ARS: Cross-layer Adaptive Request Scheduling to Mitigate TCP Incast in Data Center Networks

Jiawei Huang[§], Tian He[‡], Yi Huang[§], Jianxin Wang[§]
[§]School of Information Science and Engineering, Central South University, ChangSha, China 410083
[‡]Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA 55455
Email: jiaweihuang@csu.edu.cn, tianhe@umn.edu, huangyi90@csu.edu.cn, jxwang@csu.edu.cn

***Abstract*—In data center networks, many network-intensive applications typically suffer TCP incast throughput collapse when bursty concurrent TCP flows share a single bottleneck link. To address the TCP incast problem, we first reveal theoretically and empirically that controlling the number of concurrent flows is much more effective in reducing the incast probability than controlling the congestion window. We further propose a novel cross-layer design called Adaptive Request Schedule (ARS), which dynamically adjusts the number of concurrent TCP flows by batching application requests according to the congestion state acquired from transport layer. ARS is deployed only at the aggregator-side, while making no modification on hundreds or thousands of workers. Broad applicability is another advantage of ARS. We integrated ARS transparently (i.e., without modification) with DCTCP and TCP NewReno on NS2 simulation and a physical testbed, respectively. The experimental results show that ARS significantly reduces the incast probability across different TCP protocols and that the network goodput can be increased consistently by on average 6x under severe congestion.**

***Keywords*—Data center; TCP; incast; Congestion control**

I. Introduction

With the rise of cloud computing and storage, a significant number of online service providers like Amazon, Google, and Microsoft construct their data centers to handle core applications such as Web search and MapReduce. These applications typically implement a data flow computation model, where datasets pass through different processing stages. A lot of works have been proposed to manage computation and storage resources on servers to improve application performance [1], [2]. Recent research has shown that, however, the network is a bottleneck in such applications [3], [4]. For example, Hadoop traces from Facebook show that, on average, transferring data between successive stages accounts for 33% of the running times of jobs with reduce phases [5].

These applications concurrently transfer large amounts of data across hundreds or thousands of machines in data centers. From the viewpoint of networking, the performance of these network-intensive applications is degraded for the following reasons. First, though the main features of modern data center networks (DCNs) are high-bandwidth links (at least 1Gbps) and ultra-low round-trip propagation delay (around 100μs), the Commercial Off-the-Shelf (COTS) Ethernet switches with small-size buffers are widely used as the Top of Rack (ToR) switches in large-scale data centers to save cost. Second, data center networks generically use the commodity TCP/IP and Ethernet networks because of their ease-of-use. Furthermore, the data center applications usually adopt the barrier-synchronized and many-to-one communication pattern to achieve high performance and service reliability. All of these features together bring about the TCP incast problem [6], [7].

In a typical barrier-synchronized scenario, the aggregator server initiates requests to all workers in parallel to fetch data. When highly concurrent workflows are sent back to the aggregator, the large number of packets may exhaust the TOR switch's shallow buffer, resulting in packet losses or even TCP Time-Out. Compared to the typical 400us and 1ms RTTs in DCNs with respective 10Gbps and 1Gbps bandwidth, the 200ms idle time of Retransmission Time-Out (RTO) unavoidably leads to gross under-utilization of link capacity. Moreover, if some workflows are not able to execute timely and exceed a specified Service Level Agreement (SLA), the aggregation results may be unavailable or useless, which adversely wastes both computing and network resource. Compared with the increasingly faster access to memory or disk in novel applications (e.g., Memcached and pNFS) and the shorter packet forwarding delay in switches, TCP incast problem significantly impairs the transfer efficiency of data center networks and makes those innovative hardware unable to maximize their performance. Therefore, how to avoid Time-Out in highly concurrent flows becomes a challenging issue for network operators and researchers.

Existing proposals [6], [8], [9], [10] have developed miscellaneous schemes to tackle this problem. In the transport layer, based on the congestion notification such as ECN and RTT, several TCP variants [8], [10] shrink congestion window to avoid too many packets synchronously injecting into the bottleneck link. These transport-layer approaches could mitigate the impact of TCP incast problem to a certain degree yet still suffer from a common limitation. That is, though TCP flows cut their congestion windows, the switch buffer is still easy to overflow in the severe congestion scenarios with a large number of concurrent flows,

In this work, we argue that existing TCP protocols designed for data center only focus on the adjustment of congestion windows, which is fundamentally unable to solve the TCP incast problem caused by highly concurrent flows. We reveal theoretically and empirically that controlling the number of concurrent flows is much more effective in reducing the

incast probability than controlling the congestion windows. The reason is that fewer flows allow more statistically fair sharing of shallow buffers in switches, preventing Time-Out due to a full window of packet losses in a single TCP flow.

Specifically, to mitigate TCP incast problem in a low-cost manner, we propose a novel cross-layer approach named Adaptive Request Schedule (ARS). From the application layer, ARS schedules the requests by batches to prevent too many concurrent flows injecting into bottleneck link. While the adjustment strategy of congestion window has been extensively studied in previous congestion control research, our design joints the control of congestion window and number of flows, solving the TCP incast problem fundamentally. We briefly summarize the contributions of this paper, as follows:

- We conduct the first extensive study to exploit controlling number of concurrent flows to solve TCP incast problem. We demonstrate experimentally and theoretically why controlling number of concurrent flows is more effective in avoiding incast than cutting congestion window under severe congestion.
- We propose a cross-layer design by carefully sending the application requests by batches. Taking into account real-time state of network congestion, our ARS design rationally adjusts number of concurrent flows to resolve incast congestion. The design only needs to be deployed at the aggregator-side without any changes made to TCP protocol design, thus ensuring the minor deployment overhead and providing general support for various TCP protocols.
- By using both NS2 simulations and small-scale Linux testbed, we demonstrate that with the aid of ARS, TCP protocols such as TCP NewReno and DCTCP perform remarkably better than these without using ARS in TCP incast scenarios. Especially, ARS greatly reduces the number of Time-Out events and helps TCP NewReno and DCTCP yield up to 6x goodput improvement.

The remainder of this paper is structured as follows. In Section II, we describe our design motivation. The design detail of ARS is presented in Section III. In Section IV and V, we show the NS2 simulation and real testbed experimental results, respectively. In Section VI, we demonstrate existing approaches. Finally, we conclude the paper in section VII.

II. Design Motivation

To understand the challenges facing data center transport protocols, in this section we first compare the application latency and communication latency in data centers. Then we describe the typical barrier-synchronized scenario that motivates why the full window loss event is a critical factor to the transport layer performance. Furthermore, we demonstrate theoretically the fundamental problem of transport protocols and show that the incast probability is effectively reduced by controlling number of concurrent flows.

A. Application latency and communication latency

In the data center, end-to-end application latency is the sum of communication latency and application latency. The first one is the amount of time it takes a packet to traverse the network. The latter one is the time required to process a message or request, perform the application logic, and generate a response on servers. In the past, when most network requests resulted in disk I/Os, application latency is an issue for data center operators. For example, to avoid unacceptable response time, Facebook's applications set the limit of 100-150 sequential data accesses for each Web page returned to a browser.

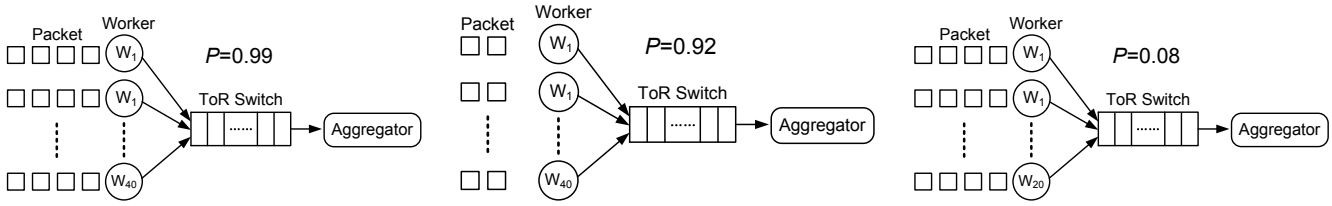
Fortunately, the application latency on the endhosts is decreasing thanks to the increased cores per server, increased DRAM capacity, and the availability of low-latency, flash-based SSDs [4], [11]. We use Memcached as an example, which is a popular, in-memory Key-Value (KV) store, deployed at Zynga and Twitter. In Memcached, the application latency has been reduced to as small as less than 10 microsecond, though it includes the time to parse the client request, search a key in the hash table, determine the location of data and generate the corresponding response [12].

Compared with the application latency, however, the communication latency is still an important issue, given the fact that COTS switches are widely deployed in data center networks. For example, a Triumph switch has 48 1Gbps ports and its 4MB buffer size is shared by all ports, which means that each port only shares 85 KB buffer space in average, namely - accommodating 56 packets with 1500 Bytes packet size. During periods of congestion, this means that an incoming packet has to wait for up to 68 microsecond before it can leave the switch, taking up to 85% of end-to-end application latency in the previously mentioned Memcached application. It should be noted that this is only queuing delay, without consideration of Time-Out under high concurrency. In this following part, we give the deep analysis of the large communication latency in the network-intensive applications.

B. Impact of full window loss

TCP detects a packet loss by two mechanisms: triple-duplicate ACKs or Time-Out. The first mechanism is faster. If a packet is lost during transmission, packets sent in the same congestion window will trigger duplicate ACKs. When a sender receives a few (usually three) duplicate ACKs, it retransmits the missing packet that was signaled by duplicate ACKs. The recovery time is about one RTT period (i.e., 10s or 100s microseconds). The second mechanism is much more time consuming. If all packets in the congestion window are lost (i.e., full window loss), the sender has to rely on the retransmission timer to retransmits the lost packets. The retransmission timer should be set large enough to help the network recover from severe congestion. For example, the recommended minimum RTO is 200 or 300 millisecond, about three orders of magnitude slower than recovery time in mechanism of duplicate ACKs.

In the barrier-synchronized transfer, single packet loss and full window packet loss have significantly different impacts on the network efficiency. In the typical barrier-synchronized scenarios, multiple servers (termed 'workers') connect to a single aggregator via a ToR switch. The data blocks are striped across a number of workers. Each worker has its own data



(a) At the beginning (b) After cutting congestion window (c) After cutting number of flows

Fig. 1: Cutting congestion window vs. number of flows

fragment denoted as Server Request Unit (SRU), which is typically less than 100KB [8]. The aggregator initiates requests to all workers in parallel to fetch all SRUs. When the workers receive requests, they transmit data back to the aggregator. The next round of requests cannot be initiated until all workers have finished transferring data in the previous round.

Obviously, for a typical COTS TOR switch with 44 1Gbps ports, if there is no full window loss in the whole transfer, transmitting 44 100KB-SRUs from different workers via a 1Gbps link will only take around 40ms. However, if the shallow buffer of TOR switch is overloaded under highly concurrent flows traversing the bottleneck link, TCP connections may suffer full window loss, resulting into Time-Out. Once Time-Out happens, the new round of transfer must wait for the stalled TCP flows. The idle period of RTO (i.e., 200ms) unavoidably causes a significant TCP throughput collapse. Under this condition, the duration of transmitting 44 100KB-SRUs will be elongated to at least 240 ms, meaning the almost 85% throughput reduction.

C. Fundamental problem of transport protocols

It is well-known that TCP protocols such as DCTCP and TCP NewReno [13] control the sending rate by adjusting congestion window size. These approaches perceive congestion state by different indications, including packet losses, ECN and RTT, and then adjust their congestion windows.

We firstly analyze the impact of congestion window and flow concurrency on the incast probability P. Let B, C and RTT_{min} denote the size of switch buffer, link capacity and round-trip propagation delay (i.e., RTT without queueing delay), respectively. When the number of in-flight packets of n concurrent flows exceeds the total capacity of buffer size and bottleneck link, packet losses and even Time-Out will occur. We use Equation (1) to illustrate this condition:

$$nw \times MSS > C \times RTT_{\min} + B, \tag{1}$$

where MSS is the size of a TCP segment, w is the average congestion window size of all n TCP flows. Since all n synchronized flows share the bottleneck link fairly using statistical multiplexing, the packet loss rate p is calculated as

$$p = 1 - \frac{C \times RTT_{\min} + B}{nw \times MSS}. \tag{2}$$

It is known that TCP Time-Out is mainly caused by a full window of packet losses [14], we calculate the probability of a full window of packet losses as the Time-Out probability P_{TO}:

$$P_{TO} = p^w = (1 - \frac{C \times RTT_{\min} + B}{nw \times MSS})^w. \tag{3}$$

Since TCP incast happens when at least one flow experiences Time-Out in n flows, we obtain the incast probability P as

$$\begin{aligned} P &= 1 - (1 - P_{TO})^n \\ &= 1 - (1 - (1 - \frac{C \times RTT_{\min} + B}{n \times w \times MSS})^w)^n. \end{aligned} \tag{4}$$

From Equation (4), we find that P is determined by both the congestion window w and number of flows n. We intuitively deduce that, the value of P is reduced with smaller w or n.

However, compared with decreasing the congestion window, decreasing the number of flows allows the same buffer room to accommodate more packets for each flow. Consequently, under statistical multiplexing, each flow has a higher chance to enqueue at least one packet. The chance of a full window of packet losses is reduced. Here, we use an example to demonstrate that decreasing the number of flows is remarkably more efficient in reducing probability of full window losses and incast event.

Fig.1 (a) shows that 40 concurrent flows share a bottleneck link with a TOR switch. The total capacity of switch buffer size and bottleneck link is only 60 packets. Let us assume the congestion window size of each flow is 4 packets (i.e., totally 160 in-flight packets). Fig.1 (b) and (c) compare the effect of cutting congestion window with that of cutting number of flows. Fig.1(a) shows that 100 packets are dropped due to overflow. Based on Equation (3) and (4), we derive that the probability of full window losses (i.e., Time-Out probability P_{TO}) of each flow is $(\frac{5}{8})^4 = 0.15$, and the incast probability P is 0.99. As shown Fig.1 (b), if all 40 flows halve their congestion windows, P_{TO} and P become $(\frac{1}{4})^2 = 0.06$ and 0.92, respectively. In contrast, Fig.1 (c) shows that the number of flows is cut into half, and the congestion window size is still 4. We find that, after cutting the number of flows, P_{TO} is $(\frac{1}{4})^4 = 0.004$, and P becomes as small as 0.08 in comparison with 0.92 in case of halving the window size.

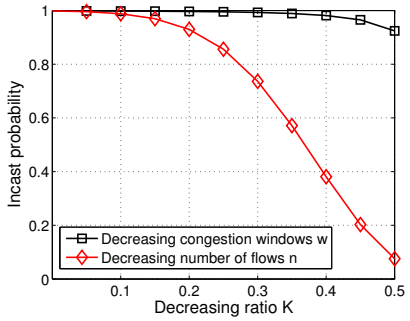


Fig. 2: Incast probability with decreasing w and n

Furthermore, we use Fig.2 to compare the incast probability when w or n is decreased. Here, K is the decreasing ratio, describing how much w or n is reduced. When K is increasing from 0 to 0.5, Fig.2 shows that, incast probability is significantly reduced in cutting number of flows, while cutting congestion window has negligible effect on incast probability. This confirms that, in high flow concurrency (i.e., $n = 40$), large number of in-flight packets still easily brings about Time-Out, even when w is cut from 4 into the very small value of 2 (i.e., $K = 0.5$).

D. Summary

Our analysis of these transport protocols leads us to conclude that (i) TCP congestion control alone cannot solve the TCP incast problem by adjusting congestion window, and (ii) compared with controlling congestion window, adjusting the flow concurrency is a much more efficient strategy to mitigates the incast problem. These conclusions motivate us to tackle TCP incast problem through a cross-layer design to overcome the limitation of congestion control, which exploits congestion information from transport layer to regulate the number of concurrent TCP flows from the application level.

III. The Design of ARS

In this section, we describe the design detail of ARS, including how to obtain the optimal batch size and adjust the batch size according to the congestion state.

A. Design insight

Data center's applications (e.g., Hadoop, pNFS and Web search) usually adopt barrier-synchronized communication pattern, in which the aggregator will not issue a new round of requests until all requested data are received. This means that the number of requests issued in a round determines the number of concurrent TCP connections.

The main idea of ARS is to schedule a round of requests into several batches. Specifically, after issuing a batch of requests, the aggregator will not issue a new batch of requests until the successful receipt of all the requested data in the current batch. Moreover, the size (i.e., number of flows) of the next batch depends on the congestion state acquired from transport layer. By batching requests, ARS adjusts the number of concurrent TCP connections to avoid the Time-Out event.

The design of ARS involves several key challenges. First, we need to obtain the optimal batch size, taking incast probability into consideration. Second, we need a low-cost adjustment strategy of batch size to deal with rapid changes of network dynamics. Last, it should be compatible with existing transport layer protocols for practical deployment.

B. Optimal batch size

In our design, the first issue is to obtain the optimal batch size. Let W denote the maximum number of outstanding packets in a round of RTT. Suppose that there are n concurrent flows in a batch, we obtain the maximum number of in-flight packets Y_a as

$$Y_a = n \times W. \tag{5}$$

Let Y_b denote the total number of packets that can be accommodated in link pipeline and switch buffer as

$$Y_b = \frac{C \times RTT_{\min} + B.}{MSS}. \tag{6}$$

To avoid the Time-Out event and in the meantime accommodate as more concurrent flows as possible, the optimal batch size n^* is set as the minimum value satisfying $Y_a > Y_b$.

Since n^* should be an integer, we have

$$n^* = \lfloor \frac{Y_b}{W} \rfloor + 1. \tag{7}$$

The duration of Time-Out is the key to throughput performance in the whole transfer process. In the following, we calculate the Time-Out probability of concurrent N flows.

Note that here we only compare the Time-Out probability in TCP's slow-start stage with following two reasons. First, TCP starts with the slow-start phase, namely exponentially increasing congestion window, which is very aggressive and the main reason causing Time-Out [6]. The other reason is that, server request unit is typically very small (normally less than 100KB [8], [10]) and usually can be finished transmission in the slow-start stage (e.g., the transfer of a 60KB server request unit only needs 6 rounds of RTT).

We compare the Time-Out probability of concurrent N flows with/without ARS in following two cases.

- When ARS is enable, the number of concurrent flows (i.e., batch size) is set as n^*. For each flow, its congestion window exponentially increases in every round of RTT during slow-start process. The congestion window w_k for the k-th round of RTT is 2^{k-1}. Since the size of server request unit Y_s (pkts) determines how many rounds of RTT is needed in the transfer, we get the maximum number of outstanding packets in a round of RTT, $W = 2^{\lfloor log_2(Y_s+1) \rfloor - 1}$. Then the incast probability P is calculated as

$$\begin{aligned} P &= 1 - (1 - (1 - \frac{Y_b}{n^* \times W})^W)^{n^*} \\ &= 1 - (1 - (1 - \frac{Y_b}{n^* \times 2^{\lfloor log_2(Y_s+1) \rfloor - 1}})^{2^{\lfloor log_2(Y_s+1) \rfloor - 1}})^{n^*}. \end{aligned} \tag{8}$$

- When ARS is disabled, all N flows are concurrently transferred. Assume the packets are evenly distributed among each flow when the number of in-flight packets reaches Y_b, then the average congestion window w of each flow is Y_b/N. In the next round of transfer, all flows use multiplicative-increase to enlarge their respective congestion windows by 2, and thus $2Y_b$ packets are injected into the network system. Since the switch buffer and link pipeline only accommodate Y_b packets, a half of packets are lost and the packet loss rate becomes 0.5. In this case, we get the incast probability P' as

$$P' = 1 - (1 - (0.5)^{2Y_b/N})^N. \tag{9}$$

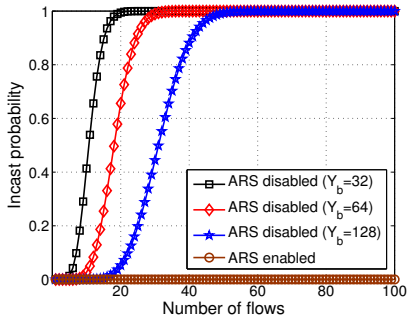


Fig. 3: Numeric comparison of incast probability

The numeric comparison of incast probability is shown in Fig. 3. We change the number of TCP flows N and the number of packets accommodated in the switch buffer and link pipeline Y_b. Each flow sends 64KB data with 1.5 KB packet size. Fig. 3 shows, when ARS is disabled, with the increasing number of flow N, the incast probability becomes larger due to the severer congestion. Furthermore, we find that it is much easier to enter TCP incast with the smaller Y_b. If Y_b is 128, the incast probability reaches 1 when n is 52, indicating that the network system accommodates more than 50 concurrent TCP flows. When Y_b becomes 32 or 64, the corresponding maximum numbers of flows, without introducing incast, decrease to as small as 20 or 32, respectively. However if ARS is used, the incast probability is always kept near 0, showing that the Time-Out event is greatly eliminated.

C. Adjustment strategy of batch size

Based on above analysis, we obtain the optimal batch size to avoid severe congestion. Under dynamic network traffic scenarios, however, it is not reasonable to adopt a fixed batch size. Thus, we design the adjustment strategy shown in Algorithm 1, which consists of the following two modules.

1) **Cross-layer Congestion Detector**: ARS employs congestion information from transport layer to determine the congestion state. Specifically, whether the aggregator receives out-of-order TCP packets or not is used as congestion indicator. Upon receiving a packet, the congestion indicator CI is set to 1 if the sequence number of arriving packet is out-of-order.

2) **Request scheduler:** When ARS starts working, it sends n^* requests. However, when the newly arriving background flows inject packets into the bottleneck link, the buffer overflow will happen easily because the network system is not able to accommodate all in-flight packets. Thus, if the network congestion is detected, n should be adjusted to avoid severe congestion (i.e., Time-Out).

Algorithm 1: Request scheduling algorithm

Initialization:
$n \leftarrow n^*$
$CI \leftarrow 0$

On receiving a packet from TCP flow i:
begin
  if the packet is out-of-order **then**
    $CI \leftarrow 1$

On finishing receiving all requested data in current batch:
begin
  if $CI == 1$ **then**
    $n \leftarrow \lfloor \frac{C \times RTT_{\min}}{W \times MSS} \rfloor + 1.$
  else
    if $n < n^*$ **then**
      $n \leftarrow n + 1$
  $CI \leftarrow 0$
  send a batch of $min(\lfloor n \rfloor, N)$ requests

Here, if CI becomes 1, we set n as a smaller value in conservative manner to prevent such packet loss and Time-Out event. Since the bandwidth-delay product (i.e., $C \times RTT_{\min}$) of bottleneck link is typically smaller than switch buffer size B due to very small $RTT_{\min}$ in data center, we only use $C \times RTT_{\min}$ to cap the batch size, leaving switch buffer to accommodate the potential background traffic. Specifically, when congestion is detected (i.e., CI is 1), the batch size (i.e., number of flows) n is set as

$$n = \lfloor \frac{C \times RTT_{\min}}{W \times MSS} \rfloor + 1, \tag{10}$$

where W is the maximum number of outstanding packets in a round of RTT.

If CI is 0, the batch size n increases to n^* in additive manner (i.e., $n = n + 1$) to probe available bandwidth. Before the aggregator issues a new batch of $min(\lfloor n \rfloor, N)$ requests, the client sets CI to 0. Since ARS gently increases the batch size by at most one per batch. Even the traffic from the added single flow brings about buffer overflow and packet losses, all concurrent flows will bear the packet losses. It is a very small chance for a single flow to experience full window losses and trigger Time-Out.

We should note that, all these implementation issues can be addressed without modifying the transport protocols – we simply adjust the number of requests to control the concurrency of the original transport protocols. By doing so, the transport

protocols automatically avoid the incast problem when the congestion happens, thereby achieving higher network goodput.

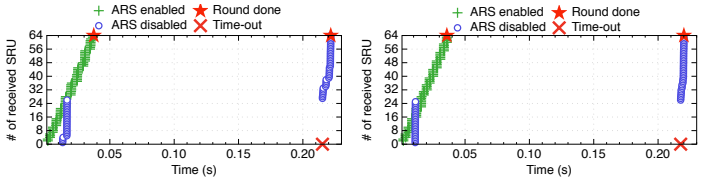
IV. Simulation Evaluation

In this section, to test ARS's broad applicability and effectiveness, we use network simulator NS2 to evaluate the performance of DCTCP and TCP NewReno with ARS enabled and disabled. Note that, DCTCP is a well-known transport protocol designed for data center networks, while TCP NewReno is the most widely used TCP version. For simplicity, we use symbols ARS_{NR} and ARS_{DC} to denote TCP NewReno and DCTCP both with ARS enabled, respectively.

In the simulation test, multiple workers are connected to an aggregator via a single ToR switch with 64KB shallow buffer per output port. We set the marking threshold to 20 for DCTCP according to [8]. Since the topology is homogeneous, the bandwidth of all links is set to 1Gbps, and the round-trip propagation delay is 100μs. RTO_{min} is set to 200ms as default setting for most Linux kernels. The TCP packet size is set to 1KB and the unit size of server request is 64KB. We use goodput perceived in application layer as our performance metric.

A. Basic performance

In this test, we investigate the performance of batch control process in detail. The aggregator sends requests to 64 workers to fetch their own data. Then, each worker will send its response to the aggregator. Fig. 4 shows the distribution of moments when the aggregator receives the responses.

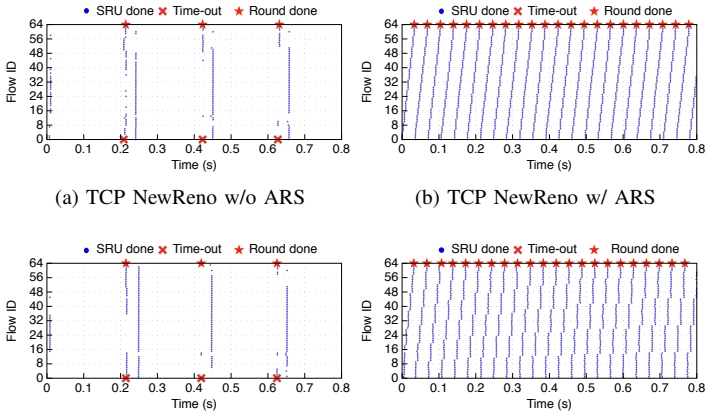


(a) TCP NewReno w/ and w/o ARS (b) DCTCP w/ and w/o ARS

Fig. 4: The serial number of received response over time

In Fig. 4 (a) and (b), both TCP NewReno and DCTCP experience Time-Out. At about 0.22s, the timer is fired and other server response units finish right after this moment, indicating that some flows experience Time-Out events and the whole transfer process is stalled by such flows for 200ms. However with ARS enabled, the time distribution of receiving data is ladder-like, which means a round of requests is scheduled into several batches with different batch size, gradually adapting to network congestion. There is no Time-Out and all 64 responses are finished in much shorter time, about 0.04s.

Next, we change the simulation scenario from a single round to multiple rounds of requests. Fig. 5 shows the time when each flow finishes the transfer of its own data with ARS enabled or disabled. The dot at (t, ID) represents that the IDth flow completes the transfer at time t. As shown in Fig. 5 (a) and (b), by batching requests, TCP NewReno's whole transfer time has 6x reduction as 6 rounds of requests finishes in 0.22s while only a single round of requests can be done without ARS. From Fig. 5 (c) and (d), we find that ARS also significantly quickens the whole transfer for DCTCP. Without ARS, the goodput still experiences collapse if 64 workers simultaneous transmit data to an aggregator. In Fig. 5 (d), we observe that *rungs* of each *ladder* are taller than that in Fig. 5 (b). This is because ECN-based DCTCP is more effective than loss-based TCP protocols in controlling queue length, and thus the batch sizes of DCTCP is typically larger than that of TCP NewReno.



(a) TCP NewReno w/o ARS (b) TCP NewReno w/ ARS

(c) DCTCP w/o ARS (d) DCTCP w/ ARS

Fig. 5: The distribution of moments when server request unit is completely transmitted.

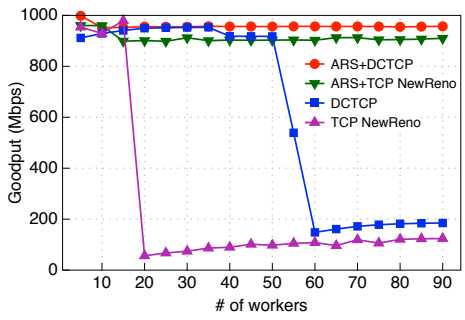
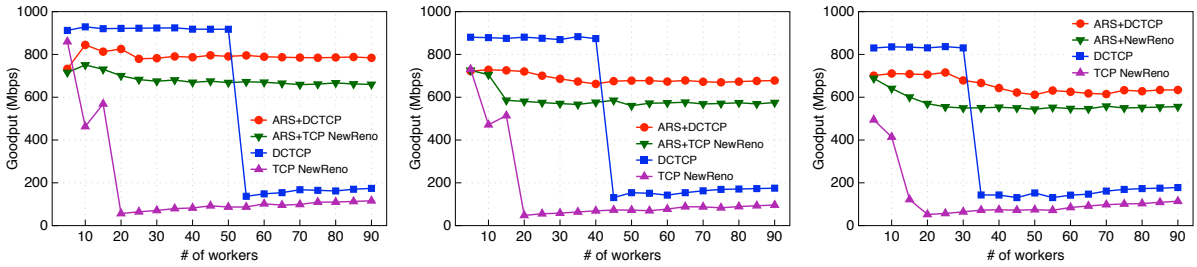


Fig. 6: Network goodput with different number of flows

To evaluate the overall network performance, we test the total goodput with up to 90 workers. In Fig. 6, we find that TCP NewReno experiences severe degradation when the number of workers rises to 20. DCTCP utilizes ECN as more accurate congestion signal, and therefore achieves very high goodput until the number of workers increases to 50. Due to the limitation of congestion control in TCP protocols, DCTCP still experiences goodput collapse when flow concurrency increases. By adaptively adjusting batch size, ARS with both TCP NewReno and DCTCP achieves high goodput as the number of workers increases.



(a) 2 background TCP flows (b) 3 background TCP flows (c) 4 background TCP flows

Fig. 7: Performance of different schemes under background workload

B. Impact of background traffic

According to the measurement results in [8], long TCP flows are quite common in data center networks. The 75th percentile and 95th percentile of concurrent large flows sharing a ToR switch is 2 and 4, respectively. Hence, we examine the performance of ARS with the long-lived background TCP flows in this test.

We design the experiments that contain 2, 3 and 4 long-lived TCP flows and each of them continuously sends data throughout the test. In Fig. 7, the goodput of each scheme experiences decline as the number of background TCP flows increases. Furthermore, both DCTCP and TCP NewReno experience striking performance degradation earlier. We also find that the goodput of DCTCP is initially higher than ARS_{DC}. This is because that ARS cuts the number of concurrent flows to mitigate congestion, thus ARS_{DC} shows relatively less aggressive in competing for available bandwidth. However, under highly concurrent flows, ARS still achieves much higher goodput compared to approaches without ARS.

V. Testbed Evaluation

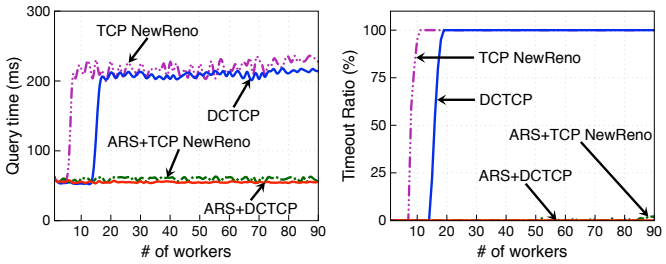
We use a real testbed to evaluate ARS performance in two typical scenarios (Web search and MapReduce-like application). We implement our code patches into the Linux kernel according to the ARS algorithm, which contains only about 20 lines of additional code.

A. Parameters and topology

The testbed is made up of four servers, which are DELL T1500 workstations with Intel 2.66 GHz dual-core CPU, 8 GB DDR3, and 500 GB hard disk. Specifically, one of the four servers is with four 1Gbps NIC cards to act as the ToR switch and other servers all connect to this server. We denote this server as 'switch' in the following parts. The other servers are equipped with a single 1Gbps NIC card and act as workers and aggregator. One of the two workers is only responsible for sending background flows. All servers are running CentOS 5.5 with Linux kernel 2.6.38 with our patches applied. The RTT without queuing delay is approximately 100μs between any two servers. Since we limit the link speed of switch output port to 100Mbps, the oversubscribed link will be the bottleneck. Therefore, we use a worker to emulate multiple workers sending data to the aggregator via the bottleneck link, which is similar to [15]. The buffer size of the bottleneck link is 64KB. The packet size and RTO_{min} is set to 1KB and 200ms, respectively.

B. Web search application scenario

We choose the large Web search service as the first case study. Here, we focus on the search response time (SRT), that is, the delay between when the query is sent by the aggregator and when the response data is completely rendered by all workers. The response data size of each worker is 500KB/n, where n is the total number of workers.

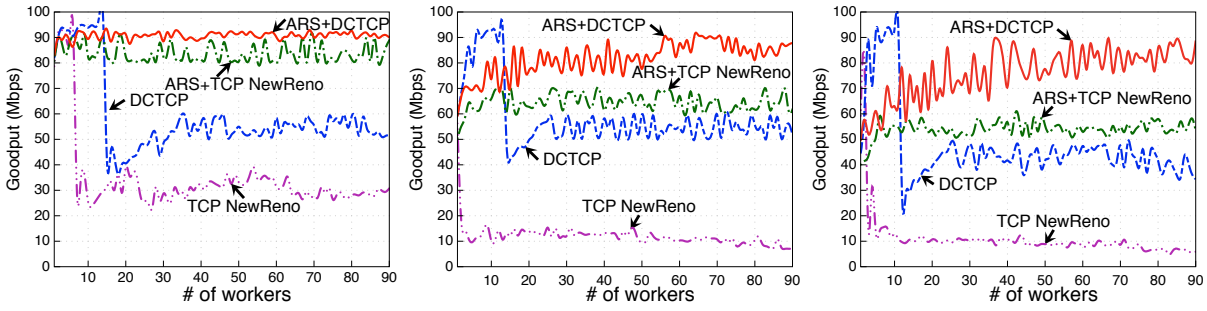


(a) Search response time (b) Time-out ratio

Fig. 8: Web search application

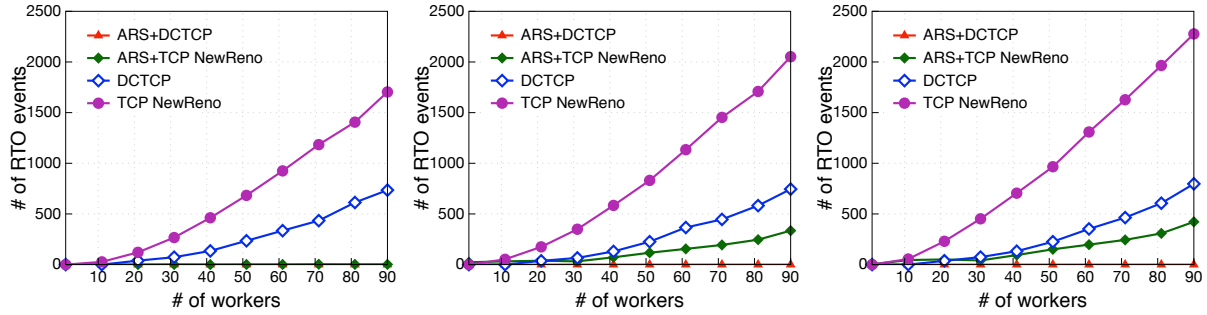
We measure SRT when n increases from 1 to 90, as shown in Fig. 8 (a). During the experiments, ARS_{DC} shows generally low SRTs ranged between 54 ms and 60 ms in all cases. ARS_{NR} also achieves at most 63 ms SRT. However, DCTCP's SRT falls to around 215 ms and TCP NewReno's SRT increases by up to 230 ms because of the severe incast congestion.

We also measure the timeout ratio, the fraction of queries that suffer at least one timeout as shown in Fig. 8 (b). TCP NewReno suffers at least one Time-Out event among the workers in most experimental rounds when the number of workers is more than 8, and it directly results in high SRT. DCTCP starts to experience Time-Out when the number of workers is 15 since DCTCP cannot prevent TCP incast congestion when



(a) No background flow (b) 1 background TCP flow (c) 2 background TCP flows

Fig. 9: The goodput of each scheme in MapReduce-like application



(a) No background flow (b) 1 background TCP flow (c) 2 background TCP flows

Fig. 10: The number of Time-out events in MapReduce-like application

the number of workers becomes larger. Finally, we observe that ARS_{DC} suffers no Time-Out in all cases and ARS_{NR} maintains the very small Time-Out ratio as well, which directly results in the low SRT.

C. MapReduce-like application scenario

In this test, a receiver generates a query to each sender, and each of them immediately responds with 128 KB of data. First, we test ARS_{DC} and ARS_{NR} without background flows and compare to the results with original TCP NewReno and DCTCP. Fig. 9 (a) shows that both ARS_{DC} and ARS_{NR} utilize more than 80% link bandwidth. DCTCP's performance is better than TCP NewReno, but still experiences goodput degradation when number of workers is larger than 15.

We present the total number of Time-out events in Fig. 10 (a). No Time-Out event is found in the result of ARS_{DC} and the maximum number of RTO events for ARS_{NR} is 5. This is because that ARS avoids congestion by adaptively batching requests, reducing the number of concurrent flows via the bottleneck link. TCP NewReno and DCTCP experience a large number of RTO events as the number of workers increases, which is validated by the performance degradation shown in Fig. 9 (a).

Next, we add up to 2 long-lived background TCP flows to test the performance under background workload. The background TCP flows are sent from the worker that is only responsible for sending background flows and we limit the output rate of this server to 100Mbps to avoid overwhelming the switch buffer. The goodput of each scheme is illustrated in Fig. 9 (b) and 9 (c), respectively. Although the goodput of ARS_{DC} and ARS_{NR} are lower than that without background workload due to congestion caused by TCP background flows, they both achieve significantly higher goodput than the original TCP NewReno and DCTCP, which suffer from remarkably goodput collapse.

Fig. 10 (b) and (c) show the number of RTO events corresponding to the experiments performed in Fig. 9 (b) and (c), respectively. The number of Time-Out events for TCP NewReno exceeds 2000 for both experiments. Therefore, its goodput is almost an order of magnitude lower than the link capacity. DCTCP also experiences high number of Time-Out events under highly concurrent flows. ARS_{NR} experiences higher number of RTO events due to severe congestion resulted from background TCP flows. However, it still outperforms the original TCP NewReno and DCTCP as the total number of workers increases. ARS_{DC}, in this case, maintains almost zero Time-Out events and achieves the highest goodput.

VI. Related Works

Many solutions have been proposed to address TCP incast problem. Since the mismatch of RTO_{min} (at least 200ms) and RTT (hundreds of microsecond) directly leads to TCP throughput collapse, a well-known solution was proposed to use fine-grained timer to estimate RTT [6]. In this way, it largely quickens the process of loss recovery. Nonetheless, this solution may stir up a multitude of spurious Time-outs and unnecessary retransmissions, which waste available bandwidth and degrade delivery efficiency. Other schemes also attempt to alleviate TCP incast problem by modifying TCP parameters, including decreasing duplicate ACK threshold [14], adding a random factor in RTO calculation to de-synchronize retransmission [6]. Although these proposals could alleviate the impact of Time-Out or reduce its frequency, they still cannot fundamentally address TCP incast problem.

Furthermore, various proposals target enhancing TCP protocol to address TCP throughput collapse. In order to perform accurate congestion control, DCTCP [8] leverages Explicit Congestion Notification (ECN) to adjust congestion window size. Thereby, DCTCP not only reduces the queueing delay and but also achieves high throughput. To maintain TCP self-clocking, TCP-PLATO [16] introduces *labelling* system to ensure that *labelled* packets are preferentially enqueued at switch. Therefore, the sender can utilize duplicate ACK to trigger fast retransmission instead of waiting for Time-Out. To avoid Time-Out, Packet Slicing [17] uses only ICMP messages to adjust the IP packet size with low overhead, allowing the switch buffer to accommodate more packets. On the receiver side, ICTCP [10] adaptively adjusts the advertisement window to control the aggregate throughput. Similarly, PAC [18] throttles the sending rate of ACKs on the receiver to prevent incast congestion.

Compared with the enhanced TCP protocols focusing on the congestion window adjustment, our solution ARS tackles the TCP incast problem by controlling the number of concurrent flows. This key difference enables ARS to reduce significantly Time-Out probability under highly concurrent flows, where existing protocols become ineffective. Moreover, ARS is easily deployed only at the aggregator-side, avoiding modifying any switches or workers.

VII. Conclusion

We have presented ARS, a novel adaptive request scheduling design that mitigates TCP incast problem under highly concurrent flows in data center networks. ARS utilizes end-to-end congestion signals to perform cross-layer congestion control of the concurrent and bursty flows. ARS is deployed only at the aggregator, without any hardware modifications on switches and thousands of servers. ARS is a supporting design compatible with a wide range of transport control protocols. In other words, ARS can obtain great goodput improvement without upgrade in existing transport protocols. To test ARS's broad applicability and effectiveness, we transparently integrate ARS with DCTCP and TCP NewReno and evaluate ARS with both NS2 simulations and small-scale Linux testbed. The results indicate that, with ARS enabled, DCTCP and TCP NewReno significantly reduce the Time-Out probability, therefore alleviating the incast problem. The network goodput is remarkably increased by up to 6x. In future work, we plan to study ARS in various data center network topologies.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (61572530, 61502539, 61462007, 61420106009).

References

[1] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, E. Harris, Reining in the outliers in mapreduce clusters using mantri, in: Proc. USENIX OSDI, 2010, pp. 1–16.

[2] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, I. Stoica, Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling, in: Proc. EuroSys, 2010, pp. 265–278.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat, Hedera: Dynamic flow scheduling for data center networks., in: Proc. USENIX NSDI, 2010, pp. 19–19.

[4] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, S. Sengupta, Vl2: A scalable and flexible data center network, in: Proc. ACM SIGCOMM, 2009, pp. 51–62.

[5] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, I. Stoica, Managing data transfers in computer clusters with orchestra, in: Proc. ACM SIGCOMM, 2011, pp. 98–109.

[6] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, B. Mueller, Safe and effective fine-grained tcp retransmissions for datacenter communication, in: Proc. ACM SIGCOMM, 2009, pp. 303–314.

[7] J. Zhang, F. Ren, C. Lin, Survey on transport control in data center networks, IEEE Network 27 (4) (2013) 22–26.

[8] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, M. Sridharan, Data Center TCP (DCTCP), in: Proc. ACM SIGCOMM, 2010, pp. 63–74.

[9] J. Zhang, F. Ren, X. Yue, R. Shu, C. Lin, Sharing Bandwidth by Allocating Switch Buffer in Data Center Networks, IEEE J. Sel. Areas Commun. 32 (1) (2013) 39–51.

[10] H. Wu, Z. Feng, C. Guo, Y. Zhang, Ictcp: incast congestion control for tcp in data-center networks, IEEE/ACM Trans. Netw. 21 (2) (2013) 345–358.

[11] H. Ballani, P. Costa, T. Karagiannis, A. Rowstron, Towards predictable datacenter networks, in: Proc. ACM SIGCOMM, 2011, pp. 242–253.

[12] R. Kapoorl, G. Porter, M. Tewari, G. M. Voelker, A. Vahdat, Chronos: Predictable low latency for data center applications, in: Proc. SOCC, 2012.

[13] S. Floyd, T. Henderson, A. Gurtov, The newreno modification to tcp's fast recovery algorithm, RFC 2582.

[14] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, S. Seshan, Measurement and analysis of tcp throughput collapse in cluster-based storage systems., in: Proc. USENIX FAST, 2008, pp. 1–14.

[15] C. Wilson, H. Ballani, T. Karagiannis, A. Rowtron, Better never than late: Meeting deadlines in datacenter networks, in: Proc. SIGCOMM, Vol. 41, 2011, pp. 50–61.

[16] S. Shukla, S. Chan, A. S. W. Tam, A. Gupta, Y. Xu, H. J. Chao, Tcp plato: Packet labelling to alleviate time-out, IEEE J. Sel. Areas Commun. 32 (1) (2014) 65–76.

[17] J. Huang, Y. Huang, J. Wang, T. He, Packet slicing for highly concurrent tcps in data center networks with cots switches, in: Proc. ICNP, 2015, pp. 22–31.

[18] W. Bai, K. Chen, H. Wu, W. Lan, Y. Zhao, Pac: Taming tcp incast congestion using proactive ack control, in: Proc. ICNP, 2014, pp. 385–396.