Chapter 3
TRANSPORT

Distributed
Computing
Group

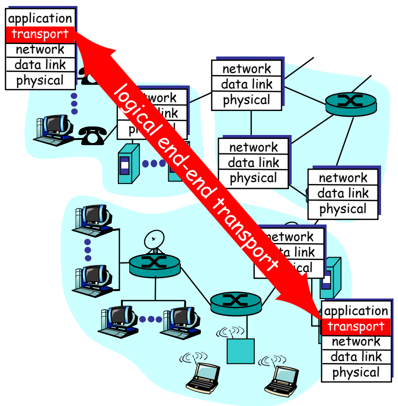
Computer Networks
Winter 2002 / 2003

Overview

- Transport layer services
- Multiplexing/Demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - reliable transfer
  - flow control
  - connection management
- Principles of congestion control
- Introduction to Queuing Theory
- TCP congestion control

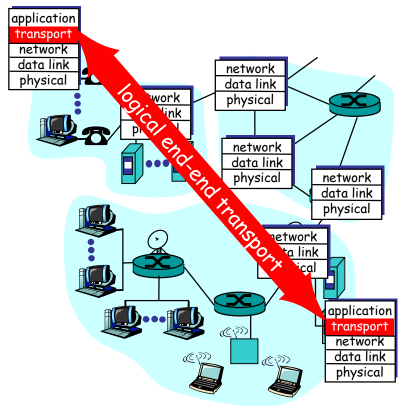
Transport services and protocols

- provide *logical communication* between application processes running on different hosts
- transport protocols run in end systems
- transport vs. network layer services
  - *network layer*
    - data transfer between end systems
  - *transport layer*
    - data transfer between processes
    - relies on, enhances, network layer services



Transport-layer protocols

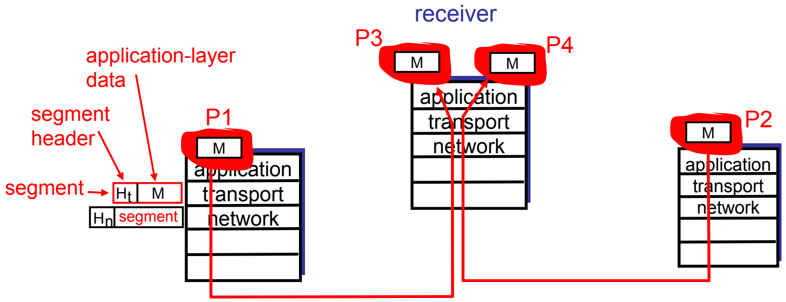
- Internet transport services
  - reliable, in-order unicast delivery (TCP)
    - congestion control
    - flow control
    - connection setup
  - unreliable ("best-effort"), unordered unicast or multicast delivery (UDP)
- Services not available
  - real-time / latency guarantees
  - bandwidth guarantees
  - reliable multicast



Multiplexing/Demultiplexing

- *Segment:* unit of data exchanged between transport layer entities
- a.k.a. TPDU: transport protocol data unit

Demultiplexing: delivering received segments to correct application layer processes

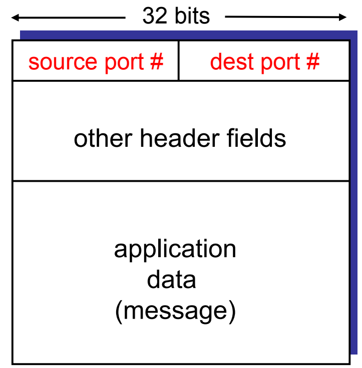


Multiplexing/Demultiplexing

Multiplexing:
gathering data from multiple application processes, enveloping data with header (later used for demultiplexing)

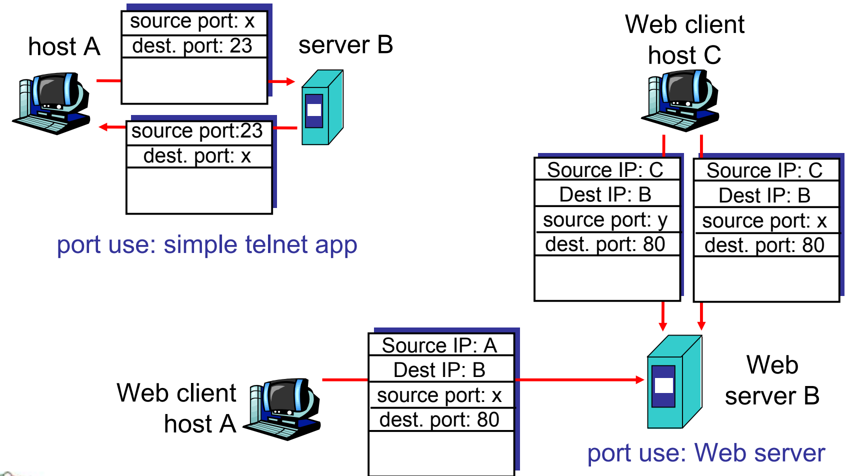
multiplexing/demultiplexing:

- based on sender, receiver port numbers, IP addresses
- source, destination port numbers in each segment
- recall: well-known port numbers for specific applications



TCP/UDP segment format

Multiplexing/Demultiplexing: Examples



UDP: User Datagram Protocol

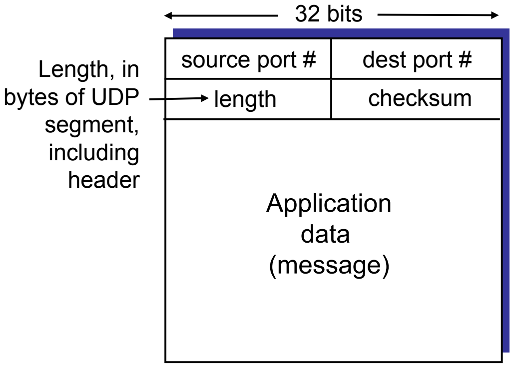
- RFC 768
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be
  - lost
  - delivered out of order to application
- UDP is *connectionless*
  - no handshaking between UDP sender and receiver
  - each UDP segment handled independently of others

Why is there a 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 Segment Structure

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses
  - DNS
  - SNMP
  - [Why?]
- reliable transfer over UDP
  - add reliability at application layer
  - application-specific error recovery



UDP segment format

UDP checksum

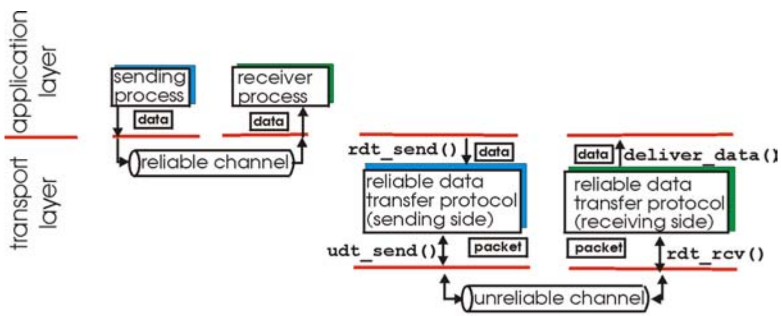
Goal: detect "errors" (e.g., flipped bits) in transmitted segment

Sender
- treat segment contents as sequence of 16-bit integers
- checksum: 1's complement sum of addition of segment contents
- sender puts checksum value into UDP checksum field

Receiver
- add all 16-bit integers (including checksum)
- check if computed sum is "11…1"
  - NO → error detected
  - YES → no error detected. *But maybe errors nonetheless?!?* More later …

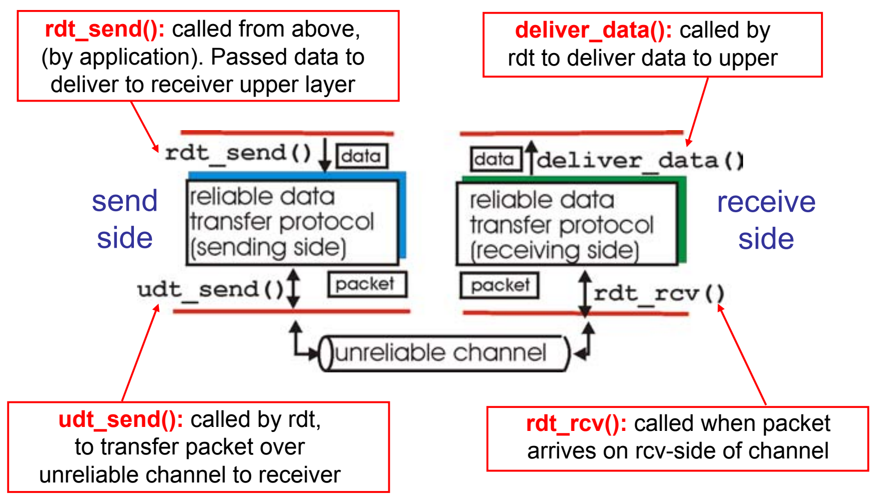
Principles of Reliable data transfer

- Important in applications, transport, link layers
- On the top 10 list of important networking topics...
- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)



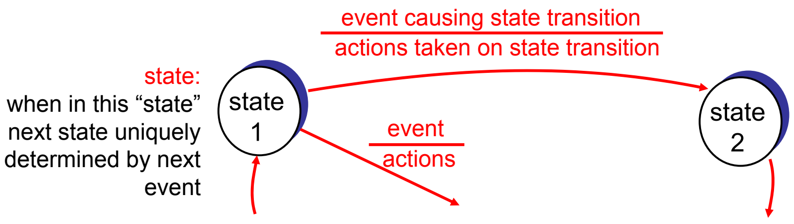
(a) provided service (b) service implementation

Reliable data transfer: getting started



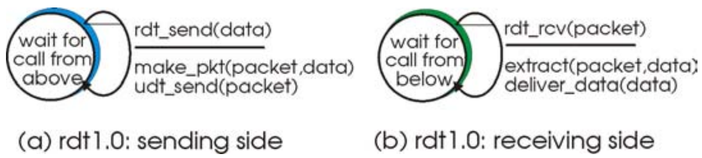
Reliable data transfer: getting started

We will
- 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



rdt 1.0: Reliable transfer over a reliable channel

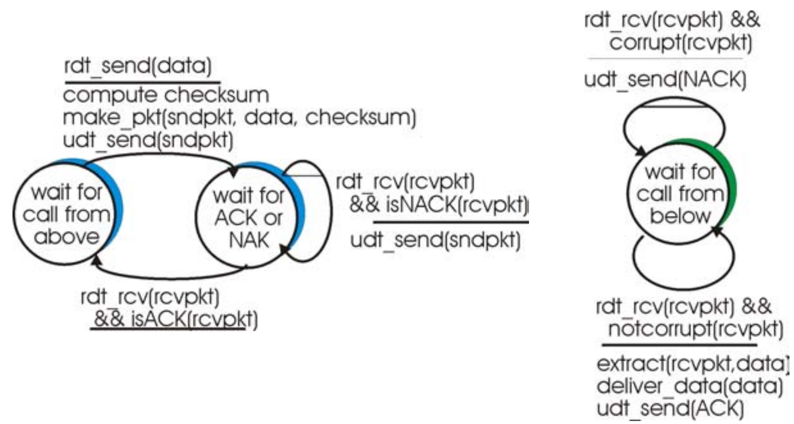
- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver
  - sender sends data into underlying channel
  - receiver reads data from underlying channel



rdt 2.0: channel with bit errors

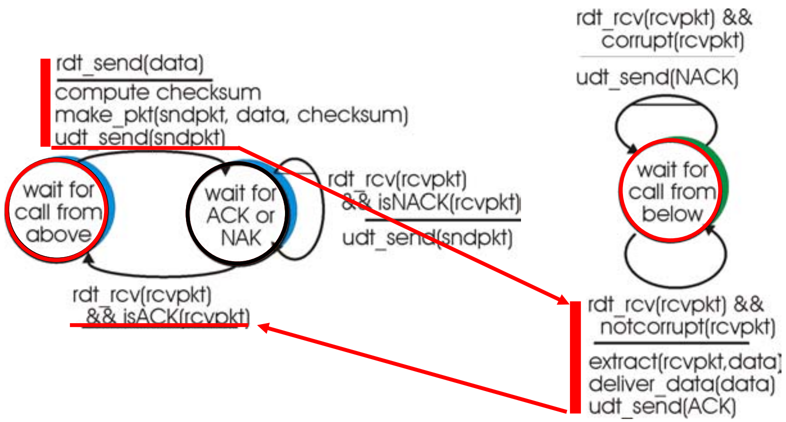
- There is no packet loss
- Underlying channel may flip bits in packet
  - recall: UDP checksum to detect bit errors
- Question: How do we recover from errors?
  - human scenarios?
  - *acknowledgements (ACKs):* receiver explicitly tells sender that packet received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that packet had errors
  - sender retransmits packet on receipt of NAK
- New mechanisms in rdt 2.0 (beyond rdt 1.0):
  - error detection
  - receiver feedback: control messages (ACK,NAK) receiver $\rightarrow$ Sender
  - retransmission

rdt 2.0: FSM specification



sender FSM　　receiver FSM

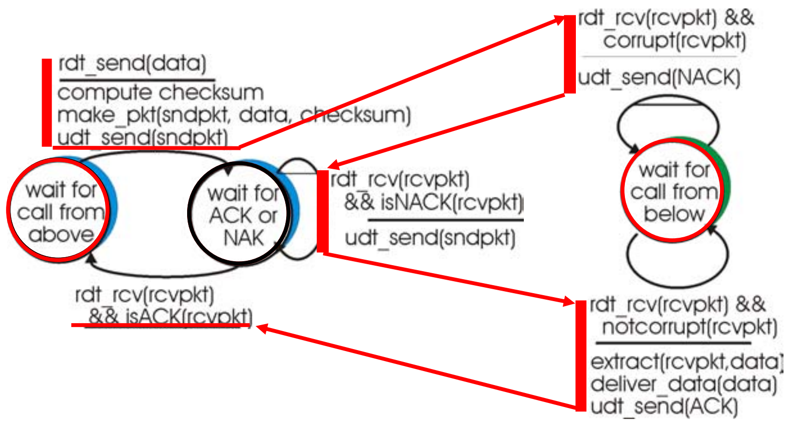
rdt 2.0 in action (no errors)



sender FSM

receiver FSM

rdt 2.0 in action (error scenario)



sender FSM

receiver FSM

rdt 2.0 has a fatal flaw!

- What happens if ACK/NAK is corrupted?
  - sender doesn't know what happened at receiver!
  - can't just retransmit: possible duplicate
- What to do?
  - sender ACKs/NAKs receiver's ACK/NAK? What if sender ACK/NAK lost?
  - retransmit, but this might cause retransmission of correctly received packet!

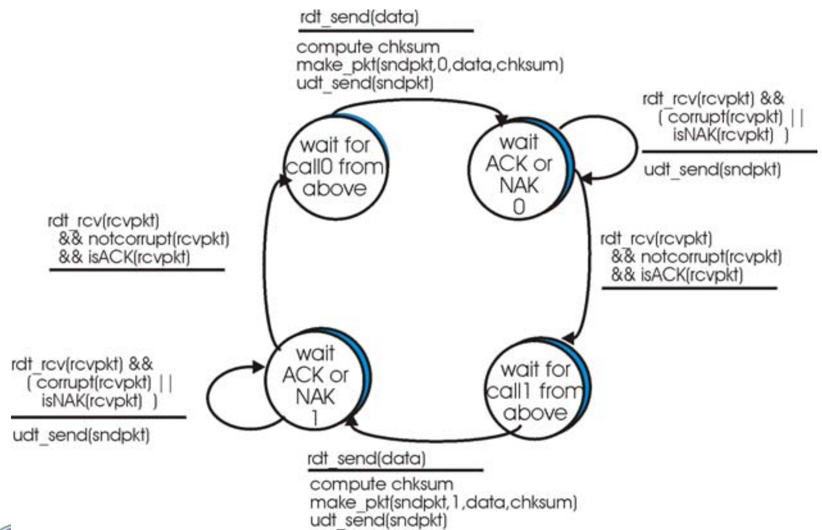
Handling duplicates

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

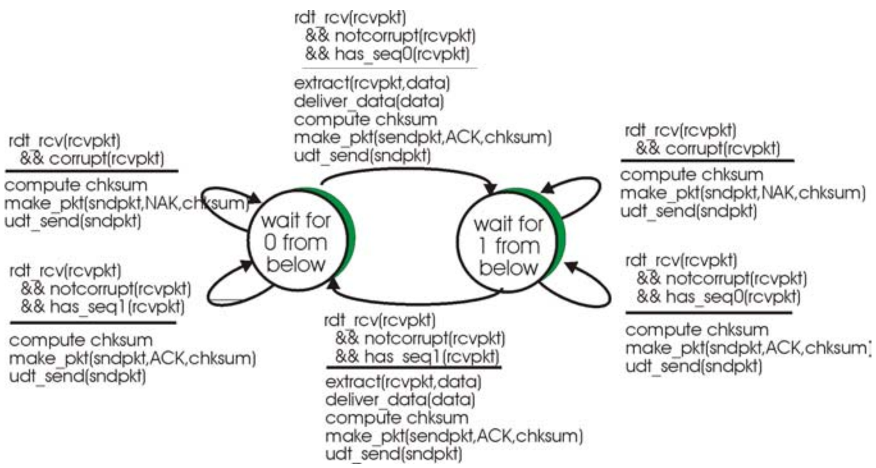
stop and wait

Sender sends one packet, then waits for receiver response

rdt 2.1: handles garbled ACK/NAKs (sender side)



rdt 2.1: handles garbled ACK/NAKs (receiver side)



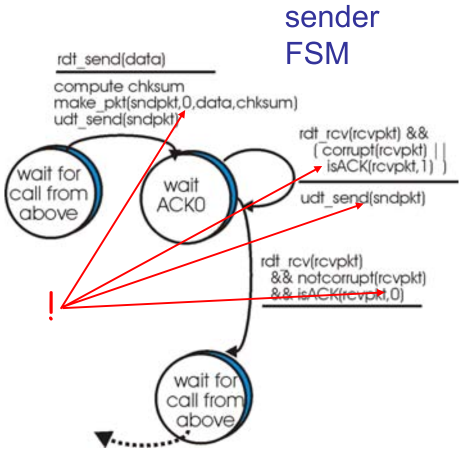
rdt 2.1: Discussion

Sender
- sequence number added to packet
- two sequence numbers (0,1) are sufficient. Why?
- must check if received ACK/NAK corrupted
- twice as many states because state must "remember" whether "current" packet has sequence number 0 or 1.

Receiver
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected packet sequence number
- note: receiver *cannot* know if its last ACK/NAK was received OK by sender

rdt 2.2: a NAK-free protocol

- same functionality as rdt 2.1, using ACKs only
- instead of NAK, receiver sends ACK for last packet received OK
  - receiver must *explicitly* include sequence number of packet being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current packet*



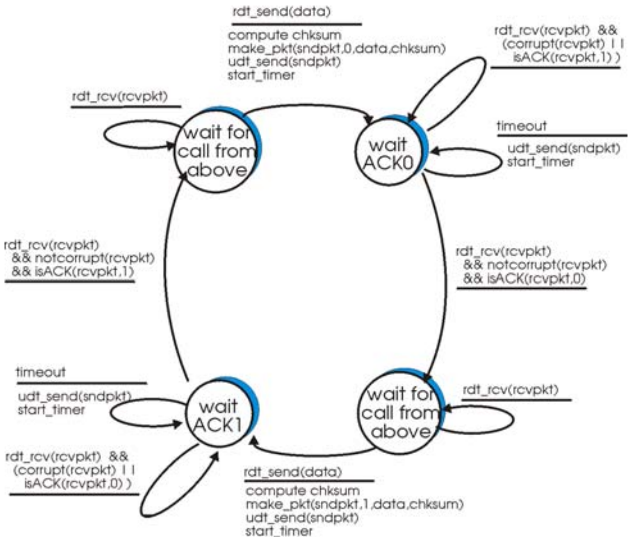
rdt 3.0: channels with errors *and* loss

- New assumption: underlying channel can also lose packets (data or ACKs)
  - checksum, seq. #, ACKs, retransmissions will be of help, but not enough

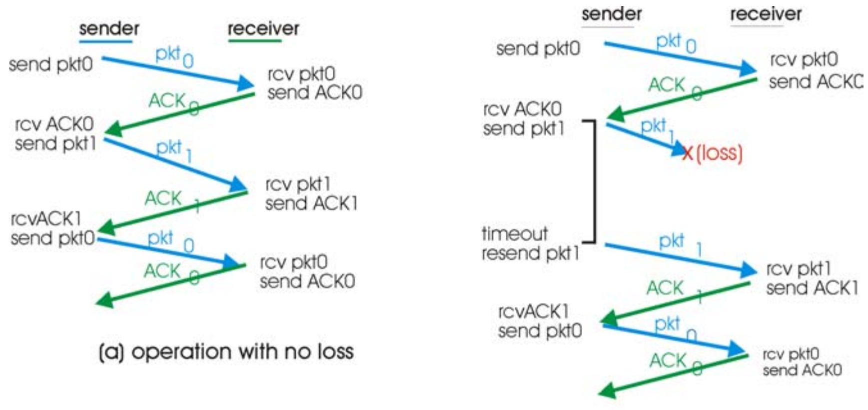
Q: How can we deal with loss?
- sender waits until certain data or ACK lost, then retransmits
- yuck: drawbacks?

- Approach: sender waits "reasonable" amount of time for ACK
- Sender retransmits if no ACK received in this time
- If packet (or ACK) just delayed (but not lost):
  - retransmission will be duplicate, but use of sequence numbers already handles this
  - receiver must specify sequence number of packet being ACKed
- Requires countdown timer

rdt 3.0 (sender side)



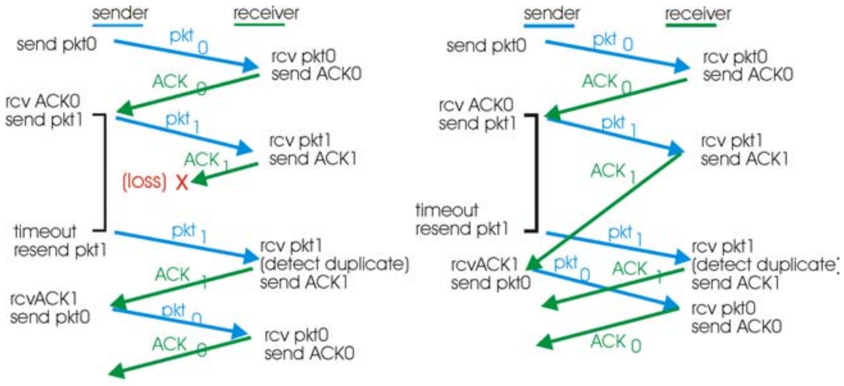
rdt 3.0 in action



(a) operation with no loss

(b) lost packet

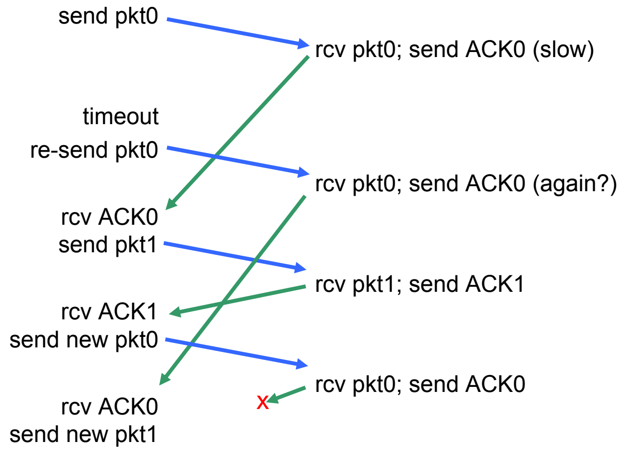
rdt 3.0 in action



(c) lost ACK

(d) premature timeout

rdt 3.0 in action (more problems?)



now everything seems to be OK… Problem: FIFO channel

Performance of rdt 3.0

- Back of envelope calculation of performance of rdt 3.0
- example: 1 Gbps link, 15 ms propagation delay, 1kB packet [b=bit, B=Byte, Gbps = Gb/s]

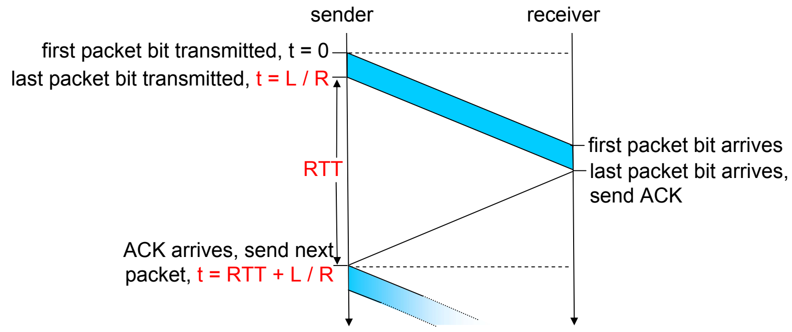
$$T_{transmit} = \frac{8kb/pkt}{10^9 b/s} = 8\mu s/pkt$$

- With the propagation delay, the acknowledgement arrives 30.008ms later (assuming that nodal and queuing delay are 0)
- That is, we only transmit 1kB/30.008ms instead of 1Gb/s

$$\text{Utilization U} = \frac{8kb/30.008ms}{1Gb/s} \approx 0.027\%$$

- network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation



$$\text{Utilization U} = \frac{L/R}{RTT + L/R} \approx 0.027\%$$

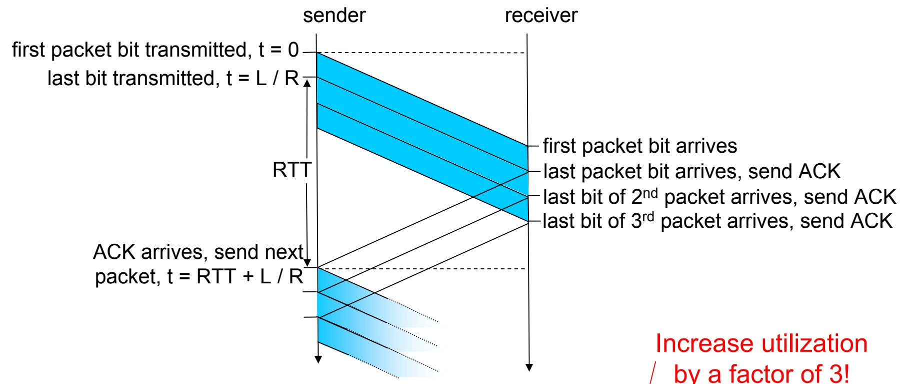
Pipelined protocols

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

- There are two generic forms of pipelined protocols
  - *go-Back-N* and *selective repeat*

Pipelining: increased utilization



Increase utilization by a factor of 3!

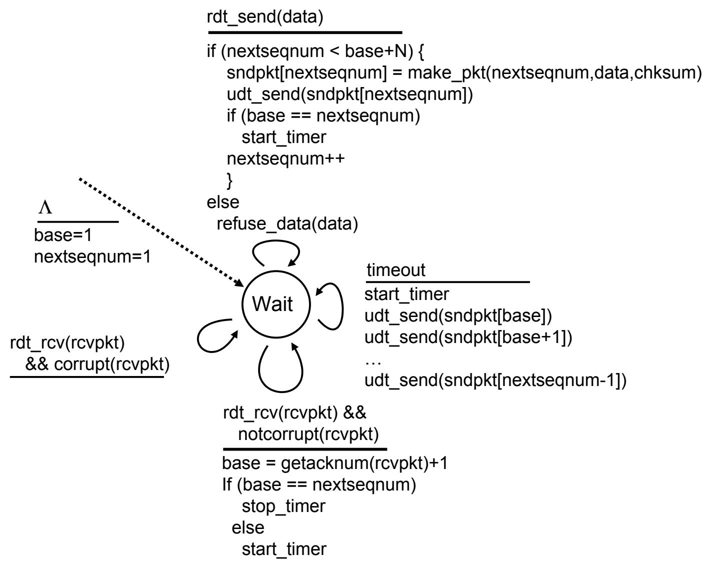
$$\text{Utilization U} = \frac{3 \cdot L/R}{RTT + L/R} \approx 0.08\%$$

Go-Back-N

Sender
- Multiple-bit sequence number in packet header
- "Window" of up to N consecutive unack'ed packets allowed

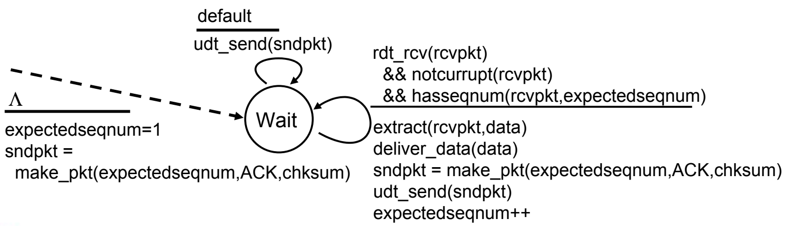
- ACK(n): ACKs all packets up to and including sequence number n
  - a.k.a. cumulative ACK
  - sender may get duplicate ACKs
- timer for each in-flight packet
- *timeout(n):* retransmit packet n and all higher seq# packets in window

GBN: sender extended FSM

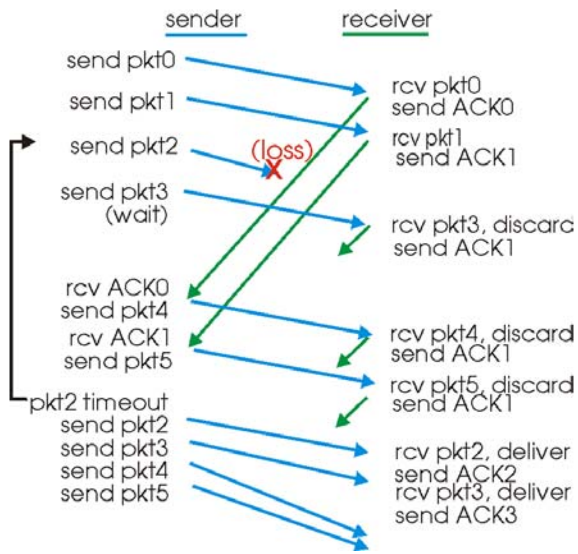


GBN: receiver extended FSM

Receiver (simple version)
- ACK-only: always send ACK for correctly-received pkt with highest *in-order* sequence number
  - may generate duplicate ACKs
  - need only to remember the expected sequence number
- out-of-order packet:
  - discard (don't buffer) → no receiver buffering!
  - re-ACK packet with highest in-order sequence number



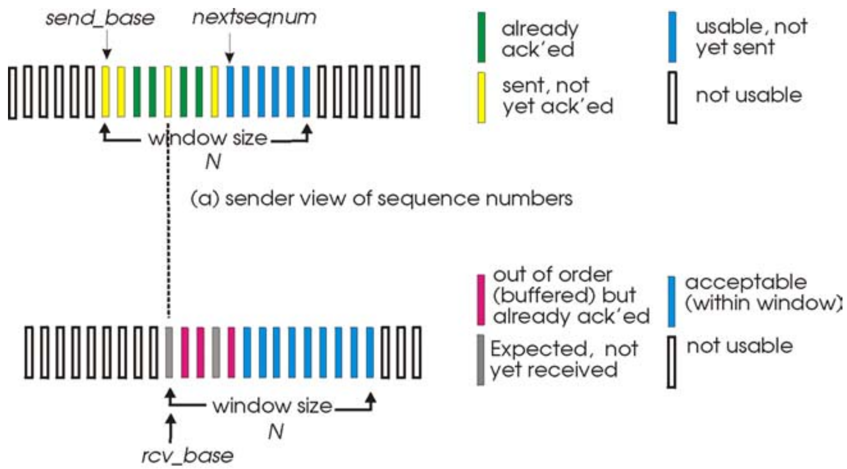
GBN in action



Selective Repeat

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

Selective repeat: sender, receiver windows



(a) sender view of sequence numbers

(b) receiver view of sequence numbers

Selective repeat

sender

Get data from layer above
- if next available sequence number in window, send packet

timeout(n)
- resend packet n, restart timer

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

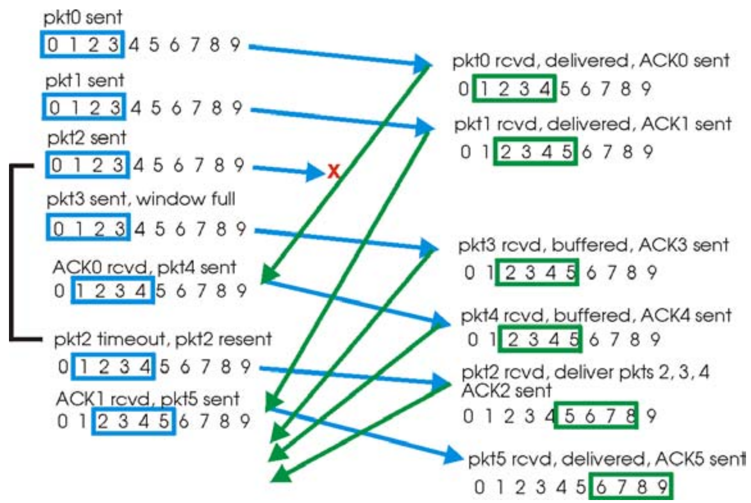
receiver

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

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

otherwise
- ignore

Selective repeat in action



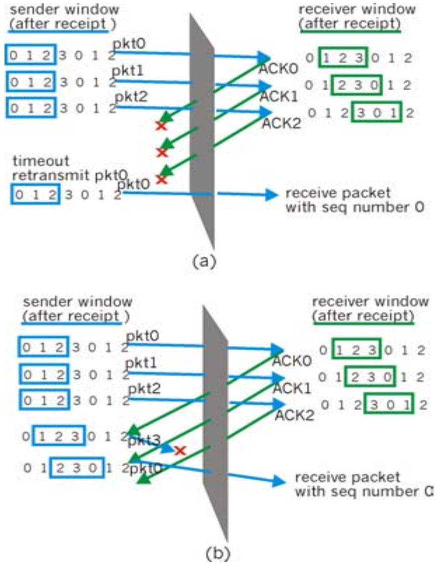
Selective repeat: dilemma

Example

- sequence numbers: 0…3
- window size = 3

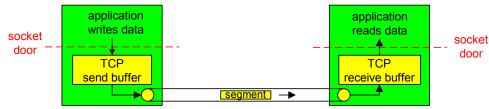
- Receiver sees no difference in two scenarios on the right…
- Receiver incorrectly passes duplicate data as new in scenario (a)

Q: What is the relationship between sequence number size and window size?

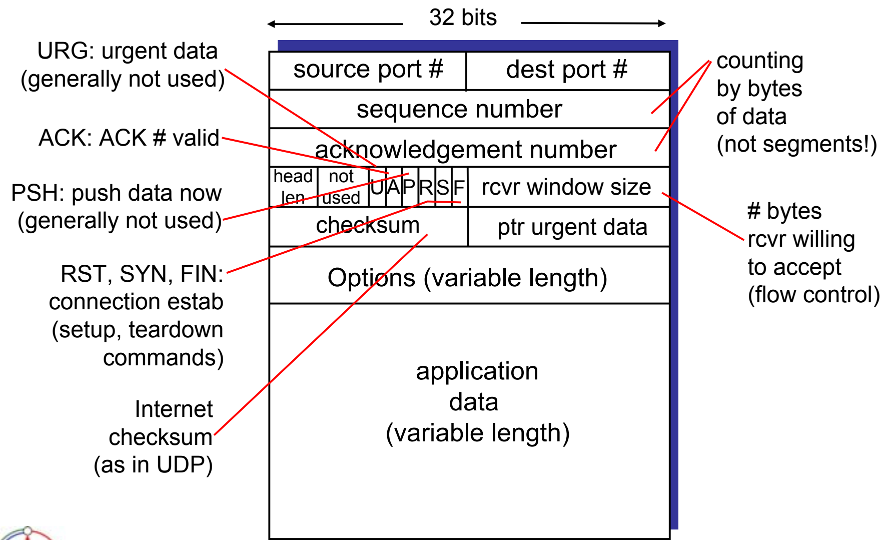


TCP: Overview

- RFCs
  - 793, 1122, 1323, 2018, 2581
- point-to-point
  - one sender, one receiver
- reliable, in-order *byte stream*
  - no "message boundaries"
- pipelined
  - send & receive buffers
  - TCP congestion and flow control set window size
- connection-oriented
  - handshaking (exchange of control msgs) to init sender and receiver state before data exchange
- full duplex data
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- flow controlled
  - sender will not overwhelm receiver



TCP segment structure



TCP sequence numbers and ACKs

Sequence numbers

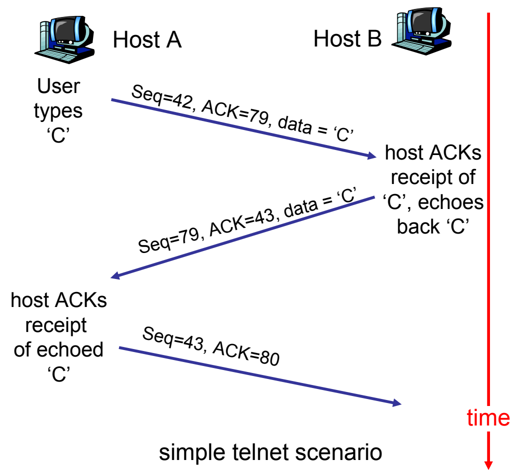
- byte stream "number" of first byte in segment's data

ACKs

- Sequence number of next byte expected from other side
- cumulative ACK

Q How does receiver handle out-of-order segments?

- TCP spec doesn't say; it is up to implementation!



simple telnet scenario

TCP Round Trip Time and Timeout

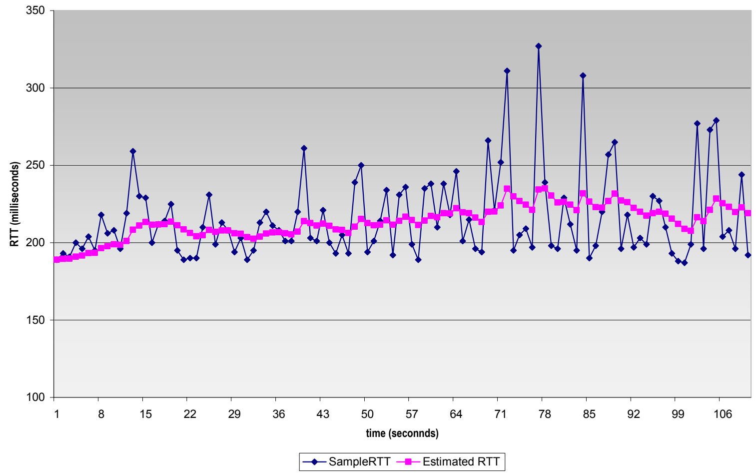
Q: How do we set TCP timeout value?

- longer than RTT
  - note: RTT will vary
- 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, cumulatively ACKed segments
- SampleRTT will vary, we want estimated RTT "smoother"
  - use several recent measurements, not just current SampleRTT

Example RTT estimation



TCP Round Trip Time and Timeout

`EstimatedRTT = (1-`α`)·EstimatedRTT + `α`·SampleRTT`

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

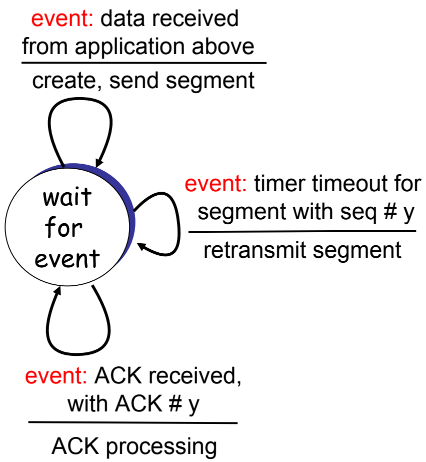
Setting the timeout

- **`EstimatedRTT`** plus "safety margin"
- large variation in **`EstimatedRTT`** $\rightarrow$ larger safety margin

`Timeout = EstimatedRTT + 4·Deviation`

`Deviation = (1-`β`)·Deviation`
`+ `β`·|SampleRTT-EstimatedRTT|`

TCP: reliable data transfer



- simplified sender, with
  - one way data transfer
  - no flow control
  - no congestion control

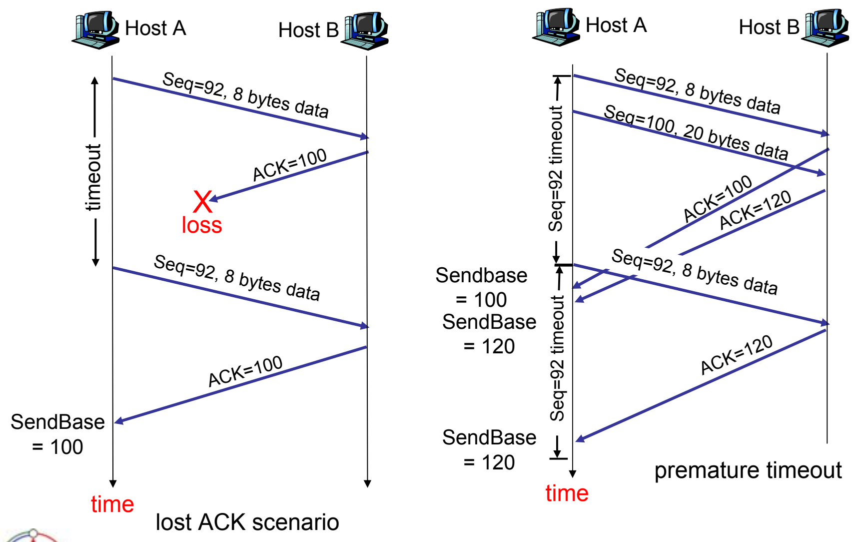
sender (simplified)

```
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 */
```

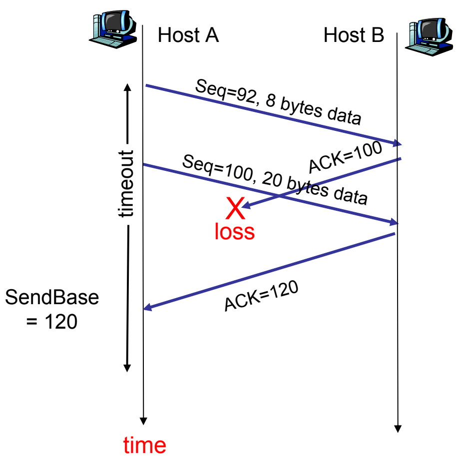
Comment:
- SendBase-1: last cumulatively ack'ed byte

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

TCP: retransmission scenarios



TCP: retransmission scenarios



Cumulative ACK scenario

TCP ACK generation (RFC 1122, RFC 2581)

Event	TCP Receiver action
in-order segment arrival, no gaps, everything else already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
in-order segment arrival, no gaps, one delayed ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
out-of-order segment arrival higher-than-expect seq. # gap detected	send duplicate ACK, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate ACK if segment starts at lower end of gap

Fast Retransmit

- Time-out period often 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.
- Hack: 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

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
               }
```

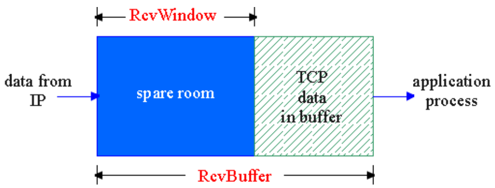
a duplicate ACK for already ACKed segment

fast retransmit

TCP Flow Control

- **RcvBuffer**
  – size of TCP Receive Buffer
- **RcvWindow**
  – amount of spare room in Buffer
- Receiver
  – explicitly informs sender of (dynamically changing) amount of free buffer space
  – **RcvWindow** field in TCP segment
- Sender
  – keeps the amount of transmitted, unACKed data less than most recently received **RcvWindow**

flow control
sender won't overrun receiver's buffers by transmitting too much, too fast



TCP Connection Management (opening connection)

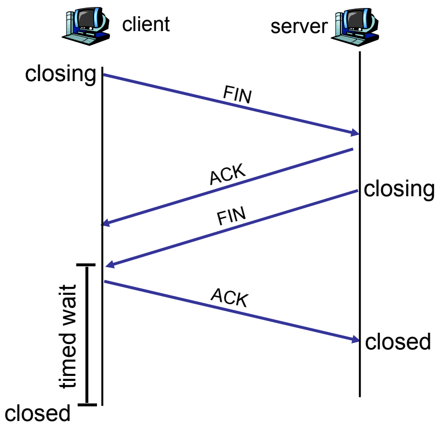
- Recall: TCP sender, receiver establish "connection" before exchanging data segments
- They initialize TCP variables
  – Sequence numbers
  – buffers, flow control info (e.g. RcvWindow)
- Client: connection initiator
  `Socket clientSocket = new Socket("host,port");`
- Server: contacted by client
  `Socket connectionSocket = welcomeSocket.accept();`

Three way handshake

1) client host sends TCP SYN segment to server
   – specifies initial seq. number
   – no data
2) server host receives SYN, replies with SYNACK segment
   – server allocates buffers
   – specifies server initial seq. #
3) client receives SYNACK, replies with ACK segment, which may contain data

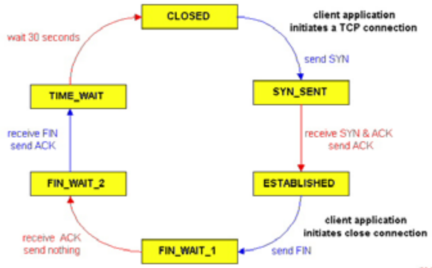
TCP Connection Management (closing connection)

clientSocket.close();



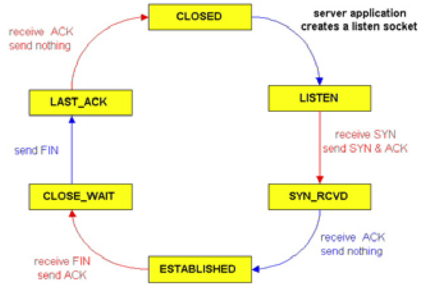
1) client end system sends TCP FIN control segment to server
2) server receives FIN, replies with ACK. Closes connection, sends FIN.
3) client receives FIN, replies with ACK. Enters “timed wait” - will respond with ACK to received FINs
4) server receives ACK. Connection closed.

TCP Connection Management (continued)



TCP client lifecycle

TCP server lifecycle



Principles of Congestion Control

- Different from flow control (both are often mistaken)
- Manifestations
  - long delays (queuing in router buffers)
  - lost packets (buffer overflow at routers)
- Another top 10 problem!

> **congestion**
> too many sources sending too much data too fast for *network* to handle

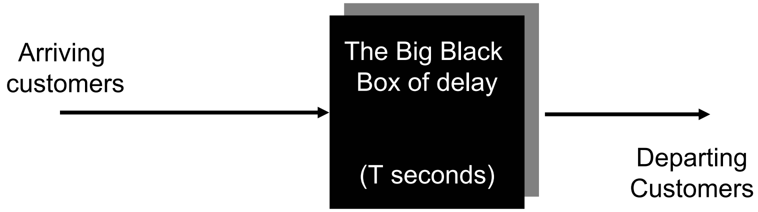
- Example:
  - Router with infinite buffer size can handle 1Mb per second.
  - There are 10 connections through router with 200kb/s each.
  - Delays are growing with time!
  - Question: How long are delays if 10 connections have 150kb/s only? What about 100 kb/s? 90kb/s? 50kb/s? 10kb/s?!?

Excursion to Queuing Theory

- Queuing theory considers a system where “customers” enter, wait, and then leave.
- For example, banks, parking lots, stores, multi-user operating systems, router queues, networks of routers.
- There are complete courses for queuing theory.
- We do queuing theory for dummies only.
- Queuing theory studies the following
  - Arrival Process (distribution of arrivals)
  - Service Process (distribution of process completion)
  - Buffer size
  - Number of servers
  - Service discipline (FIFO, LIFO, etc)
  - Queuing networks

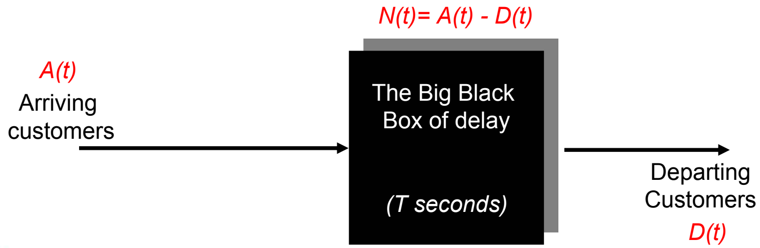
What we want out of this

- We use queuing theory to determine qualities like
  - Average time spent by a customer/packet in the system or queue
  - Average number of customers/packets in system or queue.
  - Probability a customer will have to wait a certain large amount of time.



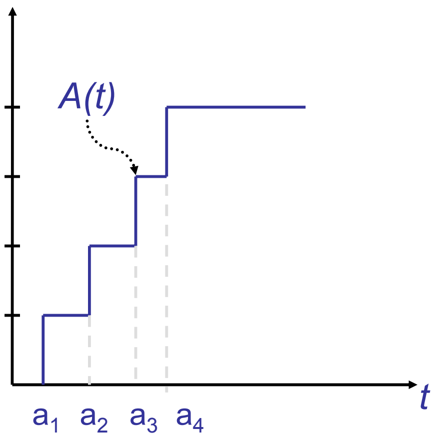
Some terms

- Each customer spends *T* seconds in the box, representing service time.
- We assume that system was empty at time *t* = 0.
- Let *A(t)* be the number of arrivals from time *t* = 0 to time *t*.
- Let *D(t)* be the number of departures.
- Let *N(t)* represent the number of customers in the system at time *t*.
- Throughput: average number of customers/messages per second that pass through the system.

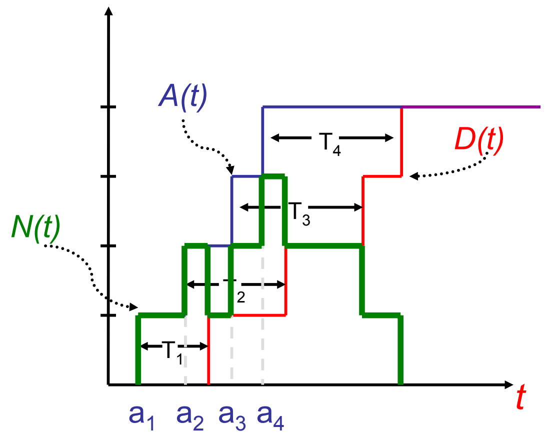


Arrival Process

- Let a_1 be the 1st arrival in the system. The 2nd comes a_2 time units later.
- Therefore the *n*th customer comes at time $a_1 + a_2 + a_3 + \ldots + a_n$.



Average number in system at time t



Arrivals, Departures, Throughput

- The average arrival rate λ, up to the time when the n^{th} customer arrives is $n / (a_1 + a_2 + \ldots + a_n) = \lambda$ customers/sec
- Note the average interarrival rate of customers is the reciprocal of λ: $(a_1 + a_2 + \ldots + a_n) / n$ sec/customer
- Arrival rate = 1/(mean of interarrival time)
- The long-term arrival rate λ is therefore $\boxed{\lambda = \lim_{t \to \infty} \frac{A(t)}{t}}$ cust./sec.
- Similarly, we can derive throughput μ
- Throughput $\boxed{\mu = \lim_{t \to \infty} \frac{D(t)}{t}}$ customers/sec
- Note the average service time is $1/\mu$.

Example

- We are in line at the bank behind 10 people, and we estimate the teller taking around 5 minutes/per customer.
- The throughput is the reciprocal of average time in service
  = 1/5 persons per minute
- How long will we wait at the end of the queue?
  The queue size divided by the processing rate
  = 10/(1/5) = 50 minutes.

Offered Load (or Traffic Intensity)

- If we have the arrival rate, and the throughput (the rate at which customers leave), then we can define the offered load ρ as

$$\boxed{\rho = \lambda/\mu}$$

- If the offered load is less than 1, and if packets arrive and depart regularly, then there is no queuing delay.
- If the offered load is less than 1, and packets arrive not quite regularly (there will be bursts now and then), we will have queuing delay. However, packets will be serviced eventually.
- Long term offered load greater than (or equal to) one will cause infinite delay (or dropped packets).

Little's Law

- We have
  - the arrival rate λ
  - and the average number of customers E[N]
- **Little's law** relates the average time spent in the system E[T], to the arrival rate λ, and the avg number of customers E[N], as follows

$$\boxed{E[N] = \lambda \cdot E[T]}$$

- First some examples, then let's derive it!

Example

- In a bank, customers have an arrival rate of 4 per hour. Customers are served at a rate of 6 per hour. The average time customers spend in the bank is 25 minutes.
- Is the system stable?
- What is the average number of customers in the system?

- $\rho = \lambda/\mu = (4/60) / (6/60) = 2/3 < 1$. Yes, the system is stable!
- $E[N] = \lambda E[T] = (4/60)\cdot(25) = 5/3$ customers

Example (Variations of Little's Law)

- What is the average queue length, $E[N_q]$?
- $E[N_q] = \lambda E[Q]$, where E[Q] is the average time spent in queue.
- Customers enter at rate $\lambda = 4$/hour.
- We know average service time is $1/\mu = 1/(6/60) = 10$ min.
- Average time spent in system is 25, thus in queue 25-10=15.
- Average queue length: $E[N_q] = \lambda\, E[Q] = (4/60)\cdot(15) = 1$.

- What is the average number of customers in service, $E[N_s]$?
- $E[N_s] = \lambda E[X]$, where $E[X] = E[T] - E[Q] = 1/\mu$
- $E[N_s] = \lambda\,(1/\mu) = (4/60)\cdot 10 = 2/3 = \rho$

- Average in queue 1, average in service 2/3, average in system 5/3.

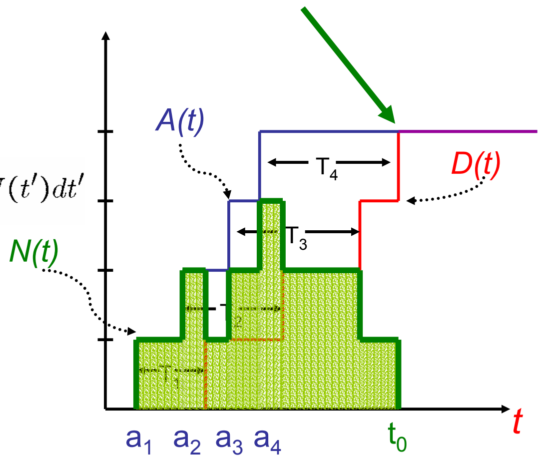
Deriving Little: Step 1

We look at a special point in time t_0 with $N(t_0) = A(t_0) – D(t_0) = 0$.

The average number in the system for $[0,t_0)$ is

$$E[N] = \lim_{t_0\to\infty} \frac{1}{t_0}\int_0^{t_0} N(t')dt'$$

The integral is equivalent to the averaged sum of time spent by the first $A(t_0)$ customers.

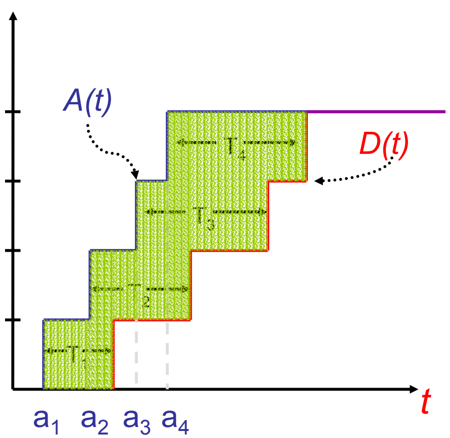


Deriving Little: Step 2

Each customer contributes T_i time to the integral.

The integral is equivalent to the averaged sum of times spent by the first $A(t_0)$ customers.

$$\frac{1}{t_0}\int_0^{t_0} N(t')dt' = \frac{1}{t_0}\sum_{j=1}^{A(t_0)} T_j$$



Deriving Little: Step 3

- We extend the last equation by $A(t_0)/A(t_0)$ to equation (1):

$$\frac{1}{t_0}\int_0^{t_0} N(t')dt' = \frac{A(t_0)}{A(t_0)}\frac{1}{t_0}\sum_{j=1}^{A(t_0)} T_j = \left(\frac{A(t_0)}{t_0}\right)\left(\frac{1}{A(t_0)}\sum_{j=1}^{A(t_0)} T_j\right)$$

- By definition we have $\lambda = A(t_0) / t_0$.
- We also have

$$E[T] = \lim_{A(t_0)\to\infty} \frac{1}{A(t_0)}\sum_{j=1}^{A(t_0)} T_j$$

- Then equation (1) is Little's Law: $E[N] = \lambda \cdot E[T]$
- Little's Law applies to any work-conserving system: one where customers are serviced in any order, but there is never an idle period if customers are waiting. It works for FIFO, LIFO, etc.

Random Variables & Binomial RV

- Random variables define a real valued function over a sample space. The value of a random variable is determined by the outcome of an experiment, and we can assign probabilities to these outcomes.
- Example: Random variable X of a regular dice:
  P[X=i] = 1/6 for any number i=1,2,3,4,5,or 6.
- Suppose a trial can be classified as either a success or failure. For a RV X, let X=1 for an arrival, and X=0 for a non-arrival, and let p be the chance of an arrival, with p = P[X=1].
- Suppose we had n trials. Then for a series of trials, a binomial RV with parameters (*n*,*p*) is the probability of having exactly *i* arrivals out of *n* trials with independent arrival probability *p*:

$$p(i) = \binom{n}{i} p^i (1-p)^{n-i}$$

Poisson Random Variables

- It is hard to calculate Binomial Random Variables, however, they can be approximated with Poisson Random Variables.
- With $\lambda = np$, the distribution of a Poisson RV is

$$p(i) = P[X = i] = e^{-\lambda}\frac{\lambda^i}{i!}$$

- The mean is λ
- Given an interval [0,t]. Let N(t) be the number of events occurring in that interval. (Parameter is λt: n subintervals in [0,t]; the prob of an event is p in each, i.e., $\lambda t = np$, since average rate of events is λ and we have t time.) Without additional derivation, we get

$$\boxed{P[N(t) = k] = e^{-\lambda t}\frac{(\lambda t)^k}{k!}}$$

- The number of events occurring in any fixed interval of length t is stated above. (It's a Poisson random variable with parameter λt.)

Exponential Random Variables

- The exponential RV arises in the modeling of the time between occurrence of events, for example packet inter-arrival times
- Again consider the interval [0,t] with $np = \lambda t$. What is the probability that an inter-event time T exceeds t seconds.

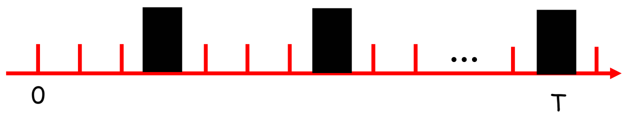
$$P[T > t] = (1-p)^n = (1-\lambda t/n)^n \approx e^{-\lambda t}$$

- For an exponential random Variable T with parameter λ

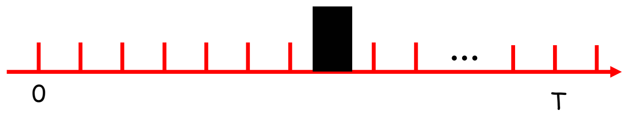
$$F(t) = 1 - e^{-\lambda t}, f(t) = \lambda e^{-\lambda t}, t \geq 0$$

- For a Poisson random variable, the time between the events is an exponentially distributed random variable, and vice versa.

Relationship Between RVs



- The interval [0,T] is divided into n sub-intervals.
- The number of packets arriving is a binomial random variable.
- With a large number of trials, it approaches a Poisson RV.



- The number of trials (time units) until the arrival of a packet is a geometric random variable.
- With a large number of trials, it approaches a exponential RV.

Memoryless Property

- The exponential random variable satisfies the "memoryless" property.
- The probability of having to wait at least *h* seconds is

$$\boxed{P[X > h] = e^{-\lambda h}}$$

- The probability of having to wait *h* additional seconds given that one has already waited *t* seconds, is

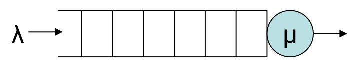
$$P[X > t+h | X > t] = \frac{P[X > t+h]}{P[X > t]} = \frac{e^{-\lambda(t+h)}}{e^{-\lambda t}} = e^{-\lambda h}$$

Kendall Notation

- Queuing systems are classified by a specific notation denoting:
  1. The customer arrival pattern
  2. The service time distribution
     - 1 and 2 can be either M = Markov (Poisson or Exponential), D = Deterministic, E_k = Erlang with param. k, G = General
  3. The number of servers
  4. The maximum number of customers in the system (std. = ∞)
  5. Calling population size (std. = ∞)
  6. Queuing discipline (FIFO, LIFO, etc.; std. = FIFO)
- Examples:
  - M/M/1: Markov inter-arrivals, Markov service times, 1 server.
  - M/D/c/K: Markov inter-arrivals, deterministic service times, c servers, K customers can queue.

M/M/1 Queue

- The most basic queuing analysis.
- Let p_0 be the probability of that the system is idle.



- The system is defined to be in the *equilibrium*, so what goes in must come out. This gives:
- $\lambda = p_0 \cdot 0 + (1-p_0) \cdot \mu$ (idle: nobody goes out; not idle: μ go out)
- Then $1-p_0 = \lambda/\mu = \rho$, thus $p_0 = 1-\rho$.
- With other words, the probability that an M/M/1 system is not idle is ρ; that's why ρ is also called the *traffic intensity* or *utility*.

M/M/1 Queue

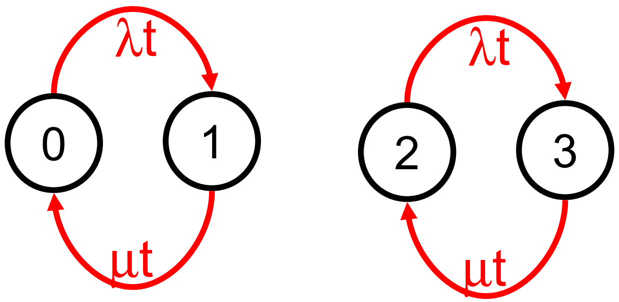
- Since arrival and service process are both Markov, we know that $E[A(t)] = \lambda t$ and $E[D(t)] = \mu t$.
- With some derivation, we can figure out probabilities and expected means of
  - The mean number of customers in the system
  - The mean time customers spend in the system
  - The mean number queued up
  - The mean time spent being queued up
- To do this we are going to set up a state diagram.

States

- Let the "state" of our system be equal to the number of customers in the system.
- The M/M/1 queue is memoryless. This means that the transition to a new state is independent of the time spent in the current state, all that matters is the number of customers in the system.
- In the equilibrium, the probability of being in state *i* is denoted by p_i. The probabilities p_i become independent of time.
- (Remark: p_0 is the probability that nobody is in the system.)

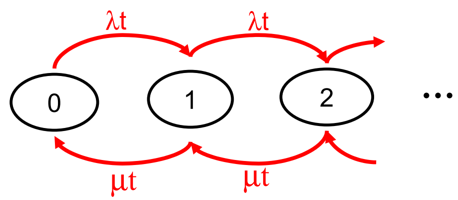
Markovian Models

- For any small interval of time *t*, there is a small chance of an arrival, and a small chance of a departure.
- If we make *t* small enough the chance of both a departure and arrival is negligible.



Markov Chain of M/M/1

- For the M/M/1 queue, we have infinitely many states and the following set of transition probabilities between them



- Because we are in the equilibrium (eq, the flow between states (the transition probabilities) must balance, that is:

$$(\lambda p_i)t = (\mu p_{i+1})t \quad \rightarrow \quad \rho \cdot p_i = p_{i+1}$$

What is the mean number of customers?

- We therefore express p_i as $p_i = \rho^i \cdot p_0$
- All probabilities must sum up to 1, that is

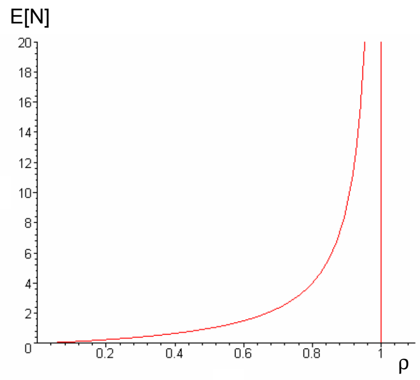
$$1 = \sum_{i=0}^{\infty} p_i = \sum_{i=0}^{\infty} (\rho^i \cdot p_0) = p_0 \sum_{i=0}^{\infty} \rho^i = p_0 \frac{1}{1-\rho}$$

- We have $p_0 = 1-\rho$ (we knew this already). We get $p_i = \rho^i(1-\rho)$
- This tells us the probability of having *i* customers in the system.
- We can find the mean easily:

$$\begin{aligned} E[N] &= \sum_{i=0}^{\infty} i \cdot p_i = (1-\rho) \sum_{i=0}^{\infty} i \cdot \rho^i \\ &= (1-\rho) \frac{\rho}{(1-\rho)^2} = \frac{\rho}{1-\rho} \end{aligned}$$

M/M/1 summary

- In the equilibrium, the number of customers in the system is $E[N] = \rho/(1-\rho)$, as shown in the chart on the right hand side.
- You can see that the number grows infinitely as ρ goes to 1.
- We can calculate the mean time in the system with Little's law: $E[T] = E[N]/\lambda = 1/(1-\rho)/\mu$.
- Since $E[X] = 1/\mu$, one can also calculate $E[Q]$ easily…

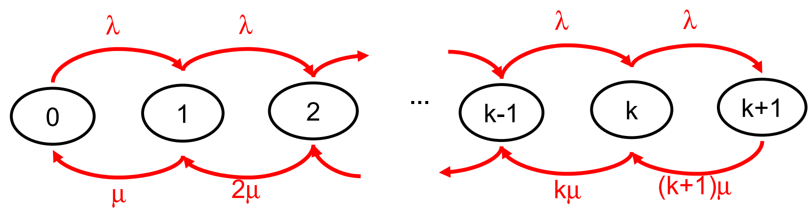


Example

- A local pre-school has 1 toilet for all the kids. On average, one toddler every 7 minutes decides he or she has to use the toilet (randomly with a Poisson distribution). Kids take an average of 5 minutes using the toilet.
- Is one bathroom enough if kids can hold it in for an unlimited amount of time? Yes, because $\rho = \lambda/\mu = (1/7) / (1/5) < 1$.
- If time to get to and from the bathroom is 1 minute, how long will a kid be gone from class on average?
$1+E[T]+1 = 2 + 1/(1-\rho)/\mu = 2 + 5 / (1-5/7) = 19.5$ minutes.
- George W. Bush visits the pre-school, and needs to go pee. He gets to the back of the line. He can only hold it in for 11 minutes. On average, would he make it to the toilet on time?
$E[Q] = E[T]-1/\mu = 12.5$ minutes… What's the probability…?

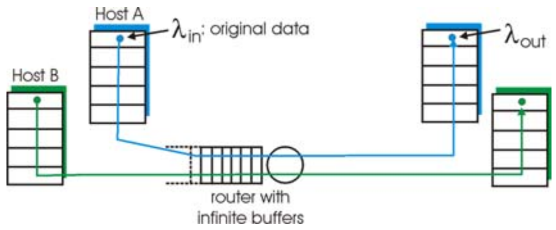
Birth-Death and Markov Processes

- The way we solved the M/M/1 "Markov chain" can be generalized:
- A *birth-death process* is where transitions are only allowed between neighboring states. A *Markov process* is where transitions are between any states; states do not need to be "one dimensional".
- You can solve such systems by the same means as M/M/1; probably the derivation is more complicated.
- Below is for example the birth-death process of $M/M/\infty$.

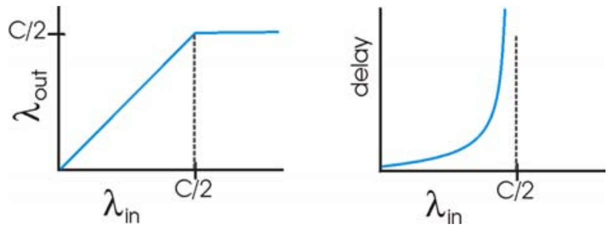


Back to Practice: Congestion scenario 1

- two equal senders, two receivers
- one router with infinite buffer space
- with capacity C
- no retransmission

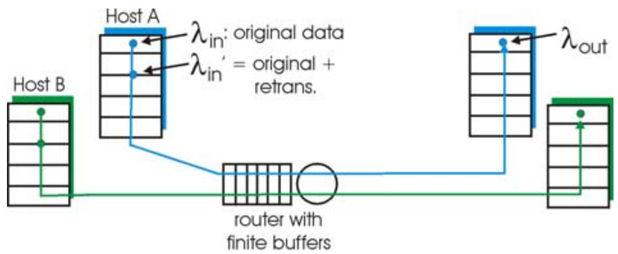


- large delays when congested
- maximum achievable throughput



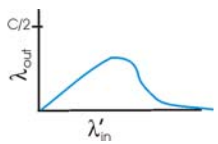
Congestion scenario 2

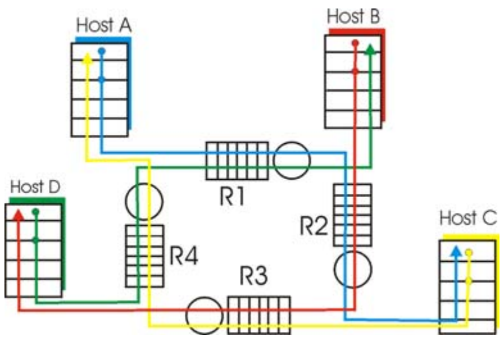
- one router with only *finite* buffer
- sender retransmission of lost packet
- more work for the same throughput



Congestion scenario 3

- A "network" of routers (queues), with multihop paths.
- Still analytically solvable when streams and routers are Markov.
- But there are retransmissions, timeouts, etc.
- Typical behavior: throughput gets worse with more and more input:





Approaches towards congestion control

Two types of approaches usually used:

End-end congestion control

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

Network-assisted cong. control

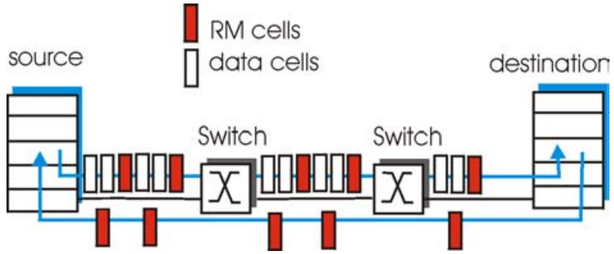
- routers provide feedback to end systems
  - single bit indicating congestion (used in SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should send at

Example for Network-Assisted Cong. Control: ATM ABR

ABR: available bit rate
- "elastic service"
- if sender's path "underloaded":
  - sender should use available bandwidth
- if sender's path congested:
  - sender is 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

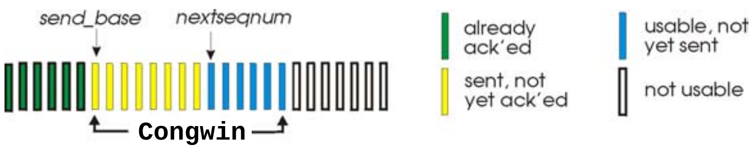
Example for Network-Assisted Cong. Control: ATM ABR



- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - Sender's 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

TCP Congestion Control

- end-end control (no network assistance)
- transmission rate limited by congestion window size, **`Congwin`**, over segments:



- w segments, each with MSS bytes sent in one RTT:

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

TCP Congestion Control

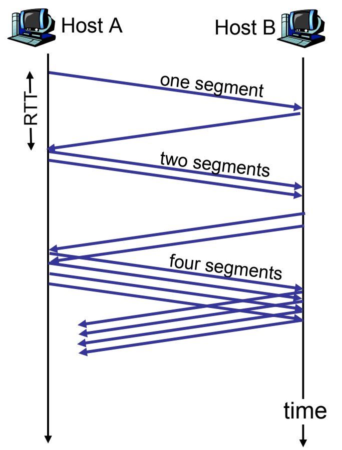
- "probing" for usable bandwidth
  - ideally: transmit as fast as possible (**`Congwin`** as large as possible) without loss
  - *increase* **`Congwin`** until loss (congestion)
  - loss: *decrease* **`Congwin`**, then begin probing (increasing) again
- TCP has two "phases"
  - slow start
  - congestion avoidance
- Important variables:
  - **`Congwin`**
  - **`Threshold:`** defines where TCP switches from slow start to congestion avoidance

TCP Slowstart

Slow start algorithm

initialize: Congwin = 1
for (each segment ACKed)
    Congwin++
until (loss event OR
    CongWin > threshold)

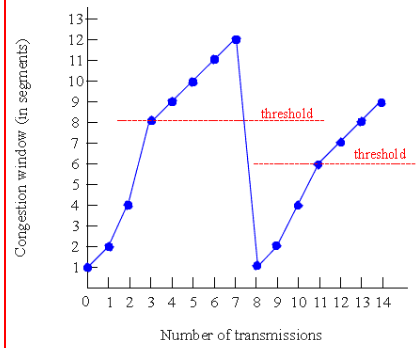
- exponential increase (per RTT) in window size (not so slow!)
- loss event: timeout (Tahoe TCP) and/or three duplicate ACKs (Reno TCP)



TCP Congestion Avoidance

Congestion avoidance

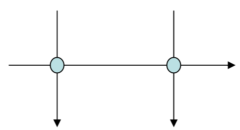
```
/* slowstart is over       */
/* Congwin > threshold */
Repeat {
  w = Congwin
  every w segments ACKed:
     Congwin++
} until (loss event)
threshold = Congwin/2
Congwin = 1
Go back to slowstart
```



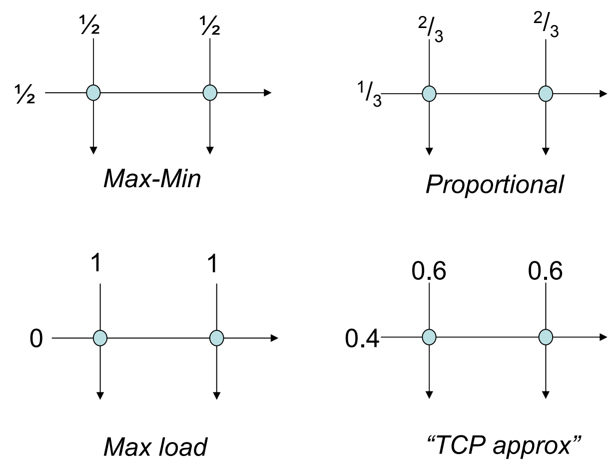
Remark: TCP Reno skips slowstart (fast recovery) after three duplicate ACKs

TCP Fairness

- A transport protocol must obey several objectives besides correctness
  - The protocol should not waste bandwidth
  - It should be fair! …?!?
  - It should be robust and not oscillate
- What is fair?
  - Two resources (routers)
  - Each with capacity normalised to 1
  - Vertical streams use one resource
  - Horizontal stream uses two resources



Various Forms of Fairness



Max-Min Fairness

- Definition
  - A set of flows is *max-min fair* if and only if no flow can be increased without decreasing a smaller or equal flow.

- How do we calculate a max-min fair distribution?
  1. Find a bottleneck resource r (router or link), that is, find a resource where the resource capacity c_r divided by the number of flows that use the resource (k_r) is minimal.
  2. Assign each flow using resource r the bandwidth c_r/k_r.
  3. Remove the k flows from the problem and reduce the capacity of the other resources they use accordingly
  4. If not finished, go back to step 1.

Is TCP Fair?

The good news

- TCP has an additive increase, multiplicative decrease (AIMD) congestion control algorithm
  - increase window by 1 per RTT, decrease window by factor of 2 on loss event
  - In some sense this is fair…
  - One can theoretically show that AIMD is efficient (→ Web Algorithms)
- TCP is definitely much fairer than UDP!

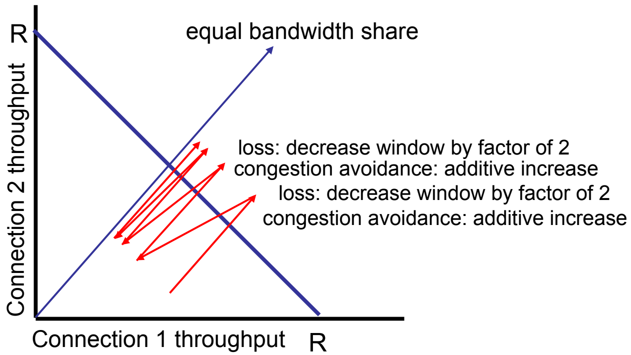
The bad news

- (even if networking books claim the opposite:) if several TCP sessions share same bottleneck link, not all get the same capacity
- What if a client opens parallel connections?

TCP fairness example

Two competing TCP sessions

- Additive increase for both sessions gives slope of 1
- Multiplicative decrease decreases throughput proportionally
- Assume that both sessions experience loss if $R_1+R_2 > R$.



TCP latency modeling (back-of-envelope analysis)

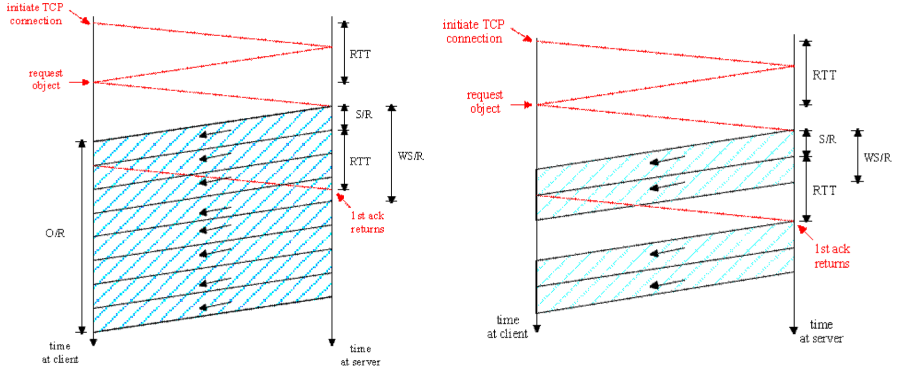
- Question: How long does it take to receive an object from a Web server after sending a request?

- TCP connection establishment
- data transfer delay

Notation & Assumptions

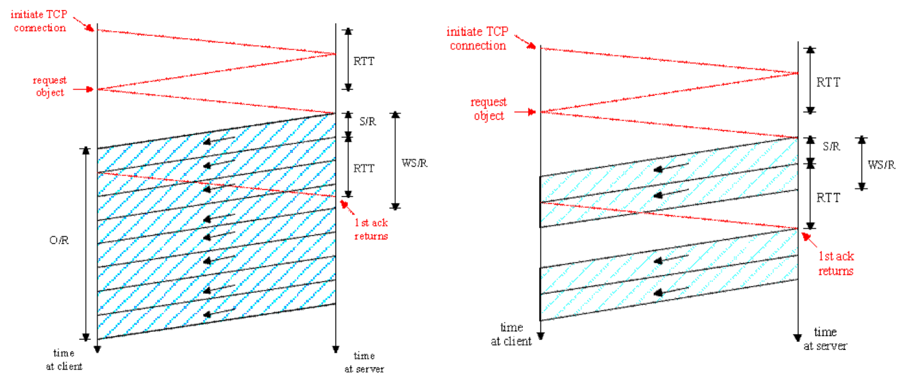
- Assume one link between client and server of rate R
- Assume: fixed congestion window with W segments
- S: MSS (bits)
- O: object size (bits)
- no retransmissions (no loss, no corruption)

TCP latency modeling: Two cases to consider



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

TCP latency modeling



latency = 2RTT + O/R

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

TCP Latency Modeling: Slow Start

Example

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