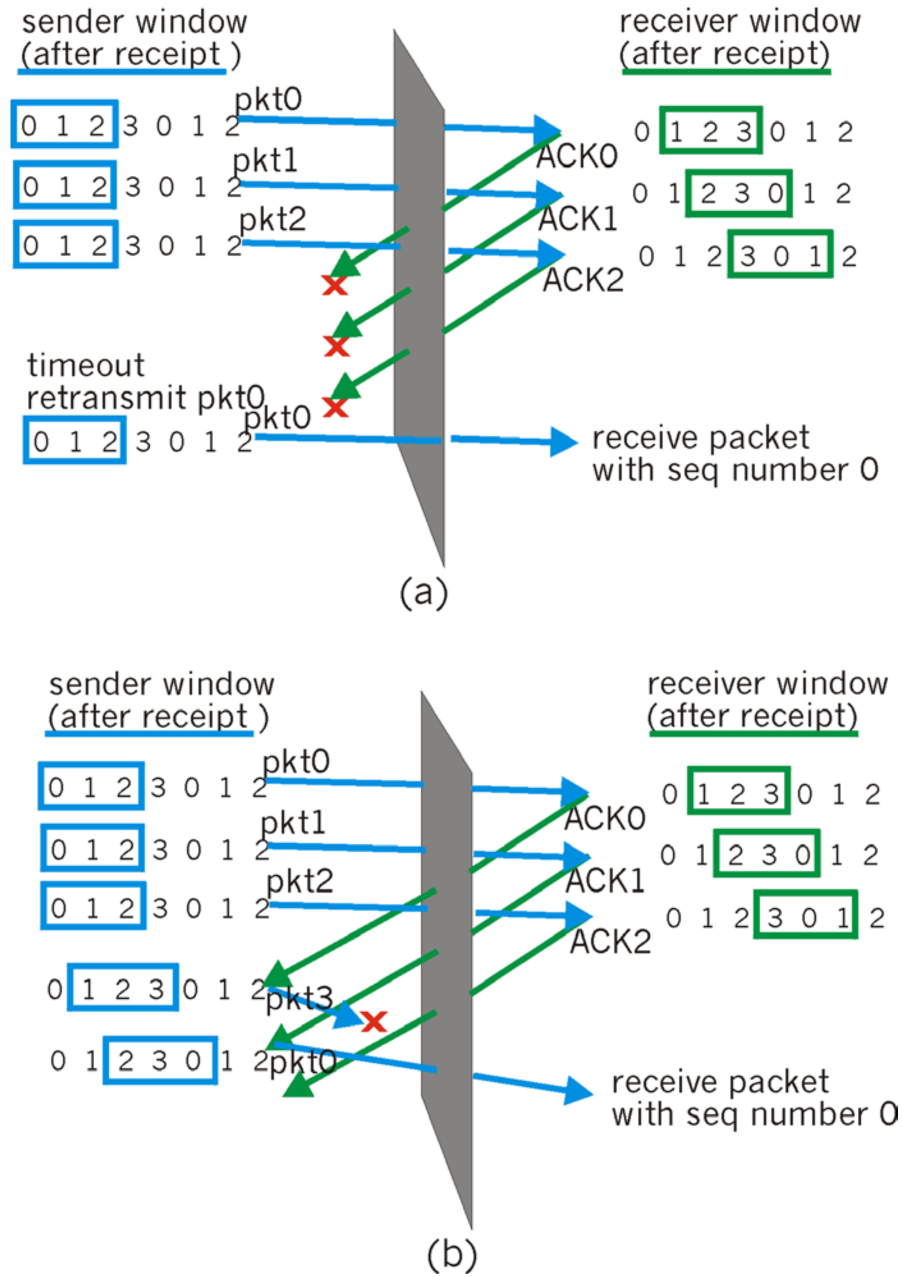
Selective repeat in action



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

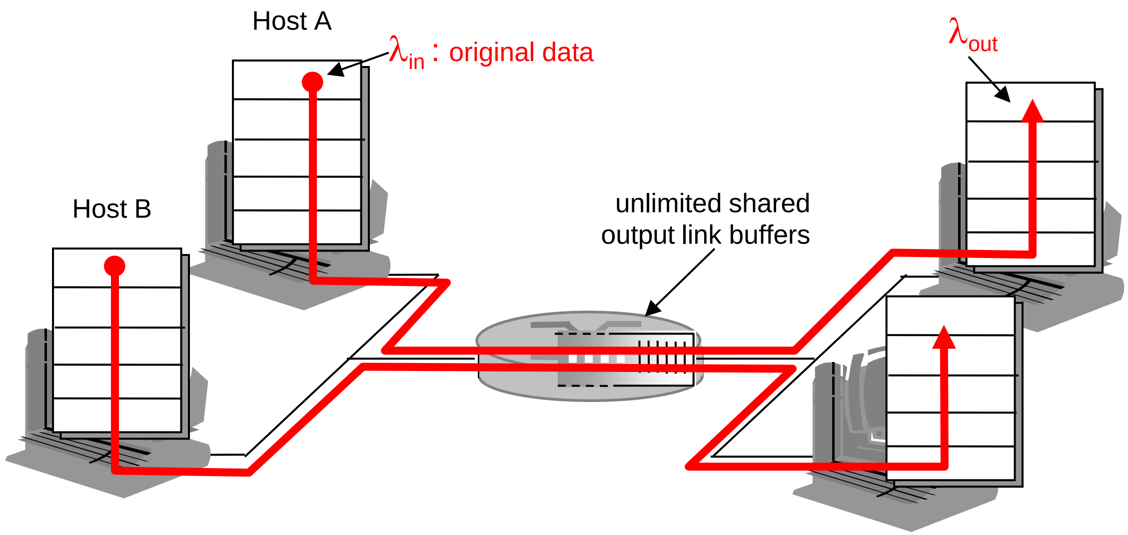
Principles of Congestion Control

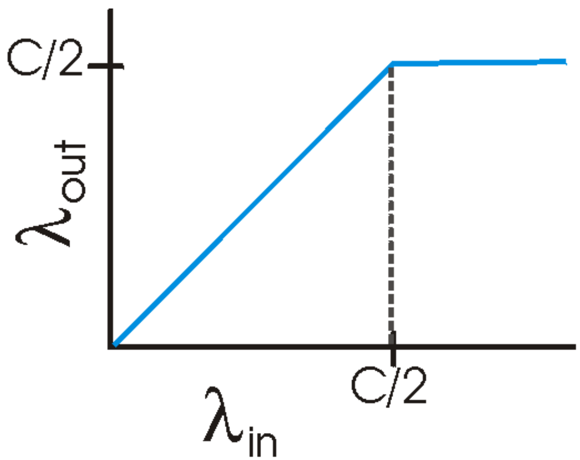
Congestion:

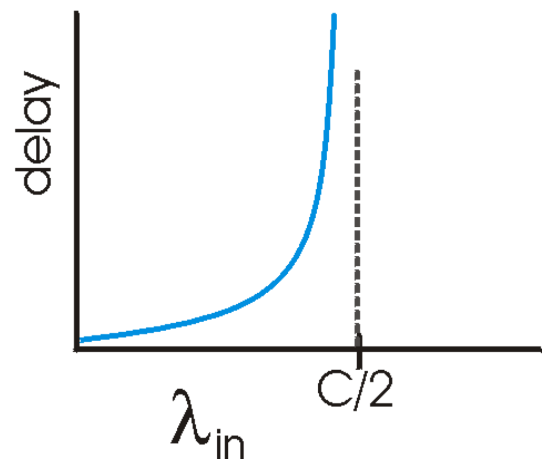
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

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



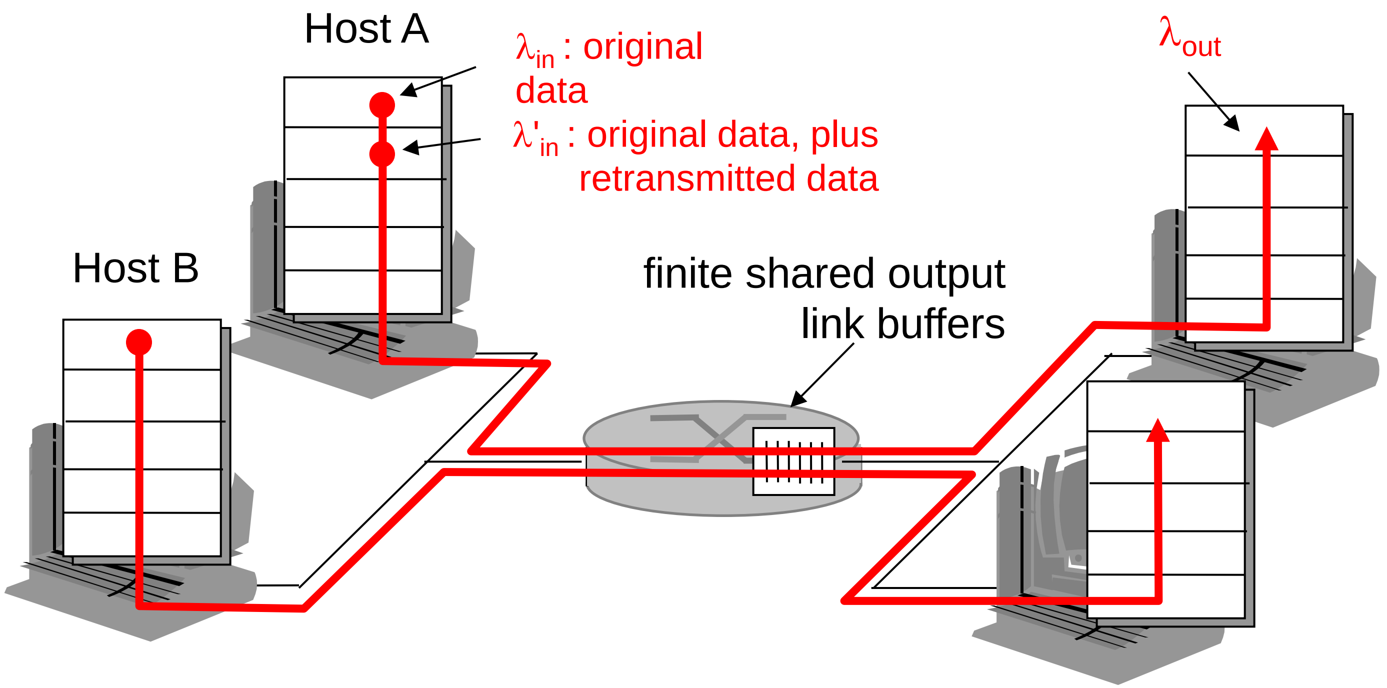




- large delays when congested
- maximum achievable throughput

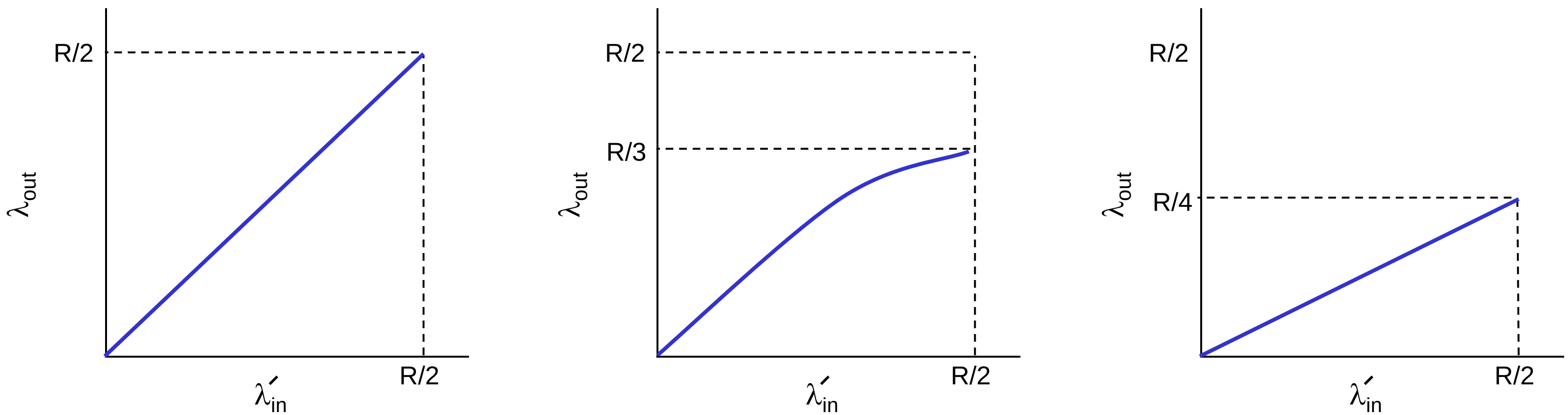
Causes/costs of congestion: scenario 2

- 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.

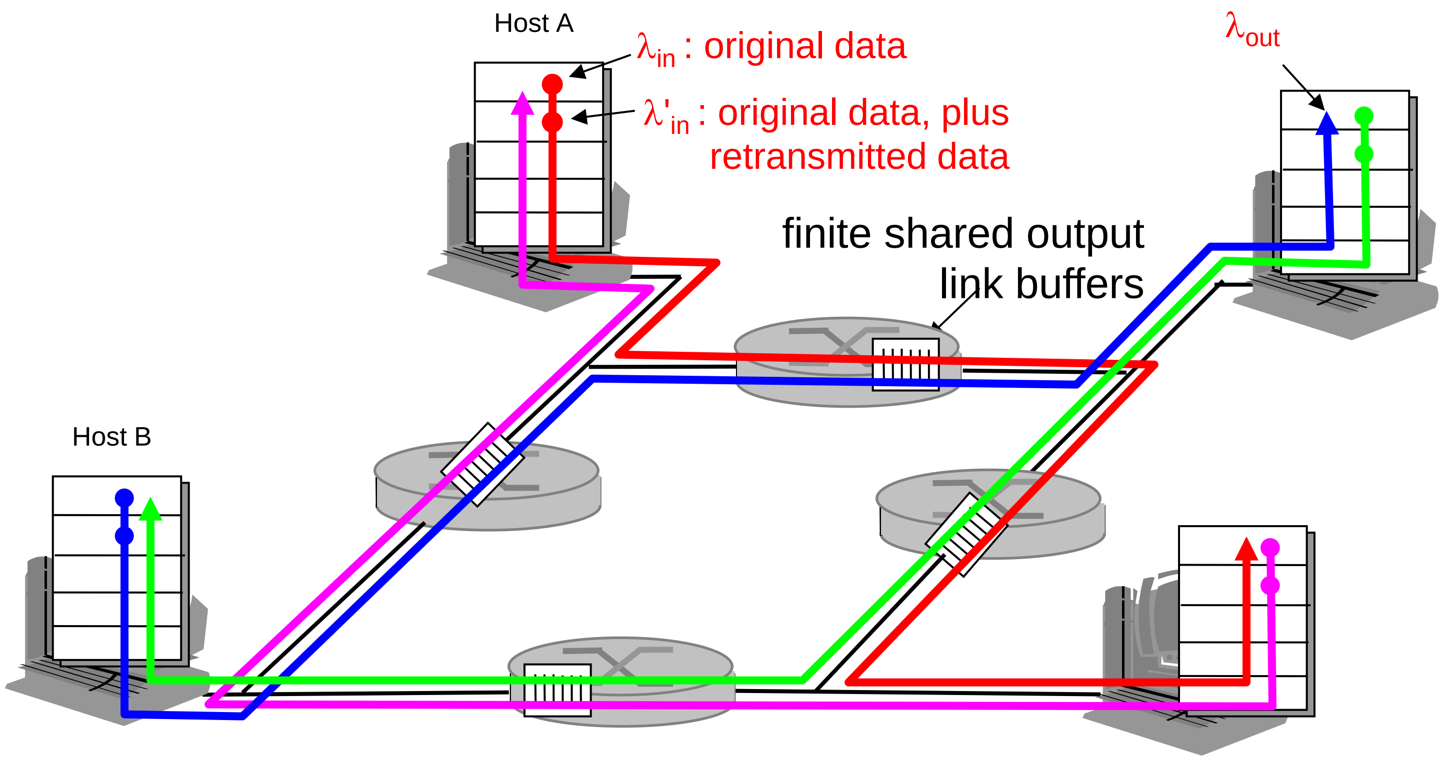
"costs" of congestion:

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

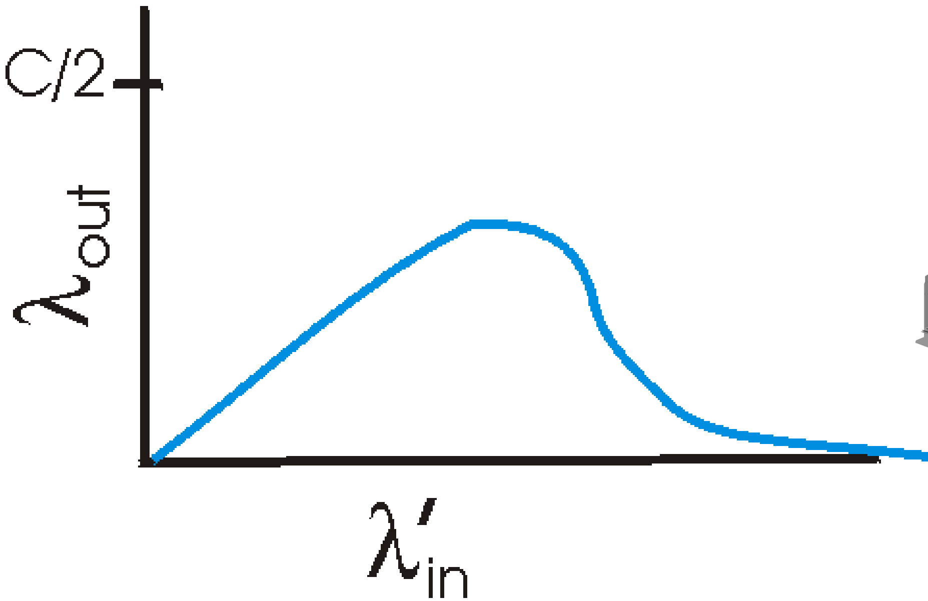
Causes/costs of congestion: scenario 3

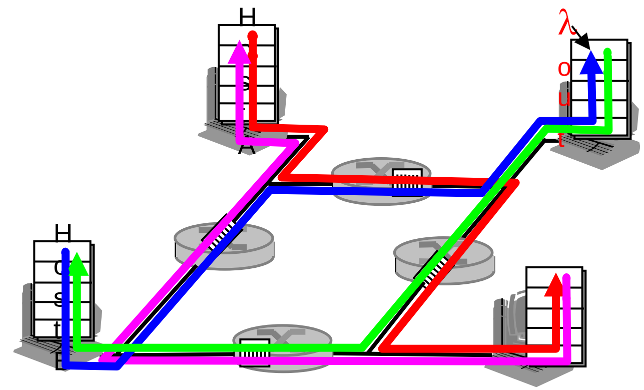
- four senders
- multihop paths
- timeout/retransmit

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

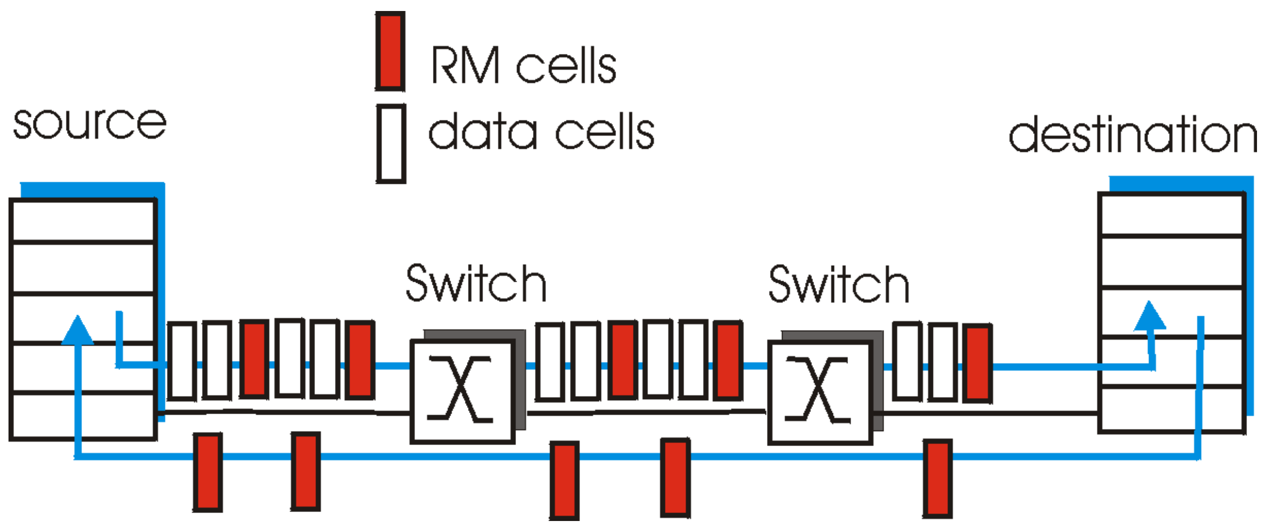
ABR: available bit rate:

- "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 maximum 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

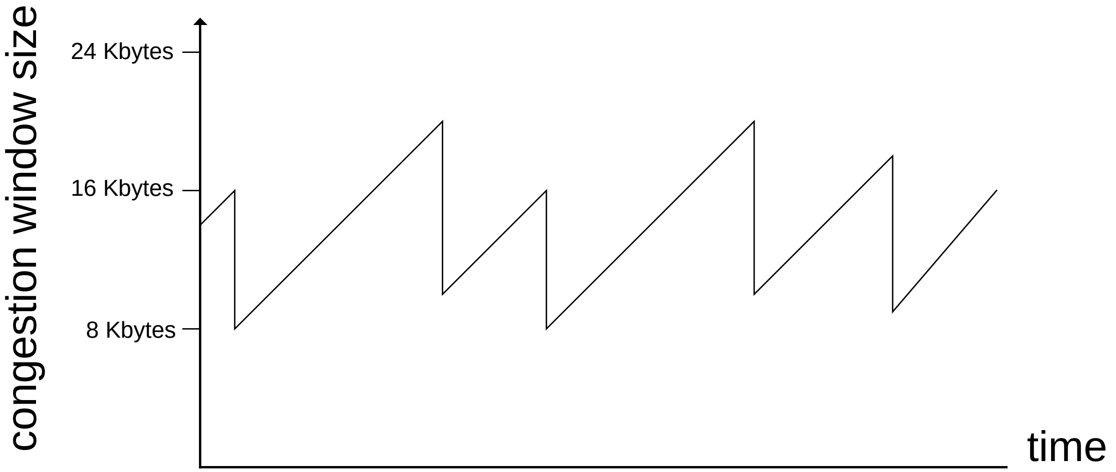
Chapter 3 outline

- 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

TCP congestion control: additive increase, multiplicative decrease

- *Approach:* increase transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **CongWin** by 1 MSS every RTT until loss detected
  - *multiplicative decrease*: cut **CongWin** in half after loss

Saw tooth behavior: probing for bandwidth



TCP Congestion Control: details

- 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:

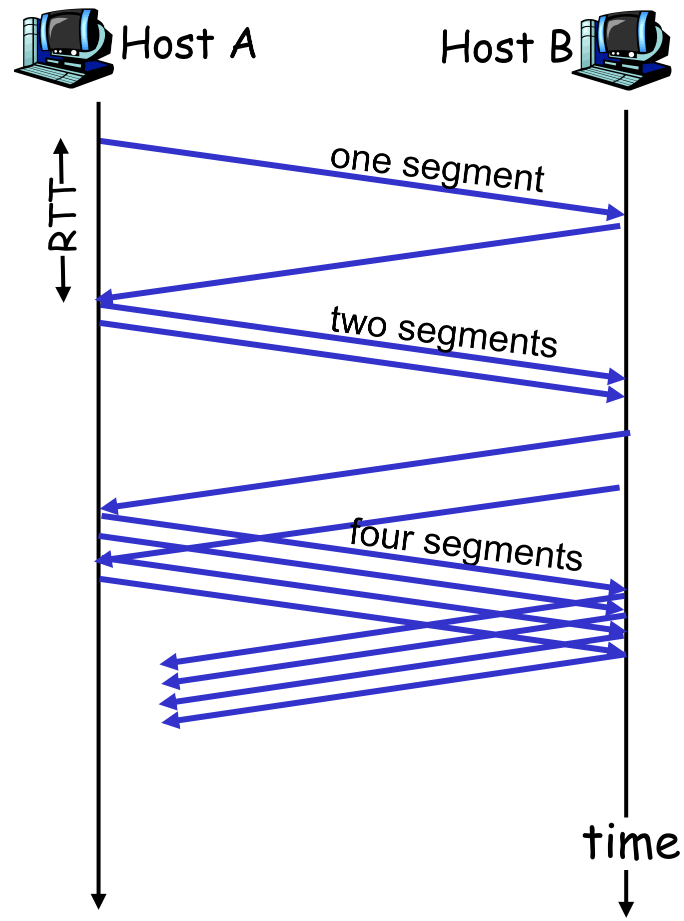
  - AIMD
  - slow start
  - conservative after timeout events

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

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



Refinement: inferring loss

- 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

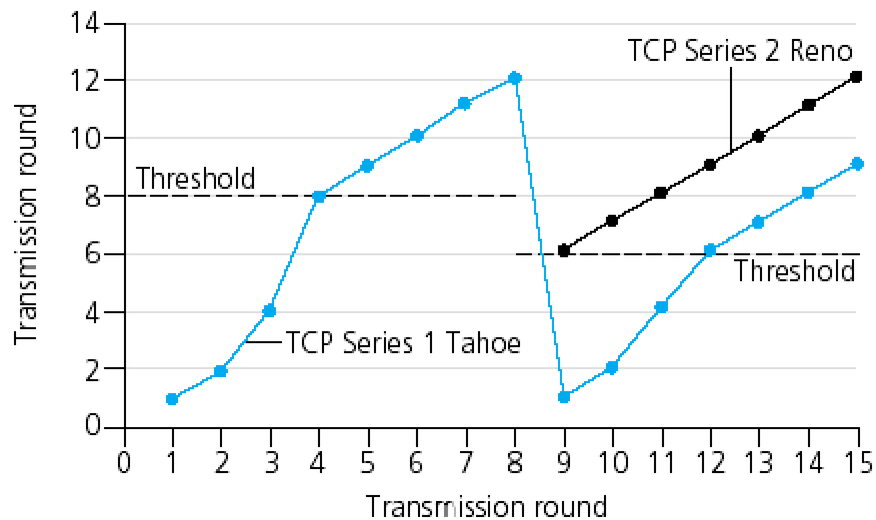
Philosophy:

- 3 dup ACKs indicates network capable of delivering some segments
- timeout indicates a "more alarming" congestion scenario

Refinement

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

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.

TCP sender congestion control

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

TCP throughput

- What's the average throughout of 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: TCP over "long, fat pipes"

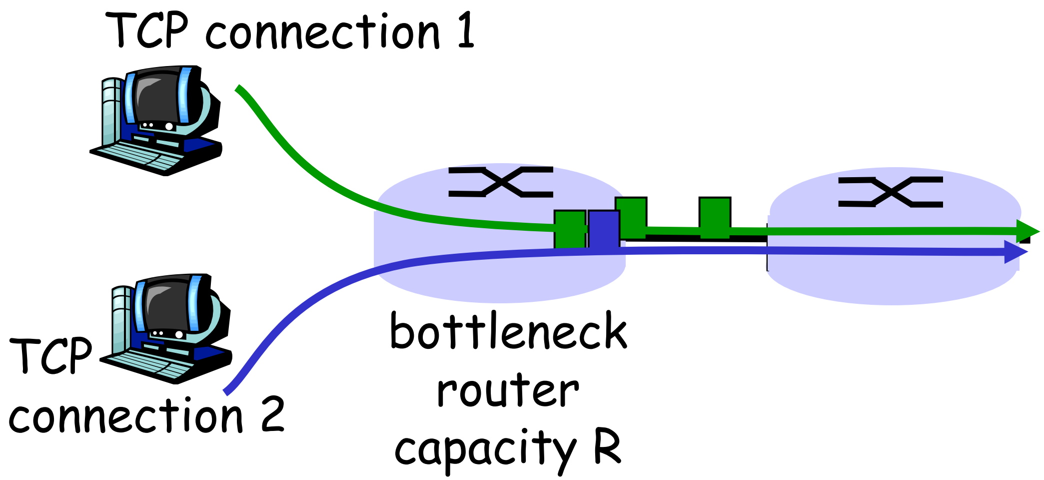
- Example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- Requires window size W = 83,333 in-flight segments
- Throughput in terms of loss rate:

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

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

TCP Fairness

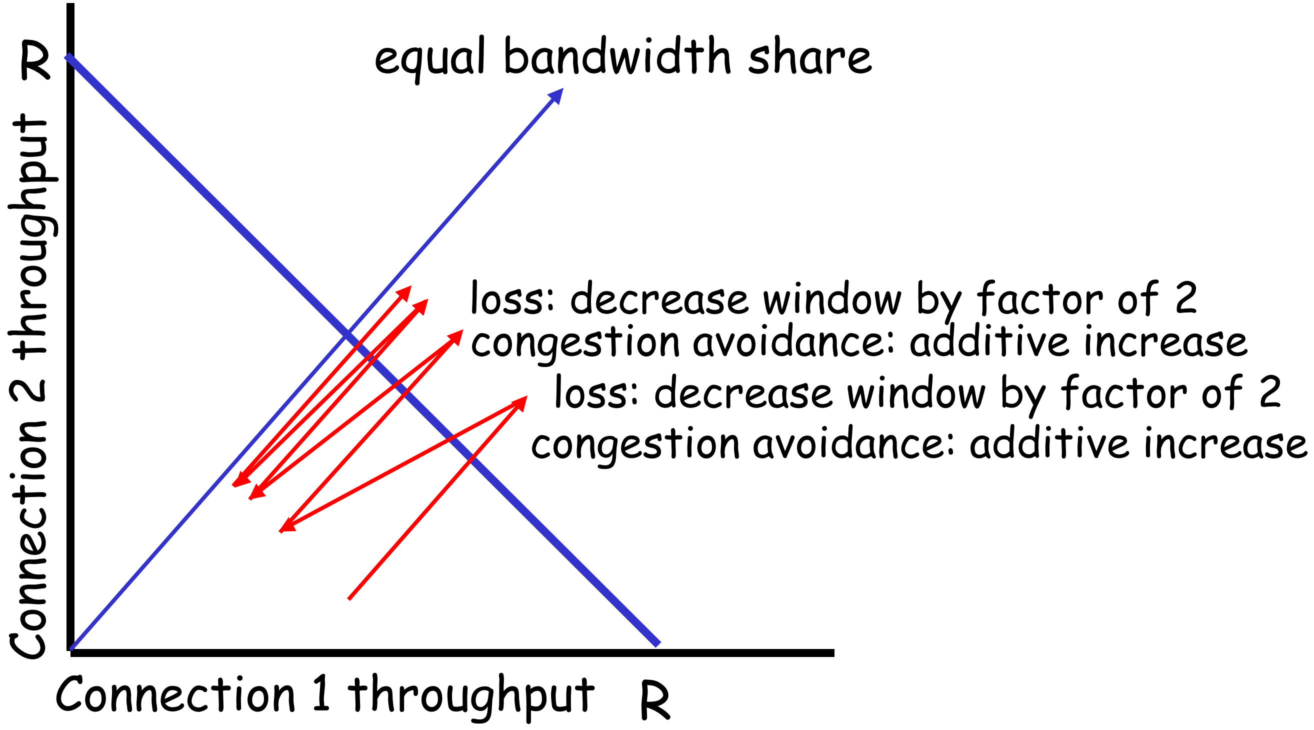
Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



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 connections between 2 hosts.
- Web browsers do this
- Example: link of rate R supporting 9 connections;
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2 !