# DCP: An Efficient and Distributed Data Center Cache Protocol with Fat-Tree Topology[†]

Zhihui Jiang, Zhiming Ding, Xiaofeng Gao*, Guihai Chen
Department of Computer Science and Engineering, Shanghai Jiao Tong University
{zhihuijiang730, dingzhm93}@gmail.com, {gao-xf, gchen}@cs.sjtu.edu.cn

***Abstract*—In the big data era, the principal traffic bottleneck in a data center is the communication between hosts. Moreover, the redundant packet contents indeed take a major part of the bottleneck. In this paper, we propose a distributed and efficient cache sharing protocol, named DCP, to eliminate redundant traffic in data center with Fat-Tree topology. In DCP, each switch holds packet caches, which are used for eliminating redundant packets. A cache sharing mechanism using Bloom Filter is also proposed to share local caches of switches with servers. Moreover, we propose another technique leveraging Bloom Filter to reduce false positive ratio caused by sharing cache information between switches and servers. We build up a Fat-Tree topology with $k = 16$ to evaluate the performance of DCP, and the simulation results show that DCP can eliminate $40\%$ to $60\%$ redundant packets, which validate its efficiency.**

**Index Terms–Data Center Network, Fat-Tree, Bloom Filter, Cache Protocol, Redundancy Elimination**

## I. Introduction

As the appearance of cloud services like social network, cloud storage, and cloud computing, the server-side takes more responsibility for storing and processing data. To achieve better performance of these tasks, many famous platforms are introduced, such as Google App Engine (GAE). All of them have heavy workloads and require enormous computing capacity, which lead to a large quantity of communications in the network. For example, a lot of communication among different servers is required to fetch or search for data in distributed storage systems, like GFS [1] and BigTable [2], since one server has to access local information of the remote server to proceed local search and computation. MapReduce [3] shuffles its intermediate files to distinct destination servers between its map phase and reduce phase, which costs much inter-node bandwidth. Thus, it is important to eliminate redundant traffic in a data center and increase data center's utilization ratio.

Some protocol-independent approaches are proposed to cache the duplicate contents of different applications [4]. Y. Cui *et al.* [5] proposed a mechanism of redundancy elimination in data center. They gave a linear programming with an objective function which maximizes the redundancy elimination of the data center. They also proposed an approximate centralized algorithm. However, their approach is impractical in the real data center because it needs too much memory regarding the number of packets and a central server is required to store cache information of all the switches. Researchers have done a lot of work on redundancy elimination in Internet. Internet Cache Protocol (ICP) [6] and Summary Cache [7] are two well-known web cache protocols, both of which propose efficient web cache mechanisms to remove duplicate contents in Internet and reduce transmission delay of web request. However, they are specially designed for web applications in Internet, which cannot handle packets of various protocols in a data center. Moreover, they are not suitable for data center caches because they do not leverage the specific topology and routing algorithms of a data center. Thus, it is necessary to raise a practical cache protocol for a real data center.

Fat-Tree is a scalable data center topology which has been adopted by Cisco's massively data center using commodity servers and switches to construct a large scale cluster. Correspondingly, in this paper, we propose a practical redundancy elimination protocol named DCP with Fat-Tree topology. As far as we know, we are the first to cope with such a data center cache protocol. DCP meets the following goals:

- Distributed: Caches are well-distributed on different switches which share local cache information with servers periodically so that each server obtains a global view of all cache information on switches.
- Efficient: We employ Bloom Filter to encode the cache information to achieve better efficiency of cache sharing.
- Scalable: Our approach is scalable based on the scalability of Fat-Tree and our simulation proves the results.

The rest of the paper is organized as follows. Section II presents the detailed design and implementation of DCP. Performance evaluation of DCP is presented in Section III. Section IV concludes our paper.

## II. Design of DCP Protocol

In this section, we present the detailed design of DCP protocol. DCP specifies how packets are cached in data center switches. Moreover, it defines how switches and servers communicate with each other and share cache information with each other in a Fat-Tree based data center and how to eliminate redundant traffic in the data center using the cache.

### A. Overview

Data center Cache Protocol is a specified cache protocol for data center with Fat-Tree topology [8]. Fat-Tree topology connects commodity switches and servers based on Clos network [9], and it allocates IP addresses to servers and switches within the 10.0.0.0/8 block following specific criteria. Moreover, they provide an efficient routing algorithm using two level routing table. A sample Fat-Tree topology is shown in Fig. 1.

The goal of DCP is to eliminate traffic redundancy in a data center network. Briefly speaking, DCP contains two strategies: the strategy for packet routing and caching; and the

[†]This work has been supported in part by Shanghai Educational Development Foundation (Chenguang Grant No.12CG09), Shanghai Pujiang Program 13PJ1403900, the Natural Science Foundation of Shanghai (Grant No.12ZR1445000), and the National Natural Science Foundation of China (Grant number 61202024).

*X.Gao is the corresponding author.

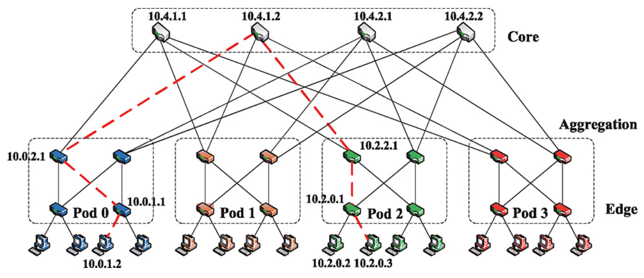


Fig. 1: A Sample Fat-Tree Topology with 4 pods

strategy for cache sharing and cache replacing. The strategy for packet routing and caching is the key part of DCP, which can reduce redundant traffic significantly. Correspondingly, when a server receives a packet request from another server, the source server firstly looks up its local cache table to check whether there exists a switch storing the cache of this packet along the determined path to the receiver. If yes, then the source server will send a compact packet with a key representing the packet instead of the original packet to the switch containing the required cache. When the compacted packet reaches this switch, the switch will check its local packet cache according to the key, extract the original data from its cache to replace the compacted one, and then continue forward the new packet to the destination. The principle is depicted in Fig. 2.

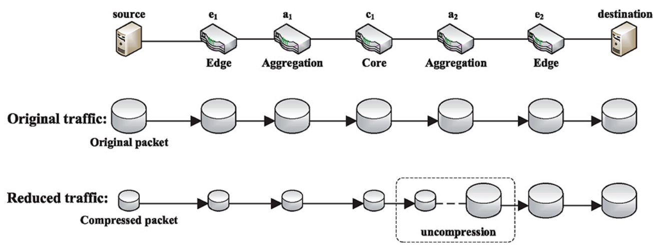


Fig. 2: Principle of Redundancy Elimination

The strategy for cache sharing and replacing is used to help the servers get the latest information and also grab the most popular packet. Firstly, because of the limited switch memory and the variety of packets, the switch can replace some caches with others to reduce the traffic. Secondly, switches will periodically share its cache information among the servers, so that each server is aware of the locality of every individual cache information thus getting a global view of caches.

The data flow of DCP is as follows: Whenever a server wants to send a TCP packet to another server, the server firstly encapsulates the TCP packet into a DCP packet. The DCP packet may be a compressed form of the TCP packet or the original TCP packet with additional information. The DCP packet is sent to the edge switch connecting to the server, and then the DCP packet is routed through several switches until reaching the destination server. The destination server will decapsulate the DCP packet and get the original packet. When the DCP packet flows through the switches, the switches may take more actions according to the DCP packet type: The packet may be uncompressed, or a new cache is placed in the switch, or a feedback packet of cache miss is sent.

## *B. Packet Header of DCP*

Each application packet is encapsulated in a DCP packet as data with extra DCP headers added beforehand. The DCP header is presented in Fig. 3 with a variable length from 32 to 96 bits. The first 8 bits of each header is a tag identifying the category of data carried in a DCP packet. We use 6 of the 8 bits with each bit identifying one kind of data while another 2 bits are reserved for future use. The packet header is depicted in Fig. 4. We describe each kind of DCP packet as follows:

**Original packet:** The application level TCP packet is sent from source server to destination server without any compression or uncompression.

**Compressed packet:** The TCP packet is sent after compression. Moreover, another 64 bits will be appended to DCP header representing the compressing server and uncompressing switch IP address respectively.

**Uncompressed packet:** When the compressed packet reaches the uncompressing switch, the uncompressing switch searches its local cache and get a cache hit, then the compressed packet is uncompressed.

**Bloom Filter packet:** The payload of this kind DCP packet is a bit array. It is used for sharing switch's local information with servers, more details are provided in Section II-C.

**False positive feedback packet:** When a compressed packet reaches its uncompressing switch, if the switch does not find the packet in its local cache, the uncompressing switch will send back a feedback to the compressing server.

**Cache placement packet:** When a server sends an original packet, we need to place a cache item for the packet in some switch along its routing path.

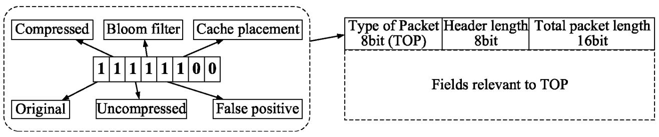


Fig. 3: Packet Header Fields of DCP Protocol

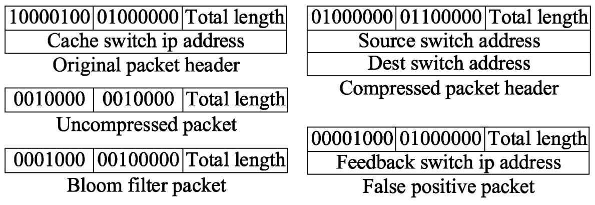

10000100 | 01000000 | Total length
Cache switch ip address
Original packet header
0010000 | 0010000 | Total length
Uncompressed packet
0001000 | 00100000 | Total length
Bloom filter packet
01000000 | 01100000 | Total length
Source switch address
Dest switch address
Compressed packet header
00001000 | 01000000 | Total length
Feedback switch ip address
False positive packet

Fig. 4: Header for Each Kind of DCP Packet

## *C. Cache Sharing Mechanism*

In each switch, there is an in memory data structure storing the cache of packets. Each entry of the cache is in the form of $(key, value)$ pair where $key$ is the 128-bit digest when MD5 algorithm [10] applied to value and $value$ is the original TCP packet content. The number of cache entries, denoted as *cache-no*, is a parameter that can be configured according to the main memory size of each switch.

We use Bloom Filter to store the summary of local cached packets and share the summary between servers. Every switch has a data structure using Bloom Filter named *packet-set*, as depicted in Fig. 5, storing the summary of the *cache-no* cached packets in a switch as an array of $m$ bits.

Every server has two Bloom Filters named *cache-table* and *fp-table* as in Fig. 5. The *cache-table* is a map of $s$ entries, each storing an (ip,packet-set) pair, where $s$ is the number of switches in the data center. The key is the ip address of a switch in the data center and the value is *packet-set* of

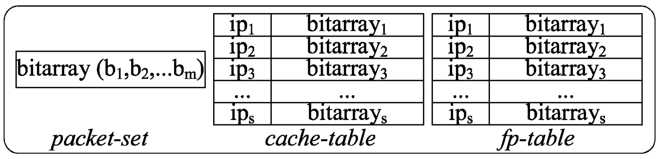


Fig. 5: Three Data Structures using Bloom Filter

the corresponding switch. The *fp-table* has the same structure as the *cache-table* which stores the false positive feedback information from the corresponding switch. How these Bloom Filter structures are used to share cache information and reduce false positive ratio are described below:

When a server wants to send a packet to another server, it will first get the routing path and the switches along the path, then get the packet-set (named $ps$) for each of these switches in *cache-table*. If the packet cache is found in one of the down-link switches, then a compressed packet is sent, else an original packet is sent.

When a server receives a Bloom Filter packet from a switch, the server will update the $value$ of the entry in *cache-table* corresponding to the source switch.

When a compressed packet is sent from source server, but actually the packet is not cached in the uncompressing switch, in such case a false positive occurs. For each packet that has a cache in switch with address $swt$ according to *cache-table* in a server, but it turns out the packet is not cached in switch $swt$, then this packet will be inserted into *fp-table* corresponding to entry $swt$. The false positive ratio reduction is achieved by checking *fp-table* before *cache-table* every time searching for a packet cache in some switch. If the packet is found in *fp-table*, then the *cache-table* is not searched. If the packet is not found in *fp-table*, then *cache-table* is searched.

## *D. Packet Encapsulation and Routing Mechanism*

To encapsulate a TCP packet in a DCP packet, the server first gets all the switches this TCP packet will pass through. Then the server computes the packet's MD5 digest and get $k$ hash values from the digest. If there is cache hit, the server sets the compressed sign bit of DCP header and fills the source server address and destination cache switch address in the DCP packet header and fills the packet content with digest. If there is no cache, the server first chooses a switch along the path to place a cache for this TCP packet, then sets the original packet bit and fills DCP packet content with original TCP packet. Each time a server wants to send a packet to another server, it first generates an original packet or a compressed packet, and then send this DCP packet to the connected edge switch.

When the DCP packet reaches the destination server, the server will decapsulate the DCP packet to a TCP packet. A destination server will receive two kinds of DCP packet: an uncompressed packet or an original packet. The packet content of them will be extracted as original TCP packet.

Next, we propose how a DCP packet passes through switches until it reaches destination server. How the switch forwards the packet is shown below:

For an original packet, the switch first checks the $cache\ switch\ ip$ address. If this address equals to the local IP address, then the switch extracts packet content and computes MD5 digest to add this packet to its local cache by LRU algorithm [11]. Meanwhile, the switch computes $k$ hash values from the MD5 digest to update *packet-set*. If $cache\ switch\ ip$ address is not local IP address, the switch will forward the packet directly to next hop switch.

For compressed packet, the switch first extracts source server IP $sip$ and destination switch IP $dip$ from packet header. If $dip$ equals local IP address, then it gets the MD5 digest of the packet as $dgst$. Then the switch searches its local caches for $dgst$. If the cache exists, the switch retrieves the original packet content from the local cache and makes up an uncompressed packet with the original packet. Then the uncompressed packet is forwarded to the next hop switch or server. If the cache does not exist, the switch sends a false positive packet back to $sip$ to tell $sip$ that a false positive happens on $dip$ with packet $dgst$, afterwards drops this packet. If $dip$ is not the local IP address, then forward the packet to next hop switch.

For uncompressed packet, just forward it to next hop switch until it reaches the destination server.

For Bloom Filter packet, core switches and aggregation switches route it following Fat-Tree's routing protocol. Edge switches broadcast the packet to all the servers in its subnet. When a server receives a Bloom Filter packet, it extracts the switch IP address and bit array from the header, then update local *cache-table* corresponding to the switch.

For false positive packet, the packet is routed by Fat-Tree's routing protocol until it reaches the source server. When a server receives a false positive packet, it extracts the switch IP address where false positive happens at $sip$, and MD5 digest $dgst$ from packet header. Then $dgst$ is inserted into the entry corresponding to $sip$ in *fp-table*.

## III. Performance Evaluation

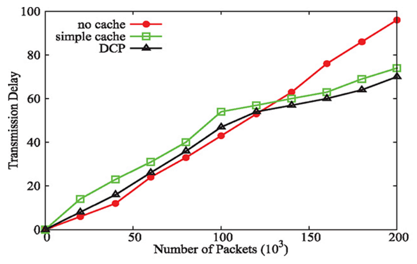
The following metrics are used for evaluations. *Packet transmission delay*, the time for a packet to transmit from one server to another server in the data center. *Bandwidth reduction*, how much network bandwidth it saves by a cache scheme. *Cache hit ratio*, the ratio of packets that has a cache in switches of its routing path.

Experiment Setup: We build up a Fat-Tree data center with 16 pods, which has 1024 servers, 128 edge switches, 128 aggregation switches, and 64 core switches. The packets are generated by servers which have a default Zipf distribution.

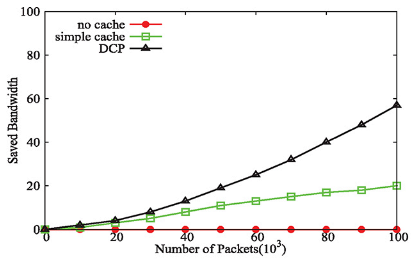
Fig. 6(a) presents the result of transmission delay time of packets. When the packet number is below 100K, the transmission delay time of DCP is the largest because it spends much time on the calculation of MD5 and Bloom Filter. Fig. 6(b) shows the reduced network bandwidth. DCP beats SimpleCache because cache information are shared with Bloom Filter which increase the hit ratio a lot. Fig. 6(c) shows the ratio of a packet to hit a cache item in the switches. The hit ratio of DCP is about twice higher than SimpleCache.

Figure 7(a) presents the bandwidth reduction of Bloom Filter. When Bloom Filter is used, each switch shares its local cache items with a Bloom Filter bit array. Otherwise, the MD5 digest for each cache items are shared among switches. Sharing MD5 digest of each packet does reduce the consumed bandwidth because it increases the hit ratio of cache item.

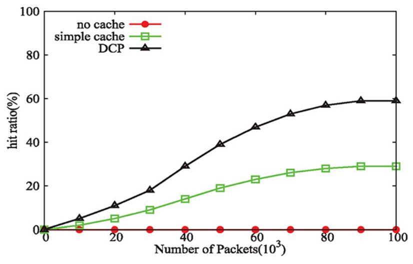
In Fig. 7(b) and Fig. 7(c), we evaluate the impact of cache information update interval by bandwidth reduction and hit ratio. The total number of packets sent by servers is 100K and 150K respectively. From Fig. 7(b), we can see that when switch's local cache information is shared too frequently, the bandwidth reduction is not very sharp. As the interval



(a) Transmission Delay

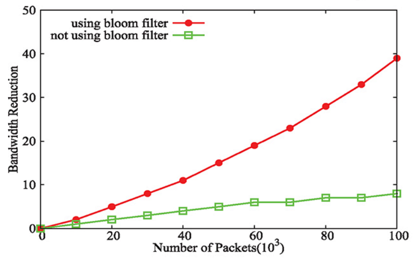


(b) Saved Bandwidth

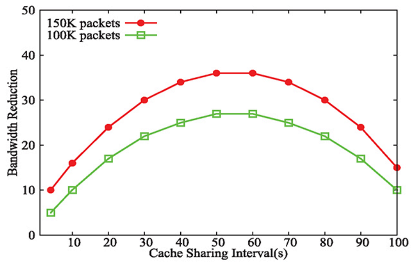


(c) Cache Hit Ratio

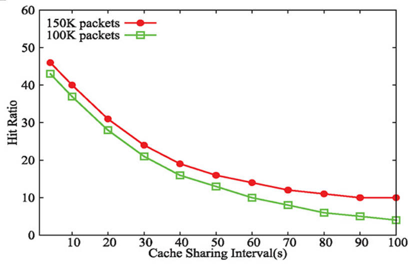
Fig. 6: Performance of DCP vs NoCache and SimpleCache



(a) Performance of Bloom Filter

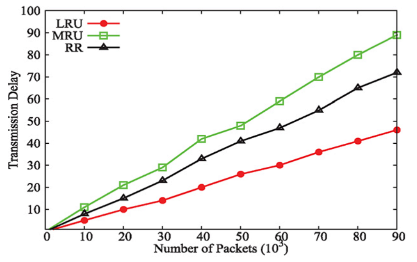


(b) Bandwidth Impact of Update Interval

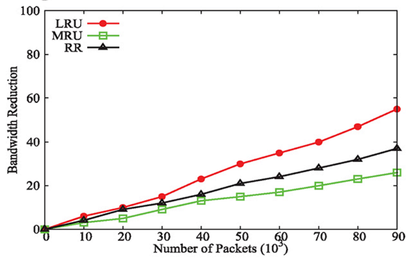


(c) Hit Ratio Impact of Update Interval

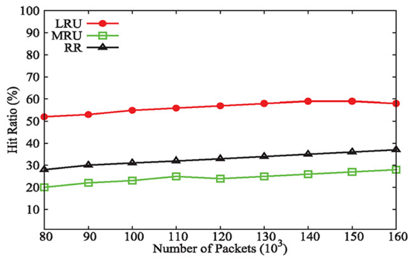
Fig. 7: Effectiveness of Bloom Filter



(a) Transmission Delay



(b) Saved Bandwidth



(c) Cache Hit Ratio

Fig. 8: Performance of Different Cache Replacement Algorithm used by Switch

increases, there is a balance between cache information and packets transmitted in the network. From Fig. 7(c), we evaluate the relationship between hit ratio and cache update interval. It is very clear that as the update frequency decreases, the hit ratio decreases since each switch holds more out-of-dated cache information.

There are various cache replacement algorithms proposed, we choose to evaluate three classical algorithms(Least-Recently-Used,Most-Recently-Used,Random-Replacement) to test which one is best suited for our DCP protocol. From Fig. 8(c), we can see that the hit ratio of LRU is the highest. Similar to the hit ratio, the saved bandwidth of LRU is the best, with RR follows in Fig. 8(b). Besides, we evaluate the transmission delay for various number of packets, the delay of MRU is the highest, while LRU is lowest. Taken these aspects into consideration, we choose LRU as the default cache replacement algorithm for our DCP protocol.

## IV. Conclusion

In this paper, we propose a distributed and efficient cache sharing protocol for data center network with Fat-Tree topology. The protocol header are presented in detail, and the communication algorithms among switches are described thoroughly. We also evaluate the performance of DCP from different aspects. The simulation results show that our protocol is practical and very efficient in reducing packet transmission delay and eliminating redundancy packets in a data center.

## References

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS*, 37(5): 29–43, 2003.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, 26(2): 4, 2008.

[3] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, 51(1): 107–113, 2008.

[4] D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic," in *ACM SIGCOMM*, 30(4): 87–95, 2000.

[5] Y. Cui, S. Xiao, C. Liao, I. Stojmenovic, and M. Li, "Data centers as software defined networks: Traffic redundancy elimination with wireless cards at routers," *IEEE Journal on Selected Areas in Communications*, 31(12): 2658–2672, 2013.

[6] D. Wessels, "Application of internet cache protocol (icp)," 1997.

[7] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions*, 8(3): 281–293, 2000.

[8] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOM*, 38(4): 63–74, 2008.

[9] C. Clos, "A study of non-blocking switching networks," *Bell System Technical Journal*, 32(2): 406–424, 1953.

[10] R. Rivest, "The md5 message-digest algorithm," 1992.

[11] T. Johnson and D. Shasha, "X3: A low overhead high performance buffer management replacement algorithm," 1994.