Integration of Linux TCP and Simulation: Verification, Validation and Application

Songrith Kittiperachol, Zhili Sun, Haitham Cruickshank

Centre for Communication Systems Research
Faculty of Electronic and Physical Science
University of Surrey
Guildford, Surrey, GU2 7XH, United Kingdom

Abstract—**Network simulator has been acknowledged as one of the most flexible means in studying and developing protocol as it allows virtually endless numbers of simulated network environments to be setup and protocol of interest to be fine-tuned without requiring any real-world complicated and costly network experiment. However, depending on researchers, the same protocol of interest can be developed in different ways and different implementations may yield the outcomes that do not accurately capture the dynamics of the real protocol. In the last decade, TCP, the protocol on which the Internet is based, has been extensively studied in order to study and re-evaluate its performance particularly when TCP based applications and services are deployed in an emerging Next Generation Network (NGN) and Next Generation Internet (NGI). As a result, to understand the realistic interaction of TCP with new types of networks and technologies, a combination of a real-world TCP and a network simulator seems very essential. This work presents an integration of real-world TCP implementation of Linux TCP/IP network stack into a network simulator, called INET. Moreover, verification and validation of the integrated Linux TCP are performed within INET framework to ensure the validity of the integration. The results clearly confirm that the integrated Linux TCP displays reasonable and consistent dynamics with respect to the behaviors of the real-world Linux TCP. Finally, to demonstrate the application of the INET with Linux TCP extension, algorithms of other Linux TCP variants and their dynamic over a large-bandwidth long-delay network are briefly presented.**

Keywords- Linux TCP, Network Stack, INET, Integration, Verification, Validation, Application

I. INTRODUCTION

A combination of a Transmission Control Protocol (TCP) implementation and a simulation framework allows realistic evaluation of TCP with less complication and lower cost than a network experiment. This paper presents an integration of an existing TCP based on Linux implementation and OMNeT++ (v3.3) simulation framework [1]. OMNeT++ is a well-known open-source Discrete Event Simulation (DES) environment. It utilizes a modular component design architecture where small and simple components, programmed in C++, can be assembled into larger and more complex modules by using OMNeT++ built-in NEtwork Description (NED) language. In addition, OMNeT++ features Graphical User Interface (GUI) that enables the design and the execution to be visualized and animated. Most importantly, OMNeT++ presents simulation kernel support that allows other kernels to be embedded into the framework. Although OMNeT++ itself is not a network simulator, it is gaining widespread acceptance in scientific communities as a network simulation platform due to its generic and flexible framework. Many open-source network simulator models have been developed with OMNeT++, for examples, INET Framework [1] in the field of the Internet, Mobility Framework [1] in mobility and ad-hoc networks, Castalia [2] wireless sensor network, Chsim [3] in wireless channel modeling and OverSim [4] in Peer-to-Peer network.

INET Framework (v20062010) was developed mainly for the simulation of the Internet. Although INET maintains a number of protocol implementations, only the 4th layer protocols or the transport layer protocols are of interest. In general, the responsibility of the transport layer protocols is to establish end-to-end data connections between peers in an either reliable or unreliable manner. For a reliable service, TCP is primarily used. TCP offers byte-oriented in-order-delivery data transport services. As documented by INET, two TCP variants, i.e. *Tahoe* and *Reno*, already exist in the framework. However, study [5] shows that both variants do not perform well when they are deployed in the network with high data loss rate. Thus, the use of *Tahoe* and *Reno* is no longer practical in the modern Internet where data loss rate can be high. As opposed to the original Internet, i.e. slow networks and best-effort services, Next Generation Internet (NGI) aims at offering high speed broadband wired or wireless Internet access and multiservice applications of different qualities. Undoubtedly, NGI is growing beyond the original design of both *Tahoe* and *Reno* [6] where network reliability and best effort service are often assumed. The main weakness of both variants

Manuscript received November 14, 2008; revised April 28, 2009; accepted June 1, 2009

S. Kittiperachol, Centre for Communication Systems Research, Faculty of Electronic and Physical Science, University of Surrey, Guildford, Surrey, GU2 7XH, UK (phone: +44-(0)1483-683465; fax: +44-(0)1483-300803; email: s.kittiperachol@surrey.ac.uk).

© 2009 ACADEMY PUBLISHER
doi:10.4304/jnw.4.9.819-836

lies in the ineffectiveness of their fast recovery algorithm when having to deal with multiple data losses from the same transmission window. In order to mitigate this deficiency, the fast recovery algorithm is modified and the latest standard TCP known as *NewReno* is defined [7].

From the research point of views, it sounds more logical and more productive to concentrate on the latest standard protocol, *NewReno*, rather than its previous incarnations. With INET as a simulation tool, the addition of the missing standard TCP is necessary. On one hand, taken from the *NewReno* description, a new TCP module can be coded into the INET. Although this straightforward approach allows quick and flexible realization, the resulting module may not contain enough details to produce accurately the dynamics of the real-world protocol, most likely due to high level of abstraction, variations in the implementation or the lack of certain features. On the other hands, an existing protocol implementation, after some changes, can be directly ported into the INET. Although this latter approach introduces development complexity due to architecture differences of the protocol and the network simulator, the benefit is very important since it allows a real-world protocol to be studied within any given simulated network environments without having to rely on costly and complex network experiment setup. In addition, by using real-world code, researcher can get acquainted to real-world implementation. Understanding the importance of the real-world code, this paper follows the porting approach.

This paper is organized as follow. Section II states related work whereas section III overviews Linux TCP/IP network stack and discusses TCP functionalities and enhancements. Section IV outlines integration methodology, followed by simulation setup in section V. Then, verification, validation and application of the integrated tool, i.e. INET with Linux TCP extension, are illustrated respectively in section VI, VII and VIII. Finally, section IX presents conclusions and the future works.

II. Related Work

Acknowledging the importance of real-world codes, other independent developments on porting TCP/IP network stack into a network simulation platform have been promoted by several active research groups; examples of such works are OppBSD [8] and A Linux TCP Implementation [9].

A. OppBSD

OppBSD is a network simulator, developed by Institute of Telemetics, university of Karlsruhe, Germany [10]. It allows the actual FreeBSD kernel (v6.2) working in the OMNeT++ platform as a simulation model. In addition, it features dual stacks IPv4 and IPv6. However, FreeBSD only supports one TCP implementation and does not have necessary interface that allows simple addition of congestion control algorithms. Moreover, OppBSD only sports point-to-point and Ethernet-like connections; restricting the simulated environments to wired network only. Although OppBSD is provide a precise emulation of FreeBSD, another integrated network simulator with more selections regarding TCP variants and network technologies is required.

B. A Linux TCP Implementation

A Linux TCP Implementation (NS2 TCP-Linux), a part of NS2 [11] main distribution, is developed and maintained by Network Laboratory, California Institute of Technology, USA. By loosely following Linux kernel (v2.6.22), thirteen additional Linux TCP variants can be run in NS2 [12, 13]. However, data processing with respect to TCP enhancement and modifications needs to be individually implemented and handled by NS2 since TCP-Linux only includes congestion control. Again, the true TCP dynamics may not be produced here due to lack of features or over simplification. Although NS2 Linux-TCP provides more choices on TCP variants and network technologies, another integrated network simulator is needed since Linux-TCP only includes main algorithms, i.e. slow start and congestion avoidance while excluding the majorities of the real-world codes of Linux kernel that directly controls the behaviors of real-world TCP.

III. Linux TCP/IP Network Stack

Linux TCP/IP network stack [14] is found to be one of the most suitable sources for a real-world TCP implementation as firstly it is a part of the Linux Operating System (OS) or the Linux kernel; secondly it has been widely used by many research communities; and thirdly it already contains the standard TCP, *NewReno*. In addition, Linux TCP/IP network stack has a well-defined congestion control interface where new TCP algorithms can be adopted easily and very well matches the INET software architecture

Like *Tahoe* and *Reno*, *NewReno* employs a window-based algorithm to govern the *flow control*, and thus transmission rate. In general, TCP can send more data if the following condition is satisfied.

$$in_flight < min(cwnd, awnd) \quad (1)$$

where in_flight is the estimated number of outstanding packets, i.e. packets that has been transmitted but not yet acknowledged, *cwnd* is the congestion window, i.e. the amounts of packets that can be handled by a network, and *awnd* is the advertised window, i.e. the amounts of packets that can be handled by a receiver. In addition, the operation of TCP is defined by four algorithms; slow start, congestion avoidance, fast retransmit and fast recovery.

A. The Standard TCP

Slow Start

Slow start is an algorithm that is used in one of the following circumstances; at the beginning of connection,

© 2009 ACADEMY PUBLISHER

after a long idle period and after a retransmission timeout. The main goal of the slow start is to quickly probe network of unknown conditions for available bandwidth that can be utilized. Since TCP is a closed-loop protocol, the growth of congestion window mainly depends on the reception of new acknowledgement. For each new acknowledgement,

$$cwnd \leftarrow cwnd + \alpha \quad (2)$$

where α is the additive increase parameter and $\alpha = 1$ in the cases of *Tahoe, Reno* and *NewReno*. Therefore, congestion window is approximately doubled every round trip time. The slow start remains active provided that the congestion window does not exceed the slow start threshold, $ssthresh,$ i.e.

$$cwnd < ssthresh \quad (3)$$

After that, congestion avoidance begins.

Congestion Avoidance

Congestion avoidance is another algorithm that is used in order to continue probing the network for more bandwidth but at a much slower rate when compared to the slow start. On the reception of new acknowledgement,

$$cwnd \leftarrow cwnd + \alpha/cwnd \quad (4)$$

and $\alpha = 1$. The congestion window is approximately increased by one for every round trip time. In this manner, the transmission rate can be increased steadily without inflicting any sudden changes in network conditions. TCP remains in the congestion avoidance until data losses are detected due to congested network. After which, TCP enters a transient phase in order to recover lost data.

In addition to the flow control given in (1), TCP relies on the error control in order to provide a reliable connection. However, the reliability of the TCP is in the sense that lost data will eventually be detected and retransmitted.

Fast Retransmit

In loss events, TCP reacts in a different way depending on the variants and this is where similarity between *Tahoe* and the others ends. On one hand, *Tahoe* relies only on retransmission timer to detect data losses. The timer will timeouts if the acknowledgement of a packet is not received after a certain time period, based on the measured round trip time [15]. As soon as the timeout occurs, the lost data as indicated by the left window or the highest sequence number of data transmitted but not yet acknowledged, i.e. snd_una, is retransmitted and the congestion window is reset,

$$cwnd \leftarrow 1 \quad (5)$$

On the other hand, *Reno* and *NewReno* uses retransmission time as a last resource. Rather, *Reno* and *NewReno* use fast retransmit algorithm to speed up the loss detection process by inferring certain numbers, typically three, of duplicate acknowledgements, i.e. an acknowledgement that does not advance snd_una, as an indication of network congestion and thus data losses. On the reception of three duplicate acknowledgements, data, starting from *snd_una*, is assumed lost and retransmitted.

Fast Recovery

After the fast retransmit, fast recovery becomes active. Then, the slow start threshold and the congestion window are reduced as follow,

$$ssthresh \leftarrow \beta \cdot cwnd \quad (6)$$

$$cwnd \leftarrow ssthresh + 3 \quad (7)$$

where β is the multiplicative decrease factor and $\beta = 1/2$ in the cases of *Reno* and *NewReno*. During fast recovery, the congestion window increases per the reception of duplicate acknowledgement as follow,

$$cwnd \leftarrow cwnd + 1 \quad (8)$$

Now, this is the place where the similarity between *Reno* and *NewReno* ends. For *Reno*, congestion window continues to increase until a new acknowledgment arrives. After that, congestion window is reset as follow,

$$cwnd \leftarrow ssthresh \quad (9)$$

and *Reno* continues in congestion avoidance.

Unlike *Reno*, *NewReno* records the highest transmitted sequence number in $high_data$ and classifies two types of the new acknowledgement, i.e. *full* and *partial*. A new acknowledgement advances snd_una; However, a full one also cover $high_data$ but a partial one does not. If a partial acknowledgment arrives, the congestion window is deflated by the amounts equal to the number of data being covered by that partial acknowledgement and fast recovery continues since partial acknowledgement implies more data losses. If a full acknowledgement arrives, fast recovery ends and congestion window is reset as in (9). After all losses are recovered, TCP resumes in congestion avoidance. It is worth mentioning that during fast recovery, more data are allowed to be transmitted if and only if (1) is satisfied so that the network does not suffer extended congestion.

In summary, TCP is the 4th layer protocol whose main responsibility is to establish end-to-end data connection for a reliable in-order byte-oriented data delivery service. Most importantly, TCP provides flow control and error control for network by reducing its transmission rate when the network is heavily congested and retransmitting lost data when there are data losses. In addition, due to the additive increase of the congestion avoidance and the multiplicative decrease of the fast recovery, the operation of TCP can be characterized as an Additive Increase Multiplicative Decrease scheme or AIMD. Finally, it is evident that TCP performs extremely well in the traditional Internet where connection is relatively slow and round trip time delay is in a few ten-milliseconds as TCP strongly remains the most dominant protocol over the Internet up until today.

© 2009 ACADEMY PUBLISHER

kind	length	option field
1 octet	1 octet	length – 2 octets

Fig. 1 TCP option field

NOP	NOP	8	10
TS Value (TSval)			
TS Echo Reply (TSecr)			

Fig. 2 Timestamp option

NOP	3	3	shift.cnt

Fig. 3 Window scaling option

NOP	NOP	5	*var*
Left Edge of the 1st block			
Right Edge of the 1st block			
...			
Left Edge of the nth block			
Right Edge of the nth block			

Fig. 4 Selective acknowledgement option

B. The Linux TCP

In addition to the standard TCP algorithms, Linux TCP is equipped with several options and extensions [16-21]. TCP option can be used if and only if the required options are negotiated at the connection setup. TCP option is indicated by the following structure; one octet *kind* field followed by one octet *length* field, and followed by (*length* - 2) octets of option fields as seen in Fig. 1. In addition, no operation or NOP is used to align the option to the four-octet boundary.

Timestamp Option

Timestamp option (TSopt) is indicated by kind (8) and length (10) as seen in Fig. 2. With timestamp option, transmitted time can be imprinted in the TCP option field. The transmission time of a data segment is recorded in the *TSval* field. On the reception of data segment, the recorded time in *TSval* is copied over to the *TSecr* field and the transmission time of an acknowledgement is recorded over in the *TSval* field. Once the acknowledgement arrives, a more accurate round trip time can easily be measured by finding the difference between *TSecr* and *TSval*. In addition, a more refined retransmission timeout is a byproduct of a more accurate round trip time measurement.

Window Scaling Option

Window scaling option (WSopt) is indicated by kind (3) and length (3) as seen in Fig. 3. With window scaling option, the window size based on the 16-bit *window* field of the TCP header, can be extended to a 30-bit value by scaling the *window* field. For the sending window, snd_wnd,

$$snd_wnd \leftarrow window \ll shift.cnt \tag{10}$$

and for the receiving window, rcv_wnd,

$$window \leftarrow rcv_wnd \gg shift.cnt \tag{11}$$

This option allows more data to be sent in one transmission window. Thus, higher data rate can be achieved, particularly in a large-bandwidth long-delay network.

Selective Acknowledgement Option (SACK)

Selective acknowledgement option (SACK) is indicated by kind (5) and length (*var*) as seen in Fig. 4. The length of SACK is varied since it depends on the numbers of blocks that will be set in the option field, i.e. $4 \cdot n + 2$. With SACK option, non-contiguous blocks of data that have successfully been received and queued at the receiver can be sent back to the sender. These gaps can be filled by the retransmission or the late arrival of the packet. On the reception of the acknowledgement with SACK blocks, the non-contiguous blocks of data that are queued at the receiver are reproduced. By utilizing this extra information, lost data can be quickly and accurately determined.

Large Initial Window

Large Initial Window (IW) is an extension that speeds up slow start, by increasing the transmission window size at the beginning of the connection according to

$$snd_wnd \leftarrow min\big(4 \cdot MSS, max(2 \cdot MSS, 4380)\big) \tag{12}$$

Depending on the given size of the Maximum Segment Size (*MSS*), the initial congestion window can be four segments at most. With large initial window, the transmission window can be opened more quickly during slow start.

Delayed Acknowledgement

Delayed acknowledgment (DACK) is an extension that controls the rate at which acknowledgements are transmitted in the return channel. Customarily, the receiver immediately sends an acknowledgement for every data segment that is received. With DACK, the receiver delays the transmission of an acknowledgement up to a certain time period, typically 200 ms. If the receiver does not receive more data within the given time period, the pending acknowledgement is sent. If the receiver receives the next data within that time period, the pending acknowledgement is discarded and the new one corresponding to the latest data is transmitted. However, if the receiver receives out-of-order data, it will cancel DACK and immediately transmit a duplicate acknowledgement. Therefore, single acknowledgment can be used to cover two data segments that are successfully received and bandwidth usage in the return channel can be reduced up to two folds. However, DACK also affects congestion window growth rate since the growth rate, as given in (2) and (4), depends on the numbers of the acknowledgements that are received, i.e. an acknowledgement counting scheme.

© 2009 ACADEMY PUBLISHER

Appropriate Byte Counting

Appropriate byte counting (ABC) is an extension that controls the amounts at which congestion window increases. Instead of being based on the acknowledgement-counting scheme, a byte-counting scheme is used. The byte-counting scheme increases congestion window relative to the number of bytes covered by an acknowledgement. In other word, the additive increase parameter is changed to

$$\alpha \leftarrow max(\lfloor byte/MSS \rfloor, 2) \quad (13)$$

where $\lfloor x \rfloor$ indicates the largest integer that is smaller than x and the increase parameter is limited to 2 to prevent large data burst. As a result, ABC can mitigate the impact of DACK and reduce the effect of lost acknowledgements that reduces the growth rate of congestion window.

Fast Retransmit with SACK

In addition to using the three-duplicate-acknowledgement scheme, the sender can detect data losses more quickly by utilizing the non-contiguous blocks of data that are derived from SACK. blocks. If the cumulative size of the gaps, stating from a sequence number, is equivalent to at least three segments, the segment beginning with that sequence number is considered lost and retransmitted. With SACK, the sender can quickly detect data losses by less than three duplicate acknowledgements

Fast Recovery with SACK

The use of SACK to accelerate loss detection process in fast retransmit is also extended to fast recovery. After each retransmission, the sender marks the retransmitted segment and virtually fills the non-contiguous blocks of data with that segment in order to ensure that the sender will not retransmit the same packet more than once. For every duplicate acknowledgement received, the sender can search the non-contiguous blocks of data, after being updated, for possible data losses. Therefore, rather than having to wait another round trip time for the acknowledgement of the previous retransmission to arrive, subsequent data losses can quickly be found and retransmitted. With SACK, the sender can retransmit lost data with more efficiency and complete loss recovery in shorter time period. *Nonetheless, the slow start threshold is still reduced as in (6).*

Fast Recovery with Forward Acknowledgement

Forward acknowledgement (FACK), i.e. rate halving, is an extension that regulates the rate at which new data are transmitted. With FACK, a new data segment can be sent for every two duplicate acknowledgements received during loss recovery regardless to the control parameters given in (1). In other word, rate halving decouples the loss recovery from the flow control. Because of this independence, the self-clocking behavior, i.e. acknowledgement feedback, can be better maintained and needless retransmission timeout can be avoided. In addition, congestion window is deflated per two duplicate acknowledgements as follow

$$cwnd \leftarrow cwnd - 1 \quad (14)$$

Therefore, the congestion window is reduced approximately to half of the window prior to the reduction. After all losses are recovered, either slow start or congestion avoidance resumes depending on the final congestion window and the new slow start threshold.

TABLE I
LINUX TCP INTERFACE

Data Structure	Definition/Declaration	Interface
tcp_prot	/net/ipv4/tcp_ipv4.c	command interface
tcp_protocol	/net/ipv4/af_inet.c	data interface
tcp_hdr	/net/tcp.h	TCP header
Ip_hdr	/net/ip.h	IP header
timer_list	/linux/timer.h	timer interface
sock	/linux/sock.h	socket
sk_buff	/linux/skbuff.h	data segment

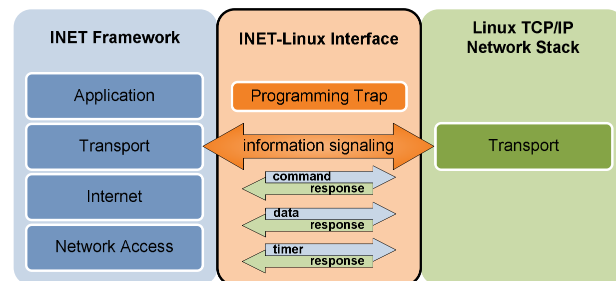

Fig. 5 Overview of interface-based integration methodology

IV. INTEGRATION METHODOLOGY

According to the layering principle, the transport layer only communicates to its adjacent layers, i.e. to the Internet on the lower layer and to the application on the upper layer. The integration methodology loosely follows this principle. By utilizing certain data structures and definitions, given in TABLE I, INET can access internal parameters as well as internal functions of Linux TCP/IP network stack. Thus, an integration of the real-world Linux TCP and INET becomes possible, i.e. INET with Linux TCP extension. An interface-based integration approach is proposed and is illustrated in Fig. 5. Hereafter, the integrated Linux TCP is to be referred to as *LinuxTCP* for short.

Note that although the given methodology is specific to INET framework, it can also be applied to other network simulators provided that the simulators can exploit the TCP interface in TABLE I.

At the transport layer, information signaling is initiated by INET, based on the events triggered by the simulation. The INET-Linux interface functions primary as an input-output interface in which arguments; ‘*data*’, ‘*command*’ and ‘*timer*’, are passed into the stack. The stack processes the given arguments and completes by either returning data output, invoking designated commands, calling relevant timer functions or in any combination. As the

© 2009 ACADEMY PUBLISHER

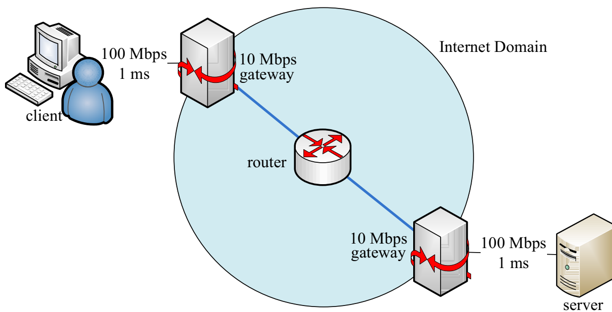

Fig. 6 Simulation Scenario

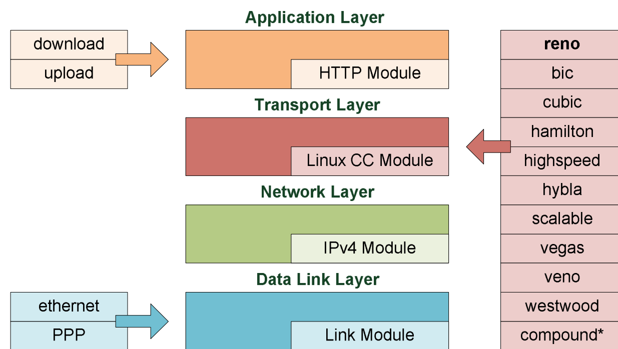

Fig. 7 INET Node with Linux TCP extension

stack is in fact a real-world code, it can be assumed to have correct TCP/IP functionality. Due to this assumption, this interface-based integration scheme extremely relies on validity and integrity of the input arguments, therefore making *LinuxTCP* highly sensitive to any '*inappropriate'* arguments. Thus, it is also responsibility of the INET-Linux interface to ensure validity and integrity of the input arguments. By using programming traps,

$$ASSERT(cond) \rightarrow if(cond\ fail)\ then \text{ terminate simulation} \quad (15)$$

the interface then can check the input arguments against any given conditions before passing them into the stack. TCP/IP header, i.e. *tcp_hdr* and *ip_hdr*, of data packet, i.e. *sk_buff*, is always checked against known INET connection parameters (source/destination IP addresses and source/destination TCP ports). This is to first validate if the transmitted or received packets in fact belong to the correct TCP flow or not and second to verify if data packet has been compromised by invalid memory access caused by the simulation or not. In addition, without loss of generality, other extensive modifications that make the integration possible will not be discussed in this paper due to irrelevancy to TCP/IP related functionalities and limited spaces.

V. SIMULATION SETUP

Similar to the previous work [22], the simulated network consists of three parts; a client, a server and an Internet domain, and it is illustrated in Fig. 6. Both client and server are connected to Internet gateways via a 100-Mbps Ethernet connection with 1-ms delay and no errors. This connection represents a typical Local Area Network (LAN). Along the client-server path, a high speed 10-Mbps backbone link is provided. This link can be viewed as a Wide Area Network (WAN) connection. Every WAN interface has a simple drop tail queue and the Maximum Transmission Units (MTU) is 1500 bytes, i.e. Ethernet friendly. In addition, the buffer size (*IFQ*) of the WAN interfaces, the round trip time (*RTT*) and the bit error rate (*BER*) of the backbone connection are the controlled parameters.

In addition, Fig. 7 describes INET node with Linux TCP extension. Each INET node can be equipped with *HTTP module* at the application layer for download and upload services, with *Linux Congestion Control (CC) module* at the transport layer for different Linux TCP variants, with *IPv4 module* at the network layer for simple IP addressing and forwarding, and with *link module* at the data link layer for different access technology. Accordingly, the client and the server will have all four modules while the gateway and the router only have two modules from the lower two layers.

A file download over the Internet scenario is considered. Specifically, a web client requests a 700-MB data download, i.e. a CD containing a movie file, from a web server over the Internet. In addition, *RTT* and *BER* are configured as follow.

$$RTT \in \{0.1, 0.2, \cdots, 1.0\} \text{ in seconds}$$

$$BER \in \{0, 10^{-9}, 10^{-8}, 10^{-7}, 10^{-6}\}$$

A large data file is used to investigate long-term behavior of TCP, *RTT* is used to characterize network having different bandwidth delay product (*BDP*) and *BER* is used to describe connection having different data loss rate. In addition, *IFQ* is setup such that it is approximately equal to the bandwidth delay product of the given network, i.e. for a fixed 10-Mbps bandwidth, *IFQ* of 84 or (1×84) packets is for the network with 100-ms *RTT*, 168 or (2×84) packets for the network with 200-ms *RTT*, and so on. These configurations are used for the verification process.

VI. VERIFICATION OF *LINUXTCP*

According to [23] and [24], model verification is referred to as '*building the model right*' which generally means if the model in question is working properly or not. Based on the dynamic testing, the model in question has to be executed under various conditions and results are examined in order to ensure the accuracy of the implementation of the model. Verification of *LinuxTCP* follows this guideline.

Based on the two-tuple (*RTT* and *BER*) parameters, there are 50 unique simulation setups. Together with ten different random sequences numbers that are applied to each setup, a total of 500 simulations are run and used in the verification process. Furthermore, programming traps, described in (15), are systematically inserted along the INET-Linux interface to catch any possible

© 2009 ACADEMY PUBLISHER

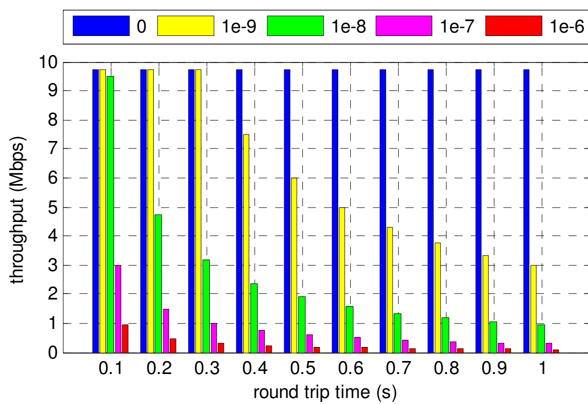

Fig. 8 Long-term average TCP throughput (analytical)

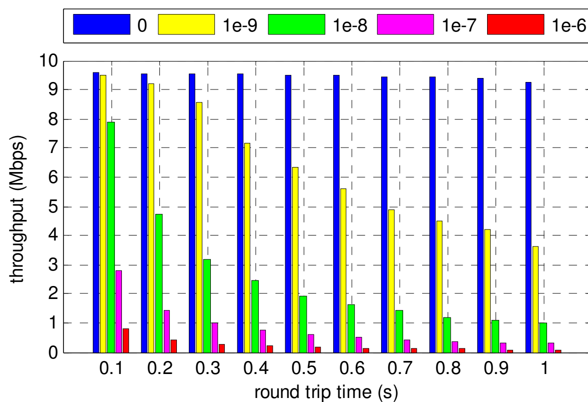

Fig. 9 Long-term average TCP throughput (simulated)

misbehaviors of *LinuxTCP* due to '*improper* input arguments. If any of the assertions is not satisfied, simulation will be terminated and verification will be considered failed.

Study [25] shows that the long-term average throughput of TCP is inversely proportional to the round trip time and the square root of packet lost rate, and is approximated by the following expression,

$$hroughput = min\left(\frac{MSS}{RTT}\cdot\sqrt{\frac{3}{2bp}},\frac{WND_{max}}{RTT}\right) \quad (16)$$

where *MSS* is the maximum segment size, *RTT* is the round trip time, *b* is the number of data segments being covered by an acknowledgement, *p* is the average packet loss rate and WND_{max} is the maximum congestion window size.

Fig. 8 displays long-term average TCP throughput from the analytical expression given in (16). The result shows that in an ideal case where there is no bit error, the average throughput remains unchanged regardless of the *RTT* (and *BDP*). This is because throughput is limited by the 10-Mbps link and the buffer size is in the same order of magnitude as *BDP*. However, in a more realistic case where *BER* exists, the average TCP throughput drops considerably as *RTT* (and *BDP*) or *BER* increases. Even in the case where the *BER* is very low, i.e. 10^{-9}, TCP can achieve 10-Mbps throughput only in the network with *RTT* up to 0.3 s. This simulation result confirms that real-world TCP does not perform well in the network with large *BDP* and high *BER*.

Fig. 9 displays long-term average TCP throughput from the simulation, i.e. downloading a 700-MB movie file from a web site. In this case, the average throughput is simply the ratio of the file size to the service waiting time. The results show that the average throughput complies with the 10-Mbps backbone link and decreases as either *RTT* or *BER* increases. The results are not unexpected since respectively shorter *RTT* and lower *BER* implies faster acknowledgement responses and fewer data losses in which they contribute to faster congestion window growth rate.

Of 500 simulation runs, no assertion flag is triggered. In addition, by comparing Fig. 8 and Fig. 9, it is evident that the simulation results agree with the analytical ones. As a result, it can be concluded that *LinuxTCP* is in fact verified.

VII. VALIDATION OF *LINUXTCP*

The verification process, previously discussed in section VI, clearly indicates that *LinuxTCP* behaves reasonably and consistently in the given simulated network environments of INET. However, additional investigations are necessary in order to validate *LinuxTCP* operationally.

Model validation according to [23] and [24] is referred to as '*building the right model*' which generally means that the model in question behaves consistently and accurately with respect to its intended application. In this paper, operational graphic is used as a main validation technique. Thus, the 500-ms and error-free configuration is chosen as a reference scenario. With the 500-ms round trip time, the buffer size is set to 420 packets. This reference scenario can be viewed as a typical geostationary satellite network [26] with two peers. The Internet gateways function as satellite gateways and the Internet router assumes the role of a geostationary satellite. Lastly, the 10-Mbps link implies that the capacities of both forward and return channels are fixed and pre-allocated.

A. Standard Track

To have a better understanding of *LinuxTCP* mechanisms, the congestion window, i.e. *cwnd*, and the sequence number are closely observed. In the error-free network environment, the evolution of congestion window can be best described as a 'perfect' uniformly-spaced saw-tooth like trajectory. The '*up ramp*' and the '*sharp drop*' of the trajectory are products of linear window inflation during congestion avoidance and rapid window deflation during loss recovery. This behavior is quite typical to any TCP that uses an AIMD algorithm. Fig. 10 illustrates a complete congestion window evolution that includes slow start, congestion avoidance, loss detection and loss recovery. Clearly, the result is consistent with the description of the saw-tooth behavior or AIMD.

© 2009 ACADEMY PUBLISHER

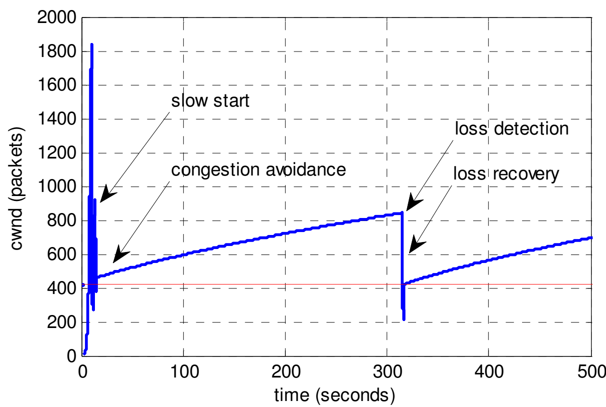

Fig. 10 Congestion window evolution

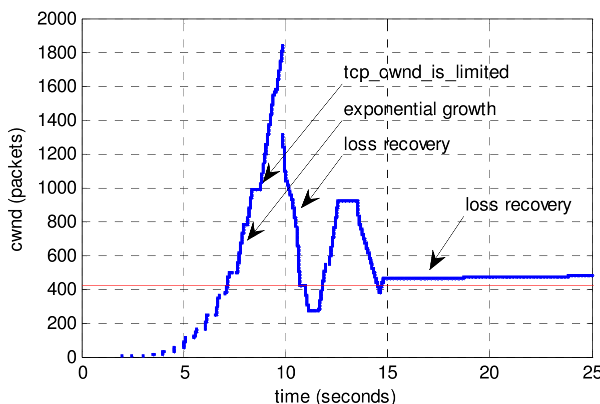

Fig. 11 Congestion window evolution during slow start

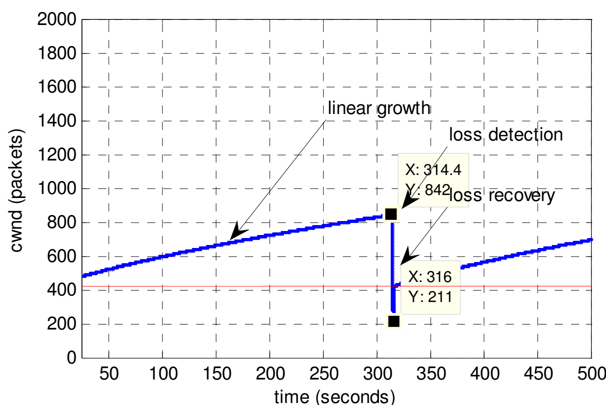

Fig. 12 Congestion window evolution during congestion avoidance

Fig. 11 highlights the evolution of Fig. 10 during slow start and show that congestion window grows exponentially which agrees with (2). In addition, the result uncovers that congestion window does not always increase per reception of an acknowledgement. This '*pause*' behavior is due to the restrictions imposed by *tcp_is_cwnd_limited* function that stops the congestion window from advancing if the current window is much '*larger*' than the numbers of outstanding data. Note that this TCP function is usually left unimplemented in many network simulators.

Fig. 12 highlights the window evolution of Fig. 10 during congestion avoidance and shows that congestion window grows linearly, i.e. additive increase, which agrees with (4). In addition, after a loss event, congestion window decreases sharply, i.e. multiplicative decrease which agrees with (14).

To further consolidate the results, simulation details are discussed. In general, without window scaling option, the maximum window size is limited to 44 (1500-byte) MTU-sized segments. However, the results show that congestion window well exceeds 44 segments, explicitly implying the use of window scaling option. In addition, the reference network, i.e. 10-Mbps link and 500-ms round trip time, provides a rough bandwidth delay product of 420 segments. Together with 420 packets from the buffer, the maximum network capacity is 840 segments. Beyond that, data losses start to occur. By comparison, Fig. 11 shows that congestion window first exceeds 1800, i.e. more than twice the network capacity, before congestion is detected. This is not unusual for congestion window to overshoot the network capacity by two folds since congestion window is doubled every round trip time during slow start. After a few congestion events, the network is properly probed and the slow start threshold is set to a more accurate value. Then, congestion avoidance begins. Fig. 12 shows that congestion window reaches 842 before congestion is detected. This is typical for congestion window to grow slightly above the network capacity by a few segments due to additive increase nature of congestion avoidance. Therefore, it is evident that *LinuxTCP* behaves consistently and accurately with respect to the slow start and the congestion avoidance algorithms.

B. Enhancement Track

Unlike the standard TCP, *LinuxTCP* are augmented with a number of options and extensions in order to improve its performance particularly when having to deal with multiple data losses from the same transmission window and to cope with large-bandwidth long-delay type of network. A number of enhanced mechanisms, i.e. delayed acknowledgement, appropriate byte counting, fast retransmit with SACK, fast recovery with SACK, fast recovery with FACK, and round trip time measurement with timestamp, are investigated in this section.

Delayed Acknowledgement

Delayed Acknowledgement (DACK) is a technique that reduces bandwidth usage in the return link by allowing an acknowledgement to cover two consecutive data segments. Fig. 13 concentrates on the sequence numbers of received acknowledgements with respect to those of transmitted segments and clearly shows that, during slow start, only one acknowledgement is transmitted in response to two in-order data segments that are successfully received.

Appropriate Byte Counting

Appropriate Byte Counting (ABC) is a technique that increases congestion window based on the numbers of full-size segments, i.e. MTU, being covered by each

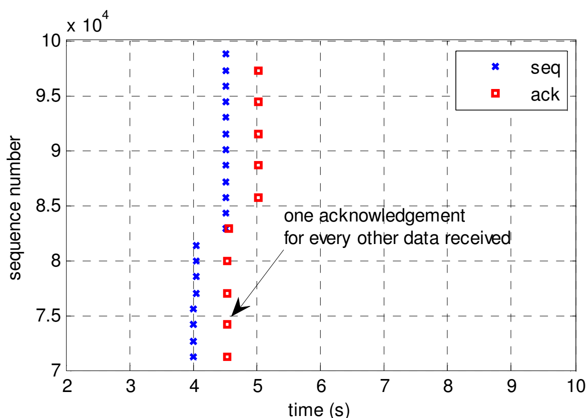

Fig. 13 Delayed acknowledgement

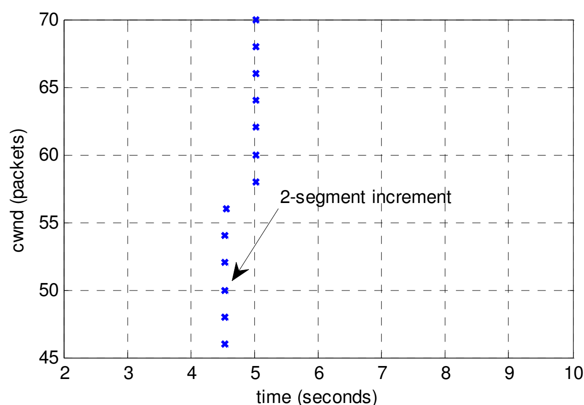

Fig. 14 Appropriate byte counting

acknowledgement given in (13). Fig. 14 concentrates on congestion window during slow start and clearly shows that congestion window increases by two segments instead of one. Note that the same behavior occurs during congestion avoidance and thus without loss of generality it can be omitted. Note that Fig. 13 and Fig. 14 are displayed over the same time but with different *y-axis*.

Fast Retransmit and Fast Recovery with SACK

With SACK option, extended information regarding the reception of non-contiguous blocks data at the receiver can be made available to the sender. By observing these blocks, lost segments can be quickly identified and the numbers of outstanding data can be accurately estimated. Fig. 15 and Fig. 16 concentrate on retransmission of the lost segments in relation to the reception of duplicate acknowledgement. The results clearly show that initial data losses can instantly be detected by the one duplicate acknowledgement (compared to a typical three) and a number of subsequent data losses can be recovered within one round trip time (compared to multiple round trip time). Note that Fig. 15 is a closed-up version of Fig. 16. Clearly, *TCP* with SACK is found to be more robust against data losses.

Fast Recovery with FACK

With FACK, i.e. rate halving, a data segment can be sent without having to follow the flow control given in (1) and congestion window quickly decreases by one

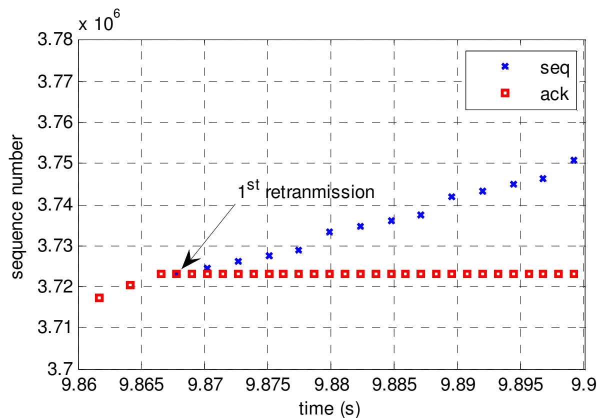

Fig. 15 Fast retransmit with SACK

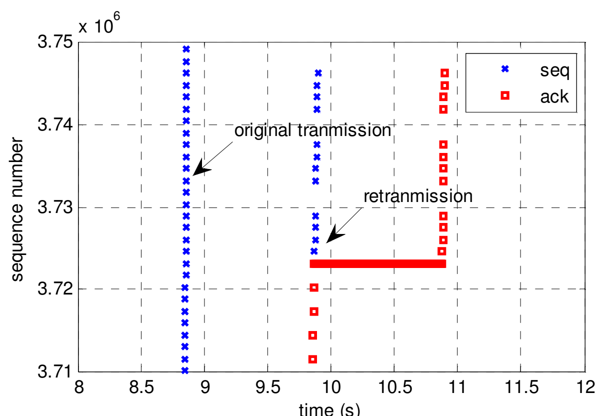

Fig. 16 Fast recovery with SACK

segment per two duplicate acknowledgements received. Fig. 17 and Fig. 18 concentrate on the impacts of rate halving algorithm on the congestion window deflation as well as transmission of new data segments and clearly show that for every two duplicate acknowledgements received, one data segment is transmitted and congestion window is reduced by one. Note that both figures are displayed over the same time period but with different value of the *y-axis*.

Round Trip Time Measurement with Timestamp

In general, round trip time is measured by first recording sequence number and transmission time of a data segment that is being transmitted. After an acknowledgement that covers the recorded sequence number arrives, the difference between the current time and the recorded time measures a coarse round trip time. However, timestamp option allows round trip time to be measured per segments, which in turn provides a more accurate result. Fig. 19 displays round trip time measurement over time. The result clearly shows that the measurement closely resembles the saw-tooth behavior, which is comparable to congestion window evolution found in Fig. 10. The result is not unexpected since the round trip is directly proportional to the queue occupancy, which is in turn proportional to the congestion window. Furthermore, Fig. 19 shows the minimum and the maximum of round trip times to be approximately 506 ms and 1013 ms respectively. Given

© 2009 ACADEMY PUBLISHER

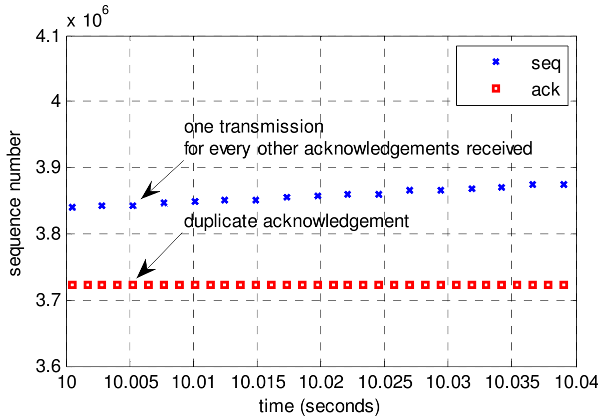

Fig. 17 Fast recovery with rate halving

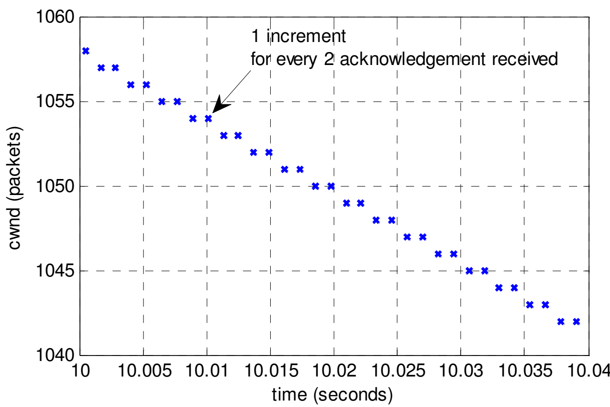

Fig. 18 Fast recovery with rate halving

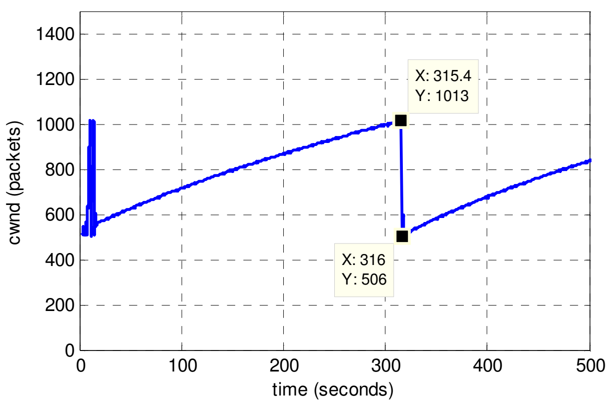

Fig. 19 Round trip time and retransmission timeout

500 ms from the round trip time and an additional of 504 ms from the time required to dequeue 420 packets from the buffer, clearly the result matches the reference network.

According to the simulation results, the behaviors of *LinuxTCP* in the simulated network environments are found to be accurate and consistent since they coincide with the anticipation of how real-world Linux TCP responds to real-world network. As a result, it can finally be concluded that *LinuxTCP* is valid with respect to Linux TCP functionalities and INET simulation platform.

VIII. APPLICATION OF *LINUXTCP*

The ultimate achievement of *LinuxTCP* is the ability to use real-world TCP within network simulator. Thus, studies and evaluations of simulated Internet become more accurate and more consistent with the real world Internet. In addition, *LinuxTCP* acts as a bond between simulator and real world since when researchers become familiar with *LinuxTCP*, they also become familiar with the real-world code of Linux TCP/IP network stack.

Linux TCP/IP network stack in this study is taken from the Linux kernel (v2.6.21) which contains ten TCP variants in addition the standard *reno*, namely *bic*, *cubic*, *highspeed*, *hamilton*, *hybla*, *low priority*, *scalable*, *vegas*, *veno* and *westwood*. The main objective of these variants, except *low priority*, is to improve their performances when facing with an emerging next generation Internet. Details of these TCP variants should be referred to the references [27-34]. Owing to the common Linux congestion control interface, these TCP variants and future TCP variants can be included into the stack with relative ease. For an example, TCP based on Window Vista implementation, also known as *compound* [35, 36] is re-implemented in order to make it compatible with the Linux congestion control interface so that it can be included into the stack for research uses. The source code of *compound* can be obtained at Linux TCP implementation for NS2 website [9].

Owing to the real-world Linux TCP/IP network stack that has been incorporated into INET framework, any new TCP can be implemented in INET environments as if it were in the real network stack. However, this TCP can easily be debugged, corrected and studied in any given simulated network environments. After development and evaluation are completed, this TCP can be put into real-world tests or uses, simply by incorporating it to the network stack and accessing it via the congestion control interface.

Here, the applications of *LinuxTCP* are demonstrated by discussing algorithms and displaying dynamics of different Linux TCP variants in a simulated environment provided by INET. The network setup is still the same one that is used previously in the validation section, i.e. 10-Mbps link, 500-ms round trip time and no random errors.

TCP-*bic*

TCP-*bic* relies on *binary increase* algorithm; i.e. *binary search increase* and *additive increase*. Let S_{max} and S_{min} be the maximum and the minimum window increments that are used in *binary search increase* phase. TCP-*bic* updates its congestion window as follow,

$$cwnd \leftarrow cwnd + bic_inc/cwnd \quad (17)$$

and

© 2009 ACADEMY PUBLISHER

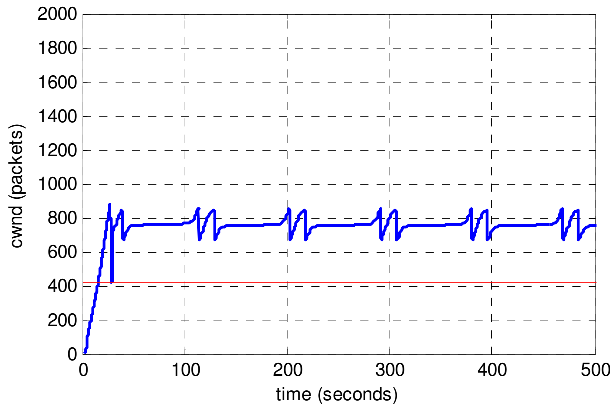

Fig. 20 Congestion window evolution of TCP-*bic*

$$bic_inc = \begin{cases} S_{max} & \text{if } diff \geq S_{max} \\ S_{min} & \text{if } diff \leq S_{min} \\ diff & \text{if otherwise} \end{cases} \quad (18)$$

where bic_inc is the congestion window increment and $diff$ is the '*distance*' between congestion window ($cwnd$) and the maximum window (W_{max}) that is used in *additive increase* mode. In addition, the relation between $cwnd$ and W_{max} divides two modes of operation; i.e. if $cwnd < W_{max}$, then *binary increase* is used and if $cwnd \geq W_{max}$, then *max probing* is used instead. Moreover, $diff$ takes the following form,

$$diff = \begin{cases} \frac{W_{max} - cwnd}{2} & \text{if } binary\ increase \\ cwnd - W_{max} & \text{if } max\ probing \end{cases} \quad (19)$$

Similarly, for any loss events, W_{max} takes the following form,

$$W_{max} \leftarrow \begin{cases} \frac{(1+\beta) \cdot cwnd}{2} & \text{if } binary\ increase \\ cwnd & \text{if } max\ probing \end{cases} \quad (20)$$

If data losses occur during either binary increase or max probing case, slow start threshold and congestion window is reduced as in (6) and (9) with $\beta = 819/1024$. Lastly, TCP-*bic* is set to revert back to *reno* if congestion window size is smaller than 14 packets.

TCP-*bic* controls the congestion window increment by bic_inc which is also limited to the range $[S_{min}, S_{max}]$ or $[1, 16]$ based on Linux implementation. Thus, congestion window can increment up to 16 packets in one round trip time. Fig. 20 shows the dynamic of TCP-*bic*. It is clear that slow start is absent in the figure which is due to the fact that initial slow start threshold is set to 100 packets. In steady state, i.e. at the 100th second, congestion window begins to grow faster as it moves away from W_{max} according to *max probing*, resulting in data losses. Reacting to the loss event, W_{max} is reset as in (20) under *max probing* condition and congestion window quickly deflates. Once all losses are recovered, congestion window continues to grow in *binary increase* mode, eventually causing another data losses. Reacting to another loss event, W_{max} is reset as in (20) under *binary increase* condition and congestion window quickly deflates. After all losses are recovered, congestion window begins to grow faster at the beginning but gradually becomes slower as it approaches W_{max} due to *binary increase*. Afterward, the whole process repeats.

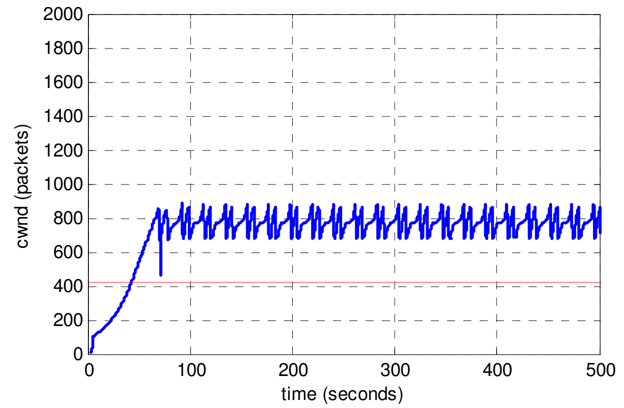

Fig. 21 Congestion window evolution of TCP-*cubic*

TCP-*cubic*

TCP-*cubic* is designed primarily to simplify and enhance the algorithm of TCP-*bic* by replacing *binary increase* with *cubic function*. Therefore, congestion window increases as follow,

$$cwnd \leftarrow C \cdot (t - K)^3 + W_{max} \quad (21)$$

where C is the scaling factor and $C = 41/1024$, t is the elapsed time since the last window reduction in ms, W_{max} is the window size assigned after each loss event and K is the origin of the *cubic function* $K = \sqrt[3]{W_{max}\,\beta/C}$. For any loss events, W_{max} is reset as follow,

$$W_{max} \leftarrow \begin{cases} \frac{(1+\beta) \cdot cwnd}{2} & \text{if } cwnd < W_{max} \\ cwnd & \text{if otherwise} \end{cases} \quad (22)$$

whereas both slow start threshold and congestion window are reduced as in (6) and (9) with $\beta = 717/1024$. Lastly, unlike TCP-*bic*, TCP-*cubic* does not revert back to *reno* in any cases.

TCP-*cubic* control the congestion window increment by means of *cubic function* that uses the elapsed time t (in ms) and the origin K. Fig. 21 shows the dynamic of TCP-*cubic*. It is clear that slow start is absent since initial slow start threshold is set to 100 packets. In the steady state, i.e. at the 100th second, according to *cubic function* as described in (21), congestion window begins to grow faster initially but slower as it moves closer toward the origin K and grows faster as it moves further away from the origin K. Once data losses occur, the origin is reset based on W_{max} as described in (22) and congestion window quickly deflates. Once all losses are recovered, congestion window grows according to *cubic function* until another data losses occurs in which the origin is reset again but with different W_{max}, and congestion window quickly deflates. After all data losses are recovered, the whole process repeats.

© 2009 ACADEMY PUBLISHER

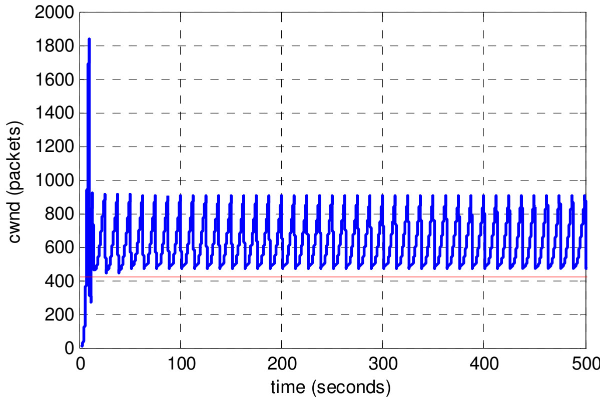

Fig. 22 Congestion window evolution of TCP-*hamilton*

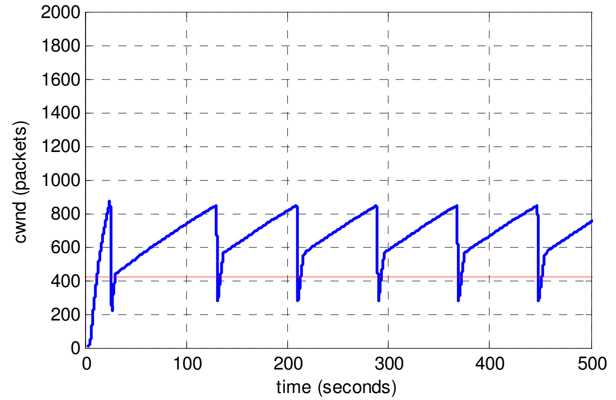

Fig. 23 Congestion window evolution of TCP-*highspeed*

TCP-*hamilton*

TCP-*hamilton* is a variation to the standard congestion control described in (4) and (6). In particular, the additive increase parameter and the multiplicative decreases factor, α and β, are no longer of fixed values but are designed to be adaptive according to network conditions, i.e. low-speed or high-speed regime. For every new acknowledgement that is received, the additive increase parameter changes as follow,

$$\alpha \leftarrow \begin{cases} 1 & \Delta_i \le \Delta^L \\ 1 + 10 \cdot (\Delta - \Delta^L) + \left(\frac{\Delta - \Delta^L}{2}\right)^2 & \Delta_i > \Delta^L \end{cases} \quad (23)$$

where Δ is the elapsed time in ms since the last data loss event and Δ^L is the elapsed time threshold distinguishing low-speed regime from high speed one, and $\Delta^L = 100$ ms. Then, α is scaled by the minimum round trip time so that it becomes less sensitive to round trip time as follow,

$$\alpha \leftarrow \alpha / RTT_{min} \quad (24)$$

but is restricted to $\alpha \in [0.5, 10]$, and then

$$\alpha \leftarrow 2 \cdot (1 - \beta) \cdot \alpha \quad (25)$$

The updates in both (24) and (25) follow the fairness and the friendliness requirements coming from the protocol design. Obviously, TCP-*hamilton* behaves exactly like *reno* in the low speed region. In any data loss events, the multiplicative decrease factor is reset as follow,

$$\beta \leftarrow \begin{cases} 0.5 & \left|\frac{B_{k+1}^{-} - B_k^{-}}{B_k^{-}}\right| > 0.2 \\ \frac{RTT_{min}}{RTT_{max}} & \text{if otherwise} \end{cases} \quad (26)$$

where B_k^{-} is the instantaneous throughput just before the kth loss event, RTT_{max} and RTT_{min} are the maximum and the minimum round trip time seen since the last congestion event. In other words, if the change in the instantaneous throughput between two consecutive loss events is larger than 20 percents, the standard $\beta = 1/2$ is used. Otherwise, a more adaptive choice of β is used and $\beta \in [0.5, 0.8]$ since the ratio RTT_{min}/RTT_{max} can approach unity if network buffer is small. After β is updated, α is also updated as described in (24) and (25) to reflect the recent β. Then, slow start threshold and congestion window is reduced as given in (6) and (9) with the value of β that is derived from (26)

TCP-*hamilton* controls the congestion window increment by using the elapsed time since the last loss event. Fig. 22 shows the dynamic of TCP-*hamilton*. It is clear that slow start is still effective due to window overshoot at the beginning of connection. In the steady state, i.e. at the 100th second, after each loss event, congestion window grows faster and faster as elapsed time becomes longer and longer until data losses occur at which point congestion window begins to deflate quickly. After all data losses are recovered, the whole process repeats.

TCP-*highspeed*

TCP-*highspeed* is a TCP variant that is derived directly from *reno* algorithms; slow start and congestion avoidance as described in (2) and (4), and fast recovery as described in (6). However, the main difference is that TCP-*highspeed* utilizes a more progressive window increment scheme. For each new acknowledgement received, the additive parameter and the multiplicative factor are updated,

$$\alpha \leftarrow f_1(cwnd) \quad (27)$$

$$\beta \leftarrow f_2(cwnd) \quad (28)$$

where $f_1(\cdot)$ and $f_2(\cdot)$ are two increasing functions that map congestion window to an integer, i.e. as congestion window increases, these values also increase. However, if congestion window is smaller than 38, α and β take the same default values as *reno* does, i.e. $\alpha = 1$ and $\beta = 0.5$.

TCP-*highspeed* controls the congestion window growth rate by allowing the increment to increase with congestion window. Fig. 23 shows the dynamic of TCP-*highspeed*. The result shows that slow start increases congestion window very quickly; however, it does not cause window overshoot. This is due to the fact that congestion window increases so quickly that connection pipe is fully filled and data losses become gradual rather than abrupt. In the steady state, i.e. at the 100th second,

© 2009 ACADEMY PUBLISHER

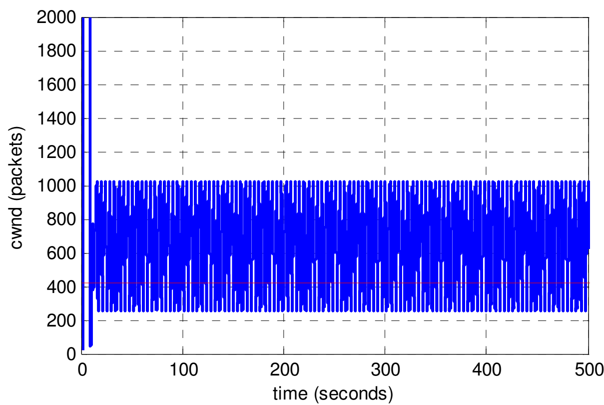

Fig. 24 Congestion window evolution of TCP-*hybla*

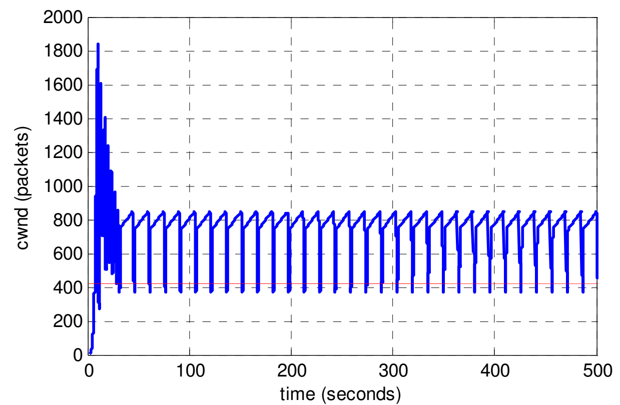

Fig. 25 Congestion window evolution of TCP-*scalable*

congestion window continues to steadily grow during congestion avoidance until data losses occur at which point congestion window begins to deflate quickly. After all data losses are recovered, slow start (or congestion avoidance) resumes if congestion window is smaller than (or larger or equal to) slower start threshold. Afterward, the entire process repeats.

TCP-*hybla*

TCP-*hybla* is another variation to the standard congestion control in (4) and (6). In particular, the additive increase parameter and the multiplicative decrease factor, α and β become dependent on normalized round trip time ρ which is defined as follow,

$$\rho = max(RTT_{min} / RTT_0, 1) \tag{29}$$

where RTT_{min} is the minimum round trip time seen during the connection lifetime and RTT_0 is the reference round trip time which is typically set to 25 ms. In the case of new RTT_{min} is found, α changes according to the mode at which TCP-*hybla* operates; i.e. slow start,

$$\alpha = 2^{\rho} - 1 \tag{30}$$

and congestion avoidance

$$\alpha = \rho^2 \tag{31}$$

Cleary, exponential growth in (30) is much faster than geometric growth in (31). In the event of data losses, slow start threshold and congestion window are set based on the standard algorithms described in (6) and (9) with $\beta = 1/2$. Lastly, the minimum value of ρ derived in (29) is 1 in order to ensure that TCP-*hybla* falls back to *reno* if the round trip time is less than 25 ms, i.e. the reference round trip time.

TCP-*hybla* controls the congestion window increment by allowing the increment to grow with the ration of elapsed time to the reference time. Fig. 23 shows the dynamic of TCP-*hybla*. It is clear that slow start increases congestion window considerably quickly. In this setup, $\rho = 8$, and thus $\alpha = 127$ in slow start and $\alpha = 64$ in congestion avoidance as described by (30) and (31) respectively. Even with very high window increment, window overshoot does not occur. This is due to the fact that congestion window increments so fast that connection pipe is fully filled and data losses become gradual instead of sudden. In the steady, the evolution of congestion window is very short due to the very high value of window increment and therefore congestion window appears to be very dense. Nonetheless, the entire process runs the following routine; after each loss events, congestion window increases very quickly until another data losses occur; at which congestion window begins to deflate very quickly; after all data losses are recovered, TCP-*hybla* continues in either slow start or congestion avoidance, based on the condition in (3). Then, the entire process repeats.

TCP-*scalable*

TCP-*scalable* is to some extent a modified version of the standard congestion control algorithm. Specifically, instead of relying on the standard congestion avoidance as given in (4), congestion window increases according to

$$cwnd \leftarrow cwnd + max(1/cwnd, \Delta) \tag{32}$$

where $\Delta = 0.01$ if DACK is inactive and $\Delta = 0.02$ if the DACK is active. Therefore, congestion window increments at constant rate provided that window size is large enough.. In other words, TCP-*scalable* increases its congestion window by 1 for every either 100 or 50 acknowledgements received, depending on the state of DACK. For each loss event, slow start threshold and congestion window is reset according to the standard algorithm as described in (6) and (9) with $\beta = 0.875$.

TCP-*scalable* control the congestion window increment by assigning the minimum value of window increment to Δ instead of $1/cwnd$, causing congestion window to grow at constant rate when congestion window size is large enough. Fig. 24 shows the dynamic of TCP-*scalable*. It is clear that window overshoot at the beginning of connection is caused by slow start. In the steady state, i.e. at the 100th second, congestion window grows at a linear rate until data losses occurs at which point congestion window begins to quickly deflate. Once all losses are recovered, congestion window continues to grow at an exponential rate, i.e. slow start, if congestion window is less than slow start threshold and then a linear rate, i.e. congestion avoidance, if otherwise. Then, the entire process repeats.

© 2009 ACADEMY PUBLISHER

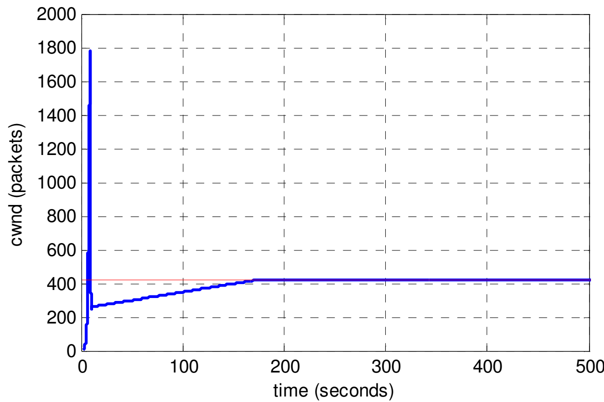

Fig. 26 Congestion window evolution of TCP-*vegas*

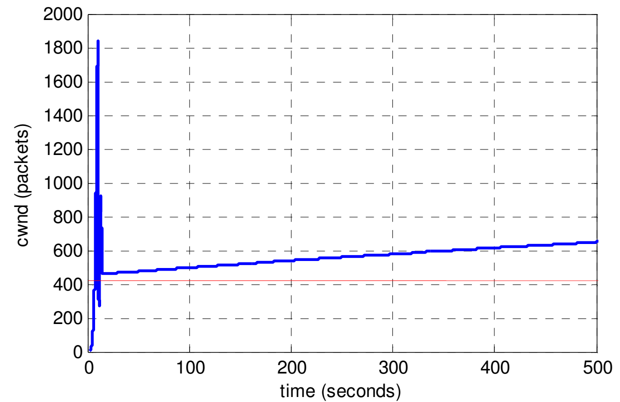

Fig. 27 Congestion window evolution of TCP-*veno*

TCP-*vegas*

TCP-*vegas* is a completely new and different congestion control protocol. Unlike the standard *reno*, i.e. a loss-based algorithm that uses data losses as an indication of network congestion, TCP-*vegas* is a delay-based algorithm that uses an increase in delay as an indication of network congestion. In particular, TCP-*vegas* aims at sending the right amounts of packets into network without causing congestion. Let $diff$ be the difference between the congestion window that is opened in the last round trip time and the target window, i.e.

$$diff = wnd - wnd_{target} \quad (33)$$

where wnd_{target} is the window size that does not cause any increases in round trip time and is defined as

$$wnd_{target} = wnd \cdot baseRTT/minRTT \quad (34)$$

where $baseRTT$ is the minimum round trip time seen during the connection lifetime and $minRTT$ is the minimum round trip time during the last transmission round. Now, TCP-*vegas* operates as follow. During slow start, if $diff$ is larger than a threshold, γ typically set to 2, it indicates that the current congestion window is larger than network capacity and network congestion begins to accumulate. Thus, slow start stops and *linear increase/decrease* phase begins. At this point, congestion window either increases or decreases based on network condition as follow,

$$cwnd \leftarrow \begin{cases} cwnd + 1 & diff < \alpha \\ cwnd & \alpha < diff < \beta \\ cwnd - 1 & diff < N \end{cases} \quad (35)$$

where α and β are two thresholds that indicate the minimum and the maximum numbers of backlogged data inside the network. $diff < \alpha$ means the current congestion window is just the right size, and finally $diff < \beta$ means the current congestion window is too big. Typically, α and β are set to 1 and 3 respectively. In other words, TCP-*vegas* constantly adapts its congestion window increment according to network conditions.

TCP-*vegas* controls the dynamic congestion window by observing an increase in end-to-end delay. In particular, it aims at preventing network congestion from happening by tuning congestion window to evolve around the state at which bandwidth is fully utilized and buffer is minimally occupied. Fig. 25 show the dynamic of TCP-*vegas*. It is clear that window overshoot at the beginning of connection is caused by slow start. Nonetheless, the result does not show any indication of the steady state. However, after the initial slow start, i.e. at the 100th second, congestion window continues to grow according to *linear increase/decrease* rule as described in (35). It is clear that once congestion window reaches the *right* size, it neither increases nor decreases and it remains unchanged until network conditions change.

TCP-*veno*

TCP-*veno* is considered a mixture of the standard *reno* and *vegas*. In particular, TCP-*veno* relies on numbers of backlogged packet as another criterion to update congestion window. Unlike *vegas*, TCP-*veno* does not mainly aim at preventing network congestion from happening and still uses the standard congestion control algorithm for updating congestion window as in (4) with $\alpha = 1$, i.e.

$$cwnd \leftarrow cwnd + 1/cwnd \quad (36)$$

However, instead of updating congestion window regularly, $diff$ that is derived from (33) and (34) is used to dictate how frequently congestion window is updated. Let r be the number of new acknowledgements required for congestion window to be updated, i.e.

$$r = \begin{cases} 1 & \text{if } diff < \gamma, \\ 2 & \text{if } diff \geq \gamma, \end{cases} \quad (37)$$

where γ is the threshold and $\gamma = 3$. The relation between $diff$ and γ is used to indicate network congestion; i.e. if $diff < \gamma$, network is in non-congestive state but if $diff \geq \gamma$, network is in congestive state. In the event of data losses, slow start threshold and congestion window is reduced according to the standard rules as described in (6) and (9) but with two choices of β, i.e.

$$\beta = \begin{cases} 4/5 & \text{if } diff < \gamma, \\ 1/2 & \text{if } diff \geq \gamma, \end{cases} \quad (38)$$

In other words, congestion window is reduced by $1/2$ if network is in congestive state and congestion losses occur and by $4/5$ if in non-congestive state and

© 2009 ACADEMY PUBLISHER

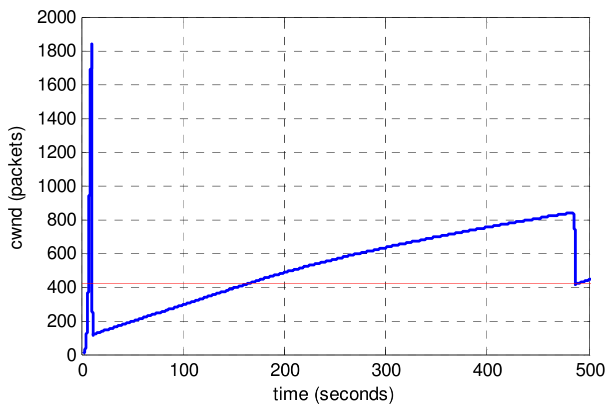

Fig. 28 Congestion window evolution of *westwood*

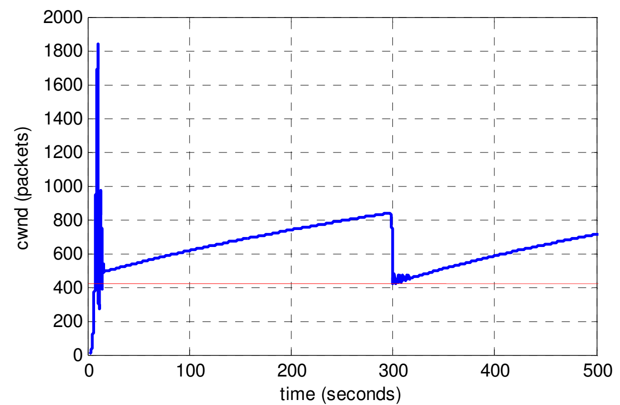

Fig. 29 Congestion window evolution of TCP-*compound*

corruption losses occur. Lastly, TCP-*veno* behaves exactly like *reno* when network is in non-congestive state.

TCP-*veno* controls the congestion window increment by allowing the frequency of window update to be dependent on the state of network congestion, i.e. the relation between $diff$ and γ. Fig. 26 show the dynamic of TCP-*veno*. The result clearly shows the presence of slow start resulting in window overshoot during initial connection but it does not seem to show the steady state. After the initial slow start, congestion window continues to grow linearly and slowly. However, it can be deduced that congestion window will continue to grow linearly as described in (36) with the rate corresponding to network conditions as given in (37) until data losses occur at which point congestion window begins to deflate quickly. After all losses are recovered, TCP-*veno* resumes in either slow start or congestion avoidance based on the condition in (3). Then, the entire process repeats.

TCP-*westwood*

TCP-*westwood* is another extension to the standard *reno*. The main algorithm is still the same as the standard slow start and congestion avoidance as described in (2) and (4) with $\alpha = 1$; but rather than halving slow start threshold as suggested in (6) with $\beta = 1/2$, a new mechanism, called bandwidth estimation (BWE), is used to estimate the available bandwidth based on the inter-arrival time of data acknowledgements and the amounts of transmitted bytes. For each loss events, slow start threshold is reset according to

$$ssthresh \leftarrow (BWE \cdot RTT_{min})/MSS \qquad (39)$$

where RTT_{min} is the minimum round trip time seen during the entire connection lifetime and MSS is the maximum segment size. Therefore, slow start threshold is assigned to a more accurate estimation of available network capacity; particularly in the event of corruption induced losses, i.e. no congestion, bandwidth estimation algorithm better reflects network condition than the simple window halving scheme.

TCP-*westwood* is not designed to control the congestion window increment but rather to mitigate the impacts of random data losses. Fig. 27 shows the dynamic of TCP-*westwood*. The result shows that window overshoot during initial connection is caused by slow start. However, after initial slow start, congestion window unfortunately drops below bandwidth delay product and resumes in congestion avoidance. This is due to the fact that BWE can estimate only bandwidth with respect to the amount of outstanding data and after slow start, the number of outstanding bytes can be small and so does the estimation. Thus, in certain case, BWE can lead to an underestimation of bandwidth. In the steady, congestion window continues to grow linearly until data losses occur at which point congestion window begins to deflate quickly. After all losses are recovered, TCP-*westwood* resumes in either slow start or congestion avoidance depending on the condition given in (3). Then, the whole process repeats.

TCP-*compound*

TCP-*compound* is a hybrid TCP variant that combines the loss-based scheme of *reno* and the delay-based scheme of *vegas* into single scheme. The effective window size is simply the sum of the loss-based and the delay-based components, i.e.

$$win = cwnd + dwnd \qquad (40)$$

and in the case of no network congestion and no data losses, it is incremented as follow,

$$win \leftarrow win + \alpha \cdot win^{k} \qquad (41)$$

where α and k are the two tunable parameters, and $\alpha = 1/8$ and $k = 3/4$. TCP-*compound* is designed to have binomial behavior in the absence of congestion and data losses. In order to achieve this binomial behavior, first the standard congestion avoidance algorithm in (4) has to be modified to compensate for an increase in the effective window due to the delay-based component, i.e.

$$cwnd \leftarrow cwnd + 1/win \qquad (42)$$

Second, the delay-based component is mainly designed to complement the loss-based component. Similar to *vegas*, the relation between $diff$ that is derived in (33) and (34), and γ is used to approximate network utilization; i.e. underutilized if $diff < \gamma$ and highly utilized if $diff \geq \gamma$. Therefore, the delay-based component is updated per new acknowledgment as follow,

© 2009 ACADEMY PUBLISHER

$$dwnd \leftarrow \begin{cases} dwnd + [\alpha \cdot win^k - 1]^+ & \text{if } diff < \gamma \\ dwnd - \xi \cdot diff & \text{if } diff \geq \gamma \end{cases} \quad (43)$$

where $[\cdot]^+$ is the maximum between the argument and zero and ξ is the parameter that indicates how rapidly the delay-based component is reduced when needed. In addition, $\xi = 1$ and $\gamma = 30$. Moreover, the delay-based component will eventually disappear in the case of full utilization but it will grow much larger in the case of underutilization. In the case of no congestion; neither increase in queue nor data losses, slow start threshold is halved while the effective window is reduced as follow,

$$win \leftarrow \beta \cdot win \quad (44)$$

corresponding to the modification of the loss-based and delay-based components as follow,

$$cwnd \leftarrow \beta \cdot cwnd \quad (45)$$

$$dwnd \leftarrow \beta \cdot win - cwnd/2 \quad (46)$$

where β is the multiplicative decrease factor and $\beta = 1/2$. Lastly, TCP-*compound* behaves exactly like *reno* when network is fully utilized, i.e. $diff \geq \gamma$.

TCP-*compound* controls the congestion window growth rate by allowing the delay-based component to grow much faster when it senses neither network congestion nor data losses. Fig. 28 shows the dynamic of TCP-*compound*. The result shows that at the beginning of connection, window overshoot is caused by slow start. In the steady state, i.e. at the 100th second, congestion window grows linearly until data losses occur at which point congestion window begins to deflate very quickly. After all data losses are recovered, congestion window continues to grow. Nevertheless, since TCP-*compound* senses that network is not fully utilized, the delay-based component is therefore active. The impacts of the delay-based is seen by the ups and downs of congestion window. After network is fully utilized, the delay-based component becomes inactive and congestion window grows as normal. Afterward, the whole process repeats.

Clearly, *LinuxTCP* is capable of re-producing congestion window dynamics of different Linux TCP variants in which it is believed to be much closer to reality due to the use of real-world code. Thus, different TCP variants can be studied and compared to each other to evaluate the performance of interest, i.e. link utilization, robustness, friendliness and fairness. In addition, new TCP variants can effortlessly be included into INET as long as they are complied with Linux congestion control interface, like *compound*.

IX. Conclusions and Future Works

This work presents how real-world TCP can be evaluated via a network simulator. By combining real-world Linux TCP and INET simulation framework, true dynamics of real-world TCP can be accurately captured in varieties of simulated network environments without having to rely on costly and complicated network experiment. Moreover, this works present one verification and validation technique, i.e. dynamic testing and operational graphic, and eventually concludes that *LinuxTCP* is valid within the simulated network environments provided by INET. In addition, the applications of *LinuxTCP* are demonstrated by reviewing congestion control algorithms and displaying congestion window dynamics for each of TCP variants, i.e. *bic*, *cubic*, *hamilton*, *highspeed*, *hybla*, *scalable*, *vegas*, *veno*, *westwood* and *compound.* Note that *compound* is a special case since it is not actually a native Linux congestion control but was developed based on the Window Vista based TCP. These many selections of TCP variants strengthen the decision on choosing Linux TCP/IP network stack as the source for the integration.

The novelties of this work are as follow. First, this work introduces the interface-based methodology for integrating the transport layer of Linux TCP/IP network stack from into INET in which it can be served as a guideline for integrating future network stack with INET or with other simulators. Second, this work presents independent development on integrating real-world Linux TCP with a simulator in which it can be used for cross validation among different network simulators. Third, this work realizes an evaluation of the real-world Linux TCP in simulated network environment in which it is believed to be the most practical and flexible means in studying real-world protocols for a varieties of networks and technologies. Lastly, the choice of Linux TCP/IP network stack permits new TCP variants to be developed and tested in simulated network environments and later these new TCP variant can be tested and used in real-world network with no additional efforts required. The use of INET with Linux TCP extension greatly reduces the development time.

The future works will mainly aim at exploiting *LinuxTCP* in the performance study and evaluation of different Linux TCP variants in the next generation network and the next generation Internet, i.e. large bandwidth, long delay, high bit error rate and different service quality. The secondary aim will be to develop a new TCP variant that performs well in the next generation network. The final aim will be to design a much cleaner INET-Linux interface that will speed up the integration processes in the future for new INET and Linux kernel.

Appendix

The detailed implementations of all TCP variants, except *compound*, can be found in the Linux kernel source code available at the Linux archive website [14] under /net/ipv4/ directory.

Acknowledgment

S. Kittiperachol thanks the Royal Thai Navy for the generous financial support.

© 2009 ACADEMY PUBLISHER

REFERENCES

[1] A. Varga, "OMNeT++ discrete event simulation system," http://www.omnetpp.org.

[2] "Castalia - a simulator for WSNs," http://castalia.npc.nicta.com.au.

[3] "Chsim," http://www.cs.uni-paderborn.de/en/research-group/research-group-computer-networks/projects/chsim.html.

[4] "OverSim: the overlay simulation framework," http://www.oversim.org.

[5] K. Fall and S. Floyd, "Simulation-based comparisons of Tahoe, Reno and SACK TCP," *ACM Computer Communication Review,* vol. 26, no. 3, pp. 5-21, July 1996.

[6] V. Jacobson and M. Karels, "Congestion avoidance and control," in *Proceeding of ACM SIGCOMM '88*, Stanford, California, USA, August 1988, pp. 314-329.

[7] S. Floyd, T. Henderson, and A. Gurtov, "The NewReno modification to TCP's fast recovery algorithm," *RFC 3782,* April 2004.

[8] "OppBSD - a FreeBSD network stack integrated into OMNeT++," https://projekte.tm.uka.de/trac/OppBSD.

[9] D. X. Wei and P. Cao, "A Linux TCP implementation for NS2," http://www.cs.caltech.edu/~weixl/technical/ns2linux.

[10] R. Bless and M. Doll, "Integration of the FreeBSD TCP/IP-stack into the discrete event simulation OMNeT++," in *Proceeding of Winter Simulation Conference*, Washington DC, USA2004, pp. 1556-1560.

[11] S. Jansen and A. McGregor, "Simulation with real world network stacks," in *Proceeding of Winter Simulation Conference*, Orlando, Florida, USA, 4-7 December 2005.

[12] "The network simulator - ns2," http://www.isi.edu/nsnam/ns.

[13] D. X. Wei and P. Cao, "NS-2 TCP-Linux: an NS-2 TCP implementation with congestion control algorithms from Linux," in *Proceeding of 2006 workshop of ns-2: the IP network simulator*, Pisa, Italy, 10 October 2006.

[14] "The Linux kernel archive," http://www.kernel.org.

[15] V. Paxon and M. Allman, "Computing TCP's retransmission timer," *RFC 2988,* November 2000.

[16] V. Jacobson, R. Braden, and D. Borman, "TCP extensions for high performance," *RFC 1323,* May 1992.

[17] E. Blanton, M. Allman, K. Fall, and L. Wang, "A conservative selective acknowledgement (SACK)-based loss recovery algorithm for TCP," *RFC 3517,* April 2003.

[18] M. Allman, "Ongoing TCP research related to satellites," *RFC 2760,* February 2000.

[19] M. Allman, H. Balakrishnan, and S. Floyd, "Enhancing TCP's loss recovery using limited transmit," *RFC 3042,* January 2001.

[20] A. Allman, "TCP byte counting refinements," *ACM Computer Communication Review,* vol. 3, no. 28,July 1999.

[21] M. Mathis and J. Mahdavi, "Forward acknowledgement: refining TCP congestion control," in *Proceeding of ACM SIGCOMM '96*, Stanford, California, USA1996, pp. 281-291.

[22] S. Kittiperachol, Z. Sun, and H. Cruickshank, "Integration of Linux TCP implementation: verification and validation," in *Proceeding of SPECTS '08*, Edinburgh, UK, 16-18 June 2008.

[23] O. Balci, "Verification, validation and accreditation," in *Proceeding of Winter Simulation Conference*, Washington DC, USA, 13-16 December 1998, pp. 41-48.

[24] R. G. Sargent, "Verification and validation of simulation models," in *Proceeding of Winter Simulation Conference*, Washington DC, USA, 13-16 December 1998, pp. 121-130.

[25] M. Mathis, J. Semke, and J. Mahdavi, "The macroscopic behavior of the TCP congestion avoidance algorithm," *ACM Computer Communication Review,* vol. 27, no. 3, pp. 67-82, July 1997.

[26] Z. Sun, *Satellite networking principles and protocols*: Wiley, 2005.

[27] L. Xu, K. Harfoush, and I. Rhee, "Binary increase congestion control (BIC) for fast long-distance networks," in *Proceeding of IEEE INFOCOM '04*, Hong Kong, 7-11 March 2004, pp. 2514-2524.

[28] I. Rhee and L. Xu, "CUBIC: A New TCP-friendly high-speed TCP variant," in *Proceeding of PFLDnet 2005*, Lyon, France, 3-4 February 2005.

[29] S. Floyd, "Highspeed TCP for large congestion windows," *RFC 3649,* December 2003.

[30] D. Leith and R. Shorten, "H-TCP: TCP for high-speed and long-distance networks," in *Proceeding of PFLDnet 2004*, IL, USA, 16-17 February 2004.

[31] C. Caini and R. Firrincieli, "TCP Hybla: a TCP enhancement for heterogeneous networks," *International Journel of Satellite Communciations and Networking,* vol. 22, no. 5, pp. 547-566, September 2004.

[32] T. Kelly, "Scalable TCP: improving performance in highspeed wide area networks," *ACM SIGCOMM Computer Communication Reviews,* vol. 33, no. 2, pp. 83-91, April 2003.

[33] L. Brakmo and L. Peterson, "TCP Vegas: end to end congestion avoidance on a global internet," *IEEE Journal on Selected Areas in Communications,* vol. 13, no. 8, pp. 1465-1480, October 1995.

[34] C. P. Fu and S. C. Liew, "TCP Veno: TCP enhancement for transmission over wireless access," *IEEE Communication Magazine,* vol. 21, no. 2, pp. 216-228, February 2003.

[35] K. Tan, J. Song, Q. Zhang, and M. Sridharan, "Compound TCP: a scalable and TCP-friendly congestion control for high-speed networks," in *Proceeding of PFLDnet 2006*, Nara, Japan, 2-3 February 2006.

[36] K. Tan, J. Song, Q. Zhang, and M. Sridharan, "A compound TCP approach for high-speed and long distance network," in *Proceeding of IEEE INFOCOM '06*, Barcelona, Spain, 23-29 April 2006, pp. 1-12.

Songrith Kittiperachol was born in Bangkok, Thailand. The author received both the B.Sc (Cum Laude) and M.Sc degrees in Electrical and Computer Engineering in 1998 and 2000 from University of Maryland at College Park, Maryland, USA.

He was a part-time lecturer at Mahidol University, Thailand. He also was a network and system engineer at the Royal Thai Navy, Thailand. Currently, he is studying PhD at the Centre for Communication Systems Research (CCSR), Faculty of Electronics and Physical Science, University of Surrey, United Kingdom. His main research interests are the performance evaluation and optimization of Internet over satellite network.

Zhili Sun received the BSc degree in mathematics from Nanjing University, China, and the PhD degree in computing science from Lancaster University, United Kingdom. He is the chair of Communication Networking in the Centre for Communication Systems Research (CCSR), Department of Electronic Engineering, Faculty of Electronics and Physical Sciences, University of Surrey, United Kingdom.

He is a professor in data and Internet networking, as well as satellite communication courses, at the University of Surrey. He has also been a

principal investigator for the UK Engineering and Physical Sciences Research Council (EPSRC), European Space Agency (ESA), and industrial projects on IP multicast security. His main research interests include network security, satellite network architectures, Voice over IP (VoIP), and IP conferencing over satellites. He has published a book titled Satellite Networking (Wiley) and more than 120 papers in international journals and conferences. He is a member of the Satellite and Space Communications Committee of the IEEE Computer Society and a chartered engineer and corporate member of the IET in the United Kingdom.

Haitham Cruickshank is a lecturer in data and Internet networking and satellite communication courses at the Centre for Communication Systems Research (CCSR), Department of Electronic Engineering, Faculty of Electronics and Physical Sciences, the University of Surrey, United Kingdom.

Since January 1996, he has been working on several European research projects in the Advanced CompuTational Software (ACTS), Esprit, Trans-European Telecommunications Network (TEN-TELECOM), and Information Society Technologies (IST) programs. His main research interests include network security, satellite network architectures, Voice over IP (VoIP), and IP conferencing over satellites. He is a member of the Satellite and Space Communications Committee of the IEEE Computer Society and a chartered engineer and corporate member of the IET in the United Kingdom.

© 2009 ACADEMY PUBLISHER