# An Efficient Delivery Scheme for Coded Caching

Abinesh Ramakrishnan*, Cedric Westphal† and Athina Markopoulou*
*Department of Electrical Engineering and Computer Science, University of California, Irvine
Email : {abinesh.r, athina}@uci.edu
†Huawei Innovation Center, Santa Clara, CA
†Department of Computer Engineering, University of California, Santa Clara
Email : cedric.westphal@huawei.com, cedric@soe.ucsc.edu

***Abstract*—We consider a network with several users trying to access a database of files stored at a server through a shared link. Each user is equipped with a cache, where files can be prefetched according to a caching policy which is mainly based on the popularities of the files. Coded caching tries to exploit coding opportunities created by cooperative caching and has been shown to significantly reduce the load on the shared link. Most of the prior works focused on optimizing the caching policy so as to minimize this expected load. Given the caching policy and the user demands, the problem of minimizing the load over the shared link is essentially an index coding problem. In this paper, we design a novel delivery scheme that builds on a prior scheme for the uniform demand case, but performs better in the non-uniform demand case. We also evaluate this delivery scheme for different caching policies.**

## I. Introduction

Today's Internet traffic is dominated by content distribution services like live-streaming and video-on-demand. Demand for delivery of video content to Internet capable smartphones and other mobile devices has been one of the most driving force behind the explosive growth of traffic in the recent times. Popular services like Netflix, YouTube, etc, exhibit two important features: (i) the user demands are predictable based on their statistical history [5] and (ii) they exhibit strong temporal variability, resulting in highly congested peak hours and underutilized off-peak hours.

A common approach is to take advantage of the memories distributed across the network (at end users and/or inside the network) to store some of the popular contents. This process, termed *caching*, can be done during off-peak hours, so that during peak hours user requests can be served from these locally available memories without having to burden the network. Design and analysis of caching techniques for various kinds of networks have been researched extensively in the past [1], [4], [11], [14], [18], [19] along with the impact of user demand statistics (file popularities) on the performance of caching [5], [7], [23]. This body of prior work only considered uncoded caching. It is easy to see that filling the cache with the most popular files (LFU) is indeed the optimal strategy for a system with only one user [13], but as we move on to systems with requests from multiple users, new challenges and opportunities arise.

A recently suggested method combines caching of files on the user devices with a common multicast transmission of network-coded data [20], [22]. A *coded caching* strategy was proposed by Maddah-Ali *et al.* in [17] for a system of uniform user demands and centralized content placement scheme and was extended to decentralized approach [16] and non-uniform Zipf based user demands in [21]. In [15], Llorca et al. presented a formulation for the general content distribution problem, where nodes in an arbitrary network are allowed to cache, forward, replicate, and code messages in order to deliver arbitrary user demands with minimum overall network cost. In a recent work [10], Ji *et al.* showed order optimality for a caching policy that is uniform or uniform over a subset of the files based on the Zipf parameter for a Zipf distributed file popularity distribution.

Another recently suggested approach is *Femtocaching* [6], which involves deploying a large number of dedicated helper nodes that cache popular file and serve the users' demands locally. In this approach, the caching is done at the helper nodes rather than at the end users. A third recently suggested approach combines caching of files on the user devices with short range device-to-device communications [8], [9]. In this approach, the caches of multiple devices form a common virtual cache that can store a large number of video files, even if the cache on each separate device is not necessarily very large.

In our work we study the coded caching approach, we consider a setup very similar to the one in [17]. The coded caching problem typically consists of two phases, the *placement* phase where the contents are placed in local memory based on the statistics of user demands (file popularities) and the *delivery* phase where the remaining content, which is not available locally, is delivered after the demands of the users have been revealed. In this paper, we focus mainly on the delivery phase. Our main contribution is the design and evaluation of the *Heterogenous Coded Delivery (HCD)* scheme that improves upon the current state-of-the-art in coded caching and significantly reduces the load on the server and the shared link during the delivery phase.

The structure of the rest of the paper is as follows. In Section II, we formulate the problem. In Section III, we discuss the intuition and background of coded caching. In Section IV, we present the proposed heterogenous coded delivery scheme and explain its working in detail. In Section V, we present an evaluation of the proposed scheme. Section VI concludes the paper.

## II. Model and Assumptions

In this paper, we consider a set of users accessing the content server through a shared link as shown in Figure 1. By designing an efficient delivery scheme we can reduce the load on the shared link.

### A. Problem Setting

We consider a system consisting of a server connected through a shared, error-free link to $K$ users as illustrated in Fig.

This work was partially supported by NSF Awards 0747110 (CAREER) and 1028394. A. Ramakrishnan and A. Markopoulou were affiliated with CPCC at UCI. Part of the work was conducted during A. Ramakrishnan's internship at Huawei.

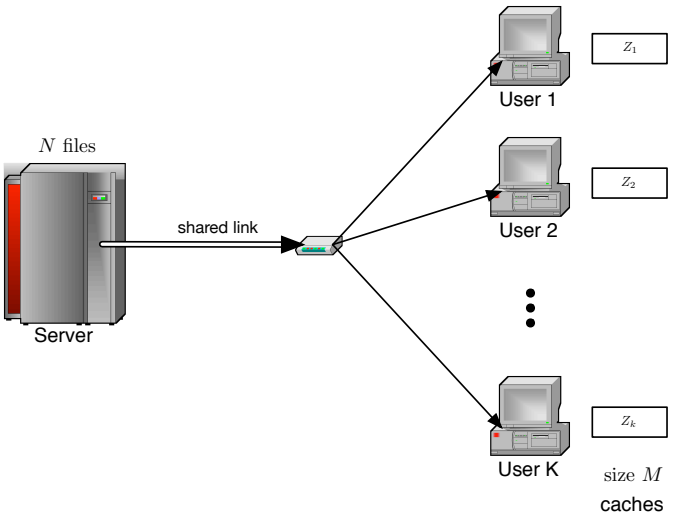


Fig. 1. Problem Setup for Coded Caching.

1. The server has access to a database of $N$ files $W_1, \ldots, W_N$ each of size $F$ bits. Without loss of generality, we assume all files to be of the same size[1]. Each user $k$ has an isolated cache memory $Z_k$ of size $MF$ bits ($MB$ packets) for some real number $M \in [0, N]$. The popularity of a file $W_n$ is the probability that this file is requested by a user. The file popularity distribution in the server is $\mathbf{p} = [p_n]_{n=1}^N$, where $\sum_{n=1}^N p_n = 1$. W.l.o.g. we can assume $p_1 \geq p_2 \geq \cdots \geq p_N$. The system operates in two phases: a *placement phase* and a *delivery phase*.

In the placement phase, the users are given access to the entire database $W_1, \ldots, W_N$ of files. Each user $k$ is then able to fill the content of its cache $Z_k$ using the database. The users follow a caching policy $\mathbf{q} = [q_n]_{n=1}^N$, where $q_n$ denotes the fraction of the cache space in each user that will be allocated to the file $W_n$ and $\sum_{n=1}^N q_n = 1$. The caching can either be done in a centralized manner, where the server besides deciding the caching policy $\mathbf{q}$ also decides what parts of each file is stored in each user's cache; or in a decentralized/distributed approach. In the decentralized approach, the server has no control over which parts of the file goes into each user's cache, it can only control what fraction of each file is cached (*i.e.* the caching policy $\mathbf{q}$). The users randomly cache some portion (the size of this portion alone is dictated by the server) of each file in their corresponding cache. The server is assumed to have complete knowledge of $Z_k$, the content of the cache of user $k$. By knowing the content of cache $Z_k$, the server knows which bits/packets of each file are stored in the cache of user $k$.

In the delivery phase, only the server has access to the database. Each user $k$ requests one of the files $W_{d_k}$ in the database, where $d_k$ represents the index of the file requested by user $k$. The vector $(d_1, \ldots, d_k)$ is a vector of indices of the files requested simultaneously by all $K$ users ordered accordingly. The file requests are independently and identically distributed across all the users and follow the popularity distribution $\mathbf{p}$. The probability that a user requests file $n$ is $p_n$. The server is informed of these requests and proceeds by transmitting a message $X_{(d_1,\ldots,d_k)}$ of size $R_{(d_1,\ldots,d_k)}F$ bits over the shared link for some fixed real number $R_{(d_1,\ldots,d_k)}$. $R_{(d_1,\ldots,d_k)}$ is referred to as the *load* in this paper and is a measure of the length of the message. Using its cache content $Z_k$ and the message $X_{(d_1,\ldots,d_k)}$ received over the shared link, each user $k$ aims to reconstruct its requested file $W_{d_k}$.

[1]Files of different sizes can be split into smaller files of the same size.

### B. Problem Statement

**Problem Statement.** Given the cache content $Z_k$ for each user $k$ and the exact demand vector $(d_1, d_2, \ldots, d_K)$, what is the optimal length $R^*_{(d_1,\ldots,d_k)}F$ of message $X_{(d_1,\ldots,d_k)}$ that the server should transmit to satisfy the given demand?

In general, this is an optimization of the delivery scheme and assumes the content of the caches to be given. As shown in the next subsection, this optimization is in general NP-hard. Here we design a practical delivery scheme for the given cache state that helps to reduce the length of the message $X_{(d_1,\ldots,d_k)}$ compared to the current state of the art scheme in [21].

The expected load is defined as the expectation of the loads over all possible demand vectors, *i.e.* $\bar{R}(\mathbf{p}) = \sum_{(d_1,\ldots,d_K)} R_{(d_1,\ldots,d_K)} p_{d_1} p_{d_2} \cdots p_{d_K}$ and will be used to evaluate how the delivery schemes perform over all demands.

### C. Formulation

In general, the process of various users caching different parts of a given file results in splitting a file into several nonoverlaping subfiles, such that each subfile is present in the caches of a distinct subset of users. Consider a file in the server, say $A$, there are $K$ users in the system and during the placement phase, based on the caching policy, different parts of this file $A$ might get placed at the cache of each user. Grouping the bits of this file based on the set of users they are cached at, the file $A$ can be split into several subfiles $A_{\mathcal{S}}$, where each subfile $A_{\mathcal{S}}$ denotes the group of bits that are stored only in the cache of the specific set of users given by the corresponding $\mathcal{S} \subseteq \{1, \ldots, K\}$. For example, let us say there are $K = 2$ users, then the file $A$ can be possibly split into $A_{\{\}}, A_{\{1\}}.A_{\{2\}}$ and $A_{\{1,2\}}$. Here $A_{\{\}}$ represents all the bits that are cached at neither of the two users, $A_{\{1\}}(A_{\{2\}})$ represents the bits that are cached only at user $1(2)$ and $A_{\{1,2\}}$ represents the bits that are cached at both users 1 and 2. Note that these subfiles are non-overlapping, *i.e.* if a bit of the file is present in one subfile it cannot be present in any other subfile. Also note that not all subfiles have to be populated, in the example above if not even a single bit of file $A$ is cached at both user 1 and 2 then $A_{\{1,2\}}$ is unpopulated and hence can be discarded. The cache $Z_k$ of the user $k$ can be thought of as a collection of such subfiles that are cached at this user $k$. The subfiles can in turn be classified into types based on the number of users they are cached at. We say *a subfile is of type $t$* if exactly $t$ users have cached the bits of this subfile. It is easy to see that there are $K + 1$ types of subfiles and there are $\binom{K}{t}$ subfiles for each type $t$.

In a centralized approach, the server has complete control over which bit of each file gets placed in the cache of each user. This basically means that the server can determine how a file is split in the subfiles defined above. In the decentralized

approach, since the users randomly choose the bits they will cache, the splitting is also randomized.

Now consider a demand vector $(d_1, d_2, \ldots, d_K)$, which indicates that user 1 demands file $W_{d_1}$, user 2 requests file $W_{d_2}$, and so on. To simplify notation, let $V_k$ denote the file requested by user $k$, *i.e.* $V_k = W_{d_k}$ and $V_{k,\mathcal{S}}$ denote the subfiles corresponding to the file $W_{d_k}$ that are requested by user $k$ and are only available in the cache of users in $\mathcal{S}$. The notation $\mathcal{S}$ is used to refer to some ordered set of users, $\mathcal{S} \subseteq \{1, 2, \ldots, K\}$. For a given demand, based on the cache content across all users, the server creates subfiles of the form $V_{k,\mathcal{S}}$, for each user $k$, which will be used for transmission. The user $k$ already knows some subfiles of the file $V_k$, as these subfiles are already present in its cache $Z_k$; so the server only needs to transmit the subfiles that are not present in its cache $Z_k$ to satisfy user $k$'s demand, *i.e.* the server needs to send all populated subfiles of the form $V_{k,\mathcal{S}\setminus\{k\}}$ $\forall \mathcal{S}$ , for that particular user $k$. Note that even if two users, $i$ and $j$, request the same subfile, say $A_{\mathcal{S}}$, we will use separate notations $V_{i,\mathcal{S}}$ and $V_{j,\mathcal{S}}$ to denote the subfile requested by the respective users.

For a given caching and demand vector, finding the delivery scheme that minimizes the code length of the message is equivalent to solving an index coding problem [2], [3] whose side information graph is determined by the caching configuration. The side information graph $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the set of vertices and $\mathcal{E}$ the set of (directed) edges, can be constructed as follows

- Each bit requested by each user is represented as a vertex. More specificially, every vertex $v \in \mathcal{V}$ corresponds to a bit in a subfile of the form $V_{k,\mathcal{S}\setminus\{k\}}$ that is requested by user $k$ and is cached only at all the users in the set $\mathcal{S} \setminus \{k\}$. If two users request the same bit of a file, then that bit is represented by two distinct vertices. Note that the notation $V_{k,\mathcal{S}\setminus\{k\}}$, when used in context of the side information graph, denotes the group of vertices that represent the bits in the subfile $V_{k,\mathcal{S}\setminus\{k\}}$.
- There exists an edge $(u, v) \in \mathcal{E}$ iff the cache of the user requesting the bit represented by $u$ contains the bit represented by $v$.

It is well-known that the index coding problem is, in general, NP-hard [12]. One can use the chromatic number based approach to get a sub-optimal solution [10] by constructing an undirected graph $G_a = (\mathcal{V}, \mathcal{E}_a)$ similar to $G$. The vertices of $G_a$ are the same as the vertices of $G$ and there exists an undirected edge $(u, v) \in \mathcal{E}_a$ between two vertices $u$ and $v$ iff the cache of the user requesting the bit represented by $u$ contains the bit represented by $v$ and the cache of the user requesting the bit represented by $v$ also contains the bit represented by $u$. The chromatic number solution for the complement graph of $G_a$, which is equivalent to the minimum clique cover of the graph $G_a$, will give a sub-optimal approach for the delivery scheme in our problem, but finding the chromatic number is also known to be NP-hard, in general.

## III. Background on Coded Caching

In this section, we present background on coded caching and a rather detailed overview of prior work, including observations and insights that inspired our new scheme.

### A. Uniform Demands

We start from the intuition of the caching policy and the delivery scheme for the uniform demands scenario, considered in [17]. The $N$ files in the database of the server are assumed to have equal popularity here, *i.e.* the probability that a given file is requested by a user is $1/N$. Due to the uniform popularity, it only makes sense that an equal portion of each file be cached at the user caches and since each user has a cache of size $M$ files, some $M/N$ portion of each file should be cached at every user. In other words, we choose a caching policy where $q_n = M/N$ for all $n$. Also note that, [17] considers a centralized placement scheme, where the server can determine which bits of each file gets cached at each user.

Let us consider a file $A$. The caching policy dictates that each user caches $MF/N$ bits of this file. In order to determine which bits get placed at which user's cache, the server splits this file $A$ into $\binom{K}{t}$ subfiles of type $t$ each labeled as $A_{\mathcal{S}}$, for some $\mathcal{S}$. The subset $\mathcal{S}$ comprises of $t$ users and it is easy to see that there are $\binom{K}{t}$ such subsets for a system with $K$ users. Since the content of the subfiles are nonoverlapping, each subfile will contain $F/\binom{K}{t}$ bits of the file $A$. A user $k$ must cache all the subfiles $A_{\mathcal{S}}$ , where $k \in \mathcal{S}$. There are $\binom{K-1}{t-1}$ such subfiles for each user and the total bits in them should add up to $MF/N$ bits for each user.

$$\frac{\binom{K-1}{t-1}F}{\binom{K}{t}} = \frac{MF}{N} \Rightarrow t = \frac{M}{N}K$$

Thus the server will split each file into subfiles of type $t = MK/N$ in order to help the users determine which bits they will be storing in their respective caches.

Designing an optimal delivery scheme for a given demand vector is equivalent to solving an index coding problem. Due to the NP-hardness of this problem, practical solutions consider suboptimal solutions. The centralized placement scheme, described above, coupled with the uniform caching policy introduces a symmetry in the system that makes it easier to find the minimum clique cover (chromatic number) solution for any given demand vector. The undirected graph $G_a = (\mathcal{V}, \mathcal{E}_a)$ can be constructed for a given demand vector as explained in section II-C. For every user, there are $\binom{K-1}{t}$, subfiles each with $F/\binom{K}{t}$ bits that are not cached at that particular user. So the server needs to transmit a total of $K\binom{K-1}{t}F/\binom{K}{t}$ bits to satisfy the demand, *i.e.* $|V| = K\binom{K-1}{t}F/\binom{K}{t} = K(1-M/N)F$. An uncoded delivery scheme would require the server to send all those bits one by one (in the worst case scenario, when each user demands a different file) to satisfy the demand. The minimum clique cover provides a coding mechanism which would significantly reduce the number of bits that the server needs to transmit. Consider a subset $\mathcal{S}$ of $t+1$ users, for each $k \in \mathcal{S}$ consider a node in $G_a$ that represents a bit in the subfile $V_{k,\mathcal{S}\setminus\{k\}}$. These $t+1$ nodes can form a clique of size $t+1$. By coding (XORing) together all the $t+1$ bits represented by these nodes and transmitting them to all users in $\mathcal{S}$, each user in $\mathcal{S}$ will be able to decode its desired bits using the information stored in its own cache. Note that $F/\binom{K}{t}$ cliques are required to cover all the bits in a subfile and each clique covers a bit in $t+1$ subfiles. Each clique here is a

maximal clique and subsequently it is not hard to see that the cover indeed uses a minimum number of cliques. This whole process of clique cover based coding can be thought of bitwise coding (XOR) the subfiles of the form $V_{k,\mathcal{S}\setminus\{k\}}\ \forall\ k \in \mathcal{S}$ and repeating this for all $\mathcal{S}$ of size $t+1$. In the end there will be $F/\binom{K}{t}$ cliques for each subset $\mathcal{S}$ of size $t+1$ and $\binom{K}{t+1}$ such subsets, so the server only needs to send a message of size $\binom{K}{t+1}F/\binom{K}{t}$ bits to satisfy all the demands. The transmitted message length $R_{(d_1,\ldots,d_k)}F = \binom{K}{t+1}F/\binom{K}{t}$ remains the same across all demand vectors because of the symmetry and the uniformity in the placement and delivery schemes. For easier notation, let us denote this load $R_{(d_1,\ldots,d_k)}$ as just $R$. In [17], $R$ is shown to be information theoretically optimal.

$$R = \binom{K}{t+1}\frac{1}{\binom{K}{t}} = \frac{K-t}{t+1} = \frac{K(1-M/N)}{1+KM/N} \qquad (1)$$

In the decentralized approach for content placement, the server has no control over which bits are cached at each user, so it is not possible to look at the files as a collection of subfiles of a single type. The users randomly select $MF/N$ bits to store in their cache and based on this random placement of bits, each file can be seen as a collection of subfiles of various types. An algorithm was provided in [16] for the delivery in the decentralized setup, which was still based on the clique cover approach, but the clique cover solution yielded by this algorithm is not necessarily the minimum.

*Insight.* More importantly, the algorithm in [16] was restricted to form cliques only between nodes (i.e., code the corresponding bits together) in the subfiles of the same type, and thus missed coding opportunities We build upon this observation and design a new delivery scheme, which considers more coding opportunities by forming cliques between nodes not only of the same, but also of different type.

### B. Non-Uniform Demands

Although uniform demands facilitate analysis, file popularities are far from uniform in practice; in fact, they could vary several orders of magnitude. In the uniform case, each file has the same probability of being requested by a user, thus it is natural to allocate equal cache to each file at every user. In the non-uniform case, the least popular file almost never get requested by users. Therefore, cache should not be allocated equally to highly popular files and least popular files. An intuitive way to share the cache is to make the caching policy $\mathbf{q}$ follow the popularity distribution $\mathbf{p}$.

The work in [21] considers Zipf-distributed file popularities and they propose a caching policy that groups files together. The files are divided into groups based on the closeness in their popularities and the caching policy $q$ is designed such that all the files in the same group will have the same amount of cache space, $q_n$, which is determined by dividing the cumulative popularity of all the files in the corresponding group by the total number of files in that group. However, files in different groups will have different $q_n$ (cache space allocation). They also explicitly state that the grouping can be optimized to minimize the expected load.

The work in [10] considers Zipf file popularities, but proposes a simpler and different caching policy than the one in [21]. [10] divides the files into just two groups. The files in one group are not allocated any cache space at all, *i.e.* $q_n = 0$ for all the files in this group, whereas the files in the other group get to divide the entire cache space equally among themselves ($q_n = M/group\ size$, for all files in this group). Both [21] and [10] assume a decentralized placement scheme, because a centralized approach would require a lot of work from the server, which is not as practical as the decentralized approach.

Unlike the uniform case, the load $R_{(d_1,\ldots,d_k)}$ varies across all demand vectors, thus it is more meaningful to consider the expected load $\bar{R}(\mathbf{p})$ instead. The delivery scheme used in [21] is essentially the same as the one in [16], as discussed briefly in section III-A. The key idea behind the scheme in [21] is to code (bitwise XOR) together all the subfiles of the form $V_{k,\mathcal{S}\setminus\{k\}}\ \forall\ k \in \mathcal{S}$ and repeating this coding procedure for all valid $\mathcal{S}$. If the size of these subfiles is not the same, which is usually the case when employing non-uniform caching policy and/or decentralized placement scheme, the scheme just pads them with zeros to make all their sizes the same.

Considering a graph $G_a$ constructed based on the cache content of the users, for each $\mathcal{S}$, the algorithm tries to cover all the nodes in the groups of the form $V_{k,\mathcal{S}\setminus\{k\}}\ \forall k \in \mathcal{S}$, by trying to form cliques of maximum size $|\mathcal{S}\setminus\{k\}|$ between the nodes across the groups $V_{k,\mathcal{S}\setminus\{k\}}\ \forall k \in \mathcal{S}$. If it cannot find any uncovered node within the groups $V_{k,\mathcal{S}\setminus\{k\}}\ \forall k \in \mathcal{S}$ to form a clique of size $|\mathcal{S}\setminus\{k\}|$, then the algorithm simply chooses to form a clique of lower size with the available nodes.

*Insight.* In the situation just described above, the algorithm unnecessarily restricts itself from considering all coding opportunities: it could form a bigger clique with nodes from a different group of the form $V_{k,\mathcal{S}'\setminus\{k\}}$, where $\mathcal{S} \subset \mathcal{S}'$. This is the key point we will exploit in the improved algorithm that we present in the next section. The delivery scheme in [10] makes use of the minimum clique cover solution, but finding the minimum clique cover is NP-hard and [10] does not provide any practical algorithm to do that efficiently. Both [21] and [10], prove that their respective placement and delivery scheme combinations are indeed order optimal. In particular, [21], chooses a specific grouping, where files with popularity differing by at most a factor of two are grouped together, to prove the order optimality of their approach.

## IV. Heterogenous Coded Delivery

In this section, we propose the Heterogenous Coded Delivery scheme (HCD). First, we define the algorithm and we discuss the core ideas and intuition on how this scheme achieves better performance than the state-of-the-art in [16] and [21]. Then, we walk through the details of this new scheme through an illustrative example.

### A. The Scheme

The pseudocode for the HCD scheme is presented in Algorithm 1. HCD can be used with both centralized and decentralized placement approaches. The new scheme, similar to the ones in [16], [21], is still based on finding a clique cover for the graph $G_a$. But HCD exploits the possibilities of forming cliques with nodes from subfiles of higher types, which could potentially reduce the number of cliques required for the cover. The users have already populated the cache

based on some caching policy and placement scheme. Once the server is informed of the user requests, *i.e.* the demand vector $(d_1, \ldots, d_K)$, it would be able to format the subfiles $V_k$ that are required for transmission.

**Algorithm 1** Heterogenous Coded Delivery (HCD)

**procedure** DELIVERY($d_1, \ldots, d_K$)
  **for** $t = 1, 2, \ldots, K-1, K$ **do**
    **for** $\mathcal{S} \subseteq [K] : |\mathcal{S}| = t$ **do**
      $binsize = \max_{k \in \mathcal{S}} V_{k,\mathcal{S}\setminus\{k\}}$
      **for** $k \in \mathcal{S}$ **do**
        **if** $|V_{k,\mathcal{S}\setminus\{k\}}| < binsize$ **then**
          Move $(binsize - |V_{k,\mathcal{S}\setminus\{k\}}|)$ bits from non-empty bins $V_{k,\mathcal{S}'\setminus\{k\}} : \mathcal{S}' \supset \mathcal{S}$ to $temp$
          Create new subfile $V'_{k,\mathcal{S}\setminus\{k\}}$
          $V'_{k,\mathcal{S}\setminus\{k\}} \leftarrow V_{k,\mathcal{S}\setminus\{k\}} + temp$
          $coded \leftarrow coded \oplus V'_{k,\mathcal{S}\setminus\{k\}}$
        **else**
          $coded \leftarrow coded \oplus V_{k,\mathcal{S}\setminus\{k\}}$
        **end if**
      **end for**
      server transmits subfile $coded$
    **end for**
  **end for**
**end procedure**

The procedure DELIVERY is called for the given demand vector to determine the message $X_{(d_1,\ldots,d_k)}$ that needs to be transmitted to satisfy the demands. The scheme requires to iterate over all possible subsets of users, $\mathcal{S}$, and the two outermost loops in the algorithm help to do that. Note that in Algorithm 1, we use the notation $[K]$ to refer to the set $\{1, 2, \ldots, K\}$, the operator $+$ refers to concatenation and $\oplus$ refers to bitwise XOR operation. The innermost loop helps to iterate through each user within a given subset $\mathcal{S}$. The steps within the inner most loop are the core of the algorithm and the difference between our algorithm and those presented in [16], [21]. Recall that the algorithms in [16], [21] code together all the subfiles of the form $V_{k,\mathcal{S}\setminus\{k\}}\ \forall\ k \in \mathcal{S}$, after appending zeros to make them all of the same size (equal number of bits). In our algorithm, instead of appending zeros right away, we first try to borrow bits from the *immediate* higher type subfiles of the form $V_{k,\mathcal{S}'\setminus\{k\}}$, where $\mathcal{S}' \supset \mathcal{S}$. After exhausting all the options for borrowing, we append zeros. In the algorithm, this borrowing process is accompanied by a step involving the creation of new subfile $V'_{k,\mathcal{S}\setminus\{k\}}$. We do this to avoid conflict with the definition of $V_{k,\mathcal{S}'\setminus\{k\}}$. Note that there are bits in the new subfile that, although present in the caches of users in $\mathcal{S}\setminus\{k\}$, are no longer cached *only* at these user in $\mathcal{S}\setminus\{k\}$. Since $\mathcal{S}'$ can be any superset of $\mathcal{S}$, all the bits in the new subfile $V'_{k,\mathcal{S}\setminus\{k\}}$ are cached at all the users in $\mathcal{S}\setminus\{k\}$ and so they would still be able to decode their respective bits using the information present in their cache. Also note that the borrowing process actually involves moving bits from the original subfile, not just copying them. It is important to first code and transmit the subfiles of lower type before moving on to higher types, because a subfile can only borrow bits from the corresponding subfiles of higher types.

In graph theoretic terms, we consider a graph $G_a$ constructed based on the cache content of the users, as in [16], [21]. For each $\mathcal{S}$ the algorithm tries to cover all the nodes in the groups of the form $V_{k,\mathcal{S}\setminus\{k\}}\ \forall k \in \mathcal{S}$, by trying to form cliques of maximum size $|\mathcal{S}\setminus\{k\}|$ between the nodes across the groups $V_{k,\mathcal{S}\setminus\{k\}}\ \forall k \in \mathcal{S}$. Our algorithm differs in the fact that, if it cannot find any uncovered node within the groups $V_{k,\mathcal{S}\setminus\{k\}}\ \forall k \in \mathcal{S}$ to form a clique of size $|\mathcal{S}\setminus\{k\}|$, instead of just settling with forming a clique of lower size, our algorithm tries to cover the nodes from the corresponding higher type groups $V_{k,\mathcal{S}'\setminus\{k\}}$, where $\mathcal{S}' \supset \mathcal{S}$. We would like to point out that, since our algorithm tries to cover nodes from higher type groups whenever possible, even in the worst-case scenario the performance of our algorithm will be at least as good as the ones in [16], [21].

| | |
|---|---|
| $W_1$ | $p_1 = 0.3$ |
| $W_2$ | $p_2 = 0.2$ |
| $W_3$ | $p_3 = 0.2$ |
| $W_4$ | $p_4 = 0.2$ |
| $W_5$ | $p_5 = 0.1$ |

Fig. 2. File Server with $N = 5$ and popularity distribution $\mathbf{p}$.

Observe that we do not try to optimize the borrowing step, *i.e.* we do not try to determine the subfiles to borrow bits from such that final clique is minimized. The optimization of this step is the core of the minimum clique cover and so could be NP-hard. Instead, we choose to go with a simpler and more practical approach wherein a subfile $V_{k,\mathcal{S}\setminus\{k\}}$, if required, will first borrow from a valid *immediate* higher type subfile of the form $V_{k,\mathcal{S}'\setminus\{k\}}$, where $\mathcal{S}' \supset \mathcal{S}$. For example, consider an iteration where we are trying to code together the subfiles $V_{1,\{2\}}$, which has 5 bits, and $V_{2,\{1\}}$, which only has 1 bit, and the other subfiles related to user 2 present in the system are $V_{2,\{1,3,4\}}, V_{2,\{1,4,5\}}$ and $V_{2,\{1,3,4,5\}}$ with 2 bits each. The algorithm will first try to borrow from the immediate higher type subfiles, which in this case are $V_{2,\{1,3,4\}}$ and $V_{2,\{1,4,5\}}$. The next higher type subfile $V_{2,\{1,3,4,5\}}$ will only be considered if $V_{2,\{1,3,4\}}$ and $V_{2,\{1,4,5\}}$ do not have enough bits to borrow from.

### B. Example

We now present a detailed walkthrough of our algorithm through an illustrative example. This will help highlight the subtleties in the proposed scheme.

*Example 1.* Consider a system with a server consisting of $N = 5$ files, each of size $F$ bits, $K = 5$ users, each with a cache of size $M = 2$ files and popularity distribution $p$ as shown in fig. 2. The file popularity distribution takes three distinct values: $p_1 = 0.3$, $p_2 = p_3 = p_4 = 0.2$ and $p_5 = 0.1$.

For this example, we will consider a caching policy that follows exactly the popularity distribution, *i.e.* $\mathbf{q} = \mathbf{p}$, and a centralized placement scheme. We choose the centralized placement scheme as it helps to highlight the key difference between our algorithm and the one in [16], [21]. The centralized scheme for the non-uniform caching policy is quite similar to the one described in section III-A. The server splits a file $W_i$ into $\binom{K}{t_i}$ subfiles of type $t_i = p_i MK$, each getting $F/\binom{K}{t_i}$ bits as shown in fig. 3. The server splits the file $W_1$ into $\binom{5}{3} = 10$ subfiles of type $t_1 = p_1 MK = 3$, namely $W_{1,\{1,2,3\}}$,

TABLE I.

| User $i$ | Subfiles requested but not cached at user $i$ | Size of each bin |
|---|---|---|
| 1 | $V_{1,\{2,3,4\}}, V_{1,\{2,3,5\}}, V_{1,\{2,4,5\}}, V_{1,\{3,4,5\}}$ | $F/10$ |
| 2 | $V_{2,\{1,3\}}, V_{2,\{1,4\}}, V_{2,\{1,5\}}, V_{2,\{3,4\}}, V_{2,\{3,5\}}, V_{2,\{4,5\}}$ | $F/10$ |
| 3 | $V_{3,\{1,2\}}, V_{3,\{1,4\}}, V_{3,\{1,5\}}, V_{3,\{2,4\}}, V_{3,\{2,5\}}, V_{3,\{4,5\}}$ | $F/10$ |
| 4 | $V_{4,\{1,2\}}, V_{4,\{1,3\}}, V_{4,\{1,5\}}, V_{4,\{2,3\}}, V_{4,\{2,5\}}, V_{4,\{3,5\}}$ | $F/10$ |
| 5 | $V_{5,\{1\}}, V_{5,\{2\}}, V_{5,\{3\}}, V_{5,\{4\}}$ | $F/5$ |

$W_{1,\{1,2,4\}}$, $W_{1,\{1,2,5\}}$, $W_{1,\{1,3,4\}}$, $W_{1,\{1,3,5\}}$, $W_{1,\{1,4,5\}}$, $W_{1,\{2,3,4\}}$, $W_{1,\{2,3,5\}}$, $W_{1,\{2,4,5\}}$ and $W_{1,\{3,4,5\}}$, each with $F/10$ bits. Similarly, each of the files $(W_2, W_3, W_4)$ with popularity 0.2, will be split into $\binom{5}{2} = 10$ subfiles of type 2 ($t_2 = t_3 = t_4 = 0.2MK = 2$), each with $F/10$ bits. For instance, the file $W_2$ will be split into the following subfiles: $W_{2,\{1,2\}}$, $W_{2,\{1,3\}}$, $W_{2,\{1,4\}}$, $W_{2,\{1,5\}}$, $W_{2,\{2,3\}}$, $W_{2,\{2,4\}}$, $W_{2,\{2,5\}}$, $W_{2,\{3,4\}}$, $W_{2,\{3,5\}}$ and $W_{2,\{4,5\}}$. Finally, the file $W_5$ will be split into $\binom{5}{1} = 5$ subfiles of type $t_5 = 0.1MK = 1$, each with $F/5$ bits, namely $W_{5,\{1\}}, W_{5,\{2\}}, W_{5,\{3\}}, W_{5,\{4\}}$ and $W_{5,\{5\}}$. A user $i$ will store in its cache all the subfiles of the form $W_{k,\{\mathcal{S}:i\in\mathcal{S}\}}\ \forall\ k$, *i.e.* it will store all the subfiles with a label of the form $W_{k,\mathcal{S}}$, where $i \in \mathcal{S}$. Observe that now each user will have cached $p_iMF$ bits of the file $W_i$ and since $\sum p_i = 1$, the users end up with their caches fully occupied.

We will first consider the demand vector $(W_1, W_2, W_3, W_4, W_5)$ to see the improvements this new scheme offers. After the server is informed about the demand vector, it would be able to create and populate the subfiles $V_{k,\mathcal{S}}$ using the corresponding $W_{d_k,\mathcal{S}}$. User 1 demands the file $W_1$. It already has the subfiles $V_{1,\{1,2,3\}}$, $V_{1,\{1,2,4\}}$, $V_{1,\{1,2,5\}}$, $V_{1,\{1,3,4\}}$, $V_{1,\{1,3,5\}}$ and $V_{1,\{1,4,5\}}$ in its cache, so the server only needs to send the remaining subfiles $V_{1,\{2,3,4\}}$, $V_{1,\{2,3,5\}}$, $V_{1,\{2,4,5\}}$ and $W_{1,\{3,4,5\}}$. Table I shows the subfiles, along with the number of bits in each, that the server has to send for each user.

The algorithm goes through all the subfiles in Table I, starting with the lower type ones. In this example the lowest type subfiles are $V_{5,\{1\}}, V_{5,\{2\}}, V_{5,\{3\}}$ and $V_{5,\{4\}}$, each containing $F/5$ bits of the file $V_i$ that is not cached at user 5. If there is a subfile $V_{1,\{5\}}$ containing bits demanded by user 1 and cached only at user 5, we will be able to code (bitwise XOR) $V_{5,\{1\}}$ with $V_{1,\{5\}}$. But such a subfile is not present, so we create a new subfile $V'_{1,\{5\}}$ and populate it with bits borrowed from $V_{1,\{2,3,5\}}$, $V_{1,\{2,4,5\}}$ and $V_{1,\{3,4,5\}}$, which we refer to as the donor subfiles. $V_{1,\{5\}}$ has $F/5$, so we borrow $F/15$ bits from each of the three donor files reducing their size to $F/30$ bits. Note that the bits in these donor subfiles are all demanded by user 1 and are cached at user 5 (and also at few other users), so by transmitting the $F/5$ bits obtained by coding $V_{5,\{1\}}$ with $V'_{1,\{5\}}$ user 1 would still be able to recover its requested bits and at the same time user 5 would also be able to receive and recover some of its requested bits. This process is repeated for all the remaining subfiles type 1 as shown in table II and fig. 4.

Moving on to the type 2 subfiles, we can see that there are 18 subfiles under this type. Originally, all of these subfiles had $F/10$ bits each, but the borrowing steps shown in table II has reduced the size of some of them to $F/30$ bits. Consider the subfiles $V_{2,\{3,4\}}$, $V_{3,\{2,4\}}$ and $V_{4,\{2,3\}}$, none of them were used in table II and so these subfiles can be coded together and transmitted as a single subfile of size $F/10$ bits.

TABLE II.

| Subfiles | Size | Donor subfiles : Bits left (before) | Bits taken | Bits left (after) |
|---|---|---|---|---|
| $V'_{1,\{5\}}$ | $\frac{F}{5}$ | $V_{1,\{2,3,5\}} : \frac{F}{10}$<br>$V_{1,\{2,4,5\}} : \frac{F}{10}$<br>$V_{1,\{3,4,5\}} : \frac{F}{10}$ | $\frac{F}{15}$<br>$\frac{F}{15}$<br>$\frac{F}{15}$ | $\frac{F}{30}$<br>$\frac{F}{30}$<br>$\frac{F}{30}$ |
| $V'_{2,\{5\}}$ | $\frac{F}{5}$ | $V_{2,\{1,5\}} : \frac{F}{10}$<br>$V_{2,\{3,5\}} : \frac{F}{10}$<br>$V_{2,\{4,5\}} : \frac{F}{10}$ | $\frac{F}{15}$<br>$\frac{F}{15}$<br>$\frac{F}{15}$ | $\frac{F}{30}$<br>$\frac{F}{30}$<br>$\frac{F}{30}$ |
| $V'_{3,\{5\}}$ | $\frac{F}{5}$ | $V_{3,\{1,5\}} : \frac{F}{10}$<br>$V_{3,\{2,5\}} : \frac{F}{10}$<br>$V_{3,\{4,5\}} : \frac{F}{10}$ | $\frac{F}{15}$<br>$\frac{F}{15}$<br>$\frac{F}{15}$ | $\frac{F}{30}$<br>$\frac{F}{30}$<br>$\frac{F}{30}$ |
| $V'_{4,\{5\}}$ | $\frac{F}{5}$ | $V_{4,\{1,5\}} : \frac{F}{10}$<br>$V_{4,\{2,5\}} : \frac{F}{10}$<br>$V_{4,\{3,5\}} : \frac{F}{10}$ | $\frac{F}{15}$<br>$\frac{F}{15}$<br>$\frac{F}{15}$ | $\frac{F}{30}$<br>$\frac{F}{30}$<br>$\frac{F}{30}$ |

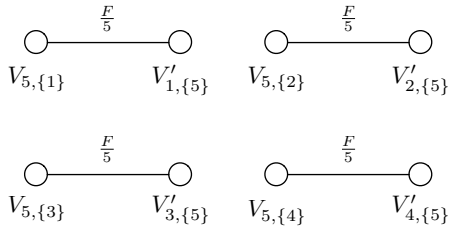


Fig. 4. Cliques of size 2

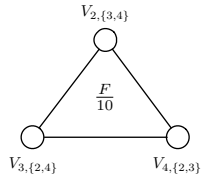


Fig. 5. Cliques of size 3, no bits borrowed.

TABLE III.

| Subfiles | Size | Donor subfiles : Bits left (before) | Bits taken | Bits left (after) |
|---|---|---|---|---|
| $V'_{1,\{2,5\}}$ | $\frac{F}{30}$ | $V_{1,\{2,3,5\}} : \frac{F}{30}$<br>$V_{1,\{2,4,5\}} : \frac{F}{30}$ | $\frac{F}{60}$<br>$\frac{F}{60}$ | $\frac{F}{60}$<br>$\frac{F}{60}$ |
| $V'_{1,\{3,4\}}$ | $\frac{F}{30}$ | $V_{1,\{2,3,5\}} : \frac{F}{60}$<br>$V_{1,\{3,4,5\}} : \frac{F}{30}$ | $\frac{F}{60}$<br>$\frac{F}{60}$ | 0<br>$\frac{F}{60}$ |
| $V'_{1,\{4,5\}}$ | $\frac{F}{30}$ | $V_{1,\{2,4,5\}} : \frac{F}{60}$<br>$V_{1,\{3,4,5\}} : \frac{F}{60}$ | $\frac{F}{60}$<br>$\frac{F}{60}$ | 0<br>0 |

Next, consider the subfile $V_{2,\{1,5\}}$, it could be potentially coded together with bins $V_{1,\{2,5\}}$ and $V_{5,\{1,2\}}$ if they were present. We can create new subfile $V'_{1,\{2,5\}}$ and fill it up with $F/30$ bits from donors $V_{1,\{2,3,5\}}$ and $V_{1,\{2,4,5\}}$, but there is no point in creating a subfile $V_{5,\{1,2\}}$ as it does not have any donor subfiles of higher types to borrow from. $V_{3,\{1,5\}}$ and $V_{4,\{1,5\}}$ encounter a similar situation and the entire process for those subfiles can be seen in table III and fig. 6.

Consider subfile $V_{2,\{3,5\}}$, it could be potentially coded with $V_{3,\{2,5\}}$ and $V_{5,\{2,3\}}$, one of which is already present. As in the

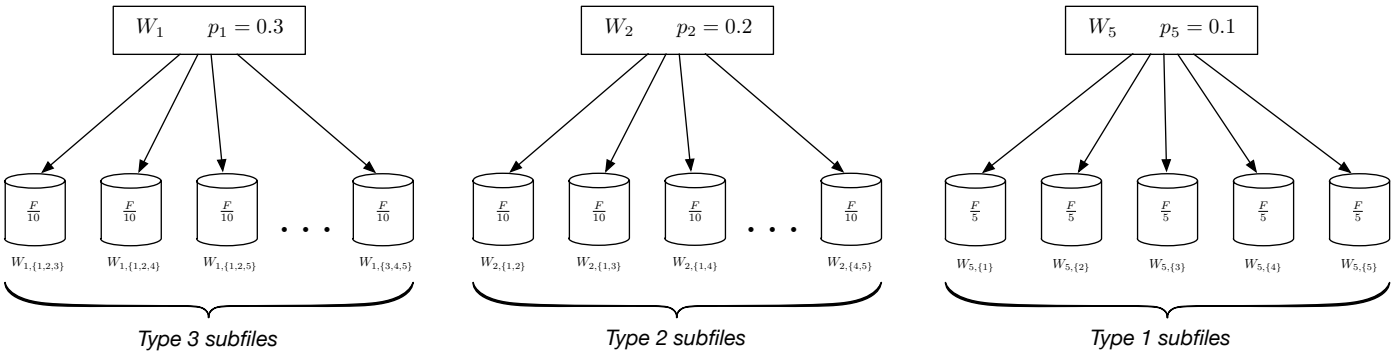


Fig. 3. Placement Phase

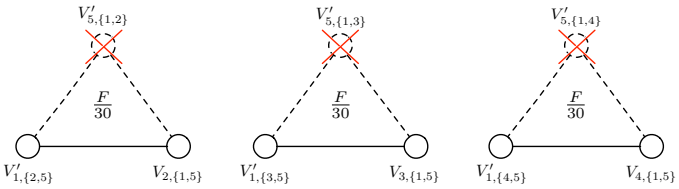


Fig. 6. Cliques of size 3 that are reduced to cliques of size 2, bits borrowed.

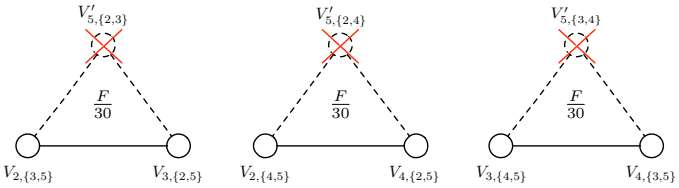


Fig. 7. Cliques of size 3 that are reduced to cliques of size 2, no bits borrowed.

previous case, it does not make sense to create a new subfile $V'_{5,\{2,3\}}$ as it does not have any higher type donor subfiles to borrow from. Both $V_{2,\{3,5\}}$ and $V_{3,\{2,5\}}$ have $F/30$ bits as seen in table II each, so we could code these two together and transmit a single subfile of size $F/30$ bits. This, along with a similar process for $V_{2,\{4,5\}}$, $V_{4,\{2,5\}}$ , $V_{3,\{4,5\}}$ and $V_{4,\{3,5\}}$, are shown in fig. 7.

There are still a few more subfiles of type 2 that have not been transmitted yet, namely $V_{2,\{1,3\}}$, $V_{3,\{1,2\}}$, $V_{2,\{1,4\}}$, $V_{4,\{1,2\}}$, $V_{3,\{1,4\}}$ and $V_{4,\{1,3\}}$. None of them were used as a donor so far and so each of them still has $F/10$ bits. $V_{2,\{1,3\}}$ and $V_{3,\{1,2\}}$ can be coded together with $V_{1,\{2,3\}}$, which is not present. So we create a new subfile $V'_{1,\{2,3\}}$ and try to fill it up with bits from donor subfile. We can see from Table IV that there is only one non-empty donor subfile that we could borrow from and we will borrow all $F/10$ bits from it. For the other subfiles, we are left with no donor subfiles to borrow from and hence it does not make sense to create new subfiles. The process is illustrated in table IV and fig. 8.

In total, we have transmitted 4 coded subfiles of size $F/5$ bits (fig. 4), 4 coded subfiles of size $F/10$ bits (figs. 5 and 8) and 6 coded subfiles of size $F/30$ bits. Therefore, the total length of the message that was transmitted to satisfy the demands vector $(W_1, W_2, W_3, W_4, W_5)$ is $4 \times \frac{F}{5} + 6 \times \frac{F}{30} +$

TABLE IV.

| Subfiles | Size | Donor subfiles : Bits left (before) | Bits taken from each bin | Bits left (after) |
|---|---|---|---|---|
| $V'_{1,\{2,3\}}$ | $\frac{F}{10}$ | $V_{1,\{2,3,4\}} : \frac{F}{10}$<br>$V_{1,\{2,3,5\}} : 0$ | $\frac{F}{10}$<br>0 | 0<br>0 |
| $V'_{1,\{2,4\}}$ | $\frac{F}{10}$ | $V_{1,\{2,3,4\}} : 0$<br>$V_{1,\{2,4,5\}} : 0$ | 0<br>0 | 0<br>0 |
| $V'_{1,\{3,4\}}$ | $\frac{F}{10}$ | $V_{1,\{2,3,4\}} : 0$<br>$V_{1,\{3,4,5\}} : 0$ | 0<br>0 | 0<br>0 |

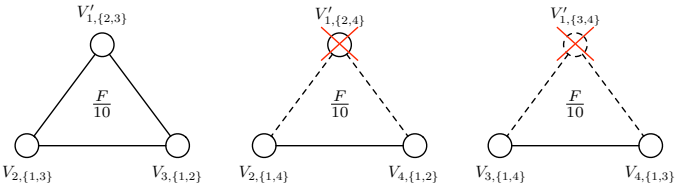


Fig. 8. Cliques of size 3, some are reduced to size 2.

$4 \times \frac{F}{10} = 1.4F$ bits

*Baseline.* The scheme in [16], [21] does not borrow bits from higher type subfiles during the coding process, so their scheme will end up transmitting 4 subfiles of size $F/5$ bits for user 5's request, 10 coded subfiles of size $F/10$ bits for user $2, 3$ and 4's requests and 4 more subfiles of size $F/10$ for user 1's request. This sums up to a total of $2.2F$ bits, which is considerably more than the length of our scheme.

## V. Evaluation

We present an evaluation of the Heterogeneous Coded Delivery (HCD) scheme using simulations for both the centralized and decentralized approaches. Our focus is mainly on the evaluation of the delivery scheme, *not* of the caching policy; the latter is fixed for each evaluation. The load $R_{(d_1,d_2,\ldots,d_k)}$ of the coded message depends on the demand vector $(d_1, d_2, \ldots, d_K)$,. Therefore, to evaluate the overall performance, we will use the expected load metric $\bar{R}(\mathbf{p}) = \sum_{(d_1,\ldots,d_K)} R_{(d_1,\ldots,d_K)} p_{d_1} p_{d_2} \cdots p_{d_K}$ throughout this section. We will mainly be comparing the expected load of our scheme to the state-of-the-art delivery scheme provided in [16], [21].

*A note on the decentralized scenario.* We would like to highlight some points before diving into the evaluation results. In the decentralized approach, the users randomly cache $q_n MF$ bits of file $W_n$ in their respective caches. Because of the randomness here, the bits of a file are distributed across subfiles of various types, unlike the centralized approach where they are contained in subfiles of a single type. It is easy to see that a bit in file $W_n$ has a probability $q_n M$ of being cached at any given user. Note $q_n M \leq 1$, as it does not make sense to allot more that the $1F$ bits for caching the file $W_n$. The probability that a bit gets cached at $t$ users can be easily deduced as:

$$Pr(\text{a bit is cached at } t \text{ users}) = \binom{K}{t} (q_n M)^t (1 - q_n M)^{K-t}$$

Thus the number of bits in type $t$ subfiles can be modeled as a binomial distribution $B(K, q_n M)$ and the bits within a type $t$ are uniformly distributed across all the subfiles of type $t$. We will use this modeling to simulate the decentralized caching.

### A. Uniform Caching

Under a uniform caching policy, all $N$ files will be allocated an equal amount of cache space at each user, *i.e.* $q_n = 1/N \ \forall \ n$. If such a caching policy is used, irrespective of the file popularity distribution, both our scheme and the scheme in [16] would yield similar load values. Especially, a centralized placement scheme following a uniform caching policy will have a strong symmetry in the cache contents, which would result in our scheme essentially reducing to the scheme in [16]. In the centralized approach, the bits of a file are split within subfiles of the same type, and due to the uniform nature of the caching policy, a similar phenomenon can be observed across all the files. Here the step creating new subfiles that borrow bits from subfiles of higher type becomes unnecessary.

In the decentralized approach, the subfile types are no longer limited to a single value. The bits of a file get distributed across several subfiles types, which can be modeled as a binomial distribution. This distribution maintains its parameters across all files because of the uniformity of $q_n$. All the subfiles which would be coded together in the delivery phase will have almost the same number of bits, thereby minimizing the need to borrow bits from higher type subfiles in our algorithm. Thus the improvement we get from our algorithm, compared to the state of the art, would be negligible. This was indeed observed in our simulations.

### B. Non-Uniform Caching

We define non-uniform caching as a caching policy that does not allocate equal cache space for all files irrespective of their popularities. This definition includes the multilevel grouping in [21] and the two level policy used in [10].

The caching policy considered in [10] divides the files into two groups, one group with files of low popularity which will not be cached at any user and the remaining files of higher popularity which will divide the cache space equally among themselves. The bits of these files cannot be coded together with another bit, as the users do not have the side information necessary to decode. The bits of the files in the later groups are uniformly divided in subfiles of single type in a centralized approach. Therefore, the HCD scheme will not have performance gain compared to the state-of-the-art. The same arguments from decentralized uniform caching can be used to explain the negligible difference in load observed in the decentralized approach for this caching policy.

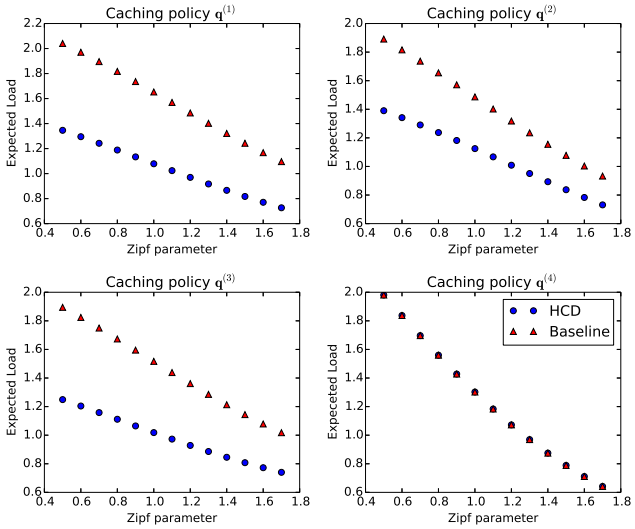


Fig. 9. Plot of expected load for $N = 10, K = 4$ and $M = 5$ and centralized caching. $\mathbf{q}^{(1)}, \mathbf{q}^{(2)}, \mathbf{q}^{(3)}$ : arbitrary multilevel caching polices - around 30% improvement can be noted. $\mathbf{q}^{(4)}$ : two level caching scheme with levels 0 and 0.25 - no improvement.

If the considered caching policy has multiple levels like the one in [21] or example 1, then we can see a significant difference in the expected load. Example 1 uses a caching policy where $\mathbf{q} = \mathbf{p}$, we were able to calculate the expected load for HCD scheme as 1.156 and the expected load of state of the art scheme as 1.696. This brings an improvement of nearly 30% for this example. We also evaluated our scheme for a system with $N = 10$ files, $K = 5$ users and $M = 4$ sized cache, where the user demands are Zipf distributed. The results of the evaluation for a centralized caching approach is presented in fig. 9. The figure shows a plot of the expected load vs Zipf parameter for four different caching policies. The first three are arbitrary policies with multiple level groups, we are able to see a significant improvement for these; the fourth one is similar to the policy in [10], where an uniform caching policy is followed for the first four popular files (level $1/4$)and the remaining files are not cached (level 0). HCD scheme does not provide any improvement in this case because of the uniform nature of the caching policy.

The performance evaluation of HCD for a decentralized caching approach is shown in fig. 10, again for a system with $N = 10$, $K = 5$ and $M = 4$. The plots in the top row of fig. 10 are for a caching policy that is also Zipf based, namely $Zipf(0.5, N)$, $Zipf(0.7, N)$ and $Zipf(1, N)$. We can see that for the first two plots in the top row, there is a performance benefit of around 10%. The third plot in the top row shows a huge performance difference due to the inefficiency of the caching policy; this caching policy results in allocating more than $F$ bits in the cache of each user for some of the most

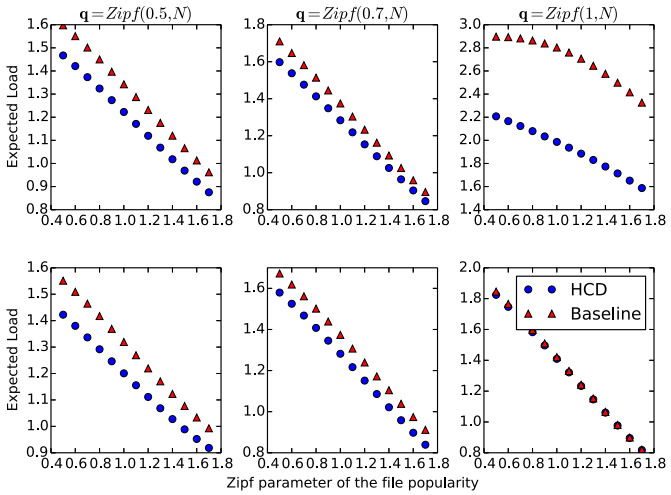


Fig. 10. Plot of expected load for $N = 10, K = 4$ and $M = 5$ and decentralized caching. Top row: Zipf based caching policy; Bottom row: Zipf based caching policy with factor 2 grouping applied. Big gap in performance in the top right plot is an anomaly due to $q_i M > 1$. Improvements of around 10% are seen in almost all cases.

popular files which is obviously inefficient.

### C. Comparison to State-of-the-Art.

*Performance.* HDC performs always at least as well as the baseline scheme, by design: HCD considers all the coding opportunities that the baseline does, and then some more. In the uniform case, however, there is no substantial benefit: due to the symmetry of the problem, our scheme essentially degenerates to the baseline. The significant benefits come in the non-uniform case, for which our scheme was specifically designed to extend beyond the baseline approach.

*Complexity.* HDC has a polynomial complexity in the number of subfiles $V_{k,\mathcal{S}}$ (nodes of the graph $G_d$), the same as the coded caching schemes in [16] and [21]. The main difference between HCD and the one in [21] is the added step of borrowing bits from higher type subfiles to fill up the lower type ones. Consider the worst case scenario, where a subfile of very low type needs to borrow some bits, the algorithm will first access the next immediate higher type and will keep moving up to higher type until it is full. The algorithm will move to higher types only after exhausting all bits from the immediate higher type. So even if it ends up borrowing bits from the highest type subfiles, in the process it has covered all the bits for that user's demand.

## VI. Conclusion

The coded caching problem consists of two phases: one involves optimization of the caching policy and the other is the design of the delivery scheme that minimizes the load on the shared link for any given demand. The caching policy optimization is a very interesting problem, with recent works from [10], [21] showing some order optimality results, but *not* the focus of this paper. Given the cache content and the demands of the users, the delivery phase optimization is basically an instance of the index coding problem, which is known to be NP-hard. We provide a practical algorithm, called the *Heterogenous Coded Delivery (HCD)* scheme, that performs significantly better than the current state-of-the- art scheme in coded caching for all multilevel (more than two) caching policies. An open question for future work is whether multiple levels are necessary for optimal caching policies, or two levels are sufficient.

## References


[1] I. Baev, R. Rajaraman, and C. Swamy. Approximation algorithms for data placement problems. *SIAM Journal on Computing*, 38(4):1411–1429, 2008.

[2] Z. Bar-Yossef, Y. Birk, T. Jayram, and T. Kol. Index coding with side information. *Information Theory, IEEE Transactions on*, 57(3):1479–1494, 2011.

[3] Y. Birk and T. Kol. Coding on demand by an informed source (ISCOD) for efficient broadcast of different supplemental data to caching clients. *IEEE/ACM Transactions on Networking (TON)*, 14(SI):2825–2830, 2006.

[4] S. Borst, V. Gupta, and A. Walid. Distributed caching algorithms for content distribution networks. In *IEEE INFOCOM*, 2010.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE INFOCOM'99*, volume 1, 1999.

[6] N. Golrezaei, K. Shanmugam, A. Dimakis, A. Molisch, and G. Caire. Femtocaching: Wireless video content delivery through distributed caching helpers. In *INFOCOM, 2012 Proceedings IEEE*, pages 1107–1115, March 2012.

[7] M. Hefeeda and O. Saleh. Traffic modeling and proportional partial caching for peer-to-peer systems. *Networking, IEEE/ACM Transactions on*, 16(6):1447–1460, 2008.

[8] M. Ji, G. Caire, and A. F. Molisch. Fundamental limits of distributed caching in D2D wireless networks. *CoRR*, abs/1304.5856, 2013.

[9] M. Ji, G. Caire, and A. F. Molisch. The throughput-outage tradeoff of wireless one-hop caching networks. *CoRR*, abs/1312.2637, 2013.

[10] M. Ji, A. M. Tulino, J. Llorca, and G. Caire. On the average performance of caching and coded multicasting with random demands. *arXiv preprint arXiv:1402.4576*, 2014.

[11] M. R. Korupolu, C. G. Plaxton, and R. Rajaraman. Placement algorithms for hierarchical cooperative caching. *Journal of Algorithms*, 38(1):260–302, 2001.

[12] M. Langberg and A. Sprintson. On the hardness of approximating the network coding capacity. In *IEEE ISIT*, pages 315–319, 2008.

[13] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. on Computers*, 50(12):1352–1361, 2001.

[14] A. Leff, J. L. Wolf, and P. S. Yu. Replication algorithms in a remote caching architecture. *Parallel and Distributed Systems, IEEE Transactions on*, 4(11):1185–1204, 1993.

[15] J. Llorca, A. Tulino, K. Guan, and D. Kilper. Network-coded caching-aided multicast for efficient content delivery. In *IEEE ICC*, pages 3557–3562, June 2013.

[16] M. A. Maddah-Ali and U. Niesen. Decentralized caching attains order-optimal memory-rate tradeoff. *arXiv preprint arXiv:1301.5848*, Jan. 2013.

[17] M. A. Maddah-Ali and U. Niesen. Fundamental limits of caching. *arXiv preprint arXiv:1209.5807*, Sept. 2012.

[18] F. Malandrino, M. Kurant, A. Markopoulou, C. Westphal, and U. Kozat. Minimizing the peak load from information cascades: Social networks meet cellular networks. *IEEE Trans. on Mobile Computing*, 2015.

[19] A. Meyerson, K. Munagala, and S. Plotkin. Web caching using access statistics. In *ACM-SIAM symposium on Discrete algorithms*, 2001.

[20] M.-J. Montpetit, C. Westphal, and D. Trossen. Network coding meets information-centric networking: an architectural case for information dispersion through native network coding. In *ACM NOMEN workshop*, pages 31–36. ACM, 2012.

[21] U. Niesen and M. A. Maddah-Ali. Coded caching with nonuniform demands. *arXiv preprint arXiv:1308.0178*, Aug. 2013.

[22] J. Wang, J. Ren, K. Lu, J. Wang, S. Liu, and C. Westphal. An optimal cache management framework for information-centric networks with network coding. In *IFIP/IEEE Networking Conference*, June 2014.

[23] A. Wolman, M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *ACM SIGOPS Operating Systems Review*, volume 33, 1999.