Oriental Journal of Computer Science & Technology Vol. **3**(1), 95-102 (2010)

Packet loss detection using CPR and WCPR in diverse platforms

ANURAG LAL[1] and VIVEK DUBEY[2]

[1]Computer Science & Engineering, CIT, Rajnandgaon, (India).
[2]Computer Science & Engineering, SSCET , Bhilai (India).

(Received: April 23, 2010; Accepted: May 17, 2010)

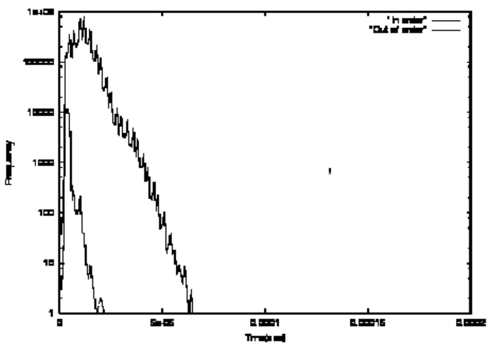
ABSTRACT

In this paper loss of packets in TCP is detected using two diverse methods CPR (Constant Packet Re-arranging) and WCPR (Without Constant Packet Re-arranging) in diverse platforms. This paper proposes a new version of the TCP which gives the high throughput when the packet rearranging occurs and in another case if the packet rearranging is not occurs then in that case also it is friendly to other version of the TCP. The key feature of Constant packet rearranging is that duplicate ACKs are not used as an indication of packet loss. Instead the timer is used to detect the packet loss From a computational view-point, CPR is more demanding than WCPR. Because CPR does not rely on duplicate acknowledgments, packet rearranging (including out-or-order acknowledgments) has no effect on CPR performance.

Keywords: CPR, WCPR, congestion control, packet rearranging.

INTRODUCTION

Here the two methods TCP-WCPR old and TCP CPR is compared. In this paper we proposed and evaluated the performance of TCP-CPR, a variant of TCP that is specifically designed to handle constant rearranging of packets (both data and acknowledgment packets). Our results show that TCP-CPR is able to achieve high throughput when packets are rearranged and yet is fair to standard TCP implementations, exhibiting similar performance when packets are delivered in order. From a computational view-point, TCP-CPR is more demanding than TCP WCPR. Because TCP-CPR does not rely on duplicate acknowledgments, packet reordering (including out-or-order acknowledgments) has no effect on TCP-CPR's performance. This packet loss is detected by sender's retransmission timeout (RTO) expiring that is sender has to set the threshold time after sending the packets and waiting for acknowledgement of each packet. If the elapsed time of acknowledgement exceeds the threshold time, the sender can assume the packet is lost and resend the corresponding packets. It increase the throughput, load balancing, and security; protocols that provide diverseiated services and traffic engineering approaches. The main idea behind retransmit packet this is to improve the performance of TCP throughput by avoiding sender to timeout. Using fast retransmit can continuously improve the TCP's performance in the presence of irregular rearranging but it still operates under the assumption of that out-of-order packet which indicate the packet loss and which leads to congestion. As a result its performance degrades in the presence of constant rearranging. This is procedure for rearranging both data and acknowledgment packet. Packet rearranging is generally attributed to transient conditions pathological behavior and erroneous implementation. TCP uses two strategies for the detection of the packet loss the first one is based on the sender's retransmission timeout which is also referred as coarse timeout. When the senders timeout which is responded by the congestion control by slow start which leads into decreasing congestion window to one segment. The packet detection loss is detected at the receiver side by using the sequence number. In this case receiver checks the sequence number of received packet. The hole in the sequence indicates that there is loss of the packet in such case the receiver generates the duplicate acknowledgement for every "out-of-order" segment it receives. Until the lost packet received, the entire reaming packet with higher sequence number is consider as out of order and will cause to creation of duplicates packets. After that sender retransmit the lost packet without waiting for timeout which helps to reduction of congestion windows.


F_{300}(LA->Pitt, 500), O_1 =0.59%

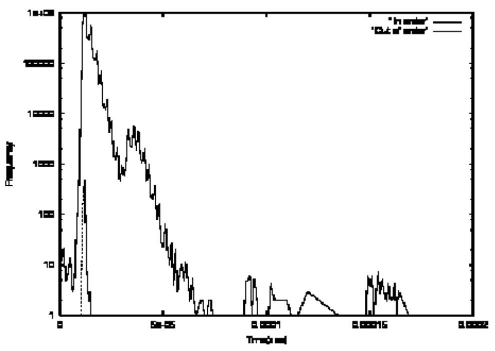

F_{800}(Pitt->DC, 1500), O_1 = 0.02%

Fig. 1: Packet inter-arrival time

• Packets enter network evenly spaced; • Packets arriving:
⇒ in-order experience expected amount of dispersion;
⇒ out-of-order have smaller inter-arrival times relative to in-order packet arrivals.

Packet Inter Arrival Time Algorithm

Packets being processed by the sender are kept in one of two lists: the to-be-sent list contains all packets whose transmission is pending, waiting for an "opening" in the congestion window. The to-be-ack list contains those packets that were already sent but have not yet been acknowledged. Typically, when an application produces a packet it is ûrst placed in the to-be-sent list; when the congestion window allows it, the packet is sent to the receiver and moved to the to-be-ack list; ûnally when an ACK for that packet arrives from the receiver, it is removed from the to-be-ack list (under cumulative ACKs, many packets will be simultaneously removed from to-be-ack). Alternatively, when it is detected that a packet was dropped, it is moved from the to-be-ack list back into the to-be-sent list.

As mentioned above, drops are always detected through timers. To this effect, whenever a packet is sent to the receiver and placed in the to-be-ack list, a timestamp is saved. When a packet remains in the to-be-ack list more than a certain amount of time it is assumed dropped. In particular, we assume that a packet was dropped at time when exceeds the packet's timestamp in the to-be-ack list plus an estimated maximum possible round-trip time $\texttt{mxrtt}$.

As data packets are sent and ACKs received, the estimate of the maximum possible round-trip time is continuously updated. The estimate used is given by

$$\texttt{mxrtt} := \beta \times \texttt{srtt} \qquad (1)$$

where β is a constant larger than 1 and $\texttt{srtt}$ an exponentially weighted average of past

RTTs. Whenever a new ACK arrives, we update

$$\mathtt{srtt} = \max\left\{\alpha^{\frac{1}{\lfloor \mathtt{cwnd} \rfloor}} \times \mathtt{srtt}, \mathtt{sample-rtt}\right\} \qquad (2)$$

where α denotes a positive constant smaller than 1, $\lfloor \mathtt{cwnd} \rfloor$ the ûoor of the current congestion window size, and $\mathtt{sample-rtt}$ the RTT for the packet whose acknowledgment just arrived. The reason to raise to the power is that in one the formula in (2) is iterated times. This means that, e.g., if there were a sudden decrease in the RTT then $\mathtt{srtt}$ would decrease by a rate of $(\alpha^{1/\lfloor \mathtt{cwnd} \rfloor})^{\lfloor \mathtt{cwnd} \rfloor} = \alpha$ per ,

independently of the current value of the congestion window. The parameter α can therefore be interpreted as a smoothing factor in units of RTT. As discussed in Section IV, the performance of the algorithm is actually not very sensitive to changes in the parameters β and α, provided they are chosen in appropriate ranges.

Note that $\mathtt{srtt}$ tracks the peaks of RTT. The rate that decays after a peak is controlled by α. The right-hand plot shows how large jumps can cause $RTT > \mathtt{mxrtt}$ (for this data set, occurrences at 15 s, 45 s, 75 s, etc.) resulting in spurious time- outs (note that the jumps in RTT in the right-hand plot were artiûcially generated). In order for these jumps to cause a spurious timeouts, the jumps in RTT could occur no sooner than every 15 seconds. In this case, 1500 packets were delivered between these jumps. If the jumps occurred more frequently, then, as can be seen from the figure $\mathtt{mxrtt}$, would not have decayed to a small enough value and spurious timeouts would not occur. Furthermore, if the jumps were larger, then the time between jumps to cause a timeout would be no smaller. The issue of spurious timeouts is closely examined.

Two modes exist for the update of the congestion window:slow-start and congestion-avoidance. The sender always starts in slow-start and will only go back to slow-start after periods of extreme losses. In slow-start $\mathtt{cwnd}$, starts at 1 and increases exponentially (increases by one for each ACK received). Once the first loss is detected, $\mathtt{cwnd}$ is halved and the sender transitions to congestion-avoidance, where $\mathtt{cwnd}$ increases linearly ($1/\mathtt{cwnd}$ for each ACK received). Subsequent drops cause further halving of $\mathtt{cwnd}$, without the sender ever leaving congestion-avoidance. An important but subtle point in halving is that when a packet is sent, not only a timestamp but the current value of $\mathtt{cwnd}$ is saved in the to-be-ack list. When a packet drop is detected, then is actually set equal to half the value of at the time the packet was sent and not half the current value of $\mathtt{cwnd}$. This makes the algorithm fairly insensitive to the delay between the time a drop occurs until it is detected.

To prevent bursts of drops from causing excessive decreases in $\mathtt{cwnd}$, once a drop is detected a snapshot of the to-be-sent list is taken and saved into an auxiliary list called memorize. As packets are acknowledged or declared as dropped, they are removed from the memorize list so that this list contains only those packets that were sent before $\mathtt{cwnd}$ was halved and have not yet been unaccounted for. When a packet in this list is declared dropped, it does not cause $\mathtt{cwnd}$ to be halved. The rational for this is that the sender already reacted to the congestion that caused that burst of drops. This type of reasoning is also present in TCP-New Reno and TCP-SACK.

METHODS

TCP without Constant Packet Rearranging

If we transmit a message without packet rearranging, then If part of a message is lost during the transmission then we need to retransmit the entire message or we need to retransmit from that particular part. Therefore, upon detecting loss, the TCP sender backs off its transmission rate by decreasing its congestion window.

TCP with Constant Packet Rearranging

If we transmit a message as packets then we need to retransmit only the packet which is lost and not the entire message. The message is send from the source to the ingress router and then to

Table 1 : code for TCP-CPR

Event		Code
initialization	1	mode := *slow-start*
	2	cwnd := 1
	3	ssthr := $+\infty$
	4	memorize := $\emptyset$
time > time(n) + mxrtt (drop detected for packet n)	5	remove(to-be-ack, n)
	6	add(to-be-sent, n)
	7	if not is-in(memorize, n) then /* new drop */
	8	memorize := to-be-ack
	9	cwnd := cwnd(n)/2
	10	ssthr := cwnd
	11	else /* other drops in burst */
	12	remove(memorize, n)
	13	flush-cwnd()
ack received for packet n	14	srtt = max $\{\alpha^{\frac{1}{\text{cwnd}}} \times$ srtt, time − time(n)$\}$
	15	mxrtt := $\beta \times$ srtt
	16	remove(to-be-ack, n)
	17	remove(memorize, n)
	18	if mode = *slow-start* and cwnd + 1 $\leq$ ssthr then
	19	cwnd := cwnd + 1
	20	else
	21	mode := *congestion-avoidance*
	22	cwnd := cwnd + 1/cwnd
	23	flush-cwnd()
flush-cwnd()	24	while cwnd > |to-be-ack| do
	25	k=send(to-be-sent)
	26	remove(to-be-sent, k)
	27	add(to-be-ack, k)
	28	time(k) = time

the intermediate routers and then to the outgress router and the destination. The basic idea behind TCP constant packet rearranging is to detect packet losses through the use of timers instead of duplicate acknowledgments. This is prompted by the observation that, under constant packet rearranging, duplicate acknowledgments are a poor indication of packet losses. Because TCP constant packet rearranging relies solely on timers to detect packet loss, it is also robust to acknowledgment losses as the algorithm does not distinguish between data (on the forward path) or acknowledgment (on the reverse path) losses.

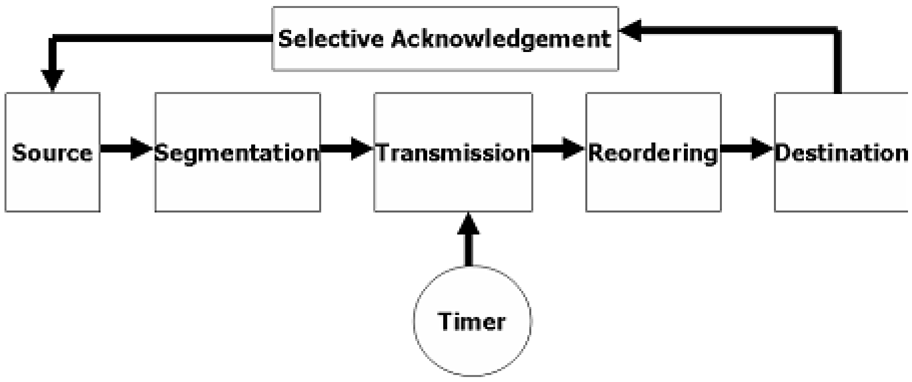


Figure 2(a) : Packet rearranging

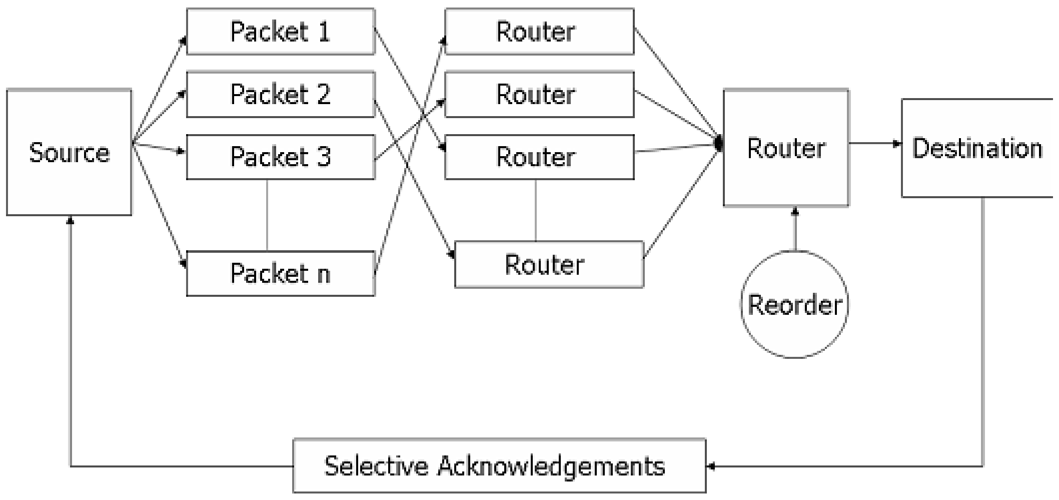


Figure 2(b) : Packet rearranging

Segmentation

Segmentation is the process of dividing the source code into small number of packets and transmitting the packets through the routers. We define certain limits for the size of the packets. The header information includes source machine name, destination machine name, position of the packet and the related information.

Timer control

Whenever each and individual packet starts sending a timer is started. The system current time is taken as a start time and added with delay and it acts as a threshold time and if the threshold time exceeds the maximum elapsed time of the packet then the packet is retransmitted. If the time doesn't exceed then the packet may arrive safe. If so the next packet is transmitted else the current packet is transmitted until it arrives safely. Thread concept is used to implement the timer.

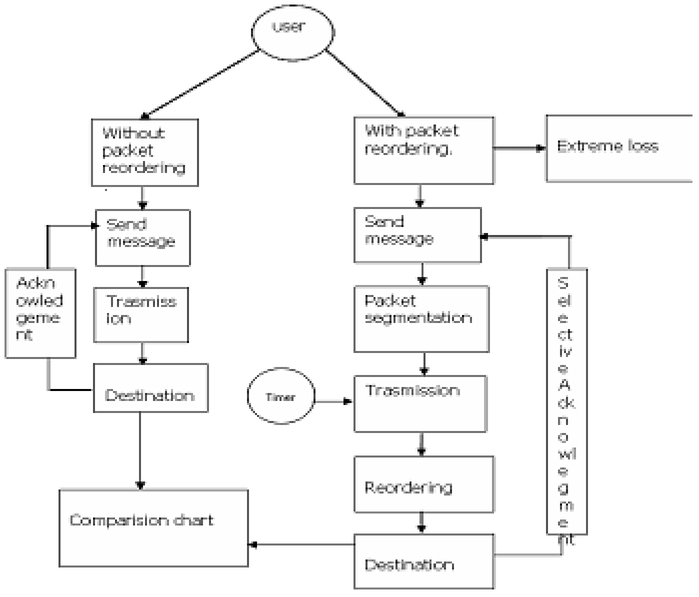


Fig. 3: Data flow diagram

RESULTS

The performance of the transmission control protocol with packet rearranging is tested on WINDOWS XP, LINUX and SOLARIS. Comparison chart compares the throughput of TCP-WCPR with TCP-CPR. The performance is shown by comparing the Transmission rate of existing system with proposed system

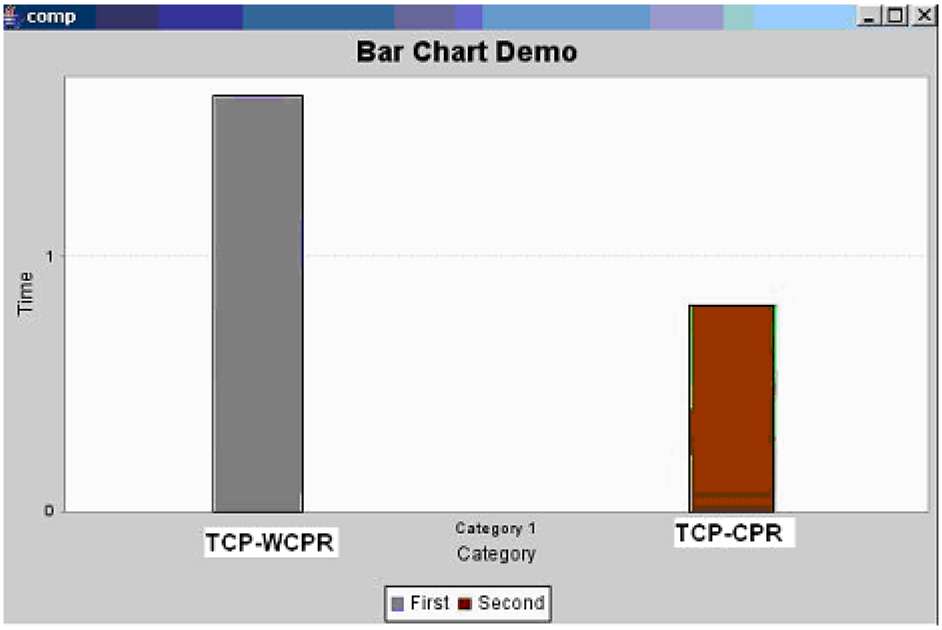


Fig. 4(b) Comparison chart of TCP without packet rearranging and with packet rearranging in Linux

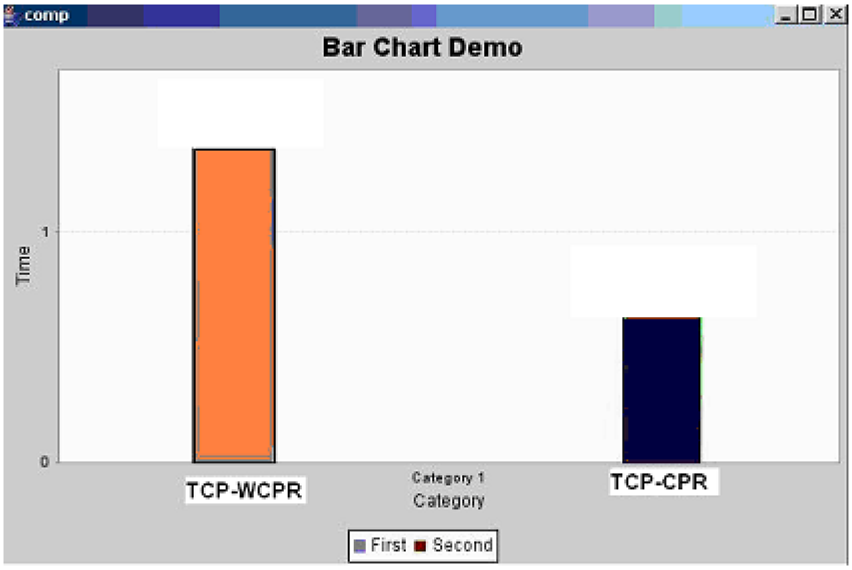


Fig. 4(a) Comparison chart of TCP without packet rearranging and with packet rearranging in Windows Xp

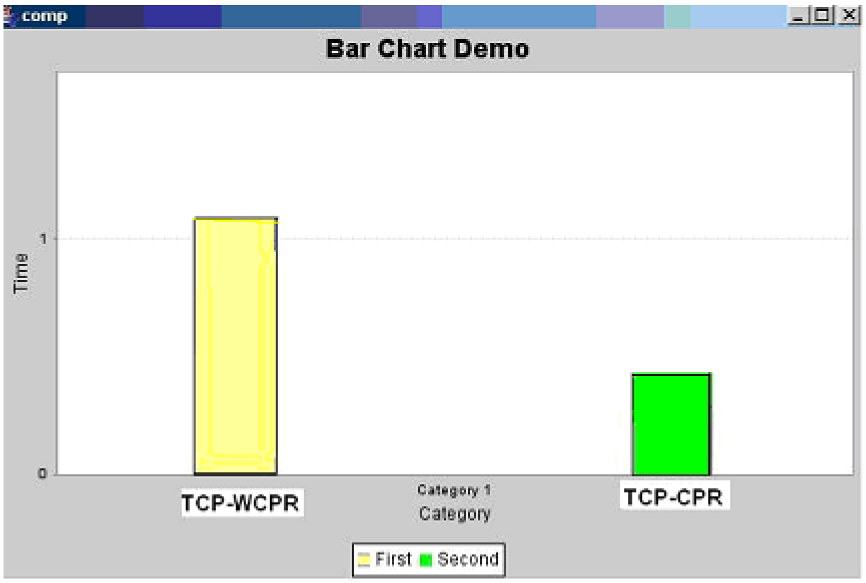


Fig. 4(c) Comparison chart of TCP without packet rearranging and with packet rearranging in Solaris

CONCLUSION

In this paper we proposed and evaluated the performance of TCP constant packet rearranging, a variant of TCP that is speciûcally designed to handle constant rearranging of packets (both data and acknowledgment packets). Our simulation results show that TCP constant packet rearranging is able to achieve high throughput when packets are reordered and yet is fair to standard TCP implementations, exhibiting similar performance when packets are delivered in order. Such mechanisms include proposed enhancements to the original Internet architecture such as multi-path routing for increased throughput, load balancing, and security; protocols that provide diverseiated services; and traffic engineering approaches. As shown from result that the delay time for XP is more than that of Linux ,which is much more then of Solaris

REFERENCES

1. Luo X. and Chang R. K. C., "Novel Approaches to End-to-end Packet Rearranging Measurement" Proc. ACM/ USENIX Conf. Internet Measurement, (2005).
2. Bare, A.A., "Measurement and Analysis of Packet Rearranging," Masters Thesis, Dep. Computer Science, Colorado State University, (2004).
3. Colin M. Arthur, Andrew Lehane, David Harle, "Keeping Order: Determining the Effect of TCP Packet Rearranging," icns, pp.116, International Conference on Networking and Services (ICNS '07), (2007).
4. Sharad Jaiswal, G. Iannaccone, C. Diot, J. Kuorose, and D. Towsley. Measuring and classiûcation of out-of-sequence packets in a Tier-1 IP Backbone, International Measurement Workshop(IMW), (2003).
5. Piratla, N. M. , Jayasumana A. P. ,and Bare A. A., "A Comparative Analysis of Packet Rearranging Metrics," Proc. IEEE/ACM 1st

Int. Conf. Communication System Software and Middleware (COMSWARE 2006), New Delhi, (2006).

6. Jaiswal, S., Iannaccone, G., Diot, C., Kurose, J. and Towsley, D., "Measurement and Classification of Out-of-sequence Packets in Tier-1 IP Backbone," Proc. IEEE INFOCOM, pp. 1199- 1209 (2003).
7. R. Teixeira, K. Marzullo, S. Savage, and G. M. Voelker, "Characterizing and measuring path diversity of Internet topologies," presented at the ACM SIGMETRICS, San Diego, CA, (2003).
8. S. Bohacek, "A stochastic model of TCP and fair video transmission," in *Proc. IEEE INFOCOM*, pp. 1134–1144 (2003).
9. M. Franklin, T. Wolf. A Network Processor Performance and Design Model with Benchmark Parameterization. First Workshop on Network Processors, Cambridge, MA, (2002).
10. S. Bohacek, J. Hespanha, K. Obraczka, J. Lee, and C. Lim, "Secure stochastic routing," presented at the ICCCN'02, Miami, FL, (2002).