Flow Isolation

Matt Mathis
ICCRG at IETF 77
3/23/2010
Anaheim CA

http://staff.psc.edu/mathis/papers
FlowIsolation20100323.{pdf,odp}

The origin of “TCP friendly”

$$Rate = \left(\frac{MSS}{RTT}\right)\left(\frac{0.7}{\sqrt{p}}\right)$$

[1997]

- Inspired “TCP Friendly Rate Control”
  - [Mahdavi&Floyd '97]
  - Defined the language
- Became the IETF dogma

The concept was not at all new

- 10 years earlier it had been assumed that:
  - Gateways (routers&switches) are simple
    - Send the same signals (loss, delay) to all flows
  - End-systems are more complicated
    - Equivalent response to congestion signals
    - Which was defined by Van's TCP (BSD, 1987)
    - Pushed BSD as a reference implementation
- This is the Internet's “sharing architecture”

Today TCP Friendly is failing

- Prior to modern stacks
  - End-system bottlenecks limited load in the core
  - ISPs could out build the load
  - No sustained congestion in the core
  - Masked weaknesses in the TCP friendly paradigm
- Modern stacks
  - May be more than 2 orders of magnitude faster
  - Nearly always cause congestion

Old TCP stacks were lame

- Fixed size Receive Socket Buffer
  - 8kb, 16kB and 32kB are typical
    - One buffer of data for each RTT
    - 250 kB/s or 2 Mb/s on continental scale paths
  - Some users were bottlenecked at the access link
    - AIMD works well with the large buffer routers
  - Other users were bottlenecked by the end-system
    - Mostly due to socket buffer sizes
  - The core only rarely exercised AIMD

Modern Stacks

- Both sender and receiver side TCP autotuning
  - Dynamically adjust socket buffers
  - Multiple Mbyte maximum window size
- Every flow with enough data:
  - Raises the network RTT and/or
  - Raises the loss rate
  - e.g. causes some congestion somewhere
- Linux as of 2.6.17 (~Aug 2004)
  - Ported from Web100
  - Now: Windows 7, Vista, MacOS, *BSD

Problems

- Classic TCP is window fair
  - Short RTT flows clobber all others
- Some apps present infinite demand
  - ISPs can't out build the load
- TCP's design goal is to cause congestion
  - Meaning queues and loss everywhere
- Many things run much faster
  - But extremely unpredictable performance
  - Some users are much less happy
- See backup slides (Appendix)

Change the assumption

- Network controls the traffic
  - Segregate the traffic by flow
  - With a separate (virtual) queue for each
  - Use a scheduler to allocate capacity
  - Don't allow flows to (significantly) interact
  - Separate AQM per flow
    - Different flows see different congestion

This is not at all new

- Many papers on Fair Queuing&variants
  - Entire SIGCOMM sessions
- The killer is the scaling problem associated with per flow state

Approximate Fair (Dropping)

- Follows from Pan et al CCR April 2003
- Good scaling properties
  - Shadow buffer samples forwarded traffic
  - On each packet
    - Hardware TCAM counts matching packets
      - Estimates flow rates
    - Estimates virtual queue length
      - Very accurate for high rate flows
    - Implements rate control and AQM
      - Per virtual queue

Flow Isolation

- Flows don't interact with each other
  - Only interact w/ scheduler and AQM
- TCP doesn't (can't) determine rate
- TCP's role is simplified
  - Just maintain a queue
  - Control against AQM
  - Details are (mostly) not important

The scheduler allocates capacity

- Should use many inputs
  - DSCP codepoint
  - Traffic volume
    - See: draft-livingood-woundy-congestion-mgmt-03.txt
  - Local congestion volume
  - Downstream congestion volume (Re-Feedback)
- Lots of possible ICCRG work here

Cool Properties

- More predictable performance
- Can monitor SLAs
  - Instrument scheduler parameters
- Does not depend on CC details
  - Aggressive protocols don't hurt
- Natural evolution from current state
  - Creeping transport aggressiveness
  - ISP defenses against creeping aggressiveness

How aggressive is ok?

- Discarding traffic at line rate is easy
- Need to avoid congestive collapse
  - Want goodput=bottleneck BW
- Must consider cascaded bottlenecks
  - Don't want traffic that consumes resources at one bottleneck to be discarded at another
  - Sending data without regard to loss is very bad
- But how much loss is ok?

Conjecture

- Average loss rate less than 1 per RTT is ok
  - Some RTTs are lossless, so the window fits within the pipe
  - Other RTTs only waste a little bit of upstream bottlenecks
  - Rate goes as 1/p
- NB: higher loss rates may also be ok
  - but the argument isn't as simple

Relentless TCP [2009]

- Use packet conservation for window reduction
  - Reduce *cwnd* by the number of losses
  - New window matches actual data delivered
- Increase function can be almost anything
  - Increases and losses have to balance
    - Therefor the increase function directly defines the control function/model
  - Default is standard AI
    - Increase by one each RTT)
    - Resulting model is 1/p

Properties

- TCP part of control loop has unity gain
  - Network drops/signals what it does not want to see on the next RTT
    - e.g. if 1% too fast, drop %1 of the packets
  - Greatly simplifies Active Queue Management
  - Very well suited for *FQ
- The deployment problem is “only” political
  - Crushes networks that don't control their traffic

Closing

- The network needs to control the traffic
- Transport protocols need to be even more aggressive

Appendix

- Problems cause by new stacks

Problem 1

- TCP is window fair
  - Tends to equalize window in packets
  - Grossly unfair in terms of data rate
  - Short RTT flows are brutally aggressive
  - Long RTT flows are vulnerable
    - Any flow with a shorter RTT preempts long flows

Example

- 2 flows old TCP (32kB buffers)
  - 100 Mb/s bottleneck link
- Flow 1, 10 ms RTT, expected rate 3 MB/s
- Flow 2, 100 ms RTT, expected rate 0.3 MB/s
- Both: no interaction – they can't fill the link
  - Both users see predictable performance

With current stacks

- Auto tuned TCP buffers
  - Still 100 Mb/s bottleneck (12.5 MB/s)
- Flow 1, 10 ms RTT, expected rate 12 MB/s
- Flow 2, 100 ms RTT, expected rate 8(?) MB/s
- Both at the same time
  - Flow 1, expected rate 10(?) MB/s
  - Flow 2, expected rate 1(?) MB/s
    - Wide fluctuations in performance!

Problem 2

- Some apps (e.g. p2p) present “infinite” load
- Consider peer-to-peer apps as:
  - Distributed shared file system
  - Everybody has a manually manged local cache
- As the network gets faster
  - Cheaper to fetch on whim and discard carelessly
  - Presented load rises with data rate
  - Faster network means more wasted data

Problem 3

- TCP's design goal is to fill the network
- By causing a queue at every bottleneck
  - Controlling hard against drop tail
  - RED (AQM) really hard to get right
- You don't want to share with a non-lame TCP
  - Everyone has experienced the symptoms
- TCP friendly is an oxymoron
  - Me, at the last IETF

Impact of the new stacks

- Many things run faster
- Higher delay or loss nearly everywhere
  - Intermittent congestion in many parts of the core
  - Impracticable to out-build the load
  - The network needs QoS
- Very unstable or unpredictable TCP performance
  - Vastly increased interactions between flows

The business problem

- Unpredictable performance is a killer
  - Unacceptable to users
  - Can't write SLAs to assure performance
- A tiny minority of users consume the majority of the capacity
  - Trying to out-build the load can be very expensive
  - And may not help anyhow

ISPs need to do something

- But there are no good solutions
- ISP are doing desperate (&misguided) things
  - Throttle high volume users or apps to provide cost effective and predictable performance for small users

TCP is still lame

- Cwnd (primary control variable) is overloaded
- Many algorithms tweak cwnd
  - e.g. burst suppression
- Long term consequences of short term events
  - May take 1000s of RTT to recover from suppressing one burst
- Extremely subtle symptoms
  - Not generally recognized by the community

Desired fix

- Replace *cwnd* by (*cwnd*+*trim*) “everywhere”
- *Cwnd* is reserved for primary congestion control
- *Trim* is used for all other algorithms
  - Signed
  - Converges to zero over about one RTT
- Would expect more predictable and better modeled behavior

A slightly better fix

- *trim* can be computed implicitly
  - It is the error between *cwnd* and *flight_size*
- On each ACK:

$$trim = flight\_size - cwnd$$

  - Existing algorithms update *cwnd* and/or *trim*

Even better

- The entire algorithm can be done implicitly

On each ACK compute:

flight_size = (Estimate of data in the network)

delivered = (The quantity of data accepted by the receiver)

(= the change in snd.una, adjusted for SACK blocks)

willsend = *delivered*

If *flight_size* < *cwnd*: *willsend* = *willsend* + 1

If *flight_size* > *cwnd*: *willsend* = *willsend* - ½

heuristic_adjust(*willsend*) // Bursts suppression, paceing, etc

send(*willsend*, *socket_buffer*)

Properties

- Strong packet conserving self-clock
- Three orthogonal subsystems
  - Congestion control
    - Average window size (&data rate)
  - Transmission control
    - Packet scheduling and burst suppression
  - Retransmissions
    - Reliable data delivery

Congestion control revisited

- Can use standard AIMD congestion control:

  On loss: cwnd = cwnd/2

  On ACK: cwnd = cwnd + (1/cwnd)

  - Expect cleaner behavior than current stacks
- Can trivially use other algorithms
  - No collisions with algorithms overloading *cwnd*
  - Unconstrained choices for both increase and decrease functions
    - Huge research opportunities