Proactive Incast Congestion Control in a Datacenter Serving Web Applications

Haoyu Wang and Haiying Shen
Department of Computer Science
University of Virginia, Charlottesville, VA 22903, USA
Email: {hw8c, hs6ms}@virginia.edu

***Abstract*—With the rapid development of web applications in datacenters, network latency becomes more important to user experience. The network latency will be greatly increased by incast congestion, in which a huge number of requests arrive at the front-end server simultaneously. Previous incast problem solutions usually handle the data transmission between the data servers and the front-end server directly, and they are not sufficiently effective in proactively avoiding incast congestion. To further improve the effectiveness, in this paper, we propose a Proactive Incast Congestion Control system (*PICC*). Since each connection has bandwidth limit, *PICC* novelly limits the number of data servers concurrently connected to the front-end server to avoid the incast congestion through data placement. Specifically, the front-end server gathers popular data objects (i.e., frequently requested data objects) into as few data servers as possible, but without overloading them. It also re-allocates the data objects that are likely to be concurrently or sequentially requested into the same server. As a result, *PICC* reduces the number of data servers concurrently connected to the front-end server (which avoids the incast congestion), and also the number of connection establishments (which reduces the network latency). Since the selected data servers tend to have long queues to send out data, to reduce the queuing latency, *PICC* incorporates a queuing delay reduction algorithm that assigns higher transmission priorities to data objects with smaller sizes and longer queuing times. The experimental results on simulation and a real cluster based on a benchmark show the superior performance of *PICC* over previous incast congestion problem solutions.**

I. Introduction

Web applications, such as social network (e.g., Facebook, Linkedin), video streaming (e.g., YouTube, Netflix), are widely used in our daily life. A data query of a web application always needs to retrieve many data objects concurrently from different cloud data servers [1–5]. As shown in Figure 1, after receiving a client's data query, the front-end server sends many data requests for data objects stored in all targeted servers and receives thousands of responses simultaneously. Although a high parallelism of data requests can achieve better performance on the back-end side, when a large number of concurrent responses arrive at the front-end server, the switch buffer can not handle all the concurrent responses [6]. Then, it causes packet drops and TCP timeouts, which can introduce retransmission delay and up to 90% throughput reduction [7]. This kind of network congestion is called *incast congestion*, which is a non-ignorable reason of the delay in modern datacenter [1, 2, 8, 9]. Indeed, the incast congestion occurred in *Morgan Stanley's* datacenter greatly degrades the performance [10]. Web application users often have a strict requirement on response latency [11, 12]. For example, data query latency inside Azure storage system needs to be less than $100ms$ [13] to meet user satisfaction and some web applications require much shorter latency such as $178\mu s$ [1]. Moreover, user loyalty is affected by the application's response latency. For example, the sale of Amazon will degrade by one percent when the latency of its web presentation increases as small as $100ms$ [14]. In order to reduce response latency and improve users' experience, it is critical to avoid incast congestion.

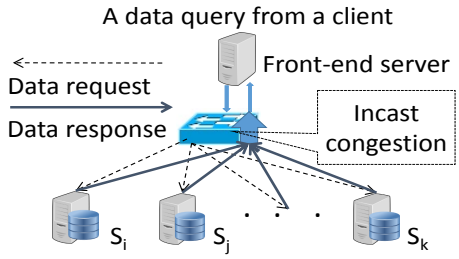


Fig. 1: Illustration of incast congestion.

The incast congestion problem is common and critical in the TCP protocol environment. Many previous solutions for this problem can be classified into three categories: link layer solutions [15–18], transport layer solutions [6, 19–23] and application layer solutions [24–27]. These solutions usually handle the data transmission between the data servers and the front-end server directly, but they are not sufficiently effective in proactively avoiding incast congestion.

To address this problem, in this paper, we propose a new application layer solution, called Proactive Incast Congestion Control (*PICC*) for a datacenter serving web applications. The root cause of the incast congestion problem is the many-to-one concurrent communications between the single front-end server and multiple data servers [8]. Since each connection has bandwidth limit, *PICC* novelly limits the number of data servers concurrently connected to the front-end server to avoid incast congestion through data placement. In this case, a challenge faced by *PICC* is how to satisfy the response latency requirements from users.

To handle this challenge, *PICC* places popular (i.e., frequently requested) data objects into as few data servers as possible, and also stores correlated data objects in the same data server, but without overloading the servers (called gathering servers). Correlated data objects are the data objects

that tend to be requested concurrently or sequentially. For example, different data objects for a webpage are usually requested concurrently, and a webpage indexed by another webpage may be requested sequentially. In addition, since gathering servers tend to have long queues to send out data, in order to reduce the queuing latency in gathering servers, *PICC* assigns different transmission priorities to data object responses in the queue; a data object with a smaller size and longer waiting time has a higher priority to be transmitted out. As a result, when a client sends a data query to the front-end server, the front-end server is likely to send data requests to a limited number of servers. It decreases the number of data servers concurrently connected to the front-end server, which reduces the probability of the incast congestion occurrence. Also, it reduces the number of establishments of the connections between the data servers and the front-end server (especially for the situation that most connections only carry transient transmission of only a few data objects), which reduces connection establishment time and hence the data response latency to the client. Within our knowledge, *PICC* is the first work that focuses on the data placement to proactively avoid incast congestion. We summarize our contribution below:

1. Popular data object gathering. We dynamically gather the popular data objects into as few data servers as possible but without overloading them. Therefore, when a client sends out a data query, the number of data servers concurrently transmitting data to the front-end server is constrained and the probability of incast congestion occurrence is decreased. Also, the number of connection establishments between the data servers and the front-end server is reduced, which reduces data query latency.

2. Correlated data object gathering. We periodically cluster correlated data objects into a group, and then allocate each group to the same data server but without overloading it. In this way, for a client's data query, the data requests issued from the front-end server have a high probability to be sent to only a few servers. It helps reduce the number of data servers concurrently connected to the front-end server and the number of connection establishments, which avoids incast congestion and reduce data query latency.

3. Queuing delay reduction. After we gather popular or correlated data objects into several gathering servers, more data object responses need to be sent from a gathering server and it increases the queuing latency. We propose a queuing delay reduction algorithm to reduce the adverse effect of the head-of-line blocking (i.e., a queue of packets is held up by the first packet which increases the queuing latency). The algorithm assigns a higher priority to the data objects with a smaller size and longer waiting time to be transmitted out, so that the average request latency in the gathering servers is reduced.

The remainder of the paper is organized as follows. Section II presents an overview of the related work. Section III describes the detailed design of *PICC*. Section IV and Section V present the performance evaluation based on a benchmark in simulation and on a real-world testbed, respectively. Section VI concludes this paper on our future work.

II. Related Work

Link layer solutions. The quantized congestion notification (QCN) method [15] was developed for congestion control at the link layer in the datacenter network. It is composed of two parts: the congestion point algorithm which samples the packets only when congestion occurs to evaluate the congestion situation, and the reaction point algorithm which recovers the network congestion reported by the congestion point algorithm. QCN runs on special switch, which is costly and hard to implement in practice. Devkota *et al.* [16] modified QCN by sampling each packet regardless of the occurrence of the congestion in order to get better performance in avoiding congestion. Zhang *et al.* [17] improved QCN by distinguishing each flow based on their sending rates and adjusting the feedback to each flow accordingly. Huang *et al.* [18] proposed to slice the TCP packet into a smaller size to reduce the congestion possibility.

Transport layer solutions. The transport layer solutions are mainly focused on improving TCP protocols. Vasudevan *et al.* [6] proposed disabling the delayed ACK mechanism and conducts a retransmission when the retransmission timeout is reached. ICTCP [19] 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 versa. To improve the downlink bandwidth utilization, DCTCP [20, 21] and MPTCP [28] reduce the window size by a flexible ratio according to current network status. Multipath TCP [29, 30] tries to seek a possible path to transfer data from servers to the front-end server among multiple paths in order to fully utilize the bandwidth of each link of all paths and avoid passing congested links. It also uses the different window sizes for different TCP sub-flows.

Application layer solutions. In [24–27], a short delay is introduced between two consecutive requests by manually scheduling the second response with an extra short delay or re-schedule the transport path for each request in order to reduce the number of concurrently connected data servers to avoid incast congestion. Yang *et al.* [24] proposed inserting one unit time delay between two consecutive requests. The methods in [25, 26] ask the target server to wait for a certain time before transmitting the requested data. Wang *et al.* [27] proposed a transport path re-scheduling method by concentrating all the data requests into a limited number of data servers within one rack. However, the added extra delay or re-scheduled path will increase the response latency of data requests, which may not satisfy the user low-latency requirement on web applications.

Unlike the previous solutions, *PICC* handles the incast congestion problem from a completely different perspective. It novelly uses data placement to limit the number of concurrently connected data servers to the front-end server, while reducing the response latency.

III. Design of the PICC System

In this section, we present the details of our proposed Proactive Incast congestion Control system (*PICC*). The main cause of incast congestion is that many responses arrive at the front-end server simultaneously. The number of concurrent responses, or in another word, the number of servers simultaneously connected to the front-end server may exceed the processing capacity of the front-end server, and then some of the responded data objects will be lost. Therefore, reducing the number of servers concurrently connected to the front-end server can avoid the incast congestion. Rather than relying on short delay insertion between two requests (which introduces extra latency) or sliding window protocol (which may not fully utilize bandwidth), *PICC* has data object placement methods to proactively reduce the number of data servers concurrently connected to the front-end server.

In the current datacenter, when the front-end server receives a query from a client, it sends out multiple data object requests and receives responses from a number of data servers concurrently. In *PICC*, due to its data object reallocation strategies, fewer data servers need to respond to the front-end server for a data query, which helps avoid the incast congestion. Note that when data is already stored in the data storage system, *PICC* conducts data reallocation by transferring data between servers. For a data storage system that follows a certain rule to store and fetch data (e.g., MongoDB [31] key-value storage system), before *PICC* moves data objects from server S_i to server S_j, it creates a link for the data object in S_i pointing to S_j. Then, the data fetching still follows the original procedure and the only difference is that the real data response is transmitted from S_j to the front-end server rather than S_i.

A. Popular Data Object Gathering

The front-end server runs the popular data object gathering algorithm periodically after each time period T. In the following, we explain how to find popular data objects, how to determine the gathering servers that store popular data objects, and how to store popular data objects to the gathering servers.

The popular data objects are identified based on the data object request historical log. Like [32], we also assume that we can use the historical request frequency of one data object to predict its request frequency in the next time period. We use T to denote a time period and use $t \in T$ to denote a time-slot. Upon receiving a client's data query, the front-end server will send out multiple requests to different data servers for data objects. After time period T, the front-end server has a log recording the requesting frequency of each requested data object and its host data server. Based upon the log, the front-end server first sorts all the data objects in the system in descending order of their requesting frequencies. It then selects θ data objects on the top of the sorted list, and notifies their host data servers to transfer these data objects to the gathering servers in the same rack. The front-end server updates the log and conducts the data reallocation periodically. In this way, the popular data list are dynamically updated and popular data objects are stored in several gathering servers.

We need to select the gathering servers from all data servers. For the purpose of minimizing the total size of data objects transferred from other data servers to the gathering servers, we need to select a data server which stores the largest size of data objects requested by the front-end server within each rack during T. Accordingly, based on the recorded log, the front-end server calculates the weight W of data server S_i: $W_{S_i} = \sum_{o \in S_i} B_o * F_o$. Here, $o \in S_i$ denotes each data object stored in data server S_i, B_o denotes the size of data object o, and F_o denotes the requesting frequency of data object o during time period T. For each rack, the front-end server sorts data servers in descending order of server weights. To limit the transfer distance of popular data objects in order to constrain the overhead, we select gathering servers in each rack, so that popular data objects only need to be transferred within a rack.

Among the selected θ popular data objects, the front-end server finds the data objects in each rack and estimates their total demand on different resources (storage, computing, I/O and bandwidth). Then, from the top of the sorted data server list of the same rack, it selects a few data servers to function as gathering servers, whose total resource capacity for each resource is no less than the total resource demand of the popular data items in the rack. This step is to avoid data object transmission from their original servers to the gathering servers and limit the number of gathering servers while avoiding overloading them. We could select only one gathering server first and only when the server cannot host more popular data objects, we then select the next server. However, it is possible that a popular data object is already stored in a top server, which is not the gathering server. Therefore, selecting top a few servers avoids data transmission in such a case.

Next, the front-end server assigns the popular data objects in each rack to the gathering servers in the rack. For each data object, the font-end server checks whether it is stored in a gathering server in the rack. If yes, it continues to check the next data object; otherwise, the data object is transferred from its current data server to the nearest gathering server in the same rack that has sufficient capacity for it. Note that it is possible that all selected gathering servers do not have enough capacities to host a data object due to resource fragmentation in placing data objects to the servers, though their total available capacity is no less than the total demand on each resource of the popular data objects in their rack. In this case, the top server in the sorted data server list is selected as a new gathering server to host this data object. This process repeats until the popular data objects in each rack are all reallocated to gathering servers in the same rack. As a result, the popular data objects in a rack are always gathered in a few gathering servers (but without overloading them), which limits the number of servers concurrently connected to the front-server and hence helps avoid incast congestion. Note that since popular data objects are stored in these gathering servers, the weights of these servers maintain high in their racks. As a result, the gathering servers are unlikely to be changed once they are selected at the first time.

Algorithm 1: Popular data object gathering algorithm.

```
/* select popular data objects periodically */
Record the request frequencies of all the data objects;
Sort the data objects in descending order of request frequency;
Select top θ data objects in the sorted list as popular data objects;
/* select gathering servers and reallocate popular data objects */
for each rack do
    Sort the data server list in descending order of server weight W;
    From the top of the sorted server list
    Select gathering servers that can host all the popular data objects
      in the same rack;
for each data object o in popular data list
do
    if o is in a gathering server in its rack then
        Continue;
    else
        if a gathering server in the same rack has enough capacity
          then
            Reallocate to the nearest gathering server in this rack;
            else
                Select a new gathering server from the sorted server
                  list of the same rack and reallocate;
```

Algorithm 1 shows the pseudo-code of the popular data object gathering algorithm. Figure 2 shows an example of this algorithm. In this figure, each red rectangular represents one popular data object. Without this algorithm, in time period T_1, the datacenter distributes all popular data objects into several random servers. Then, one data query to the front-end server generates several data object requests targeting servers S_i,S_j,.....,S_k. As a result, a larger number of servers will be connected to the front-end server and respond to the front-end server concurrently, which is likely to cause incast congestion on the front-end server side.

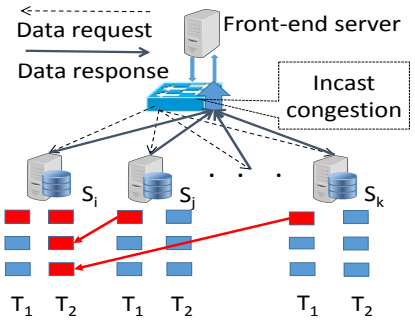


Fig. 2: Popular data object gathering.

In our proposed popular data object gathering algorithm, the front-end server reallocates popular data objects into only a few gathering servers in each rack periodically. In this way, when the front-end server processes data queries from the clients, it requests and receives data objects mainly from the gathering servers. In Figure 2, in time period T_2, S_i stores one popular data object, so it is selected as the gathering server to store all popular data objects. The popular data objects are transferred from their previous servers to S_i, as indicated by red lines in the figure. After the popular data reallocation, when the front-end server receives queries from the clients, it mainly requests data objects from S_i. As the number of data servers concurrently connected to the front-end server is reduced from three to one, the probability of incast congestion occurrence is reduced.

B. Correlated Data Object Gathering

In the popular data object gathering method, we allocate top θ popular data objects to as few gathering servers as possible to reduce the number of servers concurrently responding to the front-end server. In order to further reduce the number of servers concurrently responding to the front-end server, we propose the correlated data object gathering method. Note that some data objects are usually requested concurrently or sequentially. For example, different data objects for a webpage are usually requested concurrently, and a webpage indexed by another webpage may be requested sequentially. Our method clusters the correlated data objects into the same group and allocate each group into the same gathering server.

After we gather correlated data objects into the same data server, then data requests for correlated data objects are generated from the front-end server to the same data server through the connection that is already established. It reduces the data transmission latency caused by the connection rebuilding delay between the front-end server and data servers [33]. It also decreases the number of servers concurrently connected to the front-end server since the front-end server generates requests to a limited number of servers at the same time and the connection lifetime is longer than the previous method. Consequently, data objects are transferred from a limited number of data servers with non-stop connections with the front-end sever.

In the following, we introduce how to find correlated data objects. We first introduce a concept of *data object closeness* between two data objects to represent the likelihood that the two data objects will be requested concurrently or sequentially. After each time period T, from the data request log, the front-end server can derive the frequency that two data objects, o_1 and o_2, are requested concurrently during each time-slot t, denoted by $P_t(o_1, o_2)$. It can also derive the frequency that two data objects, o_1 and o_2, are requested sequentially during each time-slot t, denoted by $Q_t(o_1, o_2)$. Then, the closeness of o_1 and o_2 for time period T is calculated by:

$$C_T(o_1,o_2) = \alpha * \sum_{t\in T} P_t(o_1,o_2) + \beta * \sum_{t\in T} Q_t(o_1,o_2) + (1-\alpha-\beta)C_{T-1}(o_1,o_2). \quad (1)$$

Here, α and β are the weights for the concurrent request frequency and sequential request frequency. The inclusion of $(1-\alpha-\beta)C_{T-1}(o_1,o_2)$ is for the purpose of reflecting the closeness of two data objects in the long term.

After the front-end server calculates the closeness of every two data objects, it builds an undirected graph $G(V,E)$, where V is the set of all data objects, E is the set of all edges connecting data objects, and the weight of each edge connecting data objects o_1 and o_2 is their current closeness $C_T(o_1,o_2)$. It then uses the minimum cut tree based algorithm [34] to divide the graph vertices to clusters. The data objects in each cluster are correlated data objects. Algorithm 2 shows the pseudo-code of the data object clustering algorithm. The algorithm

returns all linked sub-graphs as the clusters of G so that the front-end server can store data objects in one cluster to the same data server.

When the front-end server notifies the data server of a popular data object to transfer it to a gathering server, it also notifies the data servers of other data objects in the same cluster of this popular data object to transfer them to the gathering server. Some data object clusters may not contain popular data objects. For such a data object cluster, the front-end server finds the data server that stores the most of the data objects in the cluster and has sufficient capacity to store other data objects in the cluster, say S_i, and notifies the data servers of the other data objects in the cluster to transfer them to S_i. Since the popularity of data objects may vary [35], we periodically run this algorithm to maintain the data objects with high closeness in the same gathering server to avoid incast congestion and reduce response latency.

Algorithm 2: Correlated data object clustering algorithm.

```
V' = V ∪ s; //s is the artificial sink;
Graph generation Link s to each data object v to generate graph G'(V', E');
for all nodes v ∈ V do
    Link V to s with the weight w;
Generate the minimum cut tree T' of G' [36];
Remove s from T';
Divide G into N clusters; //N is the number of servers in the system;
Return the clusters of G
```

C. Queuing Delay Reduction

A gathering server stores a large number of popular data objects, which are requested frequently to send a large amount of data by the front-end server, or stores correlated data objects that tend to be concurrently or sequentially requested by the front-end server. Each gathering server maintains a sending queue of all requested data objects and sends them out sequentially. In order to minimize the average waiting and transmission time per data object, we propose a queuing delay reduction algorithm that reduces the adverse effect of head-of-line blocking by setting different priorities for the data objects based on their sizes and waiting times. That is, a data object with a smaller data size and longer waiting time has a higher priority to be transmitted first. According to [37], the transmission latency and the queuing latency is in microsecond scale. For a data object o, we use τ_o to denote its waiting time in the queue. We then calculate the priority value of data object o, denoted by M_o, by:

$$M_o = \tau_o^3 / B_o \tag{2}$$

To place more weight on waiting time when determining the priority of data objects, we triple the value of τ. This exponent can be set to another value depending on how much weight the system wants to give to the waiting time. The data objects in the queue will be re-ordered based on their priority values.

Figure 3 shows an example of queuing delay optimization process. Each rectangular represents one data object and the size of the rectangular means the size of the data object. Assume that the four data objects have the same waiting time. In situation 1, the four data objects in the sending queue have sizes *400kb, 2kb, 2kb, 200kb* in sequence. Then, the queue will be blocked by the 400kb red data object and other three blue objects have to wait in the queue before the red data object is sent out. Assume that the data uploading speed is g and the

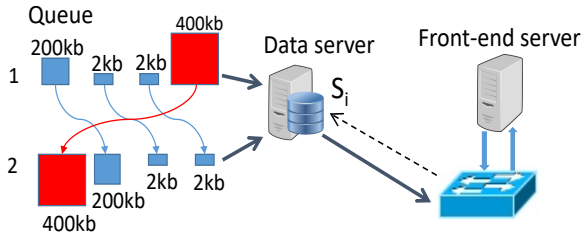


Fig. 3: An example of queue reordering.

size of a transmission unit is 400kb. Then, the average waiting and transmission latency equals $(400/g + 604/g)/4 = 251/g$. In situation 2, we use the queuing delay reduction algorithm to reorder the data objects in the queue. The optimized order of the four data objects in the queue is *2kb, 2kb, 200kb, 400kb*, and the average latency equals $(204/g + 604/g)/4 = 202/g$. The improved queue achieves about *49/g* less latency than the original queue. In our proposed queuing delay reduction algorithm, although the latency of the lower priority objects will be increased, the average latency per data object in the queue will be greatly reduced.

Since the length of the entire sending queue of a gathering server is very long and the long reordering time may introduce high latency before data transmission, we propose to only schedule the beginning m (m is a much smaller integer than the length of the sending queue, e.g., 10) data objects at a time, which form a sub-queue. Considering that the length of a sub-queue is much shorter than the entire queue, the reordering will be faster, which prevents delaying data transmission. Algorithm 3 shows the pseudo-code of the queuing delay reduction algorithm.

Algorithm 3: Queuing delay reduction algorithm.

```
for all the data objects waiting to transfer out;
do
    Select top m data objects to generate a sub-queue;
    for each data object o in the sub-queue;
     do
        Calculate the priority value M_o according to Formula (2);
    Sort the data objects in the sub-queue according to M_o;
    Transfer out all the data objects in the sub-queue;
```

IV. Performance Evaluation in Simulation

We used simulation to conduct large-scale experiments since a real cluster cannot provide large-scale experiment environment. We developed a simulator in Java with packet-level transmission and constructed a typical fat-tree structure [38] using 4000 data servers with 50 data servers inside each rack [39]. In the simulation, we set the capacity of the downlink, uplink and buffer size of each edge-switch to 10Gbps, 10Gbps and 1000kb, respectively [19]. For each data object, it has three replicas randomly distributed on three different data servers in different racks. The size of each data object was

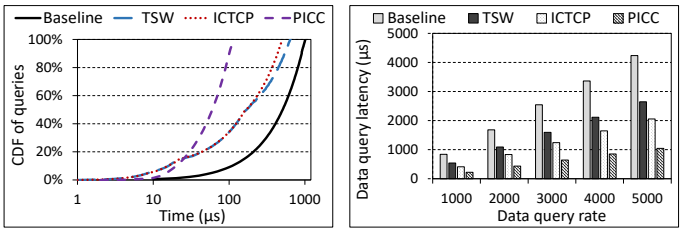
randomly chosen from [20B,1024B] [1]. There are 10^5 data objects in the datacenter.

We used the Yahoo! Cloud Serving Benchmark (YCSB) [40] to generate workload for data requests. YCSB is an open source benchmark used to test the performance of storage system. We chose the *zipfian* YCSB setting, in which the front-end server requests data objects according to the Zipfian distribution. We also manually set the request probabilities of popular data objects and regular data objects, generated the workload based on the probabilities [35], and repeated all experiments. These experimental results are similar to those with YCSB. Due to space limit, please refer to [41] for the experimental results with the manual workload setting. The timeout and the number of TCP packet retransmissions were set to $1ms$ and 5, respectively [19]. We simulated the incast congestion situation with one front-end server requesting data objects from multiple data servers. In every experiment, all the data queries are generated by the front-end server and considered as the historical data for determining gathering servers. The data query rate means the number of data queries sent from the front-end server per hour. We set T to $5mins$ and set the default θ to 1000. We measured the performance of each query after the front-end server receives all the queried data objects of the query and then calculated the average per query [19]. We repeated each experiment for ten times, and report the average result.

We compared the performance of *PICC* with three other representative methods: *Baseline*, TCP sliding window protocol (We denote it as *TSW*) [1], and *ICTCP* [19]. We used *Baseline* as a baseline for the comparison without using any incast congestion problem solutions. In this method, data objects are randomly distributed to data servers. For a data query from a client, the front-end server sends requests simultaneously to all the targeted data servers. In *TSW*, the front-end server can adjust the number of its concurrently connected servers using the classical sliding window protocol. The window size will increase one by one until the front-end server detects the incast congestion occurrence, and then the window size will decrease to half of the previous window size. *ICTCP* [19] adjusts the sliding window size by detecting the bandwidth utilization. It divides total bandwidth into two parts. The first part is used to receive all the traffic and predict the bandwidth utilization. It then adjusts the window size in the second part based upon the predicted bandwidth utilization in order to fully utilize the available bandwidth without over-utilizing the bandwidth capacity. In the *transport layer solutions,* since *ICTCP* performs better than DCTCP [19], which is an optimized TCP protocol widely used in current datacenters, we selected *ICTCP* as a comparison method.

A. Performance of Query Latency

Each data query consists of multiple data requests for different data objects. The request latency for a data object is the time period between the time a front-end server sends a request to a data server and the time it receives the requested data object. The longest time among all the requests of a query is the latency of this query. Figure 4(a) and Figure 4(b) show the Cumulative Distribution Function (CDF) of data queries versus the data query latency, and the data query latency versus data query rate. From both figures, we see the query latency results follow *PICC*<*ICTCP*<*TSW*<*Baseline*. In *Baseline*, without any solutions to avoid incast congestion, many simultaneous responses to the front-end server cause incast congestion, leading to data packet retransmission and hence high transmission latency. *TSW* reduces the number of servers connected to the front-end server once it exceeds the capacity of the maximum window size (which causes incast congestion). Therefore, *TSW* achieves better performance than *Baseline* in terms of data query latency. However, *TSW* produces a longer request latency than that of *ICTCP*. *TSW* cannot fully utilize the bandwidth when the window size decreases to its half size. Also, the window size increases until the incast congestion occurs, which leads to packet loss and data retransmission. *ICTCP* improves the sliding window protocol to fully utilize available bandwidth and meanwhile avoids increasing the window size beyond the receiving capacity of the receiver. Thus, *ICTCP* generates shorter query latency than *TSW*. *PICC* generates lower query latency than *ICTCP*. *PICC* stores popular data objects into several gathering servers and stores correlated data objects into the same server. In this way, most of the requests can be responded continuously from a limited number of servers by fully utilizing the bandwidth. Furthermore, since many of the responses can be generated by one server, there is no extra delay introduced from new connection establishments.



(a) CDF of data query latency (b) Data query latency

Fig. 4: Performance of data query latency.

Figure 4(b) also shows that the data query latency of all methods increases proportional to the data query rate. A higher data query rate means that more data objects need to be transmitted from each data server during unit time, which generates a longer data transmission time. These figures indicate that *PICC* generates the shortest data query latency among all methods by proactively avoiding incast congestion and fully utilizing the downlink bandwidth of the front-end server.

B. Performance of Data Transmission Efficiency

Data object transmissions and retransmissions from the data servers to the front-end server lead to inter-rack packet transmissions. *PICC* additionally produces inter-rack packet transmissions caused by the inter-rack data reallocation. A smaller number of inter-rack packets leave more bandwidth for the connection between the front-end server and data servers, leading to higher throughput. At the same time, the bandwidth

of links of an aggregation router is much lower than the total downlink bandwidth of all data servers connecting to this router. Therefore, it is important to reduce the number of inter-rack packets (that are transmitted between racks).

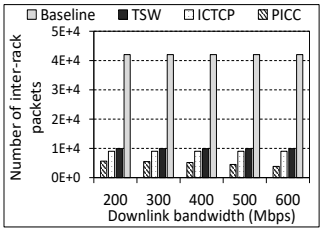


Fig. 5: Inter-rack transmissions.

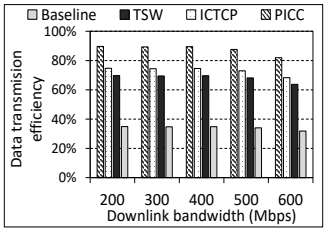
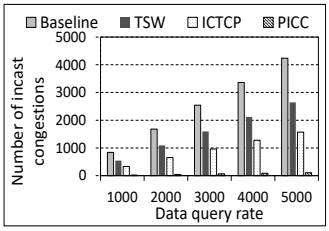


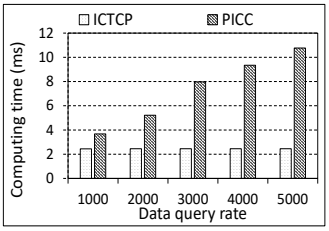
Fig. 6: Transmission efficiency.

Figure 5 shows the number of inter-rack packets of different methods with different downlink bandwidths of the front-end server. We see that the results follow *PICC*<*ICTCP*<*TSW*<*Baseline*. *Baseline* has the largest number of inter-rack packets since it has the highest probability of generating incast congestion without any solutions to avoid the incast congestion. For *TSW*, all the data servers directly respond to the front-end server based upon the sliding window protocol, which can reduce the inter-rack packet retransmission. In *ICTCP*, it improves the sliding window protocol in avoiding the incast congestion by predicting the bandwidth utilization to fully utilize the available bandwidth. With fewer incast congestion occurrences, *ICTCP* generates fewer data object retransmissions so that it reduces the number of inter-rack packets compared with *TSW*. By gathering popular data objects and correlated data objects into a limited number of servers, *PICC* proactively avoids incast congestion and produces the lowest number of inter-rack packets even though it sometimes needs inter-rack packet transmission for data reallocation. In summary, *PICC* generates the smallest number of inter-rack packets since it reduces the incast congestion occurrences and data retransmissions compared with other methods.

We then measure the data transmission efficiency by $\frac{size}{latency}/BW$, where $size$ is the total size of all the requested data of a query, $latency$ is actual query latency of the query, and BW is the downlink bandwidth of the front-end server. Figure 6 shows the average data transmission efficiency per query of the four methods versus different downlink bandwidths of the front-end server. The results follow *PICC*>*ICTCP*>*TSW*>*Baseline*. Furthermore, the result of *PICC* increases as the downlink bandwidth decreases, while other methods keep nearly constant. For *Baseline*, all the requests are sent from the front-end server and may be responded concurrently, which leads to incast congestion and long transmission latency due to retransmissions. *TSW* with the sliding window protocol achieves better performance than *Baseline*. In *ICTCP*, it adjusts half of bandwidth based upon the bandwidth utilization, which leads to higher bandwidth utilization and higher data transmission efficiency than *TSW*. *PICC* transfers popular and correlated data objects into a limited number of servers, and then the front-end server has a higher probability to request data objects from these data servers, so that its downlink bandwidth can be more fully utilized. In summary, *PICC* achieves the best performance in data transmission efficiency and bandwidth utilization compared with other three methods.



(a) Number of incast congestions



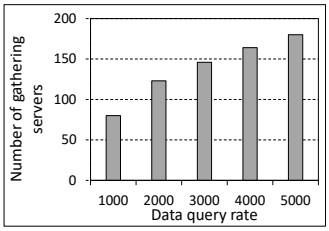
(b) Algorithm computing time

Fig. 7: Incast congestion avoidance and computing time.

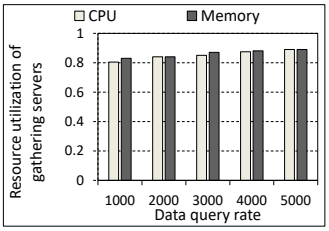
Figure 7(a) shows the number of incast congestions occurred. We see that *TSW* and *Baseline* generate dramatically more incast congestions than *ICTCP* and *PICC*, and *PICC* generates significantly fewer incast congestions than *ICTCP* due to the reasons explained above. Also, as the data query rate increases, the number of incast congestions of these four methods grows since more data queries in a unit time period lead to a higher possibility of incast congestion occurrence.

Figure 7(b) shows the average computing time of *PICC* and *ICTCP* for the data reallocation scheduling and window size adjustment calculation per data query, respectively. The computing time of *Baseline* is 0 and the computing time of *TSW* is negligible. The results show that the computing time of *PICC* is higher than that of *ICTCP*. Also, as the data query rate increases, the computing time of *PICC* increases. Because *PICC* needs to find popular data objects and correlated data objects, more data queries cause more computing time. We also see that even for 5000 data query rate, the computing time is only $11ms$, which is very small compared with the entire data transmission latency reduction in Figure 4. In summary, compared with *ICTCP*, *PICC* greatly reduces the number of incast congestions with reasonably higher computing time.

C. Sensitivity Evaluation and Effectiveness of Each Method



(a) Number of gathering servers

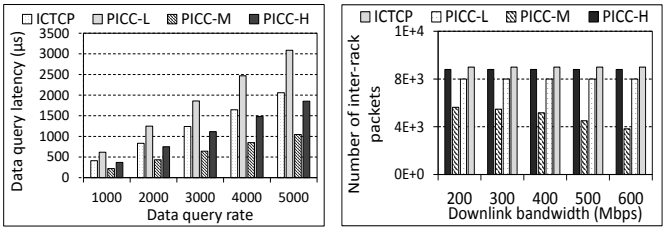


(b) Effectiveness of gathering servers

Fig. 8: Performance of gathering servers.

1) Performance of the Popular Data Object Gathering Method: Figure 8(a) shows the number of gathering servers versus the data query rate. We see that the number of gathering servers is increased with the data query rate. This is because the resource capacities of gathering servers must be high enough to handle the resource demands of the selected popular data objects, and higher query rate causes higher resource demands. Figure 8(b) shows the resource utilization (the usage

percent of CPU and memory) of gathering servers versus the data query rate. Both of the CPU and memory utilizations are around 80% to 90%, which means that the resources of the gathering servers are almost fully utilized but they are not overloaded.

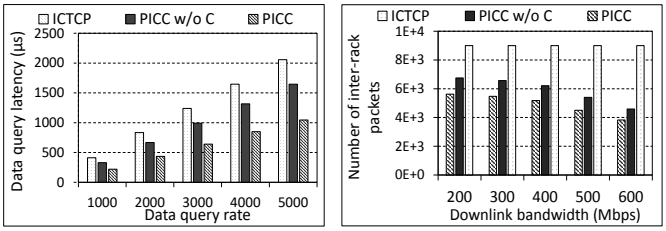


(a) Data query latency (b) Inter-rack data transmission

Fig. 9: Performance of different θ settings.

We measured the performance of the popular data object gathering method with different θ threshold settings. We use *PICC-L*, *PICC-M* and *PICC-H* to denote *PICC* when θ equals to 10, 1000 and 10000, respectively. Since *Baseline* and *TSW* always have worse performance than *ICTCP*, we only compared *ICTCP* with *PICC* here. Figure 9(a) shows the data query latency versus the data query rate. It shows that *PICC-M* produces the lowest query latency, but *PICC-H* generates higher query latency than *ICTCP*. *PICC-H* sets a high θ threshold, so that more data objects being transferred to a gathering server may lead to congestion on it, and lower the transmission bandwidth between the gathering server and the front-end server. Finally, it leads to more data retransmission and increases data query latency. *PICC-L* does not aggregate enough popular data objects in a few gathering servers, so the number of concurrently connected data servers to the front-end server is not sufficiently reduced, leading to incast congestion and higher query latency. Therefore, an appropriate setting for the θ threshold is important.

Figure 9(b) shows the number of inter-rack packets versus the downlink bandwidth of the front-end server. The results follow *ICTCP*>*PICC-H*≈*PICC-L*>*PICC-M* due to the same reasons as explained above.



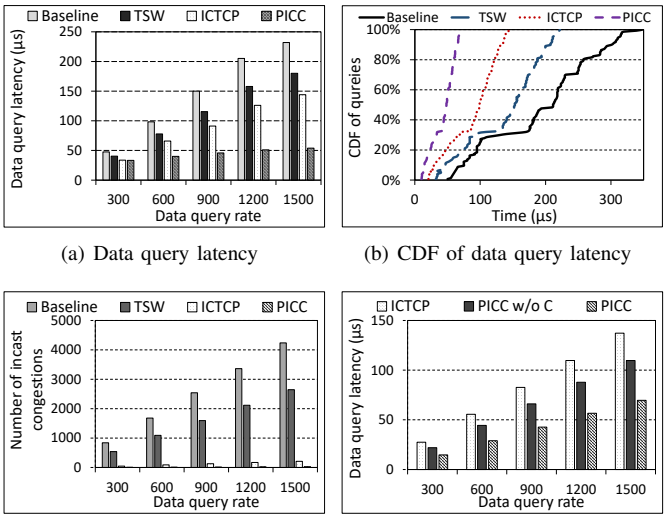
(a) Data query latency (b) Inter-rack data transmission

Fig. 10: Performance of the correlated data object gathering method.

2) Performance of the Correlated Data Object Gathering Method: In order to measure the effectiveness of the correlated data object gathering method, we tested the performance of *PICC* without this method, denoted by *PICC w/o C*. Figure 10(a) shows the data query latency versus the data query rate. Figure 10(b) shows the number of inter-rack packets versus different downlink bandwidths of the front-end server. The correlated data object gathering method gathers the data objects that tend to be requested concurrently or sequentially in the same data server. Then, the front-end server sends requests to fewer data servers, which reduces the number of data servers concurrently connected to the front-end server and the probability of incast congestion occurrence. Therefore, *PICC* produces lower query latency than *PICC w/o C*.

V. Performance on a Real Testbed

We implemented *PICC* and other comparison methods on Palmetto [42]. All the servers we use are with 2.4G Intel Xeon CPUs E5-2665 (16 cores), 64GB RAM, 240GB hard disk and 10G NICs. The operating system of each server is Ubuntu 15.10 LTS version. The CPU, memory and hard disk never became a bottleneck in any of our experiments. We randomly selected 150 servers and one front-end server from all servers, each of which has the downlink and uplink as 10Gbps. We randomly distributed 5000 data objects into the data servers, and the size and the number of replicas of each data object follow the same settings as in our simulation. We also use YCSB to generate workload with the same settings as in the simulation.



(a) Data query latency (b) CDF of data query latency

(c) Number of incast congestions (d) Effectiveness of *PICC* methods

Fig. 11: Performance on a real cluster.

Figure 11(a) shows the query latency of all methods versus the data query rate. Figure 11(b) shows the CDF of queries over time of all methods. Both figures show the same order and relationship between different methods as in Figure 4 in the simulation due to the same reasons. Both of the figures indicate that *PICC* has the best performance in query latency. Figure 11(c) shows the number of incast congestion occurrences versus the data query rate. It shows the same order and trend of different methods due to the same reasons as in Figure 7(a). The figure indicates that *PICC* can greatly reduce the number of incast congestion occurrences. Figure 11(d) shows the data query latency of *ICTCP*, *PICC* and *PICC w/o C* versus the data query rate. *PICC* has shorter query latency than *PICC w/o C*. The results indicate that both the popular data object gathering method and the correlated data object

gathering method are effective in reducing the data query latency.

VI. Conclusion

Web applications are featured by a very large number of data object responses for a data query, which may cause incast congestion and makes it difficult to meet the stringent low-delay response requirements. We propose a Proactive Incast Congestion Control system (*PICC*), which is the first work that focuses on the data placement to proactively avoid incast congestion within our knowledge. First, *PICC* reallocates popular data objects into as few gathering servers as possible. Second, *PICC* reallocates data objects that tend to be concurrently or sequentially requested into the same data server. Such data reallocation reduces the number of data servers concurrently connected to the front-end server and reduces the number of connection establishments, which help avoid incast congestion and reduce query latency. Third, considering that the gathering servers may introduce extra queuing latency, *PICC* further incorporates a queuing delay reduction algorithm to reduce the average latency per data object. The experiments both in simulation and a real cluster based on a benchmark show that *PICC* greatly reduces data query latency and the probability of the incast congestion occurrence. In the future, we will study how to determine appropriate parameter (e.g., θ) values and how to further reduce the overhead of data reallocation.

Acknowledgment

This research was supported in part by U.S. NSF grants OAC-1724845, ACI-1719397 and CNS-1733596, and Microsoft Research Faculty Fellowship 8300751. We would like to thank Dr. Yuhua Lin's help on this paper.

References

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

[2] A. Shalita, B. Karrer, I. Kabiljo, A. Sharma, A. Presta, A. Adcock, H. Kllapi, and M. Stumm. Social hash: an assignment framework for optimizing distributed systems operations on social networks. In *Proc. of NSDI*, 2016.

[3] 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.

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

[5] H. Wang, J. Gong, Y. Zhuang, H. Shen, and J. Lach. Healthedge: Task scheduling for edge computing with health emergency and human behavior consideration in smart homes. In *Proc. of Big Data*, 2017.

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

[7] L. Li, K. Xu, D. Wang, C. Peng, K. Zheng, R. Mijumbi, and Q. Xiao. A Longitudinal Measurement Study of TCP Performance and Behavior in 3G/4G Networks Over High Speed Rails. *Trans. on Networking*, 2017.

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

[9] J. Huang, T. He, Y. Huang, and J. Wang. ARS: Cross-layer adaptive request scheduling to mitigate TCP Incast in data center networks. In *Proc. of INFOCOM*, 2016.

[10] G. Judd. Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter. In *Proc. of NSDI*, 2015.

[11] W. Chen, J. Rao, and X. Zhou. Preemptive, Low Latency Datacenter Scheduling via Lightweight Virtualization. In *Proc. of ATC*, 2017.

[12] D. Crankshaw, X. Wang, G. Zhou, M. Franklin, J. Gonzalez, and I. Stoica. Clipper: A Low-Latency Online Prediction Serving System. In *Proc. of NSDI*, 2017.

[13] Z. Wu, C. Yu, and H. Madhyastha. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *Proc. of NSDI*, 2015.

[14] R. Kohavl and R. Longbotham. Online Experiments: Lessons Learned. *Computer*, 2007.

[15] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, R. Pan, and B. Prabhakar. Data center transport mechanisms: Congestion control theory and ieee standardization. In *Proc. of Allerton*, 2008.

[16] P. Devkota et al. Performance of quantized congestion notification in TCP incast scenarios of data centers. In *Proc. of MASCOTS*, 2010.

[17] Y. Zhang and N. Ansari. On mitigating TCP incast in data center networks. In *Proc. of INFOCOM*, 2011.

[18] J. Huang, Y. Huang, J. Wang, and T. He. Packet slicing for highly concurrent TCPs in data center networks with COTS switches. In *Proc. of ICNP*, 2015.

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

[20] 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.

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

[22] M. Flores, A. Wenzel, and A. Kuzmanovic. Enabling router-assisted congestion control on the Internet. In *Proc. of ICNP*, 2016.

[23] G. Vardoyan, N. S. Rao, and D. Towsley. Models of TCP in high-BDP environments and their experimental validation. In *Proc. of ICNP*, 2016.

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

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

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

[27] H. Wang, H. Shen, and G. Liu. Swarm-based Incast Congestion Control in datacenter Serving Web Applications. In *Proc. of SPAA*, 2017.

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

[29] Q. Peng, A. Walid, J. Hwang, and S. Low. Multipath TCP: Analysis, design, and implementation. *Trans. on Networking*, 2016.

[30] B. Hesmans and O. Bonaventure. Tracing multipath TCP connections. *Proc. of SIGCOMM*, 2015.

[31] MongoDB. https://www.mongodb.com/, [Accessed in July 2017].

[32] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, and H. Li. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proc. of ATC*, 2013.

[33] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proc. of NSDI*, 2014.

[34] G. W. Flake and K. Tarjan, R.and Tsioutsiouliklis. Graph clustering and minimum cut trees. *Internet Mathematics*, 2004.

[35] D. Boru, D. Kliazovich, F. Granelli, P. Bouvry, and A. Y. Zomaya. Energy-efficient data replication in cloud computing datacenters. *Cluster computing*, 2015.

[36] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *J. SIAM*, 1961.

[37] C. Lee, C. Park, K. Jang, S. Moon, and D. Han. Accurate Latency-based Congestion Feedback for Datacenters. In *Proc. of ATC*, 2015.

[38] J. McCauley, M. Zhao, E. J. Jackson, B. Raghavan, S. Ratnasamy, and S. Shenker. The Deforestation of L2. In *Proc. of SIGCOMM*, 2016.

[39] A. Putnam, A. M. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, and J. Fowers. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proc. of ISCA*, 2014.

[40] B. F. Cooper, A. Silberstein, E. Tam, and R. Ramakrishnan. Benchmarking cloud serving systems with ycsb. In *Proc. of SOCC*, 2010.

[41] Evaluation supplement. https://www.dropbox.com/s/m4sndwtrw51zsf5/Infocom-long-version.pdf?dl=1, [Accessed in July 2017].

[42] Palmetto Cluster. http://citi.clemson.edu/palmetto/index.html, [Accessed in MAY 2016].