Congestion Control Schemes for TCP/IP Networks

By Darius Buntinas

Abstract

This paper describes six methods for controlling congestion for TCP connections. The first four, Slow Start algorithm [5], Tri-S [8], DUAL [7], and TCP Vegas [1] treat the network as a black box, in that the only way to detect congestion is through packet loss and changes in round trip time, or throughput. The last two, Random Early Detection [4] and Explicit Congestion Notification [3] depend on the gateways to provide indications of congestion.

Table of Contents

- **Introduction**
- **Source based schemes**
  - Slow Start
  - Slow Start and Search: Tri-S
  - DUAL
  - TCP Vegas
- **Gateway based schemes**
  - Random Early Detection
  - Explicit Congestion Notification
- **Conclusion**
- **Works Cited**

Introduction

The algorithms are judged in several categories:

Performance.

The algorithm is judged by increases in throughput and decreases in retransmission as compared to other algorithms.

Fairness.

The algorithm is judged by how well connections share the resources with other connections and whether there are any biases towards connections with certain characteristics such as burstiness.

Compatibility with current technology.

The algorithm is judged by how well it interacts with current technology. Are the gains of this algorithm at the cost of other connections not using this algorithm? How will the algorithm perform in the presence of non-compliant sources?

Complexity.

The algorithm is judged by the complexity of implementation. Algorithms with lower overhead are preferred.

In the first section we will compare algorithms which use changes in performance, i.e., packet loss, increase in round trip time, change in throughput, to detect congestion. These algorithms have the advantage that they require only a new implementation of TCP and do not involve changing the network, as opposed to the algorithms discussed in the second section which require changing gateways and possibly adding fields to IP packets.

Back to the Table of Contents.

Source based schemes

Slow Start algorithm

Introduction

Jacobson and Karels developed a congestion control mechanism for TCP following a congestion collapse on the internet. Prior to this no congestion control mechanism was specified for TCP. Their method is based on ensuring the 'conservation of packets,' i.e., that the packets are entering the network at the same rate that they are exiting with a full window of packets in transit. A connection in this state is said to be in equilibrium. If all connections are in equilibrium, congestion collapse is unlikely The authors identified three ways for packet conservation to be violated:

1. The connection never reaches equilibrium.
2. A source sends a new packet before an old one exits.
3. Congestion in the network prevents a connection from reaching equilibrium.

TCP is 'self clocking,' i.e., the source sends a new packet only when it receives an ack for an old one and the rate at which the source receives acks is the same rate at which the destination receives packets. So the rate at which the source sends matches the rate of transmission over the slowest part of the connection.

Algorithm

To ensure that the connection reaches equilibrium, i.e., to avoid failure (1), a *slow-start* algorithm was developed. This algorithm added a congestion window. The minimum of the *congestion window* and the destination window is used when sending packets. Upon starting a connection, or restarting after a packet loss, the congestion window size is set to one packet. The congestion window is then increased by one packet upon the receipt of an ack. This would bring the size of the congestion window to that of the destination window in RTT $\log_2$ W time, where RTT is the round-trip-time and W is the destination window size in packets. Without the slow start mechanism an 8 packet burst from a 10 Mbps LAN through a 56 Kbps link could put the connection into a persistent failure mode.

Violation (2) would occur if the retransmit time is too short, making the source retransmit a packet that has not been received and is not lost. What is needed is a good way to estimate the round trip time:

$$\text{Err} = \text{Latest\_RTT\_Sample} - \text{RTT\_Estimate}$$

$$\text{RTT\_Estimate} = \text{RTT\_Estimate} + g*\text{Err}$$

where g is a 'gain' ($0 < g < 1$) which is related to the variance. This can be done quickly with integer arithmetic. This is an improvement over the previous method which used a constant to account for variance.

The authors also added exponential backoff for retransmitting packets that needed to be retransmitted more than once. This provides exponential dampening in the case where the round trip time increases faster than the RTT estimator can accommodate, and packets, which are not lost, are retransmitted.

Congestion avoidance takes care of violation (3). Lost packets are a good indication of congestion on the network. The authors state that the probability of a packet being lost due to transit is very rare. Furthermore, because of the improved round trip timer, it is a safe assumption that a timeout is due to network congestion. A additive increase / multiplicative decrease policy was used to avoid congestion.

Upon notification of network congestion, i.e., a timeout, the congestion window is set to half the current window size. Then for each ack for a new packet results in increasing the window by *1/congestion_window_size*. Note that if a packet times out, it is most likely that the source window is empty and nothing is being transmitted and a slow-start is required. In that case, the slow-start algorithm will increase the window (by one packet per ack) to half the previous window size, at which point the congestion avoidance algorithm takes over.

Performance

To test the effectiveness of their congestion control scheme, they compared their implementation of TCP to the previous implementation. They had four TCP conversations going between eight computers on two 10 Mbps LANs with a 230.4 Kbps link over the internet. They saw a 200% increase in effective bandwidth with the slow-start algorithm alone. The original implementation used only 35% of the available bandwidth due to retransmits. In another experiment, using the same network setup, the TCP implementation without congestion avoidance resulted in 4,000 of 11,000 packets sent were retransmitted packets as opposed to 89 of 8,281 with the new implementation.

Discussion

According to [8] this scheme suffers from oscillations when the network is overloaded. Because the window size is increased until a packet is dropped to indicate congestion, the bottleneck node is kept at maximum capacity. The window size oscillates between the maximum window size allowable by the bottleneck and half that size on timeouts. This leads to long queuing delays and high delay variation.

Wang, et. al., also point out that this scheme is biased toward connections with fewer hops. The additive increase / multiplicative decrease algorithm forces, eventually, the window sizes for all connections to be equal. However, this would take many iterations, in which time, the shorter connections, with shorter round trip times, would increase faster.

Back to the Table of Contents.

Slow Start and Search: Tri-S

Introduction

Wang, et. al. proposed an improved congestion control scheme called Slow Start and Search (Tri-S) in [8]. He sites the drawbacks to the Slow Start scheme as noted above. The Tri-S scheme uses a change in throughput as an indication of congestion. The algorithm computes the normalized throughput gradient (NTG) and compares it to thresholds to keep the connection at the optimal operating point which maximizes throughput while avoiding congestion.

As load is increased throughput increases linearly under light load and levels off as the network reaches its capacity. The gradient of the throughput curve can be used as an indication of congestion. As connections are added the curve starts to level out and as connections are removed the curve becomes more linear. The throughput gradient is

$$TG(W_n) = (T(W_n)-T(W_{n-1})) / (W_n-W_{n-1})$$

where W_n is the window size of the nth window, and $T(W_n)$ is the throughput at the window size of W_n. This is normalized to account for round trip times of different connections:

$$NTG(W_n) = TG(W_n)/TG(W_1)$$

The NTG ranges over [1,0] with changes in traffic. In a lightly loaded network, NTG would be close to 1, because increase in throughput is approximately proportional to an increase in load. As the network load approaches capacity, NTG approaches 0.

The average throughput of a connection while the nth packet is in transit is $T_n = W_n/RTT_n$ where W_n is the number of bytes in the network at the time the nth packet is transmitted i.e., the size of the window when the nth packet was sent, and RTT_n is the round-trip-time of the nth packet. If the window size is increased by an equal amount every time, the NTG would be:

$$NTG(W_n) = (T(W_n)-T(W_{n-1})) / T(W_1)$$

Algorithm

When a connection is initiated, the window size is set to one *basic adjustment unit* (BAU) and is increased by one packet for each ack received until the window size is equal to the destination window.

When a packet times out, the window size is set to one BAU and is increased by one BAU for each ack while the NTG is above a certain threshold, NTG_d.

At this point the window size is increased, for each ack received, by *BAU/ Current_Window_Size*. This will increase the window by one BAU per window of packets acked. When the increase in window size is larger than a whole packet, NTG is computed. If the NTG is less than a certain threshold NTG_i, the window is decreased by one packet. Otherwise leave the window size unchanged.

Upon initiation, the connection attempts to utilize all of the bandwidth it can. If it is unavailable, the buffers at the bottleneck node will overflow, packets will be lost for the connections going through the bottleneck and all connections will restart and reach the operating point. This allows new connections to "muscle" their way into the network to use their fair share of the resources.

Notice that while the BAU is small the algorithm uses an additive increase (BAU/Current_Window_Size per ack) / multiplicative decrease (1 packet per ack) method also.

Performance

The authors ran simulations with five different scenarios: 1) one connection on one path, 2) two connections sharing a path, 3) two connections sharing part of a path, 4) one connection joining a steady flow for part of a path, 5) and one connection sharing part of the path of a steady stream terminating. The values of the parameters were as follows BAU = 1 packet, $NTG_i = NTG_d = 0.5$.

In the first scenario, where one connection ran by itself, the throughput of the Tri-S scheme is slightly higher, but the average queue length is only half as large.

In the second scenario, where two connections shared a path, for the Tri-S scheme the bandwidth is balanced very well between the two connections. This algorithm provides better fairness than the Slow Start algorithm.

In the third scenario where the two connections share only part of the path, the bandwidth was not shared as fairly as in the previous scenario, but still better than the Slow Start scheme. The reason why the connections were not balanced well is because the algorithm increases the window size once per round trip time. The connection with the shorter round trip time will increase its window faster than a longer connection.

The results of the fourth scenario, where a connection joins a connection in progress, are similar to the previous indicating that the added connection provided very little disruption to the steady flow.

The last scenario, where a connection is terminated while another remains, when compared to the first scenario shows that the Tri-S scheme was able to utilize the added bandwidth better than the Slow-Start scheme.

Discussion

Setting the thresholds to the correct values is very important. NTG_d determines the operating point of the

connections while NTG_i determines the sensitivity of detecting freed up bandwidth. Setting NTG_d to below 0.3 or 0.8 above has set the average queue length at the bottleneck to be above 20 or below 0.5, respectively, during experimentation.

The algorithm uses the round trip time of the first packet as an estimate of the propagation delay of the connection. This may give an incorrect estimate of the propagation delay if the connection is already under load. Wang, et. al., acknowledge this problem and state that in their experiments the effect of queuing delay on the estimate of the round trip time is small.

As discussed before this scheme does suffer from a bias towards connections with shorter round trip times because the window size is increased for each ack, so shorter connections, with shorter round trip times will receive acks at a faster rate resulting in a larger window for shorter connections.

Back to the Table of Contents.

DUAL

Introduction

In [7] Wang et. al. propose a method called DUAL to correct the oscillation problem associated with the Slow Start algorithm. This is similar to the algorithm described in [8] in that it uses the round trip time to detect congestion as well as packet loss as in the Slow Start algorithm [5].

The round trip time of a packed consists of the propagation delay and the queuing delay. The minimum round trip time would be equal to the propagation delay:

$$RTT_{min} = D_p$$

The maximum round trip time would be the sum of the propagation delay and the delay for the bottleneck node to process a full queue:

$$RTT_{max} = D_p + Max\_Queue\_Size/Processing\_Rate$$

The Slow Start algorithm detects congestion when a packet is lost due to a queue overflow. The solution used in DUAL is to estimate RTT_{min} and RTT_{max} and using a threshold, avoid overflowing the queue at the bottleneck node. They defined this threshold as follows:

$$RTT_i = (1-a)RTT_{min} + aRTT_{max}$$

for some $a < 1$. They chose $a = 0.5$ to stay well away from the maximum queue capacity.

Algorithm

DUAL uses the same slow-start algorithm for initiating and restarting a connection. The algorithm differs from Slow Start in that every two round trip times DUAL checks if the current RTT is greater than RTT_i and reduces the congestion window by 7/8. It also recomputes RTT_{max} and RTT_{min} with every new RTT measurement. On a timeout, in addition to reducing the congestion window to one packet and restarting

slowly as in the Slow Start algorithm, it resets RTT_{max} and RTT_{min} to 0 and infinity respectively.

Performance

DUAL was simulated in [7] under three scenarios: 1) single connection on a path, 2) two connections share a path but one starts before the other and 3) two way traffic.

In the first scenario the DUAL showed an almost identical slow start phase, it did show a substantial reduction of oscillations as compared with the Slow Start algorithm.

The second scenario tests the ability of the algorithm to adjust to the addition of connections. The results are similar to that observed in scenario one, except that as the algorithms progress, the window sizes of the connections converge. This is exactly what is expected.

The last scenario tests the effect of rapid queue fluctuation on the round trip time based algorithm. The algorithm performed well despite the fluctuations. Wang, et. al. attribute this to the fact that many of the packets in the queue are ack packets which are small and therefore the actual queuing delay change is smooth.

Discussion

Because the window size is based on RTT_{min} and RTT_{max}, their accurate estimation is important. The Wang, et. al. noted that RTT_{max} is fairly accurate when calculated just before a buffer overflow. However if RTT_{min} is estimated when there are multiple connections on the path the RTT_i threshold would be too high. When different flows obtain different RTT_{min} values the bandwidth is shared unevenly. Wang et. al. claim that in their simulations, during a slow start the load is relatively light so queuing time is low and that the RTT_{min} estimate was a reasonable approximation of the propagation delay.

Back to the Table of Contents.

TCP Vegas

Introduction

TCP Vegas is a new implementation of TCP proposed by Brakmo et. al. in [1]. The authors claim an 40 to 70% increase in throughput and one fifth to one half of the packet losses as compared to the current implementation of TCP (Reno which implements the Slow Start algorithm with the addition of Fast Retransmit and Fast Recovery). Vegas compares the measured throughput rate to the expected, or ideal, throughput rate. This differs from the Tri-S scheme which looks at the change in throughput.

Algorithm

Vegas uses a new retransmission mechanism. This is an improvement over the Fast Retransmit mechanism. In the original Fast Retransmit mechanism, three duplicate acks indicate the loss of a packet, so a packet can be retransmitted before it times out. Vegas uses a timestamp for each packet sent to

calculate the round trip time on each ack received. When a duplicate ack is received Vegas checks to see if the difference between the timestamp for that packet and the current time is greater than the timeout value. If it is, Vegas retransmits the packet without having to wait for the third duplicate message. This is an improvement over Reno in that in many cases the window may be so small that the source will not receive three duplicate acks, or the acks may be lost in the network.

Upon receiving a non-duplicate ack, if it is the first or second ack since a retransmission, Vegas checks to see if the time interval since the packet was sent is larger than the timeout value and retransmits the packet if so. If there are any packets that have been lost since the retransmission they will be retransmitted without having to wait for duplicate acks.

To avoid congestion, Vegas compares the *actual* throughput to the *expected* throughput. The expected throughput is defined as the minimum of all measured throughputs. The actual throughput is the number of bytes transmitted between the time a packet is transmitted and its ack is received divided by the round trip time of that packet.

Vegas then compares the difference of the expected and the actual throughputs to thresholds *a* and *b*. When the difference is smaller than *a*, the window size is increased linearly and when the difference is greater than *b* the window size is decreased linearly.

Reno's Slow-Start mechanism, according to Brakmo et. al., can lead to many losses. Because the window size is doubled every round trip time, when the bottleneck is finally overloaded, the expected losses are half the current window. As network bandwidth increases the number of packets lost in this manner will also increase. Brakmo et. al. propose a modified slow start mechanism where the window size is doubled only every other round trip time. So every other round trip time the window is not changed which allows for an accurate comparison of the expected and actual throughput. The difference is compared to a new threshold called c at which point the algorithm switches to the linear increase / decrease mode described above.

Performance

Vegas was simulated to test its performance when used in the same environment as Reno. In this experiment, a 1MB file was transferred using one implementation with a 300KB file being transferred at the same time with another implementation. In all four combinations, Vegas performed better with higher throughput and fewer retransmissions.

In another experiment the performance of Reno and Vegas, with two sets of values for *a* and *b*, were compared on a network with background traffic. For both configurations of Vegas, throughput was increased by more than 50% and the number of retransmissions was about half that of Reno.

Other experiments included an experiment like the first one but with background traffic. The results were similar to that of the first experiment. Vegas and Reno were compared on a network with two way traffic. Again the results were similar to those in the second experiment.

Another experiment was to test the fairness of both schemes by putting multiple connections through a bottleneck. They tested the scheme were some connections that had different propagation delays and where the propagation delays were the same. In the case where the propagation delays were the same Reno was slightly more fair than Vegas, but in the case where the propagation delays were not equal,

Vegas was more fair than Reno. The authors claim that overall, Vegas is at least as fair as Reno.

In transfers over the internet Vegas had a higher throughput fewer retransmits than Reno.

Discussion

Because this implementation is an enhancement of Reno, in the worst case, e.g., where the load increases or the number of router buffers decreases, Vegas "falls back" to the behavior of Reno.

In this algorithm as with Tri-S and DUAL, the correct estimation of the BaseRTT is important. If the BaseRTT estimate is too high the algorithm is in danger of overloading the queues. As in the previous two schemes, the authors believe that the current method of estimating BaseRTT is sufficient.

Back to the Table of Contents.

Gateway based schemes

A problem with end to end congestion control schemes is that the presence of congestion is detected through the effects of congestion, e.g., packet loss, increased round trip time, changes in the throughput gradient, etc., rather than the congestion itself e.g. overflowing queues. There can also be a problem with fairness and non-compliant sources. It seems logical then to place the congestion control mechanism at the location of the congestion, i.e., the gateways. The gateway knows how congested it is and can notify sources explicitly, either by marking a congestion bit, or by dropping packets. The main drawback to marking packets with a congestion bit, as opposed to simply dropping them, is that TCP makes no provision for it currently. Floyd in [3] states that some have proposed sending Source Quench packets as ECN mesages. Source Quench messages have been criticized as consuming network bandwidth in a congested network making the problem worse.

Back to the Table of Contents.

Random Early Detection

Introduction

One method for gateways to notify the source of congestion is to drop packets. This is done automatically when the queue is full. The default algorithm is when the queue is full drop the any new packets. This is called Tail Drop. Another algorithm is when the queue is full and a new packet arrives, one packet is randomly chosen from the queue to be dropped. Floyd [4] mentions these schemes in their paper. The drawback to Tail Drop and Random Drop gateways is that it drops packets from many connections and causes them to decrease their windows at the same time resulting in a loss of throughput.

Early Random Drop gateways are a slight improvement over Tail Drop and Random Drop in that they drop incoming packets with a fixed probability whenever the queue size exceeds a certain threshold. Floyd notes that none of these algorithms handled misbehaving connections well.

Other methods noted by Floyd are the DECbit scheme and IP Source Quench. In the DECbit scheme the

gateway calculates the average queue length over a period of time and marking every packet (rather than dropping it) if the average is greater than 1. The sources then decrease their window size multiplicatively once every two round trip times if at least half of the packets in the last window were marked, otherwise the window is increased additively.

Algorithm

Floyd [4] proposes a new method called Random Early Detection (RED) gateways. In this method, once the average queue is above a certain threshold the packets are dropped (or marked) with a certain probability related to the queue size. In describing this paper we will consider only the case were the RED gateway drops packes to indicate congestion rather than marking them. In describing the next algorithm we will discuss marking packets.

To calculate the average queue size the algorithm uses an exponentially weighted moving average:

$$avg = (1-w_q)avg + w_q*Queue\_Size$$

The author goes into detail describing how to determine the upper and lower bounds for w_q.

The probability to drop a packet, p_b, varies linearly from 0 to max_p as the average queue length varies from the minimum threshold, minth, to the maximum threshold, maxth. The chance that a packet is dropped is also related to the size of the packet. The probability to drop an individual packet, p_a, increases as the number of packets since the last dropped packet, count, increases:

$$p_b = max_b(avg-min_{th})/(max_{th}-min_{th})$$

$$p_b = p_b*Packet\_Size/Max\_Packet\_Size$$

$$p_a = p_b/(1-count*p_b)$$

In this algorithm as the congestion increases, more packets are dropped. Larger packets are more likely to be dropped than smaller packets which use less resources.

Performance

Simulations were run for Tail Drop, Random Drop and RED gateways. The RED gateway showed higher throughput for smaller buffer sizes than the other algorithms. It was also shown that RED was not as biased against burst traffic as were Tail Drop or Random Drop.

Discussion

When RED is implemented to drop packets rather than mark them, it handles misbehaving sources well. If a source is using more than its fair share of the bandwidth, then by the probabilistic function, more of its packets will be dropped. However if the gateway marks the packets, it is up to the sources to comply.

Back to the Table of Contents.

Explicit Congestion Notification

Introduction

In [3] the author considers RED gateways that produce explicit congestion notification in the form of a message to the source or by marking a congestion notification bit on the packet. New guidelines are proposed by Floyd for the response of the source to an ECN. Because an ECN is not an indication of queue overflow it would be undesirable for the source to treat an ECN as it would a dropped packet and start TCP's conservative congestion control mechanism. The guidelines are as follows:

- The response to ECN should, over time, be similar to that of a lost packet.
- The immediate response should be much less conservative than the response to a dropped packet
- The receipt of a single ECN should trigger a response.
- The source should respond to an ECN only once per round trip time.
- The source should follow the existing rules for sending packets in response to acks.
- The source should slow-start and retransmit on a dropped packet.

Algorithm

The algorithm implemented in the simulations in [3] is as follows. When a source receives an ECN message and no responses to congestion have been made in the last round trip time, both the congestion window and the slow start threshold are halved. Note that since no packet was lost the source shouldn't slow start.

If the source receives three duplicate acks indicating a lost packet and it has not responded to congestion in the last round trip time, it should follow the Fast Retransmit and Fast Recovery procedures. The source shouldn't respond to an ECN or another set of duplicate acks until all packets which were outstanding at the time it first responded to congestion have been acked.

If the source has responded to an ECN and soon after receives three duplicate acks it should retransmit the dropped packet but it should not change the congestion window or the slow start threshold because that has already been done on the receipt of the ECN.

Performance

The simulation compared Tail Drop gateways, RED gateways that dropped packets rather than marking them and RED gateways which set an ECN bit in the packet. The simulations were run for a LAN and a WAN configuration.

In the both the LAN and the WAN scenarios, the throughput of bulk data transfers was high and the delay of telnet packets was low for the ECN capable RED. The ECN capable RED shows much improvement over the other schemes when telnet delay is compared.

Discussion

Floyd identifies two potential problems with their scheme: non-compliant sources and the loss of ECN

messages. The problem of a non-compliant source is a hazard for any congestion control algorithm. If there can be a source which ignores ECN messages, there could also be a source that does not respond to packet drops. With a congestion control scheme that uses packet drops to control congestion, any source interested in maximizing throughput cannot ignore packet drops, however. The author states non-compliant connections can cause problems in non ECN environments as well as in ECN environments. With regards to ECN message loss, since the RED gateway continually sets ECN bits while congestion persists the loss of an ECN message will not fundamentally affect the algorithm

One major hurdle to the application of this algorithm to TCP is the incremental deployment of ECN capable gateways and sources. One proposed solution is to provide two bits in the header to indicate ECN compliance and the presence of congestion. This can also be done with one bit, where "off" represents ECN capability and "on" would represent either no ECN capability or congestion notification. When a gateway marks a packet with the bit "off" it simply switches the bit "on." If the gateway wants to mark a packet with the bit "on" it simply discards it. Notice that the one bit scheme would not work for two way traffic where data packets travel in one direction and acks in the other. If a congested node sets the ECN bit for one packet, as the ack returns to that node, it will be discarded.

Back to the Table of Contents.

Conclusion

We have described several algorithms for congestion control for TCP. The first four of these were improvements upon the implementation of previous technology. With these source based schemes, it is easy for implement a new scheme and use it without having to worry about others who may not be using the same implementation. Whereas with the gateway based schemes, global standardization is needed to implement new schemes. Furthermore, it would seem that algorithms which work on gateways would require more overhead, per connection, than the source based scheme, by considering how many computations need to be done per packet for a multi-hop connection. However, algorithms which operate at the location of the congestion, i.e., gateway based, should provide better feedback than algorithms which have to estimate the degree of congestion from performance.

While improvements may still be made to the source based schemes, it would seem that the greatest improvement to congestion control would come from gateway based schemes, or a combination there of, such as [4]. The likelihood of such schemes being implemented is low due to the problem related to converting existing equipment.

Back to the Table of Contents.

Works Cited

1. Lawrence S. Brakmo and Sean W. O'Malley, "TCP Vegas: New Techniques for Congestion Detection and Avoidance," in *SIGCOMM '94 Conference on Communications Architectures and Protocols*, (London, United Kingdom), pp. 24-35, Oct. 1994.

   This paper describes Vegas, a new implementation of TCP which is an improvement over the

current implementation in BSD Unix.

2. Douglas E. Comer, *Internetworking with TCP/IP Volume I*, Englewood Cliffs, New Jersey: Prentice Hall, 1991.

   This book is a good description of TCP/IP and associated protocols.

3. Sally Floyd, "TCP and Explicit Congestion Notification," *ACM Computer Communication Review*, vol 24, pp. 8-23, Oct. 1995.

   This paper discusses Explicit Congestion Notification (ECN) mechanisms for TCP/IP. Guidelines for ECN mechanisms are proposed and simulations are used to explore benefits and drawbacks.

4. Sally Floyd and Van Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, pp. 397--413, Aug. 1993.

   The RED scheme is described in this paper. The authors include a discussion on other congestion control mechanisms for TCP/IP.

5. Van Jacobson, "Congestion Avoidance and Control," *ACM Computer Communication Review*, vol. 18, pp. 314--329, Aug. 1988. *Proceedings of the Sigcomm '88 Symposium* in Stanford, CA, August, 1988.

   This paper describes the Slow Start mechanism.

6. Raj Jain, "A Delay-Based Approach for Congestion Avoidance in Interconnected Heterogeneous Computer Networks," *ACM Computer Communication Review*, vol. 19, pp. 56-71, Oct 1989.

   Using a black-box model of the network, the authors show an approach to detect congestion based on round trip delay. Congestion is avoided by adjusting window size.

7. Zheng Wang and Jon Crowcroft, "Eliminating Periodic Packet Losses in the 4.3-Tahoe BSD TCP Congestion Control Algorithm," *ACM Computer Communication Review*, vol. 22, pp. 9--16, Apr. 1992.

   This paper describes the DUAL congestion control mechanism. This scheme modifies the Slow Start scheme to eliminate oscillatory behavior.

8. Zheng Wang and Jon Crowcroft, "A New Congestion Control Scheme: Slow Start and Search (Tri-S)," *ACM Computer Communication Review*, vol. 21, pp 32-43, Jan 1991

   The Tri-S scheme is described in this paper. This scheme detects congestion through round trip delay as well as packet loss.

Back to the Table of Contents.

Other Reports on Recent Advances in Networking 1995

Back to Raj Jain's Home Page

Last modified August 25, 1995.

Raj Jain is now at Washington University in Saint Louis, jain@cse.wustl.edu http://www.cse.wustl.edu/~jain/