Adaptive auto-tuning of TCP pacing

Naoki Tanida[1], Mary Inaba[1], and Kei Hiraki[1]

The University of Tokyo, 7-3-1 Hongo Bunkyo Tokyo, Japan

Abstract. Distributed computing such as cloud computing is getting prevalent in high-performance computing. For super-large-scale computation by connecting supercomputers scattering all over the world, it is important to optimize the communication among them. However, it is difficult to achieve high performance in TCP communication on long-distance paths like inter-cluster networks. It is important to limit the maximum transmission speed properly by pacing according to the latency and ever-changing condition of the network. This paper proposes *adaptive auto-tuning of TCP pacing* that tunes the pacing speed dynamically depending on the network situation that is difficult by static and manual approach. We introduce two types of dynamic pacing and combine them with BIC TCP congestion control in the sender. To avoid the packet loss that causes performance deterioration, our method suppresses burst transmission, estimates available bandwidth and limit the excess growth of the congestion window size. In the experiments on 10 Gbps networks with arbitrary large delay, we found our proposal improves the TCP performance in parallel TCP communication.

Keywords: auto-tuning, transmission control protocol, congestion control, pacing

1 Introduction

The performance of computer has been advanced with the evolution of calculation and communication speed. Current supercomputers have achieved petascale computing speed by connecting many computing nodes locally in a supercomputing center. To support users without system-specific knowledge, auto-tuning technology for clusters has been well investigated in computation such as numerical libraries. However, distributed computing such as cloud computing is getting prevalent in high-performance computing because of its user-friendliness and small restriction of power consumption and installation area. In distributed high-performance computing, it is indispensable to speed up various communication including in-cluster low-latency communication and inter-cluster long-distance communication, as well as calculation. Especially for super-large-scale computation by connecting supercomputers scattering all over the world, it is important to optimize the communication among them.

Use of the transmission control protocol (TCP) is desirable for the inter-cluster communication, because clusters all over the world are generally connected on the Internet with TCP protocol. However, it is difficult to achieve high

performance in TCP communication on long-distance paths like inter-cluster networks. In the distributed supercomputing such as distributed astronomical simulation [11, 19], optimization according to the distance and condition of the network paths, especially TCP optimization, is required to achieve stable high-performance of communication on shared network. As for TCP optimization, there are several researches on auto-tuning of the congestion window size (cwnd). For example, Semke et al. [14] implemented in the NetBSD kernel automatic tuning of the TCP window size in the sender/receiver. They allocated proper size of window dynamically to improve the performance of TCP, saving as many memory as possible. However, in the real distributed simulation, manual tuning of many TCP parameters and properly limiting the maximum transmission speed by *pacing* [2] are still required as we describe in Section 2. It has been almost impossible to perform distributed supercomputing without network specialists.

When data are transferred simply from one server to the other server on dedicated networks, the pacing speed can be tuned statically according to the network paths. On the other hand, when multiple servers transfer data in parallel and the throughput dynamically changes, it is important to tune the pacing speed automatically and adaptively according to the latency and ever-changing condition of the network. This paper proposes *adaptive auto-tuning of TCP pacing* that tunes the pacing speed dynamically depending on the network situation that is difficult by static and manual approach.

The rest of this paper is organized as follows. Section 2 describes the summary of TCP congestion control and the necessity of pacing in high-speed long-distance TCP communication. Section 3 presents the design and implementation of our proposal. Section 4 includes the evaluation of our proposal. Section 5 describes the related work and Section 6 gives conclusion.

2 Background

2.1 TCP

TCP [13, 4] is the standard transport layer protocol on the Internet. TCP is a connection-oriented, end-to-end reliable protocol that supports flow control and congestion control. TCP uses *sliding window* flow control. The receiver returns an acknowledgment packet (ACK) to the sender to acknowledge a data packet. The sender decreases the *window* when packets are sent and increases it when ACKs are received. Thus the window slides and *inflight* packets, packets transmitted from the sender and not ACKed yet, are limited to the packets in the window (Fig. 1). The amount of data sent for round-trip time (RTT) period is limited to the window size and this ACK-triggered RTT periodical window size control is called *self-clocking*. This window size is limited by several windows: *congestion window*, *send window*, *receive window* and *advertised window*. Especially the congestion window is used to control throughput in TCP congestion control.

Many TCP congestion control algorithms have been proposed and improved. The first algorithm is Tahoe [9]. The *slow start*, *congestion avoidance* and *fast*

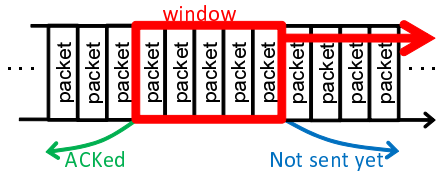

Fig. 1. Sliding window

retransmission are adopted in Tahoe. The slow start is a strategy for controlling a congestion window that the sender set its cwnd to 1 when a TCP session is established and it is exponentially increased after that. The purpose of the slow start is to prevent the traffic from suddenly inflowing to networks. The congestion avoidance is another strategy for controlling the congestion window. In the congestion avoidance phase, cwnd is linearly increased, i.e. cwnd is increased by 1 maximum segment size (MSS) for each ACK. This is called *additive increase*. The slow start is used when cwnd is smaller than a certain threshold, *slow start threshold* (ssthresh), and otherwise the congestion avoidance is used. The receiver returns a *duplicate acknowledgment* (DACK) packet when it receives an out-of-order packet. A DACK can be interpreted that a packet is likely to be lost. Then waiting for RTO to retransmit the packet is inefficient. Thus after the sender receives three consecutive DACKs, it retransmits the corresponding data packet before RTO. This is called *fast retransmission*. After the fast retransmission or RTO, cwnd is decreased to 1.

Reno [8] is the second algorithm. In Reno, *fast recovery* is added to Tahoe. When fast retransmission succeeds, the network congestion can be considered lenient. Thus, after the successful fast retransmission, cwnd is only halved to speed up the recovery. This is called *multiplicative decrease*. Thus, cwnd is controlled in an *additive increase multiplicative decrease* (AIMD) manner. When *selective acknowledgment* (SACK) option [12] is enabled, the receiver can acknowledge not only the last packets received in-order but also the range of packets received. Then the sender can retransmit only the packets that are losts. However, when SACK is unavailable and multiple packets are lost in a window, the ACK for the retransmitted packet acknowledges not all the packets transmitted before fast transmission, and unnecessary fast transmission is invoked improperly.

NewReno [3] is the revised version of Reno. The fast recovery is modified in NewReno so that it works correctly even when SACK option is unavailable [7]. NewReno keeps fast recovery until all the packets in the window at the time of entering fast recovery are acknowledged. However, NewReno has still problem in RTT fairness. When multiple NewReno flows with different RTT share the same network path, a flow with smaller RTT attains more throughput than other flows, because cwnd is increased every RTT.

Most modern algorithms are based on NewReno. BIC [18] improves RTT fairness. In the fast recovery phase, the next cwnd is set to the midpoint between current size and the size at the last packet loss. This behavior is repeated

like binary search. This is called *binary search increase*. In addition, when the distance to the midpoint from current size is too large, additive increase is used to reduce the impact to other traffic. Thus, in BIC, binary search increase is combined with additive increase to improve RTT fairness. This is called *binary increase*. When a TCP flow controlled by a congestion control algorithm does not take throughput out of a NewReno flow and vice versa, it is TCP friendly. BIC is well designed with TCP friendliness in mind. BIC also improves bandwidth scalability for the use on long fat-pipe networks (LFNs). BIC introduces *fast convergence* to fasten balancing the fairness among TCP flows. In the fast convergence, binary search increase is modified so as to slow down increasing cwnd of the flow with larger window size than other flows. These algorithms described above are classified into *loss-based congestion control* because they detect the network congestion by the packet loss.

2.2 Pacing

As we described above, TCP has been improved in many aspects. However, it is still difficult to get high throughput in TCP communication on LFNs. Many studies have been carried out to improve the performance. Self-clocking is originally a mechanism to suppress the bursty transmission by limiting the amount of data sent for RTT period to the window size. On LFNs, however, the window-size data is transmitted at once in the beginning of the RTT period and no data are sent for the rest of the RTT period. Thus TCP flows over LFNs cause RTT periodical bursty traffic. When multiple bursty flows are merged into one flow in a switch, large buffer is required to absorb the temporarily over-bandwidth packets (Fig. 2). Once the packet is lost by buffer overflow, it takes much time to recover cwnd on LFNs because it is increased in a stepwise fashion in RTT interval. The required time to recover the same cwnd is proportional to RTT when TCP is in the congestion avoidance phase.

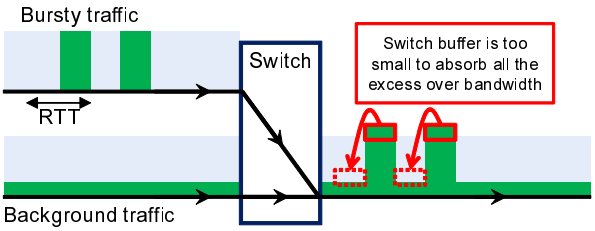

Fig. 2. Burst traffic causes the packet loss even when total traffic throughput is less than the network bandwidth

Pacing [2] is a method to smooth the bursty traffic by allocating a proper size of gap between packets to suppress the instantaneous maximum transmission speed (Fig. 3). Pacing can avoid the packet loss caused by bursty traffic and

improve the TCP performance. Pacing has grown a popular solution to the bursty behavior of the TCP flow on LFNs. However, controlling transmission speed in userland software does not work as pacing. This is because the data are first stored in the transmission queue in the kernel space and then they are sent to the network as fast as the network speed.

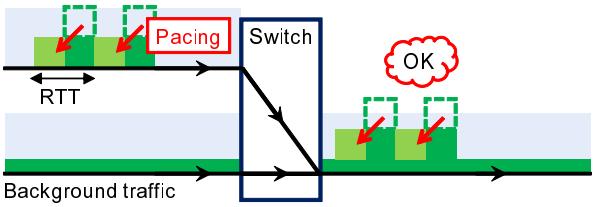

Fig. 3. Pacing

Takano et al. proposed "Precise Software Pacer (PSPacer)" [17]. PSPacer is a software implementation of pacing in the Linux kernel. PSPacer inserts a PAUSE frame, which is used in the flow control mechanism in IEEE 802.3 [1], between Ethernet frames to control the transmission speed.

Controlling the size of *inter-packet gap* (IPG), also regulated in IEEE 802.3 [1] in the *media access control* (MAC) is another way of pacing. Many network interface card (NIC) support modifying the device register to control the IPG in its MAC. *Interlayer coordination* [10] is proposed as a pacing technique for high-speed TCP communication on LFNs. IPG control is a link layer protocol technique. In interlayer coordination, however, IPG control is used for optimizing the TCP flow, i.e. optimizing the transport layer. The IPG-controlled NIC can be used for only one TCP session because this is applied to all the transmission packets of the NIC. However, this is still an effective way of pacing because it does not wastes any CPU time or memory load.

As for parallel TCP communication, network congestion occurs by the parallel flows themselves. One solution is statically calculating available bandwidth for each flow and pacing to the available bandwidth so as not to cause network congestion. However, this solution cannot apply to networks where the throughput of each flow and the number of flows are ever changing. It is shown that hardware-based fine scheduling of the packets flowing into the intermediate switch works as pacing and it can balance the throughput of each TCP flow and avoids network congestion [16]. However, it is sometimes difficult to setup such a special hardware on the network paths.

3 Adaptive auto-tuning of TCP pacing

3.1 Design

As we described above, pacing is well researched and implemented for TCP communication on LFNs. However, existing static or special hardware approach to pacing cannot always be applied. This paper proposes *adaptive auto-tuning of TCP pacing* that tune the pacing speed dynamically depending on the network situation that is difficult by static and manual approach. We introduce two types of dynamic pacing and combine them with BIC TCP congestion control in the sender. In our method, pacing speed is determined by end-to-end available bandwidth estimation. To estimate available bandwidth, no special device is used on the network path and no additional packet is transmitted to the network. In addition, there is no modification in the receiver. The first type of pacing offers three purposes: to avoid the packet loss caused by bursty traffic, to estimate the available bandwidth precisely and to evaluate the "stability" of the background traffic. The second type of pacing is to reduce the packet loss caused by excess growth of cwnd. Either one of two types of pacing is always enabled. They are switched over depending on the situation.

Congestion window pacing The first type of pacing is called *congestion window pacing* (CWP). CWP is the default pacing of our proposal. CWP limits the maximum transmission speed to $cwp\_speed = cwnd/rtt$ where *cwp_speed* is the CWP speed, *cwnd* is cwnd and *rtt* is the RTT of the network path. The first purpose of CWP is to avoid the packet loss caused by the bursty behavior of TCP traffic on LFNs. In the TCP session, inflight data size during RTT period is limited to *cwnd* and the network stack tries to transmit all the *cwnd* data during RTT period. In order to transmit *cwnd* data during RTT period, the sender must transmit packets at least *cwp_speed* and the transmission speed is limited to *cwp_speed* in CWP. Thus transmission speed is smoothed to exactly *cwp_speed* by CWP and the packet loss caused by bursty traffic is avoided. The second purpose of CWP is to estimate the available bandwidth for the TCP session. In the loss-based TCP congestion control algorithm, network congestion is detected as the packet loss. In addition, when CWP works, the transmission speed is exactly *cwp_speed*. Thus the available bandwidth at the time of packet loss is estimated precisely as *cwp_speed*. When the packet loss is detected, *cwp_speed* is recorded as *eab*, the estimated available bandwidth at the last time of the packet loss. The third purpose of CWP is to evaluate the "stability" of the background traffic. When the difference of two *eab* at the time of consecutive packet losses is small, the variation of background traffic can be considered small. Then the network path is evaluated as "stable". On the other hand, when the difference is large, the network path is evaluated as "unstable" (Fig. 4). This evaluation is used to determine the period of the second type of pacing.

Estimated available bandwidth pacing The second type of pacing is called *estimated available bandwidth pacing* (EABP). EABP limits the maximum trans-

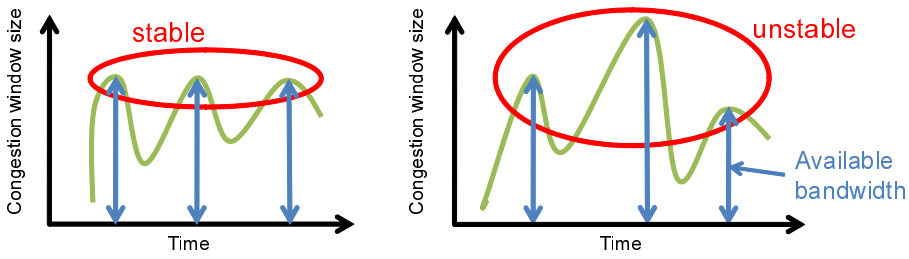

Fig. 4. Available bandwidth estimation

mission speed to *eab*,. The purpose of EABP is to reduce the packet loss caused by excess growth of cwnd. When EABP works, cwnd is not increased by ACK and the throughput is maintained. EABP begins to work when *cwp_speed* reaches more than 90% of the last-recorded *eab* and keeps working for $time_eabp$ period. At first $time_eabp$ is 1 second. When the network path is evaluated as "stable", $time_eabp$ is prolonged in an exponential backoff way: 1 second, 2 seconds, 4 seconds, ... (Fig. 5). On the other hand, when it is evaluated as "unstable", $time_eabp$ is reset to 1 second. EABP is aborted and CWP begins to work when the packet is lost or when $time_eabp$ time passes. The period of EABP is limited to $time_eabp$ because the congestion window size should be increased when available bandwidth is increased. $time_eabp$ is not reset even when EABP is aborted before $time_eabp$ passes. The "stability" is evaluated in the next CWP. This is to avoid evaluating "unstable" faultily when the packet was lost not by the network congestion but by other reasons such as frame check sequence error. Thus EABP reduce the packet loss and improve the TCP throughput on LFNs where it takes much time to recover cwnd.

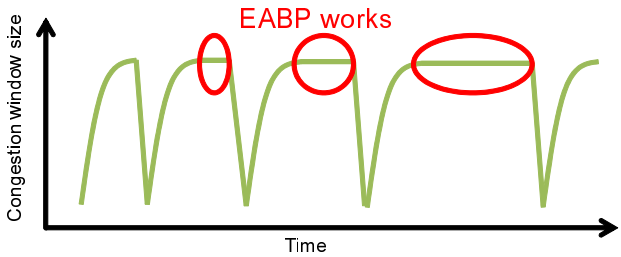

Fig. 5. Estimated available bandwidth pacing

3.2 Implementation

To evaluate our proposal, we implemented BICIPG as a TCP congestion control module in Linux kernel 2.6.30.9. BICIPG source code is based on BIC module. Pacing is implemented with MAC feature of Chelsio S310E NIC to control IPG. IPG value written to its IPG register directly in the BICIPG module during the TCP session. IPG value is calculated as $ls * (mtu + ehs)/ps - (mtu + ehs$ where ls is the wire rate (10 Gbps in this case), mtu is the maximum transmission unit (MTU), ps is the pacing speed (it is *cwp_speed* in CWP and *eab* in EABP) and ehs is the Ethernet header size. MTU value is gotten in BICIPG when the TCP session is established.

In the TCP stack in Linux kernels, there are five TCP states as shown in Table 1. In addition, We defined three BICIPG states independently in the BICIPG module as shown in Table 2. CWP works in BI_CWP_A or BI_CWP_B, and EABP works in BI_EABP. We used two hook functions to manage these state machines and the pacing speed. The first one is *bicipg_state()* that is called when TCP states is changed. In BICIPG, entering TCP_CA_CWR, TCP_CA_Recovery or TCP_CA_Loss is processed as the packet loss. In addition, when the new state is TCP_CA_CWR or TCP_CA_Recovery, new *eab* is compared to the last *eab* and the network "stability" is evaluated. The second one is *bicipg_ack()* that is called when ACK is received. BICIPG-specific processing is in this function: *cwp_speed* is recalculated with the new *cwnd* and *rtt* in CWP and the span is checked in EABP. Fig. 6 shows the BICIPG state transition diagram. BICIPG enters BI_EABP when $cwp > eab$, enters BI_CWP_B when $time_eabp$ passes and enters BI_CWP_A at the packet loss.

Table 1. TCP stack states in the Linux kernel

TCP_CA_Open	Normal, optimal code is running Initial state.
TCP_CA_Disorder	Normal, one or two packets are lost, redundant code is running.
TCP_CA_CWR	Received ECN/ECE, cwnd is being reduced.
TCP_CA_Recovery	In fast retransmission phase, congestion window is being reduced.
TCP_CA_Loss	RTO, TCP state is reset.

Table 2. Pacing states in BICIPG module

BI_CWP_A	CWP is working. Initial state.
BI_CWP_B	CWP is working, after BI_EABP time outs.
BI_EABP	EABP is working.

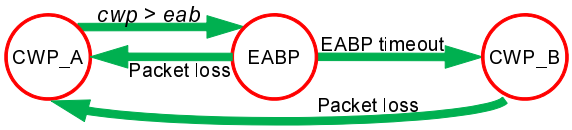

Fig. 6. BICIPG state transition diagram

4 Evaluation

4.1 Experimental setup

To evaluate our implementation, we ran a series of experiments. Fig. 7 shows the experimental network topology. All fiber connections were 10 Gbps. Four servers are used as send servers and the other four servers are used as receivers. All the paths from the senders to the receivers passed through the network emulator that adds artificial arbitrary delay to the network path. We transferred data in parallel from the senders to the receivers and recorded the sum of payload size received in every second as the TCP throughput. Each sender sent packets to one receiver and vice versa. The one-way delay of the network emulator was set to 25 ms or 100 ms. The send/receive window size were set to 300 M bytes to satisfy the bandwidth-delay product (BDP) of the 10 Gbps-200ms networks. MTU was set to 1500 bytes. The socket buffer size was set to 192 k bytes. TCP congestion control module was selected by setsockopt() system call.

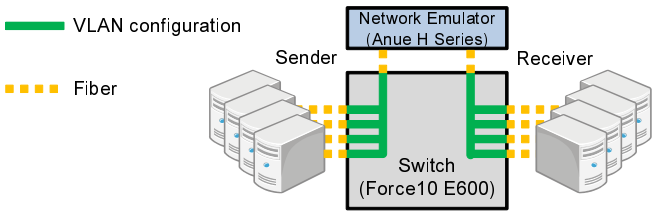

Fig. 7. Experimental Network Topology

4.2 Results

First, we compared the throughput of (a) single BIC flow and (b) single BICIPG flow on the path with 50 ms RTT and 6.4 Gbps UDP background traffic (Fig. 8). In both cases, in the slow start phase soon after the communication was started, the packet was lost and the throughput seems to reach only approximately 1.7 Gbps in Fig. 8 although there exists only 6.4 Gbps background traffic on the 10

Gbps network path. In the slow start phase, cwnd was doubled every RTT and the transmission speed was rapidly increased. On the other hand, the recording resolution of the throughput was 1 second. This is the reason Fig. 8 shows as if the packet was lost before the network congestion. In these cases, many packets were lost simultaneously and RTO happened. Then the slow start threshold was set small and the TCP stack entered the congestion avoidance phase. The congestion avoidance phase continued and cwnd had been increased in additive increase way until the 42nd second when the throughput reached approximately 3.4 Gbps and consumed nearly all the available bandwidth in both flows. Then the throughput was kept for approximately 5 seconds because the switch had buffered the packets that exceeded the wire rate of 10 Gbps until the buffer overflow. After that the packet was lost. BICIPG behaved almost the same way as BIC until this time. However, after this second packet loss, BICIPG estimated the available bandwidth approximately 3.4 Gbps and evaluated the network path as "stable". Thus EABP worked and the period of EABP was prolonged in an exponential backoff way. During the 10 minutes communication, 47 times of the packet loss were observed in BIC and 12 times of that in BICIPG. The total throughput was 3153 Mbps in BIC and 3249 Mbps in BICIPG. The packet loss was well reduced in BICIPG and the total throughput was improved more than 3%. In addition, fewer packet losses mean background traffic also lost fewer packets. BICIPG showed better characteristic in both the performance and the impact on background traffic than BIC in the experiments.

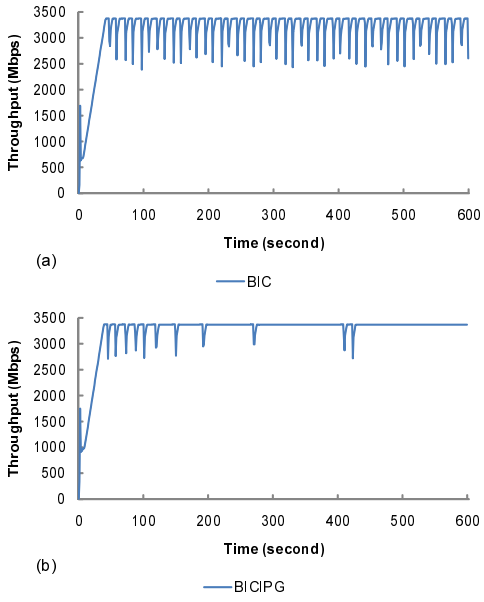

Fig. 8. Throughput: single TCP flow, 50 ms RTT, 6.4 Gbps UDP background

Then, we compared the behavior of (a) four BICIPG flows and (b) two BICIPG flows and two BIC flows on the path with 50 ms RTT (Fig. 9). When all the flows were BICIPG, EABP works and the packet loss is well reduced as well as when background traffic was constant UDP flow. This can be explained as follows: When the network congestion occurs, the packets of each TCP flow are lost and each congestion window is shrunk simultaneously. Then each TCP session enters the congestion avoidance phase. Next, cwnd are recovered "nearly" at the same time and EABP begins working. While EABP works in the flow which recovers cwnd faster than the others, all the flows recover cwnd and EABP works. Thus EABP works well in this case too. On the other hand, when two BICIPG flows and two BIC flows are on the same network path, EABP rarely worked. However, BICIPG behaved almost the same way as original BIC in this case. BICIPG invoked no performance deterioration. As a result, when EABP works, the throughput was improved more than 2% from 9024 Mbps (Fig. 9 (b)) to 9215 Mbps (Fig. 9 (a)).

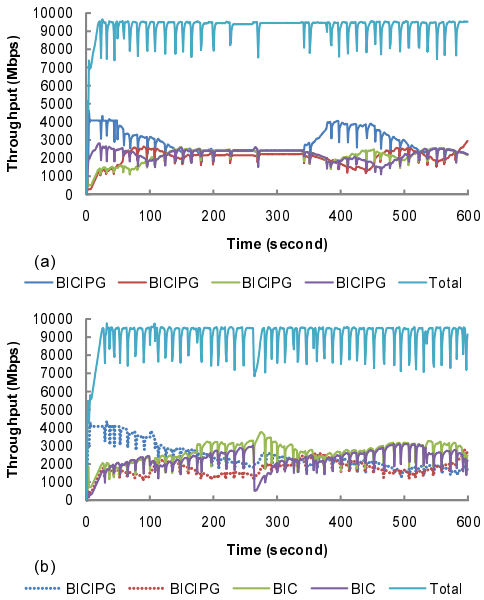

Fig. 9. Throughput: four parallel TCP flows, 50 ms RTT

Finally, we changed the delay of the network emulator into 100 ms and compared the throughput of (a) four parallel BIC flows and (b) four parallel BICIPG flows on the path with 200 ms RTT (Fig. 10).The packet loss penalty in the performance is large in this case. During 1 hour communication, the total throughput of BIC flows was 8568 Mbps and that of BICIPG was 8808 Mbps. The performance improvement was approximately 3% and BICIPG transferred

additional 108 GB compared to BIC flows in a hour. Although BICIPG flows do not seem to converge in Fig. 10 (b), they converge as well as in Fig. 9 (a) at some time.

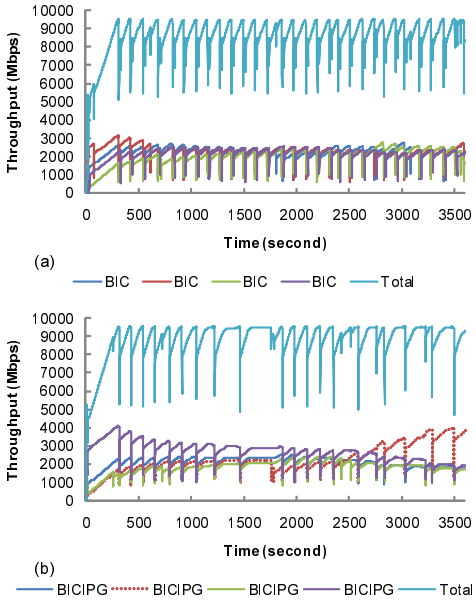

Fig. 10. Throughput: four parallel TCP flows, 200 ms RTT

5 Related work

Several researches and implementation on auto-tuning of TCP window size exist. Fisk and Feng [6] proposed *Dynamic Right-Sizing* (DRS). DRS control the advertised window size dynamically in a binary search style. The flow control of TCP is generally based on the congestion window managed in the sender. Meanwhile DRS controls the flow by advertised window in the receiver. Recent Microsoft operating systems (Windows Vista and above) offer two TCP auto-tuning mechanisms. The first mechanism is auto-tuning of the receive window (RWIN) size [5]. The RWIN size is automatically changed according to the BDP and the speed of retrieving data in the RWIN by applications. The second mechanism is Compound TCP (CTCP) [15], a congestion control algorithm introduced in Windows Vista and above. CTCP boosts the send window size when RWIN or BDP is large. These auto-tuning of TCP window size aim to speed up the performance of TCP communication on LFNs. However, they do not solve bursty traffic problem on long-distance networks.

As for pacing, the software pacing implementation in the Linux kernel, PSPacer [17], also offer dynamic pacing. However, it controls the pacing speed only by the congestion window size to prevent burst traffic. In our proposal, CWP works not only to prevent burst traffic as PSPacer but also to estimate the available bandwidth. In addition, our proposal includes pacing to the estimated available bandwidth. Our proposal is the first auto-tuning of TCP pacing including the congestion window size control.

6 Conclusion

In this paper, we proposed *adaptive auto-tuning of TCP pacing*. We introduced two types of dynamic pacing called CWP and EABP. The key idea of our proposal is combining the two types of pacing with BIC, a loss-based TCP congestion control. The pacing speed is determined dynamically by cwnd and RTT in CWP, and by estimated available bandwidth in EABP. CWP avoids the packet loss caused by bursty traffic, estimates the available bandwidth precisely and evaluates the "stability" of the background traffic. EABP stops increasing throughput and avoid the packet loss caused by the excess growth of cwnd. We implemented BICIPG as a TCP congestion control module in the Linux kernel to evaluate our method. We evaluated the performance of parallel TCP communication on pseudo LFNs Our method works well not only when background traffic is "stable" like constant UDP flow but also when it is used in parallel TCP communication. The performance improvement is especially large when the network delay is large or background traffic was "stable". Existing researches on auto-tuning is valid only for in-cluster calculation and communication. Our contribution is to focus on auto-tuning of TCP pacing on LFNs to improve the performance of inter-clusters communication for grid and cloud computing environment.

References

1. IEEE Standard for Information technology - Specific requirements - Part 3: Carrier Sense Multiple Access with Collision Detection (CMSA/CD) Access Method and Physical Layer Specifications (2008)
2. Aggarwal, A., Savage, S., Anderson, T.: Understanding the performance of TCP pacing. In: Proceedings of the Conference on Computer Communications, Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (2000)
3. Allman, M., Paxson, V., Blanton, E.: TCP Congestion Control. IETF RFC 5681 (2009)
4. Braden, R.: Requirements for Internet Hosts - Communication Layers. IETF RFC 1122 (1989)
5. Davies, J.: The Cable Guy: TCP Receive Window Auto-Tuning. TechNet Magazine (2007), http://technet.microsoft.com/en-us/magazine/2007.01.cableguy.aspx
6. Fisk, M., Feng, W.: Dynamic Adjustment of TCP Window Sizes. Los Alamos Unclassified Report 00-3221, Los Alamos National Laboratory (2000)

7. Floyd, S., Henderson, T., Gurtov, A.: The NewReno Modification to TCP's Fast Recovery Algorithm. IETF RFC 3782 (2004)
8. Jacobson, V.: Modified TCP congestion avoidance algorithm. end2end-interest mailing list (1990)
9. Jacobson, V.: Congestion avoidance and control. ACM SIGCOMM Computer Communication Review 25, 187 (1995)
10. Kamezawa, H., Nakamura, M., Tamatsukuri, J., Aoshima, N., Inaba, M., Hiraki, K.: Inter-layer coordination for parallel TCP streams on Long Fat pipe Networks. In: Proceedings of the 2004 ACM/IEEE conference on Supercomputing (2004)
11. Makino, J., Hiraki, K., Inaba, M., Ishiyama, T., Nitadori, K., Portegies Zwart, S., Groen, D., Harfst, S., de Laat, C., McMillan, S.L.W., Cosmogrid project members: Cosmogrid. In: 9th Annual Global LambdaGrid Workshop (2009)
12. Mathis, M., Mahdavi, J., Floyd, S., Romanow, A.: TCP Selective Acknowledgment Options. IETF RFC 2018 (1996)
13. Postel, J.: Transmission Control Protocol. IETF RFC 793 (1981)
14. Semke, J., Mahdavi, J., Mathis, M.: Automatic TCP buffer tuning. ACM SIGCOMM Computer Communication Review 28, 315–323 (1998)
15. Song, K., Zhang, Q., Sridharan, M.: Compound TCP: A scalable and TCP-friendly congestion control for high-speed networks. In: Proceedings of Fourth International Workshop on Protocols for Fast Long-Distance Networks (2006)
16. Sugawara, Y., Inaba, M., Hiraki, K.: Flow Balancing Hardware for Parallel TCP Streams on Long Fat Pipe Network. In: Proceedings of 2007 International Conference on Future Generation Communication and Networking (2007)
17. Takano, R., Kudoh, T., Kodama, Y., Matsuda, M., Tezuka, H., Ishikawa, Y.: Design and evaluation of precise software pacing mechanisms for fast long-distance networks. In: Proceedings of Third International Workshop on Protocols for Fast Long-Distance Networks (2005)
18. Xu, L., Harfoush, K., Rhee, I.: Binary increase congestion control (BIC) for fast long-distance networks. In: Proceedings of Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies (2004)
19. Zwart, S., Ishiyama, T., Groen, D., Nitadori, K., Makino, J., de Laat, C., McMillan, S., Hiraki, K., Harfst, S., Grosso, P.: Simulating the universe on an intercontinental grid of supercomputers. ArXiv e-prints 1001.0773v1 (2010)