between router 2 and 3 was set to 4Mbps in both directions. There were three terminals attached to the asymmetric network at router 1 and simultaneously downloaded files from the server. The results show that ACE (with a goodput of 320Kbps) outperforms TCP Reno (with a goodput of 110Kbps) by about 200%. These results indicate that when multiple users share an upstream channel (like cable network), ACE performs much better than TCP Reno.

We also did some testing where a terminal makes both upstream and downstream transfers, we found that the downstream throughput using the ACE TCP stack is slightly lower than Reno. To explain this observation, consider the scenario in ACE where an ACK that acknowledges four downstream data packets queues up behind 3 large data packets. In Reno this could have been an ACK of 2 packets queuing up behind one data packet, and another ACK of 2 packets queuing up behind another two data packets that follow the first ACK. In Reno's case, a sender would have received some packet acknowledgment significantly earlier than in ACE's case. One must carefully distinguish this scenario from the one on a cable network where upstream bandwidth is allocated to terminals in a round robin fashion. In that case, each ACK in front implies one more round of waiting. Hence it pays to reduce the number of ACKs.

For ACE to work well in scenarios where upstream data transmission is often, we might need separate buffering of data packets and ACKs [11], and give ACKs a higher priority. The upcoming DOCSIS standard supports service classes of different QOS, thus acknowledgments can be put on a more stringent service class. Given that most of the asymmetric network subscribers are usually downloading from the Internet rather than uploading, we feel that ACE is a good solution to improve downstream throughput.

All of our simulation and implementation experiments focused on the ftp application. Because of the bulk transfer nature of ftp, a TCP session is likely to achieve a larger *cwnd*, thereby making it important and worthwhile to reduce the number of ACKs per *cwnd*. For short TCP connections, the transfer will slow down significantly if we reduce the number of ACKs per cwnd. In fact, both ACE and ECWA do not reduce the number of ACKs when cwnd is small. The effectiveness of ACE and ECWA in web browsing applications would be very dependent on how a web browser uses TCP connections. In some of the older browser implementations, a browser opens one TCP connection per object. Such connections tend to be short, and hence the effectiveness of ACE and ECWA in reducing upstream ACKs is very limited. In HTTP 1.1, browsers can reuse a TCP connection, i.e., a browser maintains persistent TCP connections. Furthermore, it is possible for a browser to have multiple outstanding object GET requests sharing one TCP connection. Thus a persistent TCP connection resembles a bulk transfer ftp connection a little more.

6. SUMMARY

In this paper, we proposed ACE (Acknowledgment Based on *Cwnd* Estimation), which is an approach to speed up the TCP transfer over an asymmetric network. The idea is based on previous observations on varying the number of packets acknowledged by an ACK according to a sender's *cwnd*. When *cwnd* is small, the number of packets per acknowledgment is small and this aids in speeding up initial transfer and building up the *cwnd*. When *cwnd* is larger, the number of packets per acknowledgment is larger, this reduces the number of ACKs sent on the narrow bandwidth link without much impact to the sender. Our proposal is different from previous works in that we estimate the *cwnd* based on measurement, i.e., by measuring the number of packets arriving within a receiver measured round trip time. In addition, an ACE TCP receiver detects the possibilities of retransmission timeout, fast retransmission, sender's temporary lack of data, and adjusts the value of *ppa*. ACE has an outstanding deployment advantage over some of the previous works in that ACE does not require special network support, nor does it require changes in sender's TCP stack, nor introducing a new TCP option. Being able to improve performance by changing only one side of the TCP stack has an enormous advantage as it means that only those terminals that are attached to an asymmetric network need to have an ACE patch, the rest of the servers or terminals out there do not need to be modified. Both our simulation and implementation show that ACE improves the TCP throughput over asymmetric networks very significantly.

7. REFERENCES

[1] Mark Allman, et,al."Ongong TCP Research related to Satellites", Internet draft under tcpsat working group of IETF.

[2] Mark Allman, Aaron Falk " On the Effective evaluation of TCP", SIGCOMM Computer Communication Review, Volume 29, Number 5 (October 99).

[3] Hari Balakrishnan, Venkata N. Padmanabhan, and Randy H. Katz, "The Effects of Asymmetry on TCP Performance", ACM MOBICOM, September 1997.

[4] A. Calveras, J. Linare, J. Paradells, "Window Prediction Mechanism For Improving TCP in Wireless Asymmetric Links", Globcom 98.

[5] Reuven Cohen, Srinvas Ramanathan "TCP for High Performance in Hybrid Fiber Coaxial Broad-Band Access Networks", IEEE Transaction on Networking, Vol., 6 No. 1, February 1998

[6] http://lrcwww.epfl.ch/linux-diffserv/

[7] V. Jacobson, "Compressing TCP/IP Headers for Low-Speed Serial Links", RFC 1144, February 1990.

[8] V. Jacobson, "Congestion Avoidance and Control", ACM SIGCOMM 88, August 1988.

[9] T.V. Lakshman, U. Madhow and Bernhard Suteret, "Window-based error recovery and flow control with a slow acknowledgment channel: a study of TCP/IP performance", INFOCOM'97, April 1997.

[10] Vern Paxson and Sally Floyd, "Wide-Area Traffic: The Failure of Poisson Modeling", IEEE/ACM Transactions on Networking, June 1995.

[11] V. Paxon et, al. "Known TCP Implementation Problems", Network Working Group, Internet Draft.

[12] RFC 1323 TCP Extensions for High Performance

[13] RFC 2581 TCP Congestion Control.

[14] Keshav Srinivan, "Method and A*ppa*ratus for collapsing TCP ACKs on asymmetric connections",U.S Patent 5,793,768 .

5. IMPLEMENTATION

We have implemented the ACE algorithm on a testbed consisting of Pentium based PCs running Linux V2.2.6. The testbed shown in Figure 8 consists of three routers between a FTP server and a terminal. By using the Diffserv patch [6], we can set a packet flow as the first class traffic at a router, and assign a certain bandwidth for the flow. In this way, we can create network asymmetry by assigning different values of bandwidth to each direction. We tested our Linux modification using FTP by having the terminal download files over the asymmetric network.

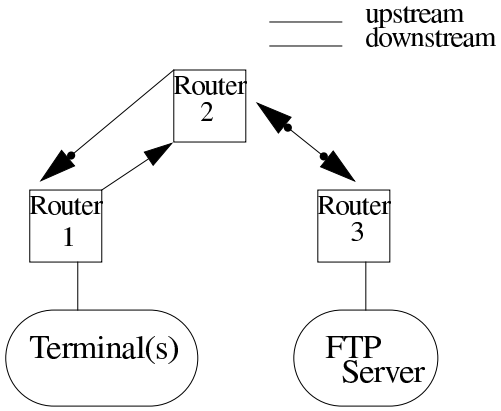

FIGURE 8. Topology of our testbed

5.1 Modifying TCP on Linux

In Linux V2.2.x, the timestamp option is enabled by default. We explored this feature and used method (4) described in section 3.2 to measure the *rcv_RTT* and *est_cwnd*. In this version of Linux, the value of ATO is equal to 1.5 times of the average measured inter-packet gap. The inter-packet gap is the time interval between arrival of data packets.

A receiver repeats the *est_cwnd* measurement process one after another. If a sender does not have more data to send, the receiver will set the *est_cwnd* and *ppa* to 1 after 1.2 times of the current *rcv_RTT*, and the current *rcv_RTT* measurement process will be aborted[1]. The *est_cwnd* measurement process is restarted only when a new data packet is received. In addition, we set the ATO to be *ppa* times the average measured inter-packet gap, thus delaying the acknowledgment according to the number of packets that we need to wait for. All through our implementation, we only changed the receiver part of the TCP stack at the terminal.

5.2 Implementation Performance

We performed several experiments with the ACE and TCP Reno to compare the performance. The FIFO buffer size in all tests was 20 packets for each direction of the traffic.

[1]. Note that this does not preclude a receiver to adjust its *RTT* measurement to true increase in *RTT*, so long as there are packets coming in within each 1.2 *RTT* interval.

1) Performance under different extent of upstream congestion

In this scenario, 1 user downloaded files using FTP. The bandwidth for router 2 to router 1 was set to 4Mbps, and the bandwidth from router 1 to router 2 was varied from 100Kbps to 10Kbps. The bandwidth between router 2 and 3 was set to 4Mbps in both directions. With 1523-byte data packets and 73-byte ACKs (10 bytes for the timestamps), the normalized bandwidth ratio k ranged from less than 2 to 15. Based on our observations in section 1.3, we expected that a TCP downstream transmission to range from fully utilizing the downstream bandwidth to only using a fraction of it. Note that *k* is similar to the 'system capacity', it is the capacity of the upstream channel to transmit ACKs, i.e., the channel can transmit one ACK per *k* data packets coming down. On the other hand, *ppa* is similar to the offered load, and *max_packets_per_ack* is like the maximum offered load which happens when *cwnd* is large. Because the arrival process of ACKs is not constant, queueing at a terminal's transmission buffer occurs. Thus downstream throughput tends to be affected even when *k* is still a little larger than *ppa*.

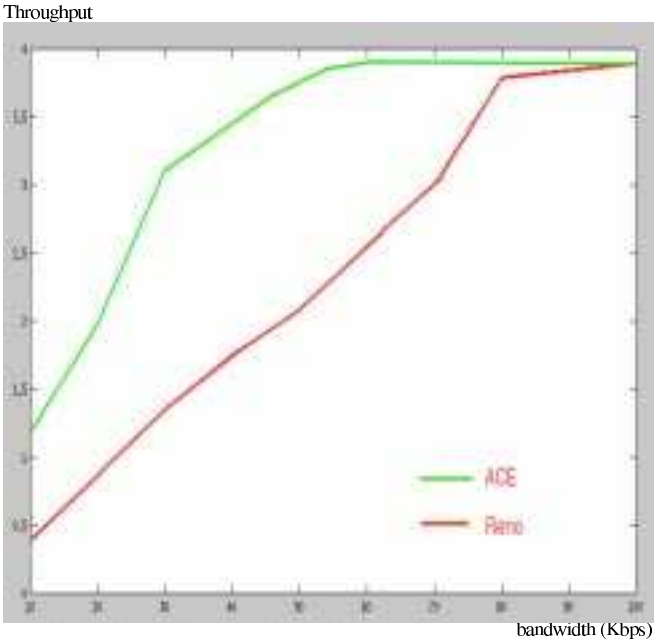

FIGURE 9: Average FTP goodput (Mbps) vs. upstream bandwidth (Kbps)

Figure 9 shows when upstream bandwidth is about 100Kbps, the performance of TCP Reno and ACE are about the same. When the upstream bandwidth is 100 kbps, k is 1.9. The throughput of TCP Reno starts to decrease when the upstream bandwidth is less than 100kbps, and the advantage of ACE becomes more and more significant. In fact, ACE maintains a performance of nearly 4Mbps until the upstream bandwidth falls below 50Kbps, corresponding to k=3.9. After that, the downstream throughput of ACE starts to decrease gradually. However, it still outperforms Reno consistently. This is in line with a *max_packets_per_ack* of 5. When upstream bandwidth is 10Kbps, ACE has about 200% improvement over TCP Reno.

2) Performance when different users/applications share the upstream link

The bandwidth for router 2 to router 1 was set to 4Mbps, and the bandwidth from router 1 to router 2 was 80Kbps. The bandwidth

mation, we have also simulated estimation of *RTT* using the initial SYN/SYNACK pair. This method is simple, it does not require extra packets nor support of the TCP timestamp option.

We simulated the SYN approach a number of times, and find that in our simulation, the *rcv_RTT* tended to be about 30% smaller than the actual *RTT*. It led to an 'underestimate' of *cwnd*, i.e., *est_cwnd* tended to be smaller. As a result, the receiver sent acknowledgment a bit more often than necessary. This in turn renders the ACE approach less effective. The average throughput in scenario 1 using the SYN approach was about 248 kbps vs. 294 kbps when the ICMP approach was used.

In general, *RTT* fluctuates according to network conditions, hence, the SYN measured *RTT* will sometimes be smaller and sometimes be larger than the current RTT in a TCP session. If the SYN measured *RTT* is significantly larger than the current *RTT*, a receiver may over estimate cwnd and over reduce the number of ACKs. This can in turn slow down the sending process.

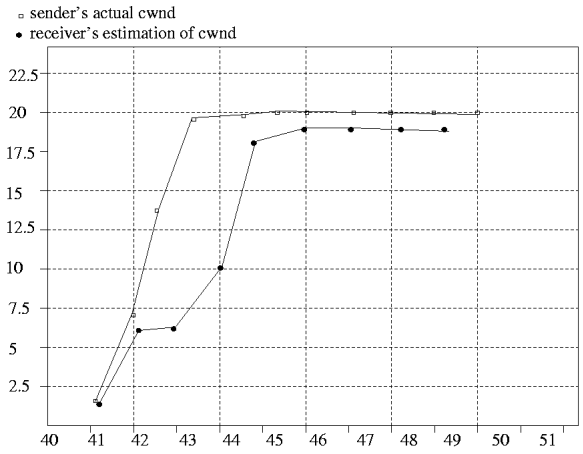

FIGURE 5: Estimated *cwnd* and the actual *cwnd*.

4.4 Burstiness Due to ACE

A potential problem with ECWA or ACE is that when the value of *ppa* is high, the sender may send out bursts of packets. This effect is known as the "stretched ack violation" [10]. We propose a solution where *max_packets_per_ack* (c.f. section 3.2) is varied according to the number of congestion incidences that a receiver has observed. In general, a receiver reduces the *max_packets_per_ack* when it sees congestions occurring frequently, and increases it if it has not seen congestion for a while. As a result, a sender receives ACKs on a smaller number of packets when there is congestion in the forward path, and this will help to reduce the burstiness of the traffic. A receiver can judge whether a congestion has occurred by using the approach described in section 3.2.(4). In essence, ACE can reduce the number of ACKs when there is no congestion and *cwnd* is big. When there is congestion ACE can reduce *ppa* to perform like Reno.

Figure 6 shows the performance of ACE when the *max_packets_per_ack* is varied from 2 to 5 in scenario 1. Not surprisingly, ACE with a *max_packets_per_ack* of 2 performs similarly to TCP Reno.

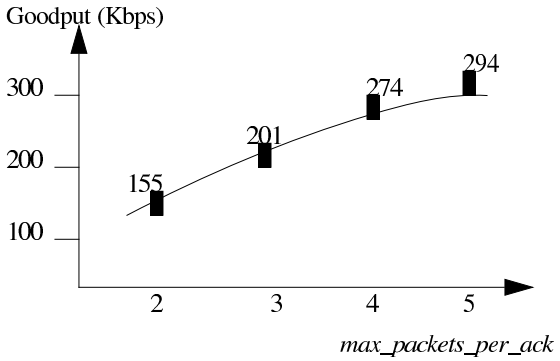

FIGURE 6: Performance of our algorithm as *max_packets_per_ack* varies from 2 to 5

4.5 Performance under TCP Cross Traffic

In the previous section, we have been using UDP as background traffic, as such the background traffic does not scale back even when there is a congestion. It is interesting to see what will happen if TCP is the major type of traffic. We did some experiments where all 19 users in the cable network used the same version of TCP. We repeated the experiments with Reno, ECWA, and ACE. We feel that this is important as the behavior of a protocol under mass adoption must be studied.

From Figure 7, we can see that the advantage of ECWA and ACE is well maintained. The throughput is in general higher since the same number of users sending UDP upstream create more overload on the upstream channel than the same number of users making TCP transfers downstream.

The key observation in this section is that both ECWA and ACE improve the performance of TCP over asymmetric network by more than 100%. Even in the case when both sides of an intermediate link between an asymmetric network and the TCP sender is congested, the two schemes still perform better. The ACE scheme lags the ECWA scheme by about 10%. This is because ACE relies on counting incoming data packets to estimate *cwnd*, whereas in ECWA a sender passes *cwnd* explicitly.

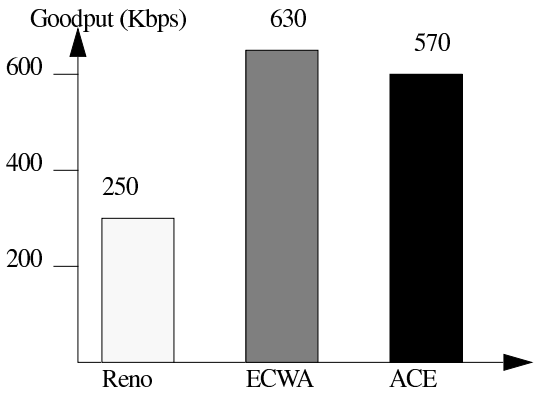

FIGURE 7: Performance of various TCP versions when all users used the same version

we simulated TCP Reno, and a variation where an ACK was delayed for 4 packets, or until ATO (Acknowledgment Timeout) occurred. We call this version FDA(4), i.e., fixed delay acknowledgment for 4 packets. The simulated ACE TCP on the cable user's computer used method (3) described in section 3.2 to measure round trip time and *est_cwnd*, i.e., by sending an ICMP echo-request packet every other *rcv_RTT*. The results for scenario 1 are shown on the following bar chart.

ECWA, FDA(4) and ACE all perform better than Reno in scenario 1. When a congestion occurs at an upstream cable channel, a queue of ACKs builds up in the cable modem. Those ACKs in the back have to wait for the ACKs in the front to be transmitted. Furthermore, after an ACK is transmitted, the upstream bandwidth allocation algorithm in the cable router will grant transmission opportunities to other modems, and will not come back until one 'round' later. Thus, by the time when an ACK at the end of the queue gets to be transmitted, significant delay would have occurred. By sending less ACKs the transmission queue will be shorter, and the number of 'rounds' that an ACK needs to wait will be less. As one would expect, ECWA has the best performance, however, the ACE comes very close in both scenarios 1 and 2. FDA(4) performs better than TCP Reno during upstream congestion, i.e., scenario 1. The problem comes when there is a downstream congestion. The reason is that a FDA(4) receiver waited for four packets or until ATO occurred before it sent an ACK. When the sender's *cwnd* was small, a receiver delayed an ACK until ATO occurred. This slowed down the transmission initially, and slowed down any subsequent recovery if retransmission timeout occurred. In scenario 1, the initial slow down causes FDA(4)'s performance to lag behind ECWA and ACE. In scenario 2, congestion occurred often and hence FDA(4)'s performance suffered more.

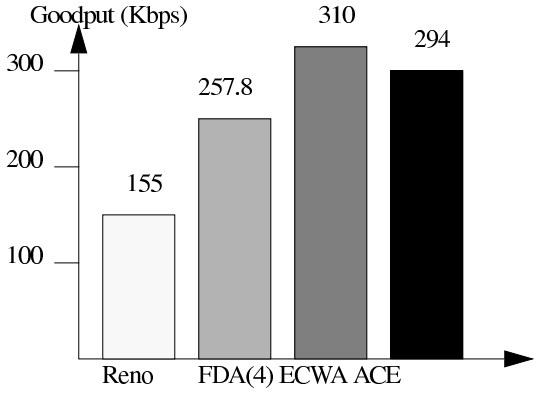

FIGURE 3: Performance in scenario 1 (upstream congestions)

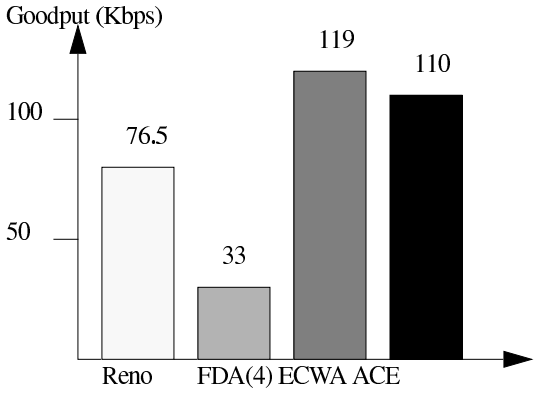

FIGURE 4: Performance in scenario 2 (two way congestions)

Because both ECWA and ACE algorithms vary the *ppa* value according to *cwnd*, when there was downstream congestion as in scenario 2 and *cwnd* was small, both algorithms acknowledged every packet, helping the window size to be built up again. Furthermore, both algorithms avoided having a receiver delaying an ACK until ATO when cwnd was 1, e.g., right after recovery from a timeout. In scenario 1 where there was adequate downstream bandwidth and *cwnd* was large, both algorithms acknowledged less frequently.

Figure 5 compares *est_cwnd* with the actual *cwnd*. There are a few reasons for the fluctuation of measurements:

(1) the measurement of *est_cwnd* is dependent on *rcv_RTT* measured by ICMP packets, which may fluctuate.

(2) because of various random elements in the system, such as transmission queuing delay, and sender's load etc., there is a certain randomness in the arrival process of data packet at the cable modem user. Hence, the number of packet arrivals during a fixed time internal can vary even if *cwnd* at the sender is constant. Overall, our discussion in section 3.1 holds, i.e., the value of *est_cwnd* lags behind the actual *cwnd*, and catches up when the sender enters congestion avoidance phase.

4.2 Acknowledgment filtering (AF)

Acknowledgment filtering (AF) mentioned in section 3.2 belongs to a different class of solutions, it is implemented at a transmission device directly connected to an asymmetric network, e.g., a cable modem. It is independent of the *cwnd* size at senders. It is efficient in the sense that no ACK is transmitted 'unnecessarily'. A new ACK always takes the place of an older ACK in the transmission queue. We find that AF is a very effective approach. In scenario 1, AF yields a throughput of 550 kbps, significantly faster than the class of *cwnd* estimation algorithms. The aggressive ACK replacement scheme in AF means that at most only one ACK is waiting for transmission in a cable modem. Thus an ACK will never have to wait for an older ACK to be transmitted first. In a sense, AF delivers the newest ACKs as fast as possible. However, AF must be implemented in cable or ADSL modems, making it less readily deployable.

4.3 RTT Estimation

In section 3.2 we described four ways for a receiver to estimate *RTT* so that it can in turn count the number of packets arrived and compute *est_cwnd*. In addition to simulating ICMP based *RTT* esti-

when the sender does not have data to send for more than one *RTT*. By setting the *est_cwnd* to 1, *ppa* will also be set to 1. When the retransmitted packet arrives, a receiver will acknowledge it immediately. The case for fast retransmission is a bit more complicated. A receiver is able to detect fast retransmission because it must have sent out at least three duplicated ACKs to the sender in the first place. However, having sent out three duplicated ACKs does not necessarily guarantee that the sender will fast retransmit rather than timeout. This is because the sender may have timed out before it receives all three duplicated acks. In addition, some of the duplicated acks may get lost on their way, and hence the sender will not be able to receive enough duplicated acks. Because of these uncertainties, we propose that a receiver changes the *est_cwnd* to 1 after sending out three duplicated ACKs.

It should be noted that a sender's *RTT* can be different from a receiver's *rcv_RTT*. Hence a receiver can guess wrong. For example, it is possible that the sender's *cwnd* has not been set to 1, when a receiver has waited for 1.2 *rcv_RTT* and set *est_cwnd* and *ppa* to 1. When data packets arrive, a receiver may end up acknowledging more frequently than necessary. However, the receiver will recover its value of *est_cwnd* one round trip later. On the other hand, if a receiver missed the timeout at the sender, it will acknowledge based on a potentially larger *ppa* and may delay the ACKs. Hence the sender will take longer to build up *cwnd*, and throughput will be reduced. Nevertheless the sender will eventually build up *cwnd* to the size of *est_cwnd* again. Based on this analysis, we favor a more conservative approach at the receiver, where *est_cwnd* should be set to 1 before the sender does so with *cwnd*.

4. SIMULATION

[2] gave many suggestions on how to effectively evaluate TCP. We evaluated our proposal using both simulation, and a modified TCP stack over the Linux platform on an emulated asymmetric network.

We used the network simulator OpNet Modeler to test the performance of our algorithm and some of the others in the literature. We built our simulation based on a cable TV network used to carry data. Over a cable network, upstream refers to the direction from a cable modem to a cable router, and downstream refers to the direction from a cable router to a cable modem. Cable networks are almost always asymmetric. In our simulations, the bandwidth of a downstream channel is 27Mbps, while that of an upstream channel is 768kbps. Figure 2 shows a typical cable network. Both upstream and downstream channels are shared by home users. Data for home users is relayed from the Internet via the cable router and broadcast to the cable modems. A cable modem acts as a bridge and passes on only those data packets destined for its home user. Upstream transmission is a bit more complicated because multiple cable modems may transmit at the same time. Hence, a multi-access scheme is needed. Current cable standard DOCSIS provides a number of mechanisms to control upstream access. For the purpose of evaluating ACE and other TCP algorithms, we use a mechanism described below. In our model, a cable router polls each of the cable modems that has recently transmitted up stream to give them opportunities for up stream transmission. A cable modem is allowed to transmit one frame each time it is polled. For those cable modems that have not been transmitting in the last 5secs, if they want to transmit up stream, they will have to wait for a `contention slot'. The cable router grants a contention slot periodically. After transmitting using a contention slot, a modem must wait for a confirmation from the router. If there is a collision it will have to go through another round of contention. If a cable modem cannot successfully send out a frame in 10 rounds, the modem will discard the frame.

4.1 Upstream and Downstream Congestion

We first experiment with two scenarios, namely, upstream congestion and downstream congestion.

Scenario 1 - Upstream congestion: Upstream congestion is quite common during the busy hours in cable networks. To simulate upstream congestion, we simulated 19 users generating upstream cross traffic from cable modems, each user generated 1460 bytes UDP packets at a mean rate of 180kbps. Upstream UDP traffic from each user followed a self-similar pattern [10]. Because there is no congestion control mechanism in UDP, the cross traffic did not back off even when severe congestion occurs. We then simulated a cable modem user who made ftp download from the Internet. The size of the file downloaded by the simulated user was 300Kbytes. The TCP segment size was 1460 bytes corresponding to the maximum segment size over the cable network. The receiver buffer is capable of holding 20 TCP packets with maximum segment size.

Scenario 2 - Internet congestion: Although down stream congestion may not occur at a cable network's downstream channel, it may occur in other parts of the Internet. It is important to test any TCP modification in this environment as well. We restricted the packet processing capability of the router in between the ftp server and the cable router to 100 packets per second, creating both upstream and downstream congestions simultaneously. We simulated self-similar traffic on this link so that on average the router is half loaded with cross traffic. A cable modem user downloaded a file with an average size of 100k bytes from the remote FTP server. In this case, there was no other cross traffic on the cable network itself.

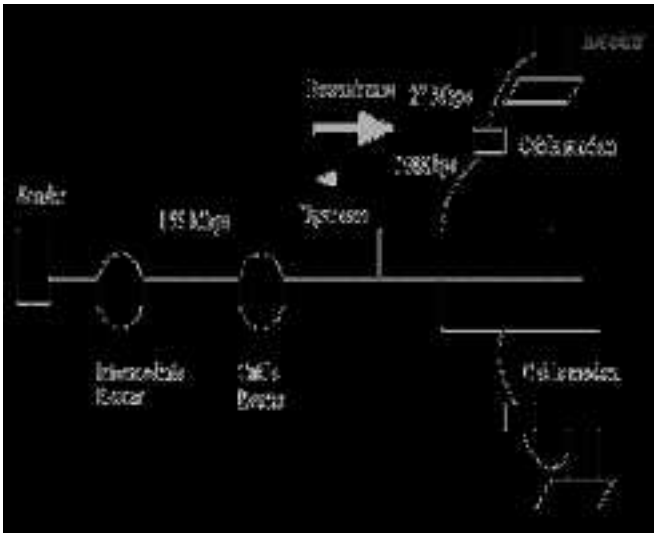

FIGURE 2: Simulated cable environments.

We would like to compare ACE with others that belong to the class of algorithms based on cwnd estimation. Algorithms belonging to this class do not require lower level network support. In particular we chose ECWA. In ECWA, *cwnd* is explicitly announced. In our algorithm *cwnd* is estimated based on measurement. Thus ECWA should represent a performance upper bound to the class of algorithms that varies acknowledgment according to *cwnd*. In addition,

time only at the beginning. Since round trip time fluctuates over the life time of a connection, using one or two measurements obtained at the beginning of the connection is probably not adequate. Method (3) requires sending out ICMP packets but the overhead is justifiable. Almost all of the hosts response to echo-requests, hence method (3) can be quite readily used. Method (4) requires both sides to support the TCP timestamp option, while there are older implementations that may not support this feature, the timestamp option will probably be a norm in the near future. There can be times when a sender does not have data to send and hence a receiver will not be able get its timestamp acknowledged. We address this problem in (4) below. Another possibility is to use method (1) but back up with method (3) when the terminal has not been sending data for a while.

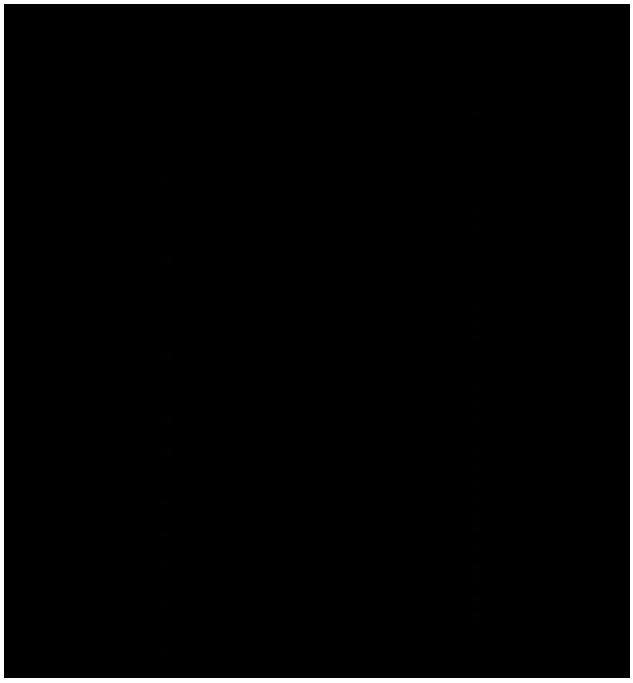

FIGURE 1: cwnd estimation process at a TCP receiving end

To count the number of data packets arriving within a round trip time, we can estimate the round trip time using the methods above, and then count the number of data packet arrivals within the period. Alternatively, we can count the packets while we are measuring *rcv_RTT*. Both method (3), and (4) support this approach. We start counting right after we send out an ICMP packet, or after we put a certain timestamp value into the receiver timestamp as part of an acknowledgment. We stop counting right after we receive the ICMP echo-reply packet or when we receive a data packet carrying the same receiver timestamp. We use the count as *est_cwnd*. The frequency of round trip time measurement is an implementation choice, but is a trade-off between accuracy and overhead. For example, the frequency can be per two to three round trip times. Using the current value of *rcv_RTT*, the measurement of *est_cwnd* can be done one round trip time after another.

2) Adjustment of est_cwnd

Using one of the above methods, a receiver obtains a value of *est_cwnd*. However, due to fluctuations of round trip time and some degree of randomness in the packet arrival process, it is possible that a receiver may find that the new value of *est_cwnd* is smaller than the old value. In this case, we use the old value of *est_cwnd* instead. The rationale behind using the old value is that 1) unless retransmission timeout or fast recovery occurs at the sender, a *cwnd* is never decreased; 2) our analysis in figure 1 shows that *est_cwnd* tends to lag behind the actual *cwnd*, therefore if the new *est_cwnd* is smaller, it is probably safe to use the old value.

3) Calculating the value of ppa

Based on the adjusted *est_cwnd*, the value of *ppa* is calculated as follows:

if (est_cwnd <= min_ack_per_win)
then ppa = 1
Otherwise
ppa=MIN(FLOOR(est_cwnd/min_ack_per_win), max_packets_per_ack)

The quantity *min_ack_per_win* refers to the minimum number of ACKs that must be sent out per window full of packets. This value cannot be too small as losing all the ACKs implies a retransmission timeout at the sender end. We pick the value of *min_ack_per_win* to be 3. Based on the calculation above, delayed acknowledgment will not occur until *est_cwnd* is at least 6. This should encourage fast growth of *cwnd* at the beginning [1]. Notice that if we were to use a ROUND function rather than a FLOOR function, delayed acknowledgment will happen when *est_cwnd* is 5. We consider this an implementation issue. Regardless of the value of *ppa*, a receiver always acknowledges out-of-sequence packets immediately.

The value of *max_packets_per_ack* is to control the number of packets that a receiver can wait at most before sending out an acknowledgment. It is applicable in those cases when both sender and receiver have larger buffer size, giving the possibility of a large *cwnd*. It is important to limit maximum *ppa* for two reasons. Firstly, a large *ppa* would mean that a receiver is holding up a number of transmission opportunities, this in turn can reduce the throughput. Secondly, if *ppa* is large, each ACK will acknowledge a large number of packets. This may cause the sender to send out bursts of packets and cause packet drops in the network. We chose 5 as the value of *max_packets_per_ack* in our experimentation.

4) Anticipating Scaling Back of cwnd

Congestion on the forward path may lead to packet drop. In general, if *cwnd* is at least 4 and a packet is dropped, the 3 duplicated ACKs sent by a receiver would trigger a fast retransmit. On the other hand, if *cwnd* is small or if there is further packet loss, then a sender's retransmission timeout will occur. In the first case *cwnd* is reduced to a half of its current value, and enters the congestion avoidance phase after fast recovery. In the second case, *cwnd* is reduced to 1, and slow start begins after the lost packet is acknowledged. In addition, when a TCP sender does not have data to send for more than one round trip time, the recommended procedure is for the sender to reduce *cwnd* to 1 so as to avoid flooding the network with a sudden burst of packets when there is data to send. Because of the adjustment described in (3) where old *est_cwnd* is taken if new *est_cwnd* value is smaller, it is necessary to detect the three scenarios and make conscious corrections, otherwise the *est_cwnd* will not be decreased. This can in turn lead to unwanted delay of ACKs.

To address the scenarios above, a receiver reduces the *est_cwnd* to 1 whenever it fails to receive any new packets after *rcv_RTO*, which is an approximation of the sender's retransmission timeout. For simplicity, the value of *rcv_RTO* in our simulations is set to 1.2**rcv_RTT*. This will address the scenarios when *cwnd* is small and the sender is forced into slow start when timeout occurs, and

downloading a file onto his PC, or browsing the web. For those applications that have traffic on both directions, e.g., telnet, response time rather than throughput is often more important.

In this section, we looked at various approaches to improve TCP performance over asymmetric networks. For those that are based on *cwnd* estimation, they either require a sender to explicitly send out *cwnd* as a TCP option, or attempt to estimate *cwnd* by running congestion control algorithm at the receiver. The first method requires modifications to TCP at terminals and on every computer that may make TCP transfer to these terminals. While it is possible for a user to download a TCP patch for his terminal, it is hard to require other computers out there to make the modifications. The second method suffers from recurring errors in estimation of *cwnd*. In the AF scheme older ACKs in a transmission queue are replaced with an incoming ACK at the point of transmission to an asymmetric link. However, this requires modifications of transmission devices such as cable or ADSL modems. Thus vendor support is necessary and upgrading existing deployment may be difficult. Finally, header compression does not reduce the number of ACKs. For some access networks like cable networks or wireless networks, it is important to reduce the amount of upstream contention regardless of the packet size. In the next section we will describe ACE (Acknowledgment based on Cwnd Estimation), where only the TCP receiving algorithm needs to be changed. This implies that only terminals attached to an asymmetric network need to have their TCP stacks modified, making the solution very readily deployable.

3. OUR PROPOSAL

3.1 The Basic Idea

We approach the performance problem of TCP over asymmetric networks by asking ourselves whether it is possible to measure a sender's *cwnd* from a receiver. Our hypothesis is that as long as *RTT* is significantly larger than the amount of time needed to send out a window of packets, a TCP sender with data to send is likely to send close to a window worth of packets in a round trip duration. This in turn implies that a receiver can estimate *cwnd* by counting the number of packets received during a round trip time. We call this value *est_cwnd*. Based on *est_cwnd*, the receiver can now determine *ppa* (the number of packets per acknowledgment). Because our algorithm is based on *cwnd* estimation, we call it ACE (Acknowledgment based on *Cwnd* Estimation). Figure 1 shows the process of *cwnd* measurement at the receiver. In this example, a receiver acknowledges every other packet, and a sender increments its *cwnd* by 1 for every acknowledgment received. For the sake of illustration, we assume that the round trip time is constant and the receiver knows the round trip time. We will address round trip time estimation and fluctuation in later sections. From the diagram, it can be seen that because a TCP receiver can only make an estimate after it has counted incoming packets for one round trip, its estimation of *cwnd* size lags behind the actual *cwnd* size during the slow start phase. However, once the sender *cwnd* reaches the congestion avoidance phase, the receiver's *cwnd* measurement catches up with the actual *cwnd*. Our simulation results shown in figure 5 confirms the above. There is in fact a certain unexpected advantage in underestimating the *cwnd* during the slow start phase, as it will drive the receiver to acknowledge more often and thereby helping the sender to build *cwnd* faster. This example illustrates that counting the number of packets received per round trip time at the receiver gives a reasonable estimate of *cwnd*. For the case where the round trip time is less than the total transmission time of a full *cwnd* of packet, bandwidth limitation on the return path has less effect, since the sender can probably keep the transmission pipe full most of the time. Being able to estimate a sender's *cwnd* from a receiver without the sender's collaboration gives our proposal a deployment advantage, i.e., only TCP software at terminals needs to be patched.

In the next section, we describe our algorithm in more detail. We will discuss the various options to measure round trip time and to estimate sender's *cwnd* from a receiver. Then we will discuss the computation of *ppa* from est_*cwnd*. In [3], *cwnd* is announced explicitly by a sender, a receiver will be informed of *cwnd* change due to retransmission timeout or fast retransmission. In our case, our receiver `anticipates' such changes. We discuss this issue in the next section as well.

3.2 A Step by Step Description

1) Estimating Cwnd

To estimate *cwnd*, a receiver needs to count the number of packets received within a *RTT*. We call the receiver's estimation of *RTT* *rcv_RTT*. There are four ways in which *rcv_RTT* can be estimated:

(1) If a receiver is also sending data packets, then a round trip time estimation will be available from the sending part of TCP.

(2) It is possible to gauge *rcv_RTT* at the beginning of a TCP connection by timing SYN, and SYNACK packets. This will give a crude estimate of the round trip time.

(3) A receiver can send an ICMP echo-request with a timestamp to solicit an echo-reply from the sender. The value of *rcv_RTT* is measured as the time interval between sending of a request and receiving of a reply.

(4) By using the TCP timestamp option that was originally used by a TCP sender to estimate round trip time. A sender places a timestamp in each data segment, and the receiver copies the timestamp back in an ACK. Then a single subtract gives the sender an accurate *RTT* measurement (more details in [12]). In fact, timestamps are always sent and echoed in both directions. Even during a one-way transfer, a receiving side who has no data to send inserts its current timestamp in each ACK, and the sender echoes back in each data packet. It is quite straight forward to modify the receiving part of TCP stack to compute *rcv_RTT* based on the timestamps

.

Method (3) implies additional packets, i.e., ICMP echo-request, have to be sent on the return path. However, it can be shown that the savings justify the additional ICMP packets. For a maximum *cwnd* of 12, by running ACE, assuming that min_ack_per_win (minimum number of ACKs that must be sent out per window of packets, c.f. section 2.1) is 3, the computed *ppa* would be 4. Three ACKs will be generated for each *cwnd* of packets. If we send an ICMP echo-request packet every other *rcv_RTT*, then we will be sending a total of 7 packets on the return path in a period of two *rcv_RTTs*. On the other hand, a current implementation of TCP with delayed acknowledgment would have sent 6 ACKs for the first round trip, and another 6 for the second one, generating a total of 12 packets on the return path over a period of two *RTTs*. Using the ICMP approach there is still a saving of 5 upstream packets, i.e., a 42% saving.

Method (1) depends on application behavior, if an application does not always have data to send, method (1) may not be applicable. Furthermore, because of bandwidth asymmetry, *RTT* observed by a sender sending upstream may be larger than *RTT* observed by a sender sending downstream. We are interested in the *RTT* of a sender sending downstream. Method (2) measures the round trip

[5] examined how TCP parameters and features can have impact on throughput over cable networks. These include delayed acknowledgment, and socket buffer size. In particular, it was pointed out that due to the limited amount of buffer in a cable modem, increasing the socket buffer size continuously may cause a TCP sender at the internet side to overrun the cable modem, resulting in packet loss. In the rest of this section, we will discuss the major approaches.

2.1 Congestion Window Dependent Acknowledgment

An approach to address upstream bandwidth limitation would be to reduce the number of acknowledgments. This can be done by sending an ACK for every few packets received. However, because a sender relies on receiving ACKs to send out more packets and to increase *cwnd*, delaying an ACK to wait for a few packets may reduce performance, especially when *cwnd* is small. [5] gave a good detailed account on how delayed acknowledgment could slow down web page transfer. A general strategy is to acknowledge more frequently when *cwnd* is small, and acknowledge less frequently when *cwnd* is larger. [1] examined a few scenarios where a receiver acknowledges every packet during a sender's slow start phase, and adapts delayed acknowledgment when the sender goes into congestion avoidance phase. This will help the sender to keep sending and increasing its *cwnd* when *cwnd* is small, and reduce the use of upstream bandwidth when *cwnd* is larger. The mechanism for a receiver to find out a sender's congestion control phase was an open issue.

[3] proposed a scheme where a sender sends back the value of *cwnd* to a receiver as a new TCP option. A receiver computes the number of packets per acknowledgment, which we will call *ppa*, by dividing *cwnd* with *min_acks_per_win* (the minimum number of ACKs to be sent per window of packet received). This approach requires modifications of TCP software on both sender and receiver. In real life it implies that terminals and any computer that transfer data to those terminals must have their TCP stacks modified. For the ease of reference, we will call this scheme the ECWA (Explicit Congestion Window Acknowledgment).

[4] proposed a window prediction mechanism for improving TCP over wireless asymmetric links. The approach taken there is for a receiver to emulate a sender's *cwnd* growth algorithm in order to predict the current *cwnd* at the sender. Based on the congestion window size, the value of *ppa* is determined in a similar fashion to [3]. However, it can be shown that under a number of circumstances, a receiver may loose track of *cwnd*. For example, a receiver would increment its *cwnd* value by one for every ACK sent during the slow start phase. If the ACK is dropped, the sender will miss the increment. The event can go unnoticeable at the sender if subsequent ACKs are received. Moreover, new TCP implementations are supposed to set *cwnd* to half of the number of outstanding packets when restarting from slow start. It is hard for the receiver to guess what is the number of outstanding packets. Once a receiver's *cwnd* goes out of sync with the sender's *cwnd*, the error will be carried forward in subsequent calculations and could have an accumulative effect.

2.2 ACC (Acknowledgment Congestion Control)

[3] also proposed a scheme (ACC) in which a receiver reduces the frequency of acknowledgment when some congestion signal is received from the sender. The ACC scheme makes use of the ECN bit setting capability of a router. In particular, if a router handling the upstream traffic notices a congestion, it will set the ECN bit of the ACKs going upstream. When a sender receives an ACK with the ECN bit set, it will set the bit on a data packet going out. A receiver seeing the ECN bit set in the data packet will infer that there is an upstream congestion and hence acknowledge more sparingly. This approach requires changes in network elements as well as the TCP software at both ends.

2.3 AF (Acknowledgment Filtering)

[14] proposed a method that collapses the ACKs at the transmission queue by allowing late coming ACKs to replace earlier ones of the same TCP flow. This technique is known as "Ack Filtering" in some literature. The idea makes use of the fact that TCP ACK with a sequence number N implicitly acknowledges those packets with sequence number smaller than N. When applied at cable modems or ADSL modems, this approach can reduce the number of ACKs transmitted by picking only what is valuable. While this approach can be very effective, it requires modifications of low level software of transmission devices such as cable modems or ADSL modems. Such modifications cannot be easily done by a user and require support from vendors of these devices. In addition, special attention needs to be made so that under severe upstream congestion, the scheme will not end up sending only one or two ACKs per window. If these ACKs are lost, a sender will have no choice but to time out. Furthermore, the algorithm must be careful of replacing duplicated ACKs in front of the queue, otherwise, the fast recovery mechanism at the sender will not work properly. It is possible to 'expand' incoming ACKs and resolve the above issues at the access router or at the sender at the price of added complexity.

2.4 Header Compression

[7] proposed a way to compress packet headers. The observation is that packets within the same flow often have some header fields having identical values, or values that change slowly. By replacing these fields with a connection identifier that is agreed upon between a compressor and a decompressor, a TCP/IP packet header can be compressed from 40 bytes to 5 bytes. In an asymmetric environment a terminal will be the compressor device and the router at the other end of the asymmetric link will be the decompressor device. However, there are some limitations in using header compression. There is no reduction in number of acknowledgment packets, in some asymmetric networks, e.g., cable networks, it means that a modem will still have to contend for a transmission slot, such overhead can be quite significant. Furthermore, the scheme requires co-operation of equipment at the network side.

2.5 Other Considerations

A consequence of acknowledging less often is that an ACK may cover a number of packets, this is also known as the `stretched ack' problem. The drawback is that a sender may send a burst of packets when it receives an ACK. This could create congestion and lead to packet loss. Acknowledgment regeneration has been proposed where multiple ACKs are regenerated when an ACK that acknowledges a number of packets is received. The regeneration process can either take place at the sender side just below the TCP module as in [3] or at the router on the upstream side of the asymmetric link as in [14].

In our discussion so far, we have concentrated on one-way traffic, i.e., TCP transfer in the broadband direction with a narrowband return path. A number of major applications over Internet have their traffic direction heavily downstream oriented, e.g., a home user

where the change of *cwnd* goes through the slow start phase followed by the congestion avoidance phase. At the beginning of a slow start, a TCP sender sets the value of *cwnd* to 1. It increases *cwnd* by 1 every time it receives a new ACK. Receiving an ACK is a good indication that the forward path (from a sender to a receiver) is not congested. Thus when a sender receives an ACK of one packet, it will be able to send out two more packets. However, this exponential growth does not carry on forever. Firstly, it is limited by the minimum of the sender's own buffer size and the receiver's buffer size dedicated to the connection. Secondly it is limited by a value called *ssthresh*. One can look at ssthresh as a safeguard parameter to moderate *cwnd*'s growth. The value of *ssthresh* is based on the recent congestion experience in the connection. When *cwnd* is equal to *ssthresh*, *cwnd* growth enters the congestion avoidance phase, a sender will only increment the *cwnd* by 1/*cwnd* per ACK received. This effectively increments *cwnd* by 1 at the end of each round trip until the upper bound on *cwnd* is reached. The idea is to slow down the growth of *cwnd* to a linear scale. In general, the more packet drops a connection encounters recently, the smaller is the value of ssthresh, and the earlier the *cwnd* growth will stop being exponential and become linear.

When there is a sender's timeout on waiting for an ACK, the congestion control algorithm regards it as a loss of the data packet in the forward path. This is an indication that the forward path may be congested. The sender sets the value of ssthresh to half of the number of outstanding unacknowledged packets and the value of *cwnd* to 1. The sender enters the slow start phase and grows *cwnd* again when it receives an ACK to its retransmission. To reduce the performance impact of occasional packet loss, a technique called fast retransmission was proposed. In TCP, a receiver receiving an out of order packet returns an ACK carrying the sequence number of next expected packet, which is the missed packet, and thereby generating a duplicated ACK. The idea of fast retransmission is that when a sender receives three of such duplicated ACKs, it will conclude that the packet is lost. Thus without waiting for a timeout, a sender will retransmit the missed packet, and reduces *cwnd* to a half. Thereafter, the window grows linearly via congestion avoidance. Notice that fast retransmission cannot take place unless the overall congestion window size is at least 4.

There are many different implementations of TCP. In our simulations and implementation discussed later, we compare our algorithm with TCP Reno, the most popular TCP implementation nowadays (more detailed description is in [12] and [2]). In the current TCP standard, a technique called delayed acknowledgment is implemented whereby when a receiver receives a data packet, it waits for the next one before sending out an ACK for both. The motivation was not really to reduce the number of ACK but rather to give the receiver some time to consume the packet, to update the receiver buffer size, and in the case of telnet, to piggy-back the ACK with the echoed characters. To avoid hold up, a receiver waits for the second packet for a maximum time of ATO (Acknowledgment Timeout) before sending out an ACK; the value of ATO is implementation dependent.

1.3 The Normalized Bandwidth Ratio

For one-way transfers, a simple ratio of the forward and backward bandwidth cannot reflect the effect of network asymmetry on TCP effectively. This is because the size of data packets is normally much larger than that of ACKs. Instead, we use the normalized bandwidth ratio *k*, (as defined in [3]), which is the ratio of the raw bandwidths divided by the ratio of the packet sizes used in the two directions. For example, if a network has 10Mbps of available bandwidth on forward channel and a 100Kbps of available bandwidth on backward channel, the raw bandwidth ratio would be 100. For a one-way transfer in the direction of higher bandwidth, assuming a data packet size of 1523 bytes (including framing) and an ACK packet size of 63 bytes, this gives a packet size ratio of 24.2. Therefore k is 100/25 = 4.13.

1.4 Performance of TCP over Asymmetric Networks

A TCP sender expects ACKs from the receiving side to advance the sliding congestion window, and to increase *cwnd*. If the bandwidth from the receiving side to the sender is very limited, an ACK may experience significant queuing delay at a transmission point from the receiver side. This will in turn slow the down the sending process of the TCP sender and reduce the throughput. Putting the above observation into the context of asymmetric networks, TCP transmission towards a terminal may not be able to capitalize on the large downstream bandwidth due to limited upstream bandwidth from the terminal. In addition, upstream data transfer can take up a lot of the upstream bandwidth, leaving little for upstream ACKs, and further worsening the performance of downstream transfer. In cable networks, because of the shared nature of upstream channel, a cable modem user may have very little control on the usage of upstream bandwidth. In ADSL networks, because a copper wire is dedicated to a user, there is a bit more control on the usage of upstream bandwidth. Using the example mentioned in section 1.3, where *k* is 4.13, if we acknowledge more often than one ACK for every 4.13 data packets, the return link will get saturated before the forward link does, possibly reducing the throughput that can be achieved in the forward direction. Because most platforms implement delayed acknowledgment, downstream TCP transfer will be affected when *k* is bigger than 2.

Our focus in this paper is on downstream transfer, i.e., in the high bandwidth direction. From now on, we will use the term 'sender' to refer to the TCP instance on a computer that is sending data downstream. We use the term 'receiver' to refer to the TCP instance on a terminal attached to an asymmetric network.

In this paper, we propose a modification to the TCP acknowledgment process that involves only changes in the TCP implementation at a terminal. This approach has an enormous advantage in that the any terminal can enjoy increased TCP performance in the downstream direction once its stack is patched, without requiring any change on the other end.

2. RELATED WORKS

There have been quite a number of efforts looking into the performance issue of TCP over asymmetric networks. One of the early research in TCP performance over asymmetric networks was presented in [9], which examined the network conditions affected by different asymmetry factors and suggested some methods to alleviate the congestion at the upstream transmission point, e.g., managing the packet queue via per connection upstream queues in the case of multiple connections. The idea is to give the ACKs a fair chance to be transmitted without being delayed by multiple large data packets. The Internet draft of the Tcpsat (TCP over Satellite) [1] gave a very good overview on a number of techniques that addressed bandwidth asymmetry in the context of satellite communications. These techniques are largely applicable to our problem. They include 1) delayed acknowledgment after slow start, 2) ACC (Acknowledgment congestion control), and 3) AF (Acknowledgment Filtering).

Improving TCP Performance Over Asymmetric Networks

Ivan Tam Ming-Chit*
Kent Ridge Digital Labs
21 Heng Mui Keng Terrace
Singapore 119613

Du Jinsong
National University of Singapore
Singapore

Weiguo Wang**
Kent Ridge Digital Labs
21 Heng Mui Keng Terrace
Singapore 119613

ABSTRACT

Bandwidth asymmetry is quite common among modern networks; e.g., ADSL, cable TV, wireless, and satellite link with a terrestrial return path. In these networks, the bandwidth over one direction can be orders of magnitude smaller than that over the other. The performance of TCP transfer in the high bandwidth direction can be severely reduced by the delay of acknowledgment packets experienced in the reverse direction. In this paper, we describe our proposed solution ACE (Acknowledgment Based on *Cwnd* Estimation). In comparison to other solutions, ACE requires only modification of the TCP stack at terminals attached to an asymmetric network. We evaluated the performance of ACE over a cable modem network by simulation. We have also implemented ACE on the Linux platform and tested it on a small testbed network with an emulated asymmetric link. The performance improvement was significant especially when there is a high degree of asymmetry.

Keywords

TCP, asymmetric networking, cable modem, ADSL, Ack filtering, congestion window estimation.

1. BACKGROUND

1.1 Asymmetric Networks

Emerging access network technologies such as cable modems over cable TV networks, ADSL over telephone lines, and wireless networks are asymmetric in nature. Usually, the bandwidth from a user terminal at home to a network operator is much smaller than the bandwidth from the network operator to the user. The first direction is often referred to as *upstream*, and the second direction is always referred to as *downstream*. For example in cable network, the downstream bandwidth per channel may be up to 27Mbps, but the upstream bandwidth per channel is only 600Kbps to 1.5Mbps. Each channel is shared by a number of users on the same cable plant. However, there can be multiple channels on each plant. In ADSL, the downstream bandwidth can be up to 8Mbps but the upstream bandwidth may only be up to 640Kbps depending on the distance from a home to a central office.

Unlike cable modem networks, an ADSL user has the sole use of the bandwidth over a twisted copper pair. In both types of networks, upstream transmission is very susceptible to noise; this further reduces the effective upstream bandwidth. Satellite transmission in one direction and terrestrial link in the other is yet another example. Because bandwidth on cross-ocean links is expensive, satellite transmission is usually used in the direction where high bandwidth is required. Cross-ocean link transmission is used in the direction where only low bandwidth is needed. For the rest of the paper, when we mention `terminal' we refer to a home or office terminal attached to an asymmetric network, where the transmission bandwidth from the terminal (upstream) is significantly smaller than the delivery bandwidth to the terminal (downstream).

In the rest of section 1, we will give a quick review on TCP, describe a more formal way to quantize network asymmetry and state the problem in more detail. In section 2, we discuss the works that have been done in this area. Section 3 describes our approach in detail. In section 4, we compare the performance of our solution with some of the others based on simulations of a cable network. We describe a preliminary implementation of our algorithm on the LINUX platform and report some performance results in section 5. Finally, in section 6, we summarize the paper.

1.2 A Quick Review of TCP

TCP provides reliable data transfer between two end points. It relies on a receiver sending back ACKs (acknowledgment packets) to inform a sender that data has been received. In order to avoid sending too many packets into the network at one time, TCP practices window based congestion control at the sender by controlling the congestion window. *Cwnd* (the size of congestion window) is in units of bytes, however, for the sake of simplicity, in the rest of the paper we will express *cwnd* in units of MSS (Maximum Segment Size). For applications such as ftp and http, a sender tends to send packets up to size of MSS, our implicit packet size in this paper will be that of a MSS. In the Internet environment this can be 1460 bytes. In essence, *cwnd* corresponds to the maximum number of packets outstanding in a network. When the number of packets sent out is equal to *cwnd*, a sender cannot send more until it receives an unduplicated ACK from the receiver. At this point, a sender can be sure that some of the earlier packets have been received, and are no longer inside the network. The sender reclaims the buffers of these packets, and advances a pointer on the sequence number of the most recently acknowledged packet. The number of packets that the sender can send out is always equal to *cwnd* minus the number of packets unacknowledged.

TCP implements the congestion control scheme suggested in [8],

* The first author can be contacted at email address: mtam@codex.cis.upenn.edu
** The third author can be contacted at email address: weiguo.wang@alcatel.com.sg