Making Better Use of All Those TCP ACK Packets

Hao Shang and Craig E. Wills
Computer Science Department
Worcester Polytechnic Institute
Worcester, MA 01609
Email: {hao,cew}@cs.wpi.edu

Abstract— **Many TCP connections show an asymmetric traffic pattern where significantly more data is sent in one direction than the other resulting in large numbers of ACK-only packets to be generated. In this work we propose and evaluate two independent approaches for how to make better use of the available space in TCP ACK-only packets to improve data transfer. The first approach provides packet-efficient throughput in the reverse direction of a connection without sacrificing forward throughput, while the second approach provides more detailed and complete information about the state of the forward connection that could be used by a TCP implementation to obtain better throughput under different network conditions.**

Keywords: **Network Measurements, Protocols, TCP, Reverse Transmission**

I. Introduction

TCP is a widely used protocol on the Internet designed for reliable bidirectional data transfer. However, large numbers of TCP connections show an asymmetric traffic pattern where significantly more data is sent in one direction than the other. It is a common scenario with applications such as HTTP or bulk data transfer that a client initiates a request to a server and then the data flow is entirely from the server to the client. Traffic in the reverse direction is only TCP ACK packets to acknowledge receipt of the downloaded data.

The large number of small TCP packets on the Internet is well documented. Results in [19] show a trimodal distribution of TCP packet sizes where the size of ACK-only packets (40 bytes with no TCP options) is one of the modes. Statistics in [13] show just under 50% of TCP packets are of TCP header size. Various traces from the IP Monitoring Project taken at monitoring points in the SprintLink IP backbone in February 2004 show 40-70% of packets in individual traces are of TCP header size [18]. Another study [17] also shows that over 40% of packets are 40 bytes for traffic collected at five different network points, including Los Nettos, a USC Internet2 connection, and three connections monitored by NLANR during December 2004 to October 2005 period.

The large fraction of small TCP packets in the Internet motivates our work to examine whether the performance of applications using TCP for data transfer in the forward channel can be improved by making better use of the reverse channel acknowledgment packets [16]. In this work we explore the potential of two approaches to enhance how TCP handles the ACK-only packets it generates.

The first approach is to piggyback application-level data onto ACK packets sent in the reverse channel. Normally TCP piggybacks acknowledgment information onto the data packets it sends. In this work we explore the potential for a mechanism to piggyback data only when an ACK packet would normally be generated. If a mechanism was available for applications to send reverse-channel data without generating additional network packets then client receivers of data could upload feedback to the server providers of the data without incurring new connections or generating additional traffic. Such a mechanism could also be used by peers in a peer-to-peer environment where the transfer of desired content from peer A to peer B could simultaneously support the exchange of useful content via piggybacked transfer from B to A.

The second approach we explore in this work for making better use of ACK-only TCP packets is for a receiver to provide additional control information to the sender via additional TCP header information. We examine the introduction of a new TCP option that provides more detailed and more complete information about the reception of data packets at the receiver compared with the existing TCP Timestamps Option [10]. This information allows a TCP sender to track the spacing between all data packets arriving at the receiver and to have complete timing information for the forward and reverse directions of the connection. The information can be used to better detect jitter and congestion in the forward direction than what is currently available.

The organization of the paper is as follows. In Sections II and III of the paper, we describe the data piggybacking and enhanced ACK approaches in more detail

and compare them with previous work. In Section IV we describe the symmetric and asymmetric network environments used to evaluate each of these approaches. The methodology and results for evaluating data piggybacking are described in Sections V and VI followed by methodology and results for evaluating enhanced ACKs in Sections VII and VIII. In Section IX we summarize observations from the evaluations and consider the impact of combining the two approaches. We conclude with a summary of our findings and directions for future work in Section X.

II. Piggybacking Data

The TCP protocol allows data transfer in both directions of a connection with ACKs for data packets received in the forward direction piggybacked onto data transferred in the reverse direction. However because many connections primarily transfer data in only one direction many ACK-only packets are generated by the receiver. The key idea of our work is to invert the traditional TCP transmission mechanism and piggyback data onto packets carrying needed ACK information. This approach creates a clear primary and secondary direction of data flow within a TCP connection.

This approach is interesting to explore because it allows data transfer to occur in both directions while being potentially more efficient in the number of packets generated for data transfer in the reverse direction. Potential applications of this approach include asymmetric connections where previous work shows that less bandwidth in the reverse direction impacts performance for forward data transfer [3]. Other work [11] has shown that bidirectional traffic can cause reduced throughput due to undesired interaction effects such as ACK compression [23].

Peer-to-peer (p2p) applications can also be written to take advantage of reverse ACK traffic where the transfer of content from peer A to peer B could simultaneously support the transfer of useful content from peer B to peer A. Incentives in a p2p application such as BitTorrent encourage clients to exchange data tit-for-tat in both directions [7], which could be done more efficiently with our approach. Others have proposed the idea of generalizing BitTorrent with a Data Exchange Market [20]. Our approach could be used with this idea as well.

Our approach does require a new mechanism to be supported by TCP with modifications to client and servers. To illustrate, Figure 1 shows the core code for a standard data transfer from a server to a client. The server continually sends data in a buffer (buf) to a socket (s) while the client reads data from its socket into a buffer and processes it.

```
// Client                  // Server
while (not done) {         while (not done) {
  recv from s into buf;      put data into buf;
  process buf data;          send buf to s;
}                          }
```

Fig. 1. Standard Core Client and Server Code

Figure 2 shows the modified code using a new data piggybacking TCP option, which causes the TCP implementation to only send buffered data if a packet is generated to ACK data received. In Figure 2 the client checks if reverse direction send buffer space is available and if so then sends data to the socket (s). Otherwise the client works just as the standard case. The server requires fewer modifications as it simply checks the availability of input data on the socket, using a call such as `select()`, and if available it receives and processes that data. Because each loop is driven by the data being sent in the forward direction a mechanism is needed for the client to query how much of the buffered data has actually been sent. Depending on the application, the client may need to finish sending any unsent data via the traditional mechanism or simply terminate the connection if the data do not need to be sent.

```
// Client                  // Server
turn on data piggyback;    while (not done) {
while (not done) {           if (revdata avail) {
  recv from s into buf;        recv from s into rbuf;
  if (send buf space)          process rbuf data;
    send revdata to s;       }
  process buf data;          put data into buf;
}                            send buf to s;
                           }
```

Fig. 2. Modified Core Client and Server Code

A primary issue with this approach is how much data can be piggybacked onto an ACK packet. In a bandwidth constrained environment trying to piggyback too much data could have a negative effect of forward traffic. This issue is examined in our testing. In our testing we did not modify the TCP implementation for this initial work, but used the existing TCP implementation with the code shown in Figure 2. The result is that each time data is received from the socket in the loop, we send reverse data to the socket if buffer space is available. The amount of reverse data sent is a parameter of each experiment. This approximated approach does not guarantee that reverse data and ACKs are sent together, but with a client buffer big enough to receive a full packet we observe that generally each application-level receive corresponds to

a packet reception thus the client generally sends one packet for each received.

Ideally, we could have a kernel-level support to ensure that each data transmission matches an ACK packet. However, we use the user-level approach to evaluate the concept of data piggybacking. The experiments over different paths do show that the results using the approximated approach are close to those we calculate under the ideal situation where the kernel-level support would be available. With only a user-level modification, we are able to test the data piggybacking method widely as no privileges to change the TCP/IP kernel are needed.

III. Enhanced ACKs

The data piggybacking approach seeks to use available bandwidth in TCP ACK packets to send reverse data. The second approach we explore in this work uses a bit of the available bandwidth to enhance the contents of ACKs to provide more detailed and complete information about the reception of data packets at a receiver. The TCP Timestamps option allows TCP implementations to calculate round-trip times (RTTs) on more than one packet per window of packets and allows time stamp echoing in either direction [10]. The contents of the option include four fields: one-byte each for the kind (k) and length (l) with four-bytes each for the packet timestamp (TSval) and echo reply timestamp (TSecr).

The use of these timestamp fields is illustrated in Figure 3, which shows two common scenarios for how the Timestamps option is used when an ACK is generated by the receiver. The example is illustrative and does not show all packets in a TCP connection nor does it show how the TCP option is used when packet loss occurs. The first ACK in the example is generated at time 30 in response to data segment 1 being received at time 28 with a TSval of 18. The delay between when a packet is received and the corresponding ACK generated is defined in [2] to be within 500ms of the arrival of unACKed packet and at least every second full-sized segment, although 200ms is a commonly used maximum time. Previous work [6] found that transmission of small data packets often incurred close to 200ms ACK generation delay in the presence of the TCP Nagle algorithm [14].

The first ACK TSval indicates it was sent at time 30 and is an echo response to the packet with timestamp 18. The second ACK is generated at time 70 after two data segments have been successfully received. This behavior occurs with the TCP delayed ACK feature, which effectively ACKs every other packet. The TSecr value of 45 indicates the timestamp of the earliest previously unACKed packet as defined in [10].

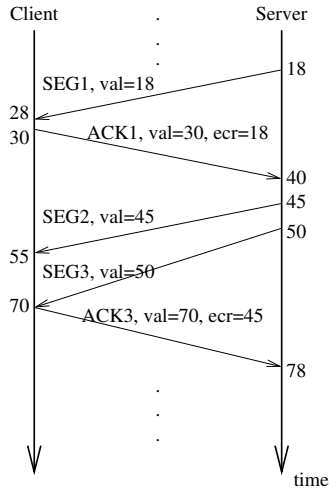

Fig. 3. Example Usage of TCP Timestamps Option

The Timestamps option was proposed to enable the sender of a TCP connection to more frequently determine the RTT of sent data segments, but as the example shows the option still does not capture as much information about the nature of the connection as it could. We conjecture that more complete information about the pattern of received packets would allow better transmission of sent packets despite issues due to congestion or the delayed ACK option. It can also help to detect the effects of ACK compression when ACKs are grouped together [23].

The layout for the variable-length enhanced TCP Timestamps option we propose is shown in Figure 4. It extends the existing Timestamps option in two ways. First, the reception time for each data segment received by the receiver is returned to the sender in a TSrcv field, and second the TSecr and TSrcv values are returned for all packets received since the last ACK was sent. Given that the maximum size of a TCP option field is 40 bytes, information for up to four packets could be included, although if every other packet is ACKed then information for no more than two packets would need to be included. For the example in Figure 3, the second ACK would have the fields: val=70, ecr1=45, rcv1=55, ecr2=50, rcv2=70. Once sent, the receiver no longer needs to retain the information thus the amount of receiver state maintained is bounded.

k	l	TSval	TSecr1	TSrcv1	TSecr2	TSrcv2	···
1	1	4	4	4	4	4	

Fig. 4. Enhanced TCP Timestamps Option Layout

This enhanced Timestamps option provides three improvements relative to current TCP functionality.

1) The capability to measure one-way jitter and

packet spacing in each direction. The enhanced option allows the packet spacing on the receiver side to be communicated to the sender rather than trying to use ACK spacing at the sender to approximate packet spacing.

2) The option explicitly captures the delay for generating an ACK at the receiver. While ACK generation may be immediate for full-size packets received during slow start or when the second of two full-sized packets is received, the sender does not know unless this information is recorded.
3) The option captures the delay information for all received packets. In the presence of delayed ACKs, information is lost because an ACK is only generated for every other received packet and there is no way to know when the first packet of each pair arrives at the receiver. The current Timestamps option is intended for a round-trip measure that includes delays, not for precise timing of each packet.

We envision the enhanced Timestamps option to be particularly useful in asymmetric networks. Balakrishnan, et al [4] describe a number of issues using TCP with asymmetric networks where low bandwidth on the reverse link causes problems for the timely arrival of ACK packets needed for the ACK-clocked nature of TCP. Variants of TCP such as TCP Vegas [5] and TCP Westwood [22] use rate estimation to drive when packets are sent based on the rate at which ACKs are received. However work such as [9] shows that in asymmetric networks TCP Vegas does not perform well because it is using ACK rate to estimate data rate at the receiver. They describe the need to encode the arrival times at the receiver and show improved results if these data are available, but imply the TCP Timestamps option can be used for gathering forward path flow rate without specifying details. Similarly, [1] uses the Timestamps option to estimate forward trip time in a TCP connection, but this option does not account for ACK generation delay and it loses information when delayed ACKs are used. Finally, the availability of one-way jitter information allows investigation of using jitter to predict congestion loss before it occurs. As part of measurement work on audio transmission, [15] found that RTT variation can be an indicator of packet loss.

IV. Testing Environment

We tested the two approaches over network connections between our home institution of WPI, on the east coast of the U.S., and seven endpoints with various RTT and throughput connectivity values as summarized in Table I. The RTT and throughput values are representative of those obtained during testing, although some variation in packet loss occurred, which is noted as appropriate. Four of the links to institutions in California and Georgia in the U.S. as well as to Italy and the Netherlands show relatively good bandwidth in both directions, although as shown in Table I demonstrate asymmetric throughput. Further investigation of this throughput asymmetry found that for the California to WPI path the advertised receiver window was 64K bytes by Linux on the WPI side while it was only 32K bytes by the Linux version running in California. For the other three links, the asymmetric throughput is primarily due to higher packet loss in one direction than the other.

The three other links in Table I do exhibit asymmetric bandwidth. These links include local machines accessing WPI via DSL and cable modem as well as one in the Netherlands connected via DSL. All seven of these links were used for data piggyback tests described in the following section while just the Calif/WPI, Italy/WPI and WPI/Local DSL links were used for the enhanced ACK tests described in Section VII because tcpdump was needed on both sides of the link to capture packets. Unless noted, all tests were run using Linux TCP implementations.

TABLE I
Summary of Network Connections Used in Experiments

End Points		A to B		B to A	
A	B	RTT (ms)	Thruput (KBps)	RTT (ms)	Thruput (KBps)
Calif	WPI	80	600	80	350
Italy	WPI	120	200	120	400
Georgia	WPI	30	600	30	1100
NL	WPI	90	150	90	520
WPI	NL DSL	90	280	90	80
WPI	Local DSL	27	190	27	40
WPI	Local Cable	10	350	10	40

V. Piggybacking Data Methodology

The question for the data piggybacking approach is to understand how much data can be piggybacked with ACK packets without introducing more packets and not influencing the performance of forward-channel traffic. We answer this question with a series of experiments conducted under the variety of real network conditions described in Table I. For each experiment we evaluate performance and efficiency of different transmission methods, where throughput (transmitted bytes divided by transmission time) is used as the metric of performance and packet counts is used as the metric of efficiency.

As a standard, we used the transfer of a 1MB file for all testing. We deliberately chose this size as its

transfer normally takes around 700 packets, which is large enough to avoid throughput effects of slow start, but small enough for a reasonable experimental time.

We evaluated the approach described in Section II where a server sends a 1M byte file in the forward direction to a client. As shown in Figure 2, each time the client reads some data it sends data of a particular size to the server if buffer space is available. Ideally the TCP implementation on the client-side only sends the data when it would normally generate an ACK, but given that we have not modified the TCP implementation the reverse data sent is not exactly matched with the sending of ACKs. However, because the client application code sends data immediately after it receives data from the server, the sending of the data is roughly matched with the sending of ACKs (that are generated for the received data). As part of the experiment we control how much data is sent in the reverse direction for each packet received. In an ideal implementation, this amount of data would be piggybacked with each ACK. Idealized values for throughput and the number of packets are shown in the results for each size.

In addition, we evaluated two control approaches: In the first, labeled "2Con" in the results, the algorithm in Figure 1 is used by two identical and separate processes on each side to transfer 1MB in each direction. In the second control approach, labeled "1Con", 1MB is again transfered in each direction, but only a single connection is used for the transfer so that ACKs may be piggybacked on reverse data traffic. The identical endpoints are written so that they do not needlessly block waiting to send or receive data.

For each network condition, experiments are conducted with each of the three approaches, where the size of piggyback data sent in the reverse direction varies from zero (i.e. no piggyback) to 1500 bytes. Note that the size of 1500 bytes is over the size of the MSS (1460 in this experiment). It is intended to check the effects when the piggybacked data is more than the MSS. The maximum piggyback size tested within the MSS is 1400 bytes in these experiments. Results are reported based on five runs for each case, although in cases where the loss rate is generally over 1%, 10 runs are conducted to mitigate fluctuations in throughput.

VI. Piggybacking Data Results

Tests for all links in Table I were made as described in the previous section. Rather than show all results, this section shows results across a representative set of network conditions. The first set of results are shown in Figure 5 for the connection from the California site to WPI. The first graph shows throughput results and the second graph shows packet count results. In each figure, the curves on the left side are for the piggyback approach, where one curve represents the download direction and the other curve represents the upload direction. There are two sets of bars on the right side of each figure. The rightmost set of bars are for the 1Con approach where one network connection is used to download and upload a 1MB file. The set of bars to its left are for the approach where two individual connections are used. In each set of bars, the left one represents the download direction and the right one represents upload direction.

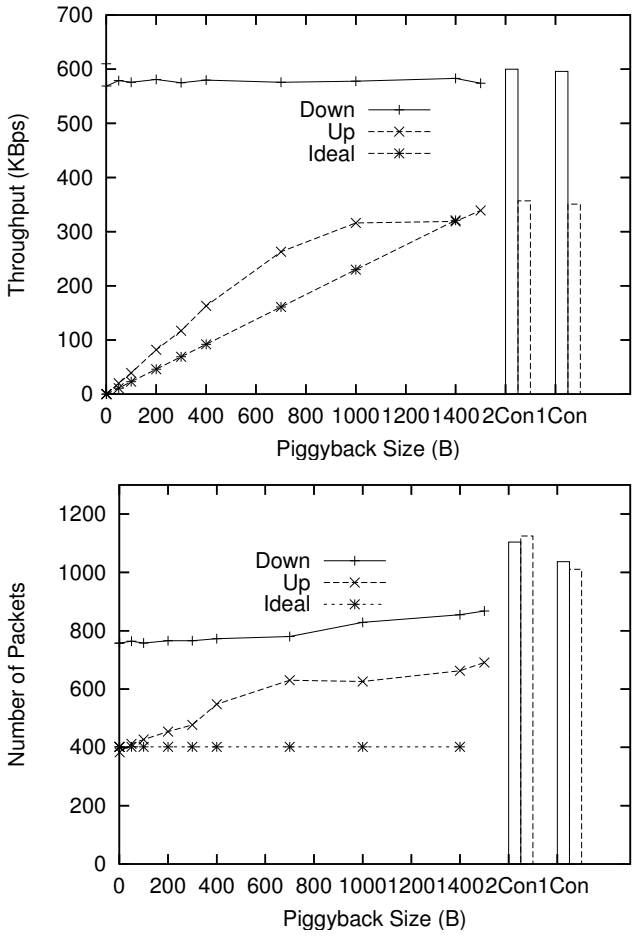

Fig. 5. Calif to WPI (less than 1% packet loss)

For the Down and Up piggyback curves, the measured throughput and packet count results are shown for piggyback sizes of 0 (no piggybacking), 50, 100, 200, 300, 400, 700, 1000, 1400 and 1500 bytes. The Ideal throughput results are obtained by multiplying the number of ACKs observed in the no piggybacking case by the piggyback size, constrained by the maximum observed reverse throughput.

The results show that the downlink throughput is unaf-

fected by the piggyback size while the uplink throughput rises to a level comparable to the 2Con and 1Con approaches. The packet count graph shows the number of uplink packets to be relatively constant and fewer than the control cases. More important, the uplink packet count is significantly less that the two control cases with comparable throughput for the largest piggyback sizes. The results show that for piggyback sizes of a few hundred bytes, the existing TCP implementation does not generate many more packets than the ideal implementation.

Figure 6 shows results for a forward network connection from WPI to a host connected locally via DSL. As the throughput results show, the downlink throughput is significantly affected once the uplink capacity of approximately 40KBps is reached. The control approaches show how saturation of the uplink negatively affects both directions. For this asymmetric network, these results do show that our approach can be effective for uploading more modest amounts of data without increasing the number of uplink packets and without impacting the downlink throughput.

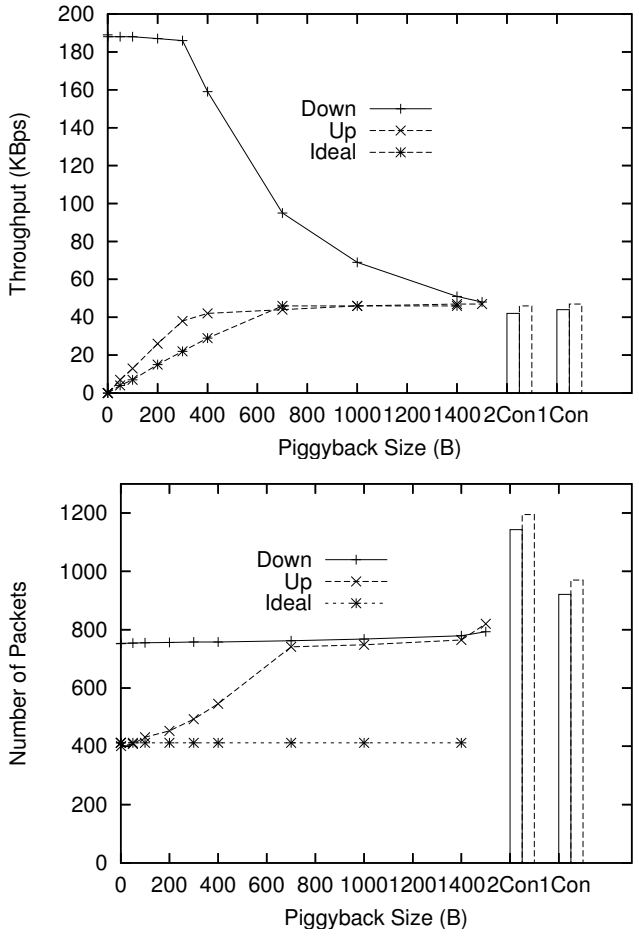

Fig. 6. WPI to Local DSL Home (less than 1% packet loss)

VII. Enhanced ACKs Methodology

The methodology for testing the enhanced ACK Timestamp mechanism described in Section III seeks to examine the measurement of connection metrics using the enhanced mechanism compared to using TCP with no options and the existing TCP Timestamps Option. With our enhanced Timestamps Option, the sender can learn exactly when the data packets are received on the receiver side. It allows several potential benefits:

- enables the capability to calculate packet interarrival spacing, which can be combined with packet sending spacing to determine forward direction jitter effects,
- provides the sender information about the delay for generating an ACK at the receiver, and
- allows more accurate calculation of RTTs by the sender.

We choose packet interarrival spacing, ACK generation delay and RTT as the three metrics to examine the potential benefits of using the enhanced Timestamps Option compared to currently available methods.

The interarrival spacing of data at the receiver is important because this information can be combined with the spacing of sent data to monitor the queuing delays in the forward direction, which may be used to infer congestion in this direction before packet drops occur. The spacing can be explicitly calculated using our enhanced Timestamps option. With no Timestamps option, the sender must match an ACK with the corresponding sent data. This match is non-trivial to do in the case of packet loss and the use of delayed ACKs. The spacing calculation also includes ACK generation delays as well as reverse direction congestion effects. The use of the current TCP Timestamps option removes reverse direction congestion effects from the calculation, but ACK generation delays are still present. In addition, with the presence of delayed ACKs, an ACK may represent two received data packets.

The delay to generate an ACK at the receiver is explicitly captured with our enhanced Timestamps Option, but unavailable using existing approaches. Variability in the ACK generation algorithm by a TCP receiver introduces variability in the ACK receiver spacing at the sender irregardless of any congestion in the network between the sender and receiver.

In terms of RTT calculation, the existing TCP Timestamps Option allows more frequent and accurate RTT calculation than without use of the option. Our approach yields a yet more accurate RTT and it allows the RTT for all packets to be calculated in the presence of the delayed ACK option, although as discussed in [10], the

sender must be less aggressive in using the RTT for retransmission time out (RTO) calculation.

In order to examine the behaviors of the delayed ACK option on different platforms, we conducted experiments on both Windows and Linux. The Linux kernel versions we tested are 2.4.21 and 2.6.11, while for Windows we used Windows 2000 and Windows XP. Our experiments show the two Linux versions have similar behavior and the two Windows variants behave similarly. However, Windows and Linux do show differences on how ACKs are generated. While both platforms set the maximal ACK delay as 200ms, Linux uses the measured RTT as the waiting time whereas Windows uses a round-robin scheme rotating from 100ms to 200ms. In addition, Linux ACKs immediately during the slow-start phase while Windows tries to acknowledge two data packets if possible even at slow-start. Furthermore, Linux sends an ACK immediately for data packets whose sizes are smaller than 500 bytes assuming they belong to an interactive session. Windows shows no discrimination based on packet sizes. In general, Linux reacts more aggressively than Windows on ACK generation.

We conducted experiments on the three links described in Section IV for both directions. We varied the client receiver platform between Linux and Windows. During the tests, packets were captured at both endpoints using tcpdump with millisecond time granularity. As comparison, [21] found about 75% of popular Web servers support the existing TCP Timestamps Option with the majority using a 10ms timestamp granularity.

We transfer a data file on each side to generate FTP-like traffic simultaneously in both directions over two separate connections. The file size is set to 1MB on both sides. We use two-way traffic to introduce cross traffic in the reverse direction, which may influence the latency of ACK packets. As expected, we only found reverse traffic to have an effect for the DSL client connected with asymmetric bandwidth. We did run each test with and without the delayed ACK option, but all results shown use the default where the delayed ACK option is turned on. We also introduce tests with the download of Web and streaming data from actual Web sites as described in the following section.

VIII. Enhanced ACKs Results

Using the file transfer test with 1MB of data, Figure 7 shows one of the more pronounced cases for differences of using ACK spacing to calculate spacing of received data packets. The results are for data sent from California to a client at WPI running on a Windows platform. The results in the CDF are obtained by comparing the data packet reception spacing (drecv) with the ACK send spacing at the receiver (asend) and ACK receive spacing at the sender (arecv). The results show the cumulative differences between the data spacing and each of the ACK spacings. In computing the difference in spacing between data and ACKs, if the sending or receiving of a delayed ACK represents the reception of two data packets, half of the ACK spacing is used to approximate each of the data spacings.

The significance of these results is that for over 50% of the packets there is a difference between the actual data reception spacing and the approximated spacings obtained using the ACK sent or ACK received results. In the figure, the two data lines overlap for these results because there is no congestion in the reverse direction. The average absolute difference is 60ms as shown in Table II, which contains a summary of all file transfer tests. There are two important points about the calculation of results in Table II: 1) we count a difference or delay as zero if it is less than 1ms; and 2) we use absolute values of packet spacing differences to calculate mean and median values.

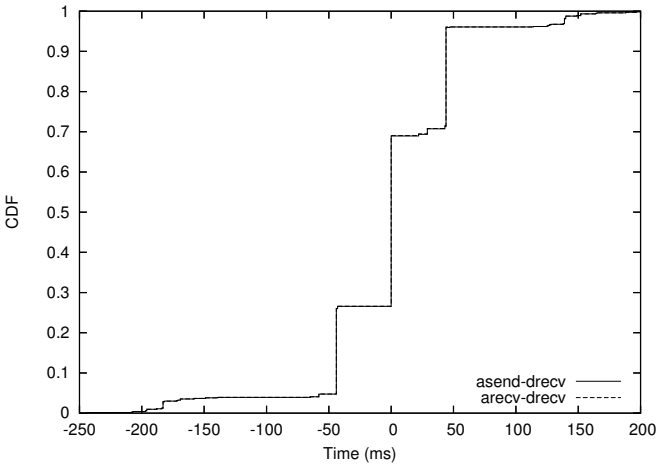

Fig. 7. Packet Spacing Difference among Data Recved, ACK Sent, and ACK Recved for California to WPI Windows Client

Table II shows the spacing differences are almost non-existent for the California to WPI connection when the client receiver is running Linux. The reason for the different results for a Windows client versus a Linux one is because there are many packet losses (about 10%) on the path from California to WPI for this test. Linux responds with an ACK immediately after a packet loss is detected and continues to respond with one ACK for each received data packet for a while even after the lost packet is recovered, which causes almost one ACK for every data packet for a high loss path. Windows also sends an ACK immediately once packet loss is

TABLE II
SUMMARY OF PACKET SPACING AND ACK DELAY UNDER FILE TRANSFER TRAFFIC (TIMES IN MS)

Connection	Recv O.S.	|Asend-Drecv| > 0			|Arecv-Drecv| > 0			Last ACK Delay > 0			First ACK Delay > 0		
		%	Mean	Med.	%	Mean	Med.	%	Mean	Med.	%	Mean	Med.
WPI→IT	Linux	8.8	26	19	13.9	17	7	2.1	41	40	56.4	2	1
IT→WPI	Linux	38.9	16	7	40.2	16	6	3.2	44	41	23.8	7	1
	Win	19.6	17	3	20.1	16	3	0.3	101	101	10.3	15	1
WPI→Calif	Linux	38.7	17	10	42.9	15	8	3.5	34	32	23.7	19	26
Calif→WPI	Linux	0.5	25	10	3.1	5	1	0.3	25	10	0	0	0
	Win	57.6	60	44	57.5	60	44	5.9	146	139	46.0	87	88
DSL→WPI	Linux	28.6	4	1	36.2	15	4	0	0	0	100.0	30	30
WPI→DSL	Linux	13.6	150	3	92.9	33	7	0	0	0	100.0	8	8
	Win	20.5	73	87	99.9	21	7	0.3	110	110	100.0	8	8

detected, but right after the loss is recovered, delayed ACK processing is immediately restored.

Except for these two extreme cases, the other file transfer tests in Table II show non-zero differences between the data reception spacing and ACK sending spacing for 8-38% of the packets received with an average difference generally between 15 and 25ms.

When using ACK reception spacing to infer data reception spacing, the estimation error is further enlarged as the reverse side congestion causes ACKs themselves to get delayed. For the path from WPI to the local DSL host, ACKs are transmitted with varied latency due to the congestion in the reverse direction. The ACK delay variance causes a difference for each estimation that is on average 20-30ms in magnitude.

The remainder of Table II shows ACK Delay results for all tests. The results show little delay to generate an ACK after the last packet is received, but frequently there is a delay between the receipt of the first and last packet of a pair.

These results are for FTP-like traffic. We postulate the method is more useful for Web and streaming traffic, where Web objects or streaming frames are not always sent at the rate allowed by TCP as is typical with during a file transfer. ACK delay on the receiver side is more significant. Table III shows the results for accessing two Web site home pages and two streaming media sites from two different client locations. We used the Internet Explorer (IE) browser for Web access on the Windows platform and Firefox for access on the Linux platform. Each browser used persistent connections. We were unable to access the streaming media sites using a Linux client. For the Amazon audio stream, which provides online listening for sample music, we used Real Player, a plug-in to IE. For the Yahoo video, which provides movie previews, we used the Window Media Player, also a plug-in to IE. As we only have client side packet traces, we are not able to determine the difference between ACK reception spacing and data reception spacing.

The "Last ACK Delay" and "First ACK Delay" results in Table III show significantly higher non-zero percentages as well as mean and median values than those results under FTP-like traffic in Table II.

Another interesting result is the ACK delay for the Amazon streaming audio to the Home DSL Windows platform. The last ACK delay is nearly a constant 110ms and there is no first ACK delay because each ACK is generated for exactly one data packet and the receiver is trying to wait for a second packet to arrive. The constant ACK generation delay does result in little difference between the data reception and ACK generation spacing for the Amazon to Home DSL connection as shown in Table III. However, in general there is a large discrepancy between these two spacing calculations in Table III because the sender of Web and streaming traffic is not always sending packets and the receiver delays waiting for a second packet to arrive before generating the ACK. These gaps between when data packets are sent combined with the delayed ACK feature mean that the gap between ACK transmissions is an unreliable estimator for the gap between data packet arrivals.

The final benefit of having more complete data and ACK packet transmission information is to calculate more accurate path RTT than using the Timestamps option. The difference is trivial when there is no ACK delay and packet loss. However, in many cases as shown in Table II and III, ACKs are delayed and consequently the RTT is overestimated. The situation becomes much worse when packet loss occurs because the RTT calculation could include packet retransmission time [10].

IX. OBSERVATIONS

Results from the two sets of experiments lead to a number of observations about the potential benefits of the data piggybacking and enhanced ACKs approaches as well as consideration of the merits in using them

TABLE III
SUMMARY OF PACKET SPACING AND ACK DELAY UNDER WEB AND STREAMING TRAFFIC (TIMES IN MS)

Traffic Type	Connection	Recv O.S.	|Asend-Drecv| > 0			Last ACK Delay > 0			First ACK Delay > 0		
			%	Mean	Med.	%	Mean	Med.	%	Mean	Med.
Web	CNN→DSL	Win	81.1	853	20	52.4	50	4	100.0	13	7
	CNN→WPI	Win	87.0	427	26	69.0	66	57	7.7	10	10
		Linux	54.2	814	15	17.1	25	39	12.5	29	29
	Cisco→DSL	Win	70.9	346	48	33.8	84	101	95.3	18	8
	Cisco→WPI	Win	73.6	155	44	34.1	81	107	41.4	37	2
		Linux	56.5	93	40	21.2	40	40	32.3	2	2
Streaming Audio	Amazon→DSL	Win	90.3	7	2	100.0	111	110	0	0	0
	Amazon→WPI	Win	30.9	3	1	100.0	108	107	0	0	0
Streaming Video	Yahoo→DSL	Win	12.3	9382	121	0.3	111	130	99.6	8	8
	Yahoo→WPI	Win	24.0	2319	7	1.1	143	151	12.0	12	13

together. First for the data piggybacking approach we found that the reverse channel throughput can match the effective reverse bandwidth limit without negative effects on forward channel throughput. Second, the number of reverse channel packets generated to achieve this throughput is significantly less than a simple bidirectional transfer over one connection or two independent connections. Even in the case of an application-only approach with no TCP implementation support there is a reduction in the number of reverse-channel packets. Third, even in the case of asymmetric links such as a home DSL connection, data piggyback sizes up to a few hundred bytes per packet can be supported in either an application-only or TCP-level implementation without reducing forward-channel throughput nor having an appreciable effect on the number of reverse-channel packets.

We also obtained many useful results from the enhanced ACK experiments to maintain more complete packet transmission information at the receiver in a TCP connection and share this information with the sender via a new Timestamps Option. A desirable feature in a TCP connection is to be able to know the spacing between data packets received at the receiver and hence establish jitter in the forward data transmission channel. Currently this spacing can only be inferred from spacing between ACK generation at the receiver or ACK reception at the sender. However, results from many file transfer experiments show discrepancies between the spacing of ACKs and the actual spacing of received data packets. In one file transfer experiment we found a difference between these spacings in over 50% of cases with an average difference of 60ms. These differences are due to reverse channel congestion, variation when data packets are received, which is masked with the delayed ACK feature, as well as delays when ACK packets are generated by the receiver. These differences occur more frequently with generally a larger magnitude when traffic is not sent as frequently as allowed by TCP as can be the case with Web or streaming data. All of these variations, combined with packet loss, make the determination of the actual RTT difficult for the sender resulting in conservative estimates to be made in determining the RTO for the connection.

An important point concerning the availability of this more complete and accurate information about the TCP connection is that it is obtained with minimal, bounded overhead by the TCP receiver and only a small amount of additional information added to an ACK packet that must be sent anyways. Results from our data piggybacking experiments show that adding a small number of bytes to reverse-channel traffic over even a bandwidth-constrained link can be accommodated.

An interesting question about the two approaches is whether they improve or exacerbate problems that occur due to dropped or out-of-order packets. In the case of data piggybacking and forward-direction congestion, duplicate ACKs in the reverse direction provide more opportunities for piggybacking. However, too much data piggybacked in the reverse direction could cause congestion and negatively affect forward throughput. If dropped or out-of-order packets occur with the enhanced ACK approach as did occur in some of our experiments then its additional information provides a more complete picture of what is happening with the data transmission, although the approach would complement, not replace, the existing selective ACK option [12] in retransmission of missing packets.

Another interesting question about the two approaches is whether they can and should be combined. Because the enhanced Timestamps approach is proposed as a TCP option it could be combined with the data piggybacking approach. In terms of whether the combination makes sense, the enhanced ACK information could help both endpoints of a connection determine the available bandwidth between them, although use of the proposed

option in the direction of data flow would cause a small reduction in the per-packet data capacity for that direction. Further research is needed to study the relative tradeoffs of combining the approaches.

X. Summary

In this work we have proposed and evaluated two independent approaches for how to make better use of the available space in TCP ACK-only packets to improve data transfer. The first approach provides packet-efficient throughput in the reverse direction of a connection without sacrificing forward throughput, while the second approach provides more detailed and complete information about the state of the forward connection that could be used by a TCP implementation to obtain better throughput under different network conditions.

Results from the work also lead to a number of directions for future work. First, both the data piggyback mechanism and the enhanced Timestamps Option need to be implemented and tested. Second, for the enhanced Timestamps Option, an obvious direction is to investigate potential improvements to TCP implementations using the more detailed and complete timestamp information. Third, for asymmetric connections, too much data transmission in the reverse direction negatively affects forward-direction throughput. Mechanisms for the data piggybacking approach to monitor and adjust to available bandwidth limits, such as with the proposed enhanced ACK option, need to be investigated. Finally, wireless network performance could improve by using the enhanced Timestamps Option to help a sender determine whether transmission problems occur because of congestion or packet loss.

References

[1] Hossam Afifi, Omar Elloumi, and Gerardo Rubino. A dynamic delayed acknowledgment mechanism to improve TCP performance for asymmetric links. In *Proceedings of the IEEE Symposium on Computers and Communications*. IEEE, June/July 1998.

[2] M. Allman, V. Paxson, and W. Stevens. TCP congestion control, April 1999. RFC 2581.

[3] H. Balakrishnan, V.N. Padmanabhan, G. Fairhurst, and M. Sooriyabandara. TCP performance implications of network path asymmetry, December 2002. RFC 3449.

[4] Hari Balakrishnan, Venkata Padmanabhan, and Randy H. Katz. The effects of asymmetry on TCP performance. *Mobile Networks and Applications*, 4:219–241, 1999.

[5] Lawrence S. Brakmo and Larry L. Peterson. TCP Vegas: End to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, October 1995.

[6] Robert Buff and Arthur Goldberg. Web servers should turn off Nagle to avoid unnecessary 200 ms delays, April 1999. `http://www.cs.nyu.edu/artg/research/speedingTCP/buff_goldberg_speeding_up_TCP.ps`.

[7] Bram Cohen. Incentives build robustness in BitTorrent, May 2003. `http://www.bittorrent.com/bittorrentecon.pdf`.

[8] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[9] Chengpeng Fu, Ling Chi Chung, and Soung C. Liew. Performance degradation of TCP Vegas in asymmetric networks and its remedies. In *Proceedings of the IEEE International Conference on Communications*, June 2001.

[10] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance, May 1992. RFC 1323.

[11] Lampros Kalampoukas, Anujan Varma, and K.K. Ramakrishnan. Two-way TCP traffic over rate controlled channels: Effects and analysis. *IEEE/ACM Transactions on Networking*, 6(6):729–743, December 1998.

[12] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgement options, October 1996. RFC 2018.

[13] S. McCreary and K. Claffy. Trends in wide area IP traffic patterns: A view from Ames Internet Exchange. In *Proceedings of the ITC Specialist Seminar on IP Traffic Modeling, Measurement and Management*, September 2000. `http://www.caida.org/outreach/papers/AIX0005/AIX0005.pdf`.

[14] J. Nagle. Congestion Control in IP/TCP internetworks, January 1984. RFC 896.

[15] Lopa Roychoudhuri, Ehab Al-Shaer, Hazem Hamed, and Greg Brewster. On studying the impact of the internet delays on audio transmission. In *Proceedings of the IEEE Workshop on IP Operations and Management*, October 2002.

[16] Hao Shang. *Exploiting Flow Relationships of Network Applications*. PhD thesis, Computer Science Department, Worcester Polytechnic Institute, May 2006.

[17] Rishi Sinha, Christos Papadopoulos, and John Heidemann. Internet packet size distributions: Some observations, October 2005. `http://netweb.usc.edu/~rsinha/pkt-sizes/`.

[18] Sprint IP monitoring project, packet trace analysis, February 2004. `http://ipmon.sprint.com/packstat/packetoverview.php`.

[19] K. Thompson, G. J. Miller, and R. Wilder. Wide-Area Internet Traffic Patterns and Characteristics. *IEEE Network*, 11:10–23, November 1997.

[20] Bryan Turner. Generalizing bittorrent: How to build data exchange markets (and profit from them!), January 2005. `http://www.fractalscape.org/GeneralizingBitTorrent.htm`.

[21] Bryan Veal, Kang Li, and David Lowenthal. New methods for passive estimation of TCP round-trip times. In *Proceedings of the Passive and Active Measurement Workshop*, Boston, Massachusetts, March/April 2005.

[22] Ren Wang, Massimo Valla, M.Y. Sanadidi, and Mario Gerla. Using adaptive rate estimation to provide enhanced and robust transport over heterogeneous networks. In *Proceedings of the International Conference on Network Protocols*, Paris, France, November 2002. IEEE.

[23] L. Zhang, S. Shenker, and D.D. Clark. Observations and dynamics of a congestion control algorithm: The effects of two-way traffic". In *Proceedings of ACM SIGCOMM*, pages 133–147, 1991.