Collaborative Content Caching in Wireless Edge with SDN

Zhen Cao and Panagiotis Papadimitriou
Institute of Communications Technology, Leibniz Universität Hannover, Germany
{zhen.cao, panagiotis.papadimitriou}@ikt.uni-hannover.de

CCS Concepts
●Networks → Wireless access networks; ●Information systems → Web indexing;

Keywords
Wireless Access Networks; SDN; Hash Indexing

ABSTRACT
The increasing computing and storage capacity at the wireless access nodes (eNodeB and Access Points) creates opportunities for content caching at the wireless edge and, therefore, further reduction of content delivery delay. However, realizing a large-scale content caching at the wireless edge entails significant challenges, due to the difficulties of managing the vast amount of cached content and handling redirections among the large number of nodes. In this respect, we propose collaborative content caching at wireless edge nodes by coupling a low-false-positive hash table to index the cached content with centralized content routing and redirection, leveraging on software-defined networking (SDN) principles. We present emulation results to assess the benefits of the proposed solution.

1. INTRODUCTION

Despite the huge investment in network infrastructures to increase bandwidth, the inborn latency of the Internet has turned out to be the major factor for the impairment of user’s web experience [1]. Content distribution networks (CDN) and web caching, i.e., the two major approaches to reduce the latency between users and stored content, have been deployed widely by carriers and service providers. However, most of Internet caches have been deployed in data centers, i.e., still 10ms–30ms away from the access network.

Wireless access nodes (eNodeB in LTE or Wi-Fi Access Points) open up new opportunities for caching the content even closer to users. In fact, cellular and Wi-Fi device vendors’ efforts to increase their device’s local storage and processing power [2, 3, 4], coupled with the standardization efforts such as ETSI mobile edge computing [5], are paving the way of realizing content caching at the edge of the network. Carriers also have incentives to deploy content cache at edge, since economic analysis shows that edge network caching results into operating cost savings ranging from 29% to 35% and TCO (Total Cost of Ownership) savings ranging from 8% to 11% over the ten-year study period [6].

Realizing a large-scale content caching system at the network edge entails significant challenges. First, since the storage and computing resources on a single node are still limited, each node can only cache a small portion of the popular content. Therefore, popular content has to be distributed among the wireless access nodes in a coordinated manner. Second, to identify cached content and redirect client requests to cached objects, we need an efficient cached content management and indexing technique.

To meet these requirements, we present a collaborative content caching system at the network edge. We develop a model to instruct the edge node to trigger on-demand caching when popular content has been identified. Subsequently, we leverage on software-defined networking (SDN) principles to provide centralized cached content management and redirection. More specifically, the cache controller employs a hash table for web content indexing. In this respect, we further discuss how to reduce the false positive rate.

The remainder of this paper is organized as follows. Section 2 lays out the background of the work. In Section 3, we present the proposed architecture and enabling technologies. In Section 4, we discuss our evaluation results. Section 5 discusses related work. Finally, Section 6 highlights our conclusions.
2. BACKGROUND

In this section, we provide some background on wireless access network architectures and transparent caching.

2.1 Wireless Access Network

Fig. 1 illustrates the network architecture of LTE and Wi-Fi. The LTE network consists of eNodeB which faces user devices directly, MME which provides mobility management, and SAE Gateway which is the data plane anchor. Correspondingly, in the Wi-Fi network, the Access Point (AP) faces mobile devices directly, the Access Controller (AC) provides centralized mobility and security session management, and the Core switch is the data plane anchor.

We term the layer of eNodeB and AP as the Access Network, and abstract eNodeB and AP as Access Node. The layer of MME, SAE-GW, AC and Core switch is termed as the Core Network. The Access Network and Core Network is connected by the optical transport network which entails huge investment from operators for wide area coverage.

As depicted in Fig. 1, the Internet access latency can be decomposed into the following three parts: (i) wireless access latency, i.e., the latency between client device and access node, which ranges from 10ms to 50ms; (ii) backhaul latency between the access nodes and core network components, which is between 10ms and 20ms; and (iii) Internet traversal latency between the core network and Internet servers, which ranges from 10ms to a multitude of 100ms, all according to existing measurements conducted in [2].

2.2 Transparent Caching

Content caching can be performed in different ways. An original form of Web caching is explicit caching, at which the user is required to add a known cache server to his browser configuration so that all Web requests are redirected to the specified cache proxy. Explicit caching has not become popular, since (i) the user should know the cache locations and manually configure his browser and (ii) content caching in explicit mode is static and oblivious to the popularity of different web objects.

Unlike explicit caching, transparent caching can make intelligent decisions about which objects should be cached locally and keep users unaware of the existence of the cache. By deploying intelligent caches strategically throughout their networks, operators are able to deliver popular content closer to subscribers, thus reduce the volume of transit traffic across their networks. Transparent caching typically assigns client requests to the most proximate cache relying on techniques, such DNS and HTTP redirection.

Most operators nowadays deploy their cache infrastructure at the core network. This entails a considerable operational expense, due to the high cost of the various fiber leased lines that connect the access nodes to the core network. In this paper, we investigate techniques that enable collaborative content caching at the wireless edge to reduce (i) content delivery delay and (ii) the operational cost of the cache infrastructure.

3. ARCHITECTURE

In our architecture, each individual Access Node makes on-demand caching decisions, while a centralized Cache Controller is employed to handle request routing and redirection.

3.1 On-Demand Caching

The Access Node triggers on-demand caching when certain content has been identified as popular. Since Web content can be uniquely identified by the Uniform Resource Identifier (URI), we use URI in the following description of the proposed mechanism.

The Access Node maintains a local cache table. Each entry of this table is indexed by the URI and contains three attributes, i.e., the request rate (e.g., number of requests per second), the cache location, and TTL. If the request rate of a certain URI exceeds a threshold, the AN will trigger the on-demand caching of this URI and assign a TTL value in the entry. The AN will refresh the local cache table periodically.

We develop a model in order to let AN make on-demand caching decision. Let $X_r$ be a random variable denoting the request rate for a URI. Assuming $X_r$ follows a Poisson process [7] with parameter $\lambda$, if a URI has been visited with rate $R$, we define its popularity as the probability that the request rate $X_r$ is equal to or smaller than $R$. More specifically, the Popularity Probability $g(\lambda; R)$ is defined as:

\[ g(\lambda; R) = \sum_{i=0}^{R} \Pr(X_r = i) = \sum_{i=0}^{R} \frac{\lambda^i e^{-\lambda}}{i!} \quad (1) \]

Fig. 2 shows the trends of $g(\lambda; R)$ when $\lambda = 20$ and 30. If $g(\lambda; R)$ exceeds a predefined threshold $t$, the AN will trigger the on-demand caching locally. The choice of the threshold value $t$ reflects the sensitivity of the lo-
3.2 Centralized Content Routing

After the local caching decision has been triggered, the AN caches the content associated with the URI, and subsequently informs the Cache Controller with the set of URIs that have been cached and the IP address of the AN. The Controller centrally manages the mapping between cached contents and their associated locations within the access network, and provides a lookup service for any access node to identify if and where a certain object has been cached.

3.2.1 Hash Table Design

We use a Hash table to manage the content directory and provide the lookup service. As shown in Fig. 3, hash table is a vector with a fixed length K. Each entry of the vector contains an integer value, representing the number of nodes a certain URI has been cached, and an associated pointer. If this integer value is greater than zero, the pointer gives the cache locations (i.e., the IP address of the Access Nodes); otherwise, the pointer is null.

The Hash table is constructed as follows. Initially, all counters and pointers are set to zero and null, respectively. After the Access Node has made a local caching decision, it informs the Controller with the cached URI associated with its own IP address. The Controller then computes a hash value of URI and performs a modulo operation (over the vector size K) with the hash result to get an index value i:

\[ i = H(URI) \mod K, \ i = 0, 1, ..., K - 1 \quad (2) \]

The counter value of the i-th entry of the Hash table is increased by one, and the IP address of the associated Access Node is inserted into the linked list.

Upon the construction, the Hash table on the Controller encodes information about whether and where a certain URI has been cached. Using this Hash table, the Controller can provide a lookup service with a specific URI. On receiving a lookup request of a specific URI, the Controller calculates the index value of the URI based on Equation (2). If the counter value of Htable[i] is greater than zero, the Controller randomly chooses (as a simple load balancing approach) an Access Node from the linked list of Htable[i].

3.2.2 Content Redirection with OpenFlow

Content routing and redirection takes place as follows (Fig. 4):

1. The user initiates a HTTP request towards a web site. AN1 on the first hop will perform a local lookup to check whether the content has been cached. In the presence of a cache on AN1, the content will be directly delivered to the user.

2. A failure to find the cached content locally on AN1 will trigger a packet-in message towards the Controller in order to identify nearby cached content.

3. The Cache Controller will perform a lookup using the Hash table, and check if there is a recorded cache node for the requested URI. In the case of a cache hit, the Controller will randomly select one node (e.g., AN2) from the set of candidates and inform the AN to redirect the request there.
Otherwise, the Controller will inform the AN to start fetching the content from the original server.

4. AN1 will redirect the HTTP request to AN2.

5. AN2 will reply with the requested content.

6. AN1 will deliver the content to the user.

In the sequence of steps shown in Fig. 4, there is a need for an interface for centralized control. We hereby investigate how to use and extend OpenFlow for this task. The motivation of employing OpenFlow is as follows: (i) architecturally OpenFlow is well suited to the content routing scenario, (ii) the packet-in message can be used to report cache miss, (iii) OpenFlow can be used to configure the network-layer routing between different ANs, which is required for requests redirection.

In the most recent release of OpenFlow specification [8], the OpenFlow device could specify three possible reasons of triggering a packet-in message, i.e., table-miss flow entry, action required, and invalid TTL. Flow table miss expresses the absence of any L2/L3 forwarding policies; thus, it cannot be used for the cache-miss case. Since the packet-in reason code is 8-bit long with a maximum of 255 different types and only three of them have been used, we can extend this field with a reason code for cache-miss.

The generation and processing of cache-miss is shown in Fig. 5. Upon receiving a Web request and failing to find a match entry in the local cache table, the AN generates a packet-in message with the reason code specified as cache-miss (Step 1). The Cache Controller handles this cache-miss by performing a lookup over the Hash table and including the result in the packet-out message to the requesting AN (Step 2). This packet-out message contains the spotted cache location of the content as well as an OpenFlow entry. The AN subsequently installs a new entry on its local cache table (Step 3) and flow table (Step 4). As such, we handle content routing and network-layer routing simultaneously.

3.2.3 Parallel Attempts to Offset Cache-miss

Cache-miss is highly undesirable, since it adds latency to Web access. For example, a cache-miss occurring at AN1 generates a packet-in message to the Controller, which unfortunately fails to find a nearby cached content. This will force AN1 to fetch content from the original server, adding one RTT (between the AN and Controller) plus the processing overhead at the Controller.

There is an even worse case: if the Controller has successfully identified nearby content at AN2 for AN1, but AN1 subsequently finds the cached content at AN2 expired, the cost of two RTTs within the access network will be added to the total latency.

We propose a simple solution called “Parallel Attempts” to address these issues. As shown in Fig. 6, once a cache-miss occurs at the AN1, two content retrieval attempts will be generated in parallel, i.e., one towards the Controller to identify nearby content followed by Web request to that cached node, and another one attempting to fetch content from the original server. The AN will accept the response received earlier and discard the late one. This solution ensures that the overall latency incurred will not be higher than the case without a cache.

3.2.4 Hash-table with Lower False Positives

A hash table technique may generate false positives, because of the possible collision of the hash values of different inputs. With the existence of false positive, an item appearing as a member of a data set may not actually belong to it.

In terms of content caching, false positives will lead the Cache Controller to false cache lookups. Particularly, a not-yet-cached content may be reported as cached somewhere by the Controller. This will result in a wasteful redirection to an access node, where the content is considered to be cached, adding at least one RTT, before the content is eventually delivered from the original server. As such, a high false positive rate will outweigh any benefits brought by hash tables.

False positive rate can be expressed as a function of the total number of stored items $N$, the size of the vector $K$ and the number of hash functions being used $n_h$.
\[ \alpha = (1 - \left(1 - \frac{1}{K}\right)^N)^n_h \]  

(3)

Fig. 7 illustrates the false positive rate with diverse table sizes and number of hash functions, based on the analytical results in [9]. The false positive rate can decrease to 5% with a use of a 20000-bit vector and three hash functions, when the number of indexed URIs as high as $10^4\text{ }$. However, false positives increase quickly with the growth of indexed URIs beyond $10^4\text{ }$.

In order to further lower the false positive rate, we trade off the spatial complexity of Hash table, i.e., storing more assisting information while making a lookup decision (Fig.8). Instead of storing only the IP address of access nodes in the linked list, the hash table further records a “fingerprint” of the URI. The fingerprint is a short hash value, e.g., the first 16-bit part of the SHA-1 hash value of the URI. This fingerprint value is inserted to the linked list during the initialization phase. During a lookup into this hash table, the Controller will first compute a hash value of URI and perform a modulo operation as defined in Equation 2 (Step 1 in Fig. 8). If the corresponding slot of the Hash table is greater than zero, the Controller will further compare the fingerprint value sequentially from the linked list, until it finds a slot where the stored value equals to the fingerprint of the URI being searched (Step 2 in Fig. 8). Only if both two operations succeed will the Hash table report a cache hit.

The false positive rate with this low-false-positive Hash table can be analyzed as follows. A false positive exists only if both the modulo operation and the fingerprint calculation have generated a collision. The new false positive rate will be bounded by the minimum of the two possibilities:

\[ \alpha_{LFP} = \min((1 - \left(1 - \frac{1}{K}\right)^N)^n_h, 2^{-L_f}) \]  

(4)

More specifically, with a 16-bit fingerprint the false positive rate of our Hash table will be bounded to as low as $2^{-16}$, irrespective of the number of indexed URIs.

4. PRELIMINARY EVALUATION

We conduct evaluations of the proposed architecture in Mininet [10]. We use a POX module for the implementation of the hash table. Therefore, when local cache table lookup of a URI does not yield a match, the access node will send a packet-in message to the POX. Subsequently, POX uses the Hash table module to examine if and where the URI has been cached. In order to assess the lookup efficiency of Hash table, we emulate a network topology with 200 nodes and a localhost controller node. Each access node sends packet-in messages to the controller, instructing the controller to perform a lookup. We measure only the first RTT, which encompasses the lookup delay at the Controller. We benchmark our Hash-table assisted cache controller against a non-Hash-table counterpart, with cache table sizes ranging from 1K to 9K entries.

According to Fig. 9, the measured lookup delay with Hash table is reduced by 2%–25%, due to the Hash table assisted lookup. Furthermore, the lookup delay with the
hash table is almost constant irrespective of flow table size. In contrast, larger flow tables inflate lookup delay with non-Hash table lookup.

Subsequently, we compare the Page Load Time (PLT) of Edge and Core cache. PLT is perceived as the foremost criterion for web experience, largely reflecting the user's web experience. As reported in [11], the average web page size is 1935 Kb. Assuming that the page can be loaded during the TCP slow start, the number of RTTs required for loading a full page is approximately eight. As such, RTT is a dominant factor of PLT. In this respect, Fig. 10 illustrates the PLT for edge cache and core cache. According to this figure, the Edge cache always yields significantly lower PLT than the Core cache.

We further investigate the benefits brought by the Edge cache compared to core network caching. To this end, we run additional tests in Mininet, with one Mininet host and one Mininet switch emulating a mobile node and wireless access node, respectively. The wireless link bandwidth is set as 2Mbps. Other link parameters, such as delay and packet loss, are configured accordingly to reflect different wireless environments. For core network caching, we further set up another Mininet node as a cache server, with a latency of 30ms from the wireless access node. Similarly, for edge network caching, we set up another host closer to the wireless access node, with a latency ranging from 0ms to 10ms.

We measure the end-to-end TCP throughput between the mobile node and two cache servers. iperf is employed as the measurement tool, with a default TCP receive window size as 85.3KB, and measuring interval set to 10s. The wireless access delay ranges from 10ms to 50ms. Specifically, longer access delay reflects more congested wireless environment. Fig. 11 shows the TCP throughput results in both cases. When the wireless access link delay is within 20ms, the Edge cache does not improve the TCP performance effectively. With the increase of the wireless access delay, the Edge cache yields higher throughput than the Core cache. Therefore, Edge cache is more beneficial for higher wireless link delays.

5. RELATED WORK

There is an extensive literature on Web caching (an overview is given by [12]). From the advent of Web caching, researchers have investigated ways to place content even closer to users. Squirrel [13] is a P2P-based distributed Web caching solution. In Squirrel, Web content is cached at fixed clients and P2P indexing and routing technique is employed for content localization and redirection. Qian et al. [14] further study the possibility of building distributed Web cache on mobile hosts. However, the feasibility of caching content on any forms of terminals (mobile or fixed) has been challenged due to (i) the lack of incentives for collaboration and (ii) the underlying difficulties of managing the content distributed over clients when an unexpected number of them will become dormant and unable to serve the stored content.

In-network caching is another topic that has been investigated intensively [15, 16]. In-network caching is able to utilize the computing and storage capacity of the always-on devices along the path, conserving significant amount of bandwidth. However, in-network caching requires interoperability across network operators, which is hard to attain without a wide-scale deployment of Information-Centric Networking (ICN) or Named Data Networking (NDN).

Since the operator has full control over its edge network devices consisting of eNodeB in LTE and AP in WiFi, we deem caching content at the edge more feasible. In this respect, Erman et al. [17] examine the characteristics of HTTP traffic generated within the world’s largest 3G cellular networks and propose a cost model for the analysis of Web caching at different levels of the network hierarchy. Ramanan et al. [18] study the cacheability of HTTP traffic in an operational LTE network and show that image type contents have the highest revisited rate. This finding can be exploited
in a deployment of our proposed collaborative content caching. SCAP [19] has been proposed for caching content at the access points, using P2P for cache indexing and management. Due to the latency caused by the multihop routing adopted by DHT algorithms, the performance of identifying similar content will be worse than the centralized management scheme (single-hop) proposed in this paper. Extreme-cache [20] enables and manages caching on a single access point, but does not address the scaling issues of managing a vast volume of distributed cached content. In contrast, we present a centralized and efficient approach to enable large-scale Web caching on wireless edge nodes.

6. CONCLUSIONS

In this paper, we presented an architecture for large-scale content caching at the wireless edge. Our approach consists in centralized cached content indexing and request routing, and on-demand caching at wireless access nodes based on content popularity. Cached content indexing is carried out using a low-false-positive Hash table, whereas we developed an OpenFlow-based solution to handle content routing and redirection in a centralized manner. While a system implementation is under way, we have used emulation results to quantify the benefits of our approach.

There are certain aspects in our wireless edge caching architecture that require further investigation, such as (i) the optimization of on-demand caching using different popularity models depending on the type of web object [21], and (ii) the optimization of request redirection, taking into account the latency as well as other characteristics of path quality (e.g., available bandwidth). In future work, we will investigate these aspects and conduct a more extensive performance study of wireless edge caching.

7. REFERENCES


