Proceedings of 2nd National Conference on Challenges & Opportunities in Information Technology (COIT-2008)
RIMT-IET, Mandi Gobindgarh. March 29, 2008

Modification in Congestion Control Mechanism for Throughput and Loss delay in Wireless Networks

Jitendra Singh*, Mr. R.L.Ujjwal**
Guru Gobind Singh Indraprastha University Delhi, India
*jitu.usit.2006@gmail.com, **ujjwal_rl@rediffmail.com

Abstract: In this paper, I consider the problem of congestion control over wireless networks. For preventing congestion control over wireless networks we propose a slow start mechanism in TCP window size for enhancement of throughput and traffic shaping for loss delay and wireless links we will proposed the behavior over wireless link, characterized by time varying capacity, random delay and non-congested packet losses. In start slow first transition from the slow start state to congestion avoidance state happens after the first packet losses. If packet losses are eliminated. We need a slow start like mechanism that can grow the window size quickly when the network is un congested & based on received packet marks, switch to congestion avoidance before the bottleneck queue grows large. In traffic shaping, the sender could use some basic traffic shaping e.g. keep a minimum inter-packet interval corresponding to half the estimated RTT, divided by the current window size in packet. Throughput is also based on bandwidth & delay in networks. My implementation and simulation results show that congestion control mechanism over wireless networks can obtain throughput enhancement as well as loss delay.

KEYWORDS: Congestion, TCP, Loss delay, Bandwidth, Timers, Wireless networks, NS-2.

I. INTRODUCTION

Two of the main objectives of congestion control are to keep the load of the network close to the available capacity, and at the same time share the available capacity fairly between flows. In window-based control, the most important concept in tcp congestion control is that of the congestion window. The window is the amount of data that has been sent, but for which no acknowledgement has yet been received. A constant congestion window means that one new packet is transmitted for each ack that is received. The sending rate is controlled indirectly by adjusting the congestion window. In acknowledgements and loss detection, at the receiving end, acknowledgement packets are sent in response to received data packets. Tcp uses cumulative acknowledgements: Each acknowledgement includes a sequence number that says that all packets up to that one have been received. Equivalently, the acknowledgement identifies the next packet that the receiver expects to see. When packets are received out of order, each received packet results in an acknowledgement, but they will identify the largest sequence number such that all packets up to that number have been received.

Packet losses are detected by the sender in two ways:

- Timeout. If a packet is transmitted and no ack for that packet is received within the Retransmission Timeout interval (rto), the packet is considered lost.
- Fast retransmit. If three duplicate acks are received, the "next expected packet" from these acks is considered lost. Note that this cannot happen if the congestion window is smaller than four packets.

Packets that are lost, as detected by either of these mechanisms, are retransmitted.

In TCP congestion control state, there are four distinctive states in the tcp congestion control; the four congestion control states are following: slow start, congestion avoidance, fast retransmit and fast recovery.

II. DEFINITIONS

This section provides the definition of several terms that will be used in TCP congestion control over wireless networks.

SEGMENT: A segment is any TCP/IP data or acknowledgment packet (or both).
SENDER MAXIMUM SEGMENT SIZE (SMSS): The SMSS is the size of the largest segment that the sender can transmit. This value can be based on the maximum transmission unit of the network, the path MTU discovery algorithm, RMSS (see next item), or other factors. The size does not include the TCP/IP headers and options.
RECEIVER MAXIMUM SEGMENT SIZE (RMSS): The RMSS is the size of the largest segment the receiver is willing to accept. This is the value specified in the MSS option sent by the receiver during connection startup or, if the MSS option is not used, 536 bytes. The size does not include the TCP/IP headers and options.
FULL-SIZED SEGMENT: A segment that contains the maximum number of data bytes permitted (i.e., a segment containing SMSS bytes of data).
RECEIVER WINDOW (rwnd): The most recently advertised receiver window.
CONGESTION WINDOW (cwnd): A TCP state variable that limits the amount of data a TCP can send. At any given time, a TCP MUST NOT send data with a sequence number higher than the sum of the highest acknowledged sequence number and the minimum of cwnd and rwnd.
INITIAL WINDOW (IW): The initial window is the size of

the sender's congestion window after the three-way handshake is completed.
LOSS WINDOW (LW): The loss window is the size of the congestion window after a TCP sender detects loss using its retransmission timer.
RESTART WINDOW (RW): The restart window is the size of the congestion window after a TCP restarts transmission after an idle.
FLIGHT SIZE: The amount of data that has been sent but not yet acknowledged.
DUPLICATE ACKNOWLEDGMENT: An acknowledgment is considered a "duplicate" in the following algorithms when (a) the receiver of the ACK has outstanding data, (b) the incoming acknowledgment carries no data, (c) the SYN and FIN bits are both off, (d) the acknowledgment number is equal to the greatest acknowledgment received on the given connection and (e) the advertised window in the incoming acknowledgment equals the advertised window in the last incoming acknowledgment. Alternatively, a TCP that utilizes selective acknowledgments can determine an incoming ACK is a "duplicate" if the ACK contains previously unknown SACK information.ref [8]

III CONGESTION CONTROL MECHANISM

There are four distinctive states in the tcp congestion control, and two state variables related to congestion control: The congestion window cwnd and the slow start threshold ssthresh. The value of ssthresh determines the window increase rule; if cwnd < ssthresh, tcp increases cwnd by one packet for each received ack (slow start state), and if cwnd = ssthresh, tcp increases the window size by one packet per rtt (congestion avoidance state). Typical initial values when tcp leaves the idle state and enters the slow start state are a cwnd of 2 packets, and an infinite ssthresh.
We look at the mechanism of each of the four states in turn

A. SLOW START: The slow start state is the first state entered when a flow is created, or when a flow is reactivated after being idle. The slow start state can also be entered as the result of a timeout. In this state, cwnd is increased by one packet for each non-duplicate ack. The effect is that for each received ack, two new packets are transmitted. This implies that the congestion window, and also the sending rate, increases exponentially, doubling once per rtt. It may seem strange to refer to an exponential increase of the sending rate as "Slow start"; the reason is that in the early days, tcp used a large window from the start, and the introduction of the slow start mechanism did slow down connection startup. Slow start continues until either

- cwnd > ssthresh, in which case tcp enters the congestion avoidance state, or
- A timeout occurs, in which case tcp enters the fast retransmit state, or
- Three duplicate acks are received, in which case tcp enters the fast recovery state.

The motivation for the slow start state is that when a new flow enters the network, and there is a bottleneck link along the path, then the old flows sharing that link need some time to react and slow down before there is room for the new flow to send at full speed. Here in this slow start state we focus on delay based slow start.
With the goal of minimizing queuing delay and loss, slow-start poses a significant problem. Its exponential increase, where cwnd is doubled each RTT, can create large bursts of packets that in turn cause large delay spikes and many losses when cwnd eventually overshoots the bottleneck queue. We can apply the same delay back off threshold $\tau 0$ as used in the delay-based algorithm to trigger an exit from slow-start when queueing delay rises. This will keep the queue from filling. However, due to the bursty nature of slow-start, with associated spikes in queueing delay, it causes slow-start to exit sooner than it should. In order to continue a fast increase rate, but avoid the queuing spikes of slow-start, we propose a mechanism based on Limited Slow-Start. This uses a parameter, max_ssthresh, which controls the maximum bottleneck queue occupancy the flow will contribute due to congestion window increase. To achieve this it bounds the increase of cwnd to no more than max_ssthresh/2 per round-trip time. This algorithm meshes nicely with the delay-based algorithm, as they both seek to keep queue occupancy below a set point. A difficulty in using limited slow-start is selecting an appropriate max ssthresh. When used in conjunction with delay-based, max ssthresh should be set so that bottleneck queue occupancy of max ssthresh corresponds to a queue delay of $\tau 0$. To find this conversion factor, we use the initial slow-start phase to estimate the bottleneck rate. An ack-clocked slow-start sends at twice the rate of the ack clock, up to twice the bottleneck rate. This results in transient queue increases of size cwnd (t)/2 where cwnd (t) is the congestion window at time t. The drain time of this queue can be observed by the returning ACKs. When RTT > RTTmax, the queue's drain time can be estimated as RTTmax − RTTmin. The queue's size is cwnd (t − RTT)/2, since the ACKs are received one RTT after the data is sent, or cwnd (t)/4, since the window doubles each RTT. This estimation of both queue lengths over drain time gives us bottleneck rate we need to convert $\tau 0$ to Max_ssthresh. Specifically, when RTT > RTTmax

$$\text{Max_ssthresh} \leftarrow \text{cwnd}/4\ (\tau 0/\ (\text{RTTmax} - \text{RTT min}))\ldots\ (1)$$

A significant advantage of using the parameter $\tau 0$ (units of time) over max ssthresh (units of bytes) is that the behavior becomes scale independent of data rate. Limited slow-start takes O (cwnd/max_ssthresh) RTTs to reach a given window, but this delay-based version takes O (RT T/ $\tau 0$) RTTs.

We note that we have some freedom in the selection of the delay threshold $\tau 0$. We exploit this and choose $\tau 0$to be proportional to the recent level of queue occupancy. In this way, the threshold automatically adjusts upwards when loss based flows are present that fill the queue, thus enabling delay

based flows to increase their congestion window. Specifically, we choose $\tau 0$according to

$$\tau 0 = (1 - \Upsilon)\, \bar{\tau} 0 + \Upsilon\, (RTTmax - RTTmin) \quad (2)$$

With $0 < \Upsilon < 1$ and where RTTmax is a quantity that tracks the maximum observed RTT and decays towards RTTmin during periods when the current RTT is below RTTmax. RTTmax − RTTmin is an estimate of the recent queue occupancy and $\tau 0$ is selected to be a convex combination of the baseline value $\bar{\tau} 0$ and RTTmax − RTTmin. When Υ= 0we recover the previous delay-based AIMD algorithm. When Y> 0, delay-based flows are able toincrease their congestion window even when the network queues are persistently backlogged due to the action of loss-based flows.

We modify the delay-based algorithm to (i) perform additive increase of the congestion window only when the queueing delay is below threshold $\tau 0$ – the effect is to disable the AIMD probing action once the queueing delay rises above $\tau 0$, and (ii) to multiplicatively decrease the congestion window when the queueing delay is at or above $\tau 0$ and the time since the last backoff is greater than a threshold $\tau 0$ –the effect is to limit the number of delay induced (as opposed to loss induced) backoffs in a given time interval. Ref [1]

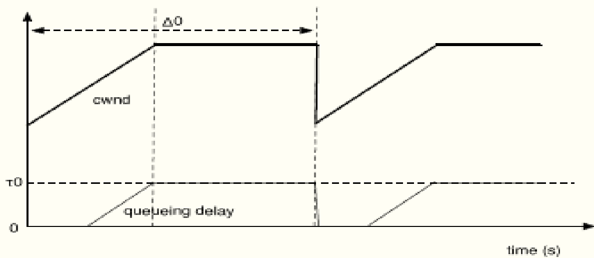
Combining these changes modified algorithm: on each ACK

Cwnd = cwnd + α/cwnd, if cwnd <=w0 or $\tau < \tau 0$ or

βcwnd, if cwnd > w0 & $\tau <= \tau 0$ & $\phi > \phi 0$ or..(3)

βcwnd, if packet loss

Where ϕ is the elapsed time since the last backoff. One option is to take $\phi 0$ to be proportional to RTTmax−RTTmin so that we recover the original delay-based algorithm when the queue backlog remains low and RTTmax − RTTmin small. Ref [7]



(Congestion window time history using algorithm) ref [1]

Fig: 1

1: On each ACK:
2: RTTmin = min (RTT, RTTmin)
3: RTTmax = max (RTT, RTTmax − a × RTT/cwnd)
4: τ = RTT − RTTmin // estimate queueing delay
5: β = min (δRTTmin/RTT, 0.9)
6: **if** cwnd <= ssthresh **then**
7: **if** cwnd <=ssthresh **then**
8: **if** τ = ˆTmax **then**
9: max ssthresh = (cwnd/4) × $\tau 0$/ (RTTmax − RTTmin)
10: **end if**
11: **if** cwnd <= max_ssthresh **then**
12: cwnd+ = MSS
13: **else**
14: cwnd+ = max_ssthresh/ (2 × cwnd)
15: **end if**
16: **end if**
17: **else**
18: **if** cwnd <= w0 $\tau 0 = (1 - Y)\, \bar{\tau} 0 + Y\, (RTTmax - RTTmin)$ **then**
19: cwnd+ = 2(1 − β) α (ϕ)/cwnd

20: **end if**
21: **end if**
22: **if** cwnd > w0 & $\tau >= \bar{\tau} 0$ & now − time of last backoff > $\phi 0$ **then**
23: cwnd = β × cwnd
24: ssthresh = cwnd
25: time of last backoff = now
26: **end if**

(Complete delay-based algorithm with limited slow-start)

B. CONGESTION AVOIDANCE

In the congestion avoidance state, cwnd is increased by one packet per rtt (if cwnd reaches the maximum value, it stays there). This corresponds to a linear increase in the sending rate. On timeout, tcp enters the exponential back-off state, and on three duplicate acks, it enters the fast recovery state. The motivation for this congestion avoidance mechanism is that since tcp does not know the available capacity, it has to probe the network to see at how high a rate data can get through. Ref [7] the basic guidelines for incrementing cwnd during congestion avoidance are:

* MAY increment cwnd by SMSS bytes
* SHOULD increment cwnd as par this eq

Cwnd += min (N, SMSS) (4)

Where N is the number of previously unacknowledged bytes acknowledged in the incoming ACK. ref [7]

Another common formula that a TCP MAY use to update cwnd during congestion avoidance is given in this equation:

cwnd += SMSS*SMSS/cwnd (5)

This adjustment is executed on every incoming ACK that acknowledges new data. This equation provides an acceptable approximation to the underlying principle of increasing cwnd by 1 full-sized segment per RTT.

When a TCP sender detects segment loss using the retransmission timer and the given segment has not yet been retransmitted, the value of ssthresh MUST be set to no more than the value given in equation 5:

ssthresh = max (Flight Size / 2, 2*SMSS) (6)

Where, as discussed above, Flight Size is the amount of outstanding data in the network. Ref [7]

On the other hand, when a TCP sender detects segment loss using the retransmission timer and the given segment has already been retransmitted at least once, the value of ssthresh is held constant.

C. FAST RETRANSMIT/FAST RECOVERY

A TCP receiver SHOULD send an immediate duplicate ACK when an out-of-order segment arrives. The purpose of this ACK is to inform the sender that a segment was received out-of-order and which sequence number is expected. From the sender's perspective, duplicate ACKs can be caused by a number of network problems. First, they can beaused by dropped segments. In this case, all segments after the dropped segment will trigger duplicate ACKs until the loss is repaired. Second, duplicate ACKs can be caused by the re-ordering of data segments by the network. Finally, duplicate ACKs can be caused by replication of ACK or data segments by the network. In addition, a TCP receiver SHOULD send an immediate ACK when the incoming segment fills in all or part of a gap in the sequence space. This will generate more timely information for a sender recovering from a loss through a retransmission timeout, a fast retransmit, or an advanced loss recovery algorithm,

The TCP sender SHOULD use the "fast retransmit" algorithm to detect and repair loss, based on incoming duplicate ACKs. The fast retransmit algorithm uses the arrival of 3 duplicate ACKs as an indication that a segment has been lost. After receiving 3 duplicate ACKs, TCP performs a retransmission of what appears to be the missing segment, without waiting for the retransmission timer to expire.

After the fast retransmit algorithm sends what appears to be the missing segment, the "fast recovery" algorithm governs the transmission of new data until a non-duplicate ACK arrives. The reason for not performing slow start is that the receipt of the duplicate ACKs not only indicates that a segment has been lost, but also that segments are most likely leaving the network (although a massive segment duplication by the network can invalidate this conclusion). In other words, since the receiver can only generate a duplicate ACK when a segment has arrived, that segment has left the network and is in the receiver's buffer, so we know it is no longer consuming network resources. Furthermore, since the ACK "clock"

is preserved, the TCP sender can continue to transmit new segments.

. TCP enters the fast retransmit mode after timeout. Several actions are taken when entering this state:

• The lost packet is retransmitted.

• The state variables are updated by ssthresh. Cwnd/2, cwnd. 1 packet.

• The rto value is doubled.

When an ack for the retransmitted packet is received, tcp enters the slow start phase. The above update of ssthresh implies that, in the absence of further packet losses, tcp will switch from the slow start phase to the congestion avoidance phase once cwnd is increased to half its value before the timeout. If the retransmission timer expires again with no ack for the retransmitted packet, the packet is repeatedly retransmitted, rto is doubled, and ssthresh is set to 1 packet. The upper bound for the rto is on the order of one or a few minutes. Exponential back-off continues until an acknowledgement for the packet is received, in which case tcp enters the slow start phase, or the tcp stack or application gives up and closes the connection. The motivation for the exponential back-off mechanism is that timeouts, in particular repeated timeouts, are a sign of severe network congestion. In order to avoid congestion collapse, the load on the network must be decreased considerably and repeatedly, until it reaches a level with a reasonably small packet loss probability. Ref [5]

TCP enters the fast recovery state after it detects three duplicate acks. When entering this mode, the first action of tcp is to retransmit the lost packet, and set ssthresh ← cwnd/2. The reason for the ssthresh update is arrange so that the later window increase from cwnd/2 and up will use the additive increase of congestion avoidance, not slow start tcp then continues to send new data at approximately the same rate, one new packet of data for each received duplicate ack. In rfc [6], this is described using a fairly complex procedure that artificially inflates cwnd. If no ack for the retransmitted packet is received within the rto interval, tcp enters the exponential back-off state. Otherwise, when an ack for the retransmitted packet is finally received, tcp sets cwnd = ssthresh, i.e., half the cwnd value at the start of the recovery procedure, and enters the congestion avoidance state. If more than one packet is lost within the same window, the original fast recovery procedure of tcp Reno is limited in that it can recover only one packet per rtt. This is the main problem addressed by both tcp New Reno and tcp sack. The motivation for the fast recovery mechanism is that the reception of duplicate acks indicates that the network is able to deliver new data to the receiver. Hence, the network is not severely congested, and we can keep inserting new packets into the network at the same rate as packets are delivered, at least for a while. On the other hand, the loss of a packet also indicates that the network is on the border of congestion. At the end of the fast recovery procedure, cwnd is halved. Tcp restarts the probing of the congestion avoidance state at a lower sending rate, at which it did not get any losses. It should also be noted that halving the cwnd also implies that tcp will stay silent for about half an rtt, waiting for acks that reduce the number of outstanding packets, until the actual number of outstanding packets match the new window size.

VI CAPACITY MEASUREMENT FOR WIRELESS

The ideal case for the source node is to have capacity information of the entire data communication path for the entire duration of the connection. In order to approach to this ideal case, many proposals are targeting an enhancement of transport layer protocols through capacity probing techniques. The entire capacity of communication path consists of two components: bandwidth and delay.

Delay could be obtained analyzing delay between packet transmission and its corresponding ACK reception. However, such way of estimation performs poor under special circumstances [8]. More accurate RTT estimation is based on transmission timestamp propagation over end-to-end data path which is included into high performance extension and finer RTT estimation for TCP protocol [8]. The second component of link capacity is bandwidth. Different bandwidth estimation solutions available in literature can be logically divided into passive measurements and active probing groups; according to the algorithm they employ [8]. Passive solutions build their measurements based on the trace history of an existing data transmission. Such solutions are limited to the network paths that have recently been used for communication. On the other hand, active probing provides faster and more accurate estimations, while having a potentiality for exploration of the whole network. Capacity measurement techniques on transport layer have obvious drawbacks which prevent their successful deployment: 1) Probing network bandwidth requires an insertion of additional traffic, which reduces the already limited network resources; 2) The bandwidth information becomes available to sender after the time required for a roundtrip propagation of the probe sequence over the network path; 3) Additional computation resources are required from the sender for statistical processing of the measured data.

A. BANDWIDTH MEASUREMENT

Basic medium access mechanism specified by IEEE 802.11 standard is the CSMA/CA with binary exponential backoff: the node is not allowed to transmit until the medium becomes free (i.e. no pending transmissions of other nodes). As a result, the whole bandwidth is divided among nodes which share the same medium. The available bandwidth B for the transmission of a certain amount of data can be obtained knowing the size of the data D and time T taken for data transmission over a specific link:

$$B = D/T \qquad (7)$$

Having a data to send at time *Tin*, the source node initiates the medium access procedure: it senses that medium is already occupied by another transmission and falls into exponential backoff with the initial size of the backoff window; during the next time of sensing the medium appears to be free, which means that the source node is allowed to initiate the transmission with RTS frame for medium reservation. At time *Tout*, the sender initiates data frame transmission onto the physical medium. Upon the successful reception of the data frame, the destination node replies with positive acknowledgement. Taking into account the described data delivery framework, the time required for transmission of a single data packet using CSMA/CA can be obtained as follows:

$$T = Tout - Tin + Ttr \qquad (8)$$

Where the difference *Tout - Tin* includes data queuing delay corresponding to the time the node was waiting for all other nodes to finalize their pending transmissions as well as channel access delay using random backoff and optional RTS/CTS exchange. A similar analysis was presented by Bianchi in his theoretical model for the performance analysis of 802.11 Ref [8]. In order to calculate the time *Ttr* required for data frame transmission and its corresponding acknowledgement, it is necessary to take into account the framework employed by the physical layer. Physical Layer Convergence Protocol (PLCP) preamble is transmitted prior any communication between nodes, for the synchronization of their physical circuits. PLCP Header contains signaling information about the subsequent MAC layer frame, such as length and bit rate. PLCP preamble and header are always transmitted at the basic rate, regardless of the maximum available bit rate on the medium. The MAC header, being transmitted at the data rate specified in the PLCP header, contains information about the delivered data on the link layer. FCS field finalizes frame containing CRC information related to both MAC header and frame body. According to physical and link layer specifications, the time required for data packet delivery (including the data frame and the corresponding ACK) is calculated as follows:

Ttr = Tdata + SIFS + Tack + DIFS

Tdata = [{(PLCP.Preamble + PLCP Header) / Basic. rate} + {(MAC. Header + FCS)/Data. Rate} + (Data / Data.Rate)]

Tack = [{(PLCP.Premable + PLCP Header) / Basic. Rate} + {(ACK.Header +FCS)/Data.Rate}]

The time required for a single data frame transmission Tdata includes the term which corresponds to physical preamble and header. This means that the value of the first term can be calculated once and reused for subsequent calculations Ref [8].

The algorithm for bandwidth measurement is the following:

1. Store the timestamp *Tin* for every data arrived to the link layer for further transmission. The data can arrive for forwarding from other nodes or it can be generated locally by upper layers of the protocol stack.
2. Prior actual data frame transmission, at time *Tout* calculate the time taken for packet delivery including queuing and packet transmission time *Ttr*.

3. Calculate the bandwidth experienced by the packet using (7).

B. DELAY ESTIMATION

1) Forward path delay. According to the considerations described above, the end-to-end delay experienced by a data burst of size *Data.Size* over *n*-hop path with a data rate specified by *Data.Rate* can be calculated as follows:

$$D_{end\ to\ end} = \sum_{i=1}^{n} Data.size/Data.Rate_i$$

Delay calculation does not include the queuing delay, the time required to gain medium access and the delay associated with physical and link layer header transmission. Such parameters are included in the bandwidth calculations in previous section. For that reason, the entire overhead which occurs before actual data transmission over the physical medium is considered to be a factor which reduces the available bandwidth on the link. The overhead added at physical and a link layer is directly related to the utilization level. For example, utilization of wired local area networks is relatively high (97%) while the utilization of wireless 802.11 networks is on low level Ref [5].

2) Backward path delay. TCP reliability is achieved through implementation of a positive acknowledgement scheme. The receiver acknowledges successful data delivery by TCP ACK packet going to back to sender. On contrary with packet delay measurement technique described above, TCP ACK delay does include both transmission and queuing delays, i.e. it is equal to the difference between the time the TCP ACK packet is generated by the receiver node and its reception by the TCP sender. Backward path delay could be calculated using two ways: the delay calculation technique presented in (8) or by simple subtraction of forward path delay from RTT value.
The presented way of delay measurements in forward and backward directions allows TCP sender to adjust the amount of outstanding data to the bandwidth-delay product in the forward path while considering the backward path as a simple delay line for the TCP acknowledgement reception.Ref [5].

CONCLUSION

In this paper, for preventing congestion control over wireless networks we propose a limited slow start mechanism in TCP window size for enhancement of throughput and traffic shaping for loss delay and wireless links. Able to obtain more precise results by gathering capacity information such as bandwidth and delay at the link layer. An additional module is inserted to the protocol stack of the node which adjusts the outgoing data stream based on the capacity measurements we are proposed the behavior over wireless link, characterized by capacity, delay, Bandwidth and non-congested packet losses.

REFERENCES

[1] D. Leith1, R.Shorten1and G.McCullagh Delay-based AIMD congestion control in Hamilton Institute. Pittsburgh Supercomputing Center. Cisco Systems.

[2] V. Jacobson, Modified *TCP congestion avoidance algorithm*. Note sent to end2end-interest mailing list, 1990.

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

[4] Rajesh Roy, Sudipto Das, Modified TCP Congestion Control Algorithm for Throughput Enhancement in Wired-cum-Wireless Networks, 2006

[5] NIELS M¨ OLLER, Window-based congestion control, Doctoral Thesis Stockholm, Sweden, 2008

[6] Kazumi KANEKO, Tomoki FUJIKAWA, Zhou SU and Jiro KATTO, A Hybrid Congestion Control Algorithm for High-speed Networks, Graduate School of Science and Engineering, Waseda University,2007

[7] M. Allman, V. Paxson, TCP Congestion Control, Purdue University September 2007

[8] Dzmitry Kliazovich and Fabrizio Granelli, Cross-layer congestion control in Multi hop wireless networks, WICON, 05

[9] *The ns-2 network simulator*, June 2007, http://www.isi.edu/nsnam/ ns/.