ORNL/TM-2002/76

A TCP-OVER-UDP TEST HARNESS

Tom Dunigan
Florence Fowler

Network Research Group
Oak Ridge National Laboratory
P.O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367
thd@ornl.gov ff8@ornl.gov

Date Published: May 9, 2002

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831-6285
managed by
UT-Battelle, LLC
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from the Office of Scientific and Technical Information. P. O. Box 62, Oak Ridge, TN 37831; prices available from (615) 576-8401.

Available to the public from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161.

Contents

Acknowledgements

This research was supported by the High-performance networking program, Office of Science, U.S. Department of Energy under contract DE-AC05-00OR22725 with UT-Battelle, LLC. Computing resources were provided by the Center for Computational Science at Oak Ridge National Laboratory and National Energy Research Scientific Computing Center. A special thanks to the Probe project for its support.

Abstract

This report describes an implementation of a TCP-like protocol that runs over UDP. This TCP-like implementation, which does not require kernel modifications, provides a test harness for evaluating variations of the TCP transport protocol over the Internet. The test harness provides a tunable and instrumented version of TCP that supports Reno, NewReno, and SACK/FACK. The test harness can also be used to characterize the TCP-performance of a network path. Several case studies illustrate how one can tune the transport protocol to improve performance.

1 Introduction

As part of our efforts in improving bulk transfers over high speed, high latency networks, we have developed an instrumented and tunable version of TCP that runs over UDP (*atou*, "almost TCP over UDP"). The TCP-like UDP transport serves as a test harness for experimenting with TCP-like controls at the application level and for characterizing network paths. The implementation provides optional event logs and packet traces and can provide feedback to the application to tune the transport protocol, much in the spirit of web100 [21] but without the attendant kernel modifications. Part of the motivation for developing such a test harness arose from bugs we discovered in the AIX TCP stack.

In the next section, we give a brief review of the TCP transport protocol. In section 3, we review other mechanisms for evaluating TCP. Section 4 describes the implementation of our TCP-over-UDP test harness. Section 5 describes experiences with tuning the transport protocol in various network settings. Section 6 describes related work. The final section summarizes our work and considers future extensions to the test harness. There is also an appendix that describes how to build and use the test harness.

2 TCP tutorial

Since the objective of our UDP protocol is to behave like TCP, we will briefly summarize TCP's characteristics in this section. TCP provides a reliable data stream over IP. IP is a packet-based protocol, where packets can be lost, duplicated, corrupted, or can arrive out of order. TCP uses sequence numbers, checksums, positive acknowledgements, and timeouts/retransmissions to provide a reliable data stream. TCP is used for most of the Internet services such as mail, http, telnet, and ftp.

TCP tries to send data in MSS-sized packets. MSS, maximum message size, is negotiated when the TCP connection is established. The MSS should reflect the minimum MTU of the interfaces comprising the path. (Path MTU discovery can be used by the hosts to find the appropriate MSS [23].) The MSS for Ethernet networks is usually 1460. Part of our TCP-over-UDP protocol study considers using larger segment sizes, though most TCP implementations have little control over MSS. Sending data segments larger than the MTU causes IP to fragment the segment, and IP fragmentation usually hurts network performance

[18]. We can avoid fragmentation by creating a "virtual MSS" by sending k segments at initial startup and after a timeout, and by adding k segments per RTT during linear recovery.

TCP uses a sliding-window protocol to implement flow control. The receiver advertises his receiver window which indicates the maximum number of bytes he can currently accept. The transmitter must not send more data than the receiver window permits. Both transmitter and receiver must reserve buffer space equal to this window size. The operating system has a default window size, but the application program can change this value. The window size is the most critical parameter affecting TCP performance, yet many applications do not support a way for changing this value.

The window size should be equal to the product of the round-trip time times the bandwidth of the link. The default window size for most OS's is 16K or 32K bytes. For high bandwidth, high delay links, this is usually inadequate. For example, for a RTT of 60ms on a 100Mbs link, when the receiver has a 16K window, the maximum throughput will be only 266 KBs! To reach full bandwidth on this link, the receiver needs to have a window of 750KB. (The sender also needs a 750KB buffer because he must buffer all data until it is acknowledged by the receiver.) This reveals another problem for the user, the size of the window field in the TCP packet is only 16-bits. To support window sizes larger than 64 KB, the OS must support window-scaling, a relatively new TCP option [17].

Prior to 1988, TCP senders would usually blast a window-full of data at the receiver. Van Jacobson [16] noticed that this was leading to congestion collapse on the Internet. TCP was modified to provide congestion avoidance. When a sender started up (slow start), it would send one segment (MSS), and then send one additional segment for each ACK, until the receiver's window was reached or a packet was lost. (Slow-start grows the window exponentially.) TCP uses a variable, *cwnd*, to keep track of this collision window. If a packet was lost as detected by a time out or three consecutive duplicate ACK's, the transmitter would halve *cwnd* (multiplicative decrease) saving the value in $ssthresh$, resend the "lost" packet (fast retransmit), then set *cwnd* to one and go through slowstart again til it reached $ssthresh$. After reaching $ssthresh$, the sender would increment *cwnd* only once per round-trip time (congestion avoidance phase, additive increase). Thus TCP tests the link for available bandwidth until a packet loss occurs and then backs off and then slowly increases the window again. This additive-increase-multiplicative-decrease converges in the face of congestion and is fair. (TCP's notion of fair is that if N TCP flows share a link, each should get about 1/Nth of the bandwidth.)

The most widely deployed implementation of TCP is Tahoe [25]. Tahoe added delayed ACK's – the receiver can wait for two segments (or a timer) before sending the ACK. This can slow both slow-start and congestion avoidance since both are clocked by incoming ACKs. Tahoe also added fast retransmit, rather than waiting for a timeout to indicate packet loss. When the transmitter receives three consecutive duplicate ACK's, it resends the lost packet.

More recent enhancements to TCP were called Reno [5], NewReno [13], SACK [19], and FACK [22]. Reno adds fast recovery to fast retransmit, once the collision window allows, new segments are sent for subsequent duplicate ACKs. NewReno handles multiple losses within a window and partial ACKs. SACK has the receiver send additional information in its ACK about packets that it has received successfully. This way the sender can re-send missing segments during fast retransmit and move the left-edge of the window. Often recovery is exited upon the receipt of an ACK cumulatively acknowledging a large number of packets, this allows the sender to transmit a burst of packets.

3 TCP over UDP

The experimental TCP-over-UDP protocol that we developed includes segment numbers, time stamps, selective ACKs, optional delayed ACKs, sliding window, timeout-retransmissions and various parameters to control window/MSS sizes and congestion avoidance. The test harness consists of a configurable client (transmitter) and an ACK-server (receiver). The transmitter supports Reno, NewReno, and SACK/FACK. The protocol transmits segments on the first two dup ACKs (limited transmit [2]).

The protocol header includes timestamp and segment number. The timestamp, from *gettimeofday()* , has higher resolution than Linux TCP timestamp (*jiffies*). The receiver's header includes the transmitter's timestamp, the segment number of the last good contiguous segment and up to three optional two-word blocks of SACK info (Figure 1). The cumulative ACK is just like TCP's, with optional delayed ACK support where the ACK delay may be specified as well.

A simple configuration file controls the transmitter. Table 1 lists the variables that can be configured on the transmitter. The receiver does not advertise a receive window, so the transport window is controlled from the transmitter. The transmitter can be configured with

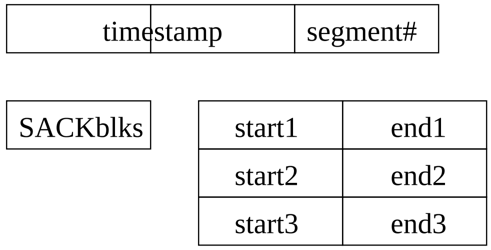

Fig. 1. Packet headers.

various MTU/MSS and window sizes, and various congestion avoidance algorithms including variables to alter the Additive-Increase Multiplicative Decrease (AIMD) parameters. The receiver can be configured with a RCVBUF size, port number, SACK, delayed-ACK time value, and debug level. The receiver keeps track of missing and duplicate segments and supports a TCP-like cumulative acknowledgement scheme with optional SACK support.

mss	UDP datagram size, normally 1472
rcvrwin	receiver window size in segments (20)
newreno	if 1, use newreno rather than reno (1)
sack	if 1, use selective ack (0)
fack	if 1, use Mathis selective ack (0)
rampdown	if 1, use Mathis rampdown FACK option (0)
timeout	timeout for retransmission (1 second)
tick	select timer, 0.5 seconds
maxidle	number of seconds with no ACKs before giving up (10)
initsegs	number of segments in initial window (2)
multiplier	cwnd = multiplier*cwnd if loss (0.5)
increment	in congestion avoidance, how much to add to cwnd each RTT (1)
thresh_init	percent of rcvwin to initially set ssthresh (1)
dup_thresh	number of dup ACKs to trigger re-transmit (3)
burst_limit	max number of segments to send at once (0 == unlimited)
maxpkts	how many segments to send (0, unlimited)
maxtime	number of seconds to send (10)
port	UDP port number (7890)
sndbuf	UDP SNDBUF size (32768)
rcvbuf	UDP RCVBUF size (32768)
droplist	list of segments to drop
debug	level of debugging (0)

Table 1. Configurable parameters for the transmitter.

The transmitter behaves like TCP with Reno, Newreno, or SACK/FACK using slow-start and congestion avoidance. Upon termination a summary of the configuration and session statistics are printed as below.

```
config: portcli $Revision: 1.29 $ port 7890 debug 4 Tue May 22 05:18:54 2001
config: initsegs 2 mss 1460 tick 1.000000 timeout 0.500000
```

```
config: rcvrwin 300  increment 1  multiplier 0.500000 thresh_init 1.000000
config: newreno 1 sack 0 delack 0 maxpkts 12000  burst_limit 0 dup_thresh 3
config: sndbuf 32768 rcvbuf 32768
swift => stingray.ccs.ornl.gov 38.835049 Mbs win 300 rxmts 0
3.609111 secs 17520000 good bytes goodput 4854.381103 KBs 38.835049 Mbs
pkts in 11542  out 11999  enobufs 0
total bytes out 17518540 loss  0.000 % 38.831813 Mbs
rxmts 0 dup3s 0 packs 0 timeouts 0  dups 0 badacks 0 maxack 24 maxburst 300
minrtt  0.065249 maxrtt 0.098987 avrgrtt 0.077747
rto 0.067194  srtt 0.066705  rttvar 0.000122
win/rtt = 45.069119 Mbs  bwdelay = 377 KB  258 segs
snd_nxt 12000 snd_cwnd 300  snd_una 12000 ssthresh 300 snd_max 12000
```

With a debug level of 4 or higher, a trace file is produced for each transmit and ACK packet. The data includes time stamp, segment number, round-trip time, congestion window, and threshold. The following illustrates trace file contents.

```
0.408133 258 xmt
0.408196 234  0.038062  26 26 ack
0.408259 259 xmt
0.408346 235  0.038069  26 26 ack
```

From the trace file, one can plot tcptrace-like information (RTT over time, cwnd over time, average and instantaneous bandwidth, ack-sequence number, etc). Writing the trace file can slow performance for high speed interfaces. With the drop-list, one can induce loss for testing purposes. The receiver also reports effective bandwidth, number of ACKs sent, and number of segments dropped (or arriving out of order) and the number of duplicated segments.

Depending on the debug level, events (timeouts, retransmissions, etc.) are logged on stderr. The events in Figure 2 illustrate a retransmit resulting from three duplicate ACKs, and a retransmit resulting from a timeout. The *packrxmit* are NewReno (or SACK) retransmits during "recovery". A *badack* is an out of order ACK or duplicated ACK.

The TCP-like UDP transport also can make good round-trip estimates from the time stamps. Most TCP stacks use a coarse (0.5 second) resolution timer for estimating round-trip time, and TCP times only one packet per round-trip time. The TCP estimates are complicated by retransmissions and delayed/cumulative ACKs. Newer TCP implementations

```
3duprxmit pkt 416 nxt 616 max 616  cwnd 200  thresh 200 recover 0
timerxmit pkt 417 snd_nxt 616 snd_max 617  snd_cwnd 100  thresh 100
3duprxmit pkt 2744 nxt 2844 max 2844  cwnd 100  thresh 100 recover 0
packrxmit pkt 2747 nxt 2844 max 2845 cwnd 100  thresh 50 recover 2844 una 2744
packrxmit pkt 2750 nxt 2845 max 2845 cwnd 98  thresh 50 recover 2844 una 2747
badack 6636 snd_max 6808 snd_nxt 6808 snd_una 6661
badack 6892 snd_max 7048 snd_nxt 7048 snd_una 6899
```

Fig. 2. Events recorded on stderr.

support a time-stamp option. From analysis of the time-stamp data for our UDP protocol over the ORNL-NERSC link, we chose to initially implement a simple one-second timeout. (Timeouts are rare, and packet loss is usually detected by multiple duplicate ACKs.)

4 Case studies

We have used our TCP-over-UDP protocol on a wide variety of networks including DOE's ESnet (OC12 and OC3), a dual ISDN link, Ethernet, FDDI, GigE, wireless (802.11b), and HiPPI2/GSN links with 64K MTU (reached 1.3 Gbs). We have compared our TCP-over-UDP performance with TCP over the same paths. We have use the protocol to evaluate the effects of altering various TCP control parameters and to characterize the performance of network paths.

1. ORNL-NERSC As noted in the introduction, much of our protocol study is directed toward characterizing network paths and improving bulk transfer data rates over high bandwidth, delay paths. Most of our experiments have been conducted over ESnet on both OC12 and OC3 links, fed by both 100T and Gigabit Ethernet hosts. We conducted various experiments between hosts at ORNL and NERSC in California using different buffer sizes, different TCP options within AIX, and different interfaces. We evaluated both TCP and our TCP-like UDP transport. Figure 3 illustrates the throughput behavior of a TCP-over-UDP flow with no loss. TCP (and our TCP-like UDP protocol) starts with a small window and then exponentially increases it until loss occurs or the available bandwidth or receiver's window size is reached. As can be seen in the figure, in less than a second, available bandwidth is reached. The rate or time to reach peak can be calculated analytically as well, since one additional segment (MSS-sized) is transmitted for each ACK received, which takes one round-trip time. We can't do anything about the round-trip time, so if we wish to improve the startup performance we need to have a larger MSS or try a larger initial window size. Our UDP transport has options for an increasing the MSS and initial window size. When TCP reaches a steady-state, the transmitter is regulated by the ACK arrival rate. Note that it takes many seconds over a high delay/bandwidth path (such as our ORNL-NERSC path) to reach capacity. The startup rate can be slowed by a factor of two if the receiver utilizes delayed-ACKs (sending an ACK for every other segment received).

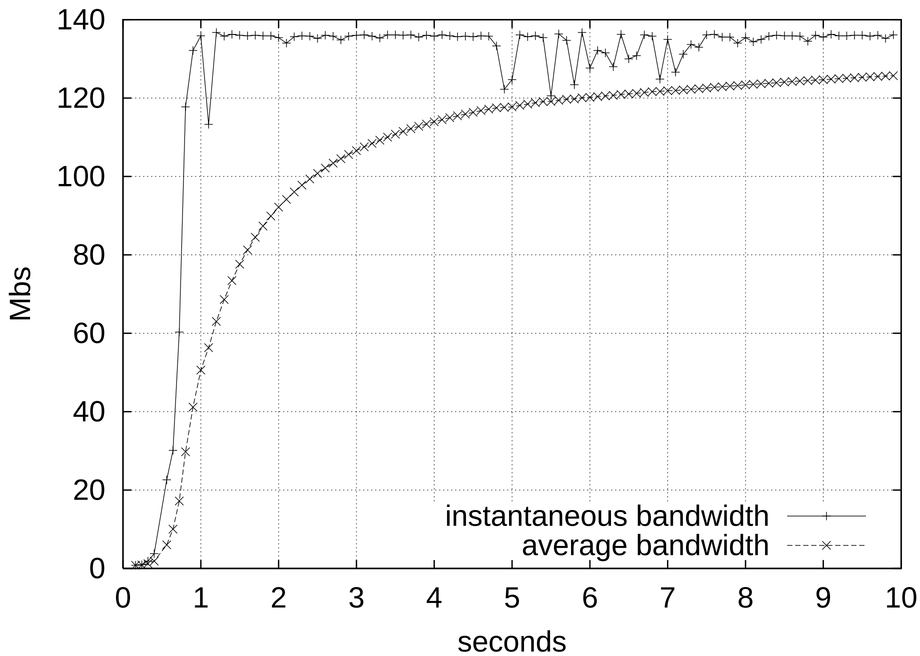

Fig. 3. Average and instantaneous (0.1 sec samples) bandwidth for transfer from stingray to swift using our TCP-like UDP transport.

To improve this recovery rate, one either needs a bigger MSS or a larger additive increase in the recovery algorithm. (TCP adds one new segment per RTT.) Using our UDP transport or the simulator, we can experiment with altering this increment. (Increasing the increment turns out to be "fair", because as it is, if multiple TCP flows are sharing a link and there is congestion, they all halve their rates, but the nodes nearer (shorter RTT) will get more of the bandwidth than the distant node.) As with startup, a delayed-ACK receiver also slows down the linear recovery. To further speed recovery, one could reduce *cwnd* by less than half. We have done a few experiments adjusting these AIMD parameters with our test harness. Figure 4 compares standard AIMD parameters (0.5,1) with setting cwnd to 0.9 of its value and adding 10 segments per RTT. With such aggressive behavior, one must worry about being fair to other competing flows [12]. There are a number of papers that look at increasing TCP's initial window size and altering the AIMD parameters [3] [9] [4] [15].

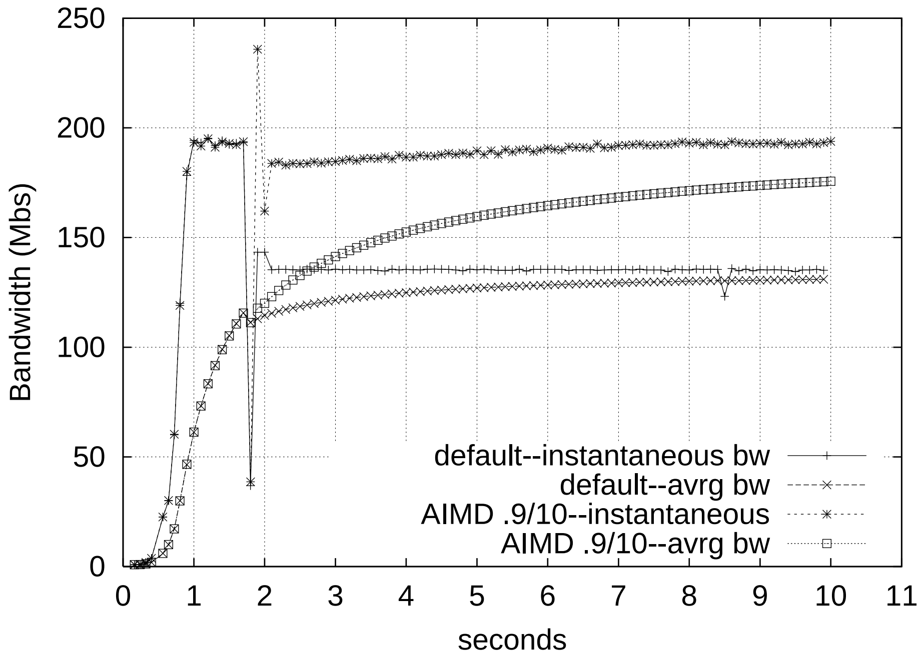

Fig. 4. Standard and aggressive AIMD recovery from a single packet loss using our TCP-like UDP protocol between NERSC and ORNL.

Most TCP-stacks utilize delayed ACKs [1], our UDP transport can disable delayed-ACKs and profit from a slightly faster startup rate. The effect of delayed-ACK's on throughput is evident in Figure 5. The figure shows average and instantaneous throughput of a TCP-like transfer with and without delayed ACK's. Using the droplist in our TCP-like UDP transport, we have dropped two packets in two transfers from ORNL to NERSC. Even though the transfers are competing with other Internet traffic, the non-delayed ACK transfer starts up faster and recovers from the loss faster.

Using the *ns* [8] simulator, Figure 6 compares average and instantaneous throughput for an Ethernet-sized segment (1500B) versus a jumbo-frame segment (9000B). As noted earlier, both startup and recovery times are effected by the segment size, and TCP generally favors flows with larger segment sizes. ATM supports a 9K segment size, and FDDI supports a 4K segment, so if both endpoints had different network interfaces and the intervening path supported the larger segment size, then throughput should be improved.

With our TCP-like UDP transport, we can experiment with larger MSS sizes. We used

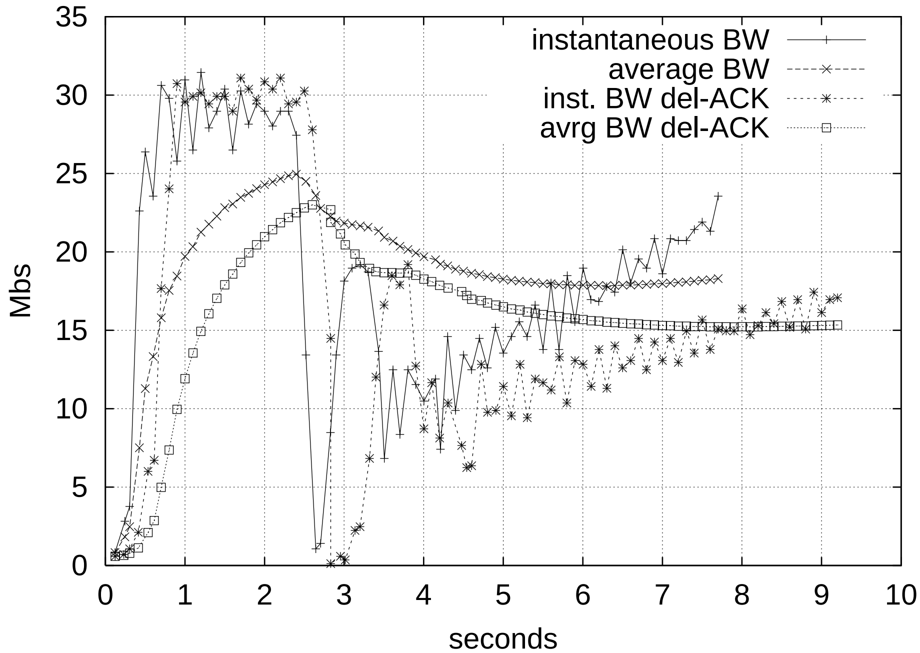

Fig. 5. Average and instantaneous throughput with and without delayed-ACKs using our TCP-like UDP protocol between NERSC and ORNL.

a larger MSS (2944 and 5888 bytes) with our UDP transport and got better throughput. For the same number of bytes transferred and the same effective window size we got 50 Mbs with 1472-byte datagram, 55 Mbs with 2944-byte datagram, and 58 Mbs with 5888-byte datagram. (Figure 7) The UDP datagrams greater than the 1500-byte Ethernet MTU are fragmented by IP. These larger datagrams have an effect similar to TCP's delayed-ACK, in that all fragments (MTU-sized) have to arrive before the receiver ACK's the datagram. Kent and Mogul, however, argue that IP fragmentation usually lowers network performance.

The same effect can be gained by using a "virtual MSS", that is, choosing an initial startup of K segments (*initsegs*) and then adding K segments per RTT (*increment*) during recovery. The virtual MSS avoids the IP fragmentation [15]. Figure 8 illustrates a transfer from NERSC to ORNL with two packet drops using the default MSS and then using a virtual MSS of 10 segments. The virtual MSS is also used for slow start, so there is improvement in starting the connection as well.

The configuration values *initsegs* and *thresh_init* affect the rate and duration of slow start. As noted above, the initial window size (*initsegs*) can be used to create a virtual MSS, and there are several papers [3] [4] on choosing the initial window size. Slow start stops when

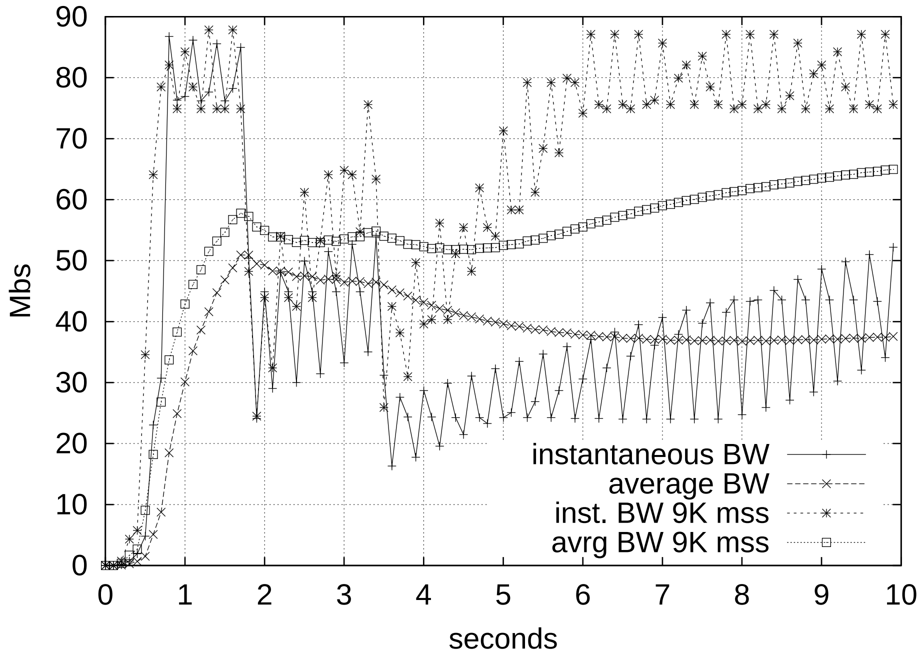

Fig. 6. Average and instantaneous throughput for 1.5K mss versus 9K mss. (ns)

ssthresh is reached. In most TCP implementations, the initial value of *ssthresh* is infinite, and is set to *cwnd/2* when congestion occurs. If the transfer is not window-limited, our experience has been that packet loss often occurs during initial slow start, so one can set *thresh_init* to some fraction of *rcvrwin* to try to avoid this early loss. Setting *thresh_init* to zero makes the inital value of *ssthresh* infinite.

2. UT-ORNL An OC3 (150 Mbs) connects ORNL and UT. We tested throughput between two 100 Mbs hosts and found that the ATM/OC3 has been provisioned as a VBR link. Using rate-based flooding of UDP packets, we can reach nearly 100Mbs for a short while, then the traffic policing cuts the sustained rate to about 50 Mbs. We saw similar behavior using AAL5/ATM. The VBR policy has an interesting affect on TCP, letting it ramp up during slow-start, then dropping segments to bring the sustained rate back down. Choosing a window of about 22KB avoids any packet loss and achieves a bandwidth of 41 Mbs. Larger window sizes induce loss and reduce bandwidth, smaller window sizes cause the transfer to be buffer-limited.

3. ORNL-cable We tested throughput and loss characteristics from a home cable-modem system to ORNL. The route is asymmetric and tortuous, some 20+ hops all the way to the west coast and back. During one test period, the route into the home cable modem

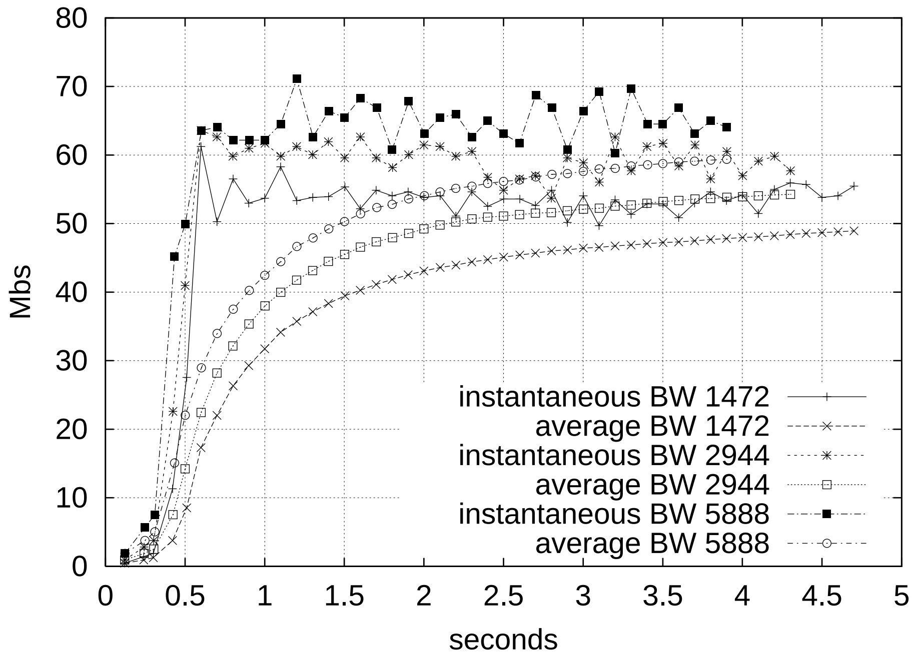

Fig. 7. Average and instantaneous throughput for various datagram sizes.

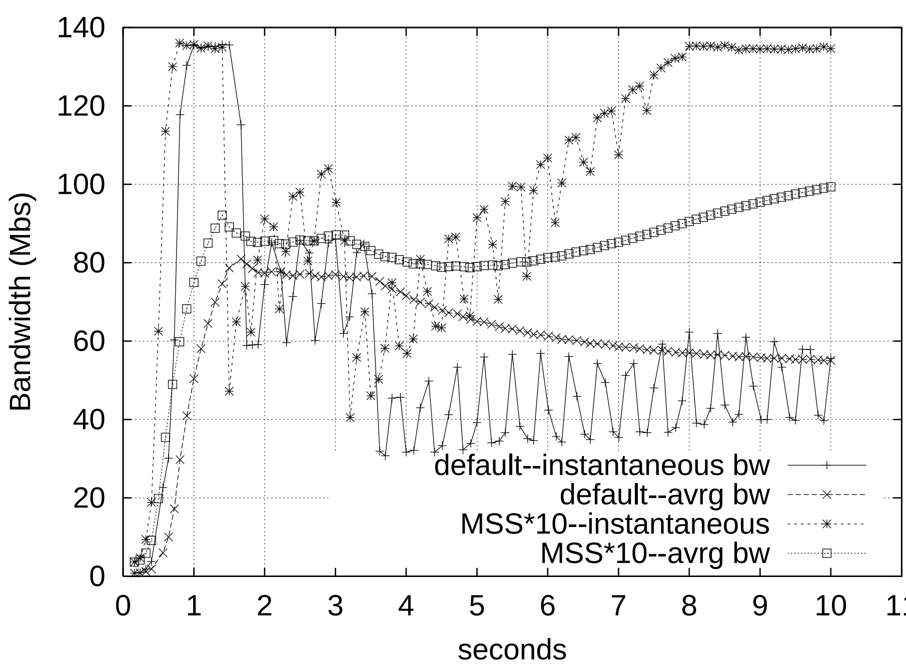

Fig. 8. Average and instantaneous throughput for virtual MSS of 10x.

appeared to split the traffic across multiple routes. Rate-based UDP flooding could reach 6+ Mbs in to the home without loss, but our test software showed that the UDP packets were wildly out of order (60% of the packets). Packets could be displaced by as much as 30 positions. TCP inbound throughput over this path could only reach 700 Kbs, because TCP treats 3 out of order packets as a loss and does a retransmit. With our TCP-like UDP protocol, we increased the "dup threshold" to 9 to account for the massive re-ordering and improved the throughput to 1.7 Mbs.

4. ORNL-cable (ACK-limited) The cable modem system we were testing limits data rates from the home to 128 Kbs. This limit in combination with the mutli-hop route actually limited our throughput into the home. Our TCP tests into the home cable modem were actually faster than what we could achieve with our TCP-over-UDP. Our TCP-over-UDP was not using delayed ACKs so there was twice as much bandwidth being consumed on the ACK path, and we were sending empty SACK blocks, further wasting ACK bandwidth. We trimmed off the empty SACK blocks in the ACK path and then enabled delayed ACKs in our TCP-over-UDP software and that permitted us to meet or exceed the TCP bandwidth. So for this configuration, delayed ACK's provided a speedup.

5 Related work

The primary objective of our TCP-over-UDP was to provide a test harness for evaluating the effect of altering TCP parameters. If one has kernel access, one can of course modify the kernel to experiment with TCP. We also have done some kernel-level testing over a NISTNet [24] testbed and as part of our Net100 [10] project. In addition, we have done protocol experiments with the *ns* [8] simulator.

The other application of our TCP-over-UDP protocol was to characterize path performance for a TCP-like protocol. The Treno tool [20] also tries to characterize the TCP behavior of a path using UDP or ICMP packets with sequence numbers, timeouts, and a TCP-like slow-start and congestion avoidance. Treno is a sender-side only test, relying on ICMP packets from the target to return the transmitted sequence numbers. More recently, Allman [6] describes a TCP-over-UDP implementation, *cap*, for measuring bandwidth carrying capacity of a path. Our transport protocol can provide detailed performance information on a given flow, the Web100 project [21] also provides detailed TCP-flow information using a modified Linux kernel.

6 Summary

We have tested our TCP-like protocol on a variety of networks including on our NISTnet [24] emulation test bed. We have tested the protocol on a number of operating systems, including Linux, Solaris, Sun OS, Irix, FreeBSD, Compaq's OSF, and AIX. We also did a port to Windows as well, though the source files do not contain those modifications. Where TCP could be tuned, we tried to get TCP to match or exceed the UDP client/server data rate. The TCP-like protocol has provided higher throughput in some cases than TCP.

The software provides both a time-stamped packet log and event-based information (entering/leaving recovery, timeouts, out-of-order ACKs, etc.) The debug modes give useful information on which packets were lost, window/threshold changes, bandwidth, round-trip times, and the classical sequence/ACK number graphs. The event and trace data have been very useful in understanding packet loss and throughput on various paths.

The TCP-like UDP protocol is appealing because it does not require kernel mods to tweak TCP-like congestion/transport parameters. We have demonstrated striking changes in performance when modifying our transport's buffer size, MSS size, or the AIMD parameters. Disabling delayed ACKs and altering the dup threshold has also improved performance in some networks. For multiple losses within a window, the SACK/FACK options improved recovery as expected. We did not see much effect in altering the burst limit or the initial ssthresh value.

Our TCP over UDP can do some things that most TCP implementations cannot do. We can easily change the TCP-like control parameters without making kernel modifications. Parameter modification will apply to only one flow, so we can can tune each flow. The application using *atou* can be network-aware and can modify in real time the TCP-like control parameters based on feedback from the packet processing layer (not unlike Web100 [21]). We can collect event and trace logs on a particular flow, though we could get similar information from tcpdump/tcptrace. We can retain the TCP-like control information across runs and pre-set various control parameters based on previous runs. We can have an MSS bigger than the MTU, though IP fragmentation results, or create a "virtual MSS". There is no Nagle algorithm, so we need not worry about disabling Nagle to improve performance.

On the other hand, our TCP-over-UDP cannot do some things that TCP can do.

- Our *atou* protocol is one-way, a transmitter and a receiver.
- Handling the UDP packets at the application level requires a context switch, and if the application is busy, UDP packets may be lost if the RCVBUF is not large enough.
- The packet handling delay can also contribute to ACK compression.
- The UDP transport must do a read/write for each MSS-size segment suffering the OS context switch. TCP can do large write's reducing OS overhead.
- Timer management for sender timeouts or delayed ACKs at the receiver is less efficient than a kernel implementation.
- Adding an "application layer", e.g. a file transfer, will require multi-threading and buffer management that could reduce effective throughput.
- We calculate RTO using the timestamps in the packet, but we do not use the RTO value at this time. We just use a 1 second timeout for our tests, that serves our uses to date. We can alter the tick resolution and support timeout's less than 1 second.
- Our protocol uses segment numbering rather than byte numbering for sequencing.
- Our receiver does not send back "available" window information.
- Our UDP won't be able to see ECN [11].

We continue to test and extend the implementation. If we can show the transport provides a distinct advantage over TCP, then we need to wrap it with a file transfer protocol. This is a non-trivial extension, requiring buffering at both transmitter and receiver, threads, and flow-control (like TCP's sliding window). We have also considered having a file transfer protocol that could take packets out of order, presumably keeping the receiver's advertised window open.

We've also looked at retaining transport control data on a per destination basis. Since our TCP-like UDP transport protocol is at the application layer, the application can get feedback from the transport (RTT, retransmits, timeouts, duplicate ACK counts, cwnd, ssthresh, bursts, cumulative ACK info, interarrival times) and possibly use this information to alter transport parameters and save the information on a link (e.g., RTT and variance, ssthresh and cwnd) for use in optimizing a later transfer to the same target. (Linux 2.4, OpenBSD and FreeBSD save this kind of data in the routing table.) It would be interesting to experiment with these possibilities.

Another protocol optimization to try is speculative recovery (done by Linux 2.4 and FreeBSD) where we would save ssthresh and cwnd on a timeout, then if the ACK for the retransmitted packet arrives "soon" (it was inflight), restore ssthresh and cwnd (and maybe increase RTO). Speculative recovery might also be useful where FACK-induced recovery (rather than 3-dup's) is initiated, yet the FACK block was only a result of an out-of-order packet. Our receiver gives a diagnostic summary of duplicates received, but we need to send that information back to the sender, like D-SACK [14]. With this information the transmitter can alter dup threshold or undo a window halving.

For easier systematic testing, the client-server could be extended with a TCP control channel that could be used to set parameters at the server side and retrieve results. With this structure we could have a persistent server (daemon). The drop list has proven effective in evaluating recovery options, but the client loss model could be extended to include loss probability functions (like *ns* or NISTNet).

We are working on a TCP Vegas [7] extension to our protocol. Vegas attempts to avoid losses by sensing congestion from its realtime bandwidth calculations. Vegas does not compete well with other TCP protocols, so we would also tune the alpha and beta parameters of Vegas to try and make it more aggressive.

References

[1] M. Allman. On the generation and use of tcp acknowledgments. *ACM Comp Commun. Rev.*, 28:4–21, October 1998.

[2] M. Allman, S. Floyd, and H. Balakrishnan. Enhancing TCP's Loss Recovery Using Limited Transmit. *RFC 3042*, January 2001.

[3] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. *RFC 2414*, September 1998.

[4] M. Allman, C. Hayes, and S. Ostermann. An evaluation of tcp with larger initial windows. *Computer Communication Review*, 28(3), July 1998.

[5] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. *RFC 2581*, April 1999.

[6] Mark Allman. "Measuring End-to-End Bulk Transfer Capacity", 2001. URL: http://www.aciri.org/vern/imw-2001/imw2001-papers/14.pdf.

[7] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP vegas: New techniques for congestion detection and avoidance. In *SIGCOMM*, pages 24–35, 1994.

[8] L. Breslau, D. Estrin, K. Fall, S. Floyd, and J. Heideman. Advances in Network Simulation. *IEEE Computer*, pages 59–67, May 2000.

[9] D. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Journal of Computer Networks and ISDN Systems*, pages 1–14, June 1989.

[10] DOE. "NET100 – Development of Network-Aware Operating Systems", 2001. URL: http://www.net100.org.

[11] S. Floyd. TCP and explicit congestion notification. *ACM Computer Communication Review*, 24(5):10–23, 1994.

[12] S. Floyd. Congestion Control Principles. *RFC 2914*, September 2000.

[13] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. *RFC 2582*, April 1999.

[14] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP. *RFC 2883*, July 2000.

[15] T. H. Henderson, E. Sahouria, S. McCanne, and R. H. Katz. On improving the fairness of TCP congestion avoidance. *IEEE Globecomm conference, Sydney*, 1998.

[16] V. Jacobson. Congestion avoidance and control. *ACM Computer Communication Review; Proceedings of the Sigcomm '88 Symposium in Stanford, CA, August, 1988*, 18, 4:314–329, 1988.

[17] V. Jacobson, R. Braden, and D. Borman. RFC 1323: TCP extensions for high performance, May 1992.

[18] C. A. Kent and J. C. Mogul. Fragmentation considered harmful. *WRL Technical Report 87/3*, 1987.

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

[20] Matt Mathis. "Diagnosing Internet Congestion with a Transport Layer Performance Tool". *Proceedings of INET '96*, June 1996.

[21] Matt Mathis. "Web100", 2000. URL: http://www.web100.org.

[22] Matthew Mathis and Jamshid Mahdavi. Forward acknowledgement: Refining TCP congestion control. In *SIGCOMM*, pages 281–291, 1996.

[23] J. Mogul and S. Deering. Path MTU Discovery. *RFC 1191*, November 1990.

[24] NIST. "NISTNet", 2001. URL: http://snad.ncsl.nist.gov/itg/nistnet/.

[25] W. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. *RFC 2001*, January 1997.

Appendix

One should read the README that accompanies the *atou* distribution to get the most recent information on installing and using the protocol. This appendix briefly summarizes the installation and use of the client and server.

A Installation and testing

The distribution consists of the following files.

```
README
atoucli.c
atousrv.c
scoreboard.h
probeall.pl
```

One can compile the server and client with your favorite C compiler.

B Using the client/server

On the target machine, one needs to start the server with, for example, *atousrv -b 1000000 -d 200*. This provides a 1 MB UDP receiver buffer to the kernel and enables delayed-ACKs with a 200 ms timeout. The full set of server options are

```
-s      enable SACK (default reno)
-d ##   amount to delay ACKs (ms) (default 0, often 200)
-p ##   port number to receive on (default 7890)
-b ##   set socket receive buffer size (default 8192)
-D ##   enable debug level
```

You must CTRL-C the server when the client finishes, and restart it for each test you wish to perform. When you kill the server, it reports the number of bytes and packets received and the number of ACKs sent. The server also reports the number of duplicate packets received and dropped or out-of-order packets received.

The client needs a *config* file. A typical minimal config file might be

```
newreno 1
rcvrwin 100
maxtime 10
debug 4
```

This would run the client for 10 seconds using a window of 100 packets and using the NewReno TCP protocol. The debug level of 4 would provide a packet trace file in *db.tmp*. Event and error information is written to stdout and stderr, so one normally runs the client with *atoucli targethost >& event.tmp*. A simple perl script *probeall.pl* is provided that will produce a number of x-y plot files from the trace file (e.g., *probeall.pl < db.tmp*) The following tables lists the variables that can be included in the config for the client.

mss	UDP datagram size, normally 1472
rcvrwin	receiver window size in segments (20)
newreno	if 1, use newreno rather than reno (1)
sack	if 1, use selective ack (0)
fack	if 1, use Mathis selective ack (0)
rampdown	if 1, use Mathis rampdown FACK option (0)
timeout	timeout for retransmission (1 second)
tick	select timer, 0.5 seconds
maxidle	number of seconds with no ACKs before giving up (10)
initsegs	number of segments in initial window (2)
multiplier	cwnd = multiplier*cwnd if loss (0.5)
increment	in congestion avoidance, how much to add to cwnd each RTT (1)
thresh_init	percent of rcvwin to initially set ssthresh (1)
dup_thresh	number of dup ACKs to trigger re-transmit (3)
burst_limit	max number of segments to send at once (0 == unlimited)
maxpkts	how many segments to send (0, unlimited)
maxtime	number of seconds to send (10)
port	UDP port number (7890)
sndbuf	UDP SNDBUF size (32768)
rcvbuf	UDP RCVBUF size (32768)
droplist	list of segments to drop
debug	level of debugging (0)

When the client terminates it produces a summary on stdout

```
config: atoucli $Revision: 1.12 $ port 7890 debug 4 Mon Mar 11 14:21:19 2002
config: initsegs 2 mss 1472 tick 1.000000 timeout 0.500000
config: maxidle 10 maxtime 10
config: rcvrwin 60  increment 1  multiplier 0.500000 thresh_init 1.000000
config: newreno 1 sack 1 rampdown 1 fack 0 delack 0 maxpkts 0
 burst_limit 0 dup_thresh 3
config: sndbuf 65536 rcvbuf 65536
wisp => thistle 33.758920 Mbs win 60 rxmts 11
21.995500 secs 92818040 good bytes goodput 4219.864979 KBs 33.758920 Mbs
pkts in 31786  out 63573  enobufs 0
total bytes out 92816580 loss  0.017 % 33.758389 Mbs
rxmts 11 dup3s 0 packs 0 timeouts 11  dups 0 badacks 0 maxack 1 maxburst 58
```

```
minrtt  0.000591 maxrtt 0.019809 avrgrtt 0.009318
rto 0.010466  srtt 0.008204  rttvar 0.000566
win/rtt = 75.211497 Mbs  bwdelay = 39 KB  26 segs
snd_nxt 63574 snd_cwnd 2  snd_una 63572 ssthresh 2 snd_max 63574
goodacks 31786 cumacks 0 ooacks 0
```

Depending on the debug level, events (timeouts, retransmissions, etc.) are logged on stderr. The events below illustrate a retransmit resulting from receving three duplicate

```
3duprxmit pkt 416 nxt 616 max 616  cwnd 200  thresh 200 recover 0
timerxmit pkt 417 snd_nxt 616 snd_max 617  snd_cwnd 100  thresh 100
3duprxmit pkt 2744 nxt 2844 max 2844  cwnd 100  thresh 100 recover 0
packrxmit pkt 2747 nxt 2844 max 2845 cwnd 100  thresh 50 recover 2844 una 2744
packrxmit pkt 2750 nxt 2845 max 2845 cwnd 98  thresh 50 recover 2844 una 2747
badack 6636 snd_max 6808 snd_nxt 6808 snd_una 6661
badack 6892 snd_max 7048 snd_nxt 7048 snd_una 6899
```

ACKs, and a retransmit resulting from a timeout. The *packrxmit* are NewReno (or SACK) retransmits during "recovery". A *badack* is an out of order ACK or duplicated ACK.

With a debug level of 4 or higher, a trace file is produced for each transmit and ack packet, data includes time stamp, segment number, round-trip time, congestion window, and threshold. Using *probeall.pl*, from the trace file, one can plot tcptrace-like information (RTT over time, cwnd over time, average and instantaneous bandwidth, ack-sequence number, etc). Writing the trace file can slow performance for high speed interfaces. A few lines from a trace file follow.

```
0.408133 258 xmt
0.408196 234  0.038062  26 26 ack
0.408259 259 xmt
0.408346 235  0.038069  26 26 ack
```

The following config file could be used to enable FACK/SACK including rampdown [22]. The server should be started with *atousrv -s* in order to provide SACK information to the transmitter. This config file will also cause packet 200 to be dropped twice (causing a timeout event).

```
newreno 0
fack 1
sack 1
```

```
rampdown 1
rcvrwin 26
maxtime 10
droplist 200 200
debug 4
```

The following config file provides a *10x* virtual MSS by starting slow-start with 10 segments, and incrementing by 10 segments per RTT during congestion avoidance. We drop a packet in order to insure our test will illustrate a recovery event.

```
newreno 1
rcvrwin 1000
maxtime 10
initsegs 10
increment 10
droplist 1400
debug 3
```

The following config file alters the congestion avoidance parameters to provide a more aggressive recovery for a high delay, bandwidth path. We drop a packet in order to insure our test will illustrate a recovery event.

```
newreno 1
rcvrwin 1000
maxtime 10
multiplier 0.9
increment 10
droplist 1400
debug 3
```

See *http://www.csm.ornl.gov/ dunigan/netperf/atou.html* for more examples and results.