Reducing Network Latency for Web Applications in a Datacenter

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

in partial fulfillment

of the requirements for the degree

Doctor of Philosophy

by

Haoyu Wang

May 2021

APPROVAL SHEET

The dissertation is submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Haoyu Wang

The dissertation has been read and approved by the examining committee:

Haiying Shen, Advisor

Yangfeng Ji, Committee chair

Lu Feng

Charles Reiss

Cong Shen

Accepted for the School of Engineering and Applied Science:

Craig H. Benson, Dean, School of Engineering and Applied Science

---

May 2021

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 congestion 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. Generally, the proposals to solve this problem have focused either on refining existing window-based congestion control like in TCP or on introducing a distributed controller to make congestion control decisions.

In this dissertation, we introduce a Swarm-based Incast Congestion Control (SICC) system and a Proactive Incast Congestion Control system (PICC) which focuses on incast congestion problem, and a Neighbor-aware Congestion Control algorithm based on Reinforcement Learning (NCC) for general congestion control. 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 incast congestion. Also, the continuous data transmission from hubs to the front-end server facilitates the development of other strategies to further control 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. In PICC, the front-end server gathers popular data objects (i.e., frequently requested data objects) into as few data servers as possible. It also re-allocates the data objects that are likely to be concurrently or sequentially requested (called correlated data objects) into the same

server. As a result, PICC reduces the number of data servers concurrently connected to the front-end server, and the number of establishments of the connections between data servers and the front-end server, which avoids incast congestion and reduces the network latency. The large number of data transmissions between the data servers storing popular or correlated data objects and the front-end server may produce high queuing latency in the data servers. To reduce the queuing latency, PICC incorporates a queuing reduction algorithm that assigns higher transmission priorities to data objects with smaller sizes and longer queuing times. In NCC, the rate limiting decisions on one node are driven by the local agent that uses reinforcement learning to optimize a combination of overall latency, throughput and the shared information. To make this approach efficient, the local agents choose overall rate limits for each node, and then a separate process assigns the traffic of individual flows within these limits. We conclude that these congestion control systems in a datacenter will help reduce network latency, avoid congestion and improve the quality of service of clients. This dissertation provides an overview of the scope of congestion control and network optimization within a datacenter, some of the key challenges in building congestion control systems, hypothesized contributions. The proposed systems achieve better congestion avoidance than several end-to-end and centralized mechanisms in prior work.

Contents

List of Figures

Acknowledgements

I would like to thank the many people who have helped and supported me through this long Ph.D. journey. First, I would like to thank my advisor, Professor Haiying Shen, whose encouragement, guidance, and support through these years to develop a comprehensive understanding of the subject. She has been instrumental in teaching me how to focus on the critical aspects of any research project, and formulate my thoughts in the right context and in the right direction. I learned from her how to approach research in a rigorous and efficient manner. I acknowledge my committee members, Professor Yangfeng Ji, Professor Lu Feng, Professor Charles Reiss, and Professor Cong Shen for providing me valuable comments and suggestions on this dissertation.

In completing this dissertation, I am also intellectually indebted to the committee members, Prof. Yangfeng Ji, Prof. Lu Feng, Prof. Charles Reiss and Prof. Cong Shen, who has been an unlimited source of honest and constructive comments.

I am grateful to my collaborators and lab mates, Guoxin Liu, Kang Chen, Chenxi Qiu, Yuhua Lin, Liuhua Chen, Bo Wu, Zhuozhao Li, Li Yan, Jinwei Liu, Ankur Sarker, Liuwang Kang, Jiechao Gao, Zetian Liu, Tanmoy Sen, Shohaib Mahmud, Tao Zhang, Dapeng Qu, Xingbo Fu, Sudipta Saha Shubha, Alexandra Dejong who shared their time, ideas and enthusiasm. This dissertation would not be possible without their earnest support. I have been fortunate to collaborate with each one of them. I also would like to thank the undergraduate students in UVA, Kevin Zheng,

Xiaoying Li, Daniel Zheng, Daniel Mao, and Grady Roberts.

I would like to thank to my parents for their love and encouragement, without which none of my achievements would have been possible. Thanks to them for always having confidence on me. All my friends and family have been a source of inspiration in my life.

Last but not least, I would like to thank my beloved girlfriend Qi Liu for her understanding and unreserved support in these years. Without having her beside me, this Ph.D. journey would have felt much longer and more stressful.

I am truly grateful to all those who helped me complete this dissertation. If I neglected to mention you, I apologize. Please know that I greatly appreciate your support.

Chapter 1

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 servers within one datacenter [49, 50, 56, 69, 80]. As shown in Figure 1.1, 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 buffers (between the data servers and the front-end server) within the datacenter may not handle all the concurrent responses [78]. Then, it causes packet drops and TCP timeouts, which can introduce retransmission delay and up to 90% throughput reduction [44].

This kind of network congestion is called *incast congestion*, which is a non-ignorable reason of high response time in modern datacenter [30, 56, 62, 69], like *Morgan Stanley's* datacenter [35]. Web application users often have a strict requirement on response latency [13, 20]. For example, data query latency inside Azure storage system needs to be less than $100ms$ [85] to meet user satisfaction and some

web applications require much shorter latency such as 178μs [56]. 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 [39]. Since the incast congestion can highly affect the web application latency, in order to reduce response latency and improve users' experience, it is critical to avoid incast congestion in a datacenter.

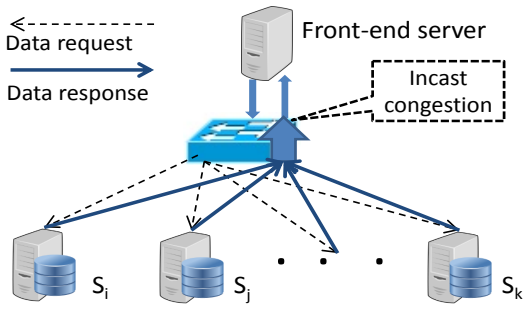


Figure 1.1: Illustration of incast congestion.

The root cause of incast congestion problem is the many-to-one concurrent communications between the single front-end server and multiple data servers. In general, incast congestion problem is common and critical in the TCP protocol environment. Many previous works have been proposed to handle this problem. They can be classified into three categories: link layer solutions [3, 21, 31, 36, 92], transport layer solutions [4, 25, 75, 77, 78, 84] and application layer solutions [63, 64, 88]. The link layer solutions mainly use the quantized congestion notification algorithm. In this algorithm, once incast congestion happens in the front-end server or TCP switch, it can send rapid packet loss notifications to the packet senders (i.e., the data servers stored the related data objects) asking them to re-send the lost packets. This algorithm needs special hardware to implement [15, 91]. The transport layer solutions try to improve the TCP sliding window protocol to avoid packet loss. The TCP sliding window protocol adaptively adjusts the size of receiver window by measuring the available bandwidth and throughput in each control interval. In the application layer

solutions, some methods schedule all the requests by introducing extra delay between requests. However, always inserting pre-determined delay between requests may make the front-end server idle with no packet arrivals. Therefore, these solutions are not sufficiently efficient for current web applications. In addition, a common approach for datacenter congestion control has involved extending what we call *end-to-end* congestion control algorithms originally designed for use on the Internet. These rely on congestion signals provided in responses to outgoing packets and typically make decisions for each flow independently. There have been variations on this approach for the datacenter context, such as DCTCP [4], CONGA [5], and Presto [28]. Most importantly, these schemes refine congestion signals (often with switch support) and refine the adjustments to flow sending rates to accommodate the high-bandwidth, low-latency datacenter environment. Sometimes, as in CONGA [5], they run the congestion control protocol between end-switches instead of between end-hosts and/or group flows together for congestion decisions. Regardless, the overall design of these approaches limits their ability to respond to congestion. Only conditions observed by a node's outgoing packets are taken into account, so decisions are made with incomplete information. To avoid this limitation, some other congestion control systems, such as Flowtune [59] and AuTo [12], use a different approach which we call *centralized*. These designs information gather information about network demands in a single 'controller' that sends congestion control decisions to each machine. Centralized approaches allow the system to plan for competing network traffic directly rather than reacting to its side effects. In addition, the amount and complexity of information the centralized controller must process make it difficult for it to make decisions quickly, which is especially important for short flows commonly found in datacenter traffic. This dissertation provides suites of solutions in application layer.

To handle the aforementioned problems, we first propose a Swarm-based Incast

Congestion Control method (*SICC*). SICC 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 the 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 latency 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.

Second, we propose a new application layer solution, called Proactive Incast Congestion Control (*PICC*) for a datacenter serving web applications. The root cause of incast congestion problem is the many-to-one concurrent communications between the single front-end server and multiple data servers [62]. 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 the gathering servers). Correlated data objects are the data objects that tend to be requested concurrently or sequentially. For example, different data objects for a web-

page 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 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.

Third, considering the drawbacks of the two kinds of congestion control algorithms introduced above, we propose a distributed congestion control algorithm *NCC, Neighbor-aware Congestion Control* algorithm based on Reinforcement Learning (RL) method. To avoid the problems of both end-to-end and centralized schemes, we propose a hybrid approach. Our 'half-distributed' scheme uses a hierarchical approach to manage network traffic. Each machine shares network information with other machines under the same switch. Combining this with local information, the machines choose a sending rate that will limit congestion overall. To limit the overhead of this decision, machines only process this information to choose an overall sending rate and do not share detailed information about each flow. Then, the machines use a privatization scheme, informed by end-to-end congestion control signals, to divide this sending rate among the pending flows. Since these decisions about individual flows are local, these decisions can be very rapid even in the presence of many short

flows. Our decisions about machine sending rates are computed using reinforcement learning, taking network statistics from neighboring machines as input. Since we do not gather information about individual flows, the reinforcement learning agents attempt to optimize a weighted combination of latency and throughput, which indicate a non-congested network.

The above discussion on congestion control problem in a datacenter leads us to the following **thesis statement**.

By exploring the congestion control model and methodologies based on spatial, temporal analysis and network topology of datacenter, considering the competition and cooperation among all the nodes, it is possible to introduce congestion control systems for congestion avoidance, latency reduction, and overall performance improvement.

We have developed a set of datacenter network systems to address the challenges mentioned above. The major contributions of this dissertation are:

(1) We design *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.

(2) We design a Proactive Incast Congestion Control (*PICC*) system, which includes three parts. *1) Popular data object gathering.* We gather the popular data objects into as few data servers as possible. 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 cluster all data objects into several groups and each group contains the data objects that tend to be requested concurrently or sequentially. We then allocate each group to the same data server. In this way, for a client's data query, the data requests issued from the front-end server have a higher 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 between the data servers and the front-end server, 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 use a query reduction algorithm to reduce the effect of the head-of-line blocking (i.e., a line of packets is held up by the first packet which increases the queuing latency). In this algorithm, the gathering server assigns a higher priority to the data objects with a smaller size and longer waiting time. Data objects with a higher priority will be transmitted out earlier than others, so that the average request latency in the gathering servers is reduced.

(3) We design *NCC* which first introduces a utility function that takes into account a combination latency and throughput. We show that, under some assumptions about network behavior, this function is approximately convex and therefore suitable for optimization via reinforcement learning (RL) techniques. We design an agent that can run each node using RL techniques to choose a sending rate that optimizes our utility function. Different from previous algorithms, the input RL algorithm replies on the network information collected among all the end-hosts under the same switch. We then propose an end-to-end priority flow scheduling method that considers size, flow remaining processing time, and flow waiting time. In this way, head-of-line blocking effect can be mitigated and the average flow completion time can be reduced.

Chapter 2

Related Work

2.1 Link Layer Solutions

The quantized congestion notification (QCN) method [3] 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 packages 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. The drawback of QCN lies in the fact that it runs on special switch, which is costly and hard to implement in practice. Devkota *et al.* [21, 36] modified QCN by sampling each package regardless of the occurrence of the congestion in order to get better performance in avoiding congestion. They found that if every packet is sampled, the performance is much better even though sampling every packet might not be necessary. Zhang *et al.* [92] improved QCN by distinguishing each flow based on their sending rates and adjusting the feedback to each flow accordingly. Huang *et al.* [31] found that controlling the TCP packets size can reduce the congestion possibility. They slice the TCP packet into smaller size to avoid congestion by special COTS switches.

2.2 Transport Layer Solutions

These transport layer solutions are mainly focused on improving TCP protocols. Vasudevan *et al.* [78] proposed disabling the delayed ACK mechanism and conducts a retransmission when the retransmission timeout is reached. When the retransmission timeout is reached, the sender assumes that a loss has occurred before it receives the acknowledgment and retransmits packets again. Disabling the ACK mechanism can achieve higher throughput than that with the ACK mechanism. ICTCP [84] 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 [4, 75] and MPTCP [83] reduce the window size by a flexible ratio according to current network status, such as the round trip delay and package loss rate. Multipath TCP [29, 58] 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 to improve the utilization of link bandwidth and avoid congestion.

2.3 Application Layer Solutions

In [63, 64, 88], a short delay is introduced between two consecutive requests by manually scheduling the second response with an extra short delay. Therefore, the number of concurrently connected data servers is reduced to avoid incast congestion. In [88], the authors proposed inserting one unit time delay between two consecutive requests. The methods in [63, 64] ask 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, the added extra delay will increase the response latency of data requests, which may not satisfy the user low-latency requirement on the web applications.

2.4 End-to-End Congestion Control Solutions

In this kind of congestion control algorithms, the traffic rate controller are deployed on each end-host, switch or both. The distributed controller makes the traffic decisions according to the network status on itself. IA-TCP [33] avoids the congestion especially incast congestion by limiting the sending rate of TCP acknowledgment (ACK) on the receiver side. FlowBender [37] proposes two techniques to manage network traffic. First, it controls the sending rate of each flow to avoid the excessive packet reordering. Second, it uses dynamic flow path reassignment to avoid congestion paths. Presto [28] splits large flows into equal-sized smaller flows on each end-host to make the flow transmission rate decision on fine-grained level. ExpressPass [16] controls the flow of credit packets at the switches and end-hosts according to the network bottleneck. The credit of packets is that a sender is allowed to send a new packet only when the receiver gives a credit to it. Thus, the speed of credit given to the sender can control the sending rate of the sender. When a congestion happens, less number of credit will be sent to the senders and then avoid the congestion. CONGA [5] deploys congestion aware agent on each end-host and switch. It splits TCP flows into flowlets, estimates the real time congestion on fabric paths and allocates flowlets to paths based on feedback from the receivers. Expeditus [81] deploys local information collector on each switch by monitoring its egress and ingress link loads. It then makes path selection decisions to avoid the congestion path. Datacenter TCP (DCTCP) [4] made ECN based congestion control algorithm the default standard in data centers. In DCTCP, a simple active queue management scheme uses a single parameter, the

queue occupancy threshold, to specify the threshold for marking packets with the Congestion Experienced code point. D2TCP [76] builds upon DCTCP to prioritize flows based on deadlines. TCP bolt [73] uses flow-level congestion control via ECN to address priority based flow control methods' limitations. ICTCP [84] iteratively adjusts the TCP receive window before incast-induced packet drops. Like TIMELY, all these TCP variants incur several RTTs (i.e., tens or hundreds) to converge to the appropriate sending rate. TCP Vegas [11] is a pioneering work in first targeting to achieve high bandwidth and low network delay. Its key insight is to use end host measurements of packet RTT as a signal to incipient congestion. MPTCP [67] splits a TCP flow into many sub-flows that may be routed independently along different paths. DeTail [89] exploits cross-layer information to reduce packet drops, prioritize latency-sensitive flows, and evenly distribute network load, effectively reducing the long tail of flow completion times.

2.5 Centralized Congestion Control Solutions

TDMA (Time Division Multiple Access) [79] divides time into multiple rounds when the centralized controller collects all the end-hosts' network information. Within each round, the time is further divided into fixed sized time slots so that each end-host can communicate to the centralized controller in a contention-less manner. Finally, all the information are processed in the centralized controller and the controller generates flow scheduling plan and send it back to each end-host. FastPass [60] sets different priorities to different size of packets to achieve high network utilization and low queuing latency. Smaller size of packets have the higher priority to be scheduled earlier. Meanwhile, FastPass sets different path for different packets with the global overview of the whole datacenter. Flowtune [59] uses a centralized controller that it first collects the information of each flow including start and end time from the

sender and receiver. The controller then calculates an optimal transmission rate and other congestion control parameter settings within a certain period. In AuTo [12], the authors first study if the deep reinforcement learning can satisfy the online control problems and make the decision for each flow in a short time. They then use a centralized controller to collect the information of all the flows in the datacenter network. In this way, AuTo can achieve near-optimal sending rate with global view while introducing extra latency for network information transmission between centralized controller and each node in the datacenter.

Chapter 3

Swarm-based Incast Congestion Control

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 [23]. The datacenter serving these applications usually support tremendous workloads. For example, Facebook serves a billion reads per second [56]. 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 [39]. The typical data request latency inside a storage system of Yahoo is on larger than 100ms [19] 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 [2, 54, 87]. 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.

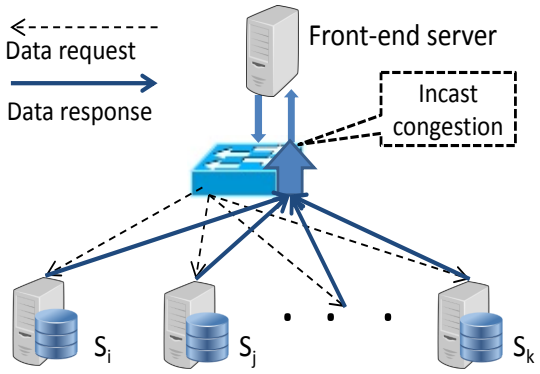


Figure 3.1: An example of incast congestion.

In Web applications, a data request for a Web page presentation needs to retrieve thousands of data objects currently [45, 56]. As shown in the Figure 3.1, 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 [49, 56].

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 [4, 55, 67, 75, 78, 83, 84, 90] 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 [41, 62, 65] 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 [46, 47]. The third group [63, 64, 88] 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 [50, 56, 70] 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 chapter, 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.

3.1 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.

3.1.1 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 3.2), 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.

For a data request, *SICC* forms the target data servers to swarms with each swarm

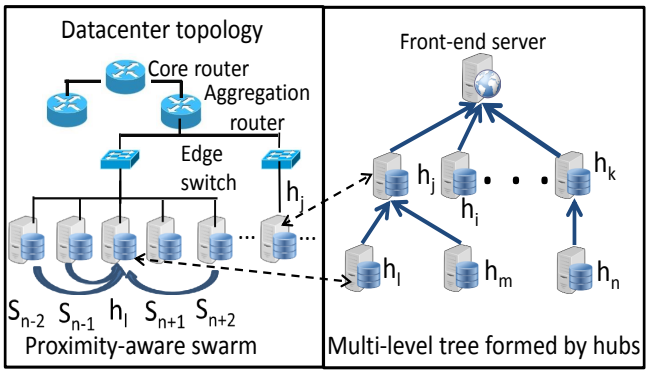


Figure 3.2: The multilevel tree with proximity-aware swarms.

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 3.2.6.

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

$$M = \frac{\frac{S_e}{B_d} * B_u}{\overline{s}}. \tag{3.1}$$

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 3.2, 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

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;
```

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).

3.1.2 Two-Level Data Transmission Speed Control

3.1.2.1 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:

$$b^a_{h_i} = \frac{B_p}{|H|}. \tag{3.2}$$

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:

$$b^a_{h_i} = b^a_{h_i} + \frac{\sum_{h_j \in H^u}(b^a_{h_j} - b^r_{h_j})}{|H^o|}, \tag{3.3}$$

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

$$b^a_{h_i} = b^r_{h_i}. \tag{3.4}$$

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.

3.1.2.2 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 3.2), 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 (3.2) 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:

$$B_p = B_p * (1 - \beta * \frac{q_{t_i}}{S_a}), \tag{3.5}$$

where β is the upper bound of the decrement of the bandwidth. We used a sliding window [61] 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 (3.2).

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

$$B_p = \min\{B_d, B_p * (1 + \alpha * \frac{B_a}{B_p})\}, \tag{3.6}$$

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 $\frac{B_a}{B_p}$.

3.1.3 Packet Compression and Object Query Redirection

3.1.3.1 Packet Compression

The data object is usually very small and no larger than 1KB [56], 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.

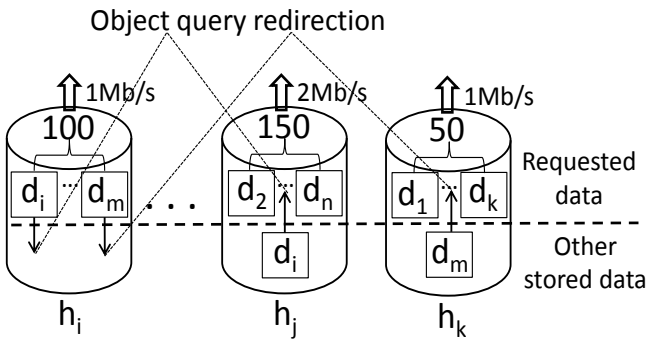


Figure 3.3: An example of query redirection.

3.1.3.2 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 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:

$$p_{h_i} = \frac{|D_{h_i}| * \overline{s}}{b^r_{h_i}}, \tag{3.7}$$

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 [6, 71]. 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 3.3, 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

$$\bar{p} = \frac{\sum_{h_i \in H} p_{h_i}}{|H|}. \tag{3.8}$$

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.

3.2 Performance Evaluation

We simulated 3000 data servers [14] in a datacenter, which forms a typical three-layer fat-tree [2] with 60 data servers inside a rack [10]. 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 [17] and 100KB, respectively. We assume a 1:4 over-subscription ratio at the ToR tier.We set the default number of requested data

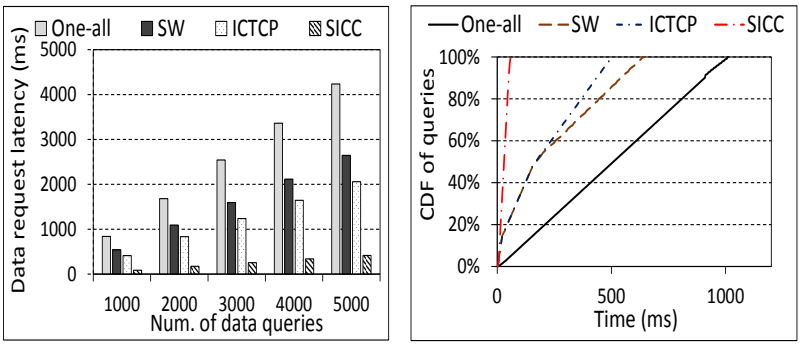
objects of a data request to 1000 [56]. Each data object has three replicas [6] randomly distributed among all data servers [72]. We set the size of each packet to a value randomly chosen from [20, 1000]B [56]. The timeout of TCP packet retransmission was set to 10 ms [57]. As [51, 84, 91], we first 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*) [56], and *ICTCP* [84].

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) [56] 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 [84] 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.



(a) Data request latency (b) CDF of queries

Figure 3.4: Performance of response latency.

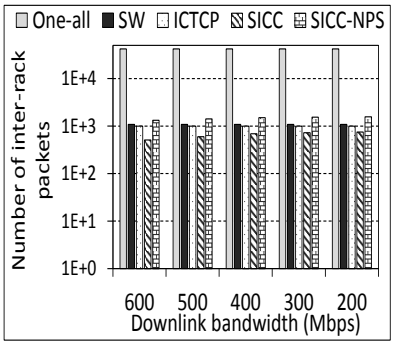


Figure 3.5: Inter-rack traffic cost reduction

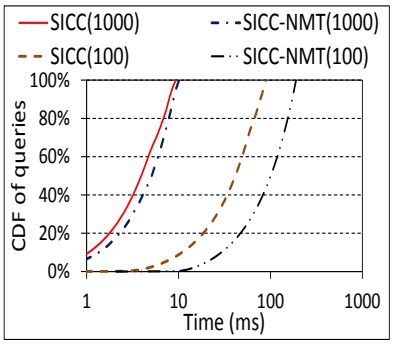


Figure 3.6: Effectiveness of swarm-based multi-level tree.

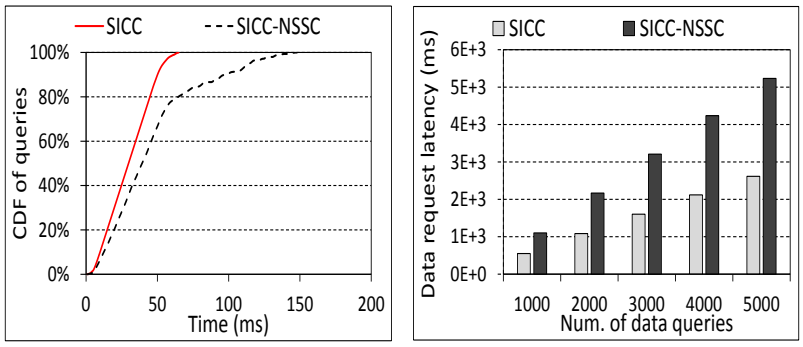
3.2.1 Performance of Data Request Latency

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 3.4(a) shows the data request latency of different methods versus the number of data queries. Figure 3.4(b) shows the CDF of data queries over time of one data request. From both figures, we see that the data request latency follows $SICC < ICTCP < SW < One\text{-}all$. 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 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 3.4(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.

3.2.2 Performance in Reducing Inter-Rack Traffic

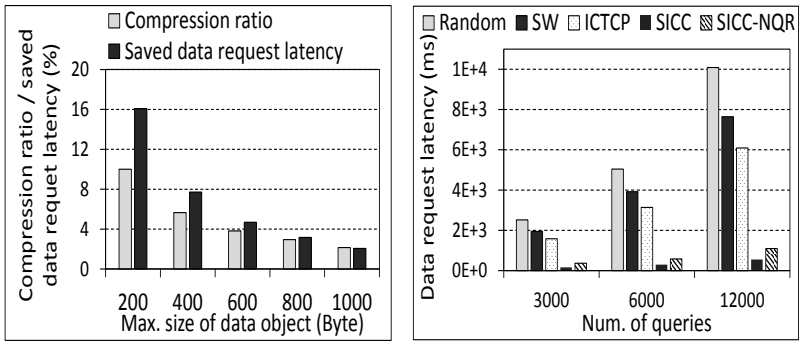
Inter-rack communication usually has a higher latency than intra-rack communication. Also, the network resources of inter-rack communication are highly required since the resources are shared by many servers under different racks. The bandwidth of links of an aggregation router is much smaller than the total downlink bandwidth of all data servers connecting to this router. Therefore, it is necessary to reduce the number of inter-rack packets. Figure 3.5 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



(a) CDF of queries of a single front-end server

(b) Data request latency with multi-front-end servers

Figure 3.7: Effectiveness of two-level speed control.



(a) Packet compression.

(b) Query redirection

Figure 3.8: Effectiveness of the enhancement methods

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.

3.2.3 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 3.6 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.

3.2.4 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 data transmission speed. Figure 3.7(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 3.7(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.

3.2.5 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 3.8(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 3.2.4. We use *SICC-NQR* to denote *SICC* without the Query Redirection method (QR). Figure 3.8(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 3.4(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.

3.2.6 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 3.9 shows the data request latency of all different 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

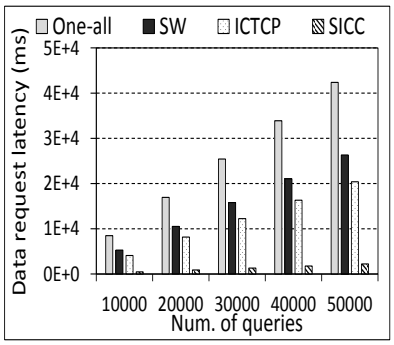


Figure 3.9: Performance of scalability.

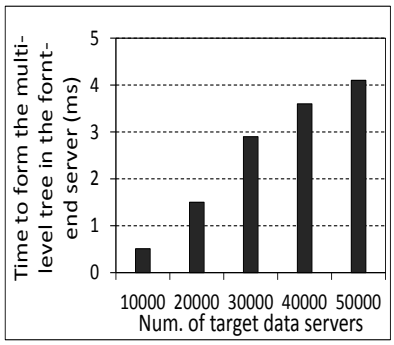


Figure 3.10: Computing time for tree creation.

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 3.4(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 3.10 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 [19].

Chapter 4

Proactive Incast Congestion Control

Previous incast problem solutions usually consider the data transmission between the data servers and the front-end server directly. They cannot proactively avoid the situation that a large number of data servers are concurrently connected to the front-end server with transient transmission of only a few data objects from each data server. To handle this problem, in this paper, we propose a new application layer solution, called Proactive Incast Congestion Control (*PICC*) for a datacenter serving web applications. Unlike the previous solutions, it handles the incast congestion problem from a completely different perspective. Based on the historical log of data object request activities in a time period, *PICC* finds popular data objects (i.e., frequently requested data objects) and the data objects that tend to be requested concurrently or sequentially (called correlated data objects). For example, different data objects for a webpage are usually requested concurrently, and a webpage indexed by another webpage may be requested sequentially. *PICC* then stores popular data objects into as few data servers as possible, and also stores correlated data objects in the same data server. We call these data servers gathering servers. In addition,

in order to reduce the queuing latency in gathering servers, *PICC* assigns different 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 reduces the number of data servers concurrently connected to the front-end server, which reduces the probability of the incast congestion occurrence. Also, it avoids the establishment of the connections between the data servers and the front-end server, which reduces the data response time to the client. Within our knowledge, *PICC* is the first work that focuses on the data placement to proactively avoid the incast congestion problem. *PICC* can cooperate with previous solutions in handling the incast congestion problem. We summarize our contribution below: *1,Popular data object gathering.* We gather the popular data objects into as few data servers as possible. 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 cluster all data objects into several groups and each group contains the data objects that tend to be requested concurrently or sequentially. We then allocate each group to the same data server. In this way, for a client's data query, the data requests issued from the front-end server have a higher 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 between the data servers and the front-end server, 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 use a query reduction algorithm to reduce the effect of the head-of-line blocking (i.e., a line of packets is held up by the first packet which increases the queuing latency). In this algorithm, the gathering server assigns a higher priority to the data objects with a smaller size and longer waiting time. Data objects with a higher priority will be transmitted out earlier than others, so that the average request latency in the gathering servers is reduced.

4.1 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. Considering the root cause of the incast congestion, reducing the number of servers concurrently connected to the front-end server can avoid the incast congestion. Therefore, 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. As a result, the front-end server only needs to communicate with a few servers for data objects to complete a client's data query, which helps avoid incast congestion. Specifically, *PICC* has three components as presented below.

In current datacenter design, all data objects are randomly stored in data servers without any optimization. When the front-end server receives a query from a client, it sends out multiple data object requests to and receive 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. In the following, we introduce each component of *PICC* in detail.

4.1.1 Popular Data Object Gathering

In the following, we explain how to find popular data objects, how to determine the gathering servers to store the popular data objects, and finally how to store popular data objects to the gathering servers.

The popular data objects are identified based on the data object request historical log. We use T to denote a time period and use $t \in T$ to denote a time-slot. Upon receiving a client's request, 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 each rack in descending order of their requesting frequencies. It then selects θ data objects on the top of the sorted list to transfer to the gathering servers in the same rack. Next, the front-end server notifies the data servers storing the top θ data objects to transfer these data objects to the gathering servers in the same rack. The log will be updated by the front-end server periodically. When the log is updated, the front-end server sorts and notifies data servers to transfer popular data to gathering servers using the above steps. In this way, the popular data list are dynamically updated and popular data objects are stored in several gathering servers.

In addition, we also need to select the gathering server(s) from all data servers in each rack. For the purpose of minimizing the total size of data objects transferred from other data nodes to the gathering server(s), we need to select a data server which

stores the largest size of data objects requested by the front-end server. 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. To limit the transfer distance of popular data objects in order to constrain the overhead, we select gathering server(s) in each track, so that popular data objects only need to transfer within a rack. Specifically, the front-end server sorts data servers in each rack in descending order of server weights. It then selects the first data server (with the largest weight) in the list as the gathering server of each rack. The popular data objects will be transferred to this gathering server until its storage, computing or I/O bandwidth capacity is fully utilized. Then, the second server in the sorted list will be selected as a gathering server to host other popular data objects in the rack. This process continues until all popular data objects are gathered into gathering servers.

In this way, we have only a few gathering servers in each rack to store the popular data objects (with higher requesting frequencies). Since popular data objects are stored in these gathering servers, the weights of these servers maintain high in their rack. Also, as more and more popular data objects move to these gathering servers, their weights will be raised. As a result, the gathering servers are unlikely to be changed once they have been selected at the first time.

After the log generation and gathering server selection, we transfer popular data objects from their original data servers to the gathering servers based upon the log. At the end of the first time period T, for each rack, the front-end server sorts all the data objects in this rack in descending order of their requesting frequencies (F_o). Here, we set threshold θ to decide how many data objects on top of each sorted list can be set as popular data objects. The front-end server then transfers the θ number

of data objects on top of the sorted list to the first selected gathering server. If the first selected gathering server does not have enough storage, computing or I/O bandwidth capacity to host all the θ data objects, another gathering server will be selected from the top of the sorted list of the data servers to host other data objects. This process repeats until all popular data objects are hosted in gathering servers. Later on, after each time period T, the log for data object requesting frequencies will be updated. Then, if a gathering server stores data objects not in the top θ popular data objects, the front-end server notifies the gathering server to transfer these data objects to other data servers in the same rack. For each data object in the top θ popular data objects, the front-end server checks whether it is stored in a gathering server. If not, it is transferred from its current data server to the nearest gathering server that has sufficient storage, computing and I/O bandwidth capacities for it. If there is no existing gathering server that has sufficient capacities for the data object, a new gathering server will be selected from the sorted data server list to host this data object. As a result, the popular data objects in a rack are always gathered in only a few gathering servers, which limits the number of servers concurrently connected to the front-server and hence helps avoid incast congestion.

Figure 4.1 shows an example of the popular data object gathering method. In this figure, each red rectangular represents one popular data object. 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 and packet loss on the front-end server side.

In our proposed popular data object gathering method, the front-end server reallocates popular data objects into only a few gathering servers in each rack. In this

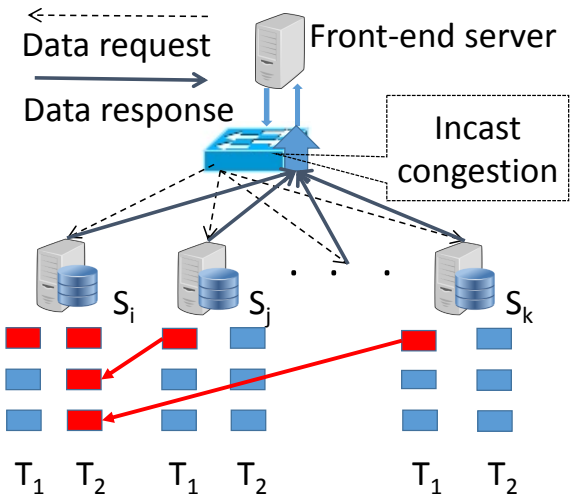


Figure 4.1: Popular data object gathering.

way, when the front-end server processes query requests from the clients, it requests and receives data objects mainly from the gathering servers. In Figure 4.1, 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 front-end server receives data objects from one data server rather than three data servers, the probability of incast congestion occurrence is reduced.

4.1.2 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 addition, if we allocate data objects that are likely to be requested concurrently or sequentially into the same gathering server, we can further reduce the number of servers concurrently responding to the front-end

server. 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. We call such data objects correlated data objects. Thus, we propose the correlated data object gathering method to cluster the concurrently or sequentially requested data objects (i.e., correlated data objects) into the same group and allocate the group of correlated data objects into the same gathering server.

If we gather sequentially requested data objects into the same data server, then the sequential requests are generated one by one from the front-end server to the same data server. It reduces the data transmission latency caused by the connection rebuilding delay between the front-end server and data servers [34]. 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 in the same time and the connection lifetime will be longer than the previous method. As a result, data objects can be 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, x and y, are requested concurrently during each time-slot t, denoted by $P_t(x, y)$. It can also derive the frequency that two data objects, x and y, are requested sequentially during each time-slot t, denoted by $Q_t(x, y)$. Then, the closeness of x and y for time period T is calculated by:

$$C_T(x,y) = \alpha * \sum_{t \in T} P_t(x,y) + \beta * \sum_{t \in T} Q_t(x,y) + (1-\alpha-\beta) C_{T-1}(x,y). \tag{4.1}$$

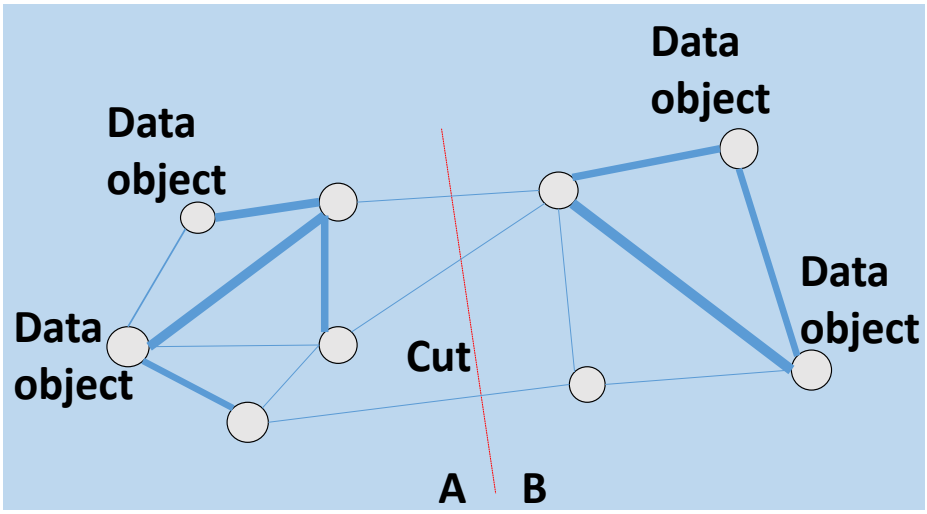


Figure 4.2: Correlated data object clustering.

Here, α and β are the weights for the concurrent request frequency and sequential request frequency. $C_{T-1}(x, y)$ is the closeness of x and y for the previous time period $T-1$. The inclusion of $(1-\alpha-\beta)C_{T-1}(x, y)$ 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 a graph in which each vertex is a data object and uses the minimum cut tree based algorithm [24] to divide the graph vertices to clusters. The data objects in each cluster are the correlated data objects. In the following, we introduce how to build the graph and how to divide the graph to clusters.

The front-end server generates an undirected graph $G(V, E)$, where V is the set of all data objects and E is the set of all edges connecting data objects. The weight of each edge connecting data objects x and y is their current closeness $C_T(x, y)$. We then use the minimum cut tree based algorithm to divide the vertices in the entire graph to subsets in order to create data object clusters. As shown in Figure 4.2, a *cut* divides all data objects V in graph G into two data object subsets A and B. The value of a cut equals the sum of the weights of the edges crossing the cut. The minimum cut tree algorithm creates clusters that have small sum of inter-cluster cut

value and relatively large sum of intra-cluster cut values.

Algorithm 2 shows the pseudo-code of the data object clustering algorithm. We construct a minimum cut tree so that we can get the minimum sum of the values of cuts. Finally, the graph G has been divided into several clusters. The algorithm returns all the 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. The minimum cut algorithm requires a computation complexity of $O(|V| \cdot |E| log(|V|^2/|E|))$ [27].

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.

Algorithm 2: The correlated data object clustering algorithm.

```
1 V' = V ∪ s; Graph generation Link each data object v to generate graph
    G'(V', E');
2 for all nodes v ∈ V do
3     Link V with the weight w;
4 Generate the minimum cut tree T' of G' [26];
5 Divide G into clusters;
6 Return the clusters of G;
```

4.1.3 Queuing Delay Reduction

A gathering server stores a large number of popular data objects, which are requested frequently with a large number of data transmissions by the front-end server. A gathering server also stores correlated data objects that tend to be concurrently or

sequentially requested by the front-end server. The 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 can reduce the effect of head-of-line blocking by setting different priorities for the data objects based on their sizes and waiting times.

We propose a queuing delay reduction algorithm to reschedule the data objects in the queue. That is, a data object with a smaller data size and longer waiting time has a higher priority to be transmitted first. We first present an example to demonstrate how this algorithm works. Figure 4.3 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 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 reschedule 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 optimized queue achieves about $49/g$ less latency than the unoptimized 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. To make the algorithm light-weighted and also constrain the waiting time of data objects, we only schedule a number (e.g., ten) of objects at the beginning of the queue based on the first-in-first-out (FIFO) rule rather than all data objects in the sending queue of the

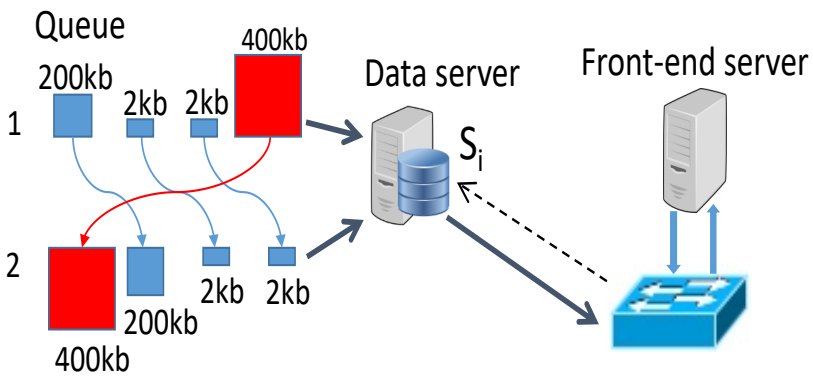


Figure 4.3: An example of queuing order optimization.

gathering server.

We consider the size and the waiting time when determining the priority of a data object in the queue. According to [42, 43], the transmission latency and the queuing latency is in microsecond scale. For a data object o, we use B_o to denote its size and use τ_o to denote its waiting time in the queue. We then can calculate the priority value of data object o, denoted by M_o, by:

$$M_o = \tau_o^3 / B_o \tag{4.2}$$

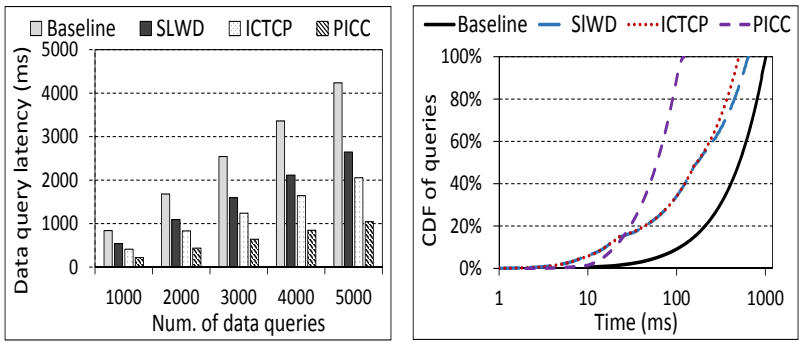
In order to place more weight on waiting time when determining the priority of data objects, we triple the value of τ. This exponent can be set to another value depending on how much weight the system wants to give to the waiting time. The data objects in the queue will be re-ordered based on their priority values. Considering that the queue length of the list is not large, this scheduling method can run fast and uses little computing resource. Thus, the overhead of the proposed queuing delay reduction algorithm is trivial.

4.2 Performance Evaluation in Simulation

To build the datacenter, we constructed a typical fat-tree [53] using 4000 data servers with 60 data servers inside each rack [66]. There is only one front-end server in this

structure which is randomly selected from all the data servers. In the simulation, we set the capacity of the downlink, uplink and buffer size of each edge-switch to 10Gbps, 10Gbps and 100kb, respectively. For each data object, it has three replicas randomly distributed on three different data servers. In order to simulate the web service applications better, the size of each data object was randomly chosen from [20B,1024B] [56]. Unless otherwise specified, we set the threshold for the number of data objects in the top of the sorted list that are considered as popular data objects θ=1000. There are 20200 data objects in the datacenter; 200 data objects are popular data objects and 20000 data objects are regular data objects. Each data query consists 1000 data requests for data objects. For each data request, it requests one data object from the popular data object list with 50% probability and from the regular data object list with 50% probability. The interval between two consecutive data queries was randomly chosen from 0.05s to 0.5s [48]. The timeout of TCP packet retransmission was set to 10ms. 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 we measured the average performance of each request after the front-end server receives all the queried data objects [84, 91].

We compared the performance of *PICC* with three other representative methods: *Baseline*, *SLDW* [56], and *ICTCP* [84]. 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. After the targeted data server receive the requests, they start the data object transmissions to the front-end server. *SLDW* is a sliding window method, in which the front-end server can adjust the number of its concurrently connected servers



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

Figure 4.4: Performance of data query latency.

using the classical sliding window protocol. The window size will increase one by one until the front-end server detects the incast congestion occurrence. Once the incast congestion occurs, the window size will decrease to half of the previous window size. *ICTCP* [84] adjusts the sliding window size by detecting the bandwidth utilization. It divides all the slots into two part. The first part can be used to receive all the traffic and predict the bandwidth utilization. It then adjusts the window size in the second part based upon the bandwidth calculation results in the first part in order to fully utilize the available bandwidth without over-utilizing the bandwidth capacity.

In the following sections, we will measure the performance of *PICC* with all of its methods and then measure the effectiveness of each method in *PICC*.

4.2.1 Performance of Query Latency

Each data query consists of multiple data queries for different data objects. The request latency for a data object is the time period between the time a front-end server sends the query to the data server and the time it receives the request data object. The longest time among all the requests of a query is the latency of this query. Figure 4.4(a) shows the data query latency of different methods versus the number of data queries. Figure 4.4(b) shows the Cumulative Distribution Function (CDF)

of data queries versus the data query latency. From both of these two figures, we see the query latency results follow *PICC*<*ICTCP*<*SLDW*<*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. The sliding window protocol *SLDW* 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). That is, the window size decreases to half of the previous window size so that it can avoid the incast congestion by reducing the bandwidth utilization. Therefore, *SLDW* achieves better performance than *Baseline* in terms of data query latency. However, *SLDW* produces a longer request latency than that of *ICTCP*. *SLDW* 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. As a result, *SLDW* generates longer query latency than *ICTCP*. *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 between request sending and response receiving for a new connection establishment.

Figure 4.4(a) also shows that the data query latency of all methods increases proportional to the number of data queries. More queries mean that more data objects need to be transmitted from each data server, which generates a longer data transmission time. These figures indicate that *PICC* generates the shortest data

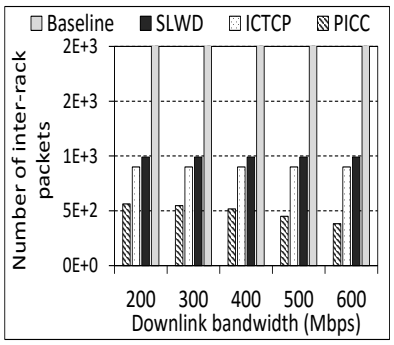


Figure 4.5: Inter-rack data transmissions

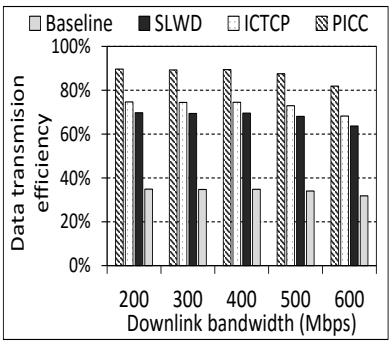
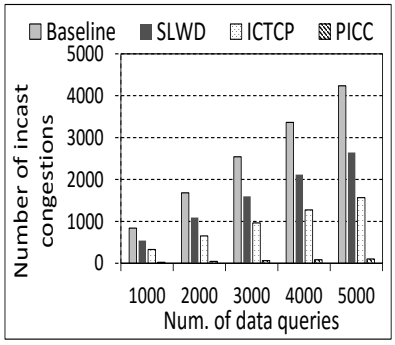
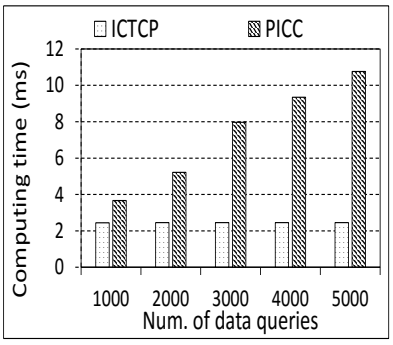


Figure 4.6: Data transmission efficiency



(a) Number of incast congestions



(b) Algorithm computing time

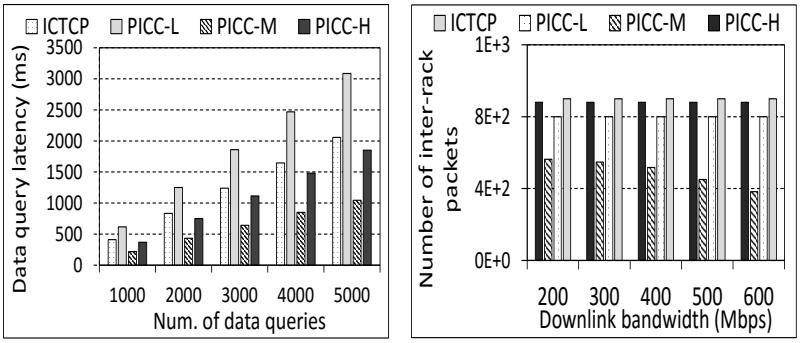
Figure 4.7: Incast congestion avoidance and computing time.

query latency among all methods by proactively avoiding incast congestion and fully utilizing the downlink bandwidth of the front-end server.

4.2.2 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. In this experiment, we measure the number of inter-rack packets versus the different downlink bandwidth of the front-end server.

A smaller number of inter-rack packets leaves more bandwidth for the connection



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

Figure 4.8: Performance of different θ threshold setting.

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 smaller 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. Figure 4.5 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*<*SLDW*<*Baseline*. *Baseline* has the largest number of inter-rack packets compared with other three methods since it has the highest probability of generating incast congestion without any solutions to avoid the incast congestion. For *SLDW*, all the data servers respond to the front-end server directly 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. Without the high probability of incast congestion occurrence, *ICTCP* generates fewer data object retransmissions so that it reduces the number of inter-rack packets compared with *SLDW*. 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.

This figure indicates that *PICC* generates the smallest number of inter-rack packets since it reduces the incast congestion occurrences and data retransmissions compared with other comparison methods.

We then measure the data transmission efficiency by $\frac{size}{latency}/BW$, where $size$ is the total size of all the requested data, *latency* is actual query latency of all the requested data, and BW is the downlink bandwidth of the front-end server. Figure 4.6 shows our measured data transmission efficiency of the four methods versus different downlink bandwidths of the front-end server. It shows that the data transmission efficiency results follow *PICC*>*ICTCP*>*SLDW*>*Baseline*. Furthermore, the result of *PICC* increases with the bandwidth downlink decreasing, 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. *SLWD* with the sliding window protocol achieves better performance than *Baseline*. In *ICTCP*, the bandwidth occupies as least 50% and it adjusts the other part of bandwidth based upon the bandwidth utilization, which leads to higher bandwidth utilization than *SLWD* and higher data transmission efficiency. *PICC* transfers popular and correlated data objects into a limited number of servers, and the front-end server has a higher probability to request data objects from the a few 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.

Figure 4.7(a) shows the number of incast congestions occurred. We see that *SLDW* 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 number of data queries increases, the number of incast congestions of these four methods grows since more data queries lead to a

higher possibility of incast congestion occurrence.

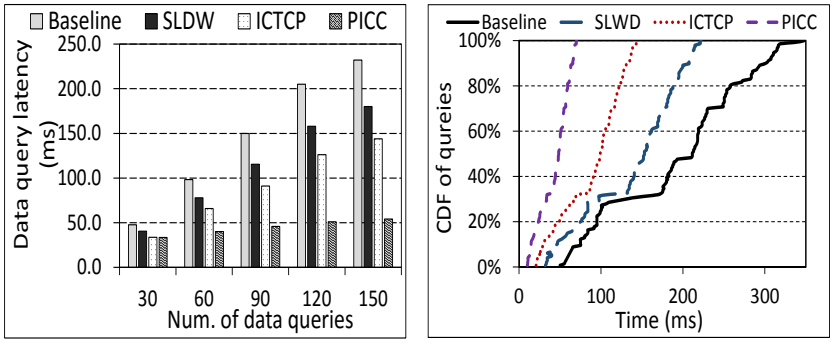
Figure 4.7(b) shows the computing time of *PICC* and *ICTCP* for the data reallocation scheduling and window size adjustment calculation, respectively. The computing time of *Baseline* is 0 and the computing time of *SLDW* is negligible. The results show that the computing time of *PICC* is higher than that of *ICTCP*. Also, as the number of data queries 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 50000 data queries, the computing time is only 11ms, which is very small compared with the entire data transmission latency reduction in Figure 4.4. In summary, compared with *ICTCP*, *PICC* greatly reduces the number of incast congestions and produces reasonable computing time compared with the entire query latency.

4.2.3 Performance of the Popular Data Object Gathering Method

We then measure the performance of the popular data object gathering method with different θ threshold settings that determine the number of popular data objects. Since *Baseline* and *SLDW* always have worse performance than *ICTCP*, in this experiment, we only compare *ICTCP* with *PICC*. We use *Low*, *Medium* and *High* to denote the three cases when the θ threshold to identify popular data equals 10, 1000 and 10000, respectively. We use *PICC-L*, *PICC-M* and *PICC-H* to denote *PICC* with these three thresholds, respectively. We intended to compare the performance with different θ threshold settings. Figure 4.8(a) shows the data query latency versus the number of data queries. It shows that *PICC-M* produces the lowest query latency compared with *PICC-L*, *PICC-H* and *ICTCP*. In *ICTCP*, there may be many data servers concurrently connected to the front-end and each data server transmits only a

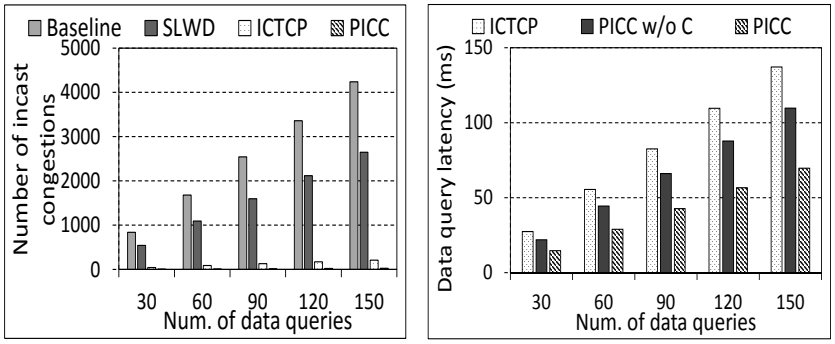
few data packets, which leads to incast congestion. Also, many establishments of the connections between data servers and the front-end server generates extra latency. As a result, *ICTCP* generates the longest latency. *PICC* gathers popular and correlated data objects into a limited number of data servers. Thus, the number of data servers concurrently connected to the front-end server is limited and also the number of establishments of new connections is reduced, which greatly reduce the query latency. *PICC-L* sets a low θ threshold, then only a few data objects can be transferred to the gathering servers. The reallocation of only a few data objects will not greatly affect the number of data servers concurrently connected to the front-end server and hence the query latency. On the other hand, *PICC-H* sets a high θ threshold. Then, more data objects transferred to a gathering server lead to congestion in the gathering server, 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. Therefore, an appropriate setting for the θ threshold is important to achieve high performance in reducing data query latency.

Figure 4.8(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, *ICTCP* has the largest number of inter-rack packets, and both *PICC-H* and *PICC-L* have larger numbers of inter-rack packets than *PICC-M*. Too many or too few popular data object transferred to the gathering servers cannot greatly reduce the incast congestions and the data retransmissions increase data query latency. To sum up, this figure again indicates that an appropriate θ threshold setting is important to avoid incast congestion and data retransmissions.



(a) Data query latency

(b) CDF of data query latency



(c) Number of incast congestions

(d) Effectiveness of individual methods

Figure 4.9: Performance on a real cluster.

4.3 Performance on A Real Testbed

We implemented the *PICC* and other comparison methods on a high performance supercomputer. 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 Linux 64-bit 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 150 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.

Figure 4.9(a) shows the query latency of all methods versus the number of queries.

Figure 4.9(b) shows the CDF of queries over time of all incast congestion control methods. They show the same order and relationship as in Figure 4.4 in the simulation due to the same reasons. Both of the figures indicate that *PICC* has the best performance in query latency.

Figure 4.9(c) shows the number of incast congestions versus the number of data queries. It shows the same order and trend due to the same reasons as in Figure 4.7(a). The figure indicates that *PICC* can greatly reduce the number of incast congestions.

Figure 4.9(d) shows the data query latency of *ICTCP*, *PICC* and *PICC w/o C* versus the number of data queries. *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.

Chapter 5

Neighbor-aware Congestion Control

A common approach for datacenter congestion control has involved extending what we call *end-to-end* congestion control algorithms originally designed for use on the Internet. These rely on congestion signals provided in responses to outgoing packets and typically make decisions for each flow independently. There have been variations on this approach for the datacenter context, such as DCTCP [4], CONGA [5], and Presto [28]. Most importantly, these schemes refine congestion signals (often with switch support) and refine the adjustments to flow sending rates to accommodate the high-bandwidth, low-latency datacenter environment. Sometimes, as in CONGA [5], they run the congestion control protocol between end-switches instead of between end-hosts and/or group flows together for congestion decisions. Regardless, the overall design of these approaches limits their ability to respond to congestion. Only conditions observed by a node's outgoing packets are taken into account, so decisions are made with incomplete information.

To avoid this limitation, some other congestion control systems, such as Flowtune [59] and AuTo [12], use a different approach which we call *centralized*. These

designs information gather information about network demands in a single 'controller' that sends congestion control decisions to each machine. Centralized approaches allow the system to plan for competing network traffic directly rather than reacting to its side effects. But this approach is problematic in terms of reactiveness and resource consumption. Minimally, decisions must be delayed by the amount of time required to communicate with this central controller. In addition, the amount and complexity of information the centralized controller must process make it difficult for it to make decisions quickly, which is especially important for short flows commonly found in datacenter traffic.

To avoid the problems of both end-to-end and centralized schemes, we propose a hybrid approach. Our 'half-distributed' scheme uses a hierarchical approach to manage network traffic. Each machine shares network information with other machines under the same switch. Combining this with local information, the machines choose a sending rate that will limit congestion overall. To limit the overhead of this decision, machines only process this information to choose an overall sending rate and do not share detailed information about each flow. Then, the machines use a privatization scheme, informed by end-to-end congestion control signals, to divide this sending rate among the pending flows. Since these decisions about individual flows are local, these decisions can be very rapid even in the presence of many short flows.

In this paper, we introduce evaluate a particular manifestation of this design focused on optimizing mean flow competition time. Our decisions about machine sending rates are computed using reinforcement learning, taking network statistics from neighboring machines as input. Since we do not gather information about individual flows, the reinforcement learning agents attempt to optimize a weighted combination of latency and throughput, which indicate a non-congested network. The contributions of this paper include:

1. We introduce a utility function that takes into account a combination latency and throughput. We show that, under some assumptions about network behavior, this function is approximately convex and therefore suitable for optimization via reinforcement learning (RL) techniques.

2. We design an agent that can run each node using RL techniques to choose a sending rate that optimizes our utility function. Different from previous algorithms, the input RL algorithm replies on the network information collected among all the end-hosts under the same switch.

3. We then propose an end-to-end priority flow scheduling method that considers size, flow remaining processing time, and flow waiting time. In this way, head-of-line blocking effect can be mitigated and the average flow completion time can be reduced.

4. We conduct comprehensive experiments in real implementation. The results show that NCC outperforms other comparison methods in the average flow completion time as well as other measures of congestion.

5.1 NCC System Structure

5.1.1 Overview

Our system enforces an outgoing rate-limit on each node, where the rate limit is determined by processing the TCP statistics from the current local node and the neighboring nodes under the same switch. Under a simplified network model, we show that by controlling these rate limits, it is possible to optimize a throughput/latency tradeoff by controlling these rate limits. Since setting rate limits on each node does not determine the order of outgoing traffic, we propose a flow priority scheduling scheme that avoids head-of-line blocking effects without delaying long flows excessively.

5.1.2 Agents and Information Sharing

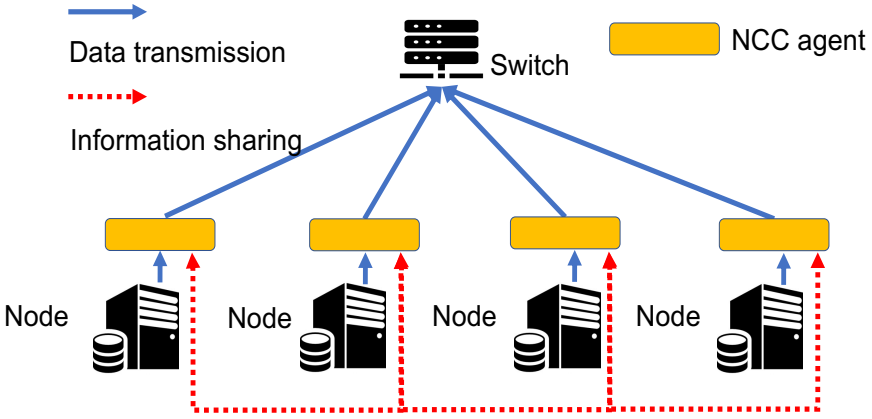


Figure 5.1: The connection between neighbor nodes.

We deploy an agent on each end-host to collect outgoing flow information (such as statistics from a TCP stack) and make outgoing rate decisions. The agent also communicates with other neighbor nodes under the same switch as shown in Figure 5.1 and controls the outgoing rate limit by prioritizing flows as described in section 5.3.3. To make sense of all these data, we use the reinforcement learning (RL) to make rate limiting decisions based on the information collected from local and neighboring nodes. The RL makes the decisions for outgoing rate limiting not for the order of the flows. After deciding the overall rate, we use a privatization method to determine the order of flows on each node. The RL method continuously makes decisions based on environmental feedback (e.g., measurements from the TCP stack). The agent runs all the time and collects the information from neighbor nodes periodically. Since many flow transmission times are short, we set the information collection period to 1ms by default. Since we train the RL offline, it can quickly make its decision based on the collected information. Thus, RL is suitable for the congestion control decision making process.

The NCC structure on each node is shown in Figure 5.2. NCC is deployed on each node including a monitoring system, reinforcement learning algorithm, and rate

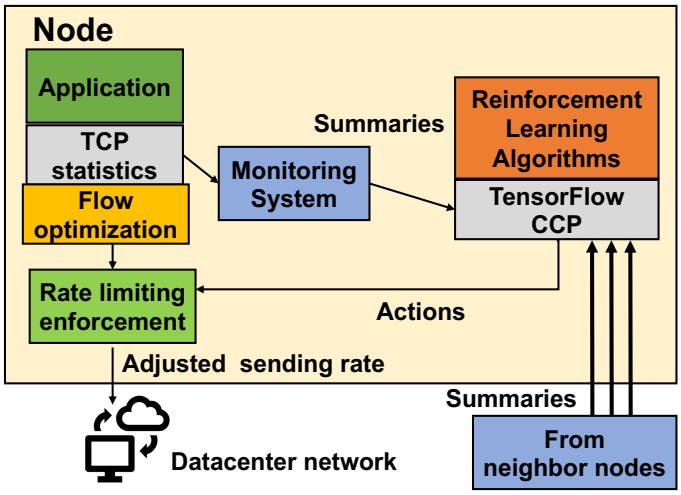


Figure 5.2: The overview of NCC structure.

limiting enforcement. The monitoring system, based on CCP platform [18], monitors the TCP related parameters and then transfers those parameters as state to the reinforcement learning algorithm based on TensorFlow [1]. Meanwhile, the state information and the actions taken by other nodes are retrieved from other neighbor nodes. The trained RL model makes the outgoing rate limiting decision and sends that decision to the rate limiting enforcement. The RL algorithm is constructed based on TensorFlow and CCP [18] (a platform for congestion control design) and the rate limiting enforcement decides which packets to send after they are queued by the TCP stack. After the rate limit on the node is determined, the flows in the queues will be transferred out according to the priorities of flows.

5.2 Utility Function

In order to choose transmission rates for each node and control the overall load of the network, we use reinforcement learning to measure network conditions and try to optimize a hybrid of overall throughput and latency for each node. We compute each

node n's utility as:

$$U_n = \ln x_n - \rho \ln d_n \tag{5.1}$$

where x_n represents the achieved throughput of each node and d_n the achieved average latency within a measurement period. ρ is a weight which determines how much our system favors latency over throughput. We discuss the choice of ρ in our evaluation in section

Nodes measure the actual x_n and d_n via locally gathered statistics from the TCP stack about the current good throughput and recent round-trip times. When packet loss is negligible, x_n will approximately equal the overall sending rate chosen by each node.

When selecting the overall transmission rates, our agents try to optimize *global* utility, which we define as the sum of the utilities of each node (in a group of neighboring nodes):

$$\sum_{n \in N} U_n(x_n) > \sum_{n \in N} U_n(x'_n) \tag{5.2}$$

where x'_n is another value of sending rate for n^{th} node, for all the N senders and all the non-negative x value.

To determine the sending rate on each node, nodes share network information about their observed latencies and throughputs with neighboring nodes. Since the mean waiting time in the queue of the bottleneck can be measured directly, the global utility $\sum U_n$ is a approximately convex function according to Equation 5.1, showing that is suitable for the local optimization techniques used by our reinforcement learning agents.

Suppose the arrival rate in the bottleneck queue is $\sum_z x_z$ which is a Poisson process and the bottleneck packet service times have an exponential distribution [7]. Therefore, the mean waiting time in the queue of the bottleneck is $W_b = \frac{1}{C - \sum_z x_z}$ where C

is the bottleneck bandwidth constraint. Thus, Equation (5.1) can be rewritten as:

$$U_n = \ln x_n + \rho \ln \left(C - x_n - \sum_{z \neq n} x_z \right). \tag{5.3}$$

In order to get the maximum of objective utilization function, we set the partial derivative $\frac{\partial \sum_i U_i}{\partial x_n}$ to 0 for each n and then get:

$$0 = \frac{\partial \sum_{i=0}^{N} U_i}{\partial x_n} = \frac{1}{x_n} + \rho \frac{1}{C - x_n - \sum_{z \neq n} x_z} \tag{5.4}$$

$$+ \sum_{i \neq n} \rho \frac{1}{C - x_n - \sum_{i \neq n} x_i} \tag{5.5}$$

Consolidating the terms we have:

$$0 = \frac{\partial \sum U_i}{\partial x_n} = \frac{1}{x_n} + \frac{N * \rho}{C - x_n - \sum_{z \neq n} x_z}. \tag{5.6}$$

$$x_n = \frac{C - \sum_{z \neq n} x_z}{1 - N * \rho}. \tag{5.7}$$

Since the second derivative of $\sum U_i$ is:

$$\frac{\partial^2 \sum_{i=0}^{N} U_i}{\partial (x_n)^2} = -\frac{1}{(x_n)^2} - N * \rho \frac{1}{(C - x_n - \sum_{z \neq n} x_z)^2} < 0 \tag{5.8}$$

$$\frac{\partial^2 \sum_{n \in N} U_n}{\partial (x_n)^2} < 0 \tag{5.9}$$

which shows that the sum of utility function for all the nodes is a convex function. Since only x_n can be varied by the agent, we use the partial derivative over x_n to prove that U_n is a approximately convex function for variable x_n. This proof suggests a closed form for the x_n that maximizes the utility function, but this is based on the

queuing model. In practice, network bursts and variance will cause some deviation from the queuing model solution, so we do not try to encode this solution in the reinforcement learning agents.

Although the deviation from the M/M/1 queuing ideal prevents the use of this analytic solution, we hypothesize that the utility function will remain approximately convex. The actual mean queue waiting time will still tend to increase as the load increases. With limited queue capacity, queue waiting times will not increase towards infinity, but will become unacceptable well before typical queue capacities are reached.

5.3 Design of Reinforcement Learning in NCC

Since the state space (the network related statistics) and the action space (the outgoing rates) are continuous, the traditional RL method with Q-learning cannot handle this optimization problem well. Since the network condition is more variable and complex, we use Actor-Critic (A3C) algorithm, which has policy gradient for continuous action space and also better convergence performance in Deep-RL [40]. Different from the original RL algorithm, with Equation (5.7), the RL agent on each node can, using information shared from other nodes, select the outgoing sending rate to achieve global optimality.

A3C [40], a state-of-art deep RL algorithm, has been applied to many network problems such as video streaming optimization [52]. To apply to network optimization problems, A3C can be trained offline on historical network conditions. These network conditions include the time-series states information will be introduced in detail in Section 5.3.1.2.

At each time epoch e, the RL agent first monitors local network conditions and receives that information from other nodes to formulate the state S. The agent then takes actions A according to the policy and receives a reward R. The policy above

is that the agent aims to map state information to a deterministic action or to a probability distribution to the actions which uses a accumulated reward calculated by reward function. While training, the agent tries to update the policy to maximize the accumulated reward for actions. A3C relies on Deep Q-learning Neural Network (DQNN) as the function approximation for the reward of an action. The DQNN takes the state-action pair (s, a) as input and outputs the corresponding Q value $Q(s, a)$ which is the expected discounted cumulative reward:

$$Q(s, a) = E[R|(s, a)] \tag{5.10}$$

where R means the reward function. The policy stored in DQNN can be achieved by:

$$\pi(s) = argmaxQ(s, a). \tag{5.11}$$

In the offline training of the DQNN, the target is that each state-action pair can be derived the related value by Bellman equation which is:

$$V = r(s, t) + \gamma Q(s, \pi(s)|\theta^Q) \tag{5.12}$$

where V is the target value need to be update and θ^Q is the parameter of DQNN. According to V, DQNN can be trained via loss function minimization which is:

$$L = E[V - Q(s, a|\theta^Q)]. \tag{5.13}$$

Since Equation (5.18) is non-linear function which is not suitable for the typical DQNN, in order to improve the stability of Q-learning, *experience replay* and *target network* are introduced. The RL agent collects and stores the state information into a replay buffer and then updates DQNN with the information stored in replay buffer

rather than using the information collected immediately. Therefore, the agent can observe network conditions and learn from the past experience which is better to build a non-biased DQNN. The DQNN's parameters will be slowly updated which makes the learning process smooth and avoids biased results. Furthermore, A3C is proposed with two functions. The actor function is used for deriving actions and the critic function is used for evaluating actions. Both of the functions are stored in two separate DQNNs. The critic function will be updated according to Equation (5.13) and the actor function is updated according to another equation which relies on cumulative reward J with the actor's parameters θ^{π}:

$$\begin{aligned} \nabla_{\theta^{\pi}} J &\approx E[\nabla_{\theta^{\pi}} Q(s, a|\theta^{\pi})] \\ &= E[\nabla_a Q(s, a|\theta^{\pi}) * \nabla_{\theta^{\pi}} \pi(s|\theta^{\pi})] \end{aligned} \tag{5.14}$$

Finally, the two DQNN are built and fully updated according to the equations above.

5.3.1 Reinforcement learning Setting

5.3.1.1 Action Space

The action space A is defined as the possible sending rates that can be allocated on one end-host. Suppose we divide the link between into d parts, the action space A is:

$$A = \{a_0, ..., a_d\}. \tag{5.15}$$

During each time period, the RL agent will select one action to execute.

5.3.1.2 State Space

The state space $\mathcal{S}$ consists of the information collects from the node itself and all the neighbor nodes which are related to each flow. It consists several parts below. For

the n^{th} node itself, it consists the sending rate b_n, the round trip time $d_{b,n}$, the actual sending rate or the good throughput g_n is the value measured in the last time period, the total flow waiting time in the queue $d_{q,n}$, the size of all the flows in the queue $s_{q,n}$ and the number of flows in the queue $f_{q,n}$. For each neighbor node, it consists the sending rate, the average flow waiting time and the length of queue. For the upper node, it consists sending rate, the length of queue. The state space can be represented as:

$$\mathcal{S} = \{S_1, S_2, ..., S_n, ...S_{n+1}\} \tag{5.16}$$

$$S_i = \{b_i, d_{b,n}, g_n, d_{q,n}, s_{q,n}, f_{q,n}\} \tag{5.17}$$

For each end-host, it receives all the states belong to all the neighbor nodes. Thus, all the end-hosts will receive the same states information except the information belongs to the end-host itself.

5.3.1.3 Reward

The reward function is defined over past sequences of actions and observations. When the agent chooses an action, and gets an observation, it receives a reward that is a function of the observation, all previous observations (e.g., the weight of all previous observations can be controlled by the RL itself) and the actions. In our method, the reward will be calculated after the action selected by the agent and all the related measured parameters are ready.

The reward function used is defined as two parts: 1) *Profit* which focuses on maximizing the total profit for all nodes under the same switch and then aiming to achieve the global optimality according to Equation (5.6). The utility function and the reward function are equivalent and then RL can use the reward function directly to optimize the utility as shown in section 5.3.1.4. We want to use utility function

as the reward function to achieve the tradeoff between latency and throughput for all nodes. 2) *Penalty* function includes the real world constraints (e.g., bottleneck bandwidth, number of cooperation nodes, and other nodes' network performance) which can reflect different network situation settings. Meanwhile, the penalty function considers the maximum delay within neighbor nodes to avoid or mitigate the possible decisions which may introduce extra delay to other nodes.

The overall reward function is therefore

$$R = Profit - Penalty. \tag{5.18}$$

At time t, *Profit* is calculated by:

$$Profit = \sum_{i=0}^{N} \frac{b_i^t}{g_i} \ln b_i^t - \rho_i \ln d_i^{t-1} \tag{5.19}$$

where, for each node i, b_i^t is the action determined by the agent at time t and g_i is the actual sending rate in the last time period of the RL running. d_i^{t-1} means the average packet round trip time measured in the last time period. The values which belong to other nodes will be transferred from other nodes to this node with the agent.

The Penalty terms takes into account network constraints. Using this, we provide direct feedback about bandwidth constraints and an extra penalty to avoid situations where one node experiences very high latency compared to a typical node. The penalty can be represented as:

$$\begin{aligned} Penalty = \sigma_1 * \frac{\sum_{z \neq n} g_z}{C - b_n} * b_n \\ + \sigma_2 * \left(\frac{Max_{z \neq n}(d_z)}{C_d} - 1 \right) d_n \end{aligned} \tag{5.20}$$

where σ_1 and σ_2 are the coefficient to optimize the penalty which can give different

weight to throughput or delay and C_d is a standard delay threshold for the network, determined from historical data or Service Level Agreement (SLA). For the first term which considers the bandwidth constraints, if $\frac{\sum_{z \neq n} g_z}{C - b_n} > 1$, then the current decided sending rate is higher than the available bandwidth in the bottleneck. Then, the first term in the penalty is inflated and larger than that of $\frac{\sum_{z \neq n} g_z}{C - b_n} < 1$. For the second term which considers the negative effect of the high latency, we use the standard delay threshold C_d which is set manually according to the engineer's experience to evaluate the latency situation. If the largest delay among these nodes larger than the threshold, all the agents will be punished. It is because all the agents' actions can affect the high latency under the same switch.

5.3.1.4 The Equivalence of Utility Function and Reward Function

Let $\mathcal{A}$ be the set of actions an agent can take, and $\mathcal{O}$ the set of observations. Assume both sets are finite (we can set discrete values for the actions and the observations). Let $\mathcal{H}$ be the set of histories (sequence of observations and actions) of an agent. Let $\mathcal{W}$ be the set of worlds which include the full set of observation history for the agent. Therefore, a reward function R is a function from histories to real numbers, while the utility function U is a function from worlds to real numbers. We then assume that the agent knows it can take ω actions and get ω observations. The $W_h | h \in \mathcal{H}_\omega$ form a partition of the possible worlds in $\mathcal{W}$. We then assume that reward function is given as R, for $w_h \in \mathcal{W}_h$:

$$U_R(w_h) = \sum_{i=1}^{\omega} R(h_i), \tag{5.21}$$

and then for history h, $V(U, \pi, h)$ and $V(R_U, \pi, h)$ differ by a constant that is a function of h only, not because of policy π. Thus, the reward function maximization and the utility function maximization will choose the same policies. In another words,

when the reward function and the utility function are the same, the policies selected from the RL agent or the decision made according to the utility function are the same aiming to maximize the utility function then.

5.3.2 Monitoring System

The RL system takes all the network information from the monitoring system as the input. Since the metrics measured in CCP programming model are not enough for our system, we deploy our own monitoring system written in Python that collects necessary information every 1ms. $d_{b,n}$ is important in reward function to evaluate the delay within the datacenter network. A timer is implemented and send a statistic to the monitoring system. The timer variable starts at 0 and represents the number of microseconds since last ACK arrived. However, it is hard to get this number directly when the system monitors the packets. The monitoring system collects all the time stamps including the start time of one packet starting to send and the end time of the corresponded ACK received by the server. For the good throughput g_n, the monitoring system first check the size of all the flow completed in the last period and also the loss rate. It then computes the good throughput for each flow and sums the throughput of all the flows to generate g_n. The other flow-related metrics are collected via *flow* class in CCP [18]. All the metrics above are also captured and validated by tcpdump [74] and WireShark [82].

5.3.3 Flow Prioritization

After the reinforcement learning determines the rate at which each node may drain their queue of packets, the simplest way to prioritize between the different flows would be FIFO or SJF. However, FIFO would introduce head-of-line blocking and SJF would degrade the performance on flow completion time by starving long flows.

In order to avoid this problem, we further consider the remaining flow completion time to optimize the order of all the flows in the queue within each end-host. In this way, the average flow completion time can be reduced.

For a flow l, we use τ_l to denote its waiting time in the queue, measured from when it was added to the queue to when it started to be sent out. The size of flow l is denoted as s_l and $\mathcal{R}_l$ is the flow completion remaining time which can be estimated by remaining flow size over the current throughput. We then can calculate the priority coefficient Υ_l of this flow l by:

$$\Upsilon_l = \frac{{\tau_l}^{\omega}}{s_l * \mathcal{R}_l} \tag{5.22}$$

where the value of ω can decide the weight of waiting time when determining the priority coefficient of each flow since sometimes the waiting time needs more weights to determine the priority coefficient compared with the flow size. Since the average flow completion time of short flows is around several millisecond, in order to reduce the overhead of the priority optimization method, we only select top κ flows to set priorities and re-order them.

Next we introduce the priority optimization process with pseudo-code shown in Algorithm 3.

Algorithm 3: Priority flow optimization algorithm.

1 **For** all the flows waiting to send out;
2 **Sub-queue Generation** top κ flows to generate a sub-queue of the flow queue in the end host;
3 **For** each flow l in this sub-queue;
4 ***Calculate*** *the priority coefficient* Υ_l *according to Equation (5.22);*
5 ***Sort*** *the flows in the sub-queue according to* Υ_l*;*
6 ***Send out*** *the flows in the order of the sub-queue under the rate limits determined by RL;*
7 ***Remove*** *those* κ *flows from the queue in the end host;*

After the priority of each flow is determined, the flow sending behavior is controlled by modifying *rate* class in CCP via command *update-fields* which is much faster than

regular setting directly. The rate enforcement will select flow one by one until the rate enforced is reached. In order to avoid overuse of bandwidth, once the rate limit for the node is almost occupied and the next first priority is larger, the rate enforcement waits until some flow's completion before adding more flows with high priorities.

5.4 Evaluation

5.4.1 Experiment Settings

We implement our *NCC* using Python based on *CCP* programming model [18] and Ubuntu 18.04 LTS. The reason why we implement our methods through *CCP* is that it makes it easy to program sophisticated algorithms in a safe user-space environment as opposed to writing C and then the risk of crashing Linux kernel. Meanwhile, using Python makes *NCC* compatible with the current machine learning programming model like TensorFlow used in this paper.

In the simulation, we use ns-3 network simulation tool to evaluate the performance of *NCC* with Fat tree topology. Since the default version of TCP in ns-3 is NewReno which is out of dated, we adopt the newest TCP BBR based on [9] in ns-3 simulation. There are 16 servers in each rack, 128 Top-of-Rack (ToR) switches, and 32 aggregation switches. We use the labmade C++ code to realize the rate control part and realize the rate limiting mechanism with the same RL model. The link capacity between servers and ToR switches is 10Gbps. The link capacity between each switch is 40Gbps. When we change the number of concurrent senders, we will further increase the workload in the same scale. For example, we double the number of concurrent senders while doubling the workload in the two benchmarks. To sum up, the rate control mechanism and decision making process are the same both in simulation and real implementation.

In the implementation, we deploy *NCC* on 48 Amazon EC2 instances *m4.10xlarge*

(40 cores CPU, 160 GB memory, and 10Gbps NIC) as end hosts and another 10 instances *m4.16xlarge* (64 cores CPU, 256 GB memory, and 25Gbps NIC) simulated as switches. The instances have sufficient resources to ensure that CPU and memory will not be the bottleneck in these workloads.

5.4.2 Workload Generation

We use Hibench [32] to generate two types of workload: data analysis and web search to simulate the most popular workload in the current datacenters [8]. In data analysis workload which consists more longer flows, we use *sort, wordcount, terasort, enhanced DFSIO* to simulate daily data analysis behavior in the datacenter. In web search workload which consists more shorter flows, we use *pagerank, nutch indexing* to simulate web search scenario in the datacenter.

For the RL training in *NCC*, in order to avoid the data bias, we use another network data in prior work [68] to train the model for 10 rounds. After the training, we use the trained RL agent to do the test below. The data has been open-sourced in "Facebook Network Analytics Data Sharing" Facebook group [22] which is the network data collected during a 24-hour period in January 2015. The clear majority of traffic is intra-cluster but not intra-rack (i.e., the 12.9% of traffic that stays within a rack is not counted in the 57.5% of traffic labeled as intra-cluster). Since the RL training from scratch can result in poor policies at the beginning of learning and long time to converge, we adopt the offline supervised learning to guide the RL policy update with the existing scheduling strategy, the default TCP-BBR sending rate mechanism. The online training of the RL agent is trained with Adam optimizer [38] with a fixed learning rate of 0.001 for offline supervised learning and 0.0001 for the online training. With these setting, the online training will spend around 8 hours to

converge after the offline learning.

We also use Yahoo Cloud Serving Benchmark (YCSB) [86] to measure the performance under web serving workload performance. We use the default settings (e.g., readproportion, updateproportion) in YCSB. We set another physical machine as the client and evenly distribute all the data files in all the nodes. We implement MongoDB on all the nodes to host all the requested data files. For different number of concurrent senders, we adjust the number of operations in the workload settings to make sure that lager number of senders have larger workload.

5.4.3 Comparison Methods and Test bed Topology

We took four comparison methods both in simulation and real world implementation.

1. DCTCP: In DCTCP [4], it provides a TCP-like protocol for datacenter network. It leverages the Explicit Congestion Notification (ECN) in the network to provide multi-bit feedback to each end host. It adjusts the sending window size according to the extent of congestion. For each packet in DCTCP, the packets are marked to convey congestion signal according to instantaneous queue length upon their arrival at the queue. We set the ECN marking threshold to be 30KB as DCTCP recommends. 2. CONGA: CONGA [5] splits TCP flows into flowlets and estimates real time congestion on fabric paths. It then allocates those flowlets to paths based on feedback from switches. 3. Flowtune: Flowtune [59] makes the congestion control decision for each flowlet not a packet. In Flowtune, they implement a centralized allocator which receives the flowlet start and end notifications from endpoints. The allocator then computes the optimal rates using network utility maximization and updates the related congestion control parameters. 4. Auto: Auto [12] controls all the flows in a centralized way. A centralized controller collects all the necessary information including the size of each flow and the number of finished flow in a unit

time within the datacenter network. The controller then makes the decisions relying on the deep reinforcement learning for all the flows to determine the priority and the sending rate of each flow within each node.

5.4.4 Metrics

To measure the performance of congestion control methods, we primarily focus on the average timeout ratio, a single metric that relates to congestion control performance and reflects whether we are avoid the congestion and provide much better throughput and latency performance. Besides average performance, a frequent concern of congestion control systems is poor performance in exceptional cases. We examine how our system fares in several metrics that reflect this problem, including the flow completion time, 99th percentile flow completion time, Jain's fairness index and the CDF of all the flows for each node's achieved performance. The measured metrics answer the questions below. In addition to evaluating performance in terms of congestion, we are also interested in our system's sensitivity to its parameters and how well it achieves its internal utility goal.

5.4.5 Overall Performance

We first show the results in the real implementation.

Figure 5.3 and 5.4 show the timeout ratio for all the flows generated from all the servers versus the number of concurrent senders. TCP timeout ratio is an abnormal signal that one packet may be lost and this packet needs to be re-transmitted or the connection is failed. In another word, the TCP timeout ratio can refplect how many percentage of packets suffer congestion and even packet lost. The results follow *NCC* $<$*Auto*$<$*CONGA*$<$*Flowtune*$<$*DCTCP*. *DCTCP* optimizes the original TCP parameters specially for datacenter networks. Based on *DCTCP*, *CONGA* splits each long

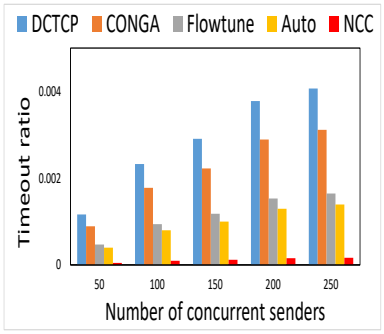


Figure 5.3: Timeout ratio for HiBench.

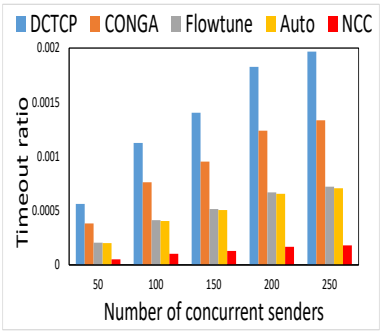


Figure 5.4: Timeout ratio for YCSB.

flows into small flowlets which can achieve better performance in packets loss avoidance since it optimize each flowlet in more fine-grain level. Thus, the performance of *CONGA* is better than *DCTCP*. *Flowtune* optimized the sending rate of each flowlet same to *CONGA*. Because of the distributed way of *CONGA*, *Flowtune* uses a centralized rate allocator for each flowlet with network utility maximization strategy which can achieve the global optimal leading to *Flowtune*$<$*CONGA*. In *Auto*, as a centralized controller, it also needs the network information of all the flows in the nodes within the datacenter. The large size of network information and also the RL model introduces extra decision making and transmission latency. The centralized controller introduced extra decision making latency which can highly affect the performance. Meanwhile, the extra latency leads to decision delay so that the centralized controller cannot make the sending rate decision in real time. The nodes sharing the bottleneck link suffer the congestion so that the decision transmission latency will be much larger than others. In *NCC*, without centralized controller, *NCC* uses a "hybrid-distributed" method which introduces the network information transmission between all the neighbor nodes. Different from the previous selfish players' congestion control game, the cooperative players' game made it possible to get the global optimal while avoiding the extra latency introduced by centralized controller since the transmission latency between each neighbor nodes is much smaller than that in

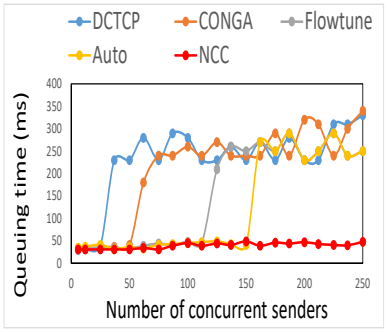


Figure 5.5: Queuing time for HiBench.

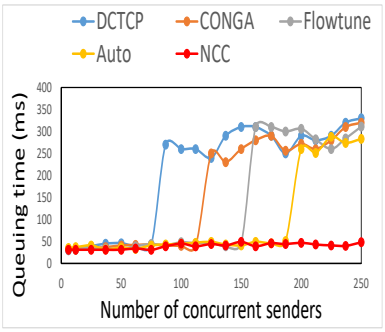


Figure 5.6: Queuing time for YCSB.

centralized methods. Furthermore, *NCC* takes advantage of RL which can response to the variable network conditions fast and also based on historical data. Therefore, *NCC* can achieve the best performance compared with other comparison methods and almost zero timeout ratio in the experiment duration.

Figure 5.5 and 5.6 show the average queuing time of all the flows versus the number of concurrent senders. We change the number of concurrent senders aiming to test the scalability holding large number of servers at the same time. The results follow that *NCC*>*Auto*>*CONGA*>*Flowtune*>*DCTCP*. Without any optimization about cooperation between each node, *DCTCP* achieves the worst performance with much lower throughput because of high query time in the bottleneck link. *CONGA* splits the flow into flowlets and determines the sending rate on each node. *CONGA* adjusts the sending rate in fine-grain level so that it can response to the high query time in the bottleneck fast with larger number of servers around 400 servers. *Flowtune* optimizes the sending rate of the flowlets with centralized controller and calculates all the optimal sending rate for all the flowlets. Therefore, *CONGA* achieves worse performance than *Flowtune*. With the flow-level optimization, *Auto* can achieve better performance than the above comparison methods. However, the centralized controller limits the decision making latency performance. *NCC* considers the congestion control algorithm as a cooperative game and takes the advantage of centralized methods

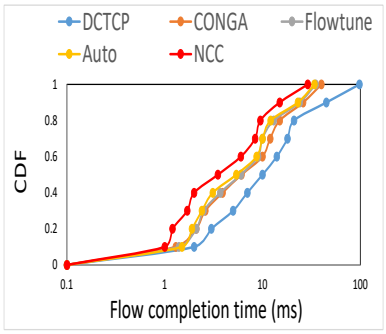


Figure 5.7: Flow completion time for HiBench.

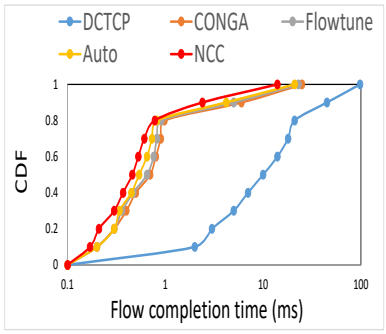


Figure 5.8: Flow completion time for YCSB.

and also lower latency caused by network information transmission between each neighbor node. The sending rate allocation among all the nodes lower the possibility of congestion and longer query in bottleneck. *NCC* achieve lower query time all the time since the query time is occupied in each node.

Figure 5.7 and 5.8 show the CDF versus the flow completion time (ms). All the results follow that $NCC < Auto < CONGA < Flowtune < DCTCP$ due to the same reason in Figure 5.3. From these two figures we also know that *NCC* can always achieve the best performance under these two different network topologies with high scalability because of the "hybrid-distributed" way. It has the network topology aware and only considers the neighbor nodes network conditions which have much higher effect on the bottleneck link. All the nodes with the RL agent can communicate with the other neighbor nodes without the topology sensitive.

Figure 5.9 and 5.10 show the CDF versus the 99% flow completion time. For HiBench workload, it consists a mixture of both short and long flows in Figure 5.9. *NCC* outperforms the other comparison methods and achieves up to 20% improvement for average flow completion time. This is because that *NCC* can achieve the global optimal network condition without introducing extra information communication latency. Considering the enhancements, *NCC* can dynamically optimize the priority of all the flows which avoids the "head-of-black" effect. For YCSB workload, it consists mostly

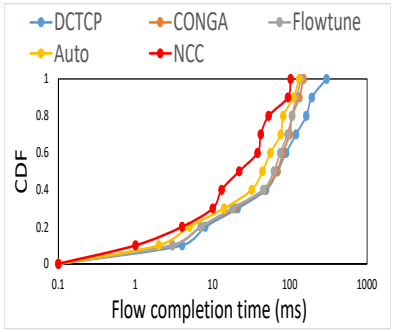


Figure 5.9: 99% of Flow completion time for HiBench.

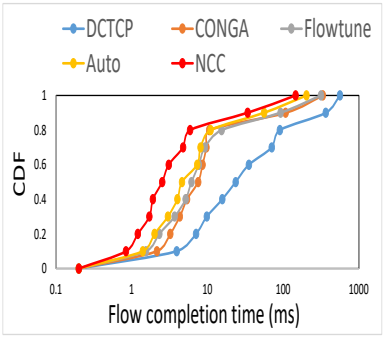


Figure 5.10: 99% of Flow completion time for YCSB.

short flows. *NCC* has the similar performance compared with *CONGA*, *Auto* and *Flowtune*.

Chapter 6

Summary

In this dissertation, we first introduced the background of the datacenter network and also the root cause of network congestion, especially for the incast congestion. The three proposed datacenter network congestion control systems outperform other comparison methods in datacenter network congestion control using different network features.

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. Web applications are featured by a very large number of data object responses for a data query, which may cause the incast congestion problem and makes it difficult to meet their stringent low delay requirements. Less previous methods focus on the data placement to proactively avoid the incast congestion problem. For this purpose, we proposed a proactive incast congestion control system *PICC*. 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 show that *PICC* greatly reduces data query latency and the probability of the incast congestion occurrence. Current datacenter hosts multiple types of workloads which require low latency and high throughput concurrently. However, the previous distributed solutions cannot achieve global optimal solutions while the centralized solutions cannot make the congestion control decision in a short time. To overcome the drawbacks, we propose *NCC* which uses "half-distributed" way to build the congestion control system. it takes advantage of the low latency of communication within one switch or router. The reinforcement learning agent deployed on each end-host can exchange all the network information in real time with all the end-hosts under the same switch. Thus, it can avoid the information exchange delay in centralized method and also achieve the global opti-

mal status for all the nodes. The extensive experiment shows that *NCC* outperforms other comparison methods under different network conditions in many aspects.

For future work, the application of congestion control algorithms in network performance optimization is vast, and this dissertation is at the beginning stage. There will be a huge amount of opportunities for network performance optimization, especially for congestion control algorithms. This section introduces one research direction to explore congestion control algorithms used in datacenter networks: Data-driven congestion control methods. As the Internet grows more complex and diverse, the traditional TCP congestion control can not react to the real-time variance in the network. Since different datacenter applications have various features of network transmission, we can design a new system that aggregates the real-time performance of similar TCP connections and predicts the suitable TCP parameters for each connection to avoid network congestion. Like our third work in this direction, we can further extend it to all the parameters determination using machine learning algorithms and data analytics tools, such as window size, in TCP connections. Second, passive network measurements involve high network probing overhead and limited datacenter coverage. Thus, some special applications like video streaming offer a new chance. Such applications can provide real-time measurements of network conditions without any probing overhead. We plan to take advantage of such special applications to measure network real-time performance and support the future congestion control algorithms. After several decades of network system development, we believe that, in the future, the network systems can be improved with the increasing amount of network measured data and the development of data analytics techniques.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, and M. Isard. "Tensorflow: a system for large-scale machine learning." In: *Proc. of OSDI*. 2016.

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

[3] 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.

[4] 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.

[5] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese, et al. "CONGA: Distributed congestion-aware load balancing for datacenters". In: *ACM SIGCOMM Computer Communication Review*. Vol. 44. 4. ACM. 2014, pp. 503–514.

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

[7] V. Arun and H. Balakrishnan. "Copa: Practical delay-based congestion control for the internet". In: *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 2018, pp. 329–342.

[8] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. "Information-agnostic flow scheduling for commodity data centers". In: *Proc. of NSDI*. 2015.

[9] *BBR in ns-3*. https://github.com/mark-claypool/bbr, [accessed in Jun. 2020].

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

[11] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. *TCP Vegas: New techniques for congestion detection and avoidance*. Vol. 24. 4. ACM, 1994.

[12] L. Chen, J. Lingys, K. Chen, and F. Liu. "Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization". In: *Proc. of SIGCOMM*. 2018.

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

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

[15] P. Cheng, F. Ren, R. Shu, and C. Lin. "Catch the whole lot in an action: Rapid precise packet loss notification in data center". In: *Proc. of NSDI*. 2014.

[16] I. Cho, K. Jang, and D. Han. "Credit-scheduled delay-bounded congestion control for datacenters". In: *Proc. of SIGDC*. 2017.

[17] *Cisco Nexus 3064 Switch*. http://www.cisco.com/c/en/ us/products/switches/nexus-3064-switch/, [accessed in May 2017].

[18] *Congestion Control Plane(CCP) programming model*. https://ccp-project.github.io/guide/intro.ht [accessed in Oct. 2019].

[19] 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.

[20] 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.

[21] P. Devkota et al. "Performance of quantized congestion notification in TCP Incast scenarios of data centers". In: *Proc. of MASCOTS*. 2010.

[22] *Facebook Network Analytics Data Sharing Group*. https://research.fb.com/blog/2017/01/data-sharing-on-traffic-pattern-inside-facebooks-datacenter-network/, [accessed in Oct. 2019].

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

[24] G. W. Flake and K. Tarjan R.and Tsioutsiouliklis. "Graph clustering and minimum cut trees". In: *Internet Mathematics* (2004).

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

[26] R. E. Gomory and T. C. Hu. "Multi-terminal network flows". In: *J. SIAM* (1961).

[27] J. Hao and J. B Orlin. "A faster algorithm for finding the minimum cut in a directed graph". In: *Journal of Algorithms* (1994).

[28] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. "Presto: Edge-based load balancing for fast datacenter networks". In: *ACM SIGCOMM Computer Communication Review* 45.4 (2015), pp. 465–478.

[29] B. Hesmans and O. Bonaventure. "Tracing multipath TCP connections". In: *Proc. of SIGCOMM* (2015).

[30] 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.

[31] 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.

[32] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis". In: *Proc. of ICDEW*. 2010.

[33] J. Hwang, J. Yoo, and N. Choi. "IA-TCP: a rate based incast-avoidance algorithm for TCP in data center networks". In: *Proc. of ICC*. 2012.

[34] 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.

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

[36] A. Kabbani, M. Alizadeh, M. Yasuda, R. Pan, and B. Prabhakar. "AF-QCN: Approximate fairness with quantized congestion notification for multi-tenanted data centers". In: *Proc. of HOTI*. 2010.

[37] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene. "Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks". In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM. 2014, pp. 149–160.

[38] D. Kingma and J. Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[39] R. Kohavl and R. Longbotham. "Online Experiments: Lessons Learned". In: *Computer* (2007).

[40] V. Konda and J. Tsitsiklis. "Actor-critic algorithms". In: *Advances in neural information processing systems*. 2000.

[41] 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.

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

[43] J. Li, N. K. Sharma, D. R. Ports, and S. D Gribble. "Tales of the tail: Hardware, OS, and application-level sources of tail latency". In: *Proc. of SOCC*. 2014.

[44] 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". In: *Trans. on Networking* (2017).

[45] 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.

[46] Z. Li, H. Shen, W. B. L. III, and J. Denton. "An Exploration of Designing a Hybrid Scale-Up/Out Hadoop Architecture Based on Performance Measurements". In: *Trans. on Parallel and Distributed Systems, TPDS* (2016).

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

[48] G. Liu, H. Shen, and H. Chandler. "Selective data replication for online social networks with distributed datacenters". In: *Trans. on Parallel and Distributed Systems, TPDS* (2016).

[49] 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.

[50] G. Liu, H. Shen, and H. Wang. "Deadline guaranteed service for multi-tenant cloud storage". In: *Trans. on TPDS* (2016).

[51] G. Liu, H. Shen, and H. Wang. "Towards Long-View Computing Load Balancing in Cluster Storage Systems". In: *Trans. on Parallel and Distributed Systems, TPDS* (2016).

[52] H. Mao, R. Netravali, and M. Alizadeh. "Neural adaptive video streaming with pensieve". In: *Proc. of SIGCOMM*. 2017.

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

[54] 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.

[55] 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.

[56] 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.

[57] V. Paxson, M. Allman, J. Chu, and M. Sargent. *Computing TCP's Retransmission Timer*. Tech. rep. RFc 2988, November, 2011.

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

[59] J. Perry, H. Balakrishnan, and D. Shah. "Flowtune: Flowlet control for datacenter networks". In: *Proc. of NSDI*. 2017.

[60] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. "Fastpass: A centralized zero-queue datacenter network". In: *Proc. of SIGCOMM* (2015).

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

[62] 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.

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

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

[65] 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.

[66] 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.

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

[68] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. Snoeren. "Inside the social network's (datacenter) network". In: *Proc. of SIGCOMM*. 2015.

[69] 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.

[70] H. Shen, A. Sarker, L. Yu, and F. Deng. "Probabilistic Network-Aware Task Placement for MapReduce Scheduling". In: *Proc. of Cluster*. 2016.

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

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

[73] B. Stephens, A. L. Cox, A. Singla, J. Carter, C. Dixon, and W. Felter. "Practical DCB for improved data center networks". In: *Proc, of INFOCOM*. 2014.

[74] *tcpdump*. https://www.tcpdump.org/, [accessed in Oct. 2019].

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

[76] B. Vamanan, J. Hasan, and T. Vijaykumar. "Deadline-aware datacenter tcp (d2tcp)". In: *Proc. of SIGCOMM* (2012).

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

[78] 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.

[79] B. Vattikonda, G. Porter, A. Vahdat, and A. Snoeren. "Practical TDMA for datacenter ethernet". In: *Proc. of EuroSys*. 2012.

[80] 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.

[81] P. Wang, H. Xu, Z. Niu, D. Han, and Y. Xiong. "Expeditus: Congestion-aware load balancing in clos data center networks". In: *Proc. of SOCC*. 2016.

[82] *WireShark*. https://www.wireshark.org/ , [accessed in Oct. 2019].

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

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

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

[86] *Yahoo Cloud Serving Benchmark (YCSB)*. https://github.com/brianfrankcooper/YCSB, [accessed in Jun. 2020].

[87] 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.

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

[89] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. "DeTail: reducing the flow completion time tail in datacenter networks". In: *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM. 2012, pp. 139–150.

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

[91] J. Zhang, F. Ren, L. Tang, and C. Lin. "Taming TCP Incast throughput collapse in data center networks". In: *Proc. of ICNP*. 2013.

[92] Y. Zhang and N. Ansari. "On mitigating TCP Incast in data center networks". In: *Proc. of INFOCOM*. 2011.