Implementation of the algorithm that calculates the sending rate in MulTFRC

TFRC with weighted fairness

draft-irtf-iccrg-multfrc-01.txt

- work in progress towards version -02

Stein Gjessing

University of Oslo, Norway

What is MulTFRC?

- Like MulTCP: a protocol that is *N*-TCP-friendly
  - $N \in R^{+}$
  - Larger range of possible values for *N* than for others, e.g. MulTCP and CP
  - Yields flexible weighted fairness (e.g. priorities between users, or between flows of a single user)

- Based on TFRC
  - Easy to implement as an extension of TFRC code
  - Change the equation + measure "real" packet loss

- Current draft: draft-irtf-iccrg-multfrc-01.txt

Research background

- Ph.D. thesis of Dragana Damjanovic
(now finished and evaluated with best marks)
  - Equation derivation: SIGCOMM poster, tech. rep., paper with derivation + MulTFRC under submission
  - MulTFRC: CCR paper

- Extensive evaluations: equation validation, MulTFRC tests, both in simulations and real life
  - MulTFRC also successfully demonstrated for Europe-China file transfer at final review of European IST FP6 STREP project "EC-GIN"

- All documentation and code available from:
http://heim.ifi.uio.no/michawe/research/projects/multfrc/

In MulTFRC, the algorithm/equation for calculating the sending rate (X_Bps bytes/sec) is, using real numbers:

```
If (N < 12) {
      af = N * (1-(1-1/N)^j);
  } Else {
      af = j;
  }
af= max(min(af,ceil(N)),1);
a = p*b*af*(24*N^2+p*b*af*(N-2*af)^2);
x = (af*p*b*(2*af-N)+sqrt(a))/(6*N^2*p);
z = t_RTO*(1+32*p^2)/(1-p);
q = min(2*j*b*z/(R*(1+3*N/j)*x^2), N*z/(x*R), N);
X_Bps = ((1-q/N)/(p*x*R)+q/(z*(1-p)))*s;
```

Implementation considerations (appendix):

For all parameter values:

Assure no overflow, underflow and rounding errors.

Can it be implemented with integers?

```
If (N < 12 && N > 1 && j  < N*2) {      // more restrictive than in draft 01
      af = N * (1-(1-1/N)^j);              // af means: affected flows
      }
      Else { af = j; }
   af = max(min(af,ceil(N)),1);
   a = p*b*af*(24*N^2+p*b*af*(N-2*af)^2);
   x = (af*p*b*(2*af-N)+sqrt(a))/(6*N^2*p);
   z = t_RTO*(1+32*p^2)/(1-p);
   q = min(2*j*b*z/(R*(1+3*N/j)*x^2), N*z/(x*R), N);
   X_Bps = ((1-q/N)/(p*x*R)+q/(z*(1-p)))*s;
```

Where:

- s   is the segment size in bytes (excluding IP and transport protocol headers).
- R   is the round-trip time in seconds.
- t_RTO is the TCP retransmission timeout value in seconds.
- b   is the maximum number of packets acknowledged by a single TCP ack.
- p   is the loss event rate, between 0 and 1.0,
  the number of loss events as a fraction of the number of packets transmitted.
- j is the number of packets lost in a loss event.
- N is the number of TFRC flows that MulTFRC should emulate.

Main approach

- Find the bounds of the parameters ?
- Use Integer arithmetic
  - (Do we need 64 bits?)

Implementation challenges:

- Good precision / no underflow
- No overflow

Two special operations

1. (1-1/N)^j (j, N: real)
    1. Define M = 1-1/N
    2. Let j = j´+ k/m, j´, k and m are integers
    3. M^j = M^j´* root(M,m)^k
2. sqrt(a)

- lots of algorithms out there
    - e.g. http://en.wikipedia.org/wiki/Nth_root

- Alternative for (1-1/N)^j : Use tables

p – the loss event rate

- From RFC 3649 - High speed TCP: " For example, for a Standard TCP connection with 1500-byte packets and a 100 ms round-trip time, achieving a steady-state throughput of 10 Gbps would require an average congestion window of 83,333 segments, and a packet drop rate of at most one congestion event every 5,000,000,000 packets (or equivalently, at most one congestion event every 1 2/3 hours). This is widely acknowledged as an unrealistic constraint."
- 10^-10 < p < 0.99 (p=1 is treated as a special case, and since p is calculated as the average of the last 8 loss intervals, p < 0.99 is ensured (see the current draft))
- New p: Integer: Multiply old p by 10^10:

  1 < p < 99*10^8

t_RTO is the TCP retransmission timeout value (in seconds)

Max t_RTO:
From RFC 2988:
"A maximum value MAY be placed on RTO provided it is at least 60 seconds."
This translates into: RTO is theoretically not bounded, but every TCP implementation must be able to cope with at least a max. of 60 seconds, and so, to make MulTFRC compatible with such low-end implementations, we choose a value even larger than this (about 16 minutes).

Min t_RTO:
Arguments for 10 ms, eg.:
http://mail.opensolaris.org/pipermail/dtrace-discuss/2006-January/000961.html
Arguments for even smaller values (down to 1 microsecond ?), eg.:
http://www.ittc.ku.edu/utime/

Let us use 10 microseconds as min value:
$10^{-5} < t\_RTO < 10^3$ (16 minutes)

Or with new t_RTO as an integer
$1 < t\_RTO < 10^8$:

b: the number of packets ACKed by one ACK:

The basic rule related to this is from RFC 1122:
"A TCP SHOULD implement a delayed ACK, but an ACK should not be excessively delayed; in particular, the delay MUST be less than 0.5 seconds, and in a stream of full-sized segments there SHOULD be an ACK for at least every second segment."
RFC 5690 (ACK-CC) describes the most conservative behavior so far: "... the TCP receiver always sends at least K=2ACKs for a window of data, even in the face of very heavy congestion on the reverse path."
A "window of data" can be as large as the bandwidth*delay product, so this could be a huge bandwidth value times the max. RTT, and divided by the smallest possible packet size.

One could also compute how many packets can at most arrive within 0.5 seconds?

RFC 3649 (High speed TCP): ...." would require an average congestion window of 83,333 segments," ...

1 <= b < 10^5

What parameter values should the algorithm accept (lower and upper bounds) ?
Real numbers

s is the segment size in bytes (excluding IP and transport protocol headers).

$40 < s < 64K$ (The algorithm is very robust to this size)

R is the round-trip time in seconds.

$10^{-5} < R < 10^3$ (ten microseconds to almost 17 minutes)

b is the maximum number of packets acknowledged by a single TCP acknowledgement

$1 < b < 10^5$

p is the loss event rate, between 0 and 1.0, of the number of loss events as a fraction of the number of packets transmitted.

$10^{-10} < p < 0.99$

j is the number of packets lost in a loss event (Can we use integer?)

$1 <= j < 10^8$

t_RTO is the TCP retransmission timeout value in seconds

$10^{-5} < R < 10^3$ (ten microseconds to almost 17 minutes)

N is the number of TFRC flows that MulTFRC should emulate. N is a positive rational number

$0.01 <= N <= 1000$

After parameter conversion to integers:

- ✓ s is the segment size in bytes (excluding IP and transport protocol headers).
  $10 < s < 64K$ (The algorithm is very robust to this size)

- ✓R is the round-trip time in 10 microseconds:
  $1 < R < 10^8$

- ✓b is the maximum number of packets acknowledged by a single TCP acknowledgement
  $1 < b < 10^5$

- ✓p is the loss event rate (per 10^10 packets):
  $1 < p < 99*10^8$

- ✓j is the number of packets lost in a loss event (do we need more precision for small j´s?)
  $1 <= j < 10^8$

- ✓t_RTO is the TCP retransmission timeout value in 10 microseconds
  $1 < R < 10^8$

- ✓N is the number of TFRC flows that MulTFRC should emulate (in 0.01 flows)
  $1 <= N <= 10000$

af: affected flows

```
If (N < 12 && N > 1 && j  < N*2) {      // more restrictive than in draft 01
      af = N * (1-(1-1/N)^j);               // af means: affected flows
      }
      Else { af = j; }
   af = max(min(af,ceil(N)),1);
```

1<= af < N+1.

4 decimal digits will be a reasonable precision (the same as for N)

Possible overflow in

```
a = p*b*af*(24*N^2+p*b*af*(N-2*af)^2);
```

Because:

- For large N: N^4, ie. 10^16
- for large p: p^2, ie. 10^20
- for large b: b^2 ie. 10^10

To do in order to avoid overflow

a = p*b*af*(24*N^2+p*b*af*(N-2*af)^2);

x = (af*p*b*(2*af-N)+sqrt(a))**/(6*N^2*p);**

Move the divison by (6*N^2*p) into the calculation of "a" so that it does not overflow.

Possible move of sqrt(a) up to the calculation of a

Preliminary investigations of the rest of the algorithm lead is to believe that there are no more overflow problems.

Additional approach

- “Scale down” the algorithm
  - Set scaling factors according to size of parameters
  - And/or:
    - For large values of a parameter, use one version
    - For small versions, use another on
    - (but hopefully not 2^n algorithms)

Conclusion

- The result will be included on version -02 of the draft
  - As implementation considerations appendix
- Should appear before the next IETF meeting
  - March/April 2011 in Prague