Inferring Queue State by Measuring Delay in a WiFi Network

David Malone, Douglas J Leith, Ian Dangerfield

Hamilton Institute, NUI Maynooth.

Abstract. Packet round trip time is a quantity that is easy to measure for end hosts and applications. In many wired networks, the round trip time can be used to estimate how full network buffers are because of relatively simple relationships between buffer occupancy and drain time. Thus RTT has been exploited for purposes such as congestion control and bandwidth measurement. In some networks, such as 802.11 networks, the buffer drain times show considerable variability due to the random nature of the MAC service. We examine some of the problems faced when using round trip time measurements in these networks, particularly in relation to congestion control.

1 Introduction

Network round-trip time is a useful measurement that is easily estimated by end hosts. It is often used as a measure of network congestion either implicitly (e.g. a human looking at the output from traceroute or ping) or explicitly (e.g. TCP Vegas [3], FAST [11] or Compound TCP [10] using RTT as a proxy measure of buffer occupancy). The assumption is that queueing is the main source of variation in RTTs, and so RTTs can be used to estimate queueing. This has also been used to build tools such as pathchar to estimate available bandwidth [6].

In wired networks, this is often a reasonable assumption: there is usually a linear relationship between queue length and queue drain time. However, this relationship is not universal. In WiFi networks there can be a significant random component associated with transmitting packets. A device usually has a back-off period before sending. The duration of this period is a combination of a randomly selected number and the duration of busy periods due to other traffic on the network [5]. Also, a packet may suffer a collision or corruption, requiring further back-off periods and retransmission.

Figure 1 shows observed queue drain times plotted against queue length from a device transmitting packets over a contended WiFi link. The most striking thing about this graph is the overlap between RTTs associated with different queue lengths: RTTs observed for a queue length of one packet could have come from a queue length of 10 packets, and measurements from a queue of 10 packets could easily have come from a queue of 20 packets. Even before other sources of delay are considered, this is going to be a challenging environment for making inference about queue length from RTTs.

In this paper we investigate the complications introduced by the random nature of the service in 802.11. We note that there have been complications in application or transport layer measurement of RTTs in wired networks (for example, some filtering is

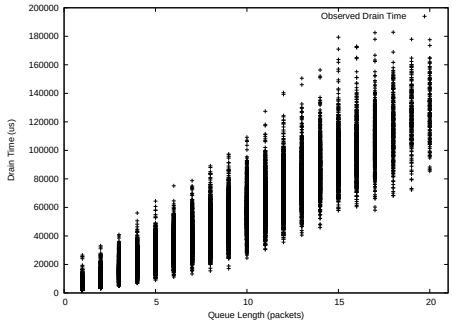

Fig. 1. The Impact of Queue Length on Drain Time: Scatter plot of observed values. Testbed setup as in Section 2, scenario described in Section 3.

necessary to remove artifacts caused by TCP's delayed ACKing or TSO [7]). However, in order to focus on the issues raised by the varying delay of a wireless link, in this paper we will assume that accurate RTT measurements are available.

2 Testbed Setup

We consider network delay associated with winning access to transmission opportunities in an 802.11 WLAN. We measure both the queue drain time (the time from when a packet reaches the driver to when the transmission is fully complete) and the MAC service time (the time from reaching the head of the hardware interface queue to when transmission is fully complete). The MAC service time can range from a few hundred microseconds to hundreds of milliseconds, depending on network conditions. We use techniques similar to those in [4] to measure MAC delay and drain time.

The 802.11 wireless testbed is configured in infrastructure mode. It consists of a desktop PC acting as an access point, 15 PC-based embedded Linux boxes based on the Soekris net4801 [2] and one desktop PC acting as client stations. The PC acting as a client records measurements for each of its packets, but otherwise behaves as an ordinary client station. All systems are equipped with an Atheros 802.11b/g PCI card with an external antenna. All nodes, including the AP, use a Linux kernel and a version of the MADWiFi [1] wireless driver modified to record packet statics at the driver layer. All of the tests are performed using the 802.11b physical maximal transmission rate of 11Mbps with RTS/CTS disabled and the channel number explicitly set.

Our measurements of delay take place in the driver, so we are observing the drain times and the queue lengths in the queue of descriptors between the driver and the Atheros WiFi card. A queue of 20 packets is configured at this level. Though we will focus on the queueing at the driver layer, similar issues arise with the whole queueing system because of the random service times.

3 Raw RTT Signal

The data used for Figure 1 is taken from a run from our testbed where 4 stations are uploading a file using TCP. The measurements are taken from one of the TCP senders, and so all of the packets sent are 1500 byte packets with a full TCP payload. The results are taken over about 110s where network conditions are essentially static.

Let us briefly consider the simple problem of determining if the queue in Figure 1 contains more than ten packets based on the observed drain time. For example, consider a simple threshold scheme: set a threshold and if the observed time is greater than the threshold, we infer it has more than ten packets, otherwise we infer it has less than ten packets. Even if we have prior knowledge of the drain time distribution in Figure 1, how effective can such a scheme be for a WiFi network?

Figure 2 shows how often this scheme makes a mistake for a range of different thresholds. First consider the upper curve in Figure 2(a). This is the chance that that the delay threshold incorrectly indicated that the queue length was either above or below 10 packets. The upper curves in Figure 2(b) show how this breaks down into situations where the queue was small but the delay was big or the queue was big but the delay was small. The best choice of threshold, around 60,000μs, makes a mistake just over 10% of the time. As we expect, this is around ten times the mean service time.

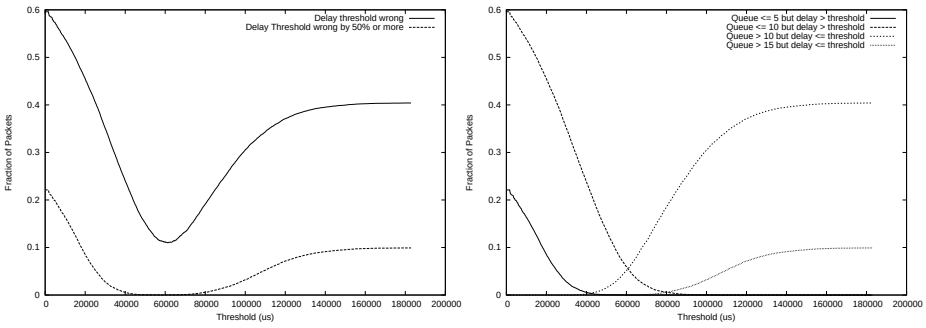

Fig. 2. Simple thresholding of delay measurements: (a) how often simple thresholding is wrong, (b) breakdown of errors into too big or too small.

Of course, it is possible that these mistakes occur when the queue mainly when the queue is close to 10 packets. To check this we also calculate the chance that while the queue had five of fewer packets that the delay is less than the threshold and the chance that the queue has more than fifteen packets while the delay is short. These represent large mistakes by the simple threshold technique. The results are the lower set of curves in Figure 2(b), with a large flat section for threshold values from 40,000 to 80,000μs. This is better news for our thresholding scheme: it is making mistakes regularly but these are not gross mistakes and a range of thresholds produce reasonable results.

These suggests that though delays measurements are quite noisy, there is some hope of learning information about queue length from them. Some basic statistics for the

drain times against queue lengths are shown in Figure 3. We show the mean drain time and a box-and-whiskers plot showing the range of measurements and the 10th and 90th percentiles. The number of drain time samples seen for each queue length is also shown.

Figure 3 also shows the estimated autocorrelation for the MAC service times, queue drain times and queue lengths. We see that the MAC service times show no evidence of correlation structure. This is what we what we intuitively expect from an 802.11 network operating without interference. In contrast, the queue lengths show a complicated correlation structure. The queue lengths are sampled at the time a transmission completes; because the queue length will not change much between these times we expect strong correlation over lags comparable to the queue length. The longer term structure in the queue length will be a function of TCP's congestion control behaviour in this network. Finally, the queue drain times show a *similar* structure to that observed for the queue lengths. This is encouraging: the drain time and the queue length are in some sense carrying similar information. We can confirm this by calculating the Pearson correlation value of 0.918.

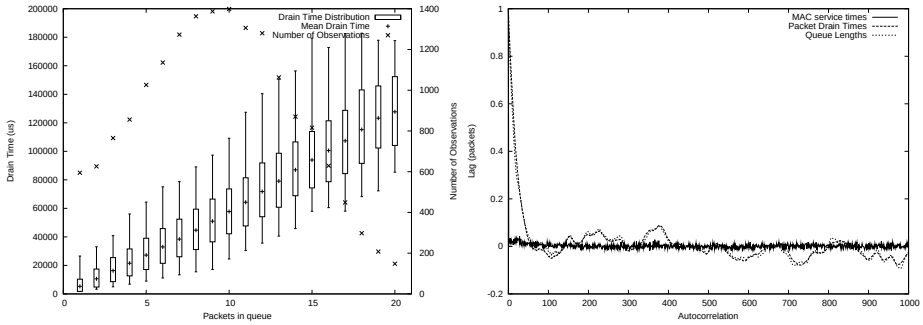

Fig. 3. Statistics for drain time against queue size (a) 10–90% box-and-whiskers plot and mean. The number of samples is also shown on the right hand axis, (b) autocorrelation for the sequence of MAC service times, queue drain times and queue lengths.

Based on the very low autocorrelation of the MAC service times, we note that it may be reasonable to approximate the drain time of the queue of length n as the sum of n independent service times. We know the variance of the sum of random variables grows like the sum of the variances. Thus we expect the range of the 10–90% percentiles to scale like $\sqrt{n}$. This is confirmed in Figure 3, where we plot the standard deviation of the drain times and compare them to $\sqrt{n}$. Note that this suggests that larger buffers will make RTT measurements even more challenging.

4 Smoothed RTT Signal

Most RTT measurements are smoothed before use, and based on the statistics we have seen in the previous section, there is a reasonable possibility that this may help in un-

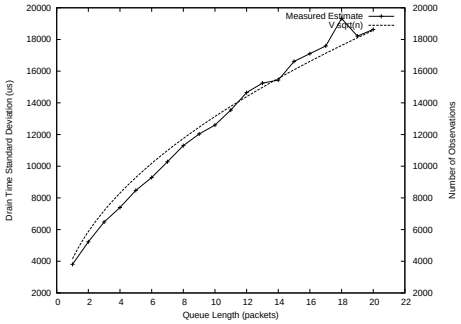

Fig. 4. Estimate of standard deviation of drain times as a function of queue length.

derstanding queue behaviour. In this section we will look at the impact of a number of commonly used filters on our ability to estimate the queue length.

Maybe the best known example of the use of a smoothed RTT is the sRTT used in TCP to estimate round-trip timeouts. This estimator updates the smoothed estimate every time a new estimate arrives using the rule

$$\mathrm{srtt} \leftarrow 7/8\mathrm{srtt} + 1/8\mathrm{rtt}. \tag{1}$$

We'll refer to this as 7/8 filter. It is also used in Compound TCP for delay based congestion control. We can also do a similar smoothing based on the time between packets:

$$\mathrm{srtt} \leftarrow e^{-\Delta T/T_c}\mathrm{srtt} + (1 - e^{-\Delta T/T_c})\mathrm{rtt}. \tag{2}$$

Here, ΔT is the time since the last packet and T_c is a time constant for the filter. This filter approximately decreases the weight of RTT estimates exponentially in the time since the RTT was observed. We'll refer to this as the Exponential Time filter.

TCP Vegas and derivatives use a different smoothing. Ns2's implementation of Vegas uses the mean of all the RTT samples seen over a window of time that is about the same as the current RTT in order to estimate the queue's length. In order to avoid spurious spikes due to delayed acking, the Linux implementation of TCP Vegas actually uses the minimum of the RTT's seen over a similar window. We'll refer to these as the Windowed Mean and Windowed Minimum filters.

We applied each of these filters to the drain time data to see if the resulting smoothed measurement was better able to predict the queue length. We used a window size/time constant of 100ms, which is comparable to the actual RTT experienced by TCP in our experiment. The results of our simple threshold test and calculation of autocorrelation are shown in Figure 5.

Interestingly, all the types of smoothing *except* the Windowed Minimum seem to have made things worse. The achievable error rate for thresholding has increased from 11% to 15, 18 and almost 20% for 7/8s, Windowed Mean and Exponential Time filters. The Windowed Minimum actually achieves an error rate of around 10.5%, which is comparable with the best raw drain time error rate.

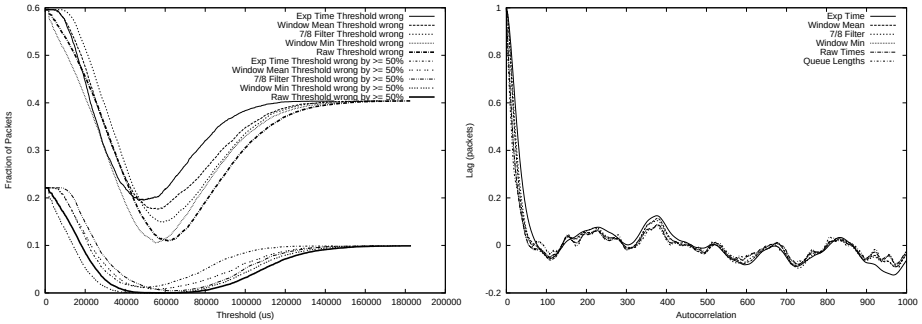

Fig. 5. Thresholding of filtered delay measurements: (a) errors while thresholding simple measurements, (b) autocorrelation of filtered measurements.

The autocorrelation graph tell a similar story, where the raw times and the Windowed Minimum actually follow the queue length most closely. We can also calculate the Pearson correlation coefficient between the queue length and the various filtered drain times, where a linear relationship would result in a high correlation. The Windowed Minimum measurements have the highest correlation with the queue length (0.922) closely followed by the raw measurements (0.918). There is then a gap before the 7/8th filter, the Windowed Mean and the Exponential Time results (0.836, 0.797 and 0.752 respectively).

5 Variable Network Conditions

As we noted, the length of 802.11's random backoff periods are not just based on the selection of a random number, but also on the duration of busy periods due to the transmissions of other stations. In addition, the number of backoff periods is dependent on the chance of a collision, which is strongly dependent on the number of stations in the network and their traffic. This means that the RTTs observed by a station depend on traffic that may not even pass through the same network buffers.

For example, consider Figure 6. This shows the time history of queue lengths and drain times as we shut down the competing stations from the setup described in Section 3. By 242s there is little competing traffic in the system, and Figure 6(a) shows that the mean drain time and variability have been radically reduced. However, if we look at Figure 6(b) we see that this is not because the queue size has been reduced. In fact TCP Reno is keeping the queue closer to full because of reduced contention.

The impact when stations join the system can also be dramatic, as shown in Figure 7, when 4 TCP uploads are joined by another 4 TCP uploads just after 120s (note, to get 8 TCP uploads to coexist in a WLAN, we have used the ACK prioritisation scheme from [8], resulting in smoother queue histories). In this case we see basically no change in queue length, but almost a doubling of round trip time.

These changes in drain time are caused by a change in the mean service rate for the queue. Clearly, any scheme for detecting queue length based on round-trip time would

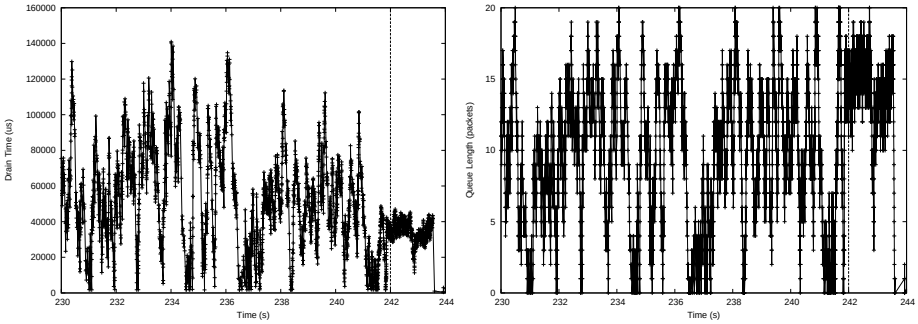

Fig. 6. The impact of other stations leaving the system: (a) drain times and (b) queue lengths. At time 242s other stations have stopped.

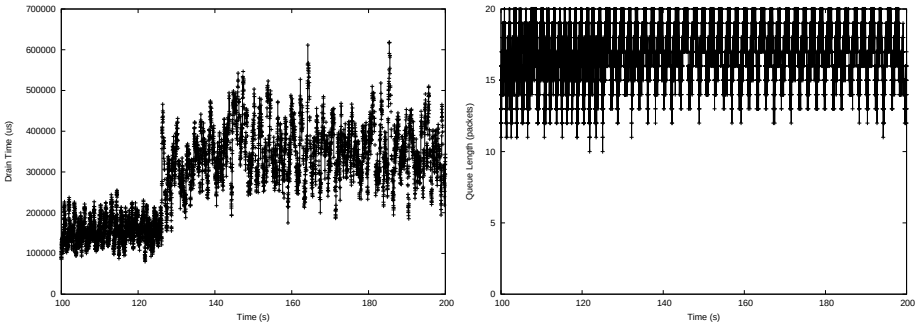

Fig. 7. The impact of other stations joining the system: (a) drain times and (b) queue lengths.

have detect changes in network conditions and re-calibrate. This also creates a problem for systems that aim to measure the base RTT, i.e. the round-trip time in the absence of queueing. Because the mean service rate depends on traffic that is not in the queue, a change in other traffic can cause a shift in the base RTT. As queueing time is usually estimated as RTT − baseRTT, this could be an issue for many systems.

6 Impact on TCP Vegas

In this section we look at the performance of Linux's TCP Vegas in light of our observations in the previous sections. We consider Vegas because it is one of the most simple delay-based congestion control schemes. We expect other delay based schemes, such as FAST and Compound, to face similar challenges. Linux's Vegas module alters congestion avoidance behaviour but basically reverts to Reno-like behaviour in other situations. Over each cwnd's worth of packets it collects a minimum RTT observed over that cwnd. It also maintains a base RTT, which is the smallest cwnd observed over the current period of congestion avoidance.

The target cwnd is then estimated as:

$$\text{target} = \text{cwnd}\frac{\text{baseRTT}}{\text{minRTT}}. \tag{3}$$

The difference between this and the current cwnd is compared to the Vegas constants $\alpha = 2$ and $\beta = 4$. If the difference is less than α then cwnd is increased, if it is greater than β it is decreased. The aim is that Vegas will introduce a few packets more than the bandwidth-delay product into the network and so result in a small standing queue.

Based on our previous observations, we anticipate two possible problems for Vegas. First, because Vegas is using RTT measurements, it is possible that the noise in these measurements will cause Vegas to incorrectly manage the queue, either resulting in an empty queue (reducing utilisation) or overfilling the queue (resulting in drops, which delay based schemes aim to avoid). Second, after a change in network conditions, Vegas may use an incorrect baseRTT. If this change results in an increased baseRTT then Vegas might continually reduce cwnd in an attempt to reduce the observed minRTT, resulting in poor utilisation.

In order to test these problems, we run a TCP flow across our testbed with various round trip times introduced with Dummynet [9]. After 60s we change the network conditions by introducing additional 11 stations, one per second, sending UDP packets at a high rate. As a baseline, we run a set of experiments with very large buffers and TCP Reno. Reno keeps these buffers from emptying, and so gives an indication of the best achievable throughput. Figures 8 shows the resulting throughput for a 50ms RTT; results for 5ms and 200ms round trips are similar.

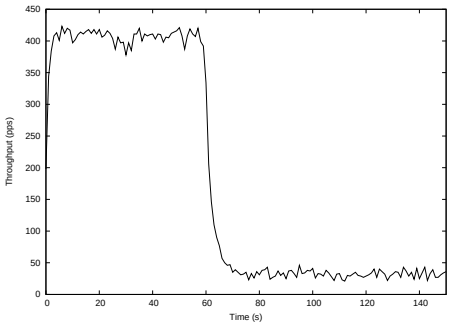

Fig. 8. Throughput for TCP Reno in an initially uncontended network with additional fbws introduced at 60s.

Figure 9 shows the throughput and cwnd histories for Vegas with a 5ms RTT. We observe that in terms of throughput, it compares well with Reno, both before and after the introduction of additional flows. Can we understand why Vegas does not keep reducing cwnd? If we calculate the value of minRTT that is the threshold for increasing

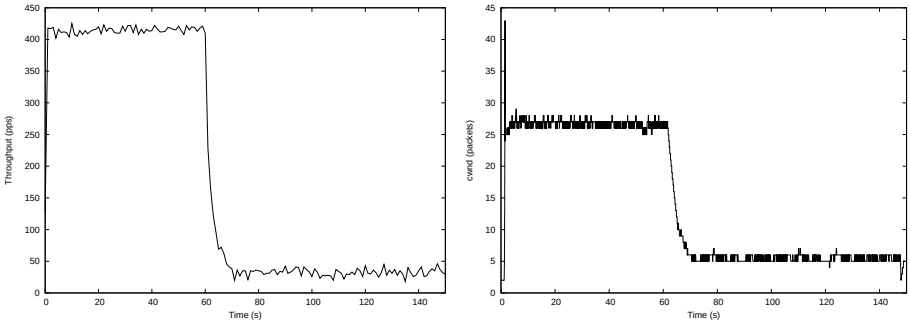

Fig. 9. TCP Vegas wth 5ms additional RTT in an initially uncontended network with additional fbws introduced at 60s: (a) throughput, (b) cwnd.

cwnd, we find that:

$$\text{minRTT} = \frac{\text{baseRTT}}{1 - \frac{\alpha}{\text{cwnd}}}. \tag{4}$$

The upper threshold for decreasing cwnd is the same, but with β instead of α. When cwnd is small, the band for maintaining or increasing cwnd becomes larger. Thus, when cwnd becomes smaller it can accommodate increased variability. Of course, it is possible that Vegas might decrease cwnd below the bandwidth-delay product before this safety net comes into play.

The throughput results for Vegas with a 50ms RTT are are broadly similar to to those with a 5ms RTT. Cwnd values being substantially larger during the first 60s and comparable after additional stations are introduced. One noticeable difference is that Vegas is maintaining a longer and more variable queue as a result of the larger RTT.

Figure 10 shows results for 200ms RTT, and we see Vegas behaving in a different way, where it begins to experience losses even when not competing with other stations. This can happen when Vegas's cwnd stabilised, but the actual queue length is fluctuating due to the random service. At 200ms the fluctuations are large enough that packets can be lost, resulting in Vegas reverting to Reno until it reenters congestion avoidance. This results in a resetting of the baseRTT, allowing Vegas to recover when new flows are introduced.

7 Conclusion

In this paper we have studied a number of interesting problems that we face when we aim to infer buffer occupancy from RTT signals in a WiFi network. We have seen that the raw RTT signal is correlated with buffer occupancy, but there is significant noise that grows as buffer occupancy increases. Standard smoothing filters do not seem to improve our prospects of estimating buffer size. We have also seen that traffic that does not share a buffer with our traffic may have a significant impact on the RTT measurements, possibly creating problems for estimation of queue length under changing

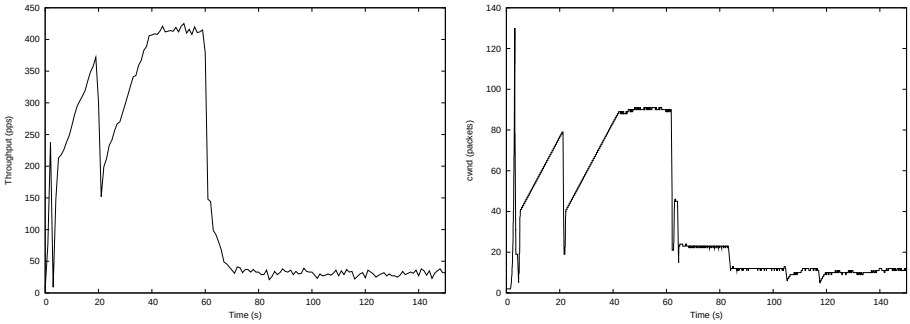

Fig. 10. TCP Vegas with an additional 200ms RTT in an initially uncontended network with additional fbws introduced at 60s: (a) throughput, (b) cwnd.

network conditions. We have briefly looked at the implications of these observations for Linux's TCP Vegas implementation. While Vegas performs well in our simple tests, we have possible weaknesses when faced with WiFi based RTTs.

References

1. Multiband Atheros driver for WiFi (MADWiFi). `http://sourceforge.net/projects/madwifi/`. r1645 version.
2. Soekris engineering. `http://www.soekris.com/`.
3. L. Brakmo and L. Peterson. Tcp vegas: End to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communication*, 13(8):1465–1480, October 1995.
4. I Dangerfi eld, D Malone, and DJ Leith. Experimental evaluation of 802.11e EDCA for enhanced voice over WLAN performance. In *Proc. WiNMee*, 2006.
5. IEEE. *Wirless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, IEEE std 802.11-1997 edition, 1997.
6. V. Jacobson. pathchar - a tool to infer characteristics of internet paths. MSRI, April 1997.
7. G. McCullagh. Exploring delay-based tcp congestion control. Masters Thesis, 2008.
8. ACH Ng, D Malone, and DJ Leith. Experimental evaluation of TCP performance and fairness in an 802.11e test-bed. In *ACM SIGCOMM Workshops*, 2005.
9. L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM/SIGCOMM Computer Communication Review*, 27(1), 1997.
10. K. Tan, J. Song, Q. Zhang, and M. Sridharan. A compound tcp approach for high-speed and long distance networks. In *INFOCOM*, 2006.
11. D.X. Wei, C. Jin, S.H. Low, and S. Hegde. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking*, 14:1246–1259, December 2006.