Making available Base-RTT for use in congestion control applications

D.J.Leith, R.N.Shorten, G. McCullagh
Hamilton Institute, National University of Ireland Maynooth

***Abstract*— In this paper we revisit the interaction between $baseRTT$ estimation and congestion control action. We develop a simple AIMD-based scheme that allows network buffers to drain and thus demonstrate in a constructive manner that, with proper design, it is indeed possible for flows traversing a bottleneck link to estimate their base RTT reliably.**

I. Introduction

Estimation of round-trip propagation delay, also referred to as $baseRTT$, is a fundamental part of many congestion control algorithms. Apart from its evident importance in delay-based algorithms such as FAST TCP [4] and TCP Vegas [1], it also plays an important role in recently proposed loss-based (and hybrid) schemes such as TCP Westwood, Microsoft Compound, and H-TCP [3] in which flows adaptively set their backoff factor to $\beta = baseRTT/RTT_{max}$, where RTT_{max} is related to the measured RTT at backoff. In this latter context, the ability to estimate $baseRTT$ effectively decouples the congestion control algorithm from the issue of queue provisioning and enables high utilisation to be achieved with small buffers [5].

Accurate estimation of $baseRTT$ is, however, known to potentially be problematic. A primary issue is interactions between $baseRTT$ estimation and the congestion control algorithm itself. For example, in TCP Vegas and related algorithms a standing queue is induced as part of the correct operation of the congestion control algorithm. Thus, when flow start times are staggered, later flows tend to over-estimate $baseRTT$ due to the standing queue created by earlier flows. Similar issues can also arise with loss-based algorithms. For example, if the AIMD backoff factor used is $\beta = baseRTT/RTT_{max}$ (as in H-TCP and some versions of Westwood), then overestimation of $baseRTT$ may mean that flows do not empty network queues, allowing the overestimate to persist indefinitely. Statistical multiplexing of flow backoffs on links shared by many loss-based flows can also lead to later flows experiencing a standing queue and so overestimating $baseRTT$.

In this paper we revisit the interaction between $baseRTT$ estimation and congestion control action. We develop a simple AIMD-based scheme that allows network buffers to drain and thus demonstrate in a constructive manner that, with proper design, it is indeed possible for flows traversing a bottleneck link to estimate their base RTT reliably.

The financial support of Cisco Systems for this work is gratefully acknowledged. This work was also supported by SFI grant 04/IN3/I460.

II. Background

The context for the present work is a Cisco funded project to investigate delay-based AIMD congestion control [2]. The basic idea here is that by backing off cwnd when queueing delay exceeds some threshold, we can avoid filling the queue (thus maintaining low queueing delay) while staying within the well-established AIMD framework. Further, by adapting the AIMD backoff factors as proposed in [5], we can also achieve high network utilisation. Since this algorithm is an AIMD strategy, networks deploying the delay-based algorithm exhibit the usual fairness and convergence properties of AIMD. For convenience, we recap the algorithm proposed in [2]:

$$cwnd \leftarrow \begin{cases} cwnd + \alpha/cwnd, & on\ each\ ACK \\ \beta cwnd, & if\ \tau \geq \tau_0 \\ \beta cwnd, & if\ packet\ loss \end{cases}$$

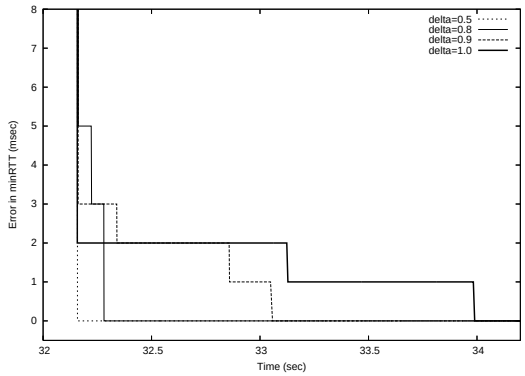
where τ is the observed queueing delay, $\tau_0 > 0$ is a delay threshold that triggers delay-based backoff (set here to $= 50ms$). The queueing delay τ is estimated as $sRTT - \hat{T}(k)$ where $\hat{T}(k)$ is the minimum round-trip time observed so far and $sRTT$ is an estimate of the current round-trip time. $\hat{T}(k)$ may be interpreted as an estimate of $baseRTT$, although it is important to stress that we do not assume that it is necessarily an accurate estimate. The backoff factor is

$$\beta(k) = \delta\hat{T}(k)/RTT(k) \tag{1}$$

where $RTT(k)$ is the measured RTT before the k'th backoff event[1] , and $0 \leq \delta < 1$ is a design parameter.

It is the impact of the choice of backoff factor (1) that is the primary focus of the present paper. While we often illustrate results with reference to the delay-based AIMD algorithm, all our analysis extends to general AIMD algorithms including loss-based algorithms. To make this explicit, we therefore also include examples illustrating loss-based AIMD operation. Our main result is that with the choice of backoff factor (1) only very mild conditions are needed for the bottlenecked buffer to drain and for the true value of $baseRTT$, to be available to network flows regardless of initial estimation errors. This fact is shown both analytically and experimentally. Details of the experimental testbed are given in the Appendix

[1]It is not essential that $RTT(k)$ equals the delay before backoff, it is only required that $RTT(k)$ is greater than or equal to this delay. In our implementation the $RTT(k)$ value used is a quantity that tracks the maximum observed RTT and decays towards $\hat{T}$ during periods when the current RTT is below $RTT(t)$.



(a) Delay-based AIMD

Fig. 1. Experimental measurements of estimation error $\hat{T} - T$ vs time. Measurements are shown for a range of values of the design parameter δ. Initial estimate of base RTT is hard-wired to an incorrect value to illustrate convergence. 10Mbps link, RTT 200ms, one delay-based AIMD TCP flow.

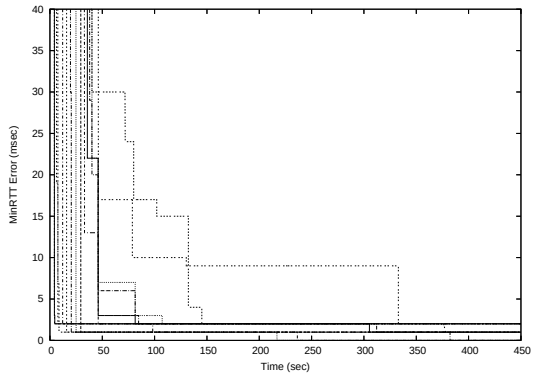


Fig. 2. Experimental measurements illustrating queue draining with multiple flows. 10Mbps link, 125KB buffer, mix of flow RTTs 20-200ms, $\delta = 0.8$, 16 TCP Reno flows with adaptive backoff and randomised start times. Initial base RTT estimates for all flows are hard-wired to incorrect values to confirm insensitivity of convergence to estimation errors.

III. Draining network buffers

To help gain some insight into the mechanics of the backoff algorithm, consider for the moment a network with a single flow. Let B denote the link bandwidth in packets/s, T the round-trip propagation delay. Consider the k'th backoff event and let $w(k)$ denote the congestion window of the flow at backoff and Q_k the network buffer occupancy. At backoff, we have that $RTT(k) = T + Q_k/B$ and $w(k) = B \times RTT(k)$. Following backoff, the flow cwnd is $\beta(k)w(k)$. Selecting $\beta(k)$ according to (1),

$$\beta(k)w(k) = \delta\frac{\hat{T}(k)}{RTT(k)}B \times RTT(k) = \delta B\hat{T}(k)$$

If $\hat{T}(k) = T$, then since $\delta < 1$ it can be seen that cwnd falls *below* the link bandwidth-delay product BT. Thus the queue empties thereby providing an opportunity for the flow to observe the propagation delay T. If $\hat{T}(k) > T$ then the queue need not empty after backoff. The buffer occupancy after backoff is $q_k = \beta(k)w(k) - BT = B(\delta\hat{T}(k) - T)$ and the round-trip delay is $T + q_k/B = \delta\hat{T}(k)$. Since $\delta < 1$, the round-trip delay is *lower* than the previous lowest observed delay $\hat{T}(k)$. Hence, the flow can update $\hat{T}$ to a value that is closer to the true propagation delay T. In effect, we are using the multiplicative decrease action to probe the network to discover whether an RTT *below* our current best estimate $\hat{T}$ is possible. After a number of congestion events (the number being dependent on the size of the initial error in $\hat{T}$ and on the value of δ), we can see the flow is eventually guaranteed to obtain an accurate estimate of the propagation delay T. This is illustrated in Figure 1, which shows experimental measurements of $\hat{T}$ converging to T.

A. Detailed Analysis

Consider n flows sharing a bottleneck link. Let $w_i(k)$ denote the cwnd of flow i at the k'th backoff event, let T_i be the round-trip propagation delay of flow i. Let Q_k be the buffer size at the k'th congestion event. Note that this need not be the maximum buffer size when delay-based AIMD is used. When delay-based congestion control it also need not be the same at every congestion event (due to burstiness etc). At congestion we have that the aggregate flow rate equals the link rate, i.e.

$$\sum_{i=1}^{n}\frac{w_i(k)}{T_i + Q_k} = B \tag{2}$$

Following backoff, the aggregate rate becomes $\sum_{i=1}^{n}\beta_i(k)\frac{w_i(k)}{T_i+q_k}$, where q_k is the queue occupancy after backoff ($q_k < Q_k$) and $\beta_i(k)$ is the backoff factor of flow i.

If the queue empties on backoff, then $q_k = 0$ and flows have the opportunity to measure their base round-trip time T_i. If the queue does not empty on backoff, then the aggregate flow rate continues to equal the link rate, i.e.

$$\sum_{i=1}^{n}\beta_i(k)\frac{w_i(k)}{T_i + q_k} = B \tag{3}$$

Assume that flow backoffs are synchronised i.e. every flow backs off at each congestion event (this assumption is relaxed later). Also assume for the moment that each flow observed the RTT at the $k-1$'th backoff when the queue occupancy was q_{k-1} (again, we relax this assumption later). The flow backoff factors then satisfy

$$\beta_i(k) \le \delta\frac{T_i + q_{k-1}}{T_i + Q_k} \quad \forall i \in 1, .., n$$

Substituting into (3),

$$B = \sum_{i=1}^{n}\beta_i(k)\frac{w_i(k)}{T_i + q_k} \le \sum_{i=1}^{n}\delta\frac{T_i + q_{k-1}}{T_i + q_k}\frac{w_i(k)}{T_i + Q_k} \tag{4}$$

Using (2), it then follows that $\exists i$ such that $\delta\frac{T_i+q_{k-1}}{T_i+q_k} \ge 1$. i.e. $q_k \le \delta q_{k-1} - (1-\delta)T_i$. Thus, provided $\delta < 1$ the queue occupancy at backoff q_k decreases monotonically until eventually the queue empties, providing an opportunity for flows to measure their base round-trip time. This is illustrated for example in Figure 2.

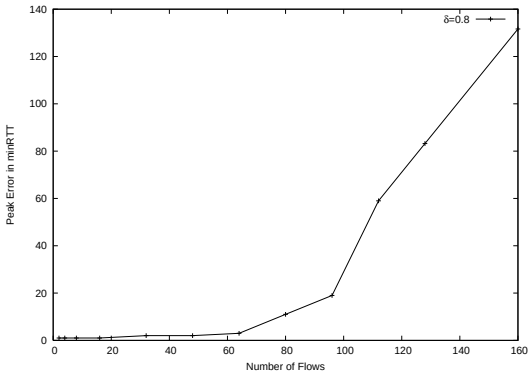
B. Discussion

Convergence Rate. The rate of decrease is evidently influenced by the choice of δ, decreasing δ increasing the rate at which the queue drains. This can be seen, for example, in Figure 1.

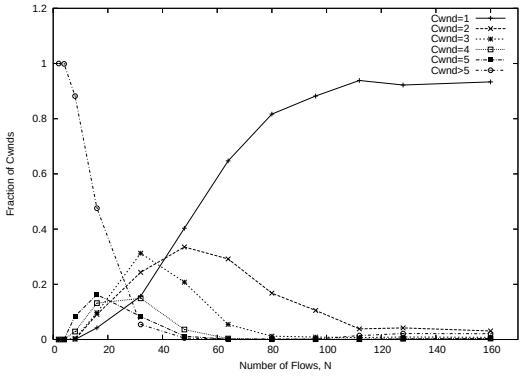
Unsynchronised Drops We can capture unsynchronised backoffs by setting $\beta_i(k) = 1$ for flows which do not backoff at the k'th congestion event. The foregoing analysis can then be immediately extended to the case of unsynchronised flows under mild assumptions. Specifically, assume that at congestion events synchronised backoffs occur with probability lower bounded by $p_s > 0$. That is, it occasionally happens that all flows backoff together at a congestion event. This assumption can be relaxed in various ways but this is beyond the scope of the present work.

Observability Our analysis assumes that each flow observes the RTT after the k'th backoff. It is easy to see that this assumption may, however, be further relaxed to the much weaker requirement that there is a non-zero probability p_i that over a congestion event flow i observes an RTT less than or equal to the RTT after the k'th backoff.

Quantisation of cwnd Our analysis assumes that the specified backoff factor (1) is successfully applied to the flow cwnd. A notable exception to this occurs when the flow cwnd is only one packet in size. Since this is the lowest admissible cwnd, the backoff factor specified by (1) cannot be applied. This is illustrated, for example, in Figure 3(a) which plots the worst-case (over all flows) error in estimated baseRTT as the number of flows is increased. Also shown in Figure 3(b) is the distribution of flow cwnd values vs the number of flows. It can be seen that the worst case estimation error begins to rise as the number of flows increases above 60. This corresponds to a regime where around 60% of flows have a cwnd of only one packet and around 35% have cwnd of two packets. Above around 100 flows, $> 90\%$ of flows have a cwnd of one packet. Since flows can no longer backoff their cwnd, a standing queue develops at the link buffer and the estimation error of later flows inevitably increases. We note that this issue can potentially be resolved by introducing more fine-grained control of the flow send rate at low cwnd via, for example, pacing. Consideration of such extensions is, however, beyond the scope of the present paper.



(a) Worst case estimation error



(b) Flow cwnd distribution

Fig. 3. Illustrating quantitisation issues as the number of flows on a link is increased and flow cwnds tend towards one packet. 10Mbps link, mix of flow RTTs 20-200ms, $\delta = 0.8$, delay-based AIMD (similar results are obtained for Reno with adaptive backoff).

IV. Conclusions

In this paper we revisit the interaction between $baseRTT$ estimation and congestion control action. We develop a simple AIMD-based scheme that allows network buffers to drain and thus demonstrate in a constructive manner that, with proper design, it is indeed possible for flows traversing a bottleneck link to estimate their base RTT reliably.

References

[1] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of SIGCOMM*, pages 24–35, 1994.

[2] D.J.leith, J.Heffner, R.N.Shorten, and G.D.McCullagh. Delay-based aimd congestion control. In *Proc. Workshop on Protocols for Fast Long Distance Networks, Los Angeles.*, 2007.

[3] D.J.Leith and R.N.Shorten. H-TCP protocol for high-speed long-distance networks. In *Proc. 2nd Workshop on Protocols for Fast Long Distance Networks. Argonne, Canada, 2004*, 2004.

[4] C. Jin, D. X. Wei, and S. H. Low. FAST TCP: Motivation, architecture, algorithms, performance. In *IEEE INFOCOM 2004*, 2004.

[5] R. Shorten and D. Leith. On queue provisioning, network efficiency and the delay-bandwidth product. IEEE Transactions on Networking, to appear, 2006.

Appendix

We implemented the adaptive backoff in Linux 2.6.23 for both the NewReno/SACK and delay-based AIMD algorithm algorithms. Experiments were carried out using a testbed consisting of commodity PCs connected to gigabit switches to form the branches of a dumbbell topology. All sender and receiver machines used in the tests have identical hardware and software configurations and are connected to the switches at 1Gb/sec. The router, running FreeBSD v4 with the dummynet module, can be configured with various bottleneck queue-sizes, capacities and round trip propagation delays to emulate a range of network conditions. TCP Flows are injected into the testbed using `iperf.` TCP stacks are instrumented using a modified version of the Linux `tcpprobe` module.