

Abstract—Data center networks continually seek higher network performance to meet the ever increasing application demand. Recently, researchers are exploring the method to enhance the data center network performance by intelligent caching and increasing the access points for hot data chunks. Motivated by this, we come up with a simple yet useful caching mechanism for generic data centers, i.e., a server caches a data chunk after an application on it reads the chunk from the file system, and then uses the cached chunk to serve subsequent chunk requests from nearby servers. To turn the basic idea above into a practical system and address the challenges behind it, we design content-centric data center networks (CCDNs), which exploits an innovative combination of content-based forwarding and location [Internet Protocol (IP)]-based forwarding in switches, to correctly locate the target server for a data chunk on a fully distributed basis. Furthermore, CCDN enhances traditional content-based forwarding to determine the nearest target server, and enhances traditional location (IP)-based forwarding to make high utilization of the precious memory space in switches. Extensive simulations based on real-world workloads and experiments on a test bed built with NetFPGA prototypes show that, even with a small portion of the server's storage as cache (e.g., 3%) and with a modest content forwarding information base size (e.g., 1000 entries) in switches, CCDN can improve the average throughput to get data chunks by 43% compared with a pure Hadoop File System (HDFS) system in a real data center.

Index Terms—Data center network, HDFS, protocol design.

I. INTRODUCTION

DATA CENTERS have become the hubs for Internet services and applications. An enormous amount of data is processed and shared among servers in a data center, which puts pressure on the data center network to provide high end-to-end throughput. Applications that depend on good network performance include those that shuffle data between mappers and reducers in a MapReduce computation [1], accessing chunks of data in distributed file systems like Hadoop File System (HDFS) [2], computing the “friends-of-friends” relation for millions of subscribers of Facebook [3]. In the past several years, considerable effort has gone into using more as well as faster switches/links and innovative topologies to increase the data center network capacity [4]–[7]. The extra hardware cost, complexity and increased energy consumption, however, may limit how far we can go in this direction.

Recently, researchers are exploring an alternative way to improve the data center network performance, i.e., by intelligent caching and increasing the access points for hot data chunks [8], [9]. The fundamental intuition comes from the fact that data chunks are usually repeatedly accessed in a data center, and hot chunks might be concurrently read by even hundreds of nodes. For example, by mining one day of Facebook traced data [10] on a cluster of 3000+ servers, we find that the most popular file is accessed by 721 times. These hot spots easily result in oversubscribing links, and consequently increasing overall application response times. Given that data center file system applies few file replicas (e.g., 3 in HDFS) with non-trivial file size (e.g., 128MB in Yahoo), such hot spot issue limits end-to-end network throughput.

Motivated by these observations, our first contribution comes up with a simple yet useful caching mechanism for generic data centers, i.e., servers cache data chunks locally after applications read them from the file system, and these caches serve subsequent data requests from nearby nodes. The only requirement is a very small fraction of the storage space in servers used as the cache, and thus it does not require deploying any additional dedicated caching infrastructure in the data center such as Memcached [8]. But it will benefit from using memory as the cache in servers to accelerate I/O speed. The simple approach embodies values in multiple ways. First, more popular file chunks automatically have more cached copies, because they will be read by more applications. Second, many data flows see a much shorter path in the typical hierarchical data center network topology, resulting in higher end-to-end network throughput and lower latency.

Although desirable, it is a challenge to turn this basic idea into a practical system. Firstly, it needs to correctly deliver each data request to the nearest data source, either a caching server or the origin server. Centralized tracking of cache locations is impractical considering current controller in data center is already highly loaded [11]. DHT (distributed hash table) based solutions [12] are not
suitable in our system either, because the locations of the cached chunks cannot be pre-determined by such a hashing mechanism. Instead, we need to depend on the servers whose applications read the data chunks. Secondly, given the need to reduce cost, today’s data center switches seek to limit the amount of fast memory used and servers also may not have enough spare storage space. The caching system needs to be effective with very limited forwarding table size in switches and a relatively small cache space in servers. Finally, considering data updates are normal, a lightweight distributed mechanism is required to address potential cache inconsistencies.

To address the challenges, we design the **Content-Centric Datacenter Network (CCDN), which is a distributed system** for caching, locating the content, and responding to subsequent requests. The fundamental contribution of CCDN is to enable forwarding data request to the nearest caching server as well as maintaining the cache consistency in a distributed and practical way. To this end, CCDN depends on a hybrid of content-based and location-based forwarding mechanism in switches to forward a content request to the correct target server on a hop-by-hop basis. Specifically, CCDN enhances the traditional content-based forwarding to be direction-aware to find the nearest target server. For example, a CCDN switch prefers to forward the content request packet to downstream port, which is closer to the server layer of a multi-rooted tree topology (e.g., Fat-Tree [4]). Given the limited size of fast memory in data center switches and the fact that the number of chunks accessed can be extremely large, the content forwarding information base (FIB) in a CCDN switch only maintains the forwarding entries for a fraction of data chunks. A least recently used (LRU) policy is used for replacement of the content FIB entries. We subtly enhance the traditional location-based forwarding by hashing the content name to get the next hop. With this hashing algorithm, we can maximize the aggregate utilization of content FIBs in each pod, which means a content name can only exist in one aggregate switch. On a miss in the content FIB, a CCDN switch resorts to location [Internet Protocol (IP)]-based forwarding to forward the content request. CCDN further enhances traditional location(IP)-based forwarding by choosing the next hop based on a hash of the content name, which will be explained in Section III-B.

Once the target server is located, CCDN pays no attention on how to accomplish the data chunk delivering between target server and requesting server. Either conventional transport protocols (e.g., TCP) or advanced ones (e.g., MPTCP [13]) can be used in this process. The most important value of this design is to make CCDN provide end-to-end reliable data transmission. Moreover, certain flow scheduling approaches, such as Hedera [14], are helpful in maximizing the aggregate network utilization for data transmission. After reading a data chunk from the target server, the requesting server stores the chunk in its local CCDN cache and updates the content FIB on switches with CCCP, to serve subsequent requests from nearby servers. We claim that caching data in memory (e.g., Memcached [8]) instead of disk will further improve the throughput by reducing I/O latency. Every caching server independently updates the local cache based on an LRU replacement policy. When the data chunks are updated in the file system, CCDN does not need to notify the caching servers. Instead, a version number is maintained in the caching servers and the content FIBs of switches, and the outdated information is updated in an on-demand manner when serving new chunks requests.

We evaluate CCDN’s performance using both NS-3 based simulations and testbed experiments. Our results show that in a Fat-Tree based data center network, with realistic workloads from Facebook [10], using just a small portion of the server storage as cache (e.g., 3%) and with a modest content FIB size (e.g., 500 entries) in switches, CCDN can improve the average throughput to get data chunks by 43% compared with pure HDFS system. The improvement is even higher with larger caches at servers, increased memory in switches for the content FIB, a more skewed distribution for the data popularity and a higher arrival rate of data requests. We have prototyped the CCDN forwarding engine with NetFPGA cards. The testbed experiments based on NetFPGA prototypes also demonstrate CCDN’s benefit over pure HDFS and common caching system.

Although CCDN shares similar idea of content-based forwarding with other recent CCN (content-centric networking) proposals [15], there are substantial differences in our design. First, CCDN uses *off-path* caching while CCN advocates *in-path caching*. Low-end switches used in data centers are not expected to have the memory to be able to support a cache. Second, CCDN uses content-based forwarding only when locating the target server (the first phase), and TCP/IP is used to deliver the data contents (the second phase). Instead, CCN uses content-based forwarding during all the process. Third, CCDN combines content-based forwarding with location(IP)-based forwarding to overcome the constraint of forwarding table size in data center switches.

The rest of the paper is organized as follows. Section II presents the design rationale and overview of CCDN. Section III describes the design details of the CCDN forwarding engine. Section IV and Section V report our simulation and testbed experiments to evaluate CCDN. Section VI discusses related work. Finally Section VII concludes the paper.

**II. Design Rationale and Overview**

A. Basic Idea and Design Challenges

CCDN is a distributed system combining both in-server caching and in-switch forwarding, working harmoniously with the distributed file system in a data center, to improve network performance. Throughout this paper, we use HDFS [16] as an example of the distributed file system, but the CCDN framework is general enough to work with other file systems.

The basic idea of CCDN is caching data chunks on servers after an application on the server reads the data chunks from HDFS, and using the cached data chunks to serve subsequent data requests. The great advantage of this idea is easily mitigating the hot spots for popular data chunks (hotter chunks will naturally be cached by more servers) and reducing the flow path length (data requests will be directed to the nearest caching server or the origin server). Given the limited free storage on servers, every server independently manages its
cache space using an LRU replacement policy, and serves data requests on a best-effort basis.

In CCDN, the greatest innovation is to enable delivering each data request to the nearest target server (caching or origin server) in a distributed and practical way. Since the requesting server’s location can be random, this mechanism also naturally leads to balancing the load among all the caching and origin servers. Therefore, a major issue in the CCDN design is how to find the nearest target server for a data request.

The most intuitive approach is to centrally track the locations of all the caches, e.g., relying on HDFS’s controller to accomplish a Centralized Name Resolution and Scheduling (CNRS) methodology. However, CNRS does not scale because it not only reduces the efficiency of job scheduling, but also raises storage overhead on the controller. We analyze the limitation of CNRS in the appendix and evaluate it in section IV-E. Alternative distributed solutions such as DHT based [12] require structured storage of the cache copies (in accordance with the hashing), which does not match the needs of a highly dynamic and non-deterministic data accesses typical in data centers.

CCDN employs content-based forwarding in switches to map a data request to the correct cache. This mechanism can be implemented by using the emerging SDN framework, and can work in a fully distributed manner to deliver the traffic. Although traditional content-based forwarding can automatically direct a data request to a caching server, in CCDN we have to enhance it to address the following challenges.

Flow and congestion control. Existing content-based forwarding protocols lack a proven flow control and congestion control mechanisms, which is required in an operational, high-speed data center environment.

Limited content FIB size. Given the very large number of data chunks and the broad adoption of low-end switches with ‘narrow’ fast memories in data centers, it is nearly impossible for switches to maintain a content FIB that contains entries for all the data chunks.

Inability to maintain requesters’ information. For the same reason above, it is also difficult for a switch to maintain the requesters’ information for all the in-flight data requests (as in other content-centric network solutions, e.g., NDN [15]). Therefore, we have to forward data packets correctly to the requesting server without depending on maintaining the requester’s information in the switches.

Determining the nearest content source. Given multiple candidates, a content-based forwarding mechanism needs to find the nearest target server (caching server or origin server), so as to reduce the average path length of data flows.

Cache inconsistency. When a chunk is updated in HDFS, we need to deal with stale caches in caching servers and stale content FIB entries in switches. Furthermore, this mechanism should have low overhead to allow for potentially very frequent updates to data chunks.

B. Technical Contributions

CCDN makes the following technical contributions by addressing the above-mentioned challenges.

First, CCDN divides the process of getting a data chunk into two phases. The first phase is the target server location, using an extension of content-based forwarding to find that target server (cache). The second phase is data transfer, which uses existing transport protocols with flow and congestion control built-in (e.g., TCP), and location(IP)-based forwarding to transfer data from the target server (cache) to the requesting server. In addition, CCDN can also leverage recently proposed flow scheduling (e.g., Hedera [14]) and transport protocol approaches (e.g., MPTCP [13]) to accelerate data transmission.

Second, CCDN switches employ a hybrid content-location forwarding in the first phase. A switch maintains both a content FIB as well as a location (IP) FIB. The content FIB only stores the forwarding entries for a fraction of data chunks, up to the limit of the available memory space, with an associated LRU replacement policy. The packet requesting the chunk carries both the content name and the address of the origin server in HDFS. If the chunk name hits the content FIB, content-based forwarding is used; otherwise, the switch resorts to location-based forwarding to direct the request towards the origin server.

Third, CCDN enhances traditional content-based forwarding to be direction-aware when using it to locate the nearest target server caching the content in the first phase. Exploiting the multi-stage graph feature of hierarchical data center network topologies, a direction-aware content-based forwarding always delivers a requesting packet to the target with the shortest path. Besides, CCDN enhances traditional location-based forwarding by hashing the content name to get the next hop, so as to make efficient utilization of the limited content FIB space in switches (refer to Section III-B).

Finally, to solve the cache inconsistency problem during chunk updates, CCDN contains the chunk’s version number in the content FIB entry at the switches. Instead of invalidation on an intentional update, we adopt a policy of invalidation on access. By letting every packet requesting a chunk carry the latest version number of the chunk (available from the HDFS controller), switches with an outdated content FIB entry or servers with an outdated cache entry will delete the stale entry when receiving a data request with a fresher version number. Thus, it is unnecessary to notify switches or caching servers when there is chunk update in HDFS, and the outdated information can be updated in an on-demand fashion by new requests.

C. Design Overview

CCDN works as a shim layer between applications and HDFS. To give an overview of our design, we first introduce the components in CCDN, which includes CCDN caching service in servers, a CCDN forwarding engine in switches, and the CCDN cache control protocol (CCCP). Then we will describe the workflow for read a data chunk in CCDN. Finally, an example is demonstrated to present the benefit of CCDN.

CCDN Caching Service: The CCDN caching service runs as a daemon in every server. It is in charge of local caching, interaction with applications, HDFS, as well as CCDN caching services at other servers.
CCDN Forwarding Engine: CCDN switches are equipped with a CCDN forwarding engine, implementing a hybrid content-location forwarding mechanism to direct a data request to the nearest target server.

CCDN Cache Control Protocol (CCCP): CCCP is responsible for controlling the cache-related interactions for the CCDN caching service, such as finding the nearest caching server, handling a cache miss, etc. CCCP messages are generated and processed by the CCDN caching service and forwarded by CCDN switches. They also signal updates to the content FIB in CCDN switches.

Work Flow: Fig. 1 illustrates the workflow for a requesting server to read a data chunk in CCDN. The first three steps just like a regular HDFS client, except that CCDN caching service intercepts the application’s data request in step 1 and extracts the location of the origin server, as well as the latest version number of data chunk at step 3. When the requesting server’s CCDN caching service intercepts the application’s data request (step 1), it first queries the HDFS controller to obtain the location of the origin server in HDFS as well as the latest version number of the data chunk (step 2-3), just like a regular HDFS client. Then, the requesting server’s caching service seeks to find a nearby target server (that has cached the chunk, if any) using CCCP messages (step 4-5). After determining a caching server or the origin server as the target server, the requesting server establishes a TCP connection with that target server and gets the data chunk (step 6) using TCP. Upon completion of the data transfer, the requesting server stores the data chunk in the local cache, so as to be able to serve subsequent data requests. The CCDN daemon simultaneously delivers the data chunk to the application process (step 7) and transmits a message to update the content FIBs in the switches (step 8). The details of CCDN forwarding in switches are described in the next section.

Example: Fig. 2 illustrates the benefit of CCDN by a simple example. In this example, the network topology is a Fat-Tree with three layers of switches. When server $S_5$ first reads a data chunk $D$, it fetches the chunk from the origin server in HDFS, $S_1$. After that, $S_5$ caches chunk $D$, and the switches along the path ($S_5$, $A_3$, $C_2$, $A_1$, $T_1$) add a new entry for chunk $D$ into the content FIB. Later, when another server $S_6$ reads the same chunk $D$, switch $T_3$ will forward the data request to server $S_5$ instead of $S_1$ by content-based forwarding. By doing this, both $S_1$ and $S_5$ can serve the data requests for chunk $D$, and the path length for $S_6$ to get the chunk is reduced from 6 hops to 2 hops. Both these factors considerably increase the data center network performance.

III. CCDN Forwarding

In this section we describe the forwarding mechanism used in CCDN switches, the main focus of our CCDN design.

A. CCDN Cache Control Protocol

CCDN needs a protocol to locate the nearest caching server for the requesting server node, handle cache consistency on caching servers and content FIB consistency on CCDN switches. In particular, it must achieve three functions: signal a request for content; update caching servers to maintain cache consistency; and update the content FIB on switches. To this end, CCDN uses CCCP (CCDN Cache Control Protocol) as the signaling protocol.

We implement CCCP as a new higher-layer protocol over IP, just like ICMP [17]. Each CCCP message type contains a content name and a corresponding version number. We provide a detailed description of the CCCP messages below.

Content Request: The requesting server sends out a content request message, which includes the name and the latest version of the requested chunk as well as the location of the origin server in HDFS. The content request message will be delivered to a nearby caching server or to the origin server by the forwarding rule in CCDN switches.

Content Reply: When a caching server receives a content request message and it has the requested data chunk in the local cache, it responds to the requesting server with a content reply message. The content reply message serves two purposes. First, it notifies the location of the caching server to the requester. Second, it updates the content FIB entry for the requested chunk in the switches involved along the path. Note that if a content request message is delivered to the origin server, the origin server responds with a content reply message. The reply from the origin server not only notifies the requesting server that the target server to fetch the data from is the origin server, but also adds an entry into the content FIB to serve subsequent data requests, which results in balancing the load among the caching servers and the origin servers.

Content Finish: When a requester successfully fetches a chunk from the target server (whether the origin or other server), it can then be a caching server to serve nearby requests...
for that chunk. It therefore sends out a content finish message towards the target server, which notifies the related switches along the path to update the content FIB entry to add a new port.

Content Rejection: When a caching server receives a content request message but its local cache does not have the chunk (because of the local cache replacement policy), it sends a content rejection message as the response to the requesting server. The content rejection message not only triggers the requesting server to send another content request message, but also deletes the corresponding content FIB entry or the corresponding port from the entry in related switches along the path. Note that after a requesting server receives three content rejection messages, it directly establishes a connection with the origin server to fetch the data chunk, without sending additional content request messages.

For the content request packet, the destination and source addresses are set to that of the origin server and requesting server, respectively. When a target server responds with a content reply or content rejection packet, the destination address is the requesting server. For the content finish packet sent by the requester, the destination address is that of the target server.

B. Hybrid Content-Location Forwarding

The core contribution of CCDN is the novel forwarding engine which is able to direct a content request packet to the nearest server (caching or origin), and to maintain a consistent state across switches and caching servers. To this end, CCDN switches use a hybrid content-location forwarding mechanism for CCCP packets. For regular IP packets (in the data transfer phase), CCDN switches use the regular forwarding mechanism, e.g., by looking up the destination addresses of the packets.

Each CCDN switch maintains two forwarding tables, a content (name) FIB and a location (IP) FIB. The location FIB is constructed by a traditional location-oriented routing protocol such as OSPF [5], or a topology-specific routing protocol such as PortLand [18]. As for the content FIB, each entry has three fields, the content name, the version number, and the output port set, as shown in Fig. 3. Since the content FIB size in CCDN switches is limited, it is not expected to necessarily contain the FIB entries for all the chunks in the network. An LRU policy is adopted to replace the content FIB entries.

Given the rich connectivity and the potential for multiple parallel paths to be available in modern data center networks, we also need to solve the following problems when forwarding CCCP packets.

Problem 1: When forwarding a content request packet and there are multiple output ports in the corresponding content FIB entry, the switch has to decide which output port is towards the nearest target server; and it is desirable to balance the requests among multiple nearby target servers with the same path length.

Problem 2: When forwarding a content rejection packet, we need to guarantee that the content rejection packet can traverse the precise switches that have installed the content FIB entry (set by content finish packets) earlier, so as to ensure that those invalid entries can be deleted.

Problem 3: Due to the limited size of the content FIBs in data center switches, it is better to use as few switches as possible to maintain the content FIB entry for the same chunk.

We solve the problems above by using two novel forwarding mechanisms in CCDN, which enhance the traditional content-based forwarding and location-based forwarding approaches, respectively. First, we enhance content-based forwarding as direction-aware content-based forwarding, which (randomly) chooses a forwarding port towards the nearest target server from multiple candidates in a content FIB entry, solving problem 1. Second, we enhance location-based forwarding as a content-hash location-based forwarding, which guarantees that all types of CCCP packets for the same chunk between two servers cross a fixed path, solving both problem 2 and 3. Next we describe the design of the two enhanced forwarding rules.

The direction-aware content-based forwarding (DACtFw) depends on the multi-rooted tree topology, e.g., Fat-Tree [4], widely used in today’s data center networks. It is used to forward the content reply, content rejection and content finish packets, as well as content request packets when there is a content FIB miss. In a Fat-Tree network, the ports in a switch can be categorized into two types. If it connects to a higher-level switch, it is called an “upward” port. Correspondingly, if a switch port connects to a lower-level switch, it is called a “downward” port. Given the topologies are multi-stage graphs, the downward ports are always closer to the servers than the upward ports. Hence in DACtFw, when there are multiple output ports in a content FIB entry, CCDN switch always preferentially chooses a downward port. If there is more than one downward port, a random one is chosen in order to keep the load balanced. If there is no downward port, the upward port is chosen. Take Fig. 4 as an example. When the switch receives a content request packet for chunk D from port $p_2$ and both port $p_1$ and port $p_3$ are in the output port set, the switch will choose $p_3$ rather than $p_1$ to forward the packet.

Algorithm 1 is the pseudocode for direction-aware content-based forwarding (DACtFw). If there is one or more downward port in the output port set (lines 1-2), a random downward
Algorithm 1 Pseudocode of DACtFw

**Input:**
- \( p \): Incoming packet; \( E \): Content FIB entry;

1. \( \text{ports} \leftarrow \text{GetDownPorts}(E.\text{portlist}); \)
2. \( \text{if} \ \text{ports} \neq \text{null} \)
3. \( I_o \leftarrow \text{RandSelectPort(ports)}; \)
4. \( \text{else} \)
5. \( I_o \leftarrow \text{SelectPort}(E.\text{portlist}); \)
6. \( \text{SendTo}(p, I_o); \)

Algorithm 2 Pseudocode of CHLocFw

**Input:**
- \( p \): Incoming packet; \( LocFIB \): Location FIB;

1. \( \text{dst} \leftarrow \text{GetDstLoc}(p); \)
2. \( \text{ports} \leftarrow \text{LocFIBlookup}(LocFIB, \text{dst}); \)
3. \( I_o \leftarrow \text{null}; \)
4. \( \text{if} \ \text{ports} \text{ are upward} \)
5. \( \text{cn} \leftarrow \text{GetCtName}(p); \)
6. \( \text{coreSw} \leftarrow \text{HashCt(cn)}; \)
7. \( I_o \leftarrow \text{FindPort}(\text{ports}, \text{coreSw}); \)
8. \( \text{if} \ I_o \text{ is null} \)
9. \( \text{return} ; \)
10. \( \text{else} \) \( \triangleright \) **Downward**
11. \( I_o \leftarrow \text{ports}[0]; \)
12. \( \text{SendTo}(p, I_o); \)

Algorithm 3 Forwarding Content Request Packet

**Input:**
- \( p \): Incoming packet; \( I_i \): Inport
- \( cFIB \): Content FIB;
- \( LocFIB \): Location FIB;

1. \( \text{cn} \leftarrow \text{GetCtName}(p); \)
2. \( \text{vn} \leftarrow \text{GetVerNum}(p); \)
3. \( E \leftarrow \text{CtFIBlookup}(cFIB, \text{cn}); \)
4. \( \text{if} \ E \neq \text{null} \text{ and } \text{vn} > E.\text{vn} \) \( \triangleright \) **Cache inconsistent**
5. \( \text{EntryDelete}(cFIB, E); \)
6. \( \text{if} \ E \neq \text{null} \text{ and } \text{vn} = E.\text{vn} \) \( \triangleright \) **Detect loop**
7. \( \text{if} \ I_i \text{ in } E.\text{portlist} \)
8. \( \text{PortDelete}(E, I_i); \)
9. \( \text{if} \ E.\text{portlist} \neq \text{null} \)
10. \( \text{DACtFw}(p, E); \)
11. \( \text{return} ; \)
12. \( \text{else} \)
13. \( \text{EntryDelete}(cFIB, E); \)
14. \( \text{CHLocFw}(p, LocFIB); \)
15. \( \text{return} ; \)

Fig. 5. Example of content-hash location based forwarding. An upward port is selected to forward CCCP packets by hashing the content name of the chunk, which results in a fixed path to forward all types of CCCP packets for the same chunk.

In content-hash location-based forwarding (CHLocFw), a CCDN switch gets the output port set for the destination address by looking up the location FIB. In a Fat-Tree network, if the output port set has any downward ports towards a server, there should be only one such downward port, and the packet is directly forwarded to it. Otherwise, there are multiple candidate upward ports, each one corresponding to a distinct core switch. CCDN hashes the content name to find an upward port.

Fig. 5 shows an example for content-hash location-based forwarding. For the incoming packet from port \( p_1 \), there are two candidate upward output ports based on the location FIB, \( i.e., \ p_3 \text{ and } p_4 \). Then the switch hashes the content name and gets the port (\( e.g., \ p_3 \)) to forward the packet out.

The pseudocode for content-hash location-based forwarding (CHLocFw) is shown in Algorithm 2. The packet is directly forwarded to the downward port if it exists in the output port set (line 10-11). Otherwise, CCDN hashes the content name to find an upward port corresponding to an active core switch (lines 4-7).

It must be noted that the path between any two servers for the same chunk for all types of CCCP packets is determined under CHLocFw, which yields the following benefits. First, a minimum number of switches are used to maintain the content FIB entry for a data chunk, which makes efficient utilization of the limited-size content FIB in CCDN switches. Second, the content request packet can traverse the exact switches with a content FIB entry for the requested chunk (if any), since the previous content finish packets from caching servers are forwarded along those paths. Third, the content rejection packet has to be forwarded along those switches that need to delete the content FIB entry or corresponding port in the entry, since the content finish packet was previously forwarded along that same path to install the FIB entries. Finally, the content reply packets can also refresh the corresponding content FIB entry by traversing the same switches as the previous content request packets.

C. Forwarding CCCP Packets

In what follows, we discuss in order how to forward the four types of CCCP packets based on the hybrid content-location forwarding engine in CCDN switches.

Content Request is used to detect the target server location. The key role is to correctly deliver the packet to the nearest data source as well as achieving cache consistency based on the latest content version number. Algorithm 3 shows the pseudocode for forwarding a content request packet in a CCDN switch. When receiving a content request packet, the switch first looks up the content FIB by the content name (line 3). If there is a corresponding content FIB entry, the version number is checked to guarantee that the entry is not outdated. If the version number in the packet header is more recent (higher) than the one in the content FIB entry, that entry
Algorithm 4 Forwarding Content Reply Packet

Input:
- \( p \): Incoming packet; \( I_i \): Inport;
- \( LocFIB \): Location FIB; \( cFIB \): Content FIB;
1: CHLocFw\( (p, LocFIB) \);
2: \( cn \leftarrow \text{GetCtName}(p) \);
3: \( E \leftarrow \text{CtFIBLookup}(cFIB, cn) \); \( \triangleright \) Return entry
4: if \( E \neq \text{null} \)
5: Refresh\( (cFIB, E) \);
6: else
7: \( vn \leftarrow \text{GetVerName}(p) \);
8: \( E \leftarrow \text{NewEntry}(cn, vn, I_i) \);
9: EntryInsert\( (cFIB, E) \);
10: return; \\

Algorithm 5 Forwarding Content Finish Packet

Input:
- \( p \): Incoming packet; \( I_i \): Inport;
- \( LocFIB \): Location FIB; \( cFIB \): Content FIB;
1: CHLocFw\( (p, LocFIB) \);
2: \( cn \leftarrow \text{GetCtName}(p) \);
3: \( vn \leftarrow \text{GetVerNum}(p) \);
4: \( E \leftarrow \text{CtFIBLookup}(cFIB, cn) \); \( \triangleright \) Return entry
5: if \( E \neq \text{null} \) and \( vn = E.vn \)
6: PortInsert\( (E, I_i) \);
7: else if \( E = \text{null} \)
8: \( E \leftarrow \text{NewEntry}(cn, vn, I_i) \);
9: EntryInsert\( (cFIB, E) \);
10: return; \\

is deleted (lines 4-5). As mentioned before, due to the independent LRU replacement policy for content FIB entries in different switches, a switch may receive a content request from a port that is in its output port set. In this case, this port should be deleted from the output port set to guarantee the consistency of states across switches (lines 7-8). After that, if there are still other ports in the output port set, the switch uses DACtFw to forward the packet (line 10). The corresponding entry is deleted if it is empty in the output port set (line 12-13). If the switch does not contain a content FIB entry for the chunk, or there is no valid forwarding port, the packet is then forwarded based on the content-hash location-based forwarding (line 14), using the destination address of the origin server, which is carried in the IP header.

Fig. 6 illustrates an example of forwarding a content request packet. Server \( S_8 \) sends the content request packet for chunk \( D \) with the latest version number \( v2 \). The packet is forwarded to a caching server \( S_5 \), along the path \( (T_4, A_3, T_3) \). In switch \( T_3 \), the version number of the content FIB entry for chunk \( D \) is outdated, so the entry is deleted and the packet is forwarded by CHLocFw. Switch \( A_3 \) and \( T_3 \) forward the packet using DACtFw, since there is a hit for that entry in the content FIB.

Content Reply is used to acknowledge the requesting server for the following data transmission, as well as updating the content FIB of switches on the path. The origin server or the caching server with requested chunks will send a content reply packet once it receives the content request packet. The pseudocode for forwarding a content reply packet is shown in Algorithm 4. The packet is forwarded using CHLocFw (line 1). If there is a content FIB entry for the chunk, the switch refreshes it (lines 2-5), so as to favor retaining the content FIB entries. In the case of active chunks at the switch, otherwise, the switch installs a new entry to serve subsequent content request messages, in which case the content reply packet must have come from an origin server (lines 6-9). Load balance among the caching servers and the origin server is achieved.

Fig. 7 shows an example of forwarding the content reply packet. Caching server \( S_5 \) sends the content reply packet for chunk \( D \) to client \( S_8 \). During the forwarding process, switches along the path refresh their content FIB entries or insert a new entry for chunk \( D \). It is worth noting that the content reply packet takes the same path as the content request packet for a given chunk.

Content Finish is sent from the requesting server after fetching the chunk, and used to add a new port or entry to the content FIB on the switches along the path. The main function is to provide better load balance for requesting such data chunks. The forwarding process for a content finish packet is shown in Algorithm 5. Recall that the content finish packet is sent from the requester to the target server. After forwarding the packet using CHLocFw (line 1), the switch inserts the incoming port to the corresponding, valid content FIB entry (line 6), or creates a new content FIB entry for the chunk (lines 8-9), so as to serve subsequent content request messages.

An example is shown in Fig. 8. After the requesting server \( S_8 \) reads chunk \( D \) from a caching server \( S_5 \), \( S_8 \) sends
a content finish packet, with $S_5$ as the destination address. Switches along the path then update their content FIBs. Specifically, switch $T_4$ creates a new entry, while switches $A_3$ and $T_3$ add the incoming port of the content finish packet to an existing entry in the content FIB. Note that a content finish packet will not hit a content FIB entry with an outdated version number, because the outdated content FIB entry should have been deleted when the content request packet was previously forwarded along the same path. The switch will not update the content FIB if the version number in the content finish packet is outdated.

**Content Rejection** is sent from the caching server, not only for informing the requesting server to try other data sources, but also for removing a related port or entry from content FIB on the path. The core mission is to maintain the consistency between switches and caching servers. Algorithm 6 shows the process to forward a content rejection packet by using $CHLocFw$ (line 1). The switch deletes the incoming port of the packet from the corresponding content FIB entry (lines 2-7), since the packet indicates a cache miss in the downstream caching server. If the port deletion results in an empty forwarding port set for the entry, this FIB entry is also deleted.

Fig. 9 shows an example. Server $S_5$ sends a content request packet for chunk $D$, which is directed to the caching server $S_5$. Assume now that there is a cache miss at $S_5$. Thus, $S_5$ sends a content rejection packet. In addition to forwarding the content rejection packet, switches $T_3$, $A_3$ and $T_4$ delete port 1 from the corresponding content FIB entry.

**Algorithm 6 Forwarding Content Rejection Packet**

```
Input: 
$p$: Incoming packet; $I_i$: Inport; 
$LocFIB$: Location FIB; $cFIB$: Content FIB;

1: $CHLocFw(p, LocFIB)$;
2: $cn ← GetCtName(p);$ 
3: $E ← CcFIBLookup(cFIB, cn);$ 
   $\triangleright$ Return entry 
4: if $E ≠ null$
5: PortDelete($E, I_i$);
6: if $E.portlist = null$
7: EntryDelete($cFIB, E$);
8: return;
```

IV. SIMULATION

We study the performance of CCDN for operational-scale data centers and compare it with that of pure HDFS, using an NS-3 simulation.

A. Simulation Setup

We implement the caching service module on the server nodes and the hybrid content-location forwarding engine on the switch nodes in the NS-3 simulator. We first compare CCDN with other schemes by examining the average throughput for flow transfers. To study the detailed performance of CCDN, we vary several related parameters (i.e., content FIB size) and measure the following metrics: the average path length of flows (in terms of hops), the average throughput, the hit ratio at the CCDN caches in the servers as well as the hit ratio at the content FIBs in switches.

We use a Fat-Tree as the data center network topology. All the switches have 24 ports. Thus, there are 3456 servers in total, which is a typical size of a containerized data center. The capacity of each link is 1Gbps. Every chunk has 3 replicas, and the chunk size is 100MB. The packet size is set to 1KB when transferring chunks. We define the server cache proportion as the ratio of server cache size to the HDFS storage size in a server, which should be a relatively small number.

We use a one day (24 hours) of a Facebook traced data [10] on a cluster of 3000+ servers running MapReduce workload as the input for the trace driven simulation. In the trace, there are 17158 file accesses and about 20 million chunk accesses (a file contains multiple chunks) in the data center over the one-day period – translating to about 100 chunk accesses every second. As shown in Fig. 10, the file popularity follows a power-law distribution, in which the most popular file is accessed
721 times, while many other files are accessed only once. With HDFS, 6000+ chunks are stored in each of the HDFS servers. 

We also consider the impact of the following parameters to study CCDN’s performance under different environments.

- CCDN cache proportion, denoted by \( r \). We vary \( r \) as 1%, 3%, 5% and 10%.
- Content FIB size in switches, denoted as \( z \). We pick values for \( z \) as 300, 500 and 1000.
- Skewness factor of accessed chunks, denoted as \( \alpha \). We pick values for \( \alpha \) as 0.8, 1.0 and 1.2.
- Poisson arrival rate, denoted as \( \lambda \). We vary \( \lambda \) as 100/seconds, 300/second and 500/second.

We built two groups of simulation to investigate: 

- The performance of CCDN as we vary the CCDN cache proportion and the content FIB size in switches. We use the one day Facebook trace data as the input while varying \( z \) and \( r \), to understand the capability of CCDN under a realistic data center workload, with different caching capabilities.
- The performance of CCDN as the skewness of chunk popularity varies as well as the (poisson) arrival rate of requests. With the same amount of chunks as the Facebook trace data, we vary \( \alpha \) and \( \lambda \) to demonstrate CCDN’s performance under different access patterns.

B. Comparison With Other Schemes

One of the key merits of CCDN is that it offers multiple replicas of a chunk in order to achieve load balance. Nevertheless, other schemes also provide such a property. A ‘regular caching system’ would also make each client fetch a chunk either from the local cache or the origin server. In other words, it is like a naive CCDN, without CCDN’s switching component. In this way, a limited gain is achieved since only one (or zero) cache is available for every requester.

A simple way is to increase the number of chunk replicas in HDFS (whose default is 3). For example, maintain 4 replicas of each chunk instead of 3. However, such an approach is not entirely scalable and is somewhat impractical. First, even with one more replica, the storage consumption of the whole HDFS increases by one third. Second, as Section IV-A illustrated, with a power-law distribution, hundreds of replicas are required to mitigate the hot spot related to the most popular file. This results in substantial additional storage.

Yet another alternative is to place a special server in each rack for caching only. However, this fails to remove the hot spot issue for popular chunks. The outgoing link of the caching server becomes the bottleneck given that all the servers in the same rack could be simultaneously fetching data from it.

So, we compare CCDN with two approaches: one is the regular caching system described above (Server-Local Caching); the second is to assign a special server, for caching only, on each rack (Dedicated Caching Server). We set the skewness factor of chunk popularity at 1.0 and the chunk access rate at 500, as we will also see in Section IV-D. For CCDN and regular caching system, the server cache proportion is 3%. Content FIB size is set to 500 in CCDN.

Fig. 11 shows that CCDN not only achieves significant improvement compared with HDFS, but also clearly outperforms the other two approaches. More specifically, examining the proportion of flows with throughput higher than 400Mbps, it is 46% in CCDN while it is only 6% and 17% with the other two approaches.

C. Impact of Server Cache Proportion and Content FIB Size

We first study the impact of the server cache proportion, \( r \), and the content FIB size, \( z \), in switches. The file popularity distribution and file access times are exactly the same as in the Facebook workload trace. The result is shown in Fig. 12.

Examining Fig. 12(a) and Fig. 12(b), CCDN outperforms HDFS with even a very small server cache proportion (e.g., 1~3%) and a modest content FIB size at the switches (300~1000 entries). Shown in Fig. 12(a) is the average path length. In HDFS it is around 5.85, while it is only 5.15 in CCDN with a server cache proportion of 3% and content FIB size of 1000. Similarly, from Fig. 12(b), the average throughput to get a chunk in HDFS is 228Mbps, but increases to 400Mbps in CCDN with server cache proportion of 3% and content FIB size of 1000. By using a very small cache in servers and a small forwarding table in the switches, CCDN improves the network throughput to fetch data by 43%. CCDN achieves much better performance with a larger server cache proportion and larger content FIB size in the switches. However, it reaches a point of diminishing returns when the server cache proportion increases beyond 5% and the content FIB size is greater than 1000, for the cases we studied in this trace-driven simulation.

The benefit of CCDN comes from both mitigating the hot spots for popular files as well as reducing the path length of file accesses. Fig. 12(c) and Fig 12(d) show the average hit ratio at the server caches and content FIBs in switches, respectively. Note that either a server cache miss or a content FIB miss will direct a data request to the origin server. Hence, it is intuitive that both the server cache hit ratio and the content FIB hit ratio increase with a larger server cache proportion and larger content FIB size. It is worth noting that even with a server cache proportion of 3% and a content FIB size of 1000, the server cache hit ratio and content FIB hit ratio are quite significant at 25% and 22%, respectively. The considerable gain comes from the fact that popular chunks are accessed much more frequently than the other chunks.

D. Impact of Chunk Popularity Distribution and Chunk Access Rate

We then fix the server cache proportion at 3% and the content FIB size at 500, and study the impact of chunk popularity
distribution and chunk access rate. Similar to the Facebook trace data, we assume the chunk popularity follows a power-law distribution. Specifically, the probability of accessing the $x$-th chunk is $f(x) = a \times x^{-\alpha}$, where $\sum_{x=1}^{N} x^{-\alpha}$ and $N$ is the total number of chunks in HDFS. We call $\alpha$ the skewness factor for chunk popularity distribution. We also assume data accesses follow a Poisson process with an expected arrival rate of $\lambda$. Fig. 13 shows the results.

Fig. 13(a) and Fig. 13(b) show that CCDN consistently achieves significant improvement in performance compared to HDFS, across the range of chunk popularity distributions and chunk access rates. Moreover, CCDN’s advantage is more obvious when the chunk popularity distribution is more skewed or the chunk access rate is higher. With $\alpha = 1.2$ and $\lambda = 500$, the average path length in CCDN and HDFS are 5.12 and 5.84 respectively, while the average throughput to get a chunk in CCDN and HDFS are 122Mbps and 23Mbps, respectively. The remarkable improvement in the average throughput comes from much higher packet loss and consequent retransmissions in HDFS due to traffic congestion, which CCDN can significantly mitigate this by having more caches. Fig. 13(b) also shows that CCDN has much better scalability by being able to support a much higher chunk access rate than HDFS, without reducing the average network throughput. This is because more chunk accesses will result in more servers caching the data, which naturally balances the load among the servers.

From Fig. 13(c) and Fig. 13(d), we conclude that the skewness of chunk popularity has an obvious impact on the server cache hit ratio and content FIB hit ratio. When the chunk popularity is more skewed (larger $\alpha$), more caching servers are available to serve data requests, which magnifies the benefit of CCDN over HDFS. For this very reason, the server cache hit ratio and content FIB hit ratio also increase with higher chunk access rate (larger $\lambda$). With $\alpha = 1.2$ and $\lambda = 500$, the server cache hit ratio is $38\%$ while the content FIB hit ratio is as high as $31\%$.

E. Comparison of CCDN With CNRS

Next, we compare CCDN with a potential centralized approach, which depends on the HDFS controller tracking cache locations and handling cache consistency. We call the centralized approach CNRS (Centralized Name Resolution and Scheduling). We still take HDFS as an example of the distributed file system in a data center. There are two major processes running on the HDFS controller: NameNode and JobTracker. The NameNode manages the metadata of HDFS, which primarily consumes the controller’s storage resource (e.g., RAM); while the JobTracker is a job scheduler that consumes the controller’s computing resource (e.g., CPU). It is reported that the NameNode is already overworked in current data centers [11], which indicates that CNRS will further overload the NameNode on both storage overhead (by saving cache locations in memory) and computing overhead (by frequently interacting with caching servers). We therefore vary two parameters in CNRS: the proportion of NameNodes’s CPU utilization by CNRS, $p$, and the number of caching replicas that can be stored for each chunk, $N$. As for the traffic pattern, we set the skewness factor of chunk popularity at 1.0 and the chunk access rate at 300. With CCDN, the server cache proportion and content FIB size are set as 3% and 500 respectively.

The comparison between CCDN and CNRS($N$, $p$) is shown in Fig. 14. From Fig. 14(a), we find that with $p = 10\%$,
CNRS increases the flow completion time. Given that the controller takes charge of scheduling jobs, fewer computing resources are left for scheduling jobs because more CPU cycles are used by CNRS. It leads to much higher waiting times for many jobs (as well as the flows in these jobs). When $p = 50\%$, CNRS is in fact no better than regular HDFS (without dynamically providing more caches for hotter chunks).

Fig. 14(b) shows the impact on the transfer throughput with varying $N$. No matter whether we set the number of caching locations as 10 or 100, CCDN significantly outperforms CNRS, and both them show advantage over regular HDFS (without dynamically providing more caches for hotter chunks).

V. IMPLEMENTATION AND TESTBED EXPERIMENTS

We have prototyped the CCDN forwarding engine on a NetFPGA [19] card, and carried out testbed experiments based on the NetFPGA prototype.

A. NetFPGA Implementation

The NetFPGA implementation includes programs in Verilog and C with 3500 lines of code. Fig. 15 shows the implementation architecture. It is divided into a hardware and a software plane. The hardware plane works at high speed but is only able to execute simple logic. For complex processing logic, we use the software plane. In our implementation, the data transfer phase and the CCCP content request messages are processed by hardware, while the other three types of messages in the CCCP protocol are handled by software.

Each incoming packet (either CCCP packet or data packet), traverses the 64-bit data bus. The packet header is processed by the Preprocess module, IP module, CCCP module and the Output process module in the hardware plane. If the packets are for content reply messages, content finish messages or content rejection messages in the CCCP protocol, they will be sent to the software plane by the Output process module. The software plane will then update the content FIB and synchronize that in the hardware plane.

The Preprocess module takes charge of the routine operations for every packet, e.g., recalculating the TTL and checksum. After that preprocessing, either a validation signal is sent to the Output process module, or an ‘unqualified’ signal is generated to drop the packet. For a data packet, the IP module extracts its destination IP address and looks up the location FIB to produce the nexthop MAC address. For a CCCP content request packet, the CCCP module looks up the content FIB to get the nexthop MAC address. In the Output process module, if the packet is a data packet or content request packet, it adds the Ethernet header by setting the destination MAC address from the IP module or CCCP module, and then sends the packet to the next hop. But if the packet belongs to the other three types of CCCP messages, the Output process module delivers the packet to the software plane.

It is important to note that the number of FIB entries (both content FIB and location FIB) in our NetFPGA prototype is limited to a maximum of 32. Moreover, NetFPGA cannot support multiple output ports for an entry, for which we have to use multiple entries to represent one content FIB entry with multiple output ports. When the content FIB size is not enough to contain all the content names, LRU is used for content FIB entry replacement.

B. Experiment Setup

We run CCDN in a testbed composed of 17 desktop machines. 12 machines ($S_1$-$S_{12}$) are data center servers running CCDN caching service, while the other 5 machines ($R_1$-$R_5$) are equipped with our CCDN NetFPGA prototype and function as switches. The servers and switches are connected by Gigabit Ethernet links in a tree topology, shown in Fig. 16. Each physical machine has an Intel E7400
C. Experimental Results

From Fig. 17(a), we see that CCDN achieves higher aggregate throughput than HDFS, and the jobs are completed in a shorter time. In HDFS, most servers have to read the chunks from a server 4 hops away, while CCDN reduces the path length by using nearby caches and achieves better load balance among the servers for popular chunks. In the first 26 seconds, CCDN has similar network throughput as that of HDFS, because there are no server caches to accelerate data transfer. But after that, the CCDN server caches help improve the aggregate throughput significantly.

Fig. 17(b) shows the CDF of the job completion time. 80% of the jobs are completed within 105 seconds in CCDN, which is 65 seconds less than what is achieved in HDFS. Moreover, completing all the jobs takes 230 seconds in HDFS, but takes only 138 seconds to finish in CCDN. CCDN reduces the job completion time by 40%. It is worth noting that the implementation experiments also demonstrate that CCDN considerably outperforms HDFS under power-law file access pattern, similar to the simulation results.

We also study the resource utilization (e.g., CPU and memory) on the controller. Fig. 18 shows the CCDN to HDFS comparison for the utilization of these resources in the controller. The ratio of CPU utilization of CCDN versus HDFS is shown as Fig. 18(a). In the first 138 seconds, where CCDN completes all the jobs, the controller has a little higher CPU utilization with CCDN than HDFS. That is because all the chunk requests are processed in CCDN, while only part of the requests are handled in HDFS. However, HDFS has the longer average job completion time. From 138 seconds and 230 seconds, CCDN use less CPU than HDFS, since the controller in HDFS is still handling the remaining chunk requests. After 230 seconds, CCDN has the same CPU utilization as HDFS. From Fig. 18(b) we see that CCDN has the same memory usage on the controller as HDFS does. This is expected since CCDN does not add any additional information in the memory at the controller. The experiment indicates that CCDN adds no overhead on the controller compared to a regular HDFS system.

VI. RELATED WORK

With the tremendous demand of bandwidth in data center networks, many research efforts in recent years have sought to increase network capacity, taking different approaches.

A. Advanced Data Center Network Topology

Recently proposed advanced data center network topologies use a new switch infrastructure to replace the traditional tree structure, or add more links to the tree to increase bisectional bandwidth. For instance, Fat-Tree [4] and VL2 [5] use commodity switches to form a three-layer Clos network to provide a 1:1 oversubscription ratio to all the servers. In contrast, CCDN seeks to increase the network performance through another dimension of the solution space by exploiting server caching.

B. Intelligent Storage

Intelligent storage has also been proposed recently to enhance overall data center performance. Memcached [8] uses a cluster of Memcache servers to cache data in memory and reduce the latency contributed by using I/O to fetch data. Exoskeleton [20] relies on OpenFlow [21] switches to enable the direct delivery of requests to cache nodes. VMTorrent [9] provides a profile-based prefetch approach to reduce the traffic load.

Unlike Memcached and Exoskeleton, CCDN exploits the servers to locally cache data after applications read the data from the file system, without substantial additional infrastructure cost. In contrast, Exoskeleton requires the controller to synchronously maintain cache state. CCDN adds no overhead.
on the controller. The prefetch operation in VMTorrent also bring additional traffic load, whereas CCDN does not generate additional (needleless) data traffic. Moreover, CCDN extends content-based forwarding to find the correct caching server, without the necessity to upload data to specific servers as in DHT based solutions [12].

C. Content Centric Networking

Content-aware routing mechanisms have attracted increasing attention, even in the commercial world. Avere [22] builds an Edge Filer as a cache pool to reduce latency of accessing files. PernixData [23], Atlantis [24] and Infinio [25] incorporate a powerful service into the hypervisor to accelerate transferring data between virtual machines. However, all these content-aware technologies are server side solutions, lacking the ability to enhance the network performance through intelligent forwarding. On the other hand, CCDN not only creates an adaptive caching scheme but also uses a forwarding algorithm to take maximum advantage of the data center network.

Named Data Networking (NDN) [15] is an information centric network architecture to support efficient content distribution. Instead of using location-based forwarding, NDN routers store/cache content to directly respond to data requests, and maintain content FIBs to forward data requests if the data is not cached locally. With pure content-based forwarding, the router has to maintain a PIT table to track a requesters’ information for all the in-flight requests, so as to correctly forward responses. PIT scalability is a concern.

CCDN has several differences from a pure NDN-based solution. First, CCDN does not cache data in switches, because data center switches are usually low-cost, with a limited amount of memory. Instead, data chunks are cached only in end servers. Second, for essentially the same reason, CCDN does not maintain the PIT table to record the requester’s information in switches, thus avoiding the PIT scalability issue. Third, the content FIB in CCDN does not have to contain the entries for all the chunks. When a cache miss occurs, CCDN gracefully turns to location-based forwarding.

VII. CONCLUSION

This paper described CCDN, a fully distributed architecture that leverages caches in servers and content-based forwarding in switches to improve data center network performance, without adding substantial infrastructure cost. The benefit of CCDN comes from mitigating hot spots for popular chunks and reducing the path length of data flows. CCDN extends content-based forwarding only for locating the target server, while continuing to use a well-established and proven transport protocol such as TCP (and its enhancements) for data transfer. To find the nearest target server even with a limited content FIB size in switches, CCDN employs a hybrid content-location forwarding mechanism, by enhancing content-based forwarding to be direction aware. We have implemented CCDN in a prototype data center with NetFPGA devices and also evaluated its performance at reasonable scale with a trace-driven NS-3 simulation. Both the simulation with real-world workloads and testbed experiments with the NetFPGA prototype demonstrate that CCDN can improve the average network throughput of fetching data by more than 40%, even with a very small cache at the servers and a small content FIB in the switches.

REFERENCES


Ming ZHU received the Ph.D. degree in computer science from Tsinghua University, in 2015. He is currently a Software Engineer with LuminateWireless Inc. His research interest includes data center networking and distributed system design.
Dan Li received the Ph.D. degree in computer science from Tsinghua University, Beijing, China, in 2007. He is currently an Associate Professor with the Computer Science Department, Tsinghua University. His research interest includes future Internet architecture and data center networking. He served as an Associate Editor of the IEEE TRANSACTIONS ON COMPUTERS from 2013 to 2015. He is a senior member of IEEE.

Fangxin Wang received the B.S. degree from the School of Computer Science, Beijing University of Post and Telecommunication, in 2013. He is currently pursuing the M.S. degree in computer science with Tsinghua University. His research interests include data center networking and software defined networking.

Anke Li is currently pursuing the Ph.D. degree with the Networking Research Group, Department of Computer Science, Stony Brook University. His current research interests lie in computer networks and network measurement. More specifically, his recent work focuses on building systems to investigate and monitor various issues happening today in different networks and places. These issues, such as online information control and traffic differentiation, are generally related to net neutrality principles.

K. K. Ramakrishnan (F’05) received the M.S. degree from the Indian Institute of Science, in 1978, and the M.S. and Ph.D. degrees in computer science from the University of Maryland, College Park, USA, in 1981 and 1983, respectively. He was a Technical Director and Consulting Engineer in Networking at Digital Equipment Corporation. From 1994 to 2013, he was with AT&T Laboratories-Research, Florham Park, NJ, where he was a Distinguished Member of the Technical Staff. From 2000 to 2002, he was with TeraOptic Networks, Inc., as a Founder and Vice President. He is currently a Professor with the Computer Science and Engineering Department, University of California, Riverside. He has authored over 200 papers and has 149 patents issued in his name. He was also an AT&T Fellow, recognized in 2006, for his work on congestion control, traffic management, and VPN services, and for the fundamental contributions on communication networks with a lasting impact on AT&T and the industry. He received an AT&T Technology Medal in 2013 for his work on mobile video delivery strategy and optimization. His work on the DECbit congestion avoidance protocol received the ACM Sigcomm Test of Time Paper Award in 2006. He has been on the Editorial Board of several journals and has served as the TPC Chair and General Chair for several networking conferences.

Ying Liu received the M.S. degree in computer science and the Ph.D. degree in applied mathematics from Xidian University, Xi’an, China, in 1998 and 2001, respectively. She is currently an Associate Professor with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing. Her research interests include network architecture design, multicast routing algorithm and protocol, and next-generation Internet. She is a member of CCF, ACM, and IEEE.

Jianping Wu received the master’s and Ph.D. degrees in computer science from Tsinghua University. He is currently a Full Professor with the Computer Science Department, Tsinghua University. He has authored over 200 technical papers in the academic journals and proceedings of international conferences, in the research areas of the network architecture, high-performance routing and switching, protocol testing, and formal methods. He is an IEEE Fellow.

Anke Li

Xue Liu received the Ph.D. (Hons.) degree in computer science from the University of Illinois at Urbana–Champaign. He was the Samuel R. Thompson Chaired Associate Professor with the University of Nebraska–Lincoln, and a Visiting Faculty Member with HP Laboratories, Palo Alto, CA. He is currently a William Dawson Scholar and an Associate Professor with the School of Computer Science, McGill University. He has authored over 200 research papers in major peer-reviewed international journals and conference proceedings in his research areas and received several best paper awards. His research has been covered by various news media. His research interests are in computer and communication networks, real-time and embedded systems, cyber-physical systems and IOT, green computing, and smart energy technologies.