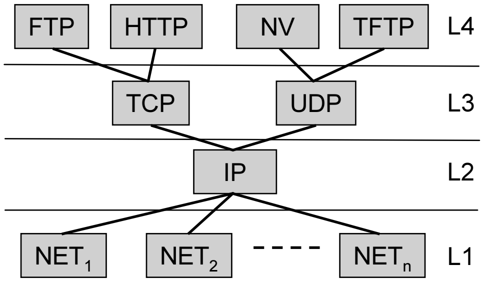
NS2: Contents

- NS2 – Introduction to NS2 simulator
- Some NS2 examples
- NS2 project work instructions

Internet and TCP

- Internet (currently) offers only best effort service
  - packets are delayed
  - packets are lost
  - packets are misordered



- TCP: end-to-end reliable byte stream
  - window based flow control
    - each received packet is acknowledged
    - lost packets are retransmitted
  - window size, w, defines an upper bound on number of unacknowledged packets
    - during one round trip time, RTT, at most w packets can be sent
    - thus, sending rate ~ w/RTT

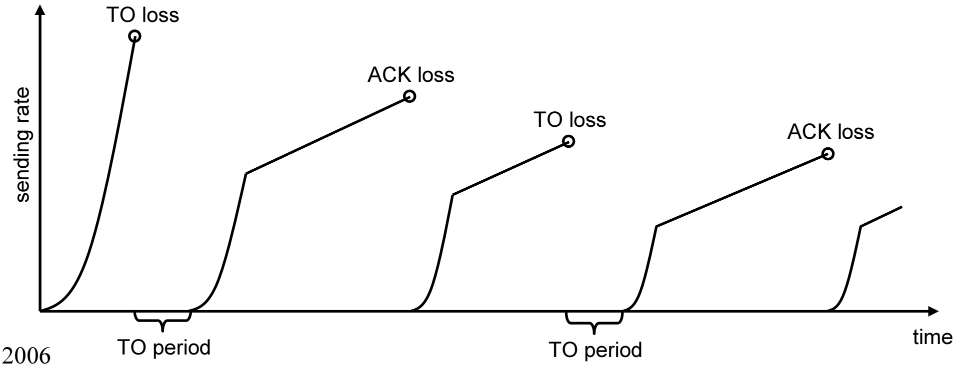
Internet congestion control

- Original TCP
  - sender starts sending immediately with max window size that receiver's buffers allow
  - works as long as network only lightly loaded (users not able to overload network)
  - early 1980's: series of "congestion collapses"
    - during overload network is only carrying retransmitted packets and (almost) no fresh offered traffic ⇒ need for congestion control
- TCP congestion control principles
  - idea: modify window size adaptively based on "available capacity"
    - assumption: packet losses caused by congested buffers (not bit errors)
  - TCP is an adaptive system with feedback in form of packet losses
    - losses interpreted as indications of congestion and are detected through timeouts (slow response) and so called duplicate ACKs
    - delayed feedback due to RTTs
  - congestion control implemented by following algorithms
    - slow start, additive increase-multiplicative decrease (AIMD), fast retransmit, fast recovery

---

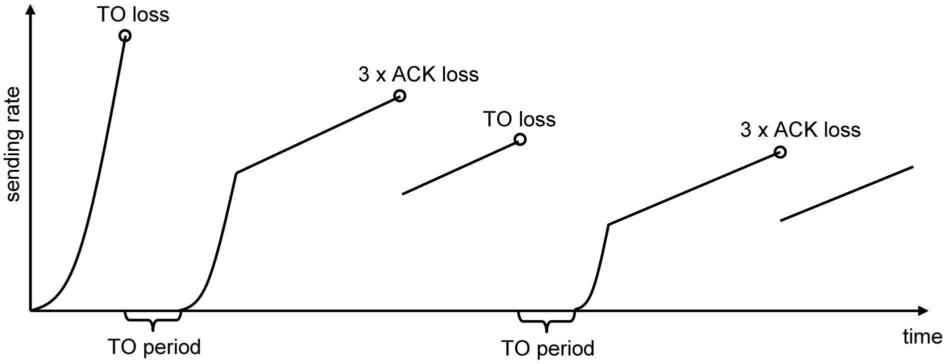
TCP Tahoe

- Slow start
  - window increased exponentially until packet loss occurs (loss event means that network capacity has been reached) or to reach congestion avoidance threshold
- AIMD
  - after reaching threshold (window size just before loss/2) switch to linear increase (congestion avoidance)
- Fast retransmit
  - detect loss from duplicate ACKs, eliminates TO periods



TCP Reno

- Fast recovery
  - assume large window sizes and a large bandwidth-delay product
  - if one packet is lost, other ACKs are still received ⇒ use these to resend lost packet (fast recovery) and new packets
  - after loss, start directly from AIMD threshold, i.e., w/2 (multiplicative decrease), and continue with linear increase (AIMD, congestion avoidance)
  - ⇒ eliminates slow starts for duplicate ACK losses



TCP performance: greedy flows

- TCP throughput influenced by packet loss and RTT, but how?
- Simple models:
  - Floyd's deterministic model
    - window grows linearly from w/2 to w and after reaching w, packet is lost

$$\Rightarrow \quad \frac{w}{2} + (\frac{w}{2}+1) + \cdots + w \approx \frac{3}{8}w^2 \quad \text{packets sent / lost packet}$$

$$\Rightarrow \quad p = \frac{8}{3w^2} \quad \Rightarrow \quad rate = \frac{w}{RTT} = \sqrt{\frac{8}{3}} \cdot \frac{1}{RTT \cdot \sqrt{p}}$$

  - Doing the analysis more carefully ⇒ Padhye's equation

$$T(p) \approx \min\left(\frac{W_{\max}}{RTT}, \frac{1}{RTT\sqrt{\frac{2bp}{3}} + T_0 \min\left(1,3\sqrt{\frac{3bp}{8}}\right) p\left(1+32p^2\right)}\right)$$

    - Includes impact of timeouts

TCP performance: flow level model (1)

- In reality TCP flows come and go randomly…
- DPS (Discriminatory Processor Sharing)
  - consider a processor sharing system where we have *M* classes of jobs
  - class-*k* jobs arrive according to a Poisson process with rate λ_k
  - class-*k* jobs require an exponentially distributed amount of time with mean $1/\mu_k$
  - class-*k* jobs have a weight g_k and jobs share the processor in a weighted manner such that the fraction of the processor allocated to class-*k* jobs equals

$$\frac{g_k}{\sum_{j=1}^{M} g_j N_j}$$

  - then the mean class-*k* delay can be solved from the system of linear equations

$$W_k \left[1 - \sum_{j=1}^{M} \frac{\lambda_j}{\mu_j + \mu_k \frac{g_k}{g_j}} \right] - \sum_{j=1}^{M} \frac{\lambda_j W_j}{\mu_j + \mu_k \frac{g_k}{g_j}} = \frac{1}{\mu_k}, \quad k = 1,2,\ldots,M$$

TCP performance: flow level model (2)

- Assuming that throughput of a TCP flow in class *k* can be approximated by $c/\left(RTT_k * \sqrt{p}\right)$, the ratio g_i/g_j becomes

$$\frac{g_i}{g_j} = \frac{RTT_j}{RTT_i}$$

- Observe that for a given TCP sender, the RTTs are random
  - simplest approximation for class-*k* RTT is to assume it consists of only the propagation delays (remember that RTT means by definition the total delay in both directions)
  - this is more accurate the less the random queuing delays impact the RTT
- Other parameters
  - flow arrival rate equals λ_k and the parameter $1/\mu_k$ equals *B*/*C*, where *B* is the mean file size (file sizes are assumed to be exponentially distributed) and *C* is the bottleneck bw
- Throughput of a class-*k* flow, denoted by T_k, is by definition the mean file size divided by the average class-*k* transfer time, i.e.,

$$T_k = \frac{B}{W_k}$$

The ns2 assignment

- We perform flow level simulations of TCP
  - event scheduling handled from Otcl level
  - scheduling concerns arrival and departure of flows
  - a skeleton code for handling this is given
- Your task is to…
  - create the topology,
  - implement the main program for controlling the simulation,
  - implement the final computation of performance statistics
- We consider two scenarios
  - Task1 & Task2

Some hints for programming (1)

- Creating an array of TCPs
  - you can create an array in TCL without declaring it first
  - example: creating 10 TCPs, configuring them and storing them in the array tcp()

```
for {set nn 0} {$nn < 10} {incr nn} {
  set tcp_s($nn) [new Agent/TCP/Reno]
  $tcp_s($nn) set packetSize_ 1460
  $tcp_s($nn) set window_ 1000
  $tcp_s($nn) set fid_ $nn
  . . .
}
```

  - multidimensional arrays: for example, `$tcp_s(2,3)` = tcp-agent in class 2 and id 3

Some hints for programming (2)

- Accessing lists
  - lists can be initialized easily
  - operations for lists:
    - `llength` : length of the list
    - `lindex` : pick element at given index from the list
    - `lappend` : insert element
    - `lreplace` : search and replace
  - Example:

```
set a {1 2 3 4}
set b [lindex $a 1] (=> b = 2, indexing starts from 0)
lappend $a 5 (=> a = {1 2 3 4 5})
```