ACCESS METHODS FOR SOCIAL INCLUSION

A thesis submitted for the partial fulfillment of requirements for the award of the degree of

B.Tech

In

Computer Science and Engineering

By

K. Adithya (106108004)

Sagar Sumit (106108071)

Venkataraman Balasubramanian (106108072)

COMPUTER SCIENCE AND ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY

TIRUCHIRAPPALLI-620015

MAY 2012

BONAFIDE CERTIFICATE

This is to certify that the project titled **ACCESS METHODS FOR SOCIAL INCLUSION** is a bonafide record of the work done by

K. Adithya (106108004)
Sagar Sumit (106108071)
Venkataraman Balasubramanian (106108072)

in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering** of the **NATIONAL INSTITUTE OF TECHNOLOGY, TIRUCHIRAPPALLI,** during the year 20011-2012.

NAME
Guide

NAME
Head of the Department

Project Viva-voce held on 9/5/12

Internal Examiner

External Examiner

ABSTRACT

Man has decided to take it upon himself, to make sure that each and every human-being has been provided with equal opportunity. We have foreseen a problem yet to arise. Bridging the Packet-gap between the poor and the rich. Poor sections of the society who lack an internet connection should be able to share the internet facility belonging to a richer community nearby for free. However, the real challenge is to make this privilege without denting the bandwidth allocated for the owner of the connection. We analyze exiting congestion control mechanisms like TCP-Cubic, TCP-Newreno, TCP-Vegas, TCP-Sack etc and we see fit in a new experimental protocol called as LEDBAT (Low Extra Delay Background Transport) [1], a delay based congestion control mechanism, which is extensively used in torrent applications (esp. by Bittorrent inc.) to solve problems with similar goals. Currently, LEDBAT version 00 has been implemented [2]. Our initial work involves, updating this work to the latest version viz. LEDBAT version 09 and also developing a Linux kernel module for the same. We have implemented the latest version in a well-known network simulator NS2, by performing extensive experiments. A Linux Kernel module has also been implemented and is proved to work in a real-test-field. In order to make sure, only a genuine poor user participates, we make sure the user implements only the LEDBAT congestion control mechanism. For this, we have fingerprinted the LEDBAT traffic based on utilization and used to classify flows and hence detect Non-LEDBAT (malicious) users. The outcome of this project has provided a concrete proof that a particular protocol (LEDBAT) maybe implemented, by providing graphs and metrics to support our view, and to provide a general working model of the same for a real-world implementation.

Keywords: Social-Inclusion; TCP Congestion Control; LEDBAT; Linux kernel Module

ACKNOWLEGEMENTS

First I would like to express my immense gratitude to Dr. (Mrs).R.Leela Velusamy for providing us with an opportunity to understand the joy of research and introduce us to the subject and develop a keen eye towards research.

We thank Dr. Arjuna Sathiaseelan, Research Fellow at University of Aberdeen, UK who introduced us to the problem, and provided us valuable guidance in our work.

We would also like to place on record our sincere respects to Dr. C.Mala, Head of the Department, Computer Science and Engineering, National Institute of Technology, Trichy who provided us with the necessary environment and tools for the implementation of the project.

We thank our reviewers Ms. B. Nithya and Sri. E. Sivasankar, who provided their timely review for our work, and guided us along our

Finally we also like to thank Stephen Vaithiya, Venkanna Udutalapally and all other research scholars in the PhD Lab of Computer Science and Engineering department, without whose useful guidance it wouldn't be possible to complete this project on time.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

ACK	Acknowledgment
AQM	Active Queue Management
CTO	Congestion Timeout Value
CWND	Congestion Window
ECN	Explicit Congestion Notification
FIFO	First In First Out
FTP	File Transfer Protocol
LEDBAT	Low Extra Delay Background Transport
LFN	Long Fat Networks
LKM	Linux Kernel Module
MAC	Medium Access Control
MSS	Maximum Segment Size
QoS	Quality of Service
RTT	Round Trip Time
SSTHRESH	Slow-start Threshold
TCP	Transmission Control Protocol
TCP-LP	TCP-Low Priority
TCP-SACK	TCP-Selective ACKnowledgement
UDP	User Datagram Protocol

CHAPTER 1

INTRODUCTION

1.1 MOTIVATION

Technology has played the most dominant role in what we, as humans think, speak and act today. The escalation of technological assets in terms of both length and breadth has been truly breathtaking, especially in connecting people the way it does today, which used to be only a mere ambition a few decades back. However, particularly in developing parts of the world like India, Indonesia etc, the rift between rich and the poor, a resultant of income inequality has only made the financially blessed communities to enjoy the privilege of Internet connection. Those who are under the segregation of the poor community as well as those who are predominantly in the even more unfortunate middle-class section are often left out of the privilege of getting their hands on Internet even though they might be literate.

Government officials of all countries find it a challenging and uphill task to break even with their targets set on providing the bare basic needs of their people. Having said that, those absent of financial resources for an Internet connection cannot really hope and expect assistance from the government since there is no reason for the policy makers to prioritize promising Internet connection over other pressing issues yet to be addressed. Even though the Indian government has triggered a technological surge among the middle class societies by bringing in 'AAKASH' tablets [18] which are available at meagre prices, the usage of such devices can only be given justice if Internet service is at reach. Otherwise, even AAKASH tablets are going to serve best only to the richer communities which only creates a 'digital divide' among the masses. There is a definite need to lend a helping hand to the less privileged people and incorporate them into the society to be connected as well. What if in a neighborhood, a blend of rich and poor civilians exist and there can be arrangements made for poor users to share the Internet connection from a next door rich user for free, without impeding the rich users' bandwidth even to the slightest measure? We identify that this as a feasible and practical approach for pulling the 'unconnected' people off their shell into the Internet world. When government cannot be relied on for connecting poor people to the rest of the world through Internet and those financially fortunate can lend a helping hand to a few of their unprivileged brothers

around them without themselves being dented in any way, we think it's a sure recipe for Social Inclusion of the poor to the Internet world.

1.2 PROBLEM DEFINITION

To study the existing access methods for creating a system of sharing of network resources (Internet connection) and to study the impact of implementing the access methods and to create a suitable working model for the same

The following diagram describes the base-model that we have considered to setup, simulate, analyze, evaluate and derive conclusions about our problem.

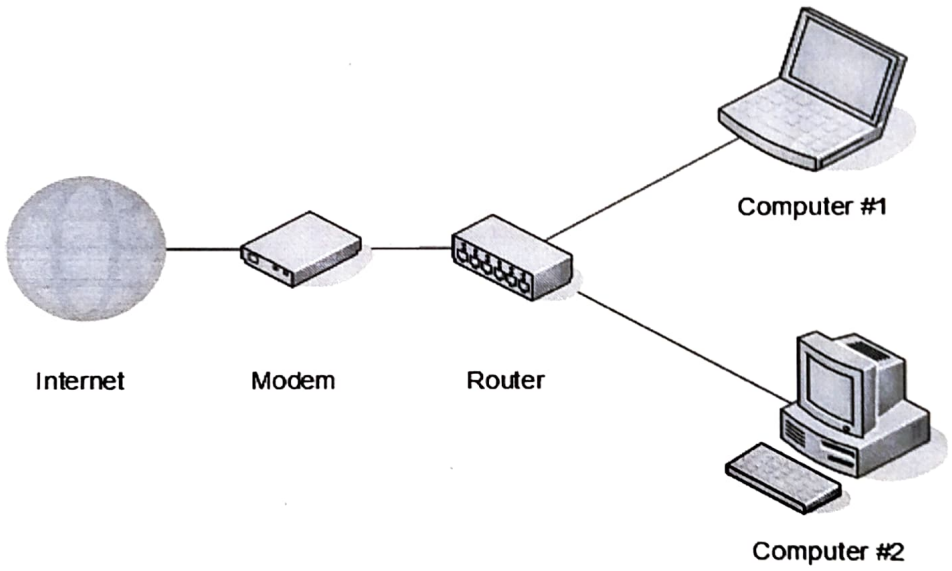

Fig. 1.1 : Base-model considered for implementation

A central router, owned by a rich user(Computer #1) who owns the internet connection. The poor user (Computer #2) who uses the Internet connection of the rich user.

The Goals are:

1) To utilize the free bandwidth of the owner of the internet connection (Computer #1)

2) To make sure, the QoS parameters of Computer #1 are not affected ie, he is unaware of the presence of another flow.

Our task is to choose the best congestion control algorithm to be executed on the poor user's computer such that the poor user is given low priority traffic and the performance of the rich user's bandwidth stays untouched with the facility of high priority traffic. We have analyzed a few existing congestion control algorithms namely, TCP-Reno, TCP-Newreno, TCP-Vegas, TCP-Sack etc by simulating them in tandem with a standard TCP-Cubic connection. TCP-Cubic connection is chosen as the base connection as it is the default mechanism in standard Linux distributions.

The next logical step is to come up with a finger printing mechanism to identify those malicious users who cover themselves with identity of a 'poor user' but draw bandwidth above the threshold of a genuine poor user. We detect those malicious users, and prevent them hogging into the bandwidth of the genuine users.

CHAPTER 2

LITERATURE REVIEW

In 2.1 we review the existing congestion control algorithms, their functionality and the exact reason they were developed for. In 2.2 we concentrate only on LEDBAT. Its development, current work and the evolution of the mechanism itself.

2.1 TCP CONGESTION CONTROL ALGORITHMS

There are fundamentally two different kinds of Congestion Control Algorithms namely Delay based and Non-delay based

. While delay based algorithms are designed based on the Round Trip Time (RTT) measure of the packets in a congestion window, non-delay based algorithms are more flexible and are more flexible in general. They adjust the congestion window according to the flow of traffic and are not confined to just the manipulation of Round Trip Time like the way delay based algorithms work

While TCP-Reno, TCP-Newreno, TCP-Sack, TCP-Cubic don't fall under delay based algorithms, they are initial algorithms developed along with TCP using its usual Congestion avoidance mechanisms.

2.1.1 TCP-Reno

To avoid congestion collapse, TCP uses a multi-faceted congestion control strategy. For each connection, TCP maintains a congestion window, limiting the total number of unacknowledged packets that may be in transit end-to-end. This is somewhat analogous to TCP's sliding window used for flow control. TCP uses a mechanism to increase the congestion window after a connection is initialized and after a timeout. It starts with a window of two times the maximum segment size (MSS). Although the initial rate is low, the rate of increase is very rapid: for every packet acknowledged, the congestion window increases by 1 MSS so that the congestion window effectively doubles for every round trip time (RTT). When the congestion window exceeds a threshold ssthresh the algorithm enters a new state, called congestion avoidance. In some implementations (e.g., Linux), the initial ssthresh is large, and so the first slow start usually ends after a loss. However, ssthresh is updated at the end of each slow start, and will often affect subsequent slow starts triggered by timeouts.

Congestion avoidance: As long as non-duplicate ACKs are received, the congestion window is additively increased by one MSS every round trip time. When a packet is lost, the likelihood of duplicate ACKs being received is very high (it's possible though unlikely that the stream just underwent extreme packet reordering, which would also prompt duplicate ACKs).

In TCP-Reno If three duplicate ACKs are received (i.e., four ACKs acknowledging the same packet, which are not piggybacked on data, and do not change the receiver's advertised window), Reno will halve the congestion window, perform a fast retransmit, and enter a phase called Fast Recovery. In this state, TCP retransmits the missing packet that was signaled by three duplicate ACKs, and waits for an acknowledgment of the entire transmit window before returning to congestion avoidance. If there is no acknowledgment, TCP Reno experiences a timeout and enters the slow-start state.

2.1.2 TCP- New Reno

TCP New Reno[3], defined by improves retransmission during the fast recovery phase of TCP Reno. During fast recovery, for every duplicate ACK that is returned to TCP New Reno, a new unsent packet from the end of the congestion window is sent, to keep the transmit window full. For every ACK that makes partial progress in the sequence space, the sender assumes that the ACK points to a new hole, and the next packet beyond the ACKed sequence number is sent.

Because the timeout timer is reset whenever there is progress in the transmit buffer, this allows New Reno to fill large holes, or multiple holes, in the sequence space - much like TCP SACK. Because New Reno can send new packets at the end of the congestion window during fast recovery, high throughput is maintained during the hole-filling process, even when there are multiple holes, of multiple packets each. When TCP enters fast recovery it records the highest outstanding unacknowledged packet sequence number. When this sequence number is acknowledged, TCP returns to the congestion avoidance state.

A problem occurs with New Reno when there are no packet losses but instead, packets are reordered by more than 3 packet sequence numbers. When this happens, New Reno mistakenly enters fast recovery, but when the reordered packet is

delivered, ACK sequence-number progress occurs and from there until the end of fast recovery, every bit of sequence-number progress produces a duplicate and needless retransmission that is immediately ACKed.

New Reno performs as well as SACK at low packet error rates, and substantially outperforms Reno at high error rates.

2.1.3 TCP-Cubic

TCP Cubic[6] is an implementation of TCP with an optimized congestion control algorithm for high speed networks with high latency (LFN: Long Fat Networks).[19]

It is a less aggressive and more systematic derivative of BIC TCP, in which the window is a cubic function of time since the last congestion event, with the inflection point set to the window prior to the event. Being a cubic function, there are two components to window growth. The first is a concave portion where the window quickly ramps up to the window size before the last congestion event. Next is the convex growth where CUBIC probes for more bandwidth, slowly at first then very rapidly. CUBIC spends a lot of time at a plateau between the concave and convex growth region which allows the network to stabilize before CUBIC begins looking for more bandwidth. Another major difference between CUBIC and standard TCP flavors is that it does not rely on the receipt of ACKs to increase the window size. CUBIC's window size is dependent only on the last congestion event. With standard TCP, flows with very short RTTs will receive ACKs faster and therefore have their congestion windows grow faster than other flows with longer RTTs. CUBIC allows for more fairness between flows since the window growth is independent of RTT.

2.1.4 TCP-SACK

Protocols which provide reliable communication over such networks use a combination of acknowledgments (i.e. an explicit receipt from the destination of the data), retransmission of missing and/or damaged packets (usually initiated by a time-out), and checksums to provide that reliability.

Selective Acknowledgment (SACK)[2]: the receiver explicitly lists which packets, messages, or segments in a stream are acknowledged (either negatively or positively). In TCP SACK the selective acknowledgement used is a positive one. That is, it lists

only those packets, messages or streams which were acknowledged and ignores those which havenot been acknowledged.

Some popular delay based algorithms are: TCP-Vegas, TCP-LP, TCP-Nice and of course the Low Extra Delay Background Transport (LEDBAT)

2.1.5 TCP-Vegas

TCP Vegas[1] is one of the first protocols that was known to have a smaller sending rate han standard TCP when both protocols share a bottleneck yet it was designed to achieve more, not less throughput than standard TCP. Indeed, when it is the only protocol on the bottleneck, the throughput of TCP Vegas is greater than the throughput of standard TCP. Until the mid 1990s, all of TCP's set timeouts and measured round-trip delays were based upon only the last transmitted packet in the transmit buffer. University of Arizona researchers Larry Peterson and Lawrence Brakamo introduced TCP Vegas, in which timeouts were set and round-trip delays were measured for every packet in the transmit buffer. After extensive analysis it is found that the most prominent properties of TCP Vegas is its fairness between multiple flows of the same kind, which does not penalize flows with large propagation delays in the same way as standard TCP. While it was not the first protocol that uses delay as a congestion indication, its predecessors are not discussed here because of the historical "landmark" role that TCP Vegas has taken in the literature.

2.1.6 TCP-Nice

TCP Nice[4] follows the same basic approach as TCP Vegas but improves upon it in some aspects. Because of its moderate linear decrease congestion response, TCP Vegas can affect standard TCP despite its ability to detect congestion early. TCP Nice removes this issue by halving the congestion window (at most once per RTT, like standard TCP) instead of linearly reducing it. To avoid being too conservative, this is only done if a fixed predefined fraction of delay-based incipient congestion signals appears within one RTT. Otherwise, TCP Nice falls back to the congestion avoidance rules of TCP Vegas if no packet was lost or standard TCP if a packet was lost. One more feature of TCP Nice is its ability to support a congestion window of less than one packet, by clocking out single packets over more than one RTT.

2.1.7 TCP-LP

Other than TCP Vegas and TCP Nice, TCP-LP[5] uses only the one-way delay (OWD) instead of the RTT as an indicator of incipient congestion. This is done to avoid reacting to delay fluctuations that are caused by reverse cross-traffic. Using the TCP Time-stamps option the OWD is determined as the difference between the receiver's time-stamp value in the ACK and the original Time-stamp value that the receiver copied into the ACK. While the result of this subtraction can only precisely represent the OWD if clocks are synchronized, its absolute value is of no concern to TCP-LP and hence clock synchronization is unnecessary. Using a constant smoothing parameter, TCP-LP calculates an Exponentially Weighted Moving Average (EWMA) of the measured OWD and checks whether the result exceeds a threshold within the range of the minimum and maximum OWD that was seen during the connection's lifetime; if it does, this condition is interpreted as an "early congestion indication". The minimum and maximum OWD values are initialized during the slow-start phase. Regarding its reaction to an early congestion indication, TCP-LP tries to strike a middle ground between the overly conservative choice of immediately setting the congestion window to one packet and the presumably too aggressive choice of halving the congestion window like standard TCP. It does so by halving the window at first in response to an early congestion indication, then initializing an "interference time-out timer", and maintaining the window size until this timer fires. If another early congestion indication appeared during this "interference phase", the window is then set to 1; otherwise, the window is maintained and TCP-LP continues to increase it the standard Additive-Increase fashion. This method ensures that it takes at least two RTTs for a TCP-LP flow to decrease its window to 1, and, like standard TCP, TCP-LP reacts to congestion at most once per RTT. The below are Non-delay based Algorithms which are also called Lower than Best Effort(LBE) protocols. Some of the popular ones are: Competitive and Considerate Congestion Control(4CP), MulTCP, MulTFRC

2.1.8 Competitive and Considerate Congestion Control(4CP)

Competitive and Considerate Congestion Control [10], is a protocol which provides a LBE service by changing the window control rules of standard TCP. A "virtual window" is maintained, which, during a so-called "bad congestion phase" is reduced to less than a predefined minimum value of the actual congestion window. The congestion window is only increased again once the virtual window exceeds this

minimum, and in this way the virtual window controls the duration during which the sender transmits with a fixed minimum rate. The 4CP congestion avoidance algorithm allows for setting a target average window and avoids starvation of "background" flows while bounding the impact on"foreground" flows.

2.1.9 MulTCP

There needs to be a protocol which allows users to assign different priorities to different flows of traffic. The first, and best known, such protocol is MulTCP,[11] which emulates N TCPs in a rather simple fashion. An improved version of MulTCP is presented in, and there is also a variant where only one feedback loop is applied to control a larger traffic aggregate by the name of Probe-Aided (PA-)MulTCP. The general assumption underlying all of the above work is that these protocols are "N-TCP-friendly", i.e. they are as TCP-friendly as N TCPs, where N is a positive (and possibly natural) number which is greater than or equal to 1.

2.1.10 MulTFRC

The MulTFRC [11] protocol, another extension of TFRC for multiple flows, is however able to support values between 0 and 1, making it applicable as a mechanism for a LBE service. Since it does not react to delay like the mechanisms above but adjusts its rate like TFRC, it can probably be expected to be more aggressive than mechanisms such as TCP Nice or TCP-LP. This also means that MulTFRC is less likely to be prone to starvation, as its aggression is tunable at a fine granularity even when N is between 0 and 1.

2.2 LEDBAT CONGESTION CONTROL ALGORITHM

TCP congestion control seeks to share bandwidth at bottleneck link equitably among flows competing at the bottleneck, and it is the predominant congestion control mechanism used on the Internet. Not all applications seek an equitable share of network throughput, however "background" applications, such as software updates or file-sharing applications, seek to operate without interfering with the performance of more interactive and delay and/or bandwidth-sensitive "foreground" applications and standard TCP may be too aggressive for use with such background applications.[] LEDBAT is an experimental delay-based congestion control mechanism that reacts early to congestion in the network, thus enabling "background" applications to use the

network while avoidoing interference with the network performance of competing flows.[7][8]. A LEDBAT sender uses one-way delay measurements to estimate the amount of queuing on the data path, controls the LEDBAT flow's congestion window based on this estimate, and minimizes interference with competing flows when latency builds by adding low extra queuing delay on the end-to-end path.The main motivation for the development of such a lower than best effort protocol has been its contribution to bit-torrent applications that incorporate peer-to peer sharing.[] Moreover it has also been a popular choice for video and audio streaming with hybrid data distribution architecture.[20]

LEDBAT congestion control seeks to:[8]

1. utilize end-to-end available bandwidth, and maintain low queueing delay when no other traffic is present,
2. add little to the queuing delay induced by concurrent flows,
3. quickly yield to flows using standard TCP congestion control that share the same bottleneck link,

The methodology used by LEDBAT to achieve Congestion Control is [21]

• Saturate the bottleneck when no other traffic is present, but quickly yield to TCP and other UDP real-time traffic sharing the same bottleneck queue.

• Keep delay low when no other traffic is present, and add little to the queuing delays induced by TCP traffic.

• Operate well in drop-tail FIFO networks, but use explicit congestion notification (e.g., ECN) where available.

The very first implementation of the LEDBAT draft was developed by Silvio Valentti et al. as Version 0[7]. Since then it has gradually gone through revisions and is currently at LEDBAT Version 9 now. There are significant differences present between Version 0 and Version 9 which are as follows which are discussed in 3.1.However extensive research show that LEDBAT achieves some of its design objectives, but not without some challenges under certain conditions. Evaluation of LEDBAT performance in a controlled test-bed and Internet experiment and found that TCP traffic on the "unrelated" backward path is capable of causing LEDBAT to significantly under-utilize the link capacity in the forward path. It has been shown that LEDBAT competes fairly with TCP in the worst case (i.e. LEDBAT mis-configuration). [22]

CHAPTER 3

PROJECT APPROACH

We present a two-fold solution to our problem:

1) Getting a QoS solution using transport protocol using a congestion control algorithm called LEDBAT.
2) Providing an authentication mechanism where, LEDBAT traffic is fingerprinted.

LEDBAT is an experimental delay-based congestion control mechanism that's reacts early to congestion in the network, thus enabling "background" applications to use the network while avoiding interference with the network performance of competing flows. We have used ns-2 to initially test the algorithm, and then used a Linux kernel module to prove the correctness.

3.1 THE LEDBAT CONGESTION CONTROL ALGORITHM

A LEDBAT sender uses a congestion window (cwnd) to gate the amount of data that the sender can send into the network in one round-trip time (RTT). A sender maintains its cwnd in bytes. LEDBAT requires that each data segment carries a "timestamp" from the sender, based on which the receiver computes the one-way delay from the sender, and sends this computed value back to the sender. The receiver may send more than one delay sample in an acknowledgment. For instance, a receiver that delays acknowledgments, that is, sends an acknowledgment less frequently than once per data packet, may send all the one-way delay samples that it gathers in one acknowledgment.

Below is the pseudo code that extends the above LEDBAT algorithm [8]. "//" indicates a comment. Comments have been included in the pseudo code for better understanding. We have explained about various parameters and how to configure them in section 3.2.1. We have also briefly explained the functional modules involved in the kernel code in the same section.

1. Initialize the parameters to the appropriate value.
2. Define a structure for the one way delay (circular buffer as we can have multiple one-way delays at every minute interval) and initialize it.

3. Define a ledbat structure and initialize it.

4. Calculate ledbat_current_delay using filter().

5. Calculate ledbat_base_delay.

6. Set a flag to indicate slow start required or not.

7. Now do the following on the **receiver side**:

    7.a On data_packet:

    7.a.1 remote_timestamp = data_packet.timestamp

    7.a.2 acknowledgement.delay = local_timestamp() - remote_timestamp

    // fill in other fields of acknowledgement

    7.a.3 acknowledgement.send()

8. Do the following on the **sender side**:

    8.a On initialization:

    // cwnd is the amount of data that is allowed to send in one RTT and

    // is defined in bytes.

    // CTO is the Congestion Timeout value.

    8.a.1 Create current_delays list with CURRENT_FILTER elements

    8.a.2 Create base_delays list with BASE_HISTORY_LEN number of elements

    8.a.3 Initialize elements in base_delays to +INFINITY

    8.a.4 Initialize elements in current_delays appropriate to FILTER()

    8.a.5 last_rollover = -INFINITY # More than a minute in the past

    8.a.6 flightsize = 0

    8.a.7 cwnd = INIT_CWND * MSS

    8.a.8 CTO = 1 second

    8.b On acknowledgment:

    // flightsize is the amount of data outstanding before this ack

    // was received and is updated later;

    // bytes_newly_acked is the number of bytes that this ack

    // newly acknowledges, and it MAY be set to MSS.

    8.b.1 For each delay sample in the acknowledgment:

        delay = acknowledgement.delay

        update_base_delay(delay)

3. Define a ledbat structure and initialize it.

4. Calculate ledbat_current_delay using filter().

5. Calculate ledbat_base_delay.

6. Set a flag to indicate slow start required or not.

7. Now do the following on the **receiver side**:

7.a On data_packet:

7.a.1 remote_timestamp = data_packet.timestamp

7.a.2 acknowledgement.delay = local_timestamp() - remote_timestamp

// fill in other fields of acknowledgement

7.a.3 acknowledgement.send()

8. Do the following on the **sender side**:

8.a On initialization:

// cwnd is the amount of data that is allowed to send in one RTT and

// is defined in bytes.

// CTO is the Congestion Timeout value.

8.a.1 Create current_delays list with CURRENT_FILTER elements

8.a.2 Create base_delays list with BASE_HISTORY_LEN number of elements

8.a.3 Initialize elements in base_delays to +INFINITY

8.a.4 Initialize elements in current_delays appropriate to FILTER()

8.a.5 last_rollover = -INFINITY # More than a minute in the past

8.a.6 flightsize = 0

8.a.7 cwnd = INIT_CWND * MSS

8.a.8 CTO = 1 second

8.b On acknowledgment:

// flightsize is the amount of data outstanding before this ack

// was received and is updated later;

// bytes_newly_acked is the number of bytes that this ack

// newly acknowledges, and it MAY be set to MSS.

8.b.1 For each delay sample in the acknowledgment:

delay = acknowledgement.delay

update_base_delay(delay)

```
                update_current_delay(delay)
8.b.2       queuing_delay = FILTER(current_delays) - MIN(base_delays)
8.b.3       off_target = (TARGET - queuing_delay) / TARGET
8.b.4       cwnd += GAIN * off_target * bytes_newly_acked * MSS / cwnd
8.b.5       max_allowed_cwnd = flightsize + ALLOWED_INCREASE * MSS
8.b.6       cwnd = min(cwnd, max_allowed_cwnd)
8.b.7       cwnd = max(cwnd, MIN_CWND * MSS)
8.b.8       flightsize = flightsize - bytes_newly_acked
8.b.9       update_CTO()

8.c         On data loss:
// at most once per RTT
8.c.1       cwnd = min (cwnd, max (cwnd/2, MIN_CWND * MSS))
8.c.2       if data lost is not to be retransmitted:
                flightsize = flightsize - bytes_not_to_be_retransmitted

8.d         if no acks are received within a CTO:
// extreme congestion, or significant RTT change.
// set cwnd to 1MSS and backoff the congestion timer.
8.d.1       cwnd = 1 * MSS
8.d.2       CTO = 2 * CTO

8.e         update_CTO()
// implements an RTT estimation mechanism using data
// transmission times and ack reception times,
// which is used to implement a congestion timeout (CTO).
// If implementing LEDBAT in TCP, sender SHOULD use
// mechanisms described in RFC 6298 <xref target= 'RFC6298'/>,
// and the CTO would be the same as the RTO.

8.f         update_current_delay(delay)
// Maintain a list of CURRENT_FILTER last delays observed.
8.f.1       delete first item in current_delays list
8.f.2       append delay to current_delays list
```

```
8.g         update_base_delay(delay)
// Maintain BASE_HISTORY_LEN delay-minima.
// Each minimum is measured over a period of a minute.
// 'now' is the current system time
8.g.1       if round_to_minute(now) != round_to_minute(last_rollover)
                last_rollover = now
                delete first item in base_delays list
                append delay to base_delays list
            else
                base_delays.tail = MIN(base_delays.tail, delay)
```

9. END

3.2 LINUX KERNEL MODULE OF VERSION 9 OF LEDBAT

We have updated tcp-ledbat source code to version 9 of the draft[8]. We implemented a LEDBAT module in C programming language. The choice of C was primarily due to easier integration into the Linux Kernel. The module consists of two components, one for receiving messages and the other one for sending messages.

3.2.1 Ledbat Parameters And How To Configure Them:

The TCP-LEDBAT module has the following parameters:

1. TARGET is the maximum queuing delay that LEDBAT itself may introduce in the network. If the TARGET value is too low, it will lead to under-utilization of the bottleneck link while a very large value will tend to make LEDBAT compete with the other TCP flow. TARGET must be 100 milliseconds or less.
2. GAIN determines the rate at which the cwnd responds to changes in queuing delay. It must be set to 1 or less. A GAIN of 1 limits the maximum cwnd ramp-up to the same rate as TCP Reno in congestion avoidance. Initially, both numerator and denominator of GAIN is initialized to 1.
3. off_target is a normalized value representing the difference between the measured current queuing delay and the pre-calculated TARGET queuing delay. It can be positive or negative, and consequently, cwnd increases or decreases in proportion to off_target.

4. BASE_HISTORY_LEN is the size of the base delays list. A higher value may yield a more accurate measurement when the base delay is unchanging, and a lower value results in a quicker response to actual increase in base delay. A value for BASE_HISTORY_LEN is thus a trade-off. Ideally, it should be 10.
5. ALLOWED_INCREASE should be 1, and it must be greater than 0 [ref]. a value of 0 results in no growth of cwnd at all.
6. INIT_CWND and MIN_CWND should both be 2. An INIT_CWND of 2 should help seed FILTER() at the sender when there are no samples at the beginning of a flow, and a MIN_CWND of 2 allows FILTER() to use more than a single instantaneous delay estimate while not being too aggressive [ref].
7. CURRENT_FILTER is the size of the list which maintains only delay measurements made within a RTT amount of time in the past, seeking to eliminate noise spikes in its measurement of the current one-way delay through the network. Its value will depend on the filter being employed but generally the number of delay samples in each RTT should be at least 4 [ref].
8. Noise_filter_len is the length of the noise_filter vector used in FILTER() function and is by default 4.
9. Do_ss specifies whether to perform slow start: default = 0. A value of 0 signifies do not do slow start while a value of 1 means do slow start setting initial threshold to infinity. A value of 2 means to do a custom slow start with a threshold of ledbat_ssthresh to switch to congestion avoidance, that is, prior to avoiding a loss. The default value of ledbat_ssthresh is 65000 (0xffff).

The LEDBAT sender module seeks to extract the actual delay elements from the current delay samples by implementing FILTER() to eliminate any outliers. "update_current_delay()" maintains a list of one-way delay measurements, of which a filtered value is used as an estimate of the current end-to-end delay. It adds delay to noise_filter and updates the current delay. "update_base_delay()" maintains a list of one-way delay minima over a number of CTO intervals, to measure and to track changes in the base delay of the end-to-end path. "ledbat_init_circbuf()" initializes the one-way delay structure and "tcp_ledbat_init()" does the same for the ledbat structure, whilst, the "tcp_ledbat_release()" frees them when we exit and unregister from tcp-ledbat. "tcp_ledbat_ssthresh()" slow starts with a threshold of ledbat_ssthresh. "tcp_ledbat_owd_calculator()" calculates the one-way delay measurement and

validates it. Our main function in the receiver module is "tcp_ledbat_cong_avoid()" which is invoked whenever there is a congestion in the bottleneck link. It obtains the current_delay and the base_delay using the functions "ledbat_current_delay()" and "ledbat_base_delay()" respectively. Furthermore, it calculates the queue_delay and off_target using 8.b.2 and 8.b.3 respectively of the pseudo code presented in the section 3.1. Next it adjusts the congestion window in the steps 8.b.4 through 8.c.2 of the same pseudo code. "tcp_ledbat_rtt_sample()" again calculates the owd and adds the delay to noise_filter. If new minute commences, it adds the delay to base_delay or updates the last delay. In the next sub-section we shall see the flow of the working of the code.

3.2.2 The Flowchart Of The Ledbat Congestion Control Algorithm:

The following figure shows the flow of control among the various functional modules in the Linux kernel code of the LEDBAT congestion control algorithm [8]:

Apart from the standard flowchart notations, the double-sided arrow indicates that a function calls another function and after the called function has executed the control is transferred back to the calling function. For example, in Fig. 3.2, "ledbat_update_current_delay()" calls "ledbat_add_delay()" to add delay to the noise_filter. "ledbat_add_delay()", after its execution, transfers the control back to "ledbat_update_current_delay()".

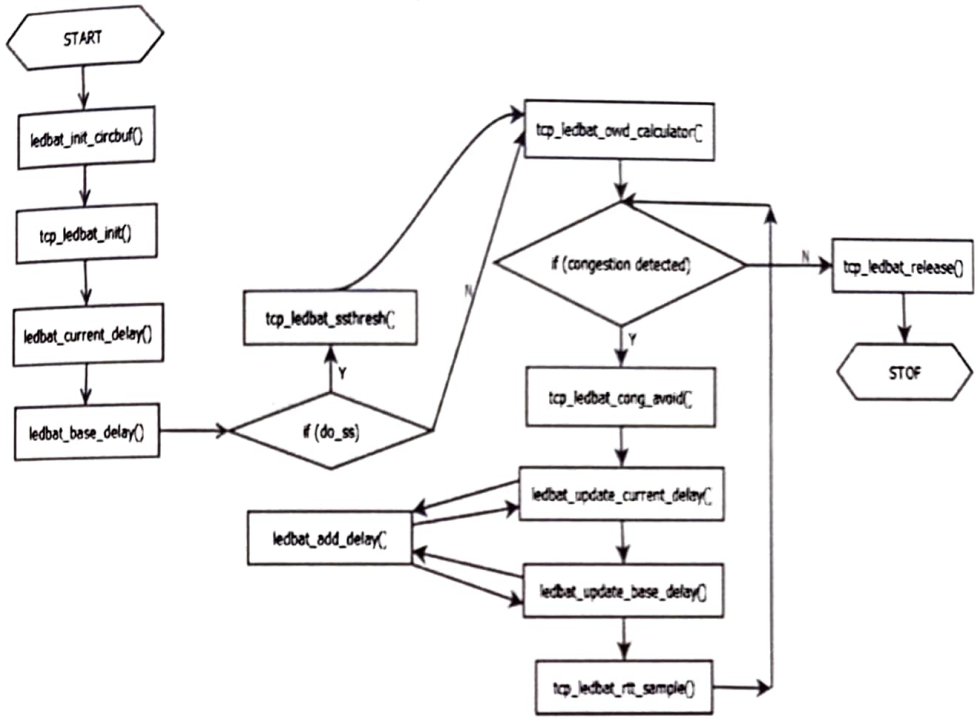

Fig. 3.1: Flowchart of the Linux kernel module of LEDBAT congestion control algorithm

3.3 AUTHENTICATION MECHANISM – FINGERPRINTING LEDBAT

3.3.1 Functional Description

The authentication mechanism is described as follows. Based on the bandwidth utilization of the competing flows, we classify each flow as LEDBAT or Non-LEDBAT flows. The first step is to find the threshold of this utilization of bandwidth. For this 4.4 explains the method of calculation of the threshold and describes the scenarios that we have considered. Based on the set of experiments, we have arrived at the value that, when both LEDBAT flow and TCP flows are happening, the LEDBAT flow shouldn't have more than 0.1 or 10% of the total bandwidth consumed. That is set as the threshold. If, any flow other than the Rich-user's TCP flow, goes beyond this threshold, we classify that particular flow as originating from a Non-LEDBAT flow and hence a malicious user.

So, at the router or the gateway level, we calculate the bandwidth utilization of all the flows. If a particular flow exceeds that of the threshold, we cut that particular flow by either implementing a firewall and dropping packets or by using a MAC filtering approach. The following diagram briefly explains the idea.

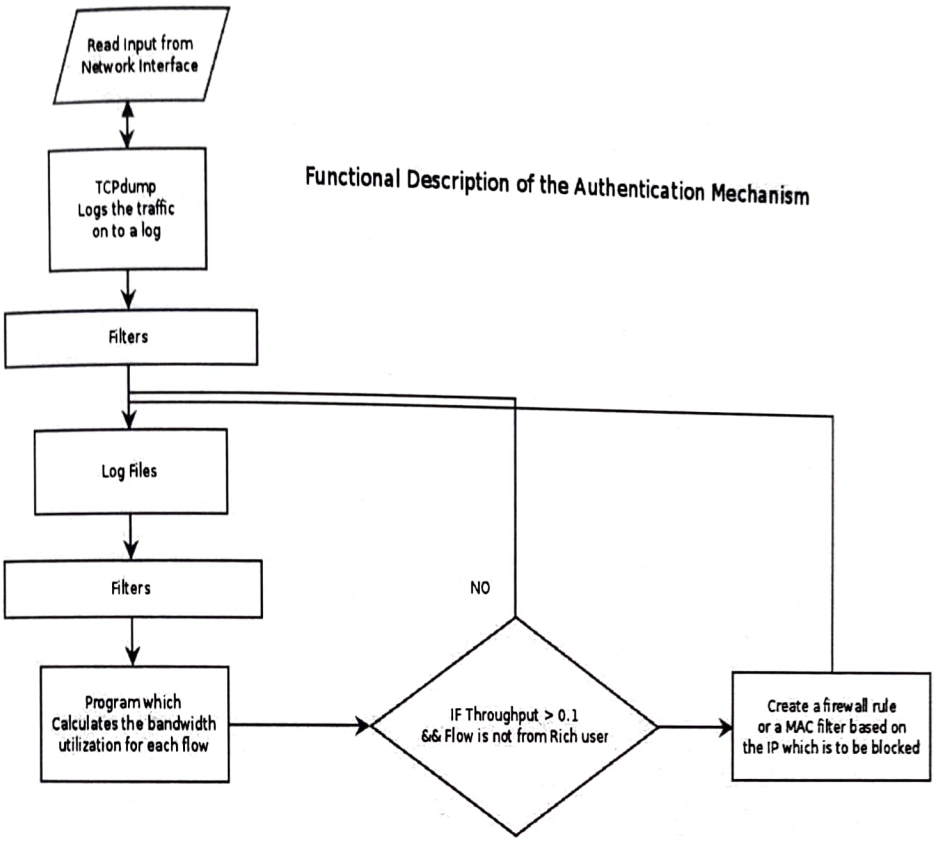

Fig 3.2 Functional Description of the Authentication Mechanism

3.3.2 Implementation issues

First of all, we have to decide where to execute this authentication mechanism. We can either create a router module, which maybe inbuilt into the router, and once a detection of a malicious node is made, we can add it to the firewall rule. Alternatively, as in our case, we may analyze this at the gateway ie the sink. We analyze the traffic at the gateway, and use a dumping mechanism to dump the log files, from where, we use a C++ file to detect the malicious node. 4.3 clearly explains the experiment performed and Appendix C lists the required script files and associated C++ files.

CHAPTER 4

RESULTS AND ANALYSIS

4.1 ANALYSIS OF CONGESTION CONTROL ALGORITHMS

Analyzing congestion control algorithms is not new. A comparative analysis of TCP Congestion control algorithms have been done by S.Patel et. Al [13] which involves AQM techniques tested out in NS2. TCP friendly congestion control algorithms have been tested by S.Jin et. al,[14] which discusses new algorithms, which tend to behave 'like-TCP'. [15] provides a taxonomy for the existing algorithms, treats the Internet as a huge distributed system and congestion control schemes are implemented in every node. While, many of these actually appreciate the working of the algorithms themselves, they don't clearly examine the scenario required for the problem. Hence, specific analysis is required, which considers a specific traffic pattern, with which the choice of the congestion control mechanism maybe concluded.

This experiment proves our very choice to use LEDBAT as a congestion control mechanism to solve the given problem. We analyze a set of well established TCP Control algorithms by simulating them for a scenario as described below, and hence infer that, LEDBAT is the best suited.

The scenario is as follows. Two nodes, are connected to a bottleneck link, whose link capacity is 512kb with a 100ms latency. The individual links capacities are 1Mbps each with a latency of 10ms. Node 1 runs, TCP-Cubic congestion control mechanism with a FTP source attached to it. Node 1 is connected to Node 3, which basically acts like a router ie, start of the bottleneck link and forwards the packet to Node 4, which is the sink. Node 3, which acts as a router, has a *droptail* queue with a queue length of 10. The sink has the capability set according to RFC 1323 [], ie, it supports LFN (Long Fat Networks) [] also. The packet size considered is 512Bytes, and the throughput is calculated for for each flow, every 10s and the graph is plotted on Time(s) vs Throughput (bps). Node 2, will run, TCP-Reno, TCP-Newreno, TCP-Vegas, TCP-Sack, TCP-BIC, TCP-Cubic, and finally TCP-LEDBAT. TCP-LEDBAT by default will not be present as part of the NS2 suite. Refer to 4.2 for installation of TCP-LEDBAT.

The traffic scenario is as follows. The entire duration of the simulation is for 1500s. The FTP0 traffic, originating from Node 1, runs from 5.0s to 400.0s and again restarts from 900.0s and runs up to 1200.0s. The FTP1 traffic, originating from Node 2, runs from 200.0s and runs till the end of the simulation. The reason for creating such a traffic pattern is explained as follows. This mechanism correctly predicts, what happens, how an existing traffic (from Node 0) reacts in the presence of new traffic (from Node 1). This is predicted at 200.0s when FTP1 begins, and continues for 200.0 more seconds. Additionally, this mechanism also predicts, how exactly the second traffic, exploits the additional bandwidth when it becomes available when an existing traffic (from Node 0) is removed. This is shown in the interval from 400.0s to 900.0s. Paraphrasing to our problem, basically, we can analyze, what congestion control algorithm to use, to test how the Rich user's traffic is affected when a poor user's traffic is introduced (in the interval 200.0s to 400.0s) and how effective the bandwidth utilization is when the poor user alone utilizes the extra bandwidth (in the interval 400.0s to 900.0s). Overall, this experiment, should tell us, in effect which algorithm is the most effective one to be considered for the given problem. The sample TCL script for this experiment is found in **Appendix A**

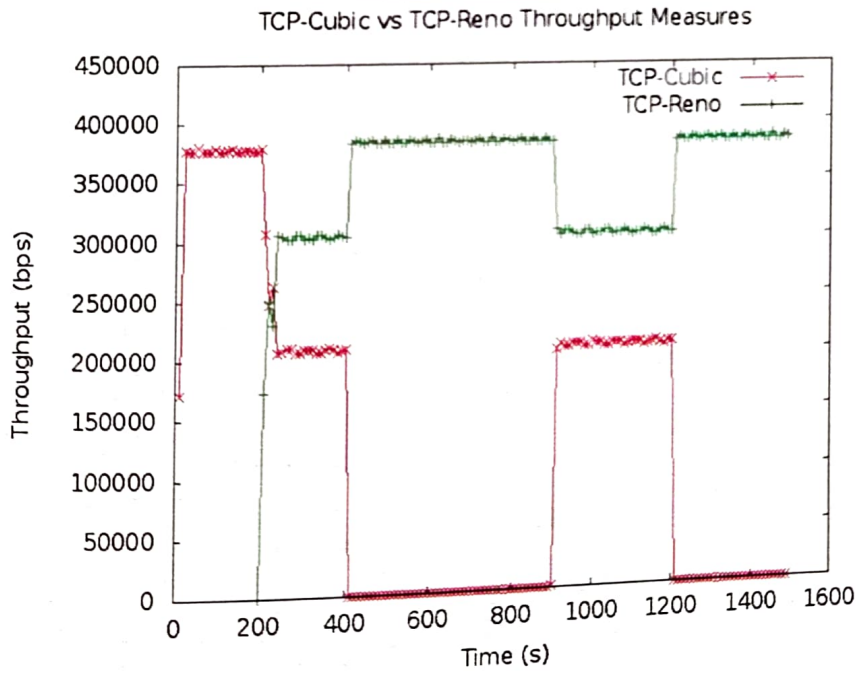

Fig. 4.1 TCP Cubic vs TCP-Reno Throughput Measures

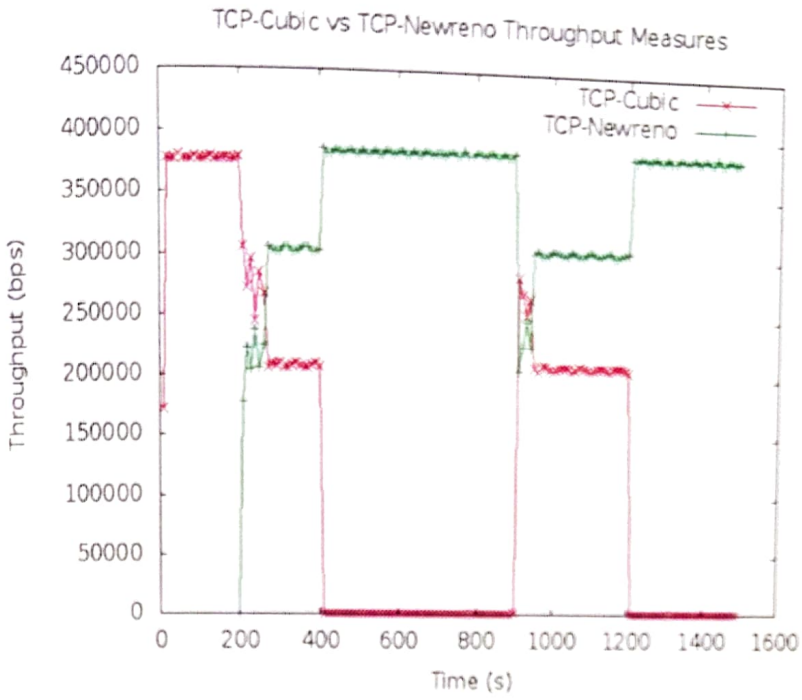

Fig 4.2 TCP-Cubic vs TCP-Newreno Throughput Measures

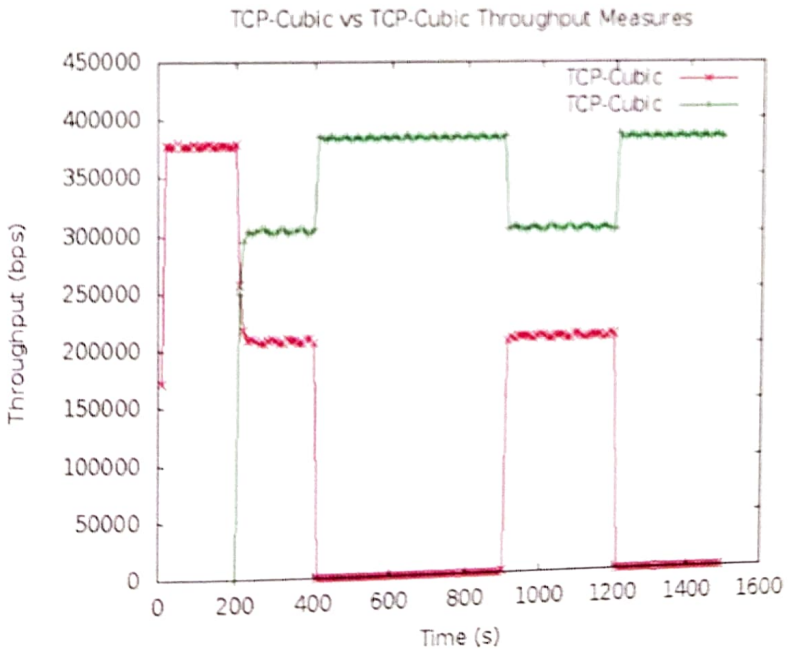

Fig 4.3 TCP-Cubic vs TCP-Cubic Throughput Measures

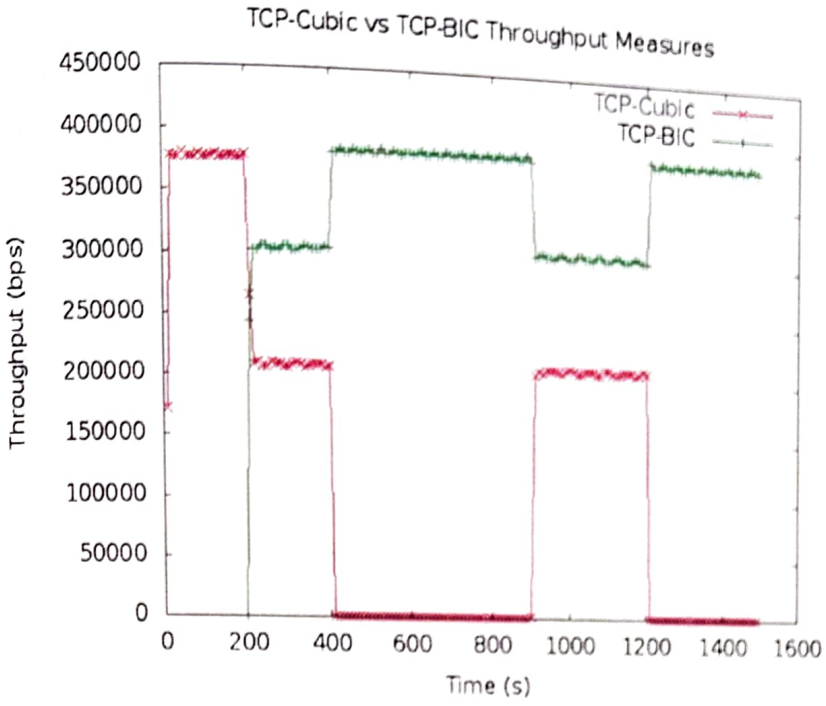

Fig 4.4 TCP-Cubic vs TCP-BIC Throughput Measures

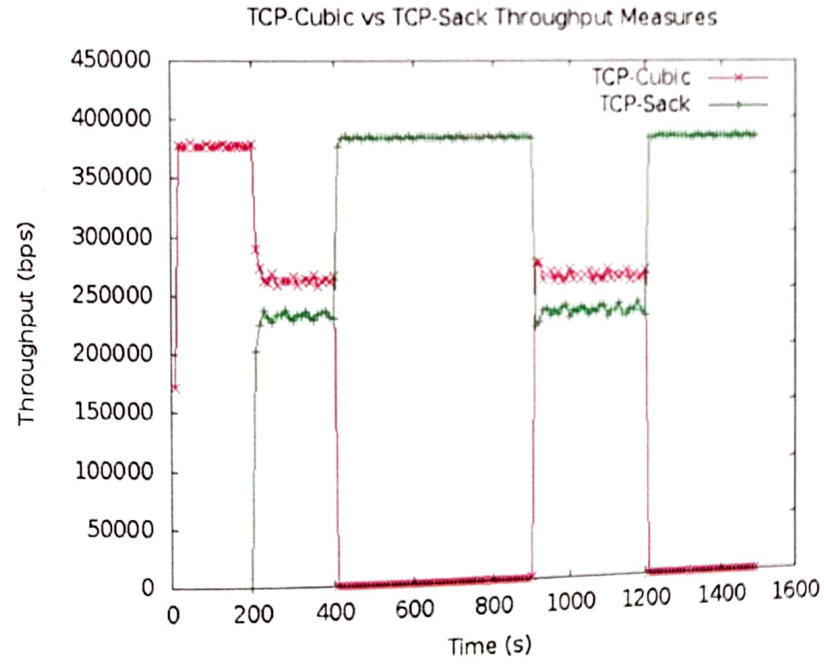

Fig 4.5 TCP-Cubic vs TCP-Sack Throughput Measures

Algorithms up to this, are not delay based congestion control. They are acknowledgment based. Based on the number of outstanding packets

unacknowledged, they increase or decrease the congestion window (Additive Increase, Multiplicative Decrease). Clearly, these algorithms may not be applied to our problem. TCP-Reno, TCP-Newreno, TCP-Cubic are aggressive by themselves. While competing with TCP-Cubic traffic, their dominance is displayed. This can clearly be seen while checking in the interval in Fig 4.1, Fig 4.2, Fig 4.3 and Fig 4.4. During this interval, the second traffic dominates over the existing traffic, and throughput of the base TCP-Cubic traffic is reduced. Even when the TCP-Cubic traffic restarts during the interval 900.0s to 1200.0s, they don't allow it to progress. This clearly doesn't solve our problem. In the case of TCP-Sack, it does provide equitable flows with TCP-Cubic. Unfortunately, we don't require, both these scenarios. In fact, we want it to be opposite, that is, these flows shouldn't hog the existing TCP-Cubic flow. Next, we consider two delay-based congestion control algorithms namely TCP-Vegas and TCP-Ledbat.

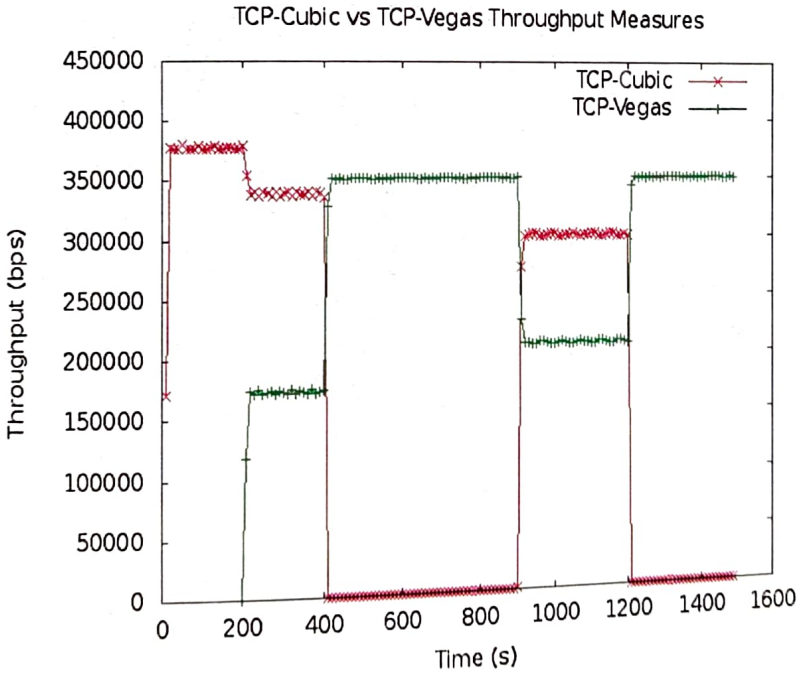

Fig 4.6 TCP-Cubic vs TCP-Vegas Throughput Measures

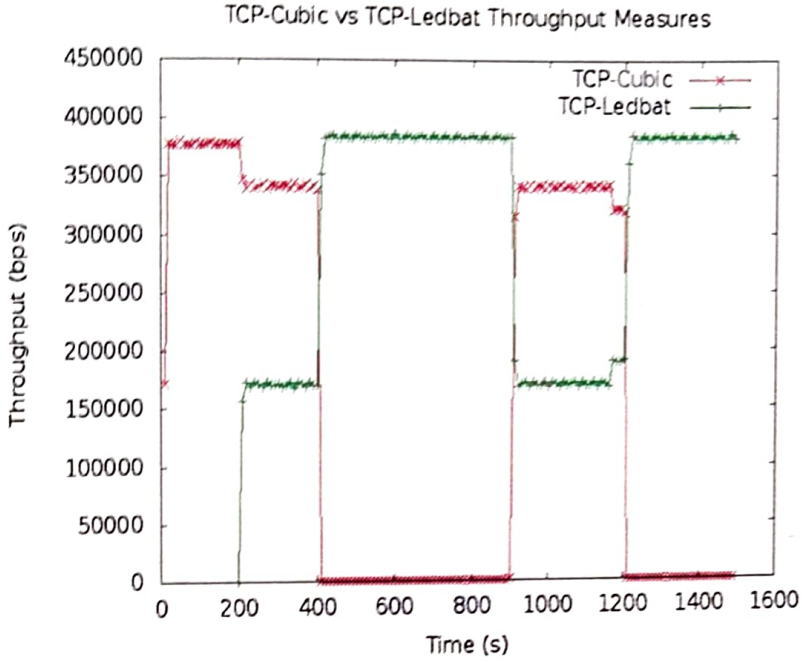

Fig 4.7 TCP-Cubic vs TCP-Ledbat Throughput Measures

As it turns out, LEDBAT is indeed the most effective one to be applied for this problem. This is justified as follows. Consider the time interval from 200.0s to 400.0s and in the interval 900.0s to 1200.0s. Only two algorithms clearly, try to back-off and try not impede the TCP Cubic flow. TCP-Ledbat and TCP-Vegas. Compared among them, TCP-Ledbat is even more less aggressive, and also produces more throughput and hence more utilization of bandwidth, in the absence of a TCP-Cubic flow (in the interval 400.0s to 900.0s). So, TCP-Ledbat, fundamentally, is less aggressive in the presence of a alternate flow, and in the absence probes for more bandwidth and hence has more utlization.

This experiment produces a result, which states that, TCP-Ledbat maybe considered as a choice for a Poor-user's flow, so that it satisfies both our goals. We do, have to consider, more in-depth analysis into Ledbat's variation to packet size, link variations and presence of alternate congestion control algorithms, which we have analyzed in 4.2 and 4.4.

4.2 ANALYSIS OF LEDBAT IN NS-2

4.1 clearly establishes the basic choice of using LEDBAT as a congestion control algorithm for testing. In 4.2 we further perform tests to prove its correctness. Mainly, we also test the Version 9 of the algorithm that we have developed as a Linux Kernel

Module. We have already discussed that Ledbat working draft version 0 has been implemented by Valenti et.al [7], the changes that are to be made (2.2) and we have modified that code (3.2) so that it reflects version 9 of the Ledbat working draft.

In order to test this congestion control mechanism in NS2, as part of the NS2 enhancement project, a Linux TCP implementation for NS2 is present. Developed my Prof. Pei Cao at Stanford and by Prof. Steven Low at Caltech, this tool provides us with a way to test a Linux Kernel Module implementing a congestion control algorithm. We take the Kernel code, and with a few changes to that code, the tutorial for which is very elegantly described in [16], we can conduct simulations in NS2 using the required congestion control mechanism. Exact steps for implementation maybe found in Appendix B.

All the simulations have been done on a Intel Core2Duo T6400 processor, with 4 GB RAM and running Fedora 16 on Linux Kernel Version 3.3.2-6. and NS-2 v.2.34.

4.2.1 Measuring Throughput, Latency, Jitter In Multiple Ledbat Flows

The simulation scenario is explained clearly in Fig. 4.8. The packet size is varied as 512 bytes, 1024 bytes and 2048 bytes. The simulation is run for 1500.0s and the TCP traffic starts at 5.0s. The LEDBAT-1 flow begins at 400.0s and LEDBAT-2 flow begins at 800.0s. They continue till the simulation ends. A Constant-Bit-Rate traffic is present in all the three nodes, with a packet generation interval of 0.01s ie 100 packets per second. The network metrics to be considered are Throughput, End-to-end delay between the node and the sink, and the jitter in the delay. These are often used to test the credibility of a working computer network. The sample TCL script for this experiment is found in Appendix 123. As an additional experiment to prove that, LEDBAT maybe used beyond doubt, we run the same simulation using two TCP-Vegas flows, and represent their throughput graphs for a packet size of 512 Bytes.

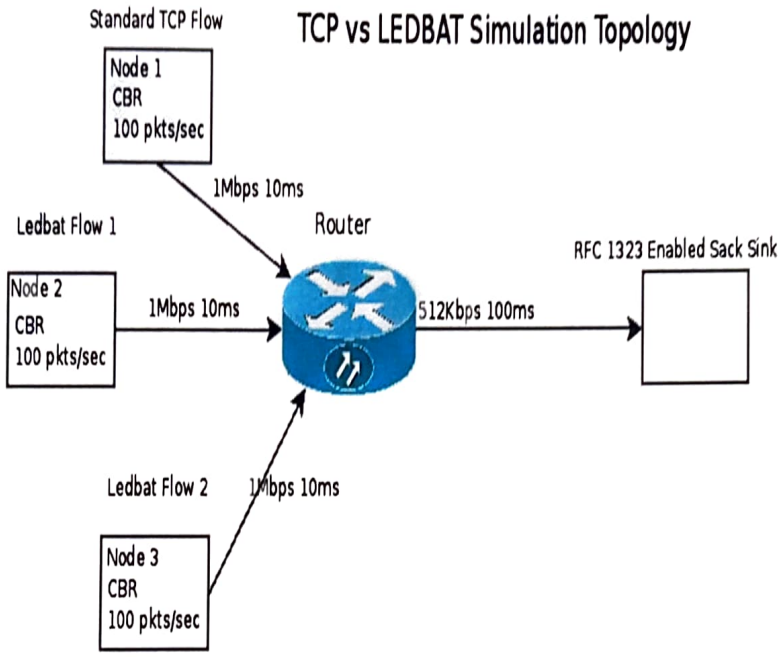

Fig. 4.8 TCP vs LEDBAT Simulation Topology

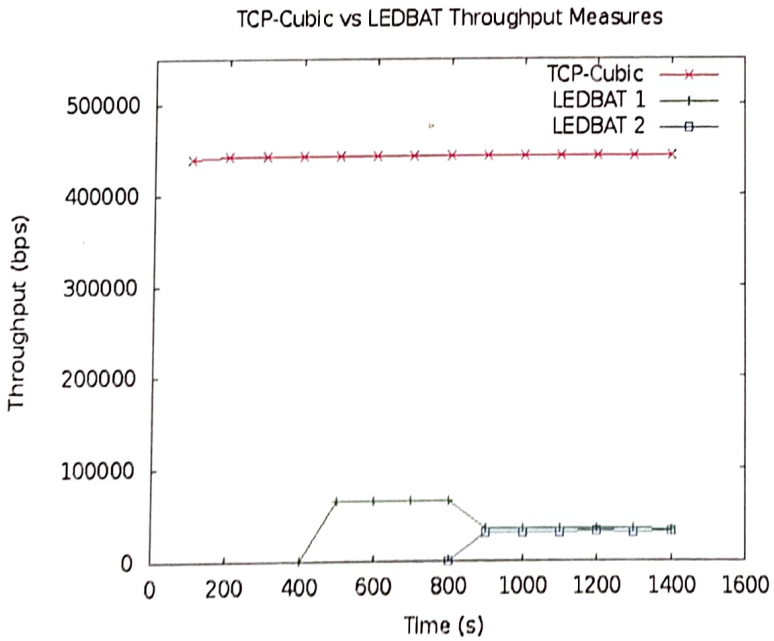

Fig 4.9 TCP-Cubic vs LEDBAT Throughput – 512 Bytes packet Size

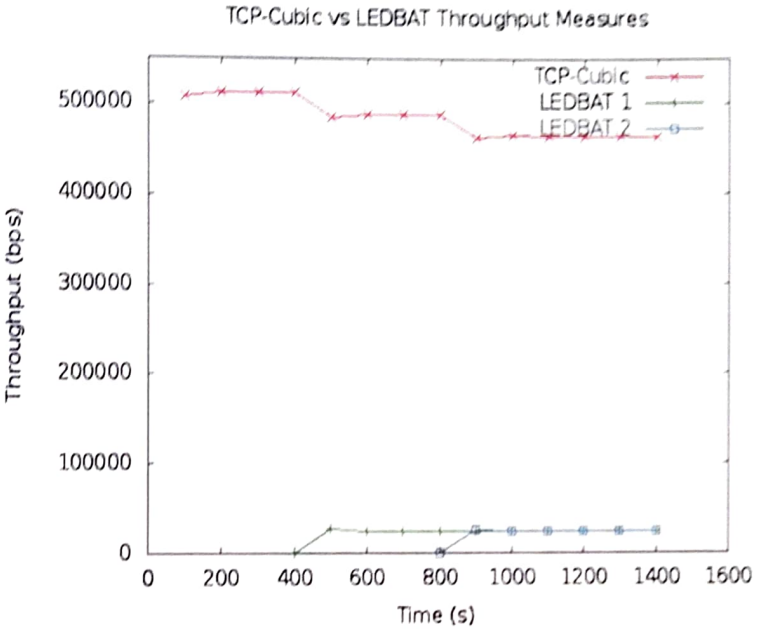

Fig 4.10 TCP-Cubic vs LEDBAT Throughput – 1024 Bytes Packet Size

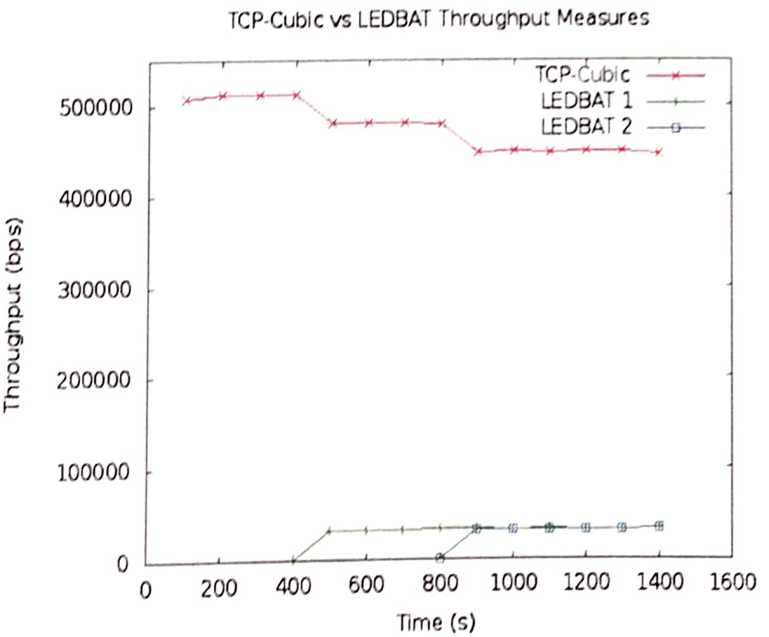

Fig 4.11 TCP-Cubic vs LEDBAT Throughput – 2048 Bytes Packet Size

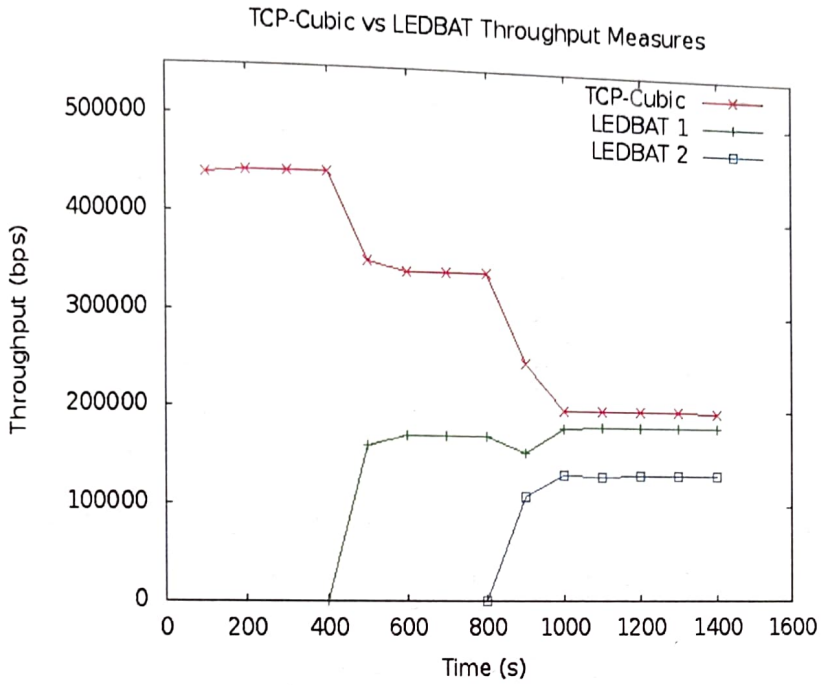

Fig 4.12 TCP-Cubic vs TCP-Vegas Throughput Measures – 512 Bytes Packet size

These graphs prove two things. First, comparing Fig 4.12 and Fig 4.10, it is proved beyond doubt that, TCP-Vegas cannot be used to solve our problem. In case of multiple flows, the throughput for the TCP-Cubic flow, reduces drastically. Unlike the case as shown in Fig 4.6 where, it maybe inferred that, TCP-Vegas maybe a potential algorithm to be implemented, this comprehensively proves otherwise. Next, in the presence of multiple LEDBAT flows, they do compete among themselves and they share the marginal bandwidth that is available. Although the LEDBAT Draft [] suggests that, there might be a scenario where, in competing flows, we have a case of scavenging effect. But, here that doesn't happen, because of the presence of a TCP-Cubic flow, which smooths the effect of all delays, and hence delays remain constant. We have now analyzed the effect of multiple LEDBAT flows ie the presence of more than one poor user in the vicinity of the rich user. Again, we have proved that, it is the excess bandwidth will be shared among the LEDBAT traffic, and the TCP-Cubic bandwidth will be unaffected. In order to verify that, packet size has no effect on LEDBAT, we run it with different values of packet size as 512 Bytes, 1024 Bytes and 2048 Bytes. As per the results, it absolutely has no effect whatsoever on the Throughput of the flows. Next we shall analyze the End-to-end Latency across the links, and the jitter of the flows. We show only the results with packet size of 512

Bytes, as the results with Packet size 1024 Bytes and 2048 Bytes are similar and are hence discarded.

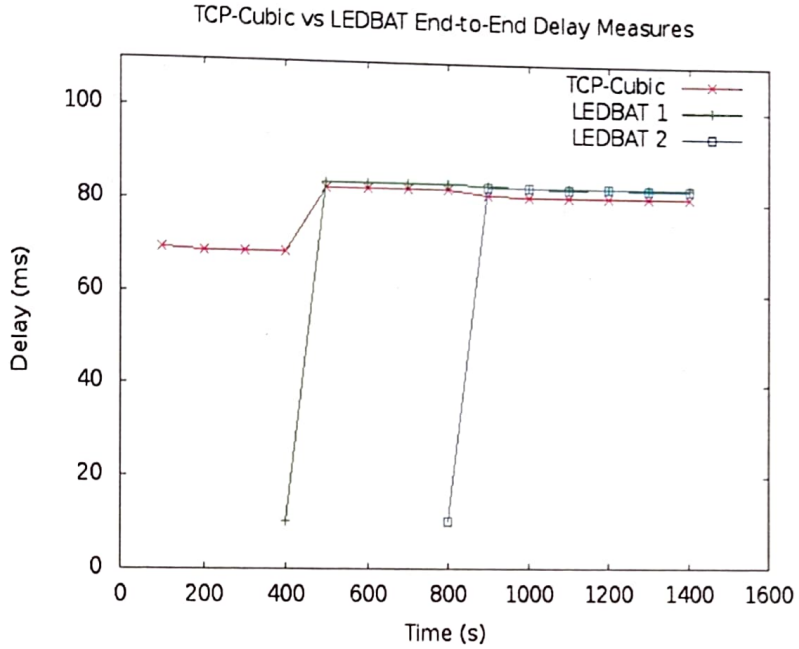

Fig 4.13 TCP-Cubic vs LEDBAT End-to-end Delay measures – 512 Bytes packet size

The delay is expected to grow in the presence of an additional flow, as the queuing delay increases. But, the delay of the TCP-Cubic flow still is marginally lesser compared to the LEDBAT flows.

The jitter values are considerably high for the LEDBAT flows, as they are reactive to the delay. The fundamental requirement that the TCP-Cubic flow's jitter should be as minimum is met. The jitter value of the TCP-Cubic flow is practically zero. The jitter values of the LEDBAT flows although are present, are only few tens of a Micro Second, which is considered to be really low.

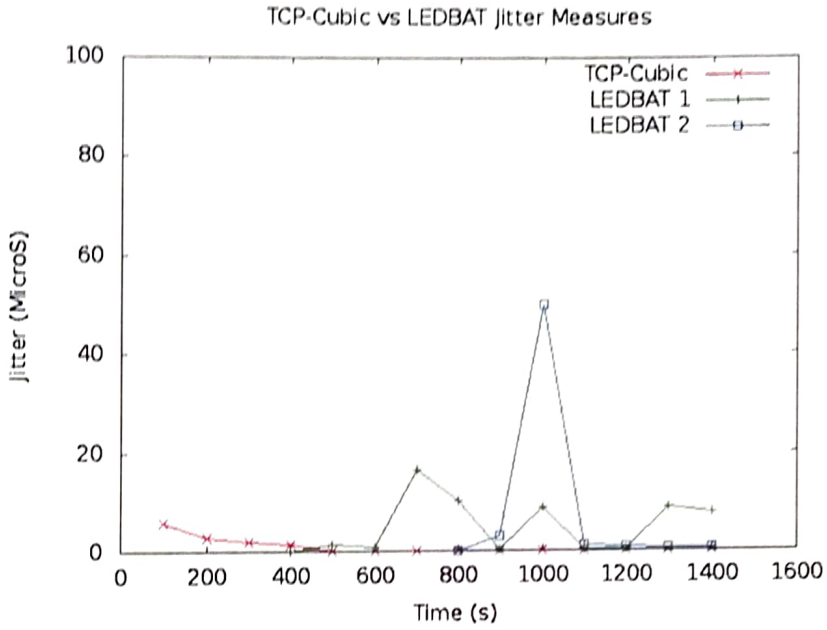

Fig 4.14 TCP-Cubic vs LEDBAT Jitter Measures – 512 Bytes Packet Size

4.2.2 Ledbat In A Wireless Scenario

The same experiment is now tested in a wireless scenario. The physical links are replaced by nodes with zero mobility. The router, is replaced by a access point. The system is described by Fig. 4.15. The system is configured based on what [23] has done. A hierarchical routing model with 2 domains viz a wired one connecting the Sink to the access point, and another one containing the Nodes with the access point as the wireless domain. We select LEDBAT as congestion control algorithm, in Node 2 and Node 3, while Node 1 implements the usual TCP-Cubic mechanism. The TCL file maybe found in Appendix 123.

We run a similar traffic scenario like in 4.2.1, and we find that, the throughput graphs in Fig 4.16 are similar to Fig. 4.09. The major issue that we foresaw was, since LEDBAT is a delay based congestion control mechanism, the variations in the delay in the link, might create problems with its working. But, simulation studies have proven otherwise, and hence we conclude, LEDBAT maybe implemented on mobile devices also (like the Aakash Tablet [18]).

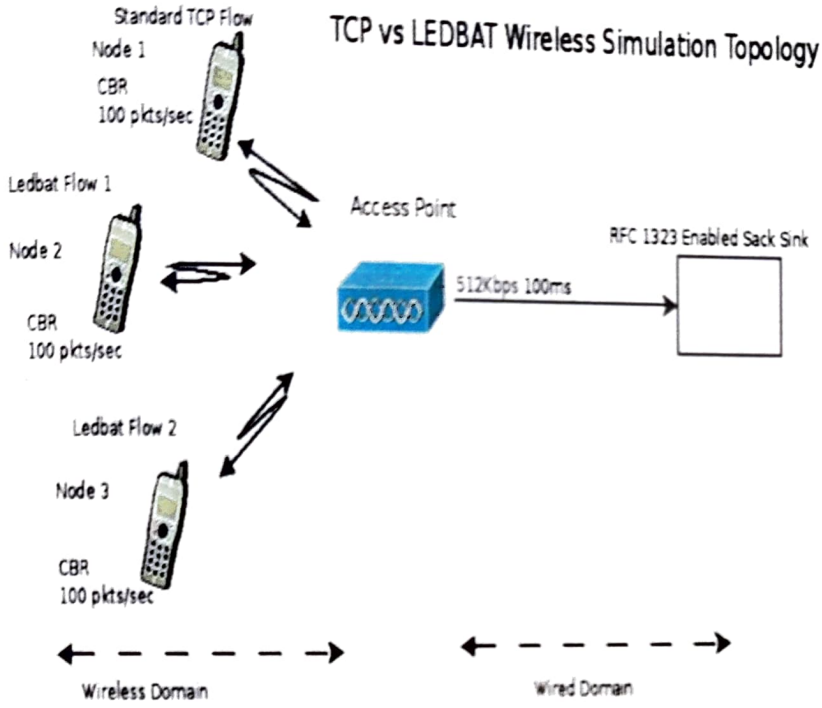

Fig. 4.15 TCP vs LEDBAT Wireless Simulation Topology

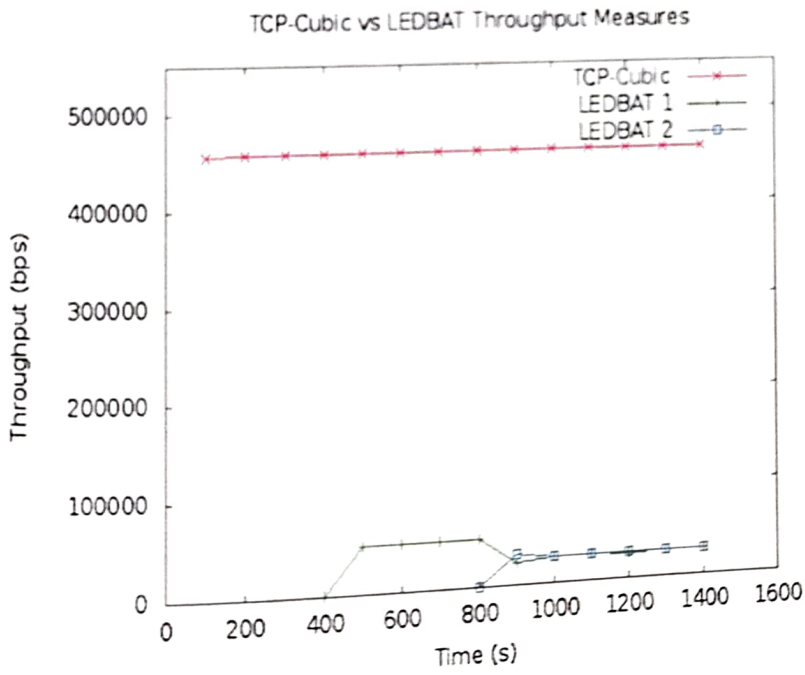

Fig 4.16 TCP-Cubic vs LEDBAT Throughtput Measures – Wireless Scenario

4.2.3 Ledbat In The Presence Of A Udp Flow

Our final analysis of LEDBAT in NS2 concludes with a simple scenario of considering LEDBAT flows in the presence of a UDP flow. Similar scenario settings like in 4.2.1 are considered, except, we change the mechanism for Node 1 to UDP agent instead of the usual TCP-Cubic agent. The graphs clearly indicates that, the UDP traffic is unaffected by the LEDBAT flow.

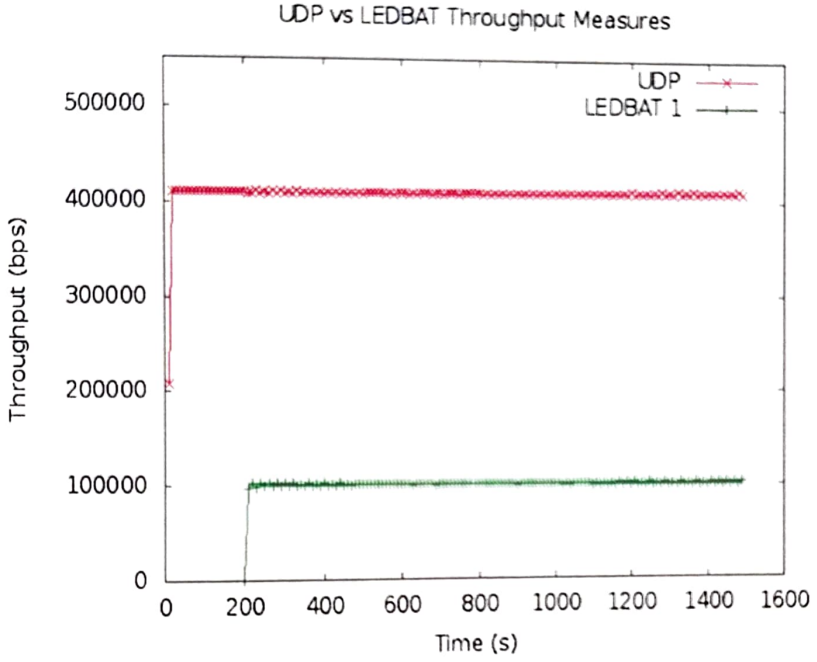

Fig 4.17 UDP vs LEDBAT Throughput Measures

With the above experiments, we have simulated and proven the use of LEDBAT as a congestion control mechanism to be used for the poor-user, and he doesn't impede the Rich-user's connectivity. We will now move on to the Linux-kernel-Module and two specific experiments which have been done with the module.

4.3 IMPLEMENTATION OF LINUX KERNEL MODULE

Section 3.2 clearly explains the Kernel code completely. The kernel module is loaded into the kernel using the *insmod* command, and then the kernel parameters are manipulated using the *sysctl* command and the congestion control algorithm is chosen as ledbat. Appendix B clearly explains the process involved in loading and the kernel module.

The following experiment is done, by connecting three computers each running a Linux Kernel version above 2.65. The three systems are connected together by using a Cisco Linksys WRT 120N router, using wired links.

Details of the Individual computer are as follows:-

Node 1 – Running LEDBAT – Intel Core2Duo T6400 running 64 bit Fedora 16 with Linux Kernel version 3.3.2-6.fc16.x86_64

Node 2 – Running TCP-Cubic – Intel Centrino running 32 bit Ubuntu 11.04 with Linux Kernel version 2.6.38.8 - Generic

Node 3 – Acting as the Sink – Intel Core2Duo running 32 bit Ubuntu 11.10 with Linux Kernel version 2.6.38.8 - Generic

The variation in the systems also makes the experiment more realistic and hence such a scenario was chosen.

Testing of bandwidth across the links has been done using Iperf [17]. Iperf is a commonly used network testing tool that can create TCP and UDP data streams and measure the throughput of a network that is carrying them. It is a tool for network performance measurement written in C++. It was developed by the Distributed Applications Support Team (DAST) at the National Laboratory for Applied Network Research (NLANR).

We start Iperf in Computer 2 running TCP-Cubic at 0.0s running for 20.0s. At 10.0s we start Iperf in Computer 1 running LEDBAT running for 60.0s. Now, at 20.0s, TCP-Cubic traffic will end, and the LEDBAT traffic will be expected to take over. Now, at 40.0s, we again reintroduce the TCP-Cubic traffic, by running Iperf once again for 20.0s.

The dump values from Iperf are taken at intervals of 1.0s and the graph is plotted in Fig 4.18. The graph explicitly proves the working of LEDBAT. Initially, LEDBAT is found to have a very low Bandwidth, and after 20.0s when TCP-Cubic traffic has subsided, it utilizes the un-used bandwidth. Once, TCP-Cubic traffic is reintroduced during at 40.0s, we see that LEDBAT is found to be dormant.

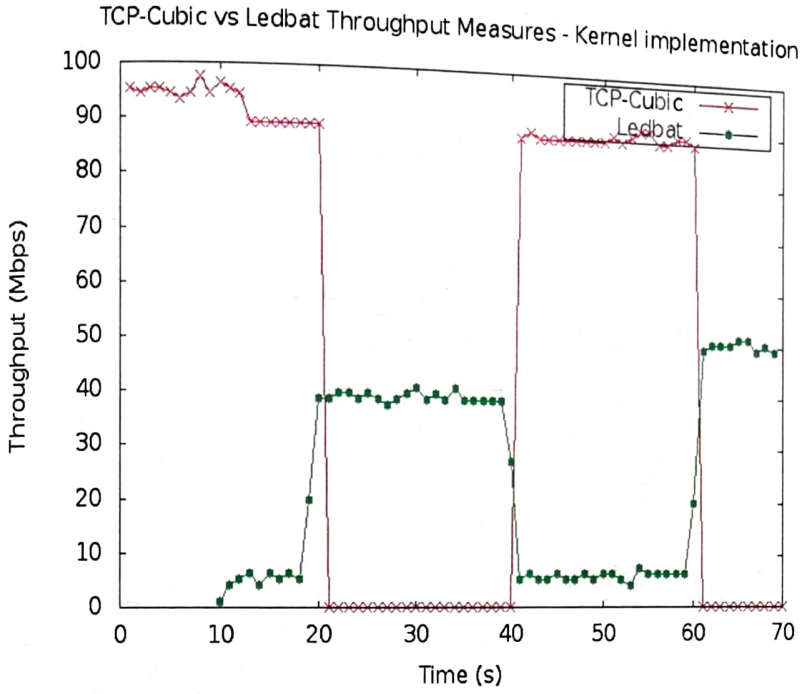

Fig 4.18 TCP-Cubic vs LEDBAT Throughput Measures – Kernel Implementation

Now, we can absolutely say that LEDBAT which was initially developed to be as a innovative method for Background Transport in the internet, now maybe used for bridging the social divide, by providing an access method for equal participation. The next obvious question is how we make sure that the poor-user runs only LEDBAT flow. As 3.3 discusses the Authentication mechanism, our next step is finding the Threshold value. We discuss this in 4.4.

4.4 DETECTION OF NON-LEDBAT FLOWS – AUTHENTICATION

4.4.1 Fingerprinting Ledbat Flows

A single LEDBAT flow is considered along with a single TCP-Cubic flow. We try to analyze the threshold of utilization of bandwidth of LEDBAT flow. We vary the Link capacities from 256Kbps to 10Mpbs and the generation rate from 512Kbps to 10Mpbs to study the percentage utilization of the LEDBAT traffic. Table 4.1 describes the scenario when using one LEDBAT flow with a Cubic flow while Table 4.2 shows the same when using two Cubic flows. This is done in order to show the variation in utilization.

Table 4.1 TCP-Cubic vs LEDBAT – Fingerprint Analysis

Scenario: TCP-Cubic vs LEDBAT	Case 1	Case 2	Case 3	Case 4	Case 5
Link capacity	256 Kbps	512 Kbps	1 Mbps	2 Mbps	10 Mbps
Generation rate	512 Kbps	512 Kbps	1 Mbps	1 Mbps	10 Mbps
Single Ledbat Traffic (Bps)	37715	54982	73663	98285	102734
Single TCP-Cubic Traffic (Bps)	151902	508133	998608	1124997	1163172
TCP-Cubic Competing flow (Bps)	145311	472353	958639	1123191	1160562
Ledbat Competing flow (Bps)	23367	35012	64038	91912	102690
LEDBAT : TCP-Cubic Flow	0.1608068	0.0741225	0.066801	0.0818311	0.088483
LEDBAT : Pure TCP-Cubic Flow	0.1538294	0.0689032	0.0641273	0.0816998	0.0882844

Table 4.2 TCP-Cubic vs TCP-Cubic – Fingerprint Analysis

Scenario: TCP-Cubic1 vs TCP-Cubic2	Case 1	Case 2	Case 3	Case 4	Case 5
Link capacity	256 Kbps	512 Kbps	1 Mbps	2 Mbps	10 Mbps
Generation rate	512 Kbps	512 Kbps	1 Mbps	1 Mbps	10 Mbps
Single TCP-Cubic1 Traffic (Bps)	151902	508184	992419	1124540	1156526
Single TCP-Cubic2 Traffic (Bps)	151902	508184	992419	1124540	1156526
TCP-Cubic1 Competing flow (Bps)	91206	153882	315500	995845	1159691
TCP-Cubic2 Competing flow (Bps)	92276	155732	295934	999173	1159387
TCP-Cubic2 : TCP-Cubic1 Flow	1.0117317	1.0120222	0.9379842	1.0033419	0.9997379
TCP-Cubic2 : Pure TCP-Cubic1 Flow	0.6074706	0.3064481	0.2981946	0.8885171	1.0024738

Consider the ratio of LEDBAT : TCP-Cubic competing flow. Barring an anomaly for 256Kbps, it is clearly within the Limits of 0.1 or 10%. This implies that, in a competing scenario, LEDBAT traffic should not consume more than 10% of the total bandwidth available. Consider the second table, and again consider the same ratio TCP-Cubic1: TCP-Cubic2 in a competing scenario. The ratio comes close to 1. This implies that, there is equal competition in the Cubic flows. This is also shown in the simulations (Fig 4.3). Hence, we can safely fix the threshold value for a valid LEDBAT flow in a competing scenario to be 10% or 0.1. If it exceeds this value, and it is not a rich user host, then we have a malicious user. This is implemented in 4.4.2.

4.4.2 Physical Testing Of The Authentication Mechanism

4.4.1 has determined the threshold value for the LEDBAT traffic utilization. In order to prove its correctness, a C++ program was written which provided with log files ie the packet size and the packet origin, would classify which traffic to be a Non-Ledbat traffic. The pseudo-code for this is explained in 3.3. Similar to 4.3, we considered two scenarios. Case 1) LEDBAT vs TCP-Cubic, and Case 2) TCP-Cubic vs TCP-Cubic.

The C++ program was compiled using gcc-4.6.3 on a Intel machine running Ubuntu 12.10 which acted as the Sink/Gateway.

The logging mechanism used was *tcpdump* along-with a few awk scripts, the input file for the C++ program was obtained. The C++ program correctly identified in Case 2) the second TCP-Cubic traffic as a malicious user. We used the Linux *iptables*, the inbuilt firewall to drop all the packets from that particular IP address. Alternatively, if the router provides a MAC filter mechanism, using the IP address, we may obtain the MAC addressing, and do a MAC filtering at the access point itself. The associated scripts and tcpdump commands maybe found Appendix 123.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 CONCLUSION

As shown by our simulations in NS-2 as well as using the Linux kernel module, we can establish beyond doubt that the best low priority congestion control algorithm to be used in a poor user's terminal is LEDBAT. Also, as our Authentication mechanism suggests, we are also able to show that, in the presence of anti-social users, we are able to prevent them from utilizing this service. Irrespective of a wired or a wireless medium, LEDBAT environment for a poor user and any high priority protocol for a rich user would be the ideal package for bridging the digital divide and as insisted at the beginning of the thesis, be a true recipe for implementing Social Inclusion in the Digital World.

5.3 SCOPE FOR FUTURE WORK

We have shown the lasting contribution LEDBAT can make in personal computers and laptops. This idea can be extended to providing platforms for LEDBAT in the Android market as well as for devices like AKASH tablets [18]. This would require that the existing Linux kernel module be ported to the Android kernel – which is a fork of the Linux kernel so that, it might be accurately implemented in Low-Cost Android devices also. We are yet to analyze a QoS implementation at the Router level, using variable queuing mechanisms eg. Using a Multi-Level queue or some form of Active Queue Management (AQM). This would require the production of custom set of hardware (Routers) for implementing it in a day-to-day scenario.. We have analyzed Transport layer protocols for this problem. Protocols at the lower end of the stack eg. Data Link Layer Protocols maybe analyzed to check the effectiveness in solving the problem.

REFERENCES

1. TCP Vegas: End to End Congestion Avoidance on a Global Internet, IEEE Journal on Selected areas in Communications, Lawrence Brakamo and Larry Peterson, 1995.

2. RFC 2018, TCP Selective Acknowledgement Options, M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, 1996.

3. RFC 3782, The NewReno Modification to TCP's Fast Recovery Algorithm, S. Floyd, T. Henderson, A. Gurtov, 2004

4. TCP Nice: A Mechanism for Background Transfers, Arun Vekataramani, Ravi Kokku and Mike Dahlin, 5th Symposium on Operating Systems Design and Implementation, 2002

5. TCP-LP: A Distributed Algorithm for Low Priority Data Transfer, A. Kuzmanovic and E.W. Knightly, IEEE INFOCOM, 2003

6. CUBIC: A New TCP-Friendly High-Speed TCP Variant, Injong Rhee and Lisong Xu

7. http://tools.ietf.org/html/draft-ietf-ledbat-congestion-00, Low Extra Delay Background Transport (LEDBAT) draft-ietf-ledbat-congestion-09, S. Shalunov, 2009

8. http://tools.ietf.org/html/draft-ietf-ledbat-congestion-09, Low Extra Delay Background Transport (LEDBAT) draft-ietf-ledbat-congestion-09, S. Shalunov, G. Hazel, J. Iyengar, M. Kuehlewind, 2011.

9. Binary Increase Congestion Control for Fast, Long Distance Networks, Lisong Xu, Khalid Harfoush and Ijong Rhee.

10. Competitive and Considerate Congestion Control for Bulk-Data Transfers, Shao Liu, Milan Vojnovic and Dinan Gunawardena, 2006.

11. MulTFRC: Providing Weighted Fairness for Multimedia Applications, Dragana Damjanovic and Michael Welzl, 2009.

12. LEDBAT: The new BitTorrent Congestion Control Protocol, Dario Rossi, Claudio Tests, Silvio Valenti and Luca Muscariello.

13. Comparative Analysis of Congestion Control Algorithms Using ns-2 by Sanjeev Patel, P. K. Gupta, Arjun Garg, Prateek Mehrotra, Manish Chhabra IJCSI International Journal of Computer Science Issues, Vol. 8, Issue 5, No 1, September 2011

14. A spectrum of TCP-friendly window-based congestion control algorithms by Shudong Jin, Liang Guo, Ibrahim Matta, Azer Bestavros

15. A taxonomy for congestion control algorithms in packet switching networks by Cui-Qing Yang IEEE, Jul/Aug 1995

16. Tutorial for implementing a Congestion control algorithm. http://netlab.caltech.edu/projects/ns2tcplinux/ns2linux/tutorial/index.html

17. Project Iperf host page and documentation.http://sourceforge.net/projects/iperf/

18. http://en.wikipedia.org/wiki/Aakash_(tablet)

19. Data Transfer over the Long Fat Networks by H. Sato, Y. Morita, Y. Karita, Y. Watase

    KEK, Tsukuba, Japan

20. Performance evaluation of a Python Implementation of the New LEDBAT congestion control algorithm by Mugurel Ionot Andrecia, Nicole Tapus, TU Delft

21. LEDBAT: The new BitTorrent congestion control protocol by Dario Rossi, Claudio Testa, Silvio valenti Luca Musciarello

22. Impact of delay variability on LEDBAT performance by Amuda James Abu and Steven Gordan, Thamassat University.

23. Creating Wired-cum-Wireless and MobileIP Simulations in ns2 http://www.isi.edu/nsnam/ns/tutorial/nsscript6.html#second

APPENDIX A

NS2 simulations

A.1 Wired tcl simulation file

This is the sample TCL file, that we have used for Wired Simulation.

```
// tcp_ledbat_congestion.tcl

set ns [new Simulator]

#set nam_trace_fd [open tcp_congestion_ledbat.nam w]

#$ns namtrace-all $nam_trace_fd

set trace_fd [open tcp_congestion_ledbat.tr w]

$ns trace-all $trace_fd

# open the measurement output files

set throughput_flow_0_trace_fd [open tcp_congestion_ledbat_0.tr w]

set throughput_flow_1_trace_fd [open tcp_congestion_ledbat_1.tr w]

set packetSize 512

#Define a 'finish' procedure

proc finish {} {

    #global ns nam_trace_fd

    global ns trace_fd

    global trace_fd throughput_flow_0_trace_fd throughput_flow_1_trace_fd

    $ns flush-trace

      # close the measurement files

      close $trace_fd

      close $throughput_flow_0_trace_fd
```

```
        close $throughput_flow_1_trace_fd
        exec gnuplot ledbat_congestion_throughput.p
    exit 0
}
# Records Statistics
proc record_stat {} {
        global flow_monitor packetSize
        global throughput_flow_0_trace_fd throughput_flow_1_trace_fd
        # get an instance of the simulator
        set ns [Simulator instance]
        # set the time after which the procedure should be called again
        set time 10
        # get the current time
        set now [$ns now]
        # how many bytes have been received by the traffic sinks?
        set packet_arrival [$flow_monitor set parrivals_]
        set packet_departure [$flow_monitor set pdepartures_]
        set packet_drop [$flow_monitor set pdrops_]
        set flow_classifier [$flow_monitor classifier]
        set flow_fd [$flow_classifier lookup auto 0 0 0]
        if { $flow_fd != "" } {
                # calculate the throughput and write it to the files
```

```
                puts $throughput_flow_0_trace_fd "$now \t [expr [$flow_fd set
bdepartures_]*8/$time]"


                $flow_fd set barrivals_ 0

                $flow_fd set bdepartures_ 0

                $flow_fd set bdrops_ 0

        }

        set flow_fd [$flow_classifier lookup auto 0 0 1]

        if { $flow_fd != "" } {

                # calculate the throughput and write it to the files

                puts $throughput_flow_1_trace_fd "$now \t [expr [$flow_fd set
bdepartures_]*8/$time]"

                $flow_fd set barrivals_ 0

                $flow_fd set bdepartures_ 0

                $flow_fd set bdrops_ 0

        }

        # reset the bytes_ values on the traffic sinks

         $flow_monitor set barrivals_ 0

         $flow_monitor set bdepartures_ 0

         $flow_monitor set bdrops_ 0

       # re-schedule the procedure

         $ns at [expr $now+$time] "record_stat"

}
```

```
# Create 6 nodes - 4 Generate packets (TCP-Reno, Ledbat1, Ledbat2, Ledbat3) + 2
nodes for bottle Neck

set node0 [$ns node]

set node1 [$ns node]

set node2 [$ns node]

set node4 [$ns node]

set node5 [$ns node]

# create links between the nodes

$ns duplex-link $node0 $node4 1Mb 10ms DropTail

$ns duplex-link $node1 $node4 1Mb 10ms DropTail

$ns duplex-link $node4 $node5 512Kb 100ms DropTail

$ns queue-limit $node4 $node5 10

# flow Monitor

set link_node4_node5 [$ns link $node4 $node5]

set flow_monitor [$ns makeflowmon Fid]

$ns attach-fmon $link_node4_node5 $flow_monitor

# monitor the queue for the link between node 2 and node 3

$ns duplex-link-op $node4 $node5 queuePos 0.5

set tcp_0 [new Agent/TCP/Linux]

$tcp_0 select_ca cubic

$ns attach-agent $node0 $tcp_0

$tcp_0 set fid_ 0

$tcp_0 set class_ 0
```

```
$tcp_0 set packetSize_ $packetSize

# create a TCP sink agent and attach it to node node5

set sink [new Agent/TCPSink/Sack1]

$sink set ts_echo_rfc1323_ true

$ns attach-agent $node5 $sink

#       connect both agents

$ns connect $tcp_0 $sink

# create an FTP source

set ftp0 [new Application/FTP]

$ftp0 attach-agent $tcp_0

$ftp0 set type_ FTP

# Setting up TCP, CBR node wise

set tcp_1 [new Agent/TCP/Linux]

$tcp_1 set timestamps_ true

$tcp_1 select_ca ledbat

$ns attach-agent $node1 $tcp_1

$tcp_1 set fid_ 1

$tcp_1 set class_ 1

$tcp_1 set packetSize_ $packetSize

# create a TCP sink agent and attach it to node node5

set sink [new Agent/TCPSink/Sack1]

$sink set ts_echo_rfc1323_ true

$ns attach-agent $node5 $sink
```

```
# connect both agents

$ns connect $tcp_1 $sink

# create an FTP source

set ftp1 [new Application/FTP]

$ftp1 attach-agent $tcp_1

$ftp1 set type_ FTP

# schedule events for all the flows

$ns at 0.1 "record_stat"

$ns at 5.0 "$ftp0 start"

$ns at 200.0 "$ftp1 start"

$ns at 400.0 "$ftp0 stop"

$ns at 900.0 "$ftp0 start"

$ns at 1200.0 "$ftp0 stop"

# call the finish procedure after 100 seconds of simulation time

$ns at 1500 "finish"

# run the simulation

$ns run
```

A.2 Wireless tcl Simulation File

```
// wireless_ledbat.tcl

// Only the settings specific to the wireless scenario is shown here. A.1 maybe referred
to for other values

#Define a 'finish' procedure

proc finish {} {....Same as in A.1.....}
```

```
# Records Statistics

proc record_stat {} {....Same as in A.1....}

$topo load_flatgrid 500 500

create-god 5

set num_wired_nodes      1

set opt(chan)          Channel/WirelessChannel    ;# channel type

set opt(prop)          Propagation/TwoRayGround   ;# radio-propagation model

set opt(netif)         Phy/WirelessPhy            ;# network interface type

set opt(mac)           Mac/802_11                 ;# MAC type

set opt(ifq)           Queue/DropTail/PriQueue    ;# interface queue type

set opt(ll)            LL                         ;# link layer type

set opt(ant)           Antenna/OmniAntenna        ;# antenna model

set opt(ifqlen)        50                         ;# max packet in ifq

set opt(nn)            3                          ;# number of mobilenodes

set opt(adhocRouting)  DSDV                       ;# routing protocol

#Wired station configuration

$ns node-config -addressType hierarchical

AddrParams set domain_num_ 2          ;# number of domains

lappend cluster_num 1 1               ;# number of clusters in each domain

AddrParams set cluster_num_ $cluster_num

lappend eilastlevel 1 4               ;# number of nodes in each cluster

AddrParams set nodes_num_ $eilastlevel ;# of each domain

#create wired nodes
```

```
set temp {0.0.0}        ;# hierarchical addresses for wired domain

set node5 [$ns node [lindex $temp 0]]

set chan_1_ [new $opt(chan)]

#Base station configuration

$ns node-config -adhocRouting $opt(adhocRouting) -llType $opt(ll) -macType
$opt(mac) -ifqType $opt(ifq) -ifqLen $opt(ifqlen) -antType $opt(ant) -propType
$opt(prop) -phyType $opt(netif) -channel $chan_1_  -topoInstance $topo
-wiredRouting ON -agentTrace ON -routerTrace OFF -macTrace OFF

#create base-station node

set temp {1.0.0 1.0.1 1.0.2 1.0.3}  ;# hier address to be used for wireless

set BS0 [$ns node [lindex $temp 0]]

$BS0 random-motion 0            ;# disable random motion

#provide some co-ord (fixed) to base station node

$BS0 set X_ 50.0

$BS0 set Y_ 50.0

$BS0 set Z_ 0.0

#nodes config

$ns node-config -wiredRouting OFF

set node0 [$ns node [lindex $temp 1]]

$node0 random-motion 0

$node0 set X_ 150

$node0 set Y_ 0

$node0 set Z_ 0

set node1 [$ns node [lindex $temp 2]]
```

$node1 random-motion 0

$node1 set X_ 150

$node1 set Y_ 100

$node1 set Z_ 0

set node2 [$ns node [lindex $temp 3]]

$node2 random-motion 0

$node2 set X_ 150

$node2 set Y_ 150

$node2 set Z_ 0

#attach nodes to base station

$node0 base-station [AddrParams addr2id [$BS0 node-addr]]

$node1 base-station [AddrParams addr2id [$BS0 node-addr]]

$node2 base-station [AddrParams addr2id [$BS0 node-addr]]

create links between the node and access point

$ns duplex-link $node5 $BS0 512Kb 100ms DropTail

$ns queue-limit $node5 $BS0 10

A.3 Method to implement a Linux Congestion Control algorithm.

Refer to [] for the exact details in converting a Linux kernel module into a congestion control algorithm. It has 5 steps as explained in []

1. The header files to link the implementation to TCP-Linux;
2. Optionally, define and decluare parameters -- parameters have to be defined static because different modules might have different parameters with the same variable names

3. Implementation of (at least) the three congestion control functions defined in struct tcp_congestion_ops: cong_avoid, ssthresh and min_cwnd;

4. A static record of struct tcp_congestion_ops to store the function calls and algorithm's name.

5. An module initialization function which calls tcp_register_congestion_control to register the module.

After all this, the required file has to be copied to the <base-location-of-install>/ns-2.34/tcp/linux/src/

The file Makefile.in in <base-location-of-install>/ns-2.34/ should be edited and the value tcp/linux/src/tcp_ledbat.o must be added to the variable OBJ_CC. After this, we should run the commands make and make install to create ns again. Now, LEDBAT will be loaded into the base network simulator, and we can use it.

APPENDIX B

LINUX KERNEL MODULE

1) Execute the Makefile and build the ledbat module using make command.
2) Execute and link the kernel object as follows: sudo insmod tcp_ledbat.ko
3) Enable the congestion control for new connections: sudo sysctl –w net.ipv4.tcp_congestion_control=ledbat
4) If you prefer not to switch all connections to ledbat, you can just add it to the allowed congestion control algorithms. For example, sudo sysctl –w net.ipv4.tcp_congestion_control="cubic reno ledbat"

The following make file is used to compile the kernel code.

// Makefile

obj-m := tcp_ledbat.o

KDIR := /lib/modules/$(shell uname -r)/build

PWD := $(shell pwd)

default:

$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

clean:

rm -rf Module.markers modules.order Module.symvers tcp_ledbat.ko tcp_ledbat.mod.c tcp_ledbat.mod.o tcp_ledbat.o

After running the command *make*, we load into the kernel by typing the command

sudo insmod tcp_ledbat.ko

After loading it into the kernel, we use the command sysctl to change the Kernel parameters and instruct it to use the LEDBAT congestion control algorithm.

sudo sysctl -w net.ipv4.tcp_congestion_control=ledbat

Option -w indicates that, we change the kernel parameters. net.ipv4.tcp_congestion_control implies that, we refer to the network stack of the kernel and choose IP version 4 and change the congestion control algorithm to LEDBAT. This can only be run after loading ledbat into the kernel by using the insmod command.

In order to read the kernel output, that we would have written in the C program (Linux Kernel Module) we use a command which accesses the Kernel ring buffer called as dmesg. This command dmesg displays all the console outputs that the kernel module might throw up.

APPENDIX C

TRACE FILES OF EXPERIMENTS AND AUTHENTICATION MECHANISM

C.1 C++ File Which Reads Log File And Checks For Malicious Nodes

```
<load-all-required-header-files-and namspaces>

vector<string> tokenizeStr(string str);

long double tmstmp; //current timestamp

long double nxtstmp; // next timestamp

long double bwuNode[4]; //b/w utilization of a particular node

long double pktNode[4]; //total pkts sent by a particular node

map<string, int> ipAddrList;

string returnipAddr(int i)

{

        map <string, int>:: iterator itr;

        string ipAddrToReturn="";

        for(itr = ipAddrList.begin();itr!=ipAddrList.end();itr++){

                if(itr->second == i)

                        ipAddrToReturn = itr->first;

        }

        return ipAddrToReturn;

}

int main(int argc, char *argv[]) {

        if(argc < 4){
```

```
            cout<<"Format to enter: <name-of-program> <name-of-log-file> <ip-
addr-of-primary-host> <period of calculation>"<<endl;
            exit(1);
        }
        assert(argv[1] != NULL);
        char *filename = (char*)argv[1];
        ifstream infile;
        infile.open(filename);
        cout<<"File open success!!!"<<endl;
        string curLine;
        vector<string> tokens;
    int i, period;
    long double tpkt; //total pkts sent by all the nodes
        //intialization
        period = atoi(argv[3]);
    int ipCount = 0;
        ipAddrList.insert(std::pair<string,int>(argv[2],ipCount++));
    cout<<ipCount<<endl;
        tpkt = 0;
        for(i=0;i<4;i++){
            bwuNode[i] = 0.0;
            pktNode[i] = 0;
        }
```

```
tmstmp = -1;
getline(infile, curLine);
while(EOF) {
        tokens = tokenizeStr(curLine);
        if(tokens.size() <3)
                break;
        assert(tokens.size() == 3);
        nxtstmp = atof(tokens[0].c_str());
        if(tmstmp == -1)
                tmstmp = nxtstmp;
        string ip = tokens[1];
        if(ipAddrList[ip] == 0)
                ipAddrList[ip]= ipCount++;
        pktNode[ipAddrList[tokens[1].c_str()]]
+=atoi(tokens[2].c_str());
        if(nxtstmp - tmstmp > period)
{
                //calculating total pkts sent by all the nodes
                cout<<"Time: "<<tmstmp<<endl;
                tpkt = 0;
                for(i=0;i<4;i++)
                        tpkt+=pktNode[i];
                cout<<"Total traffic"<<tpkt<<endl;
```

```
                    //calculating b/w utilization and identifying malicious user
                        for(i=0;i<4;i++){
                            bwuNode[i] = pktNode[i] / tpkt;
                            string ipAddress = returnipAddr(i);
                            if((bwuNode[i] > 0.1) && (i!=0))
                                cout<<"Node" <<i<<" is malicious user
and the IP address is "<<ipAddress<<endl;
                            else
                                cout<<"Node " <<i<< "shares
"<<bwuNode[i]<<" of the bandwidth with IP "<<ipAddress<<endl;
                        }
                        tmstmp = -1;
                    }
                    getline(infile, curLine);
        }
        infile.close();
   cout<<"After analysis the IP address have been found to be malicious and hence
are being written to the file blocked_ip_addr"<<endl;
   ofstream fout("blocked_ip_addr");
   for(i=0;i<4;i++){
            bwuNode[i] = pktNode[i] / tpkt;
            string ipAddress = returnipAddr(i);
            if((bwuNode[i] > 0.1) && (i!=0))
                fout<<ipAddress<<endl;
```

```
        }
        fout.close();
}
vector<string> tokenizeStr(string str) {
        vector<string> tokens;
        stringstream iss(str);
        copy(istream_iterator<string>(iss), istream_iterator<string>(),
back_inserter<vector<string> >(tokens));
        return tokens;
}
```

C.2 Script files for dumping log files and filters

// Background.sh

```
#! /bin/sh
# Author: Kaydee
# This will run the tcpdump in the background with following specs
# -C 1000 File size max as 1MB and overwrite again
# -Z kaydee Run it as kaydee user instead of root (for previleges to create filenames,
doesn't work if unspecified)
# -i p3p0 The interface to listen to. The ethernet port connects to this interface
sudo tcpdump -C 1000 -Z kaydee -i p3p0 -w output -W 2
```

// readscript.sh

```
#! /bin/sh
# Author: Kaydee
```

```
# The main command which reads from the dump files, and generates the log
files.The awk script is used for input formatting to the cpp file.
#cleaning up old files

rm log

tcpdump -nnq -ttttt -r output0 'tcp port 5001'|awk 'BEGIN{}{split($3, ip, "."); split($1,
ts, ":");ts_int=ts[3]+ts[2]*60+ts[1]*3600; if($7>0)print ts_int"
"ip[1]"."ip[2]"."ip[3]"."ip[4]" "$7}END{}' >log

tcpdump -nnq -ttttt -r output1 'tcp port 5001'|awk 'BEGIN{}{split($3, ip, "."); split($1,
ts, ":");ts_int=ts[3]+ts[2]*60+ts[1]*3600; if($7>0)print ts_int"
"ip[1]"."ip[2]"."ip[3]"."ip[4]" "$7}END{}' >>log

# The cpp file which analyzes the packet and creates a file called as blocked_ip_addr
# replace 192.168.1.199 with the ip address of the rich user
# 1 is the time unit you want to display the calculations
# NOTE: The program will terminate after reading the entire log file, and will base its
output on the entire duration
./a.out log 192.168.1.199 1

# NOTE: Needs root and ip_blocker.sh is a basic file which lists the ip in to the
firewall.
./ip_blocker.sh
```

C.3 Sample Iperf Output

```
------------------------------------------------------------
Client connecting to 192.168.1.100, TCP port 5001
TCP window size: 16.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.101 port 40432 connected with 192.168.1.100 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-01.0 sec  11.4 MBytes  95.4 Mbits/sec
[  3]  1.0-02.0 sec  11.2 MBytes  94.4 Mbits/sec
```

[3] 2.0-03.0 sec 11.4 MBytes 95.4 Mbits/sec

[3] 3.0-04.0 sec 11.4 MBytes 95.4 Mbits/sec

[3] 4.0-05.0 sec 11.2 MBytes 94.4 Mbits/sec

[3] 5.0-06.0 sec 11.1 MBytes 93.3 Mbits/sec

[3] 6.0-07.0 sec 11.2 MBytes 94.4 Mbits/sec

[3] 7.0-08.0 sec 11.6 MBytes 97.5 Mbits/sec

[3] 8.0-09.0 sec 11.2 MBytes 94.4 Mbits/sec

[3] 9.0-10.0 sec 11.5 MBytes 96.5 Mbits/sec

[3] 10.0-11.0 sec 11.4 MBytes 95.4 Mbits/sec

[3] 11.0-12.0 sec 11.2 MBytes 94.4 Mbits/sec

[3] 12.0-13.0 sec 10.6 MBytes 89.1 Mbits/sec

[3] 13.0-14.0 sec 10.6 MBytes 89.1 Mbits/sec

[3] 14.0-15.0 sec 10.6 MBytes 89.1 Mbits/sec

[3] 15.0-16.0 sec 10.6 MBytes 89.1 Mbits/sec

[3] 16.0-17.0 sec 10.6 MBytes 89.1 Mbits/sec

[3] 17.0-18.0 sec 10.6 MBytes 89.1 Mbits/sec

[3] 18.0-19.0 sec 10.6 MBytes 89.1 Mbits/sec

[3] 19.0-20.0 sec 10.6 MBytes 89.1 Mbits/sec