Proceedings of The DARPA Active Networks Conference and Exposition, San Francisco, CA, May 2002

Experience with Active Congestion Control[1]

Theodore Faber
University of Southern California/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
Phone: 310-822-1511
faber@isi.edu

Abstract

Active Congestion Control (ACC) applies active networking to feedback congestion control an a high bandwidth-delay product network, shortening the feedback loop by filtering traffic in the network near congestion. This paper describes recent simulation results showing the function of the system in TCP networks across a range of bandwidth-delay products. It also discusses the implementation of another version of the system in the ASP EE, an active networking execution environment implemented at USC/ISI. An assessment of ACC overhead in that implementation is made.

Keywords: Active Networks, Congestion Control

1. Introduction

Active Congestion Control (ACC)[1] is an enhancement to feedback congestion control that uses active networking techniques to shorten the control loop and improve network performance. The system improves feedback performance in a high bandwidth-delay product network with bursty traffic.

ACC makes feedback congestion control a more distributed process by including router actions as well as endpoint actions. ACC modifies traditional feedback systems by the addition of active applications (AAs) that detect congestion, modify packet flows in the network to relieve congestion, and finally notify endpoints of the modifications to their traffic. Unlike endpoint-only congestion control systems, the response begins at the congestion point and propagates to the endpoint, resulting in quicker response time and shorter congestion periods.

Active Networking is a networking paradigm based on the existence of highly programmable routers throughout the network infrastructure[2]. The ACC designers take the existence of such a highly programmable, secure infrastructure as given and use such an infrastructure to improve feedback congestion controls.

Feedback congestion controls are common in network protocols, from widely deployed byte-stream protocols such as the Internet Transmission Control Protocol (TCP)[3], to ATM cell-based protocols[4] and numerous paper designs[5-7]. These systems are least effective in the face of rapidly changing traffic characteristics in a network with a high bandwidth-delay product. This combination is difficult to control because the conditions change more quickly than the systems at the end can change their behavior to compensate. ACC attempts to compensate at or near the congestion point.

When an ACC router becomes congested, it chooses one of the traffic streams passing through it, and edits that stream's traffic so that it appears to the router as though the endpoint has responded immediately. As a result, the correction appears at the router more quickly than congestion could have been inferred at the endpoint or than such information could have been communicated by the congested router to the endpoint.

The router edits the traffic by asking the upstream router to filter out packets from the chosen endpoint. This upstream router effectively unsends some of the endpoint's packets. A further notification of this event is sent to the endpoint, which adjusts its future behavior, for example, by reducing its sending window and assuming the unsent packets to be negatively acknowledged.

These congestion actions are taken by AAs inserted by ACC into the programmable elements of the network. An AA, in the DARPA Active Networking Architecture[8], is any small, secure program running in the network that has access to a variety of services from the router. ACC AAs require access to the queueing levels of the router, to the header information of packets to be discarded by congestion, and the ability to set filters to remove specific packets from the data stream.

[1] This work was funded by DARPA ITO under contract DABT63-97-C-0049 (ARP)

ACC is not a specific feedback system, but an extension to existing systems. An ACC/TCP system would unsend packets and update endpoint state according to the rules of TCP congestion control. Other feedback systems can be extended by using a different set of AAs programmed to implement those backoff rules.

Although ACC for any one feedback system does not require activity of the network, active networks are a natural path for fast prototyping and deployment. Considering the variety of feedback systems currently proposed, this nimbleness is a significant asset.

We extend the analysis of ACC to demonstrate how ACC functions across a range of bandwidth-delay products. We show that ACC generally exerts control less often than endpoint controls but with greater effect. In a high bandwidth-delay product network with bursty cross traffic this property provides up to 20% improvement.

We also report on the implementation of ACC in the ASP execution environment (EE)[9], an active networking environment developed at the University of Southern California's Information Sciences Institute (USC/ISI). The implementation is complete, and shows low overhead. To facilitate ACC implementation and deployment, the Reliable Datagram Protocol (RDP)[10,11] was implemented in ASP, and we discuss that implementation as well.

Section 2 reviews the ACC system[1], Section 3 discusses new simulation results, Section 4 discusses ACC/RDP in the ASP EE, and Section 5 concludes.

2. The ACC System

ACC is designed to reduce the effect of a high bandwidth-delay product on feedback control by making congestion response a distributed network activity, rather than an endpoint activity. This reduces the time between congestion detection at the network node and the beginning of corrective action as observed at that node. Making control distributed requires network routers to take a more active role in the response to congestion and requiring endpoints to share knowledge of their internal state with routers. This model is the direct opposite of the way Internet congestion control has evolved.

The lack of safe router programmability and of limited router processing capacity in the Internet argues against complicated congestion systems in the routers, and this document is not intended to suggest that ACC is appropriate for that environment. However, in an active network, where router processing cycles are plentiful and securely available, ACC suggests a way to use them.

Figure 1 shows a passive network, like the Internet, reacting to congestion. The dashes represent packets, the circles endpoints, and the lines and squares, links and routers. The two endpoints on the left are sending at too high a rate to the same destination, resulting in congestion at the lower middle router. The two endpoints have inferred congestion exists and are slowing their sending rate, as shown by the larger space between packets entering the network at the left.

The congested router in Figure 1 will continue to experience a congestion-inducing load until the change in sender rate reaches it. The time this will take is proportional to the bandwidth-delay product of the network, which is intuitively the amount of data that the endpoints could have already en route. In practice the congested router may see a larger load than that, because most congestion inference strategies operate in real and therefore unpredictable networks. They rarely detect congestion in optimal time, because robustness requires tolerance for changing network conditions. Explicit indications of congestion, like TCP's Explicit Congestion Notification (ECN)[12], can reduce the time spent inferring congestion, but not the time lag between the router detecting the congestion and the rate reduction helping the situation.

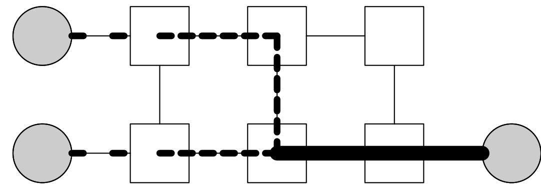

Figure 1: Congestion In A Passive Network

More sophisticated queueing and dropping strategies, e.g. Random Early Detection (RED)[13], reduce the time period that a router is overloaded by causing the endpoints to infer that the router is congested before congestion at the router has become critical. Simulations presented in Section 3 show that ACC is complimentary with RED.

Even with more sophisticated queueing and congestion notification, the basic problem of a lag between congestion's occurrence and its remedy remains.

Figure 2 shows the basic idea behind ACC reacting to congestion. This is roughly the same point in the recovery as depicted in Figure 1, but the ACC reaction spreads outward from the congestion point. The code at the congested router (represented by the small circle), has requested filtering of the two endpoints packets that approximates their eventual reaction to the congestion. This is represented in the figure by the wider spacing between packets inbound from the adjacent routers. These filters are set up much faster than the trip to the endpoint. While the filters operate, information from the congested router is also on its way to the endpoints so that they can change their state to react to the congestion in the long term, so the filters can be disengaged.

If the bottleneck link is the access link, ACC provides no advantage, because there are no other routers to offload filtering work. ECN will work just as well in that case. Although this is a real congestion situation, it is one that standard feedback is quite capable of coping with. ACC is intended for cases where the high bandwidth-delay product is a key cause of poor congestion response.

The ACC system is fail-safe, because if a message to an endpoint gets lost, that endpoint will eventually infer congestion and converge to the proper state. The changes ACC prescribes at the congestion point are the same changes that the endpoint would make if it detected congestion immediately, therefore they are the same changes that the endpoint will make when it infers congestion in the absence of ACC feedback. The filters placed by the ACC monitor will make the link appear temporarily lossier than it really is, but the filters will be removed either by the endpoint sending enough data or by the filter detecting that the endpoint has changed its behavior.

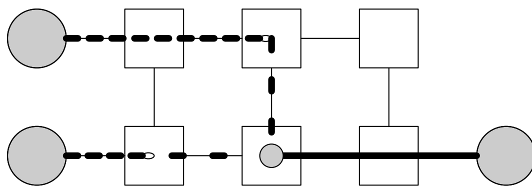

Figure 2: Congestion In An ACC Network

Because ACC is a framework for speeding up feedback congestion control systems, the criteria for congestion detection and the nature of the filters inserted in the network depend on the underlying feedback system being enhanced. For TCP congestion control, the congestion detection would be a packet being discarded, while TCP Vegas[14] would rely on queue length changes. The ACC framework can accommodate either.

A version of ACC that enhances TCP congestion control has been tested in simulation, showing up to 20% better throughput than TCP with RED for high bandwidth-delay product networks with bursty traffic[1]. This work will report on more details of the operation of ACC/TCP in Section 3.

Another version of ACC, designed to improve the performance of a TCP-like congestion control mechanism implemented in RDP has also been constructed. Section 4 will discuss that system.

3. Simulation Results

Initial simulation results reported earlier[1] showed that ACC/TCP performs better than standard TCP over large bandwidth-delay product networks. This section will demonstrate some of the details of that performance improvement, and show some shortcomings of ACC in lower bandwidth-delay product networks. We will review ACC/TCP and then explore a simulation of that protocol.

3.1. ACC/TCP

The window modulation algorithm in TCP is a classic linear increase/multiplicative decrease algorithm[3]. When congestion is detected by a retransmission timeout, the window is reduced to half its current size and a slow-start, described below, is initiated. When a full window of consecutive packets has been acknowledged without congestion being detected, the window is increased by one maximum-sized packet.

Furthermore, modern TCP stacks, TCP Reno or later, also incorporate fast retransmission. An endpoint that receives three consecutive acknowledgments of the same packet deduces that the next packet has been lost and adjusts its congestion window. Fast retransmission does not depend on a retransmission timeout, like standard congestion avoidance, and does not cause a slow-start.

Slow-start is a process by which TCP slowly opens its window from a size of one segment to the full current congestion window. The purpose is to avoid introducing packets too quickly when the pacing information provided by acknowledgments has been lost. Because this pacing information has not been lost when a fast retransmission occurs, it is safe to skip the slow-start.

ACC/TCP follows the same algorithm as TCP, including fast retransmission, except that traffic modification begins at the congested router. When a router detects a packet loss, it calculates the correct window size for the endpoint from information it provides in each packet, and forwards a packet with the new window size to the endpoint. Using TCP's window adjustment algorithm, the new window is half the old. Then the router installs a filter at the previous router on the connection's path that deletes all packets from that endpoint until it begins to act on the router's feedback (or it has detected congestion itself).

To support ACC/TCP, extra information including the endpoint's current estimate of round trip time, its window size, and slow-start and fast retransmission state is included in each packet. This information determines the feedback sent by the ACC implementation in the routers.

Under ACC, the preferred method of congestion detection is notification by the congested router, but the system still follows the TCP congestion control algorithm in the absence of that feedback. Because TCP only responds to one packet drop per round trip time, packets dropped by the filters will not cause endpoints to close their windows more quickly than they would in the face of

a single packet drop.

The reaction to congestion begins at the router with the packet filter installation. This is in contrast to TCP with Explicit Congestion Notification (ECN)[12], which uses routers to notify endpoints of congestion, but applies the corrective action from the endpoint. Thus ACC will react more quickly to congestion than TCP with ECN.

3.2. Simulation Studies

Simulation studies of ACC/TCP show that it increases endpoint's average throughput by up to 20% compared to standard TCP in the presence of bursty traffic and a high bandwidth-delay product, but incurs a small performance degradation in lower bandwidth-delay product networks. The simulations were done using links with bandwidths between 1.5 and 10 Mb/s. The bandwidth-delay product was varied by increasing the delay on an uncongested link.

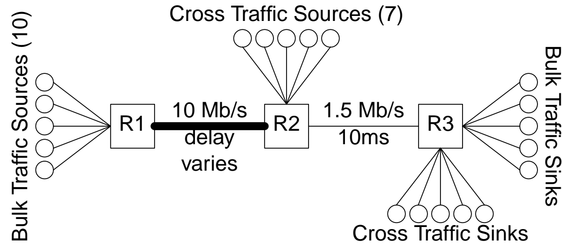

Figure 3: General Simulation Configuration

All simulations were made using ns, a simulator produced by the University of California Berkeley, the Lawrence Berkeley National Labs, and the Virtual Inter-Net Testbed (VINT) project[15]. The simulator was extended to implement the ACC algorithms in the routers and endpoints, but not to allow full AN programmability. The modified simulator does not compile or interpret code from simulated packets.

Figure 3 shows the template for simulations reported in this section. The bulk traffic endpoints send data continuously throughout the simulation. The cross traffic endpoints are discussed below. All links from an endpoint to a router have a delay of 10ms and bandwidth of 10 Mb/s. All endpoints use 1000 byte packets. Each simulation is repeated for different delays on the link from the R1 router to the R2 router. By varying that delay, the bandwidth-delay product that the bulk endpoints see is changed without changing the bandwidth-delay product that the cross traffic sees. Routers implement RED queueing, using identical parameters for ACC and non-ACC simulations.

The cross traffic endpoints are uncontrolled endpoints, sending on-off traffic with a burst and idle times of 2.5 sec and a sending rate of 100 kb/s.

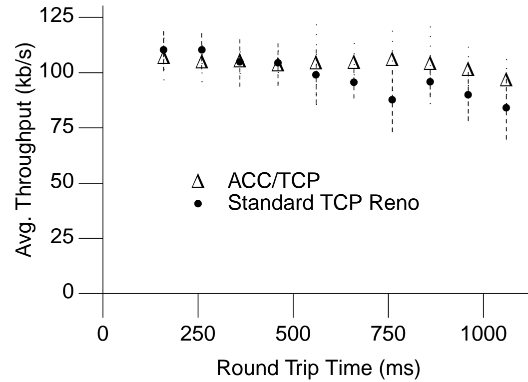

Figure 4: Simulation Throughput vs. RTT[2]

Each simulation lasted 200 seconds of simulated time. Averages and standard deviation error bars for those 10 runs are plotted. Figure 4 shows a throughput vs. round trip time (RTT) plot comparing the throughputs for TCP Reno and ACC/TCP. The RTT is proportional to the bandwidth-delay product in this scenario.

Figure 4 shows that ACC/TCP provides a more constant throughput across a range of bandwidth-delay products. At the very high values of RTT, ACC/TCP shows more than 20% improvement.

It is worth repeating that the parameter of interest here is the bandwidth-delay product. Similar results would appear if the round trip time was held constant and the bandwidth of the link modulated by a similar amount. If the reader believes the RTT values used are unbelievable, consider modulating the bandwidth similarly.

Although a 20% improvement is significant, this improvement also occurs in an operating regime where other proven congestion control enhancements are roughly equivalent and less effective than ACC. Figure 5 shows the same experiment for routers using drop-tail queues, routers using RED gateways, and TCP Reno endpoints with ECN enabled and ECN capable routers. The ECN gateways use the same RED parameters as the RED gateways in both this experiment and the ACC experiment depicted in Figure 4.

[2] A figure reporting results similar to these appeared in [1]. This figure differs in that the ACC algorithms used are somewhat more tuned, and that the throughput values are correct. The earlier work reported throughputs in the 8-10 kb/s range. The wrong packet size was used to calculate those lower rates and neither the author nor the reviewers noticed the error until months later.

Figure 5 shows that traditional congestion control systems all break down in approximately the same way a high bandwidth-delay network with bursty uncontrolled traffic. In this environment ACC/TCP provides a performance improvement that these other systems do not. In fact, they do not differentiate themselves appreciably from each other by this metric in this environment.

While it is encouraging that ACC/TCP operates well in the region for which it was designed, it is worth investigating why standard feedback works better in the low bandwidth-delay region.

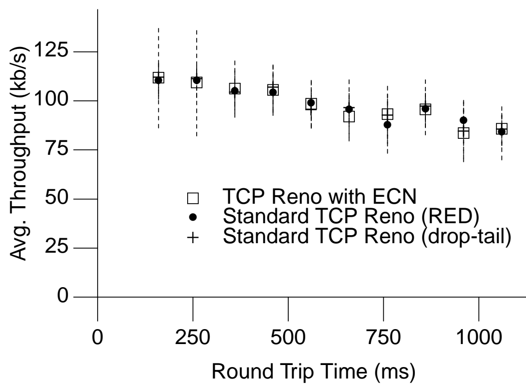

Figure 5: Various Congestion Controls

The answer lies in the rate at which control is applied and in the magnitude of control of the two systems. Figure 6 shows the rate of control application for the two systems, measured in window closes/second. Here we consider control to be a congestion response. All closes, whether requested explicitly from the ACC AA or initiated by the endpoint because it inferred congestion are counted. The value plotted is the average rate of closes of the window, per-endpoint, averaged across all the runs with standard deviation error bars.

Figure 6 shows that ACC provides less frequent close inputs than standard TCP across a range of bandwidth-delay products. The difference in control frequency is most pronounced at low bandwidth-delay products.

Secondarily, Figure 6 provides indirect evidence that ACC indeed reduces the number of congestion instances. Each control corresponds roughly to a congestion instance, and there are fewer controls for ACC.

Figure 7 plots the average amount that the endpoint's window is reduced at the time of a close event for the same experiments. This gives an indication of the magnitude of the change caused by the control.

Because ACC controls less often than standard TCP, we expect that its control inputs will be somewhat larger. However Figure 7 shows that at low bandwidth-delay products, the size of ACC's control input is smaller than that of standard TCP. The crossover point where the control magnitude switches (i.e., ACC's control input becomes larger) is the same point where the throughput values exchange in Figure 4.

The values in Figure 7 also indicate how large the sender's windows get before they close down. The sender's window must have been roughly twice the close magnitude when the sender made the change. The low numbers in the ACC plot at low bandwidth-delay products show that the sender's windows remain smaller than those of standard TCP. One explanation for this is that because the control loop is so tight in low bandwidth-delay products that the extra inputs from ACC are actually inhibiting window growth.

There are several possibilities for how to adjust ACC to deal with this problem. We could reduce the filter sizes or change the algorithms used to detect congestion for endpoints with a low RTT. But probably the simplest solution is to simply inhibit ACC for connections having a low RTT. Such connections really are not having the problem that ACC is designed to solve, and it is therefore getting in the way. Active networks programmability allows such selective deployment.

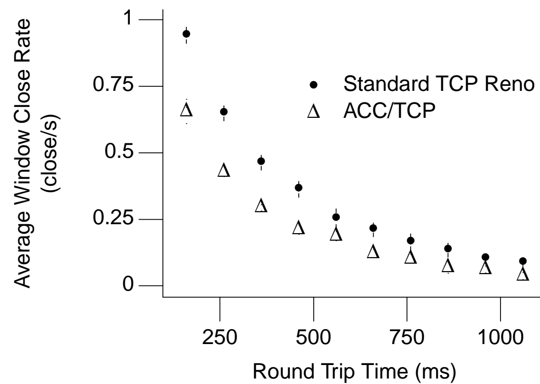

Figure 6: Control Rate

4. Implementation Under ASP

In order to assess the viability of implementing ACC in an active network, the authors undertook an implementation of ACC in USC/ISI's ASP EE[9]. Although every effort was made to implement ACC as efficiently and realistically as possible, the current state of active networking is far from a network in which secure processor cycles are readily available. Despite the limitations of current active networking environments, the implementation has pointed

out several useful lessons.

This section will briefly outline the relevant aspects of the ASP EE, discuss the implementation of ACC in that execution environment, and conclude with some simple assessments of the overhead incurred by the ACC implementation.

4.1. The ASP EE

The ASP EE is an execution environment produced by the Active Reservation Protocols (ARP) project at USC/ISI. ASP is a Java EE, which is to say that it is written in Java and hosts active applications written in Java that use the ASP Packet Programmer's Interface (PPI) to access operating system and network services.

The ACC developers chose to use the ASP EE because it supports virtual networking, implements packet diversion, and allows simple communication between long-lived AAs on separate nodes. Furthermore, because ACC involves changes to the endpoint protocol as well as programming the router, it was useful to have an environment in which the developers could directly manipulate protocol stacks. Because ASP development is centered at USC/ISI, where the authors are also employed, working with the ASP developers is easy.

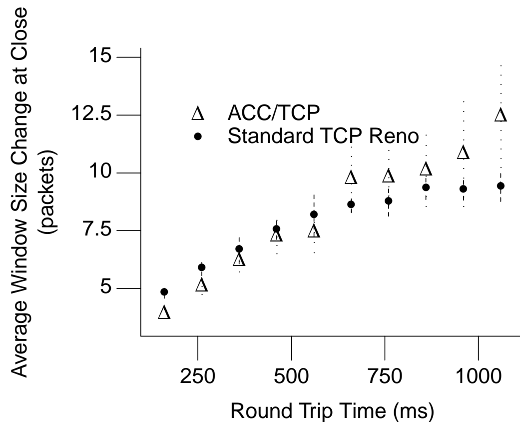

Figure 7: Control Magnitude

ASP provides a wide variety of services to AAs, only some of which are used by ACC. Among the ASP features ACC uses only peripherally are a process model that supports scheduling and memory isolation, application versioning, and dynamic code loading. ACC primarily makes use of the the VNet virtual networking system, facilities for packet interception, and AA to AA communication via NodeOS channels[8].

Using ASP's virtual networking is important to simplify deployment and testing. ASP's VNet protocols implement a simple virtual addressing scheme, using UDP as a link layer protocol. Because UDP is a multihop protocol, what appears as a single point-to-point link in a VNet topology may cross several Internet hops. Using VNet allowed us to easily configure various topologies for testing and debugging.

The ASP EE implements the NodeOS interface for packet diversion. Specifically, ACC filters can request packets for a specific RDP connection can be diverted to filter AAs that delete the packets. This facility was necessary for ACC implementation.

Finally, ASP allows AAs to communicate with other AAs. This requires AAs to be packet endpoints as well as sinks. Also, AAs must be long lived entities, allowing filters and monitors to maintain long-lived state about the connections they monitor. This general programmability is required to allow RDP to set filters and to communicate with endpoints.

The advantage of being able to co-develop the ASP EE cannot be overstated. Having a clear channel to the developers simplified many aspects of the ACC design and implementation. We also hope that the ACC development helped test new ASP primitives as well as making a reliable transport, RDP, available to other AAs running under ASP.

Another advantage of using ASP is an anticipated easy transition to the ABone[16]. We intend to use the ABone to test ACC's effectiveness on machines in a the real networking conditions.

4.2. Implementing ACC in ASP

The ACC implementation had three phases: choosing and implementing the congestion control that ACC was to enhance, implementing the congestion detection, implementing the filtering, and implementing the communication from filtering routers to endpoints. This section devotes a subsection to each of these.

Before describing those aspects in detail, we describe the operation of the ACC system as it is implemented in ASP. A transport protocol (RDP) is sending reliable packet streams and encounters congestion. The congested ASP node detects congestion, using the congestion detection features discussed below, and notifies the ACC AA running on that node. That ACC AA determines if the packet causing congestion is from an RDP connection, which ACC can affect. It determines the upstream router from information in the packet and starts an AA there to filter packets. That AA intercepts incoming RDP packets from the endpoint being controlled, and discards them until it has discarded a window's worth of traffic or until the endpoint has changed its behavior. The AA also contacts the endpoint to adjust its future sending patterns.

The process above may happen for several connections, depending on the severity of the connection.

4.2.1. The Underlying Protocol (RDP)

In order to take advantage of the ASP EE's virtual networking interface, and to have access to the protocol stack for modifications to endpoints and routers, the ACC developers implemented a simple reliable transport protocol with a feedback congestion control system. To avoid implementing the complexities of TCPs byte-streaming, we chose RDP[10,11].

RDP is attractive because it is fairly simple, provides reliable packet delivery, and has all the proper interfaces for a feedback congestion control. Because it is a packet-oriented protocol, it is significantly easier than TCP to implement. Although RDP has no feedback congestion control of its own, adapting TCP congestion control to it was relatively simple.

A useful side effect of implementing RDP for congestion control testing is that ASP acquired a congestion-controlled, reliable transport protocol that is used, among other things, to load code across virtual topologies.

The RDP implementation itself was initially done as an AA library, above the ASP kernel to allow the ACC developers full access to the protocol stack without having to modify ASP internals. Because ASP lacked a reliable transport protocol that used the VNet addresses, RDP was pushed into the ASP EE. As a result, there are still a few artifacts in the implementation, but none of these seem to impede performance excessively.

Version 2 of RDP[11] is implemented with a few changes. One change is that the packet format is modified slightly to mesh better with VNet, and uses VNet addresses rather than IP addresses. Although the IP checksum is implemented, it is usually turned off for performance reasons. Java code does not compute the IP checksum efficiently. It is possible to add a native method to do so, but for research purposes the end-to-end UDP checksum underlying the VNet links is sufficient.

Furthermore, a simplified version of TCP congestion control is also implemented. The system behaves like Reno TCP, but with a congestion window sized by packets, not bytes. The implementation increases the window by the inverse of the window size for each acknowledgment as TCP does, but until the fractional packet size reaches one packet, that capacity is unused. Either RDP can send a full sized packet, or it cannot send any sized packet. That is an approximation to TCP behavior, but a conservative one. Fast retransmission is implemented.

The protocol is extended to communicate more state information to routers and respond to control within the network. In addition to the information needed for RDP, the system also includes a few internal variables for the AAs to use in calculating filter parameters, like the current window size and slow-start state of the connection. The congestion control is further extended to allow an endpoint to interpret external messages from ACC AAs to set its window size in response to congestion.

4.2.2. Congestion Detection

ASP originally did not provide any kind of congestion detection features. Because ASP runs in Java on workstations, it generally left queueing of packets to the operating system. Because the Java Virtual Machine does not provide an interface to query operating system queues, ASP was blind to such queues building up.

As part of the ACC implementation, we extended the Java DatagramSocket class used to implement VNet links over UDP. The new class queues packets internally by pulling them from the operating system as fast as possible. A queue is kept in the socket, that ASP can access.

The extended socket allows a callback to be established to an AA when congestion is detected. A congestion callback method is called with the packet about to be dropped by ASP. Each node using ACC creates a local ACC AA to be called when congestion is detected.

When the ACC AA gets a packet that was discarded due to congestion, it determines if the packet is an RDP packet and therefore subject to ACC control. If so, it contacts upstream neighbors and sets filters. If not, currently nothing is done, although a chaining system may eventually be added so that different flavors of ACC may be running at the same time.

Currently this extended socket only supports drop-tail queueing, but can be extended to support more sophisticated queueing disciplines. The congestion detection is a simple threshold, set when ASP starts.

All the congestion detection extensions are turned off by default, so that users will not see unexpected results. However, we have seen no significant performance degradations using the congestion detection system.

4.2.3. Filter Insertion

Filters are started by the ASP EE's AA initiation facilities. Each ASP packet can contain an AA specification which contains enough information to route the packet to the appropriate running AA or to start such an AA if none is running at the node. An ACC AA that has received a congestion indication will start a filter AA by sending such a packet.

Included with the request to the filter AA is the identity of the connection, in this case the source VNet address and RDP port and destination VNet address and

RDP port. In addition the window size and slow-start state of the connection are sent, as well as the new congestion state values that were sent on to the host.

The filter uses the ASP packet diversion facilities to capture packets matching those parameters, and unsets the filter when either the requested number of packets have been filtered, or the endpoint has changed behavior. Such a change in behavior is detected using the information in the packet reflecting the endpoint's current state.

While debugging this facility, we found it useful to add a *flush* facility to the diversion channel. This facility was used to close the channel after all currently queued packets were delivered to the AA. The filter needs to read all packets sent by the endpoint while it is filtering, because some may need to be sent on to the endpoint, after the filtering AA has decided to terminate the filter. For example, if more than a window of packets have been queued, the unfiltered ones need to be sent on to the other endpoint. The standard close operation deleted any packets at intermediate points between the OS and ASP.

4.2.4. Communicating With Endpoints

Endpoints in our ACC implementation can either be AAs or a standalone user program that uses the ASP RDP stack. In either case the ACC AAs in the network need to be able to tell endpoints to modify their state and change their sending patterns.

The ACC AA communicates this information by creating an RDP packet and sending it on to the endpoint. This is the inverse operation from packet diversion. The AA opens a raw VNet channel, creates an appropriately formatted packet, and sends the packet through the channel. No retransmission is attempted, because lost control packets do not affect the correctness of the protocol.

4.2.5. Altering Network Performance

The testing and development of ACC required a simple, reproducible means of modifying the network performance seen by connections. We needed to be able to delay packets by a period of time or to drop them with a given probability. To accomplish this, we built an AA that intercepts all packets matching a given specification and delays or deletes them. Such an AA is commonly referred to as a flakeway[17]. The flakeway makes use of many of the same services as the ACC AA itself; however, because it is long lived, and packets passed to it must cross the ASP EE PPI twice, it can reduce the apparent bandwidth of the network.

4.3. Implementation Lessons

The implementation of ACC has three purposes: it is a feasibility check, to make sure that the system is implementable; it is an opportunity to evaluate the overhead of the system; and it is a chance to test the performance of the system under real network conditions. At this writing, the first two have been accomplished.

Realistic tests of the performance of the system imply testing across many hosts and in many topologies, which we have as yet been unable to complete. A congestion control system is designed to produce good results for a community of users, so many of its benefits are apparent only in the large. At this time we have not been able to deploy a broad enough test to make strong statements about the system behavior. Work to conduct such tests in the ABone is ongoing.

At this point, all elements of an ACC/RDP system have been developed and tested. Congestion detection has been integrated into the ASP EE, ACC monitoring AAs run and can both establish filters and inform endpoints of the new state. Filters are removed after appropriate conditions have been met. Each element of the system has been tested alone and combined with the other elements.

In December 2000, a prototype of the system that was functional with the exception of propagation of state changes was demonstrated interoperating with the AMP Node Operating System and an ANTS[18] denial of service avoidance system. This shows the robustness of an early implementation. The system has had its functionality fleshed out since then, and some elements that had been hard-coded into ASP has been moved more properly out to AAs.

4.3.1. Overhead

Early versions of ACC, including the one demonstrated in December 2000, showed considerable overhead. Transferring data using RDP without ACC got better performance that with ACC turned on. With two endpoints competing for bottleneck bandwidth, the overhead costs could be as high as 500 Mb/s.

Since that time several changes have been made to ASP and ACC that have reduced the ACC overhead. ASP's channel implementation, which is used by AAs to send and receive data has been substantially improved. The ACC code has generally been improved and optimized. Finally ACC functionality has been better partitioned so that it fits more cleanly into the ASP model. Generally that consists of putting as much functionality as possible into AAs rather than into special case code in the EE that may have been badly integrated.

ASPs packet forwarding code was largely rewritten and performance debugged for the ASP version 1.5 release. Redundant code was combined, and the combined code tuned. Thread synchronization was improved so that packets were more efficiently moved through the

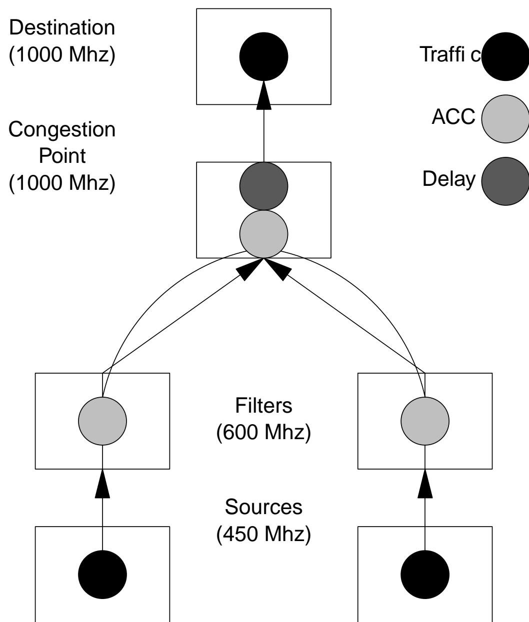

Figure 8: Topology For Overhead Experiments

stacks, and some livelock problems were removed. The result is both a faster forwarding system and a cleaner system for ACC to attach to.

After the ASP code was cleaned up, so was the ACC code. The ACC code was simpler, but there were still optimizations to be made. For example, ACC parses packets to determine which endpoint to filter and by how much. That code was optimized.

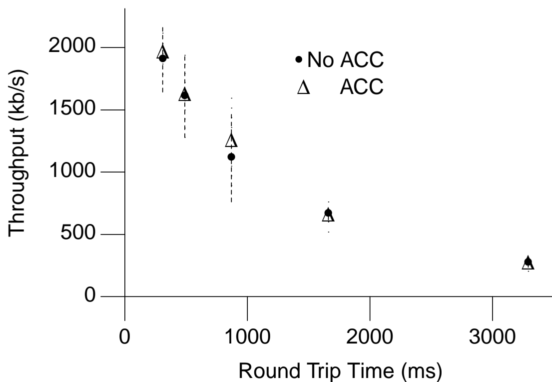

Figure 9: Throughput vs. RTT

Finally, code was partitioned more in accordance with ASP's PPI. Notably some of the filtering code that was doing ad hoc packet parsing was removed in favor of using ASP's packet diversion facilities.

To test the overhead of the improved system, we ran 10 repetitions of a 30 second bulk data transfer in the network shown in Figure 8. All the hosts are Intel Pentium class processors running Linux at the given clock rate. The two endpoints send at the highest rate they can sustain, which is sufficient to cause congestion at the node where their traffic converges. The converging node also runs a delay or fakeway AA that inserts a controlled delay into the system, increasing the bandwidth-delay product. The machines are interconnected with a switched 100 Mb/s Ethernet. That net was lightly loaded, and for the bandwidths used, is not a bottleneck.

An ACC monitoring AA also runs at the congested node, and when congestion is detected, it inserts filter AAs at the intermediate nodes on each leg. The monitor AA also notified the endpoints. During the test we monitored the system to make sure that the ACC components were being exercised, i.e., that there were filters being set.

Figure 9 shows the results of that experiment. Runs with ACC in use are plotted with a Δ and have error bars shown with a dotted line; runs without ACC are plotted with a • and use dashed lines for error bars. Average throughputs of 10 runs are plotted with standard deviation error bars. The plotted value is the sum of the throughputs of the two endpoints. In all cases the two endpoints have approximately the same throughput.

The primary result of the experiment is that the ACC overhead in simple congestion scenarios is very small across a range of bandwidth-delay products. Given that the initial implementation showed considerable overhead under the same conditions, the results are encouraging.

A secondary result is that the fakeway performs roughly as expected. The fakeway actually inserts the requested delay plus a constant overhead for packet processing. We currently leave the overhead visible rather than subtracting it out because the overhead may vary with processor speed.

The implementation has been successful in showing that the ACC system is implementable and has reasonable overhead. Experiments are continuing to show that this implementation can achieve benefits similar to the simulated system.

5. Conclusions

This paper has presented information about the workings of ACC both analytically and in implementation. Simulation results have shown that ACC's controls are effective in the regime for which it was designed, but somewhat detrimental outsided that regime. A working implementation has been presented that has a low implementation overhead.

The simulation results concentrated on exploring the frequency and magnitude of controls from ACC. Evidence has been presented that although ACC can improve

performance by as much as 20% in a high bandwidth-delay product net, it may also over-control a low bandwidth-delay product net. Fortunately, there is enough information in the system for ACC to detect the regime it is working in, and disable itself outside its effective operating range.

We have a working implementation of a fairly sophisticated ACC system in ASP. The system had been robust enough to test with two different underlying NodeOSes, AMP and UNIX (Solaris, Linux and FreeBSD). Although we have not yet been able to test ACC with many hosts in a wide area, initial estimates of the overhead indicate that the implementation has become fairly efficient. The implementation effort has helped improve ASP services as well.

In addition to the wide area testing, other areas for future work exist including moving filters toward the endpoints, exploring ways to combine router feedback, and adapting ACC to other feedback systems.

ACC filters could be moved closer to the endpoints both to reduce the time and resources used packets that will be deleted by them and to minimize the chance that a routing change moves the connection's route around the filter. Finding an optimal placement is another interesting research area.

Most of the congestion scenarios explored here have one bottleneck router that is setting endpoint congestion parameters. A condition where more than one router is doing so requires the feedback to be consolidated so that multiple filters are not working on the same traffic simultaneously and completely choking it off. Several forms of distributed agreement among routers seem promising, from a connection-following agreement protocol, to some form of filter suppression.

Most of the feedback systems that we have attached ACC to have been fairly simple additive increase/multiplicative decrease throughput modulating protocols. It would be interesting to see if ACC can also be used in conjunction with less traditional feedback systems. For example Dynamic Time Windows[19] modulates source burstiness rather than throughput. ACC may play a different role in enhancing such a system, for example, policing traffic to avoid queueing artifacts.

ACC has demonstrated improved performance with simple feedback congestion controls in high bandwidth-delay product networks, has shown itself to be implementable, and has several promising areas for further refinement.

Acknowledgments

The author thanks Deepak Ganesan, Gaurav Lochan, and Ashit Gosalia for the work they have done on the ACC/RDP ASP implementation. Each of them has at one time or another implemented and tested significant portions of the system. Vivek Shenoy also provided yeoman service in enhancing the ASP EE performance while keeping ACC working. The author also thanks the reviewers who helped improve this paper.

References

1. Theodore Faber, "ACC: Active Congestion Control," *IEEE Network,* vol. 12, no. 3 (July/August 1998).
2. David L. Tennenhouse and David J. Wetherall, "Towards an Active Network Architecture," *Computer Communication Review,* vol. 26, no. 2, pp. 5-18, ACM SIGCOMM (April 1996).
3. Van Jacobson, "Congestion Avoidance and Control," *Proc. SIGCOMM Symposium on Communications Architectures and Protocols,* pp. 314-329, ACM SIGCOMM, Stanford, CA (Aug 16-19 1988).
4. Raj Jain, "Congestion Control and Traffic Management in ATM Networks: Recent Advances and a Survey," *Computer Networks and ISDN Systems,* vol. 28, pp. 1723-1738, Elsevier Science B. V. (1996).
5. R. Jain and K. Ramakrishnan, "Congestion Avoidance in Computer Networks with a Connectionless Network Layer: Concepts, Goals, and Methodology," *Proc. IEEE Symposium on Computer Networks,* pp. 134-143, IEEE (1988).
6. D. Mitra and T. Seery, "Dynamic Adaptive Windows for High Speed Data Networks: Theory and Simulation," *Proc. SIGCOMM Symposium on Communications Architectures and Protocols,* pp. 30-29, ACM SIGCOMM, Philadelphia, PA (Sept 24-27, 1990).
7. D. Mitra and J. Seery, "Dynamic Adaptive Windows for High Speed Data Networks with Multiple Paths and Propagation Delays," *Proc. IEEE INFOCOM,* pp. 39-48, IEEE, Bal Harbour, FL (Apr. 7-9, 1991).
8. Larry Peterson, Yitzchak Gottlieb, Mike Hibler, Patrick Tullmann, Jay Lepreau, Stephen Schwab, Hrishikesh Dandekar, Andrew Purtell, and John Hartman, "An OS Interface for Active Routers," *IEEE Journal On Selected Areas In Communication,* vol. 19, no. 3, IEEE (March 2001).
9. Steven Berson, Robert Braden, Ted Faber, and Bob Lindell, "The ASP EE: An Active Network Execution Environment," *Proceedings of the 1st DARPA Active Networks Conference and Exposition,* IEEE (May 2002). unpublished.
10. David Velten, Robert Hinden, and Jack Sax, "Reliable Data Protocol," *RFC-908* (July 1984).
11. Craig Partridge and Robert Hinden, "Version 2 of the Reliable Data Protocol (RDP)," *RFC-1151* (April 1990).
12. Sally Floyd, "TCP and Explicit Congestion Notification," *ACM Computer Communication Review,* vol. 24, no. 5, pp. 10-23, ACM (October 1994).
13. Sally Floyd and Van Jacobson, "Random Early Detection gateways for Congestion Avoidance," *IEEE/ACM*

Transactions on Networking, vol. 1, no. 4, pp. 397-413 (August 1993).

14. Lawrence S. Brakmo, Sean W. O'Malley, and L. Peterson, "TCP Vegas: New techniques for congestion detection and avoidance," *Proceedings of ACM SIGCOMM Symposium on Communications Architectures and Protocols,* pp. 24-35, ACM, London, UK (August 1994).

15. Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu, "Advances in Network Simulation," *IEEE Computer,* vol. 33, no. 5, pp. 59-67, IEEE (May 2000).

16. Steven Berson, Bob Braden, and Livio Ricciulli, *Introduction To The ABone* (June 15, 2000), available from *http://www.isi.edu/abone/DOCUMENTS/ABoneIntro.ps*.

17. Jon Postel, "TCP and IP Bake Off," *RFC-1025* (September 1987).

18. David Wetherall, John Guttag, and David Tennenhouse, "ANTS: Network Services Without the Red Tape," *IEEE Computer,* vol. 32, no. 4, pp. 42-49, IEEE (April 1999).

19. Theodore Faber, Lawrence H. Landweber, and Amarnath Mukherjee, "Dynamic Time Windows: Packet Admission Control with Feedback," *Proc. ACM Symposium on Communications Architectures and Protocols,* pp. 124-135, ACM, Baltimore, MD (August 17-20, 1992).