Swarm-based Incast Congestion Control in datacenter Serving Web Applications

Haoyu Wang
University of Virginia
Department of Computer Science
Charlottesville, Virginia 22904, USA
hw8c@virginia.edu

Haiying Shen
University of Virginia
Department of Computer Science
Charlottesville, Virginia 22904, USA
hs6ms@virginia.edu

Guoxin Liu
Clemson University
Department of ECE
Clemson, South Carolina 29604, USA
guoxinl@clemson.edu

ABSTRACT

In Web applications served by datacenter nowadays, the incast congestion at the front-end server seriously degrades the data request latency performance due to the vast data transmissions from a large number data servers for a data request in a short time. Previous incast congestion control methods usually consider the direct data transmissions from data servers to the front-end server, which makes it difficult to control the sending speed or adjust workloads due to the transient transmission of only a few data objects from each data server. In this paper, we propose a Swarm-based Incast Congestion Control (*SICC*) system. *SICC* forms all target data servers of one request in the same rack into a swarm. In each swarm, a data server (called hub) is selected to forward all data objects to the front-end server, so that the number of data servers concurrently connected to the front-end server is reduced, which avoids the incast congestion. Also, the continuous data transmission from hubs to the front-end server facilitates the development of other strategies to further control the incast congestion. To fully utilize the bandwidth, *SICC* uses a two-level data transmission speed control method to adjust the data transmission speeds of hubs. A query redirection method further reduces the request latency by balancing the transmission remaining times between hubs. Our experiments in simulation and on a real cluster demonstrate that *SICC* outperforms other incast control methods in improving throughput and reducing the data request latency.

KEYWORDS

Incast congestion, Congestion Control, Data Center

ACM Reference format:
Haoyu Wang, Haiying Shen, and Guoxin Liu. 2017. Swarm-based Incast Congestion Control in datacenter Serving Web Applications. In *Proceedings of SPAA '17, Washington DC, USA, July 24-26, 2017,* 10 pages.
DOI: http://dx.doi.org/10.1145/3087556.3087559

1 INTRODUCTION

Web applications, such as online social networks (e.g., Facebook), Web search systems (e.g., Google) and online content publishers (e.g., Youtube), become the top sources of Internet traffic today [1]. The datacenter serving these applications usually support tremendous workloads. For example, Facebook serves a billion reads per second [2]. It is important to guarantee that the data requests from users are served successfully with low latency because it affects the quality of experience of users and also is negatively proportional to the incomes of the Web application providers. Take Amazon for example, its sale degrades by one percent if the latency of its Web presentation increases as small as 100ms [3]. The typical data request latency inside a storage system of Yahoo is on larger than 100ms [4] to meet the user satisfaction. However, the packet loss always occurs during data requesting due to traffic congestion and the bandwidth usually becomes the bottleneck of the performance [5–7]. The traffic congestion also greatly increase the data request latency due to the retransmissions of dropped packets. Therefore, it is important to avoid congestion caused by data requests.

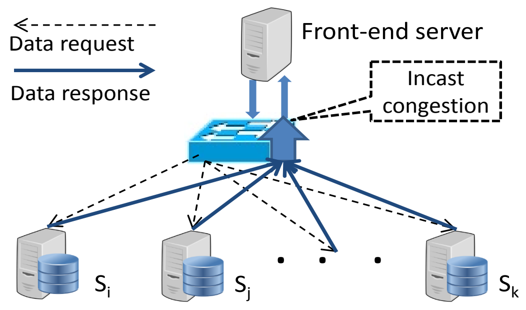


Figure 1: An example of incast congestion.

In Web applications, a data request for a Web page presentation needs to retrieve thousands of data objects currently [2, 8]. As shown in the Figure 1, for a data request, the front-end server sends out data queries concurrently to all targeted data servers, and receives hundreds or thousands of data responses simultaneously. The heavy network traffic in a short time may not reach the front-end server in time due to the bandwidth limitation. The traffic then overflows the switch buffer capacity and causes packet loss, which introduces an extra delay due to retransmissions. This kind of congestion is named as *incast congestion*, which is a major cause of the delay of data requests in datacenter [2, 9**?**].

The root cause of incast congestion is the many-to-one communication pattern between a front-end server and many data servers. Therefore, many previous methods have been proposed to handle the incast congestion problem by reducing the number of data servers concurrently connected to the front-end server. We classify these methods to three groups. The first group [10–17] improves the sliding window protocol. This approach measures the actual packet throughput variation to decide the size of the sliding window at the front-end server. When the sliding window has an available slot of download link bandwidth, the front-end server sends a query to a data server. However, there is a delay between the new query sending and the response receiving so that the download link bandwidth cannot be fully utilized. Therefore, this approach cannot meet the stringent low service latency requirement of the current Web applications. The second group [18–20] uses data reallocation that tries to reallocate or replicate the data objects of a request to a small number of servers. However, the data reallocation method requires that many requests share concurrently requested data objects (e.g., user data in online social networks) and the data replication method generates a high overhead due to data replication and consistency maintenance [21]. The third group [22–24] pre-determines a certain time interval between any two consecutive responses in order to limit the number of responses during a short time arriving at the front-end server. However, the network status varies over time and between different data servers, so it is difficult to pre-determine the interresponse interval to fully utilize the bandwidth while avoiding congestion. If we improve this approach to dynamically determine the interresponse interval based on current network status of individual data servers, it is not applicable to the current Web applications which needs hundreds or thousands of responses for one data request because of the high overhead for the front-end server to keep track of the network status of such a large number of data servers.

More importantly, all of these previous approaches usually consider the direct data transmissions from data servers to the front-end server for a request, which leads to very fast ($178\mu s$ seconds) transmission of only a few (1 or 2) data objects from each data server [2, 25, 26] for current Web applications. The transient transmission makes it difficult to timely control the sending speed or to adjust the workloads on different data servers without knowing their current transmission speeds to reduce latency. A very large number of data servers for one data request make these tasks even more formidable.

To handle the aforementioned problems, in this paper, we propose a Swarm-based Incast Congestion Control method (*SICC*). It also makes data transmission long-lasting to effectively control data transmission speeds and adjust workloads on different data servers to fully utilize bandwidth to reduce service latency. SICC forms all target data servers of one request in the same rack into a swarm. In each swarm, a data server (called hub) is selected to forward all data objects to the front-end server, so that the number of data servers concurrently connected to the front-end server is reduced. Also, instead of sending out data object queries sequentially, the front-end server sends out data queries simultaneously, so that it can receive the responses continuously without the delay on the data servers for waiting for the data queries. The long-lasting data transmission from hubs to the front-end server allows it to adjust the hubs' transmission speeds and redirect the requests to balance the workloads among them according to their current data transmission speeds. Also, each hub can receive many packets for compression in order to save bandwidth consumption for transmitting many packets.

SICC consists of the following methods.

Proximity-aware swarm based data transmission. The front-end server dynamically clusters all target data servers of a request in the same rack into a swarm. The hub in each swarm is responsible for collecting all data responses from its swarm and sends the responses continually to the front-end server. Hubs can form a multi-level tree to further reduce the number of concurrently connected servers to avoid incast congestion.

Two-level data transmission speed control. Each front-end server adjusts the data transmission speed of each hub based on its network status in order to fully utilize its bandwidth while avoiding congestion. It also adjusts its received data response traffic to its edge switch to avoid congestion at the aggregation router to avoid packet loss.

Packet compression and object query redirection. Each hub combines several data objects together to one packet to reduce the number of packets in transmission to reduce traffic. Also, the front-end server redirects data queries from an overloaded hub to an underloaded hub to reduce the longest data transmission latency in all hubs for a request.

The rest of this paper is organized as follows. Section 2 presents the related work. Section 3 presents the design of *SICC* in detail. Sections 4 and 5 present the performance evaluation of *SICC* in comparison with other methods a model on a simulator and a prototype on a real cluster, respectively. Section 6 concludes this paper with remarks on our future work.

2 RELATED WORK

There are many works focusing on the incast congestion control problem. We classify the previously proposed methods to three groups: improved sliding window protocols, data server reduction and transmission delay control.

Many works improve the sliding window protocol [27] to solve the incast congestion problem by limiting the number of concurrently connected data servers in order to reduce the number of concurrent responses. ICTCP [10] adjusts the receive window according to the ratio of the actual throughput over the expected throughput. When the ratio decreases, the window size is increased to use more available bandwidth and vice verse. It divides the slot into two sub-slots and then uses all the traffic received in the first sub-slot to calculate the available bandwidth as quota for window increase on the second sub-slot. Zhang *et al.* [14] modeled the TCP incast congestion problem based on their observation that the TCP throughput is affected by two kinds of timeouts, which are for waiting all other senders to finish and for the retransmission caused by incast congestion. To improve the downlink bandwidth utilization, DCTCP [12, 13] reduces the window size by a flexible ratio according to current network status, such as the round trip delay (RTD) and package loss rate.

Although all of the above methods can control the incast congestion by adjusting the size of the sliding window, they introduce extra round trip delay when the front-end server waits for response of new requests while the window is moving.

The methods in [18–20] use data reallocation or data replication to increase the number of data objects queried from each server, so that the number of concurrently requested servers is reduced. However, it needs data reallocation to store simultaneously requested data together in the same server. In [18, 20], concurrently requested data objects are replicated to several servers close to each other, so that a request can target a small number of servers nearby. In [19], all servers are divided into several RAID groups, and all concurrently requested data objects are put into one of the groups, so that each data request only communicates with data servers in one RAID group. The above methods can avoid the incast congestion to a certain extent by reducing the number of concurrently connected servers. However, they need data reallocation or replication, which introduce extra network loads.

In [22–24], a short delay is introduced between two consecutive requests by manually scheduling the second response in an extra short delay. Therefore, the concurrent number of connected data servers is reduced to avoid incast congestion. In [22], the authors insert one unit time delay between two consecutive requests. The methods in [23, 24] asks the target server to wait for a certain time before transmitting the requested data, so that the number of concurrently connected data servers is reduced. However, one predetermined delay for all data queries in a data request cannot adapt to the network status varying over time and between different data servers. This delay adaptation is a non-trivial task for current Web applications because they have thousands of data objects for a request and it is difficult to profile the network status for the thousands of data servers and decide the delay individually.

Previous methods use the direct connection between data servers to the front-end server. The transient data transmission from each data server to the front-end server for a request makes it difficult to control the sending speed or adjust workloads between data servers. To overcome this problem, we solve the incast congestion problem using a completely different approach. We set one of the local servers as a hub to collect all the requested data objects nearby, so that the number of concurrently connected data servers of the front-end server is reduced to avoid the incast congestion. The long-lasting data transmission between hubs and the front-end server enables data response speed control, package compression, and query redirection among data replicas to further reduce the request latency.

3 SWARM BASED INCAST CONGESTION CONTROL

In this section, we present the details of Swarm based Incast Congestion Control (*SICC*). A swarm is formed by all data servers of a request in the same rack. In *SICC*, the front-end server dynamically forms a proximity-aware swarm structure with all data servers for a request, and selects one data server from each swarm as the hub to connect to it in order to reduce the number of concurrently connected data servers to avoid the incast congestion. By monitoring the actual packet transmission speed of each hub and the traffic in the uplink of edge switch, each front-end server controls the data transmission speed of hub servers to fully utilize its bandwidth without causing congestion in its edge switch and aggregation router. *SICC* has two enhancements, a packet compression method and object query redirection method, to further reduce the network overhead and data request latency.

3.1 Proximity-aware Swarm based Data Transmission

To avoid incast congestion, *SICC* also reduces the number of concurrently connected data servers to the front-end server. For this purpose, rather than relying on sliding window protocol (that causes an extra delay) or data reallocation (that generates extra overhead), *SICC* introduces another layer between the requester (front-end server) and the responsers (data servers) of a request (Figure 2), which consists of several data servers called hubs. Hubs are responsible for data transmission between the front-end server and the data servers. We use h_i to denote the hub of the i^{th} swarm, and H to denote the set of all hubs.

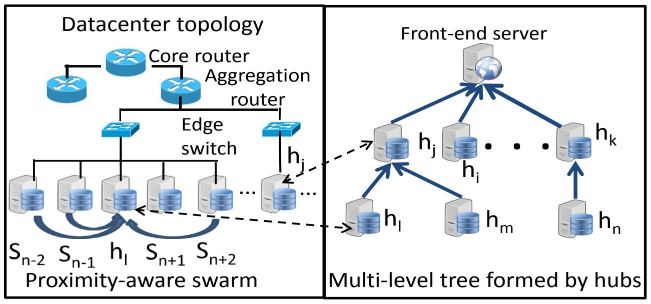


Figure 2: The multilevel tree with proximity-aware swarms.

For a data request, *SICC* forms the target data servers to swarms with each swarm consisting of data servers in the same rack. The server with the largest spare capacity to handle I/O among each rack is selected as the hub of each rack. In order to maintain the multi-level structure, we also select another server in the rack with larger spare capacity as the backup for the hub. When the current hub fails, the backup server will serve as the hub. A hub forwards data object queries from the front-end server to the target data servers, and then forwards the data responses from the data servers to the front-end server. Each hub continuously sends all queried data objects to the front-end server, starting from the data objects stored inside it and then the received data objects from other data servers inside its swarm sequentially. Since the number of data servers in a swarm is limited and also the number of hub servers, this one-to-many communication pattern is unlikely to cause incast congestion. Also, because data servers and the hub are in the same rack, the data transmission efficiency will be enough.

Note this structure is dynamically created for each request rather than fixed and it does not need to be maintained. The front-end server sends its data object queries to each hub along with its swarm information. Then, each hub knows the data servers to forward the queries. After receiving queries from a hub, the data servers know the hub to send their data responses. Finally, the hub forwards the data responses to the front-end server. If the hub layer has too many hubs that will generate incast congestion, we transform the hub layer to a tree structure. We will explain the tree creation later on.

We create the transient swarm structure from the data servers of a request to be used specifically for the request rather than creating a global tree from all data servers in the datacenter for all requests because of three reasons. First, the transient structure does not need to be maintained by periodical probing between connected servers, which avoids generating more network load. Second, transmitting data through a much smaller structure greatly reduces the latency. Third, the data servers without the requested data objects do not need to involve in the data transmission for the request, which saves data transmission time since the establishment of the data transmission connection takes a certain time. Though the front-end server needs to create a swarm structure for each request, this computing latency is negligible as is shown in Section 4.6.

Next, we discuss how to determine a suitable number of hubs. If there are too many hubs in the system, the number of concurrently received packets in a short time can still cause incast congestion. Generally, Assume the bandwidth of downlink is B_d Gbps, the bandwidth of uplink is B_u Gbps, the average size of a packet as $\overline{s}$, and the buffer size of the edge switch is S_e MB. In the case that all hubs' packets arrive at the edge switch of the front-end server in a short time, the largest number of hubs (denoted by M) connecting front-end server at a time without causing the increment of the queue size in the edge switch is

$$M = \frac{\frac{S_e}{B_d} * B_u}{\overline{s}}. \tag{1}$$

Assume that there are m requests sent from the front-end server at a time on average, then the number of hubs for one request is $N = M/m$.

In a large-scale datacenter with a lot of racks, we also need to constrain the number of hubs directly connecting to the front-end server to be less than N. To achieve this, as shown in Figure 2, all hubs need to form a multi-level tree structure with the front-end server as the root. Each child hub transmits all its requested data objects to its parent hub continuously, which transmits the data further to its parent. In order to reduce the network load, we try to reduce the transmission switches and data size in data transmission. Then, we follow two rules when building the tree.

Rule 1: We form the tree with proximity-awareness to reduce the number of transmission switches. That is, two hubs (including their children) under the same aggregation router are linked together in the tree.

Rule 2: We ensure that a hub's child always has a smaller number of requested data objects (including the data objects inside its child hub) than its parent in the tree structure.

Algorithm 1 shows the procedure to build the multi-level tree from the target data servers of a request. Based on Rule 1, *SICC* clusters target data servers inside the same rack into a proximity-aware swarm (Line 1). Inside a swarm, based on Rule 2, the data server storing the largest number of queried data objects is selected to be the hub by the front-end server, and the hub enqueues into queue Q_h (Lines 2-4). Thus, the hub can communicate with its proximity close data servers directly through the edge switch with minimized path length as 1. Due to the clustering of data servers, the number of hubs is much smaller than the number of target data servers, so that the total number of concurrently connected data servers to the front-end server is reduced to avoid the incast congestion.

When the number of hubs connecting the front-end servers is larger than N, which tends to generate incast congestion, a multi-level tree is formed from the hubs to limit the number of concurrent connections to the front-end server no larger than N. By following Rule 1, we first sort all hubs in an ascending order of the number of their stored requested data objects Q_h (Line 5). Q_h contains hubs in an ascending order of the number of requested data objects contained in the subtree with the root of each hub. While the number of hubs connecting the front-end servers is larger than N (Line 7), starting from the first hub h_i (Line 8), we try to form a subtree to connect it (as the child) and a hub nearby (as the parent) (Lines 9-13). According to Rule 1 and Rule 2, we try to link a hub to the hub with the smallest number of data objects among the hubs under the same aggregate router (Lines 9-10). Also, in order to balance the workloads among hubs and reduce the number of levels of the tree to reduce the network load of data transmission, if h_i is a parent hub of other hubs, it transfers each of its child from the one with the largest number of requested data objects to be a child of h_j until all h_i's children are transferred or h_j has N children (Lines 10-11). After that, the number of requested data objects reported by h_j is updated by adding the number of data objects reported by h_i (Line 12), and h_j's position in the queue should be updated accordingly based on the number of requested data objects contained in its subtree (Line 13).

Algorithm 1: Building a multi-level tree from hubs.

```
1  Cluster target data servers in each rack into a swarm;
2  /*Selecting a hub from each swarm*/
3  for each swarm do
4      Select the data server with the largest number of requested
         data objects as the hub; Enqueue the hub in to queue $Q_h$;
5  Sort the hubs in $Q_h$ in an ascending order of the number of
     stored requested data objects;
6  /*Creating multi-level tree from the hubs*/
7  while $|Q_h| > N$ do
8      Dequeue a hub $h_i$ from $Q_h$;
9      Select a
         hub $h_j$ with the smallest number of data objects and under
         the same aggregation router as $h_i$; Link $h_i$ as a child to $h_j$;
10     while $h_j$ has less than N children and $h_i$ has children do
11         Transmit the last child from $h_i$ to be a child of $h_j$;
12     Update $h_j$'s number of requested data objects by add $h_i$'s;
13     Update $h_j$'s position at $Q_h$ accordingly;
```

3.2 Two-Level Data Transmission Speed Control

3.2.1 Congestion Avoidance at the Front-End Server. For a data request, the front-end server sends out the data queries concurrently to all hubs in the swarm structure. While collecting all data responses from target data servers inside the same swarm, a hub continuously sends out all data responses to the front-end server. The sum of the bandwidths of the hubs' upload links may still be larger than the bandwidth of the download link of the front-end server, which may cause incast congestion.

To avoid the incast congestion caused by the hubs, we control the data transmission speed of each hub in each short time period (denoted by t_i ($i \in N^+$)) in order to fully utilize the bandwidth of the front-end server while avoiding overflows. Fortunately, the long-lasting data transmissions from hubs to the front-end servers enable to learn the transmission speeds of hubs in time t_{i-1} to adjust their assigned bandwidth in time t_i. We use $b^a_{h_i}$ to denote the assigned data transmission speed of hub h_i, and use $b^r_{h_i}$ to denote the real data transmission speed from hub h_i to the front-end server measured during the last short time period. We use B_p to denote the downlink bandwidth that the front-end server plans or is assigned to use for the next time period. At initial, B_p is set to $(B_d - B_a)$, where B_d denotes its bandwidth capacity, and B_a denotes its actual received total size of packets during time t_{i-1}. We will explain how to update B_p later on.

At the initial time of each short time period, without considering the different network status of each hub, the front-end server can allocate its bandwidth evenly to each hub as:

$$b^a_{h_i} = \frac{B_p}{|H|}. \tag{2}$$

However, the network status of each hub varies over time and the loads on different hubs are different, so some hubs may not fully utilize their assigned transmission speed $b^a_{h_i}$ while others need a transmission speed higher than $b^a_{h_i}$. In order to fully utilize the bandwidth of the front-end server without causing congestion, we reassign the over-assigned bandwidth to other hubs that need more bandwidth. We use H^o and H^u to denote the set of hubs with $b^a_{h_i} < b^r_{h_i}$ (over-utilized hubs) and the set of hubs with $b^a_{h_i} > b^r_{h_i}$ (under-utilized hubs), respectively. Therefore, we reassign the data transmission speed of hubs in H^o to:

$$b^a_{h_i} = b^a_{h_i} + \frac{\sum_{h_j \in H^u}(b^a_{h_j} - b^r_{h_j})}{|H^o|}, \tag{3}$$

and the data transmission speed of hubs in H^u to:

$$b^a_{h_i} = b^r_{h_i}. \tag{4}$$

The front-end server periodically adjusts the assigned bandwidth to each hub after each short time period. Since the expected and upper bound of data transmission speed is always $\sum_{h_i \in H} b^a_{h_i}$ which equals B_p, the front-end server overflow of the download link is avoided and the bandwidth is fully utilized.

3.2.2 Congestion Avoidance at the Aggregation Router. Considering a number of front-end servers in the same rack, due to their sharing of the uplink of the edge-switch (Figure 2), the incast congestion may occur at the uplink of the edge-switch (i.e., downlink of the aggregation router) if all front-end servers in the same rack receive many data responses at the same time. To avoid the overflow at the uplink of edge-switches, the front-end servers need to cooperatively adjust the data transmission speeds for data responses in their downlinks on the edge-switch. That is, B_p used in Equation (2) for the next time period is proactively adjusted.

At the beginning of each period t_i, each front-end server asks the total size of all queueing packets (denoted by q_{t_i}) in the aggregation router's port, which is connected to the uplink of its edge switch. We use S_a to denote the size of the buffer in the aggregation router for package queueing and use T to denote a threshold to judge a possible incoming congestion at the uplink of the edge switch. If $\frac{q_{t_i}}{S_a} \geq T$, each front-end server cuts down its B_p to avoid the congestion:

$$B_p = B_p * (1 - \beta * \frac{q_{t_i}}{S_a}), \tag{5}$$

where β is the upper bound of the decrement of the bandwidth. We used a sliding window [27] like congestion control strategy by reducing the planned bandwidth by a certain percentage. Largely reducing the planned bandwidth leads to low bandwidth utilization. Therefore, *SICC* adjusts the planned bandwidth according to the congestion conditions measured by $\frac{q_{t_i}}{S_a}$. A larger q_{t_i} compared to the buffer size S_a indicates a more serious congestion in the edge bandwidth uplink, which needs a larger decrement on B_p. After updating B_p, all the data transmission speeds of hubs are updated by keeping the same portion of their sharing of the B_p in last period based on Equation (2).

To fully utilize the bandwidth of the uplink, we need to enlarge B_p when there is no predicted congestion. Then, we set

$$B_p = \min\{B_d, B_p * (1 + \alpha * \frac{B_a}{B_p})\}, \tag{6}$$

where α is the upper bound of the increment of the bandwidth. Instead of using a slow increase as in the sliding window protocol, *SICC* increases the planned bandwidth by a certain percentage according to the network status. A larger B_a means that the hub fully utilized its planed bandwidth in current period, which indicates a better network status during packet routing. Thus, we increased B_p faster with a larger $\frac{B_a}{B_p}$. On the other hand, a smaller B_a indicates a busy network during data transmission. Therefore, B_p is increased more slowly with a smaller $\frac{B_a}{B_p}$.

3.3 Packet Compression and Object Query Redirection

3.3.1 Packet Compression. The data object is usually very small and no larger than 1KB [2], such as the text content of one friend post and status in online social networks. To put each data object in one packet, a large amount of the bandwidth along the path from a hub to the front-end server is consumed by transferring the packet headers compared to its small payload. Thus, the network resource utilization is reduced.

Actually, the maximum payload of a packet can be much larger than the size of a data object. For example, the packet in Ethernet is 1, 500 bytes. Therefore, a hub can combine several data objects into the same packet until the maximum allowed payload is reached. It reduces the total number of packets needed to be sent to the front-end server through inter-rack communication and saves the bandwidth otherwise needed to transmit a large number of packet heads. As a result, the network resource utilization is increased. The packet compression is more effective for a data request with many requested data objects. This is because more requested data objects lead to more queries inside the same rack, which enables a large packet to be more likely to find small packets in the same rack to be transmitted together in order to reduce the number of transmitted packets.

3.3.2 Query Redirection. The data request response latency depends on the hub that is the last one finishing the data transmission to the front-end server regardless of the transmission speeds of the other hubs. Therefore, an incast congestion control method needs to

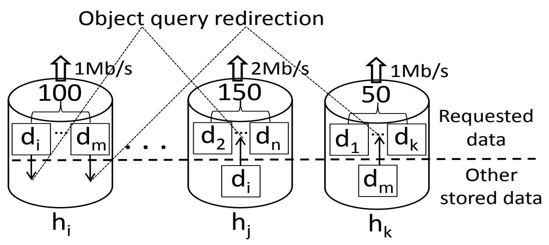


Figure 3: An example of query redirection.

reduce the longest data transmission latency in all hubs. To achieve this, *SICC* needs to balance the number of data objects transmitted from different hubs according to their data transmission speeds to minimize the data response latency. We define the data transmission progress rate of hub h_i (denoted by p_{h_i}) as:

$$p_{h_i} = \frac{|D_{h_i}| * \bar{s}}{b^r_{h_i}}, \quad (7)$$

where D_{h_i} denotes the set of all data objects stored in the data servers in the subtree of h_i that have not been transmitted yet. The data transmission progress rate p_{h_i} actually denotes the expected remaining time to finish the data transmission. In order to reduce the longest data request latency among all hubs, we need to balance the data transmission progress rate among them.

For each data object, there are usually several data replicas stored by different data servers over the datacenter in order to achieve high data availability [28, 29]. Therefore, if a hub has a long data remaining transmission time, the front-end server can redirect some of the hub's queries to another hub whose subtree has data servers hosting replicas of the data objects of the queries. As shown in Figure 3, h_i has a higher remaining time than h_j and h_k. The request of d_i and d_j are redirected from h_i to h_j and h_k, respectively. To do this, we define the average transmission remaining time as

$$\bar{p} = \frac{\sum_{h_i \in H} p_{h_i}}{|H|}. \quad (8)$$

For any hub with $p_{h_i} > \bar{p}$, we define it as a low-progress hub, and use D_l to denote the set of all low-progress hubs; for any hub with $p_{h_i} < \bar{p}$, we define it as a high-progress hub, and use D_h to denote the set of all high-progress hubs.

We aim to redirect some queries from each hub h_i in D_l to the hubs in D_h to make h_i a non-less-progress hub. We loop all data objects $d_k \in D_{h_i}$ until h_i is not a low-progress hub. Specifically, for each data object $d_k \in D_{h_i}$, if there exists a data replica inside the swarm of a high-progress hub h_j in D_h, we redirect the data query of d_k from h_i to h_j. We then update D_{h_i} and D_{h_j}, and recalculate p_{h_i} and p_{h_j} accordingly. By comparing with $\bar{p}$, if the hub h_i (h_j) is no longer a low (high) progress hub, it is removed from D_l (D_h). In this way, all hubs for a request are expected to have a similar data transmission progress rate, and the longest data request latency among all the hubs is reduced.

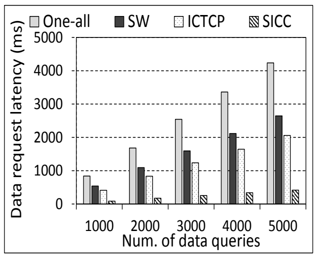
4 PERFORMANCE EVALUATION

We simulated 3000 data servers [30] in a datacenter, which forms a typical three-layer fat-tree [5] with 60 data servers inside a rack [31]. Front-end servers were randomly selected from servers. The capacity of downlink, uplink and buffer size of each edge-switch were set to 1Gbps, 1Gbps [32] and 100KB, respectively. We assume a 1:4 over-subscription ratio at the ToR tier.We set the default number of requested data objects of a data request to 1000 [2]. Each data object has three replicas [29] randomly distributed among all data servers [33]. We set the size of each packet to a value randomly chosen from [20, 1000]B [2]. The timeout of TCP packet retransmission was set to 10 ms [34]. As [10, 35, 36], we first simulated the incast congestion scenario with one front-end server requesting data objects from multiple data servers. For each experiment, the front-end server continuously initiates 10,000 data requests one after each other, and we measure the average performance per request after the front-end server receives all queried data objects. Later on, we test the scenario of multiple front-end servers. We assume that there is no any physical failure in the simulation.

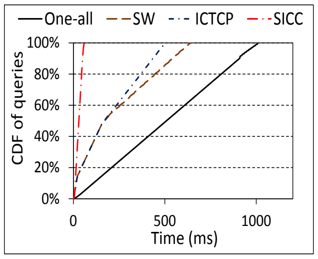
We compared *SICC* with previous incast congestion control methods: *One-all*, the sliding window protocol (*SW*) [2], and *ICTCP* [10].
One-all We use *One-all* as a baseline. In this method, the front-end server simultaneously sends out queries to all target data servers, which start the data transmission to the front-end server right after receiving the queries.
SW The sliding window protocol (SW) [2] reduces the concurrently connected data servers to the front-end server using the typical sliding window protocol, which increases the window size till the occurrence of incast congestion and then decreases the size.
ICTCP [10] improves the sliding window protocol by adjusting the receiving window according to the ratio of the actual throughput over the expected throughput. It divides the slot into two sub-slots and then uses all the traffic received in the first sub-slot to calculate the available bandwidth as quota for window increase on the second sub-slot. In the following sections, we first measure the performance of *SICC* without enhancements, and then measure the effectiveness of each enhancement method.

4.1 Performance of Data Request Latency

A data request consists of many data queries for different data objects. The latency of a query is defined as the time elapsed from the time when the front-end server initiates the query to the time when it receives the data object. The longest query latency among the queries of a request is the request's latency. Figure 4(a) shows the data request latency of different methods versus the number of data queries. Figure 4(b) shows the CDF of data queries over time of one data request. From both figures, we see that the data request latency follows *SICC<ICTCP<SW<One-all*. In *One-all*, all target data servers send data packets to the front-end server during a short time, which causes incast congestion and retransmissions for dropped packets, thus leading to the highest latency. *SW* reduces the concurrently connected data servers through the sliding window protocol. Thus, it generates a shorter data request latency than *One-all* due to lighter incast congestion. *SW* generates a longer service latency than *ICTCP*, which improves the sliding window protocol to avoid increasing the window size beyond the bandwidth of the uplink. However, the sliding window cannot fully utilize the bandwidth while moving the window forwards, and a delay is generated



(a) Data request latency



(b) CDF of queries

Figure 4: Performance of response latency.

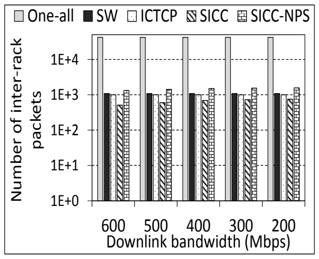


Figure 5: Inter-rack traffic cost reduction

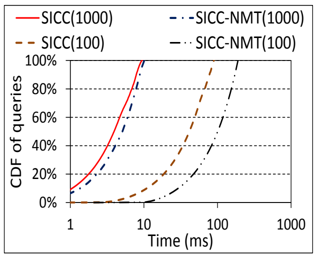


Figure 6: Effectiveness of swarm-based multi-level tree.

between querying sending and response receiving for a new available slot. *SICC* generates a shorter latency than *ICTCP* since *SICC* receives all data responses continuously by fully utilizing the bandwidth of the downlink. Figure 4(a) also shows that the data request latency of all methods increases proportional to the number of data queries of a request. More queries mean that more data objects need to be transmitted to each hub, leading to a longer data transmission time. The figures indicate that *SICC* generates the shortest data request latency among all methods by avoiding congestion and fully utilizing the downlink bandwidth.

4.2 Performance in Reducing Inter-Rack Traffic

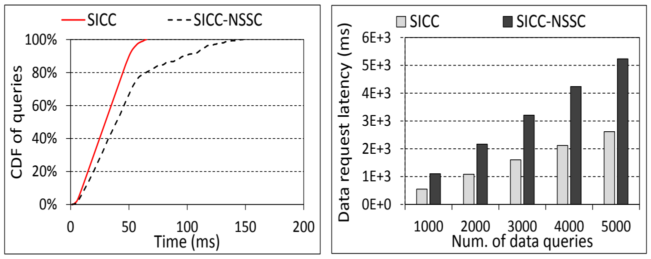
Inter-rack communication usually has a higher latency than intra-rack communication. Also, the network resources of inter-rack communication are highly required since the resources are shared by many servers under different racks. The bandwidth of links of an aggregation router is much smaller than the total downlink bandwidth of all data servers connecting to this router. Therefore, it is necessary to reduce the number of inter-rack packets. Figure 5 shows the number of inter-rack packets (including retransmitted packets) on a logarithmic scale generated by different methods while the downlink bandwidth decreases from 600Mbps to 200Mbps. We use *SICC-NPS* to denote *SICC* without the Proximity-aware Swarm method (PS), in which each hub randomly selects the same amount of target data servers as in *SICC-NPS* among all data servers as its swarm children. From the figure, we see that the result follows *One-all>SICC-NPS>SW>ICTCP>SICC*. *One-all* generates the largest number of inter-rack packets since the packet retransmissions caused by the incast congestion generate extra inter-rack packets. *SICC-NPS* can mitigate incast congestion so that it generates a smaller number of inter-rack packets than *One-all*. *SICC-NPS* generates a larger number of inter-rack packets than *SW*. This is because in *SICC-NPS*, most packets between hubs and data servers in their swarms are transmitted between racks due to the proximity-unaware clustering. In *SW*, all data servers transmit the packets directly to the front-end server without another forwarding layer between hubs and the front-end server as in *SICC-NPS*. *ICTCP* also has direct transmission without an additional forwarding layer. Since *ICTCP* avoids more incast congestion and hence reduces more packet retransmissions than *SW*, it generates a smaller number of inter-rack packets than *SW*. *SICC* generates the smallest number of inter-rack packets due to its proximity-aware swarm creation, packet compression method to send several data packets together, and the incast congestion control that avoids packet retransmission. This figure indicates that *SICC* is the most effective in reducing the number of inter-rack packets to reduce request latency and save the inter-rack network resources.

4.3 Performance of Swarm based Multi-Level Tree

We then measure the effectiveness of the swarm based multi-level tree to reduce the data request latency by avoiding the incast congestion. We use *SICC-NMT* to denote *SICC* without the Multi-level Tree (MT), so that all hubs directly connect to the front-end server. Figure 6 shows the CDF of the queries over time of different methods versus downlink bandwidth capacity and (x) in the figure means that the downlink is xMbps. It shows that *SICC-NMT* generates a longer data request latency than *SICC* due to the incast congestion caused by packets concurrently sent from all hubs. The figure also shows that a larger downlink bandwidth leads to a smaller response latency. By fully utilizing the bandwidth, *SICC(1000)* generates approximate one-tenth of the data request latency of *SICC(100)* even though it has a higher depth of multi-level tree. Since each hub starts transmitting data objects continuously from currently stored and received requested data objects, it does not need to wait for receiving all data objects from its children. Therefore, by sending and receiving data objects continuously, the hub can fully utilize its assigned bandwidth. Therefore, a tree with a large depth does not increase the data request latency. The figure further shows that with a smaller downlink bandwidth, *SICC-NMT* generates much longer latency than *SICC*. This is because, with a smaller downlink bandwidth, there should be fewer hubs directly connect to the front-end server. Therefore, *SICC-NMT* generates more serious incast congestion because it has more hubs connecting to the front-end server. In summary, the figure indicates that the multi-level tree can avoid incast congestion caused by many hubs directly connecting the front-end server, and its depth hardly affects the data request latency.

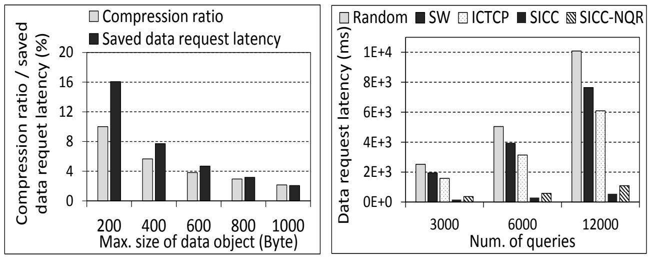
4.4 Two-level Data Transmission Speed Control

In this section, we measure the performance of our two-level data transmission Speed Control method (SSC). We use *SICC-NSSC* to denote *SICC* without this method. We adjust the assigned downlink bandwidth to each hub in every 10ms. We first present the performance of congestion control at the front-end server side and then at the aggregation router. For each experiment, we set the probability of each hub becoming overloaded to 50%, and the overloaded hub has an actual data transmission speed as 10% of its initially assigned



(a) CDF of queries of a single front-end server (b) Data request latency with multi-front-end servers

Figure 7: Effectiveness of two-level speed control.



(a) Packet compression. (b) Query redirection

Figure 8: Effectiveness of the enhancement methods

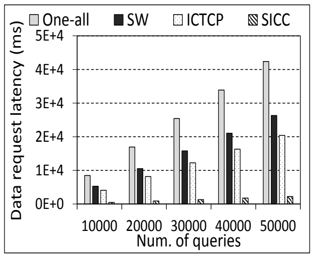


Figure 9: Performance of scalability.

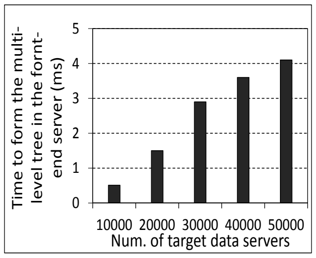


Figure 10: Computing time for tree creation.

data transmission speed. Figure 7(a) shows the CDF of queries over time of *SICC* and *SICC-NSSC*. It shows that *SICC* has a much smaller data request latency than *SICC-NSSC*. This is because *SICC* reassigns the data transmission speeds of hubs according to their actual data transmission speeds. Together with the query redirection method, *SICC* can fully utilize the bandwidth of downlink to reduce the query latency. *SICC-NSSC* also leverages query redirection to balance the progress, but without speed control, it cannot fully utilize the bandwidth, leading to a longer data request latency. The figure indicates that the data transmission speed control can effectively reduce the data query latency when the hubs are overloaded by fully utilizing the bandwidth of the edge switch downlink.

We then present the performance of congestion avoidance at the aggregation router. We set all data servers inside a rack as front-end servers, each of which conducts a request concurrently. We set $\alpha = \beta = 20\%$, $T = 10\%$ and $B_a = 200KB$. Figure 7(b) shows the average data request latency of *SICC* and *SICC-NSSC* versus the number of queries per request. It shows that *SICC-NSSC* generates a much longer data request latency than *SICC*. This is because, without the speed control method, all front-end servers aim to receive the packets at the speed of their downlink bandwidth. It causes incast congestion at the aggregation router. Then, a timeout delay is introduced to all front-end servers due to packet loss. The figure indicates that the speed control can effectively reduce the data request latency by avoiding the incast congestion at the aggregation router side.

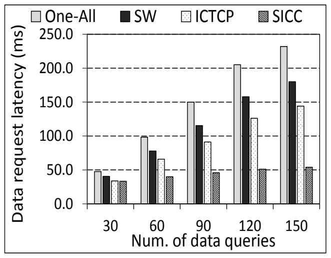
4.5 Performance of Enhancement Methods

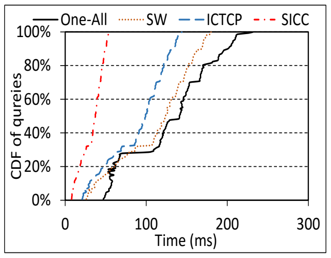
We first measure the effectiveness of the packet compression method in reducing the number of inter-rack packets and data request latency. In order not to count the inter-rack packets between hubs in the multi-level tree to show packet compression's sole effectiveness in reducing the number of inter-rack packets, we connected all hubs directly to the front-end server. We measure the compression ratio by n/n', where n and n' represent the number of inter-rack packets generated by *SICC* without and with packet compression, respectively. Recall that the size of a data object was randomly chosen from [20, 1000]B. In this test, the size of a data object was randomly chosen from [20, x]B, where the maximum size of a data object x was varied from 200B to 1000B with a step size as 200B. Figure 8(a) shows the compression ratio, which is always much larger than 1. It implies that the packet compression effectively reduces the number of packets transmitted from hubs. We also see that the compression ratio decreases as the size of the data objects increases. This is because a large maximum size of a data object leads to a lower probability to fit two packets into the same Ethernet packet with the maximum payload limitation. Besides, the figure shows the saved data request latency calculated by $(l' - l)/l'$, where l and l' are the data request latency of *SICC* with and without packet compression, respectively. It shows that the packet compression can reduce the data request latency. This is because a larger payload in packets leads to higher bandwidth utilization and then a shorter data request latency while transmitting the same amount of data. It indicates that the packet compression method is effective in reducing the data request latency of *SICC*.

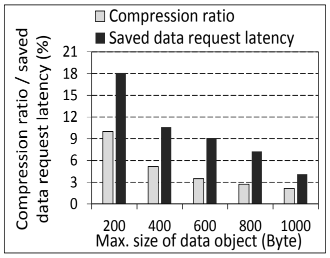
We then measure the effectiveness of the query redirection in reducing the data request latency. We use the same scenario as in Section 4.4. We use *SICC-NQR* to denote *SICC* without the Query Redirection method (QR). Figure 8(b) shows the data request latency of different methods with different number of queries. It shows the same order among all methods as shown in Figure 4(a) due to the same reasons. *SICC-NQR* generates a longer data request latency than *SICC* because of the longer latency to transmit requested data from overloaded hubs while *SICC* can redirect the requests to balance the data transmission progress rate. The figure indicates that the query redirection method effectively reduces data request latency by balancing the data transmission progress ratees among hubs.

4.6 Performance of Scalability

In this section, we measure the data request latency of different methods in a large-scale datacenter. We enlarge the number of data servers by 50 times. We varied the number of queries of a request from 10,000 to 50,000 with a step size as 10,000 to measure the performance. Figure 9 shows the data request latency of all different


(a) Data request latency


(b) CDF of queries


(c) Effectiveness of packet compression

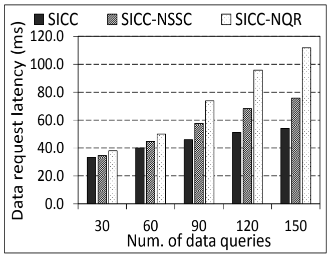

(d) Effectiveness of speed control and query redirection

Figure 11: Performance on a real cluster.

methods. We see that *SICC* always generates the shortest data request latency among all methods. Also, as the number of queries increases, its data request latency slowly increases proportionally while those of other methods increase rapidly. This is because *SICC* effectively controls all hubs data transmission progress rate and speed and the number of hubs connecting to it to avoid the incast congestion and fully utilize the bandwidth. The figure also shows the same order among all other methods as shown in Figure 4(a) due to the same reasons. The figure indicates that *SICC* generates the shortest request latency, and its performance is more scalable than other methods in a large-scale datacenter.

We also measure the time to create the multi-level tree with proximity-aware swarms in a front-end server. We measured the computing time in a laptop with 4GB memory and Dual-core 2.5GHz CPU. The computing time in a powerful front-end server in practice will be much smaller. Figure 10 shows the computing time to create the multi-level tree versus the number of target data servers. We set the number of requested data objects in each target data server to a value randomly chosen from [1, .., 10]. It shows that more target data servers lead to a higher computing time. This is because more data servers from more swarms, and there are more hubs to form the multi-level tree, increasing the computing workload. However, the computing time is around 4ms to computing a multi-level tree with 50,000 data servers, and less than 1ms for 10,000 data servers. Therefore, the latency to form the tree introduces a small delay, which is much smaller than 100ms as the typical budget for a data request in a datacenter serving Web applications [4].

5 EVALUATION ON A REAL TESTBED

We implemented *SICC* and other comparison methods on the Palmetto cluster [37], a high-performance supercomputer in Clemson University. The servers are with 2.4G Intel Xeon CPUs E5-2665 (16 cores), 64GB RAM, 240GB hard disk and 10G NICs. The OS of each server is Linux 64-bit version. The switchs are Brocade MLX-eâĂŘ32 switchsWe which can provide 40Gbps. The CPU, Memory and hard disk are never a bottleneck in any of our experiments. We randomly selected 150 servers from all servers and one front-end server in them, each of which has the downlink and uplink as 10Gbps. We randomly distributed 150 data objects into the data servers, the size and the number of replicas of each data object follow the same distribution as in Section 4. We use a batch-processing application, the Apache Hadoop mapreduce framework [33] to simulate the web application workload. The workload consists of WordCount (counting unique words in text) and PageRank (Implementation of PageRank algorithm) [38]. All other settings are the same as in Section 4.

Figure 11(a) shows the data request latency of all methods versus the number of queries. Figure 11(b) shows the CDF of queries over time of all incast control methods. They exhibit the same order and trends as in Figure 4(a) and Figure 4(b) due to the same reasons. The figure indicates that *SICC* generates the shortest data request latency of all methods on the testbed.

Figure 11(c) shows the compression ratio and the saved data request latency of the packet compression method. It shows the same trends as in Figure 8(a) due to the same reasons. It indicates that the packet compression method is effective in reducing data request latency by reducing the number of packets.

Figure 11(d) shows the data request latency of *SICC* without data transmission speed control or query redirection. It shows that *SICC-NSSC* and *SICC-NQR* generate longer data request latency than *SICC* due to the same reasons as in Figures 7(a) and 8(b), respectively. The figure shows that the data transmission speed control and query redirection reduce the data request latency of *SICC* without each of them by 40.3% and 107%, respectively when there are 1500 requested data objects. The figure indicates that the two methods can effectively improve the request latency of *SICC*.

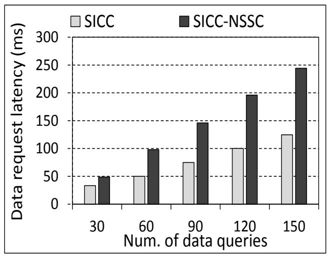


Figure 12: Data request latency with multi-front-end servers.

We then measure the performance of data transmission speed control method in avoiding the incast congestion at the aggregation router side. Due to the small scale and lack of control of the aggregation router in the real computing cluster, we use one data server to function as an aggregation router with 100Mbps downlink capacity.

We randomly selected 20 servers as front-end servers. Other settings are the same as in Figure 7(b).

Figure 12 shows the average data request latency per front-end server of *SICC* with or without the speed control method versus the number of data queries. It shows that the *SICC* generates a much lower data request latency than *SICC-NSSC* as shown in Figure 7(b) due to the same reasons. It indicates that the data transmission speed control can reduce the data request latency by avoiding incast congestion at the aggregation router.

6 CONCLUSION

Previous incast congestion control methods are not applicable to datacenter serving current Web applications because of their stringent low delay requirements and typical data access features (i.e., a very large number of responses and very fast transmission for each response). To solve this problem, we proposed a Swarm-based Incast Congestion Control method (*SICC*). *SICC* clusters the proximity-close data servers in the same rack into swarms, selects a data server as a hub to collect all transmitted data inside its swarm and continuously forwards it to the front-end server, so that the number of concurrently connected data servers to the front-end server is reduced, which avoids the incast congestion. Also, the long-lasting transmission by transmitting data together from a hub enables *SICC* to sophisticatedly control the data transmission speed to avoid congestion while fully utilizing the bandwidth. This feature also enables *SICC* to have two enhancement methods: packet compression and query redirection. The packet compression method combines different packets to one packet to increase the payload of a packet to improve the bandwidth utilization. The query redirection method transmits the data queries from swarms with long remaining data transmission latency to swarms with short remaining data transmission latency in order to reduce the data request latency. The experiments in simulation and on a real cluster show that *SICC* achieves the shortest data request latency compared with other incast control methods.

In the future, we will further consider how to make data transmission bypass congestion during the routing in order to further reduce the data request latency.

7 ACKNOWLEDGEMENTS

This research was supported in part by U.S. NSF grants ACI-1719397, CNS-1733596 and Microsoft Research Faculty Fellowship 8300751.

REFERENCES

[1] Facebook Passes Google In Time Spent On Site For First Time Ever. http://www.businessinsider.com/chart-of-the-day-time-facebook-google-yahoo-2010-9, [accessed in July 2015].

[2] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proc. of NSDI*, 2013.

[3] R. Kohavl and R. Longbotham. Online Experiments: Lessons Learned, 2007. http://exp-platform.com/Documents/IEEEComputer2007Online Experiments.pdf, [accessed in July 2015].

[4] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannona, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!s Hosted Data Serving Platform. In *Proc. of VLDB*, 2008.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. of SIGCOMM*, 2008.

[6] L. Yan, K. Chen, H. Shen, and G. Liu. Mobilecopy: Resisting correlated node failures to enhance data availability in dtns. In *Proc. of SECON*, 2015.

[7] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. Timely: Rtt-based congestion control for the datacenter. In *Proc. of SIGCOMM*, 2015.

[8] Z. Li, H. Shen, J. Denton, and W. Ligon. Comparing application performance on hpc-based hadoop platforms with local storage and dedicated storage. In *Proc. of Big Data*, 2016.

[9] G. Liu, H. Shen, and H. Wang. Computing load aware and long-view load balancing for cluster storage systems. In *Proc. of Big Data*, 2015.

[10] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast Congestion Control for TCP in Data-Center Networks. *TON*, 2013.

[11] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *Proc. of NSDI*, 2011.

[12] M. Alizadeh, A. G. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proc. of SIGCOMM*, 2010.

[13] B. Vamanan, J. Hasan, and T. N. Vijaykumar. Deadline-Aware Datacenter TCP (D2TCP). In *Proc. of SIGCOMM*, 2012.

[14] J. Zhang, F. Ren, and C. Lin. Modeling and Understanding TCP Incast in Data Center Networks. In *Proc. of INFOCOM*, 2011.

[15] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. *ACM SIGCOMM Comput. Commun. Rev.*, 2011.

[16] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. SPAIN: COTS Data-Center Ethernet for Multipathing over Arbitrary Topologies. In *Proc. of NSDI*, 2010.

[17] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G Andersen, G. R Ganger, G. A Gibson, and B. Mueller. Safe and effective fine-grained tcp retransmissions for datacenter communication. In *Proc. of SIGCOM*, 2009.

[18] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A Gibson, and S. Seshan. Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems. In *Proc. of FAST*, 2008.

[19] E. Krevat, V. Vasudevan, A. Phanishayee, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. On Application-Level Approaches to Avoiding TCP Throughput Collapse in Cluster-based Storage Systems. In *Proc. of PDSW*, 2007.

[20] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The Little Engine(s) that Could: Scaling Online Social Networks. In *Proc. of SIGCOMM*, 2010.

[21] G. Liu, H. Shen, and H. Chandler. Selective Data replication for Online Social Networks with Distributed Datacenters. In *Proc. of ICNP*, 2013.

[22] Y. Yang, H. Abe, K. Baba, and S. Shimojo. A Scalable Approach to Avoid Incast Problem from Application Layer. In *Proc. of COMPSACW*, 2013.

[23] M. Podlesny and C. Williamson. An Application-Level Solution for the TCP-Incast Problem in Data Center Networks. In *Proc. of IWQoS*, 2011.

[24] M. Podlesny and C. Williamson. Solving the TCP-Incast Problem with Application-Level Scheduling. In *Proc. of MASCOTS*, 2012.

[25] H. Shen, A. Sarker, L. Yu, and F. Deng. Probabilistic network-aware task placement for mapreduce scheduling. In *Proc. of Cluster*, 2016.

[26] G. Liu, H. Shen, and H. Wang. Deadline guaranteed service for multi-tenant cloud storage. *Trans. on Parallel and Distributed Systems, TPDS*, 2016.

[27] L. L. Peterson and B. S. Davie. *Computer Networks: A Systems Approach*. Elsevier, 2007.

[28] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc. of MSST*, 2010.

[29] Amazon DynnamoDB. http://aws.amazon.com/dynamodb/, [accessed in July 2015].

[30] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The Case for Evaluating MapReduce Performance Using Workload Suites. In *Proc. of MASCOTS*, 2011.

[31] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. of IMC*, 2010.

[32] Cisco Nexus 3064 Switch. http://www.cisco.com/c/en/us/products/switches/nexus-3064-switch/, [accessed in July 2015].

[33] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc. of MSST*, 2010.

[34] V. Paxson, M. Allman, J. Chu, and M. Sargent. Computing TCP's Retransmission Timer. Technical report, 2011.

[35] J. Zhang, F. Ren, L. Tang, and C. Lin. Taming TCP Incast Throughput Collapse in Data Center Networks. In *Proc. of ICNP*, 2013.

[36] G. Liu, H. Shen, and H. Wang. Towards long-view computing load balancing in cluster storage systems. *Trans. on Parallel and Distributed Systems, TPDS*, 2016.

[37] Palmetto Cluster. http://citi.clemson.edu/palmetto/.

[38] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *ICDE Workshops*. 2011.