Exploring Embedded Path Capacity Estimation in TCP Receiver

Cesar Marcondes. M. Y. Sanadidi and Mario Gerla
Computer Science Department
University of California (UCLA)
Los Angeles, USA
Email: cesar,medy,gerla@cs.ucla.edu

Magnos Martinello and Ramon S. Schwartz
Computer Science Department
Universidade Federal do Espírito Santo (UFES)
Vitoria, Brazil
Email: magnos,ramon@inf.ufes.br

***Abstract*— Accurate estimation of network characteristics, such as capacity, based on non-intrusive measurements is a fundamental desire of several applications. For instance, P2P applications that build overlay networks can use path capacity for optimizing network resources. The purpose of this work is to present a simple technique to estimate end-to-end Internet paths capacity. Our basic idea relies on the key observation that path capacity estimations can be obtained simply by making adequate inferences on the TCP receiver behavior. The proposed technique allows that measurements can be executed in the Internet with no requirement of having access and permission to remote machines. An off-the-shelf linux kernel is used to implement the method and provide precise measurements. In addition, a large number of experiments is collected in a high speed measurement point in order to i) validate, ii) show the accuracy and iii) explore the path capacity heterogeneity of some Internet paths. For applicability purposes, a testbed is used for two developed applications. The first application improves TCP throughput using the receiver advertised window. The second is designed to find the exact location of the narrow link (i.e. bottleneck) on an Internet path.**

Keywords: Measurements and Monitoring, Network peformance evaluation, Inline TCP.

I. Introduction

Path characteristics estimation is a fundamental desire of several applications. For instance, streaming applications using a peer-to-peer backbone (such as Skype, P2PLive) have strict demands on end-to-end performance requirements. Therefore, the knowledge of capacity, delay, jitter and loss of packets in Internet paths help on calibrating the expectations of such applications.

In order to understand the path characteristics that ensure the quality of service required by applications, measurements have proved to be an attractive doctrine on Internet research. In particular, knowing the capacity of Internet paths allows not only to optimize network resources, but also to design balancing policies according to path capacities.

Active measurements techniques is one way to perform such measurements. It has as basic principle the sending of intrusive probes from source(s) to destination(s) and posterior processing. The types of metrics obtained by this method can be either *one-way path round-trip path* characteristics. The metrics depend on the location of the measurements, assumptions of synchronization and total control of both sides. On the other hand, measurements can be performed in passive [10] way which relies on traces to infer characteristics of a path. A third mixed technique can be achieved by inferring characteristics through inspection of the useful traffic pattern only, without inserting any probes on the path.

The purpose of our work is to present a simple technique to estimate end-to-end Internet paths capacity, using a mixed methodology (i.e. without sending probes while receiving useful data). The core idea relies on the observation that path capacity estimations can be obtained simply by making adequate inferences on the TCP traditional behavior, more precisely, by measuring packet interarrival dispersions during the slow start phase. If the packets are sent back-to-back (packet pairs), such dispersion is proportional to the minimum transmission delay along the path. Equivalently, the packet pair dispersion could represent the narrow link path capacity.

The proposed technique elaborates on the receiver side of a TCP connection, allowing measurements to be executed in the Internet with no requirement of having access and/or permission to remote machines (i.e. Internet file servers). In addition, since the measurement of packets interarrival can produce either over-estimation or under-estimation of path capacity due to cross-traffic at some link [6], our technique requires that a certain packet pair interarrival have the minimal *round-trip* among all packet pairs. This minimal round-trip reduces the likelihood of a packet pair measured to have been impacted by cross-traffic.

A considerable number of experiments is executed in order to measure a subset of Internet path capacities, especially the ones formed by open source file servers. This path capacity perspective is based on a receiver side TCP located on a single central measurement point. For illustration purposes, the measurement point is used as a testbed for two developed applications. The first application improves TCP throughput using the receiver advertised window. The receiver regulates its window size based on measured capacity information. That is, it effectively limits the amount of traffic in flight reducing the usage of buffer space on the bottleneck. The second application involves a combination of traceroute and monarch [5]. The goal is to find the exact location of the narrow link (i.e. bottleneck) on an Internet path.

The rest of this paper is structured as follows. In section 2, our attention is first devoted on related techniques for capacity estimations, then we focus on the proposed TCP inference algorithms. Section 3 describes the Linux Kernel instrumentation presenting the state diagrams and pseudo-codes of the proposed algorithms. In Section 4, a large number of experiments is presented in order to i) validate and compare the performance of the TCP inference algorithms, ii) show the accuracy of the Internet capacity estimations and iii) explore the heterogeneity of the Internet paths. Section 5 presents two applications making use of the embedded TCP receiver feature and Section 6 concludes the paper.

II. Capacity Estimations Techniques

One of the earliest methods to estimate path capacity was described in [1]. The method called bprobe, relies on the idea that if two packets are travelling together, they are to be queued as a pair at the bottleneck link, then the inter-packet spacing will be proportional to the service time of the bottleneck. This work presented an early version of the packet pair techniques.

Later, in [7] the authors suggested a robust capacity estimation technique called PBM (Packet Bunch Sizes). PBM technique works by stepping through an increasing series of packet bunch sizes[1]. For each sample, the bottleneck estimation is computed based on the receiver trace. After the bottleneck distribution is constructed, the final estimated value is obtained by finding the maximum value in the density function. If two modes are similar and sufficiently separated, it suggests a change in the service rate of the bottleneck.

The meaning of the multiple modes in packet pair dispersions and packet trains was elaborated further in [2]. They showed that the strongest mode in the multimodal distribution may not correspond exactly to the path capacity, but to an under or over-estimated capacity. In this study, they presented a capacity estimation methodology (Pathrate), which uses many packet pairs (with packets of variable size) to uncover a set of possible "capacity modes". Then, as a second phase, they use long packet trains to estimate the so called "Asymptotic Dispersion Rate" or ADR. ADR corresponds to a measure of the average statistical multiplexing of the path. The ADR provides a hint about which capacity local modes to reject. Finally, Pathrate chooses the mode that has the strongest and narrowest mode from the non-rejected ones.

There are other techniques not based on dispersion of packet pairs. In the work [3], a pathchar tool is proposed using the variation of the round-trip delay as the packet size increases. This technique, based on the generation of ICMP replies from routers, is known to have scalability and accuracy problems. In fact, it tries to estimate the capacity of every link on the path in order to estimate the end-to-end path's capacity implying a high overhead.

Although these techniques are well-know and largely used, we can observe several *limitations*. First, some of them require the execution of measurement code at both ends of the measured path [2] . This constraint limits the applicability of these tools in just a few paths where the user has access at both the sender and the receiver. Second, some of the tools depend on ICMP probing packets. Such traffic is often blocked or handled in different processing path than TCP traffic. Third, the time to converge to an accurate metric is also a drawback since several of the methods require a thorough statistical analysis in order to ensure a reliable capacity estimate, taking in worst cases several minutes. And at last, all of them are active measurement tools requiring extra-traffic to be inserted in the network.

CapProbe [6] tackles the convergence speed problem by filtering out packet pairs according to a simple rule: "packet pairs with minimal end-to-end delays are sufficient to estimate consistently a narrow link capacity". This way, it rules out packet pairs impacted by cross-traffic, and no posterior statistical analysis is necessary to obtain an accurate estimate, turning the technique into the fastest estimator.

The basic ideas incorporated in CapProbe can be observed in the following Figure 1. A packet pair is sent from source to destination, and the serialization delay of the packets change according to the bottleneck speeds. As the packet pair passes the narrow link, the dispersion does not change anymore and it is preserved throughout the path until the destination. The destination then, can either filters out packet pairs according to one-way delay or it can echo the dispersed packet pair back such that the sender could also do filtering. The final dispersion of the minimum delayed packet pair represents the end-to-end narrow link capacity and it can be calculated as follows: $(Capacity(C) = \frac{PacketSize(L)}{PairDispersion(T_b)})$.

[1]Packet bunch mean back-to-back packets with a size greater or equal to two.

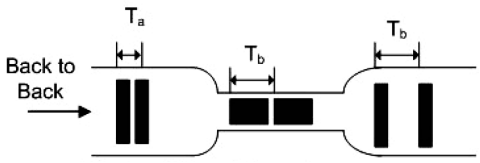


Fig. 1. Capprobe Overview

However, if a measurement point is collecting samples from a receiver side perspective, then Capprobe technique allows to measure the *backward path* capacity, i.e., the path capacity from the sender to the receiver. In this way, the evaluated measure is the Round Trip Time (RTT) instead of delay. Thus, an interesting aspect is how to determine when the minimum sum in a set of packet pair samples is actually equal to the non-queuing (minimum) RTT sum. Clearly, the higher the congestion, the lower is the probability of obtaining a minimum RTT sum sample.

Consider the set of packet pair samples, $i = 1, 2, ...$, let R_1^i and R_2^i denote the RTT's of the first and second packets of the ith sample. Let the minimum RTT sample be the jth sample. Thus, $min\{R_1 + R_2\}$ occurs at the jth sample.

Now consider the minimum among the RTT's of the first packet of all packet pair samples, i.e., $min\{R_1\}$. Let $min\{R_2\}$ denote the corresponding quantity for the second packet.

In the set of packet pair samples, it is not necessary that $min\{R_1 + R_2\} = min\{R_1\} + min\{R_2\}$. In other words, it is not necessary for the minimum RTT sum to be equal to the sum of the minimum RTT's. However, it could happen, for instance, that the RTT of the first or (the second) packet in the minimum RTT sum is greater than the minimum RTT of the first or (the second) packet among all samples. In this case, $min\{R_1 + R_2\} > min\{R_1\} + min\{R_2\}$, then we can deduce the following:

- This packet suffered some queuing delay;
- The dispersion obtained from the minimum sum could have been a distorted one;

We refer to this relation as the *minimum sum* condition and it helps to filter incorrect samples.

Then, the question that raises and main motivation of this work is: **"Could we embed a path capacity estimation within TCP, with minimum changes, relying on useful traffic while downloading some webpage from the Internet?"** This way, in spite of the problems (non-cooperation, ICMP inaccuracy, convergence, extra probing packets), TCP naturally could overcome these limitations.

A. TCP Inference Algorithms for Capacity Estimations

Our approach to develop TCP inference algorithms consists on identifying pairs of data packets sent "back to back" from the sender to the receiver. Clearly, the moments when those packet pairs were sent can be identified during the beginning of the slow start phase.

In this phase, for every Acknowledgment sent from the TCP receiver side, it triggers naturally in the sender side, the transmission of a packet pair "back to back". Our TCP inference algorithm uses this normal behavior and inspired on Capprobe, it runs on the receiver side playing the role of a passive estimator which aims to identify such packet pairs that do not suffer queuing along the path. This way, we do not change the TCP design and do not include any extra probes on this task.

A crucial aspect of TCP on obtaining unaffected packet pair samples during the slow-start is that at each Round Trip Time(RTT),

the sender is designed to send packets as long as the congestion window permits. Clearly, the amount of packets sent out on each RTT cycle increases turning the transmission of packet pairs into packet trains. Given this traditional pipe filling behavior of TCP, the problem of finding packet pairs that did not suffer queuing become harder.

At each RTT all the packets belonging to a RTT cycle are sent in trains. Therefore, these packets except the first and the second ones will be clearly queued, a behavior that we refer as **self-interference**. In other words, due to the inherent protocol design, the potential number of good packet pair samples is reduced in practice. In order to address such a problem of finding good samples related to the TCP design, we elaborate two inference algorithms. We refer to these algorithms as *short regulated rate algorithm* and *no regulated rate algorithm* .

Figure 2(a) describes the short regulated rate algorithm. The basic idea is to respond with just an accumulated ack to each pair of data packets, instead of sending two acknowledgments. Accumulated acks means to respond using the sequence number of the second packet for each received pair. This algorithm induces a regulation of the sender window size during the slow-start phase, imposing a period of path capacity sampling.

The induced regulation is explained by the *gaps* of Figure 2(a). It is important to note that each accumulated ack in a current cycle is sent i) if two data packets arrive into the current cycle, or ii) it should wait by the first data packet of the next cycle. In the case ii), there is a regulation of one RTT initially, but it decreases substantially as the cycles become close to each other. In this figure, we illustrate an example in which the receiver responds with an accumulated ack to each two data packets. In practice, our implementation allows the receiver to choose a constant setting the number of data packets to wait before sending an accumulated ack.

In fact, the regulation phase is designed to collect minimal RTT(s) samples in a very short time. The short period of time is based on the fact that 20 samples has been largely sufficient on Capprobe [6] to compute a correct estimated capacity. Doing a simple worst-case calculation, assuming an RTT of 50 milliseconds, the time taken to collect 20 samples would be 1 second, for an RTT of 100 millisencods it would take 2 seconds.

It is important to note that after this short phase, there is no more regulation and TCP will continue its slow-start phase increasing the window exponentially.

The second algorithm (Figure 2(b)) presents the no regulated rate. In this case, our aim is to identify the TCP RTT cycles based on the gaps between cycles. In contrast to the short regulated rate algorithm, the basic idea is to extract pairs at the beginning of each TCP RTT cycle, exactly the first and second packet of each phase. However, there is a difficulty related to the fact that gaps become harder to find as the window size increases. To clarify this point, recall that after e.g. the fifth cycle there will be $2^5 = 32$ packets being sent by the sender. Hence, as soon as the channel is filled up with packets, the beginning of the next cycle is sent just after the end of the current cycle. At this point, there is basically no visible gap after the sixth cycle, obviously because the window size is designed to increase exponentially.

The *no regulated rate algorithm* in contrast to the regulated one, is able to find 5 potential good samples most of the time. The advantage compared to the short regulation algorithm is that there is no regulation is needed to obtain good samples. However the capacity estimation is computed based on the results of these 5 samples compared to 20 samples of the *short regulated rate algorithm*. We explore the trade-off(s) between these algorithms described in details in the results section.

III. Kernel instrumentation

Our focus on kernel implementation is oriented to the instrumentation of the Linux 2.6.18 TCP module in which we add the new capacity estimation features. In addition, we aim for achieving high precision results in our experiments. The most important features were inserted directly into the *tcp_input.c* and *tcp_output.c* files concerning the receiver and sender machine state sides, respectively. The *tcp.h* file was also modified by inserting new data structures to collect packet pairs into a constrained vector.

It is important to note that the measured capacity is calculated simply dividing the size of the second packet by the dispersion between the packet pairs. Due to this reason, the interarrival time is crutial for a reasonable precision in packet pairs dispersion. The operating system timer can be used to measure events with an accuracy of 10 ms, depending on the kernel-heartbeat. However, the packets interarrival difference represents events happening in a granularity often much less than 10 ms.

The today's processors are running at frequencies of more than 1GHz, then we can use the instruction counter to time events with much higher accuracy than 10 ms and low overhead. Hence, we adopt the use of the RDTSC (Read Time Stamp Counter) register which returns a 64-bit value in the registers EDX:EAX representing the count of ticks from the processor. The RDTSC CPU instruction allows us to capture measures in order of magnitude of micro-seconds providing high precision for collected samples.

A. Kernel Implementation of TCP Inference algorithms

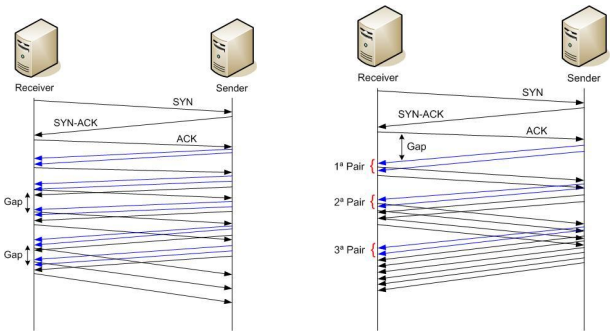
In the sequence, we present the main inference algorithms in a simplified state diagram. For each packet arrival, a transition from the state *waiting packet* occurs leading to the algorithm execution in both of Figures 3(a) and (b).

Figure 3(a) presents the short-regulated version. We can observe that the algorithm receives 2 packets ($num\_packet < 3$) to store the information of the packet pair into the vector. Simultaneously, a test is executed ($next\_ack < 2$) in order to identify when a pair arrives and after that the function send_ack() is called. Recall that the send of ACK is delayed essentially due to the wait of the first data packet in the next cycle as explained in section II-A. In practice, the implementation allows to choose a constant x representing the number of data packets to wait before sending an accumulated ack ($num\_packet \leq x$).

The no-regulated algorithm is showed in Figure 3(b) describing the basic state diagram. It captures the first_packet and second_packet of every RTT cycle by observing the expected slow start behavior. For a given RTT cycle, it counts the data packets to determine the beginning of the next cycle from the receiver side perspective. Every ACK will trigger 2 data packets ($wait\_pack = wait\_pack + 2$), which are used to find the first packet of the next cycle ($next\_first\_packet = next\_first\_packet + wait\_packet$). [2]

In the current implementation, we also take into account the send of delayed acks which can conduct to accumulated acks. Although the state diagram is a simplified version that assumes no accumulated ack, it provides a set of variables based on the effective implementation which supports the accounting of packets in a determined cycle. The current implementation provides a feature to deal dynamically

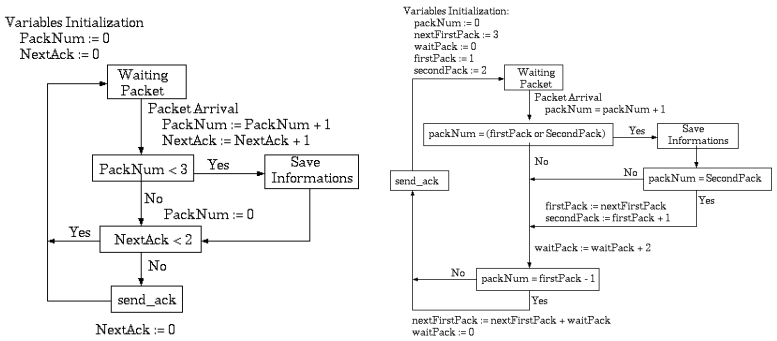
[2]In theory, we have expected to find a pair in the beginning just taking the 2^n and $2^n - 1$ data packets, where n is the sender window size. Unfortunately, the delayed acks phase affects this theorical behavior preventing it in practice.



(a) Short Regulated Rate Algorithm

(b) No Regulated Rate Algorithm

Fig. 2. Removing Packet Pair Self-Interference



(a) Short Regulated Rate Algorithm

(b) No Regulated Rate Algorithm

Fig. 3. State diagram for TCP inference algorithms

with accumulated acks. In fact, we use a variable n ($wait\_pack = wait\_pack + n$) to follow precisely the number of received packets. The variable n is computed by counting the number of received packets before one ack is sent.

The next algorithm is common for both implementations (short-regulated and no-regulated). Note that this part is triggered as soon as 20 packet pairs are received (ARRAY_SIZE). In the beginning, the algorithms finds the smallest sum of RTTs among all packet pairs. After that, the minimum RTT is found for the first packet subset as well as for the second packet subset. Finally the main IF-statement, differently than CapProbe, allows faster convergence to happen even if the minimum RTT SUM rule is not reached ($min\{R_1+R_2\} = min\{R_1\}+min\{R_2\}$). This is done by including a tolerance threshold, in order to accomodate samples for which the rule may not converge. This can be later adjusted to an anytime version of the algorithm presenting better results as more packets are exchanged.

Algorithm 1 Minimum Packet Pair Estimation

1: **if** $numPacketPairs = PACKETS\_ARRAY\_SIZE$ **then**
2: $pair \leftarrow minSumRTT(PacketPairsArray)$
3: $firstPacket \leftarrow minFirstPacketRTT(PacketPairsArray)$
4: $secondPacket \leftarrow minSecondPacketRTT(PacketPairsArray)$
5: $sumRTT \leftarrow pair[0].rtt + pair[1].rtt$
6: **if** $sumRTT < (firstPacket.rtt + secondPacket.rtt + ToleranceGap)$ **then**
7: $dispersion \leftarrow pair[1].timeStamp - pair[0].timeStamp$
8: $secondPacketSize \leftarrow pair[1].packetSize$
9: $narrowLinkCapacity \leftarrow secondPacketSize \div dispersion$
10: **end if**
11: **end if**

IV. Testbed Description and Experiments

In this section, we describe an extensive "path estimation" campaign using our TCP receiver implementations. Most of the experiments were conducted on 11/07/06 to 12/04/06 especially during the nighttime. The central measurement point was chosen to be located at UCLA (University of California Los Angeles). Measurements were collected exploring path diversity to a set of 34 open source web servers. This list of open source web servers was, initially, obtained from the website "Google Summer of Code [4] 2006", a well-known repository of open source related projects. The idea was to pick up large enough (10MB) files from these websites, in order to form a domain of websites to study. The chosen web servers belong to the following sites: sourceforge.net, opensolaris.org, osuosl.org, some .edu, bbc.co.uk, and etc.

We installed our modified Linux 2.6.18 kernel implementation in one high-end host, composed by a Dual Intel Xeon 3.2GHz processor, 1GB RAM, PCI-X 64B/133Mhz (front-bus bandwidth 8Gbps), Intel 1000 Server Pro NIC (1 Gbps Ethernet). The machine was connected directly on a Cisco Catalyst 4500 with multiple 1 Gbps interfaces, one of the main access switches to the UCLA all 10Gbps core backbone. The connectivity of the UCLA backbone with academic networks, the case for most of the open source sites, goes through the CENIC (Corporation for Education Network Initiatives in California) at 1Gbps, and at 1Gbps to Abilene (Internet2). In addition to the careful choice of time and high capacity path (good to measure slower link speeds), we create a measurement environment with

minimal interference from the machine. In this regards, we used the Linux kernel in single user mode and stored all the logs and downloaded files in a RAMdisk of 64MB, reducing the likelihood of other processes and I/O requests to impact measurements.

In terms of initial validation of the path estimation technique, we have done several internal experiments within UCLA. In more than 90% of the experiments the error was below 10% from the real physical capacity. We have tested successfully to several webservers of different UCLA departments with capacities varying from 2.5Mbps wireless LAN laptops up to 10Mbps and 100Mbps. In addition, we performed extensive validation experiments using the UFES University in Brazil to connect to some local DSL modems with capacity estimated around 800 kbps.

With respect to the open source server experiments, the measurement bulk consisted of more than 20,000 downloads to the chosen sites. Each download was limited to only 3 seconds, by killing the process after the deadline. The reasoning of such short amount of time, per TCP connection, was because we were interested in the capacity estimation over the first 40 packets.

Our analysis of the collected data starts by presenting the heterogeneity of the chosen servers, in terms of propagation delay and path capacity. The following figures (Figure 4(a)(b)) present such information. On figure 4(a), we can observe that 65% of the websites had a end-to-end delays of less than 100 msecs, therefore geographically located either inside USA, or close to USA. The rest 35% had more than 100 msecs delay, capturing servers on other continents. In terms of narrow link capacity, the measured open source sites had in its majority 75% of the E2E narrow link capacities less or equal to 100Mbps. The percentage of narrow links below or equal to 10 Mbps was about 30% of the cases. Other capacity clusters obtained were in the 155 Mbps (OC-3) region with 5%, in the 622.08 Mbps (OC-12) with 15% and finally a few more 5% close to 800Mbps.

As we narrow down the estimates per server, using the short regulated algorithm and 100 experiments per server, we can group the estimations in clusters (Figure 5(a)(b)(c)) of estimated narrow links below 100Mbps, from 100Mbps to 400Mbps and beyond 400Mbps.

In each figure, we generate a boxplot representation of the data for each cluster. The boxplot groups the data distribution inside a box area, it show the main characteristics of the data distribution, the line in the center is the median of the data (less skewed centroid than average), the upper and bottom limits of the box represent the 25% and 75% quartiles, finally the points outside the box represent outliers (3 times the standard deviation). The analysis of the results shows that the method has good accuracy in every cluster. The results "below 100Mbps" showed the smallest level of variation. The estimation of the "100Mbps to 400Mbps" narrow links presented some expected fluctuation due to cross-traffic, and possibly service policies.

As we approach higher capacities, the measurements had a higher variability, as expected. This happens because any micro-level effect on the packet pair during its traversal through the end-to-end path can disperse the pair substantially. As a short example, if we are measuring a 1Gbps narrow link, the minimal packet pair dispersion is theoretically in the order of 10 μsec granularity, assuming 10,000 bit packets. While, 800Mbps theoretically should have as the minimal dispersion about 12.5 μsec, a difference of only 2 μsecs! In the graph (Figure 5(c)) we observe that the measurement precision limit seems to lie in the region of 622 Mbps (OC-12), in other words, from the high capacity estimation, the 622Mbps ones presented the smallest distribution box.

One question that we were interested in was the "similarity" of the inference algorithms: short regulated and no regulated one. In order to compare them, we performed 21,332 experiments, to all sites, using each algorithm. After the collection process, we computed the difference of the median obtained from each algorithm and every site. The result showed that the methods are nearly identical in their estimates, since the difference of the two was smaller than 10Mbps on 81.25% of the cases.

In order to capture the remaining 18.75% case, we plotted the difference distribution for each site in figure 6(a), the difference is calculated by subtracting every "short regulated" measurement from the equivalent "no-regulated" ones. We can observe that most of the 40,000 samples are close the zero difference, while the difference has some fluctuation less than 200 Mbps and, in 4 cases less than 500 Mbps. These cases are the ones where we have high speed expected fluctuations, therefore it is not an artifact of the algorithms differences.

A thorough analysis of the position where the best packet pair occurred inside the vector that stores the packet pair estimates, showed a bias on the first 5 packet pair samples. The best packet pair is referred as the one with minimum RTT sum among all packet pairs in a connection. This result shows that the small difference observed between the algorithms has an explaination on the fact that in 68.30% of the cases, the best packet pair was located within the first 5 packet pairs.

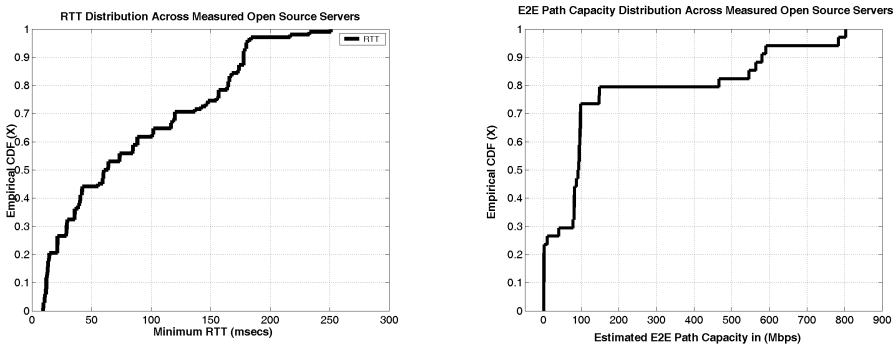
The following figure 6 presents an histogram of the position where the best packet pair was found in the vector for our 20,000 experiments. The histogram shows a strong mode in the second packet pair of the connection, while the rest of the positions appear to be uniformly distributed. We argue that the bias on the first 5 packets could be due to the expected low cross-traffic in the nightime experiments, thus the first pairs would experience minimum delay. In addition, the explanation of the first pair not being the strongest mode lies on application delay to fetch pages, so that the second pair is more likely to go through without addicional processing time. We intend to validate these claims in the future.

Finally, another issue on the algorithms is the likelihood of the minimum sum convergence ($min\{R_1 + R_2\} = min\{R_1\} + min\{R_2\}$). An analysis on dataset showed that using the 20 packet pair vector, 72.98% of the samples converged. Moreover, the convergence has a strong filtering effect on the outliers of the capacity estimation distribution, as it can be seen when comparing: samples without convergence and samples with convergence. The figure 6(c) shows the percentage of "good" samples (out of 20,000) estimated "correctly" inside a range of 2 * standard deviation from the median. Thus, it can be observed that the convergence increase the chance of obtaining "good" samples from 93% to 99.1%.

V. Applications

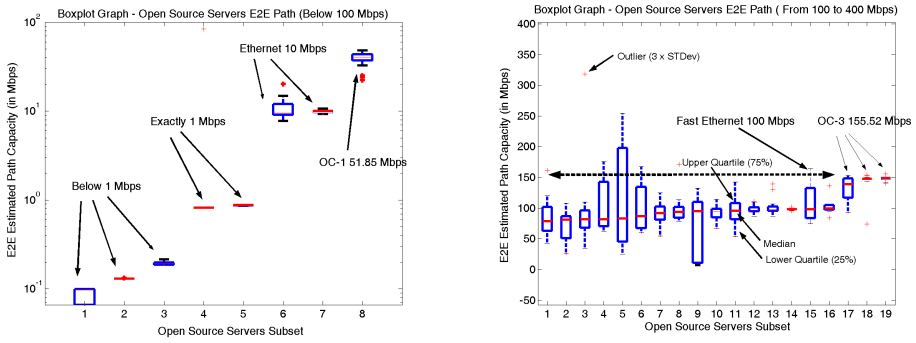
In this section, we describe two developed applications that make use of our TCP built-in capacity estimation. The first application is a direct use of the embedded narrow link estimation on the TCP flow control mechanism: i.e. receive advertised window. Thus, the receiver regulates its window size based on measured capacity and delay information. Moreover, it effectively limits the amount of traffic in flight reducing the usage of buffer space on the bottleneck, and improves TCP throughput by updating faster the amount of receiver buffer space. The second application is designed to find the exact location of the narrow link (i.e. bottleneck) on an Internet path. It involves a combination of traceroute and monarch [5] tool.

We should emphasize that the purpose of this section is only to show the potential applicability of our embedded TCP capacity

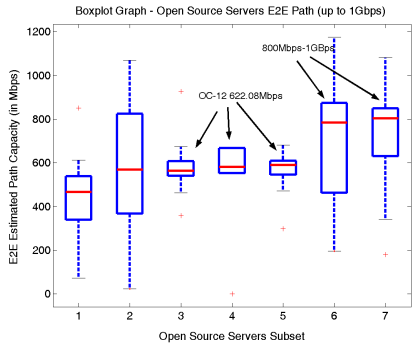


(a) RTT Diversity (b) E2E Path Capacity Diversity

Fig. 4. Exploring the Heterogeneity of the Open Source WebServers

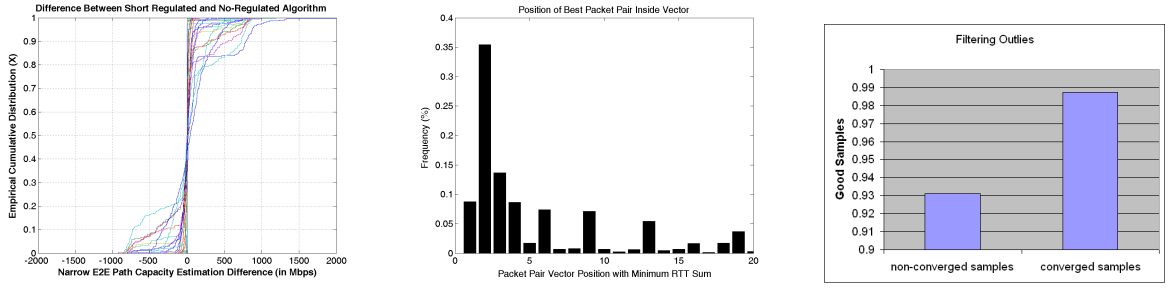


(a) Below 100Mbps Sites Subset (b) From 100Mbps to 300Mbps



(c) Above 300Mbps

Fig. 5. Narrow Down the Link Capacities



(a) CDF Measurement Difference Per Site (b) Best Packet Pair Position Distribution

Fig. 6. Similarity of Short Regulated and No Regulated & Min RTT Sum Convergence Filtering Effect

estimation. We did not perform an exhaustive experiments campaign, but we present preliminary results.

A. Narrow Link Based TCP Flow Control

The developed application assures the proper use of the narrow link capacity, by controlling the TCP data flow according to the maximum bandwidth-delay (BDP) product into a certain path. The BDP quantity is the maximum amount of packets or bytes than can be in-fligh at any moment without using buffers, sometimes it is also called pipe size. By using the BDP value, the receiver advertises (using the receive_window) the maximum allowed to sender, effectively limiting the amount of traffic in flight to be equal to the pipe size. Thus, reducing the usage of buffer space on the bottleneck. It is important to point out that, in the lack of this flow control, TCP naturally fill out buffers in the bottleneck since it keeps increasing the congestion window until it hits a packet loss.

An estimation of the optimal receive_window, and a fast update of its value within the first 40 packets also impacts the slow start phase. A normal Linux implementation would start the receive window small (i.e. 4 packets), and increment by 2 for every packet. This potentially limits the effectiveness of the slow-start increment, since it prevents a large number of initial packets during the slow start. Our method allows the regular slow start to execute free of bounds until the pipe size is reached.

We implemented the changes of receive_window estimation/update in the Linux kernel. Whenever the convergence is reached, we reset the receive window to a constant value derived from the multiplication of capacity estimate times minimum RTT of the path. We assume the minimum RTT of the path is equivalent to the propagation time, then the maximum amount of packet in flight without considering any buffer is $BDP = capacity * propagation$. A final adjustment is done, since the receive_window in Linux usually uses the window scaling factor (RFC 1323).

In addition, we did some experimental work on this new application. In a controlled lab, we emulate typical DSL conditions using dummynet [8], a popular network emulation tool. Thus, we configured two machines to be connected through an emulated link of 2 Mbps, buffer size of about 300 KB (or 300 packets) and a propagation delay of 200 msecs. After the setup phase, we configured a webserver on the TCP "sender" side and we used wget (a popular linux download application) to download a file from the server, as our TCP "receiver" side. We repeat the test using, on the same conditions: a normal Linux TCP and our capacity-based receive window modified version.

The following Tcptrace [9] graphs show that applying such limiting rate technique on an emulated environment improved the total throughput of the connection. In the first result, we compare figure 7(a) normal TCP and 7(b) modified one. The figure is the time sequence graph, where the y-axis represents packets sent/received during the connection time line. Moreover, since the sequence number is always incrementing by the packet size, the curve shows the growth rate of the TCP flow. Comparing the behavior, we can verify that as soon as the capacity estimation is known, there is a short exponential increase to the pipe size at time 2 seconds. At time 12 seconds, normal TCP had transferred 2MB, while our modified version reached 2.75MB, an improvement of almost 40%.

This better and more constant growth rate also has an impact on the flow throughput. In the figure 8 the upper line represents instantaneous throughput every 10 packets while the bottom one is a cumulative average throughput from the beginning of the connection.

```
$ sudo ./caplimiter www.cs.caltech.edu monarch -p tcp-ack -z 1001
traceroute to whirlwind.cs.caltech.edu ( 131.215.44.115)
, 30 hops max, 38 byte packets
1 131.179.80.3 0.211 ms 0.191 ms 0.189 ms
2 131.179.12.3 0.708 ms 0.678 ms 0.672 ms
3 169.232.49.65 0.442 ms 0.403 ms 0.599 ms
4 169.232.4.22 0.580 ms 0.837 ms 0.429 ms
5 169.232.4.103 0.748 ms 0.738 ms 0.454 ms
6 137.164.27.5 1.089 ms 0.927 ms 0.900 ms
7 137.164.27.248 1.117 ms 1.067 ms 1.199 ms
8 131.215.254.43 1.368 ms 1.224 ms 1.244 ms
9 131.215.44.115 1.455 ms 1.368 ms 1.318 ms
Hop kudos-pb to 131.179.80.3 at 96412857 bps
Hop 131.179.80.3 to 131.179.12.3 at 98404081 bps
Hop 131.179.12.3 to 169.232.49.65 at 95985747 bps *
Hop 169.232.49.65 to 169.232.4.22 at 97530215 bps
Hop 169.232.4.22 to 169.232.4.103 at 98773510 bps
Hop 169.232.4.103 to 137.164.27.5 at 97373535 bps
Hop 137.164.27.5 to 137.164.27.248 at 97657516 bps
Hop 137.164.27.248 to 131.215.254.43 at 96416740 bps
Hop 131.215.254.43 to 131.215.44.115 at 81429147 bps *
Min capacity hop is 131.215.254.43 to 131.215.44.115 at 81429147 bps
```

TABLE I
MONARCH EXECUTION EXAMPLE

So, analyzing the modified case, we can observe that the instantaneous throughput is much regular/constant than the normal one, in fact from 4 to 7 secs it is similar to a CBR. The cumulative throughput along the connection shows a large gap between the normal (170 KB/s or 1.36Mbps) and our modified version (225 KB/s or 1.8 Mbps), effectivelly improving the utilization of the 2Mbps link from 68% to 90%.

B. Narrow Link Location Tool

As we pointed out, this application uses a combination of traceroute and monarch [5] tool. The goal is to find the exact locations and capacity of narrow link bottlenecks on an Internet path. In addition, the tool was used to validate some local results at UFES network, estimating path capacities of non-cooperative wireless links, servers connected at 10Mbps hubs, servers at 100Mbps, and so forth accurately.

As a short overview of the monarch tool, it was implemented to experiment with new TCP implementations in the "network at large" by forcing non-cooperative hosts (i.e. routers and serverless hosts) to reply back several types of probes as if it were a normal end-to-end TCP connection. To accomplish this task, Monarch starts a TCP connection with itself (with both call-legs, sender and receiver, in the same machine). However, the TCP packets, instead of being sent directly, they are changed into probes (ICMP, UDP, dummy TCP on closed ports) and sent directly to a specific remote host. The remote host then replies with errors/control messages (like ICMP replies or TCP RST). The final step changes the reply probes back to TCP packets as they arrive, this way, sending packets to the receiver side of the local TCP connection. In summary, letting TCP generate as much probing traffic as TCP normally would do (Figure 9)).

Since our modified kernel is actually embedded in TCP Receiver Side (*) as shown in Figure 9(a), it was not necessary to change the monarch implementation. We just instrument it by estimating path capacities from the local receiver to any non-cooperating host, in addition, exporting the estimate from the kernel to userland through a regular netlink() API.

In order to perform the narrow link position discovery, our instrumented monarch tool calls traceroute to a certain destination, as shown in our execution example (Table I). Once the intermediary routers are all known, the monarch tool is used to create one "emulated" TCP connection to each discovered router along the path. Thus, estimating step-by-step the narrow link capacity, and further position, using a regular TCP loop.

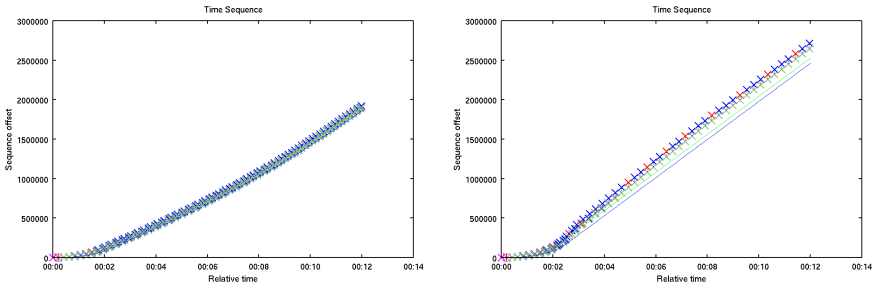


Figure A - Normal

Figure B - Modified

Fig. 7. TCP Time Sequence

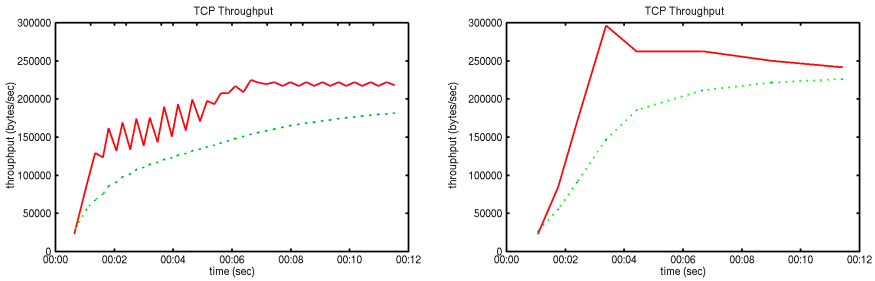
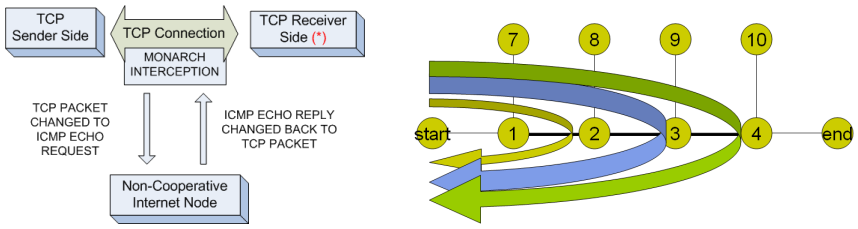


FIGURE (a)

FIGURE (b)

Fig. 8. TCP Throughput Comparison



(a) Monarch Design

(b) Find the Position of a Narrow Link

Fig. 9. Discovering the position of Path Capacities on Any Host of the Internet

VI. Conclusion

In this paper, we have presented a simple technique to estimate end-to-end Internet paths capacity within a TCP receiver. In spite of the subject of the path capacity estimation has been greatly explored in the literature, our novelty is on embedding it into an ongoing TCP receiver connection, by making use of it directly. We describe in details our algorithms, the tradeoff of capacity estimate convergence versus number of false positives. In addition, we explore a rich set of open source file server to discover the path capacities associated with them. The validation was done by extensive tests in the lab, in the department local area network, and also by contacting some of the ISPs that we measured. The results present consistent accurary up to 600 Mbps using TSC register. We finish the paper, presenting two possible applications, one augumented version of Monarch [5] that narrow down the end-to-end capacity estimation to every router along a certain path, and one advertised windows proportional to the pipe-size limiting the maximum sending rate of TCP and the buffer usage of bottleneck.

Acknowledgment

The authors would like to thank Chris Frost (UCLA) for exporting path capacity to applications.

References

[1] R. L. Carter and M. E. Crovella. Measuring bottleneck link speed in packet switched networks. *Performance Evaluation*, 27:297–318, 1996.

[2] C. Dovrolis, P. Ramanathan, and D. Moore. Packet-dispersion techniques and a capacity-estimation methodology. *IEEE/ACM Transaction on Networking*, 12(6):963–977, 2004.

[3] A. B. Downey. Using pathchar to estimate internet link characteristics. In *SIGCOMM*, pages 241–250, 1999.

[4] Google. Summer of code open source project. In *http://code.google.com/soc/*, 2006.

[5] A. Haeberlen, M. Dischinger, K. P. Gummadi, and S. Saroiu. Monarch: A tool to emulate transport protocol flows over the internet at large. In *Proceedings of the ACM/USENIX Internet Measurement Conference (IMC)*, 2006.

[6] R. Kapoor, L. Chen, L. Lao, M. Gerla, and M. Y. Sanadidi. Capprobe: a simple and accurate capacity estimation technique. *ACM SIGCOMM Computer Communication Review*, 34(4):67–78, 2004.

[7] V. E. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD dissertation, University of California, 1997.

[8] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[9] Tcptrace. Analysis of tcp dump files. In *Tcptrace. http://iarok.cs.ohiou.edu/software/tcptrace/tcptrace.html/*, 2001.

[10] B. Veal, K. Li, and D. Lowenthal. New methods for passive estimation of tcp round trip times. *Passive and Active Measurements (PAM)*, 2005.