Contrôle de transmission

Bloc 4, INF 586

Walid Dabbous
INRIA Sophia Antipolis

Plan

- Contrôle d 'erreur
  - de bit (au niveau liaison)
  - de paquet (au niveau transport)

- Contrôle de flux
  - transport
  - en particulier TCP

Le contrôle d 'erreur

Error control

- Error detection & Correction
- Basic idea is to add redundancy to detect or correct errors
- Block code (n,k)
  - add n-k redundancy bits to k data bits to form n bits *codeword*
  - e.g. parity code (k+1, k) *detects* odd number of bit errors
  - rectangular code (parity along rows and column of an array) *corrects* one bit error, with coding delay
- Hamming code
  - valid codewords are « different » enough, so that errored codeword do not ressemble valid codewords
    - *distance*: minimim number of bit inversions to transform VCW1 to VCW2
    - to *detect* E errors : minimal distance is E+1
    - to *correct* E errors : minimal distance is 2E+1
- Interleaved codes
  - transmit column wise a matrix of m consecutive CWs
  - convert burst errors to « bit » errors
  - add memory cost and delay

CRC

- Right n-k bits are the remainder of dividing (n-k)-left shifted "message" by a generator polynomial G(x) of degree (n-k)
- Adequate choice of G(x) allows to detect
  - all single bit errors ($E(x)=x^i$, G has more than two terms)
  - almost all 2-bit errors ($E(x) = x^i+x^j$; G has a factor with at least three terms, chosen not to divide neither x nor $x^{max(i-j)}$)
  - any odd number of errors (E has odd number of terms and G has factor x+1)
  - all bursts up to n-k, where generator bit sequence length is n-k+1 (i.e. n-k check bits)
  - longer bursts with probability $1-2^{-(n-k)}$,if bursts are randomly distributed

Hw/Sw Implementation

- Hardware
  - on-the-fly with a shift register
  - easy to implement with Application Specific Integrated Circuit / Field Programmable Gate Array
- Software
  - Efficiency is important
  - touch each data byte only once
  - rectangular and convolutional not suitable
  - CRC

Software schemes

- TCP/UDP/IP
  - all use same scheme
  - treat data bytes as 16-bit integers
  - add with end-around carry (add 1 to the sum)
  - 16-bit one's complement of the sum = checksum
  - needs only one lookup per 16-bit block
  - catches all 1-bit errors
  - incorrectly validates (uniformly distributed) errors with probability 1/65536

Packet errors

- Different from bit errors
  - causes of packet errors
    - not just erasure, but also duplication, insertion,etc.
  - detection and correction
    - retransmission, instead of redundancy

Causes of packet errors

- Loss
  - due to uncorrectable bit errors (e.g. in wireless environment)
  - buffer loss on overflow
    - especially with bursty traffic
      - for the same load, the greater the burstiness, the more the loss
    - packet losses are bursty (correlation btw consecutive losses)
    - loss rate depends on burstiness, load, and buffer size
  - fragmented packets can lead to error multiplication (TCP>ATM)
    - packet loss rate versus cell loss rate
    - longer the packet, more the loss
      - drop the entire packet at switch

Causes of packet errors (cont.)

- Duplication
  - same packet received twice (2nd is out of sequence)
    - usually due to retransmission
- Reordering
  - packets received in wrong order
    - usually due to retransmission
    - some routing techniques may also reorder
- Insertion
  - packet from some other conversation received
    - (undetectable) header corruption from another active connection (with different TC-identifier)
    - delayed packet from closed connection (with same TC-identifier)

Packet error detection and correction

- Detection
  - Sequence numbers
  - Timeouts
- Correction
  - Retransmission
- Bit level mechanisms active
  - no errors on header

Sequence numbers

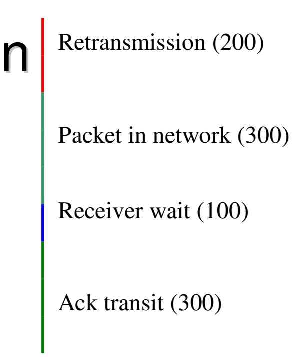
- In each header
- Incremented for non-retransmitted packets
- *Sequence space*
  - set of all possible sequence numbers
  - for a 3-bit seq #, space is {0,1,2,3,4,5,6,7}

Using sequence numbers to detect errors

- Reordering & duplication (straightforward)
- Loss
  - gap in sequence space allows *receiver* to detect loss
    - e.g. received 0,1,2,5,6,7 => lost 3,4
  - acks carry *cumulative* seq #
  - redundant information
  - if no ack for a while, *sender* suspects loss
- Insertion
  - if the received seq # is "very different" from what is expected
    - more on this later
- Two important considerations
  - choosing sequence number length
  - choosing initial sequence number

Sequence number bit length - *s*

- Long enough so that sender does not confuse sequence numbers on acks
- E.g, sending at 100 packets/sec (R)
  - sender waits for 200 secs before giving up retransmitting (T)
  - *receiver* may wait up to 100 sec (A) before sending Ack
  - packet can live in the network up to 5 minutes (300 s) (*maximum packet lifetime* or MPL)
  - can get an ack as late as 900 seconds after packet sent out
  - sent out 900*100 = 90,000 packets
  - if sequence space smaller, then can have confusion
  - so, s > log (90,000), at least 17 bits
- In general 2^s should be > R(2 MPL + T + A)



MPL

- Lower bound on *s* requires a bound on MPL
- How can we bound it?
- Generation time in header
  - additional space and computation
- Counter in header decremented per hop
  - the Time To Live (TTL)
  - crafty, but works
  - used in the Internet
  - assumes max. diameter, and a limit on forwarding time

Sequence number size (cont.)

- If no retransmissions and acks, size can be smaller: only to detect losses and reordering
- then size depends on two things
  - reordering span: how much packets can be reordered
    - e.g. span of 128 => seq # > 7 bits
  - burst loss span: how many consecutive pkts. can be lost
    - e.g. possibility of 16 consecutive lost packets => seq # > 4 bits
  - both bounds are smaller than the retrx case
  - In practice, do worst case design & hope that technology becomes obsolete before worst case hits!
- Datalink level sequence number shorter than transport
  - usually no retransmission
  - delays are smaller

Packet insertion in CO mode

- Receiver should be able to distinguish packets from other connections
- Why?
  - receive packets on VCI 1
  - connection closes
  - new connection also with VCI 1
  - delayed packet arrives
  - could be accepted
- Solution
  - flush packets on “connection closing”
  - can’t do this for connectionless networks like the Internet
    - need for more sophisticated schemes

Packet insertion (cont.)

- Packets carry source IP, dest IP, *source port number, destination port number*
- How we can have insertion?
  - host A opens connection to B, source port 2345, dest port 6789
  - transport layer connection terminates
  - new connection opens, A and B assign the same port numbers
  - delayed packet from old connection arrives with sequence number in the range used by the newer connection
  - insertion!

Solutions

- Per-connection *incarnation number*
  - incremented for each connection from each host
  - - takes up header space
  - - on a crash, incarnation numbers must be remembered
    - need stable storage, which is expensive
    - not popular in practice
- Reassign port numbers only after 1 MPL
  - remember *time* each port was assigned
  - - needs stable storage to survive crash

Solutions (cont.)

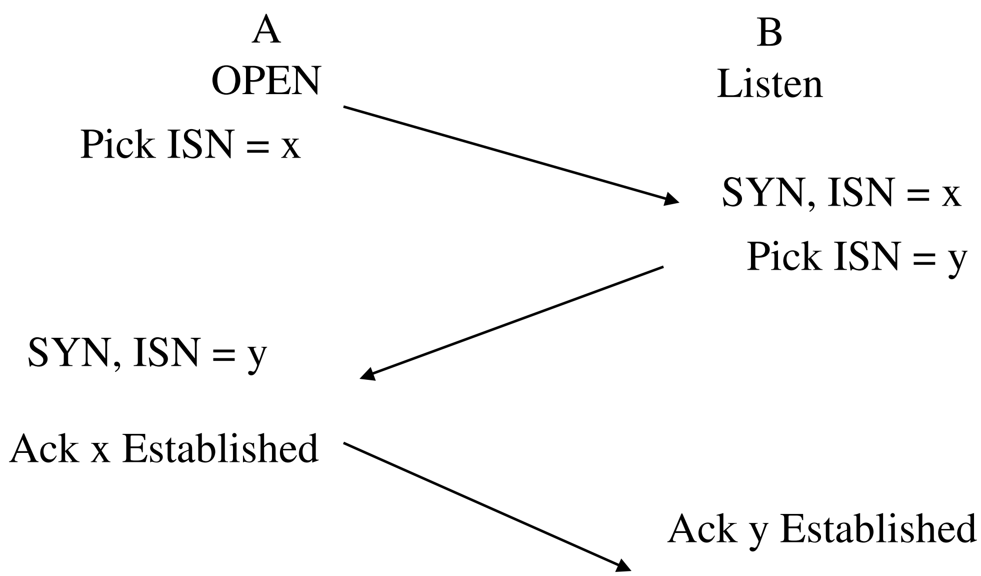
- Assign port numbers serially: new connections have new ports
  - Unix starts at 1024
  - this fails if we wrap around within 1 MPL
  - also fails if computer crashes and we restart with 1024
- chose initial sequence numbers from a clock
  - new connections may have same port, but seq # differs
  - fails on a crash
- Wait 1 MPL after boot up (30s to 2 min)
  - this flushes old packets from network
  - used in most Unix systems

Exchange of Initial Sequence Numbers

- Standard solution, then, is
  - choose port numbers serially (unless specified by user)
  - choose initial sequence numbers from a clock
  - wait 1 MPL after a crash
- Needs communicating ends to tell each other initial sequence number
- Easiest way is to tell this in a *SYNchronize* packet (TCP) that starts a connection
- 2-way handshake
  - does not protect against delayed SYN packets

3-way handshake

- Problem really is that SYNs themselves are not protected with sequence numbers
- 3-way handshake protects against delayed SYNs



Loss detection

- At receiver, from a gap in sequence space
  - send a *nack* to the sender
- At sender, by looking at cumulative acks, and timing out if no ack for a while
  - need to choose timeout interval

Nacks

- Sounds good, but does not work well
  - extra load during loss, even though in reverse direction
- If nack is lost, receiver must retransmit it
  - moves timeout problem to receiver
- So we need timeouts anyway

Timeouts

- Set timer on sending a packet
- If timer goes off, and no ack, resend
- How to choose timeout value?
- Intuition is that we expect a reply in about one round trip time (RTT)

Timeout schemes

- Static scheme
  - know RTT *a priori*
  - timer set to this value
  - works well when RTT changes little (special purpose systems)
- Dynamic scheme
  - measure RTT
  - timeout is a function of measured RTTs
    - larger than RTT to deal with delay variation

Old TCP scheme

- RTTs are measured periodically
- Smoothed RTT (*srtt*)
- *srtt(i) = a * srtt(i-1) + (1-a) * RTT(i)*
- timeout = *b * srtt*
- a = 0.9, b = 2
- sensitive to choice of a
  - a = 1 => timeout = 2 * initial srtt
  - a = 0 => no history
- doesn't work too well in practice

New TCP scheme (Jacobson)

- introduce new error term = *m*
- its smoothed estimate *sm* : mean deviation from mean
- $m(i) = | srtt(i) - RTT(i) |$
- $sm(i) = a * sm(i-1) + (1-a) * m(i)$
- $timeout = srtt + b * sm$
- Different values of *b* give different confidence intervals

Intrinsic problems

- Hard to choose proper timers, even with new TCP scheme
  - What should initial value of srtt be?
    - Particularly hard if a is close to 1 (strong memory)
  - Measuring RTT is hard in presence of losses
    - Ack may acknowledge more than one packet -> hard to determine the packet to derive RTT from
  - Timeout => loss, delayed ack, or lost ack
    - hard to distinguish
- Lesson: use timeouts rarely

Retransmissions

- Sender detects loss on timeout
  - or other "signal"
- Which packets to retransmit?
- Need to first understand concept of error control window

Error control window

- Set of packets sent, but not acked
- 1 2 3 4 5 6 7 8 9 (original window)
- 1 2 3 4 5 6 7 8 9 (recv ack for 3)
- 1 2 3 4 5 6 7 8 9 (send 8)
- May want to restrict max size = window size
- Sender blocked until ack comes back

Go back N retransmission

- On a timeout, retransmit the entire error control window
- Receiver only accepts in-order packets
- + simple
- +conservative: on loss signal, retrx every possible lost packet
- + no buffer at receiver
- - can add to congestion
- - wastes bandwidth
- p the packet loss probability, W the window
  - efficiency = $(1-p)/(1-p+p.W)$
  - low efficiency for high W and/or p
- used in TCP

Selective retransmission

- Somehow find out which packets lost, then only retransmit them
- How to find lost packets?
  - each ack has a bitmap of received packets
    - e.g. cum_ack = 5, bitmap = 101 => received 5 and 7, but not 6
    - wastes header space
  - sender may therefore periodically ask receiver for bitmap
  - or do fast retransmit (guess that a loss occured)
- requires more complex procedures at both sender and receivers
  - and requires to buffer **W**-1 packets

Fast retransmit

- Assume cumulative acks
- If sender sees repeated cumulative acks, packet likely lost
- 1, 2, 3, 4, 5 , 6
- 1, 2, 3   3   3
- Send cumulative_ack + 1 = 4
- Used in TCP
- Provides partial “selective” information for free
- does not work well in case of multiple error within a window

SMART

- Ack carries cumulative sequence number
- Also sequence number of packet causing ack
- 1 2 3 4 5 6 7 8 9 10
- 1 2 3 4 5   5 5   5
- 1 2 3 4 5 x 7 8 x 10
- (5,5) (5,7) (5,8) (5,10)
- Sender creates bitmap
- Does not use timers
- Not effective if retransmitted packet lost,
  - sender periodically check if cumulative ack increased, and retx N+1
  - on worst case, retrx entire window as in go-back-N

FEC

- Forward Error Correction can also be performed at packet level
- Sends « parity check » packets
- does not require retransmission
- adequate for real time application
  - audio/video conferencing
- But increases load and error rate!
- Not effective if « burst » packet losses
- increases end to end delay
  - wait for entire FEC block before processing

Le contrôle de flux

Flow control problem

- Consider file transfer
- Sender sends a stream of packets representing fragments of a file
- Sender should try to match rate at which receiver and network can process data
- Can't send too slow or too fast
- Too slow
  - wastes time
- Too fast
  - can lead to buffer overflow
- Main objective of flow control
  - how to find the correct rate?

Other considerations

- Simplicity
- Low overhead
  - use of network (bandwidth and buffers) resources
- Scaling to many sources
- Fairness
  - if scarcity of resources, each source gets its “fair” share
- Stability
  - for fixed number of sources, transmission rate for each source settles down to an equilibrium value
- Many interesting tradeoffs
  - low overhead for stability
  - simplicity for fairness

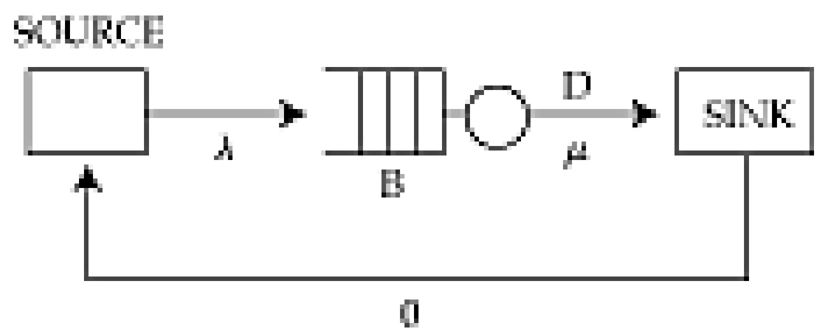
Where?

- Can be at
  - application level
  - transport
  - network
  - link
- At transport layer for end2end flow control
- At datalink layer for hop by hop flow control

- Terminology
  - Flow control vs congestion control
  - congestion is overload of *intermediate* network elements

Model

- Source sending at λ packet/s, sink acks every packet, intermediate servers, (variable) service rate μ packet/s (allocated or available), bottleneck is the slowest server, buffer size at bottleneck B, round trip time (D)



- Flow control: rate-matching with delays
- For flow control purpose: Ignore all but the bottleneck server

Classification

- Open loop
  - Source describes its desired flow rate
  - Network *admits* call and *reserves* resources
  - Source sends at this rate
- Closed loop
  - Source monitors available service rate
    - Explicit or implicit feedback
  - Sends at this rate
  - Due to speed of light delay, errors are bound to occur
- Hybrid
  - Source asks for some minimum rate
  - But can send more, if available

Open loop flow control

- Two phases to flow control, during:
  - Call setup
  - Data transmission
- Call setup
  - Network prescribes traffic descriptor parameters
  - User chooses parameter values
  - Network admits (may negotiate) or denies call
    - if OK, bandwidth and buffers are reserved
- Data transmission
  - User shapes its traffic within parameter range
  - Network *polices* users
  - Scheduling policies give user QoS

Hard problems

- Choosing a descriptor at a source
  - capture future behavior in a set of parameters
- Choosing a scheduling discipline at intermediate network elements (see block 7 - scheduling)
- Admitting calls so that their performance objectives are met (*call admission control*) (not studied in this course, chap 14 in Keshav's book).
- Or just ignore :-)

Traffic descriptors

- Set of parameters that describes behavior of a data source
- It is typically a behavior *envelope*
  - Describes in fact worst case behavior
- Three uses besides describing source behavior
  - Basis for traffic contract
    - if not violated by source, network “guarantees” QoS
  - Input to *regulator*
    - where source delays traffic
  - Input to *policer*
    - where operator delays or drops excess traffic

Descriptor requirements

- Representativity
  - adequately describes flow, so that network does not reserve too little or too much resource
- Verifiability
  - network able easily to verify that descriptor holds
- Usability
  - Easy to describe and use for admission control

Examples

- Representative, verifiable, but not useable
  - Time series of interarrival times
    - potentially very long and unknown for interactive sources
    - network may add jitter
- Verifiable, and useable, but not representative
  - peak rate
    - may send at less than peak rate -> waste resources

Some common descriptors

- Peak rate
- Average rate
- Linear bounded arrival process

- will study each with the corresponding regulator

Peak rate

- Highest ‘rate’ at which a source can send data
  - trivial bound: the link capacity but does not give a true picture
- Two ways to compute it
- For networks with fixed-size packets
  - (min inter-packet spacing)$^{-1}$
- For networks with variable-size packets, time window *t*
  - bounds total data generated over *all* intervals of duration *t*
- Regulator for fixed-size packets: buffer +
  - timer set on packet transmission to min inter-packet spacing
  - if timer expires, send buffered packet, if any
- Problem
  - sensitive to extremes: a single “drift” may result in a radical change

Average rate

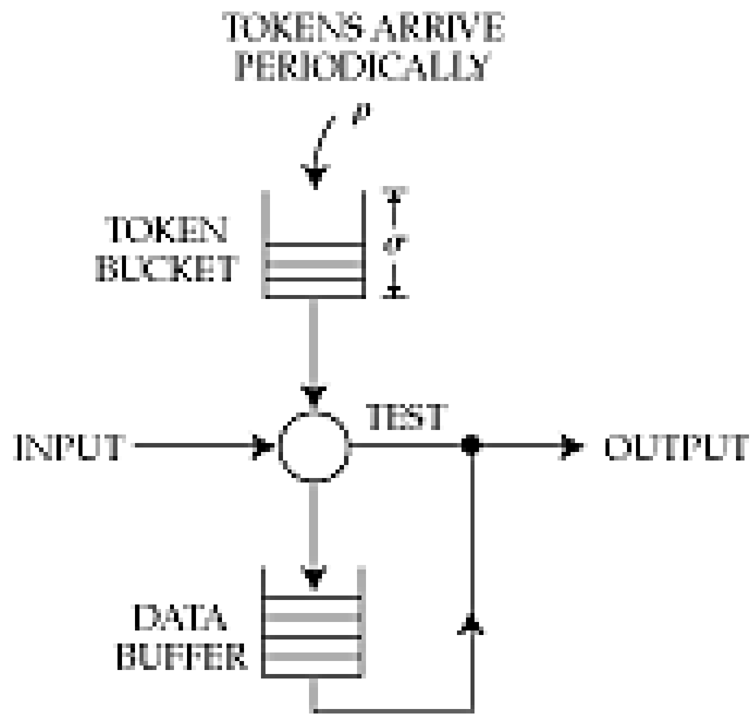
- Measure rate over some time period (*window*) '*t*'
- Less susceptible to outliers
- Parameters: *t* and *a* (number of bits to send during *t*)
- Two types: jumping window and moving window
- Jumping window
  - over consecutive intervals of length *t*, only *a* bits sent
  - sensitive to the choice of the starting time of 1^{st} window
  - regulator *reinitializes* every interval
- Moving window
  - over all intervals of length *t*, only *a* bits sent
  - regulator *forgets* packets sent more than *t* seconds ago
  - removes dependency on starting time

Linear Bounded Arrival Process

- Source bounds # bits sent in any time interval by a linear function of time
- the number of bits transmitted in any active interval of length t is less than or equal to $\rho . t + \sigma$
- ρ is the long term rate *allocated* by network to source
- σ is the burst limit (max burst a source may send)
- a generalization of average rate descriptor
  - also insensitive to outliers

Leaky bucket

- A regulator for an LBAP
- Token bucket fills up at rate ρ
- Largest # tokens = σ



Leaky bucket regulator

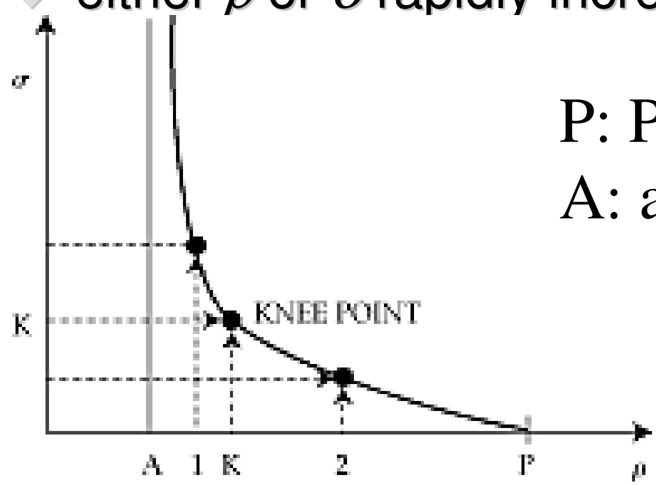
- Leaky bucket can be used as both:
  - a peak rate regulator (ρ = peak rate, σ = 1)
  - or a moving-window average regulator (ρ = average rate)
- Variant
  - Token bucket + peak rate regulator
    - allows to control: average rate, peak rate and max burst
- Has both token and data buckets
  - Sum of sizes is what matters
  - a larger token bucket offsets a smaller data buffer

Choosing LBAP parameters

- How to choose ρ and σ (e.g. for a stored video source)
- Minimal descriptor
  - no other descriptor has both a smaller ρ and a smaller σ
  - presumably costs less
- How to choose minimal descriptor?
  - Not unique
  - tradeoff between ρ and σ
    - for given size of data buffer and max loss rate, for each ρ there is a min σ so that loss rate is met
- Three way tradeoff
  - choice of σ (data bucket size)
  - loss rate
  - choice of ρ

Choosing minimal parameters

- Keeping loss rate the same
  - if σ is more, ρ is less (smoothing)
  - for each ρ in the [A,P] range, we have minimum σ
- For "common" sources choose knee of curve (K)
  - either ρ or σ rapidly increases when moving away from knee



P: Peak rate

A: average rate over a long interval

LBAP

- “Popular” in practice (ATM) and in academia
  - verifiable
  - sort of usable

- BUT do not accurately represent sources with large bursts
  - otherwise σ would be too large
  - makes network expensive
  - what about renegotiating ρ before bursts
    - possible for stored video!
    - Or just after the start of a burst in the case of long bursts
      - buffer still fills while renegotiation

Open loop vs. closed loop

- Open loop
  - describe traffic
  - network admits/reserves resources
  - regulation/policing
- Closed loop
  - can't *describe* traffic or
  - network doesn't support *reservation*
    - resources are overbooked for higher multiplexing gain (SMG)
  - source monitors *available bandwidth*
    - perhaps allocated using GPS-emulation in routers
  - *adapts* to it in order not to overload network
  - if not done properly either
    - excessive packet loss (higher "rate" than bottleneck)
    - underutilize network resources (much slower than bottleneck)

Taxonomy

- First generation (on-off, stop-and-wait, static-window)
  - ignores network state
  - only match receiver
- Second generation
  - responsive to both sink and network states
  - three choices
    - State measurement
      - explicit or implicit
    - Control
      - flow control window size or rate
    - Point of control
      - endpoint or within network

Explicit vs. Implicit

- Explicit
  - Network tells source its current rate
  - Better control
  - More communication and computation overhead
- Implicit (only in end to end schemes)
  - Endpoint figures out rate by looking at network
  - Less overhead
- Ideally, want overhead of implicit with effectiveness of explicit

Flow control window

- Recall error control window
  - Largest number of packet outstanding (sent but not acked)
- If endpoint has sent all packets in window, it must wait => slows down its rate
- Thus, window provides *both* error control and flow control
- Flow control window is also called *transmission* window
- indirectly control a source' rate by modifying transmission window
- but this coupling of error and flow control can be a problem
  - Few buffers at receiver => small window (if selective repeat) => slow rate!

Adaptive window or adaptive rate

- In adaptive rate, we directly control rate
- Needs a fine grain timer per connection
  - set after a packet trx to inverse of trx rate
- Plusses for window
  - easier to implement: no need for fine-grained timer
  - self-limiting
- Plusses for rate
  - better control (finer grain)
  - no coupling of flow control and error control
- Rate control must be carefully engineered to avoid overhead and sending too much (in case of loss of rate limiting packet)

Hop-by-hop vs. end-to-end

- Hop-by-hop
  - make first generation flow responsive to network state
    - control at each link, next server = sink
  - easy to implement
- End-to-end
  - sender matches all the servers on its path
- Plusses for hop-by-hop
  - simpler mechanisms
  - better control
  - distributes buffer usage
- Plusses for end-to-end
  - cheaper, does not require complexity in routers

Closed loop flow control schemes

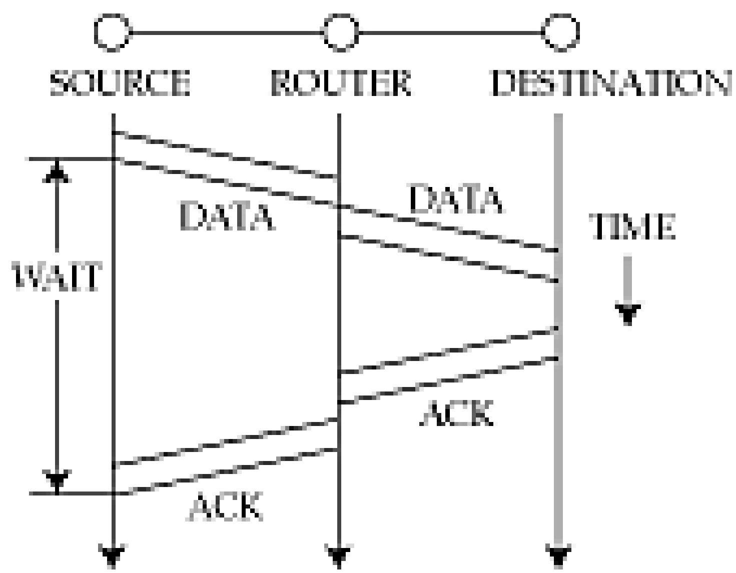
	Explicit		Implicit	
	Dynamic window	Dynamic rate	Dynamic window	Dynamic rate
End2end	DECbit	ATM EERC	TCP	NetBLT, pp
Hop-by-hop	Credit-based	Mishra/Kanakia	_	_

On-off flow control

- Receiver gives ON and OFF signals
- If ON, send at full speed
- If OFF, stop
- OK when RTT is small
- What if OFF is lost?
- Generates bursty traffic
  - packet losses in intermediate elements
- Used in serial lines or LANs
  - delays are small
  - packet loss rare
  - basis of the XON/XOFF protocol used to control serial I/O devices (printers, mice)

Stop and Wait

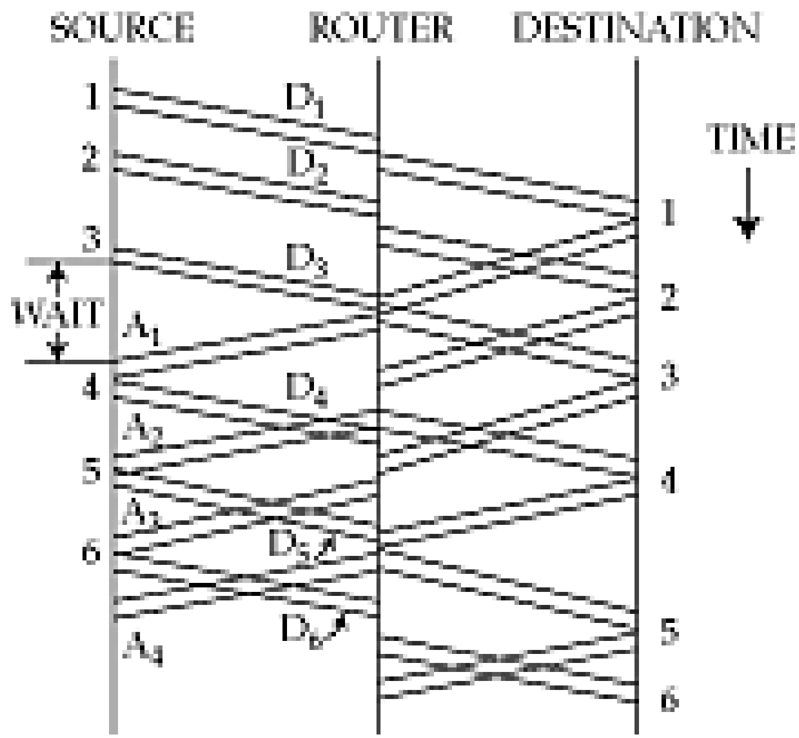
- Send a *single* packet
- Wait for ack before sending next packet
- provides error and flow control
- inefficient if delay is large
- Max throughput
  - 1 packet per RTT



Static window

- Stop and wait can send at most one pkt per RTT
- Here, we allow multiple packets per RTT (w = transmission window)

$w = 3$



What should window size be?

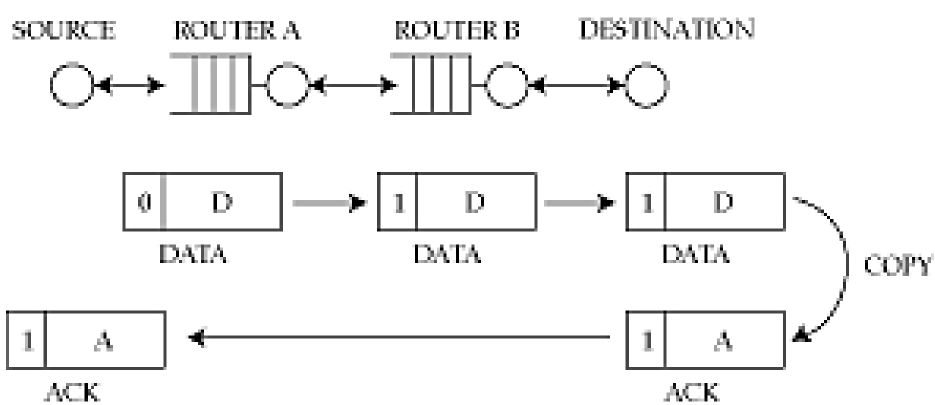
- Let bottleneck service rate along path = μ pkts/sec
- Let round trip time = R sec
- Let flow control window = *w* packet
- Sending rate is *w* packets in R seconds = w/R packets/s
- To keep bottleneck fully utilized
  - $w/R > \mu \Rightarrow w > R\mu$
- This is the *bandwidth delay product* or *optimal window size*

Static window

- Works well if μ and R are fixed
- Even for a specific bottleneck, the rate changes with time!
- Static choice of *w* can lead to problems
  - too small
    - bottleneck underutilized
  - too large
    - w - $R\mu$ packets buffered at bottleneck
- So, need to adapt window
- Always try to get to the *current* optimal value

DECbit flow control (dynamic window)

- Intuition
  - every packet has a bit in header
  - intermediate routers set bit if queue has built up => source window is too large
  - sink copies bit to ack
  - if bits set, source reduces window size
  - in steady state, oscillate around optimal size



DECbit evaluation

- Only 1 bit is required
- does not require per-connection *queuing* at routers
- can adapt and oscillates around stable optimal window value
  - (Additive Increase Multiplicative Decrease policy)

- Requires per-connection router *actions*
- Increase policy is conservative
  - increase by 1 every two RTTs
  - bad performance on eLePHaNts (Long and Fat pipe Networks)

TCP Flow Control

- Implicit
- Dynamic window
- End-to-end

- Very similar to DECbit, but
  - no support from routers
  - increase if no loss (usually detected using timeout)
  - window decrease on a timeout (or 3 duplicate acks)
  - additive increase multiplicative decrease

TCP details

- Window starts at 1
- Increases exponentially for a while, then linearly
- Exponentially => doubles every RTT
- Linearly => increases by 1 every RTT
- During exponential phase, every ack results in window increase by 1
- During linear phase, window increases by 1 when # acks = window size
- Exponential phase is called *slow start*
- Linear phase is called *congestion avoidance*

More TCP details

- On a loss, current window size is stored in a variable called *slow start threshold* or *ssthresh*
- Switch from exponential to linear (slow start to congestion avoidance) when window size reaches threshold
- Loss detected either with timeout or duplicate cumulative acks (*fast retransmit*)
- Two (early) versions of TCP
  - Tahoe: in both cases, drop window to 1
  - Reno: on timeout, drop window to 1, and on fast retransmit drop window to half previous size (also, do *fast recovery*: increase window by 1 for each duplicate ack, until new data acked)

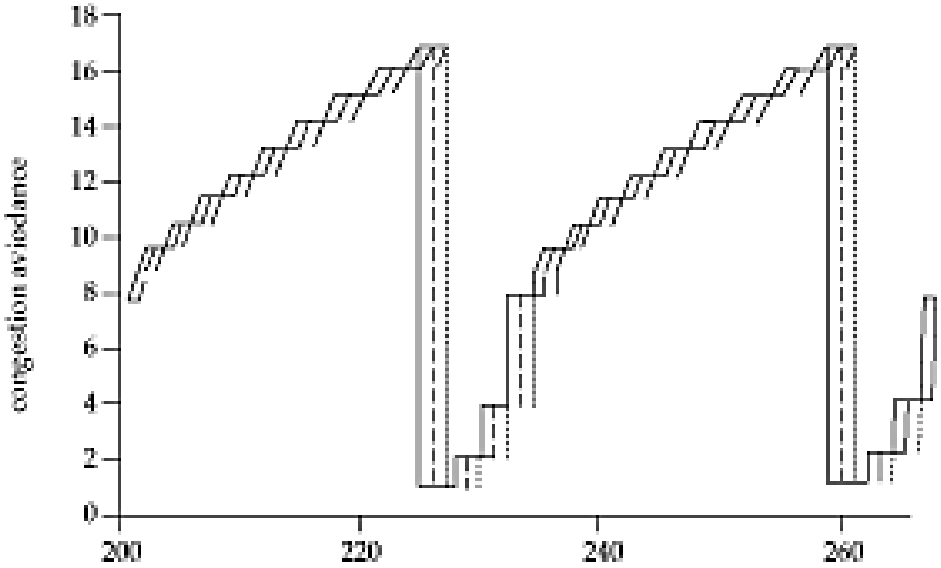
TCP vs. DECbit

- Both use dynamic window flow control and (stable) additive-increase multiplicative decrease policy
- TCP uses implicit measurement of congestion
  - probe a “black box”
- TCP source does not filter information
  - each packet loss indicates congestion
  - necessary because network operated at the *cliff* (close to overload)

Evaluation

- Effective over a wide range of bandwidths
- A lot of operational experience
- Weaknesses
  - loss => overload? (wireless)
    - link level retrx or FEC to make wireless link appear loss-free
    - link level "informs" TCP of link losses
  - loss => self-blame, problem with malicious users on FCFS
  - overload detected only on a loss
    - in steady state, source *induces* loss
  - sensitive to choice of ssthresh for short transfers
    - if large can lead to *multiple* packet losses, FastRTx will not help
  - needs per connection buffering at bottleneck

Sample trace



TCP Vegas

- Source computes Expected throughput = transmission_window_size/propagation_delay
- Numerator: known
- Denominator: measure *smallest* RTT
- Also know *actual* throughput
- Difference = how much to reduce/increase rate
- Algorithm
  - send a special packet
  - on ack, compute expected and actual throughput
  - if expected < actual, adjust propagation_delay
  - (expected - actual)* RTT packets are still in bottleneck buffer
  - adjust sending rate if this is out of L&H watermarks
- "performs better" than TCP Reno
  - but rate based and not TCP reno-fair

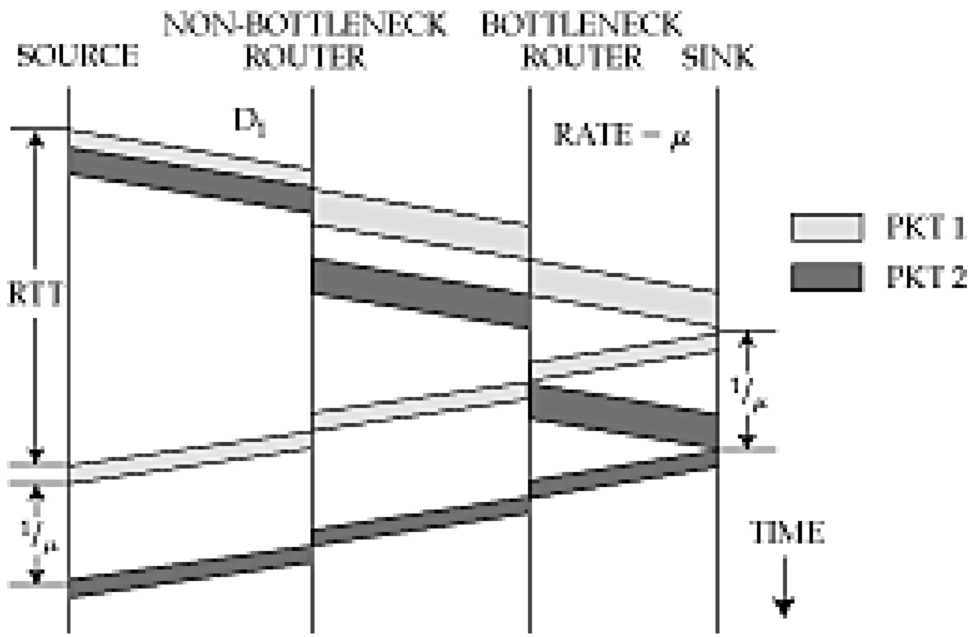
NETBLT

- “First” rate-based flow control scheme
- Separates error control (window) and flow control (no *coupling*)
- So, losses and retransmissions do not affect the flow rate
- Application data sent as a series of buffers, each at a particular rate
- Rate expressed as a burst size and a burst rate
  - so granularity of rate control = burst
- In the original scheme, no rate adjustment
- Later, if received rate < sending rate, multiplicatively decrease rate, otherwise linearly increase
- Change rate only once per buffer => slow

Packet pair

- Improves basic ideas in NETBLT
  - better measurement of bottleneck
  - control based on prediction
  - finer granularity
- Assume all bottlenecks serve packets in *round robin* order
- Then, spacing between 2 packets of same connection at receiver (= ack spacing) = 1/(rate of slowest server)
- If *all* data sent as paired packets, no distinction between data and probes
- Implicitly determine service rates if routers are round-robin-like

Packet pair



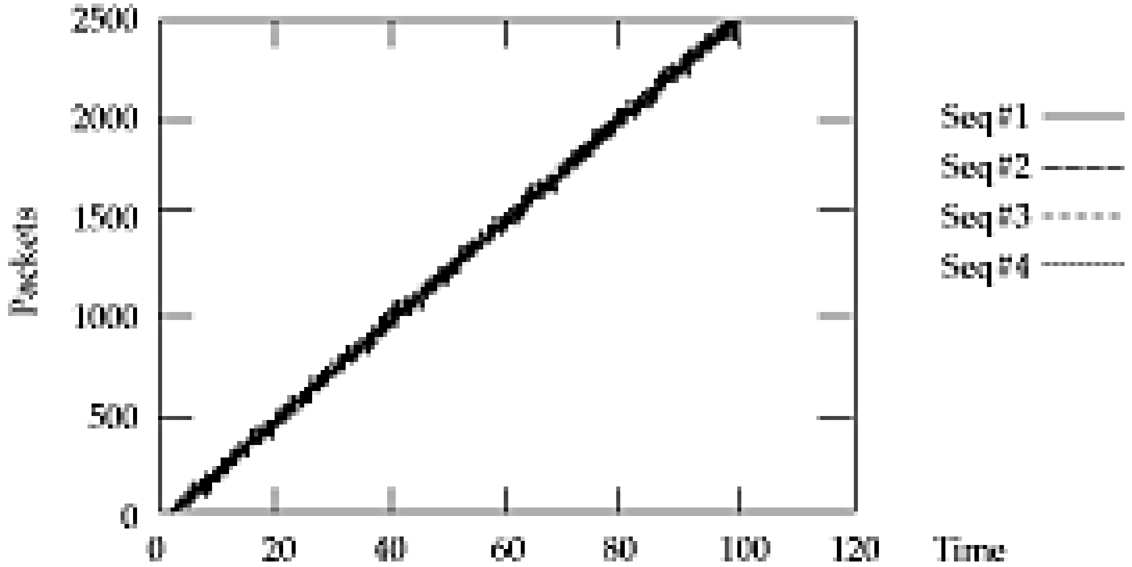
SOURCE
NON-BOTTLENECK ROUTER
BOTTLENECK ROUTER
SINK
D_1
RATE = μ
RTT
$1/\mu$
$1/\mu$
PKT 1
PKT 2
TIME

Packet-pair details

- Acks give time series of service rates in the past
- We can use this to predict the next rate

- but requires round robin in routers!

Sample trace



Four sources sharing a bottlenck

Comparison among closed-loop schemes

- On-off, stop-and-wait, static window, DECbit, TCP, NETBLT, Packet-pair, ATM Forum EERC (End2End Rate based flow Control)
- Which is best? No simple answer
- Some rules of thumb
  - flow control easier with Round Robin scheduling
    - otherwise, assume cooperation, or police allocated rates
  - explicit schemes are more robust
  - hop-by-hop schemes are more responsive, but more complex
  - try to separate error control and flow control
  - rate based schemes are inherently unstable unless well-engineered