

On the Implementation of a Capacity Estimator for Wireless Ad Hoc Networks

Andrey Shipalov, Cesar D. Guerrero, Miguel A. Labrador, and Marco Alzate
University of South Florida
Department of Computer Science and Engineering
Tampa, Florida 33620
{ashipalo, cguerrer, labrador, malzate}@cse.usf.edu

Abstract—End-to-end capacity is a useful metric for network applications such as traffic engineering, QoS verification, peer-to-peer file distribution, video/audio streaming, admission control, etc. Although several tools have been developed to estimate end-to-end capacity and available bandwidth in wired networks, estimators for wireless ad-hoc networks are still to be developed. The paper briefly describes the model of a new capacity estimation tool for wireless ad-hoc networks. Then, it discusses two critical aspects in the implementation of any estimation tool: time stamping packets and clock synchronization. The paper details how to time stamp packets under the Linux and Windows operating systems, and describes a regression-based mechanism that eliminates measurement errors due to clock drifts. The time stamping methods and the regression algorithm are implemented in the capacity estimator, which shows accurate capacity estimations.

I. INTRODUCTION

Bandwidth estimation can be useful and benefit different areas of networking such as traffic engineering, QoS verification, peer-to-peer file distribution, video/audio streaming, and admission control, among others. The estimation of the end-to-end path capacity or maximum bandwidth has been studied extensively in wired networks. As a result, several tools are already available, such as Pathrate [1], Sprobe [2], and CapProbe [3].

While the end-to-end bandwidth (BW) in wired networks is conveniently a constant value, that is not the case in wireless ad-hoc networks. As it is shown in [4], capacity in wireless networks is a variable value determined by the maximum rate that can be achieved when the resources along the path are not shared among competing traffic flows. This random nature of the bandwidth makes the path capacity estimation in wireless networks and its implementation in software a challenging task. This paper first presents the model of a capacity estimation tool for wireless networks. The model is based on the bandwidth definition and estimation method proposed by [4] where it is shown that bandwidth in wireless networks is a time-varying variable distributed within some range, determined by the packet length. Then, the paper explains two important implementation issues common to all estimation tools based on active measurements: packet time stamping and clock synchronization. The paper describes how to time stamp packets under two different operating systems and how to deal with the clock synchronization problem, two issues that, if not handled correctly, may introduce important errors in the

estimation. Finally, the paper includes measurement results to show that the proposed methods produce accurate estimations.

The rest of the paper is organized as follows. In Section II we describe the capacity estimation work that preceded and inspired the development of the tool. Section III covers the methodology and the software implementation of the tool. In Section IV we show the main implementation issues encountered when implementing the tool. Section V presents the results of the experiments made with the developed tool. Finally, Section VI summarizes the paper.

II. CAPACITY ESTIMATION

In past years, many tools and methods have been proposed and developed for the estimation of the capacity of an end-to-end path. Nevertheless, most of them are focused on capacity estimation in wired and last-hop wireless networks, and do not produce accurate results when used in ad hoc wireless networks. AdHoc Probe [5] [6] is the only available tool specifically designed for end-to-end capacity estimation in wireless mobile ad hoc networks. AdHoc Probe is based on CapProbe [3] estimation technique. CapProbe is based on the observation that a packet pair that produces either an over-estimation or an under-estimation of capacity must have suffered cross-traffic induced queuing at some link. Therefore, in order to produce accurate results, CapProbe filters out the packets with the delay sum that is bigger than the minimum delay sum since the packets that suffer no cross-traffic induced delays reflect the correct capacity.

Even though AdHoc Probe uses the same principle of minimum delay sum, it differs from CapProbe in many significant ways in order to accommodate itself for wireless networks. One of the major differences of AdHoc Probe from CapProbe is that instead of using a round-trip estimation technique, AdHoc Probe makes one-way estimations. In other words, when packets arrive at the receiving node, those with minimum one-way delay sum are considered to be "good" packet-pair samples and used to calculate the corresponding capacity, which is given by:

$$C = \frac{P}{T} \quad (1)$$

where P is the packet size and T is the dispersion of the packet pair [5].

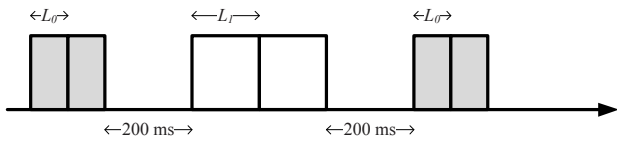


Fig. 1. Probing traffic pattern.

While AdHoc Probe has been shown to be a useful and practical tool for wireless networks, it does not consider bandwidth in wireless networks as a time-varying parameter. That variable nature is studied in [4] where the authors present an accurate method that is independent of cross-traffic and reacts timely to bandwidth variations. It is shown that a single packet of length L experiences a bandwidth of the form:

$$BW = \frac{L}{\alpha \cdot L + \beta} \quad (2)$$

where $\alpha \cdot L + \beta$ is the time it took the packet to traverse the narrowest channel in the path. Equation 2 has two unknown parameters: α , which is the cost, in seconds per bit, of transmitting one bit, and β , which is the average overhead cost [7]. Since the capacity depends on the packet length L , the the method proposed in [4] utilizes packet-pairs of two different sizes and produces results in the form of time-varying variable distributed over some range. Moreover, instead of simply calculating one-way delay (OWD), the average of OWDs is used in the estimation.

III. ESTIMATION TOOL

The estimation tool has to determine for a given L , the unknown values of α and β to provide a single estimation according to Equation 2. This is done by probing the wireless network with a carefully designed sampling method implemented in a client-server application.

A. Sampling method

To perform the estimation the network is sampled by sending a sequence of packet pairs as shown in Figure 1. This set of four packets is called a group. A group is defined as two pairs of two packets. One pair of L_0 -byte packets and the other pair of L_1 -byte packets. In order to have several samples, a good number of groups are to be sent to the network by a sender host.

Before transmission, each packet is stamped with a sequence number and its departure time. In addition, the last packet in the sequence contains a predefined number to indicate that it is the last packet in a probing traffic. When the last packet reaches the receiver, all acquired information about a packet including its sequence number, departure time, and arrival time are used to calculate the one-way delay (OWD), sum of one-way delays (SOWDs), and gap (the dispersion time between packets). Specifically, as it is shown in Figure 2, the receiver calculates OWD as the difference between arrival and departure times of a packet, and adds OWDs of two packets of the same pair to get SOWD for that particular pair.

If each packet of a pair of L_0 and L_1 bits suffers no queuing, retransmission or scheduling delays, then their SOWD would

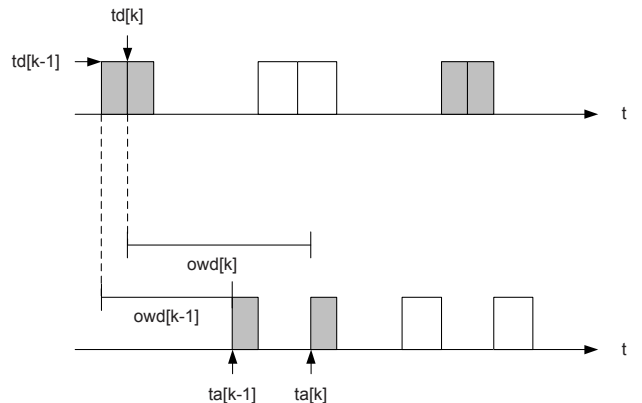


Fig. 2. One way delay.

be minimum and the gap between them will reveal a sample of the transmission time $\alpha \cdot L + \beta$. In addition, if the network is not heavily loaded, the probability of having such a pair will be greater than zero, so it will be at least one valid measure with high probability given that several pairs are sent. A linear regression of all the SOWD calculated values provides a better approximation of the desired minimum SOWD.

Using the average of the gaps that correspond to the pairs with the minimum sum of one-way delays (SOWDmin), the values of α and β can be calculated as follows:

$$\alpha = \frac{av_gap_{L_1} - av_gap_{L_0}}{L_1 \cdot 8} \quad (3)$$

$$\beta = av_gap_{L_0} - \alpha \cdot L_0 \cdot 8 \quad (4)$$

In Equations 3 and 4, $av_gap_{L_0}$ and $av_gap_{L_1}$ are the corresponding selected average gaps for the packets of size L_0 and L_1 , respectively.

B. Tool implementation

Based on the previous sampling mechanism, a tool to measure and estimate the end-to-end capacity in a wireless mobile ad hoc network was implemented. The tool consists of two applications written in the C programming language, one running at the transmitter and one at the receiver. The tool has been implemented under both Linux and Windows operating systems.

1) *Transmitter*: The transmitter performs the simple operation of sending 250 groups of UDP packets. The reason to send that number of packets (1000 since each group has 4 individual packets) is that many of them can be lost because of the wireless transmission vulnerable to packet loss or because of the UDP-nature of the probing packets. By sending 1000 packets, it is also supported the non-intrusive nature of the program and because of the way packets are processed at the receiver, it is guaranteed enough data for calculations.

The first pair in a group has two packets of 100 bytes each, and the second pair is represented by two 1400-byte packets. Each packet before transmission is stamped with a sequence number and its departure time. In addition, the last packet in the sequence contains a predefined number to indicate that it is the last packet in a probing traffic.

2) *Receiver*: The receiver application processes the received packets in a window of 400 packets in a first run and then overlaps every twenty packets. This is to have enough non-intrusive data for calculations. After extracting and storing data from the received packets, the application calculates α and β parameters. When the last packet in the probing traffic has been received, all acquired information is used to calculate the SOWD and gap for each packet in the window. However, only the gaps corresponding to the packets with SOWDmin are used in final calculations. Gaps and SOWD are calculated as follow:

```
sowd=(arr_time[1st_pck]-dep_time[1st_pck])
      +(arr_time[2nd_pck]-dep_time[2nd_pck]);
```

```
gap=arr_time[2nd_pck]-arr_time[1st_pck];
```

Given that $L_0 = 100$ bytes and $L_1 = 1400$ bytes, according to Equations 3 and 4, the estimation of the unknown parameters in Equation 2 is given by:

```
alpha=(av_gap_1400-av_gap_100)/(1400*8);
beta=av_gap_100-alpha*(100*8);
```

Using the calculated values for α and β and Equation 2, the receiver presents a timely estimate of the bandwidth observed by a packet of length L_0 and a packet of length L_1 :

```
// BW for L=100 B
bw_100 = (100*8)/av_gap_100;
// BW for L=1400 B
bw_1400 = (1400*8)/av_gap_1400;
```

It is worth noticing that in order for this procedure to work, the receiver also needs to time stamp each packet upon its arrival.

IV. IMPLEMENTATION ISSUES

The use of time stamps brings two issues of utmost importance for the correct estimation of the end-to-end capacity. First, the time stamping procedure has to be able to work at an appropriate clock granularity, otherwise events that took place at different time instances could appear as if they happened at the same time. For example, if the computers' clock granularity is coarser than the interarrival time between the packet pairs, both packets will leave the sender with the same time stamp. Second, working with time stamps always brings the issue of clock synchronization. Since the transmitter and receiver applications run on different computers, their clocks are not necessarily well synchronized. Actually, this will be the rare case. In the following, we explain how to address these two important aspects.

A. Time stamps

As said before, the clock granularity and the way time values are obtained from the system constitute critical issues, and vary according to the operating system. For example, Table I shows some of the functions that are used to obtain the time of the system in Linux and Windows. However, besides the difference in resolution, which varies from seconds to

Timing Function	Operating system	Resolution
GetSystemTime()	Windows	milliseconds
GetTickCount()	Windows	milliseconds
timeGetTime()	Windows	milliseconds
QueryPerformanceCounter() QueryPerformanceFrequency()	Windows	nanoseconds
gettimeofday()	Linux	microseconds
time()	Windows/Linux	seconds
clock()	Windows/Linux	seconds
RDTSC*	Windows/Linux	microseconds

TABLE I
FUNCTIONS USED FOR TIME ACQUISITION (*INSTRUCTION NOT A FUNCTION).

nanoseconds, those functions have other drawbacks that have to be taken into consideration.

The resolution of seconds, milliseconds, etc. does not mean that timing will be accurate up to 1 second, 1 millisecond, etc. For example, the `GetTickCount()` function, which retrieves the number of milliseconds, can return results from approximately 10 to 55 milliseconds. Such big range of possible values is system-dependent since different versions of operating systems with the same or different hardware will result in different resolutions for many, if not all, functions.

In addition, besides discrepancies in resolution on different systems, many of the functions offered by both operating systems cannot be used to time stamp the packets. Specifically, given that the transmission time of 100-byte packet is approximately 72 microseconds in a 11 Mbps link, the time query function has to have a precision in that order of microseconds. Therefore, from the Table I, it can be seen that this granularity, in the case of the capacity estimation tool presented in this paper, might be achieved only by using the `QueryPerformanceCounter()` and `QueryPerformanceFrequency()` functions in the case of Windows and the `gettimeofday()` function in the case of Linux. The RDTSC instruction can also be used and timestamps can also be performed by the kernel.

1) *Windows*: if windows is the operating system where the application will be run, the following functions are used:

```
BOOL QueryPerformanceCounter (
LARGE_INTEGER *lpPerformanceCount );
```

```
BOOL QueryPerformanceFrequency (
LARGE_INTEGER *lpFrequency );
```

where `LARGE_INTEGER` is defined as

```
typedef union _LARGE_INTEGER {
    struct {
        DWORD LowPart;
        LONG HighPart;
    };
    struct {
        DWORD LowPart;
        LONG HighPart;
    } u;
    LONGLONG QuadPart;
} LARGE_INTEGER, *PLARGE_INTEGER;
```

```

while(!lastPacket)
{
    numBytes = recvfrom(sender, packet);
    packetNumber[i] = packet.seq_number;
    // get departure and arrival times
    depTime[i] = packet.departure;
    arrTime[i] = timestamp(packet);
    lastPacket = packet.last;
    i++;
}
Process_packets();

```

Fig. 3. Receiver application loop.

Modern CPUs have a high speed counter, which increments at a rate of billions per second. The QueryPerformance function calls allow to read the values of this counter. The minimal resolution of a system is determined by $1/\text{frequency}$ where frequency can be retrieved using the QueryPerformanceFrequency function. This function retrieves the frequency of the high-resolution performance counter, if one exists, in counts per second. The QueryPerformanceCounter() function works in a similar way except that it retrieves the current value of the high-resolution performance counter in counts. Consequently, to time stamp the packets with a microsecond granularity, the number of counts is divided by the frequency. In addition, the number of counts and frequency values must be cast to doubles before the division, otherwise the division will be done as 64-bit division with no fractional part.

However, although the resolution achieved with the QueryPerformance functions can be in the order of nanoseconds, the actual minimal resolution is dependent on hardware. Therefore, it will vary from system to system. Another aspect that has to be taken into consideration when working with any timing function is the overhead of calling these functions since it might affect final results, especially when taken together with processing time, as discussed later. On the other hand, these functions usually adjust for multiple processors.

2) *Linux*: in this case, the following function is used:

```

int gettimeofday(struct timeval *tv,
    struct timezone *tz);

```

Although get time of day takes two parameters, in order to retrieve the current time of the system only one of them is needed. Therefore, the timezone structure is replaced by NULL. The function obtains the current time in seconds and microseconds, which are stored in the timeval structure and represented by tv_sec and tv_usec, respectively.

3) *RDTSC*: the next possibility for time stamping is the RDTSC instruction. RDTSC is the assembly instruction that accesses the Time-Stamp Counter (TSC), which is a 64-bit register that is incremented every clock cycle. In addition, the TSC measures cycles and not the actual time. Therefore, as for the QueryPerformanceCounter function, the value retrieved by RDTSC must also be divided by frequency. Although RDTSC is faster than QueryPerformanceCounter(),

and it was a common way for performance monitoring in previous years, today its reliability suffers from several problems. One of the problems is that some processors support out-of-order execution, meaning that instructions are not necessarily executed in the order that they appear in the code. This can be a serious problem for RDTSC instruction since it can be executed before or after its actual location in the code thereby producing misleading results. In addition, with the advent of multiprocessor, dual-core systems and power management technologies, the reliability of the results produced by the RDTSC instruction suffers even more.

Another problem is that the RDTSC instruction assumes that the thread is running on the same processor. However, multiprocessor and dual-core systems do not guarantee synchronization of cycle counters between cores. Additionally, power management technologies might put one of the cores to sleep or completely stopping it, thereby resetting the counter.

4) *Kernel Timestamps*: timestamps can be performed at the receiver's kernel as soon as the packet is seen by the operating system. That is, timestamps are not performed by the functions called from the application but is registered by the kernel to the application. This is done by setting the SO_TIMESTAMP option in the socket:

```

one = 1;
...
setsockopt(socket, SOL_SOCKET,
    SO_TIMESTAMP, &one, sizeof(one))

```

Then the packet can be get from the sender:

```

struct msghdr rcv_msg;
struct timeval rtm;
char ctrl[MSG_SPACE(sizeof(rtm))];
struct cmsghdr *rmsg=(struct cmsghdr*)
    &ctrl;
...
rcv_msg_size=recvmsg(socket,&rcv_msg, 0)

```

And finally, the kernel timestamp of the packet is stored in the received time variable rtm and if anything goes wrong, any of the functions shown in Table I can still be applied:

```

if (rmsg->cmsgh_level==SOL_SOCKET &&
    rmsg->cmsgh_type==SCM_TIMESTAMP &&
    rmsg->cmsgh_len==MSG_LEN(sizeof(rtm)))
{
    memcpy(&rtm,MSG_DATA(rmsg),sizeof(rtm));
} else {
    gettimeofday(&rtm, NULL);
}

```

B. Clock synchronization

The calculation of One Way Delays is problematic in real networks since for accurate results it is required that the system clocks of the two end hosts be synchronized, which is not usually the case. One possibility that might occur is that departure time of a packet, as interpreted at the receiving node, is bigger than the arrival time. As a result, unsynchronized clocks can lead to significant over or under estimation of

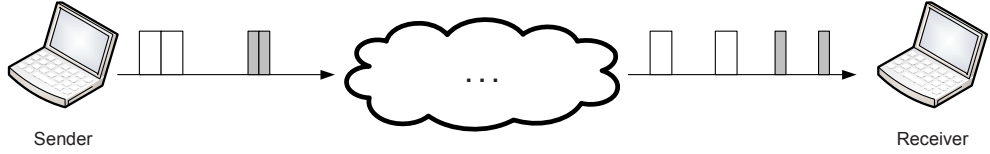


Fig. 4. Simple wireless ad hoc connection to measure capacity.

bandwidth even if the time stamps are acquired accurately at the individual host and represent the actual departure and arrival times of a packet. Thus, a successful bandwidth estimation technique must not rely on any assumptions of a perfectly synchronized system. In our tool implementation, a linear regression technique has been implemented to correct the difference between phase and frequency in the clocks of the sender and receiver hosts. Specifically, it helps to correct the minimum Sum of One Way Delays as follows:

```

For all k (pairs) and j (sizes)
{
    minSowd[0]=sowd[k][j]; //minimum sowd
    sowd[k][j]-=minSowd[0]; //avoid overflow
    // Variables for the regression
    y[k]=sowd[k][j]-sowd[0][0];
    // d_gap: departute packet pair gap
    x[k]=d_gap[k][j]-d_gap[0][0];
    // Linear Regression Coeficient:
    p=(XY-sum(X)*sum(Y))/(
        (sum(X^2)-sum(X)*sum(X)));

    // Correct sowd
    sowd[k][j] = y[k]-p*x[k];
}

```

C. Processing time

The purpose of the sender application is to send pairs of packets back-to-back. However, before transmission of any packet there are some information that has to be stored in that packet. Setting the packet size and calculating or deciding what value to put in that packet (e.g. sequence number of a packet, departure time) requires some processing time. Moreover, the receiver part of the application needs additional time to extract all that information and store it. Thus, the time that it takes to process and store that information will be interpreted as delay at the receiver host in addition to its own setback. Therefore, the processing time in the main loop of the receiver and transmitter applications must be minimized in order to avoid incorrect timing. In our tool, the information is stored temporarily inside the main loops to be processed outside them, as it is shown in Figure 3.

V. TOOL EVALUATION

The developed tool was evaluated using the simple ad-hoc configuration shown in Figure 4 with transmission rates set to a maximum of 11 Mbps. The capacity is measured from sender

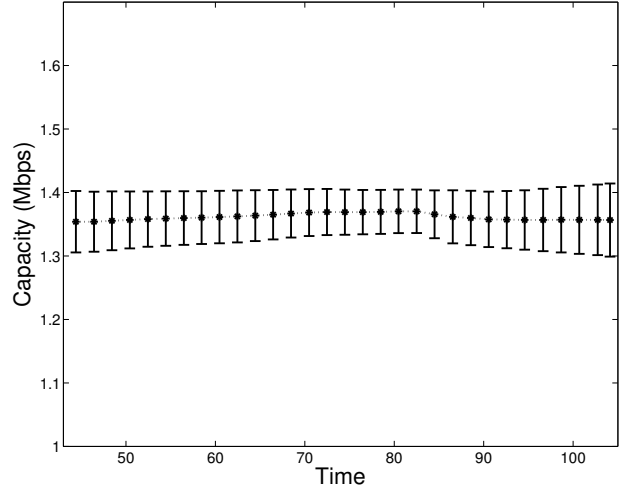


Fig. 5. Capacity seen by 100-byte probing packets.

to receiver using both 100 and 1400-Byte probing packet sizes. In order to have statistically significant results, each estimation is performed 10 times and after using the t-student distribution, a 95% confidence interval is calculated as follows:

$$\Delta_C = \frac{1.812 \times S_X}{\sqrt{10}} \quad (5)$$

where S_X is the standard deviation of the estimations X .

Figure 5 and 6 show the capacity estimation when using packet pairs of 100 and 1400 bytes, respectively. From the figures, it can be seen that the capacity reported by the tool is close to 5 Mbps for the case of 1400-byte packet pairs, which matches several results reported in the literature. In fact, it is well known that for 802.11b the expected data throughput is close to 5.5 Mbps rather than the signaling rate of 11 Mbps. Similarly, the expected throughput of an 802.11a or 802.11g network is close to 30 Mbps rather than the signaling rate of 54 Mbps [8]. These results validates the operation of the tool and the appropriateness of the time stamping and clock synchronization procedures described in the paper.

Table II presents the average results over time for the capacity seen by 100 and 1400-byte packets. After calculating the 95% confidence interval is is shown more variability by the packet with the larger size.

VI. CONCLUSIONS

This paper presents the model and implementation of a capacity estimation tool for wireless ad-hoc networks and discusses the main aspect to be considered when implementing a network capacity estimation tool. The paper shows how to

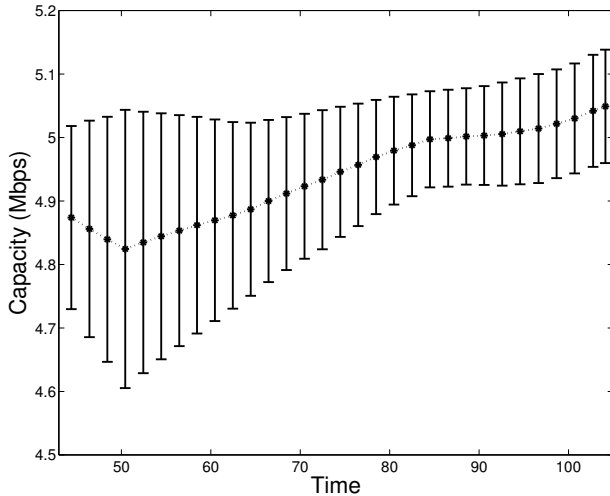


Fig. 6. Capacity seen by 1400-byte probing packets.

Probing Packet Size	Averages (Mbps)
100 Bytes	1.36 ± 0.043
1400 Bytes	4.94 ± 0.121

TABLE II
AVERAGE CAPACITY AND 95% CONFIDENCE INTERVALS AS SEEN BY 100
AND 1400-BYTE PROBING PACKETS.

overcome time stamping, clock synchronization, and processing time issues, which are the most critical problems that most capacity estimation applications face. The results of implementing the capacity estimation tool over wireless ad-hoc networks validate the correct operation of the tool.

VII. ACKNOWLEDGMENTS

This work has been supported by the National Science Foundation under grant No. 0453463.

REFERENCES

- [1] C. Dovrolis, P. Ramanathan, and D. Moore, "Packet-dispersion techniques and a capacity-estimation methodology," *IEEE/ACM Transactions on Networking*, vol. 12, no. 6, pp. 963–977, 2004.
- [2] S. Saroiu, P. K. Gummadi, and S. D. Gribble, "SProbe: A fast technique for measuring bottleneck bandwidth in uncooperative environments," in *Proceedings of IEEE Infocom*, 2002.
- [3] R. Kapoor, L.-J. Chen, L. Lao, M. Gerla, and M. Y. Sanadidi, "CapProbe: a simple and accurate capacity estimation technique," *SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 67–78, 2004.
- [4] M. Alzate, M. Salamanca, N. Pena, and M. Labrador, "End-to-end mean bandwidth estimation as a function of packet length in mobile ad hoc networks," in *Proceedings of the IEEE Symposium on Computers and Communications*, Aveiro, 2007, pp. 415–420.
- [5] L.-J. Chen, T. Sun, G. Yang, M. Y. Sanadidi, and M. Gerla, "Ad Hoc Probe: Path capacity probing in wireless ad hoc networks," in *Proceedings of the First International Conference on Wireless Internet (WICON'05)*. Budapest, Hungary: IEEE Computer Society, 2005, pp. 156–163.
- [6] T. Sun, G. Yang, L.-J. Chen, M. Y. Sanadidi, and M. Gerla, "A measurement study of path capacity in 802.11b based wireless networks," in *Proceedings of the 2005 workshop on Wireless traffic measurements and modeling*. Seattle, Washington: USENIX Association, 2005, pp. 31–37.
- [7] M. Alzate, J. Pagan, N. Pena, and M. Labrador, "End-to-end bandwidth and available bandwidth estimation in multi-hop IEEE 802.11b ad hoc networks," in *Proceedings of the 42nd Conference on Information Sciences and Systems*, Princeton, USA, 2008, pp. 659–664.
- [8] Y. Solomon, "Defining and improving data throughput in wireless LAN," *Texas Instruments White Paper*, 2003.