BUCKET-FILLING: AN ASYMPTOTICALLY OPTIMAL VIDEO-ON-DEMAND NETWORK WITH SOURCE CODING

by

ZHANGYU CHANG

A Thesis Submitted to
The Hong Kong University of Science and Technology
in Partial Fulfillment of the Requirements for
the Degree of Master of Philosophy
in Computer Science and Engineering

August 2015, Hong Kong

Authorization

I hereby declare that I am the sole author of the thesis.

I authorize the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

_____

ZHANGYU CHANG

BUCKET-FILLING: AN ASYMPTOTICALLY OPTIMAL VIDEO-ON-DEMAND NETWORK WITH SOURCE CODING

by

ZHANGYU CHANG

This is to certify that I have examined the above M.Phil. thesis

and have found that it is complete and satisfactory in all respects,

and that any and all revisions required by

the thesis examination committee have been made.

PROF. S.-H. GARY CHAN, THESIS SUPERVISOR

PROF. QIANG YANG, HEAD OF DEPARTMENT

Department of Computer Science and Engineering

25 August 2015

ACKNOWLEDGMENTS

I would never have completed this work without the help from many people. First of all, I thank my advisor, Professor S.-H. Gary Chan, for his years of mentoring, advice, and encouragement. I have learned a lot from him how to develop, evaluate, express, and defend my work. Such skills would be beneficial for my later Ph.D. study.

I thank the members of my thesis committee, Professor A and Professor B, for their insightful comments on improving this work.

I thank my colleagues in HKUST, Mr. Bo Zhang, Mr. Hongzheng Xiong, Mr. Huang Lu, Ms. Lijia Hong and many others, for their constructive suggestion and comments of this work. They helped me a lot. I also thank my parents for their support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

BUCKET-FILLING: AN ASYMPTOTICALLY OPTIMAL VIDEO-ON-DEMAND NETWORK WITH SOURCE CODING

by

ZHANGYU CHANG

Department of Computer Science and Engineering

The Hong Kong University of Science and Technology

ABSTRACT

There has been growing interest in recent years for content providers to provide video-on-demand (VoD) as a cloud service. In such a network, the content provider may rent heterogeneous resources (such as streaming and storage capacities) from geographically distributed data centers deployed close to user pools. These data centers (or proxy servers) collaboratively share contents with each other to serve their local users. A critical challenge is hence to optimize movie storage and retrieval to minimize the deployment cost consisting of streaming, storage, and network transmission between data centers.

We propose a novel and effective movie storage and retrieval using linear source coding. All the movies are source-encoded once at the repository, by taking every q source symbols of movie m to generate $n^{(m)}$ coded symbols. These coded symbols are then distributed to the servers in the cloud. Based on a general and comprehensive cost model, we optimize $n^{(m)}$ and the number of symbols to retrieve from remote servers for a local movie request. The optimal solution can

be efficiently computed with a linear programming (LP) formulation. Our solution is proved to approach asymptotically the global minimum cost as q increases. Even when q is low (say, 30), near optimality can be achieved. To accommodate large movie pool and system parameter changes, we propose algorithms for movie grouping and on-line re-optimization which significantly reduce the computational complexity with little compromise on optimality. Through extensive simulation, our algorithm is shown to achieve remarkably the lowest cost, outperforming traditional and state-of-the-art heuristics with a substantially wide margin (of multiple times in many cases).

CHAPTER 1

INTRODUCTION

There has been growing interest for content providers to provide video-on-demand (VoD) as a cloud service. In order to cost-effectively provide such distributed service, a content provider may rent resources (bandwidth and storage) at data centers. These data centers cooperatively store and retrieve movies, greatly reducing network load and scaling up the streaming and storage capacities [19, 5, 1].

We show in Figure 1.1 a cooperative VoD network with distributed data centers (servers). The network consists of a central server (repository) storing all the movies and data centers, also termed as *proxy servers* in this thesis, placed geographically close to user pools.[1] While the central server stores all the movies, the proxy servers may be of limited, and possibly heterogeneous, storage which can only locally store a fraction of the movies (full replication at all the proxy servers is not cost-effective, especially for VoD applications where movie popularity is often skewed). Each user has a home (or local) proxy server to serve his movie request. If the home server has the requested content (a hit), it directly streams to the users from its local storage. Otherwise (i.e., a miss), the home server requests the content from a remote server, which is either a proxy or the central server. The missed content is then streamed "via" the home server to the request. To minimize user delay, movies are not downloaded at the clients before they are played back. (The user startup delay can be further reduced by pre-storing the "prefixes" of the leading, say, 30 seconds of movies at each server. In any case, such technique is orthogonal to our current study.)

In the VoD cloud, a critical challenge for the content provider is to minimize the total *deployment cost* given by the sum of server and network costs through optimizing movie storage and retrieval at the servers. We consider *server cost* as a general function of its own storage and streaming bandwidth (maximum capacity or utilized). In addition, we consider *network cost* as the

[1] In this thesis, "client" and "user" are used interchangeably, and so are "movie," "video" and "content."

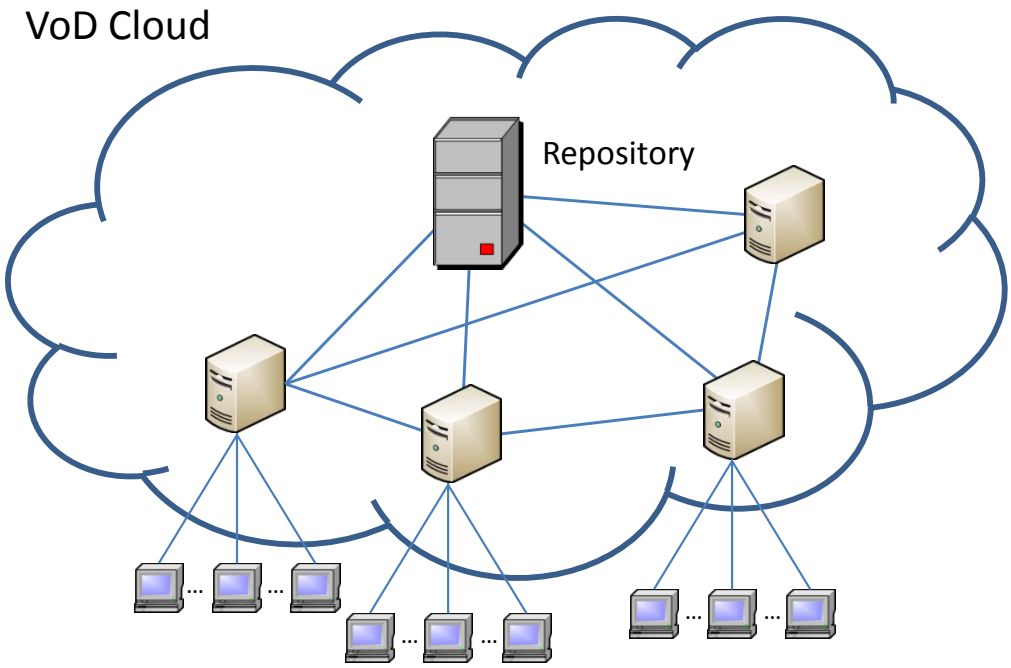

Figure 1.1: A distributed servers architecture for VoD service.

bandwidth cost to stream data from a server to another (a function of the data traffic between two locations). As different from previous work [4, 26, 34, 9, 12, 33, 2, 10, 27], our cost model is much more general and captures the important cost components for system deployment.

Optimal movie storage and retrieval in a VoD network is generally regarded as NP-hard [8, 26]. This is mainly because a movie (or its constituent segments) at a server is often regarded as either stored or not, resulting in a 0-1 integer programming problem. Because of the intractability of the problem, many heuristic algorithms have been proposed. It is often not clear how well these heuristics perform as compared with the optimum, let alone approaching such optimum.

We propose a novel movie storage and retrieval algorithm which achieves *asymptotically optimal* solution, i.e., the deployment cost can be arbitrarily close to the global minimum (by increasing a system parameter q which trades off optimality gap and coding complexity). Even under practical and realistic condition, the system performs very close to the optimum, significantly better than the other state-of-the-art heuristics (often by many times).

In the VoD network, movie m is source-encoded only once at the repository by taking every q

source symbols to generate $n^{(m)} \geq q$ coded symbols using a general linear source coding technique (such as Reed-Solomon code, Maximum-Distance-Separable (MDS) systematic erasure code, etc.), where $n^{(m)}$ is the optimizing parameter in our study, and q is a tunable network-wide parameter depending on how much coding complexity and decoding delay one is willing to accept.

The repository stores q of these $n^{(m)}$ coded symbols, and the remainder is distributed at the other proxy servers without duplication (obviously no more than q at each server). So long as any q out of the $n^{(m)}$ symbols are collected, the original source symbols can be recovered at the proxy. As the decoding overhead of linear coding has been shown to be low (as compared with video decoding), such decoding can be done either at the proxies or directly at the users [13].

To serve a local request for a movie, one hence may imagine that the request carries a "bucket" of size q symbols. The bucket is first filled by downloading the coded symbols at its home server. If this does not fully fill up the bucket, the home server collects by pulling the remaining symbols from the other servers (including the repository). Once q coded symbols are collected, the source symbols can then be played back.

It is clear from above that by distributing and retrieving symbols from servers in an intelligent manner, the deployment cost can be minimized. The major issues are hence, given q, what the optimal $n^{(m)}$ is for movie m, how many symbols should be stored at a server, and how many symbols to retrieve from each of them for a movie request.

Our contributions are three-folds:

- *Bucket-filling: A novel movie distribution and retrieval algorithm based on source coding (SC):* We propose a novel video-on-demand network using linear source coding. Our scheme, termed bucket-filling, is a remarkably simple and effective movie distribution and retrieval scheme minimizing system cost.

- *Provably asymptotically optimal performance for distributed video-on-demand:* By optimizing $n^{(m)}$ for movie m, bucket-filling is able to minimize deployment cost consisting of server storage, server bandwidth, and network access. Bucket-filling uses an efficient linear program (LP) and discretization to optimize symbol distribution and retrieval at servers. Its cost

can be proved to be arbitrarily close to the exact global minimum as q increases, i.e., asymptotically optimal in terms of q. We illustrate that even under the most general and realistic condition of low values of q (around 30), the system performs closely optimal.

- *Efficient grouping and on-line re-optimization for large movie pool:* To further address large movie pool, we propose a movie grouping algorithm based on K-means clustering which significantly reduces computational complexity (by a factor of $O(|M|)$, where $|M|$ is the number of movies) with little compromise in optimality. Our scheme also easily applies to system changes due to, for example, introduction and removal of movies, change in network cost, introduction of servers, etc.

We conduct extensive simulation and comparison study with other traditional and state-of-the-art schemes. Our results show that bucket-filling achieves asymptotic optimality in system cost, outperforming the other schemes by a significantly wide margin (multiple times in most cases). The results show that the performance of many previously proposed heuristics are still far from the global optimum, and bucket-filling can achieve performance arbitrarily close to it. Furthermore, with very low computational cost, our grouping scheme can still achieve close to optimal performance.

This thesis is organized as follows. We first review related work in Chapter 2. We then describe how the VoD system works with source coding and formulate the optimization problem in Chapter 3. In Chapter 4, we present symbol storage and retrieval solutions for bucket-filling, movie grouping algorithm for large movie pool, and system re-optimization due to parameter changes. In Chapter 5, we show illustrative simulation results on the performance and comparison of bucket-filling. We conclude in Chapter 6. We review linear source coding and prove the asymptotic optimality of bucket-filling in Appendix A and B, respectively.

CHAPTER 2

RELATED WORK

We briefly discuss previous work below. There has been much work applying network coding or fountain codes in peer-to-peer VoD [21, 16, 32, 23, 28]. These schemes incur significant processing and re-encoding overhead in the peer network and may lead to duplicated symbols, which decreases network efficiency. As receiving duplicated symbols affects stream continuity, the work in [22] discusses the design of (q, n) for source-coded VoD to reduce duplicated symbols in the network. The objective of the work, together with the recent work on peer-to-peer VoD [39, 38, 30, 3], is to fully utilize the uploading bandwidth of the peers. The work in [36, 17] presents heuristics to efficiently search for movie segments in order to support user interactivity. All the work in [21, 22, 16, 32, 23, 28, 39, 38, 36, 17, 30, 3] has not considered the *optimal* movie storage and retrieval to minimize the deployment cost due to bandwidth and storage. Our work presents an *asymptotically optimal* solution for a novel VoD network. The use of source coding is a one-step encoding process without any symbol duplication in the network, which leads to storage and access efficiency.

For the work studying the cost issue for VoD [4, 26, 34, 9, 12, 33, 2, 10, 27], they often have not sufficiently considered the general case including all the cost components. We consider a realistic and general VoD deployment model, which captures a comprehensive set of parameters with major cost components in regards to network access cost, storage constraint and streaming cost of the servers.

As movie replication and retrieval problem is typically regarded as NP-hard, various heuristics have been proposed [4, 37, 8, 26, 18]. These algorithms are generally sub-optimal. As the performance bounds of these algorithms are not easy to analyze or derive, it is not clear how far their performance is from the optimum. In contrast, bucket-filling achieves *asymptotically* optimal performance, i.e., it can be arbitrarily close to the exact minimum cost by increasing the system parameter q. Such optimality is hence achieved with increasing decoding overhead given by q.

Our optimality is shown to be significantly better than the state-of-the-art schemes. Furthermore, previous algorithms are often based on iteration [20, 11], which may have convergence issue for a large network. Bucket-filling, on the other hand, is efficient as it is not based on iteration and has a guaranteed worst-case algorithmic complexity.

In summary, our work distinguishes and advances from the previous ones in the following major aspects:

1. We propose a novel video-on-demand network based on source coding for efficient movie distribution and retrieval. This architecture ensures efficient video access without duplicated symbols.

2. We comprehensively consider the major cost components of a realistic VoD network and address its cost optimization issue.

3. Our algorithm is provably asymptotically optimal and has guaranteed worst-case time complexity, which outperforms the state-of-the-art schemes in terms of both deployment cost and algorithmic complexity.

A video-on-demand network called LP-SR has been presented in [35, 6]. In this scheme, the movies are partitioned into many segments for storage and retrieval. Compared with it, bucket-filling advances in the following ways:

1. Bucket-filling substantially reduces the time complexity of the optimization. It is not based on segments, and uses a radically different and efficient discretization process by means of source coding. As any q of $n^{(m)}$ coded symbols can recover the original source symbols, bucket-filling is flexible and amendable to system changes. Such novel use of source coding markedly reduces the time complexity of the solution, and leads to a much different joint movie storage and retrieval strategy (see Section 4).

2. Bucket-filling is provably asymptotically optimal. Due to the simplicity and tractability of bucket-filling, we are able to prove that bucket-filling approaches the exact optimum as q increases.

3. We propose an efficient movie grouping algorithm based on K-means clustering, which substantially reduces the time complexity. Simulation results show that our scheme can optimize a typical VoD network with thousands of movies in seconds with little sacrifice on performance (see Section 5.3). Besides, we give an on-line re-optimization algorithm to accommodate system changes with minimal symbol redistribution.

A preliminary version of this work has been reported in [7]. This work extends it in various major ways:

1. We prove the asymptotic optimality of bucket-filling. This proof demonstrates that bucket-filling can be arbitrarily close to the exact optimum as q increases.

2. We propose an efficient movie grouping algorithm based on K-means clustering, which greatly reduces the running time of the algorithm for large movie pool with close-to-optimal performance.

3. We present a novel and efficient re-optimization method with little overhead in symbol redistribution when re-optimizing the system.

4. We show more substantial illustrative results to validate the strong performance of bucket-filling, our new clustering and re-optimization methods.

CHAPTER 3

SYSTEM DESCRIPTION AND PROBLEM FORMULATION

In this chapter, we first present how bucket-filling works in the VoD network (Section 3.1). We then present the cost optimization problem of the VoD network for the asymptotic case $q \to \infty$ (Section 3.2). In this case, the formulation becomes an LP which can be solved efficiently. For finite q, the LP solution requires further discretization which is discussed in Chapter 4.

3.1 System Operation

A movie m is source-coded only once at the repository by taking every q equal-sized source symbols to generate $n^{(m)} \geq q$ coded symbols of the same size. (Given a certain symbol size (in bits), a movie of longer length hence generates more number of source symbols and thereof coded symbols.) Out of the $n^{(m)}$ coded symbols, the repository stores any of the q coded symbols, and distributes the remainder without replication to the proxy servers. Note that the repository and servers do not need to store more than q symbols out of $n^{(m)}$, because the original q source symbols can be fully reconstructed with any q of the $n^{(m)}$ symbols.

In bucket-filling, the symbols are "streamed" to the server and the number of symbols to retrieve from each server for a request is determined in the optimization process at the repository. Such retrieval decision is deterministic where a fixed number of symbols are retrieved from a server (a non-probabilistic approach). As the solution can be obtained with a simple table look-up, the communication overhead of servers is minimal.

In the network, movies are distributed and retrieved according to the following:

- *Coded Symbol Distribution:* Given q, the repository computes the optimal $n^{(m)}$ for each movie. It then encodes the movies accordingly once and distributes the coded symbols of the

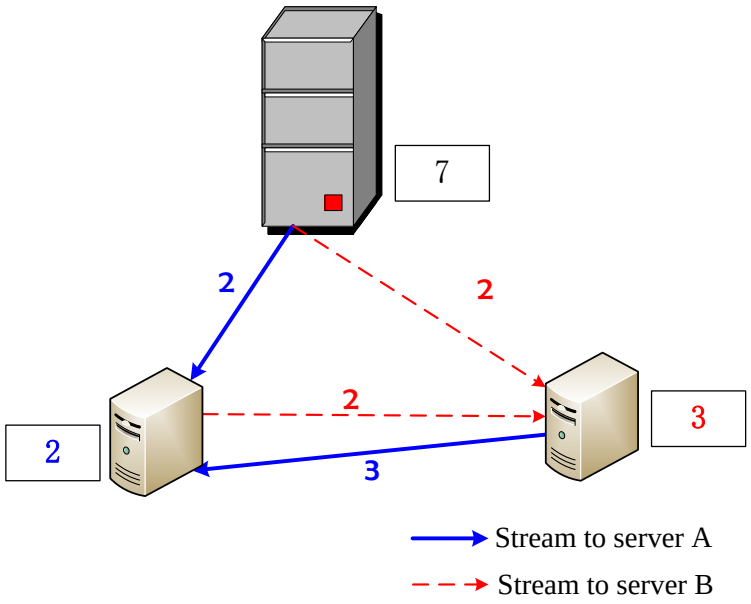

Figure 3.1: An illustrative example for bucket-filling with $q = 7$, 1 movie, $n^{(1)} = 12$, and 2 proxy servers.

movies to each server. Such symbol distribution needs to be done only upon major system changes, e.g., upon the introduction and removal of movies or change in movie popularity which affects movie storage in a major way. (We address how this re-distribution can be done efficiently due to re-optimization in Section 4.3.)

- *Coded Symbol Retrieval:* A movie request carries a bucket of size q symbols. If its home server has not stored, and hence cannot supply, q symbols to serve the request, it "pulls" the missing ones from the other proxy or central servers so as to fill up the bucket. Through this *bucket-filling* mechanism, the servers cooperatively store and supply symbols on-demand with each other to fulfill requests.

We present an illustrative example of bucket-filling in Figure 3.1, which shows a simple VoD network with 1 movie and 2 proxy servers. Given $q = 7$, suppose the optimal solution is $n^{(1)} = 12$, i.e., the repository (central server) encodes the movie using 7 source symbols to generate 12 coded symbols.

After storing $q = 7$ coded symbols itself, the repository distributes the remaining 5 $(= n^{(1)} - q)$ coded symbols to the proxy servers. In this example, servers A and B get 2 and 3 symbols,

Table 3.1: Major symbols used in this thesis.

Notation	Definition
$T(V, E)$	The directed graph representing the overlay network topology
V	The set of servers (repository and distributed proxy servers)
$\|V\|$	The number of servers
M	The set of movies
$\|M\|$	The number of movies
$L^{(m)}$	Length of movie m before linear source coding (in seconds)
$p^{(m)}$	Access probability of movie m
$I_v^{(m)}$	Amount of coded movie m server v stores (in seconds)
B_v	Storage capacity of server v (in seconds)
$r_{uv}^{(m)}$	Amount of coded movie m streamed from server u to server v (in seconds)
λ_v	Request rate at server v (requests per second)
$\alpha^{(m)}L^{(m)}$	Average holding (viewing) time of movie m (in seconds) $\alpha^{(m)} \geq 0$
Γ_{uv}	Network transmission bandwidth from server u to v (bits/s)
s	Movie streaming rate (bits/s)
R_v	Total uploading bandwidth of server v (bits/s)
C_v^{S}	Cost of server v (per second)
C^{S}	Aggregated server cost (per second)
C_{uv}^{N}	Network cost due to directed traffic flow from server u to v (per second)
C^{N}	Total network cost (per second)
C	Total deployment cost ($= C^{\mathrm{S}} + C^{\mathrm{N}}$)

respectively.

An interactive request for the movie has a bucket of size $q = 7$ symbols. To fulfill it at server A, server A supplies its stored 2 symbols and gets 2 and 3 symbols from the repository and server B, respectively. On the other hand, server B fulfills its request by supplying 3 local symbols and getting 2 symbols from the repository and server A each.

3.2 A Linear Programming Formulation

In this section, we present the cost-optimization problem of our VoD network for the asymptotic case $q \to \infty$ given movie popularity (how to estimate movie popularity is beyond the scope of this work. Interested readers may refer to [17, 25, 36] and references therein). For this case, the optimization becomes an LP which can be solved efficiently and exactly.

We show in Table 3.1 the important symbols used (similar notations have also been used in [35, 6]). The overlay network is modeled as a directed graph $T = (V, E)$, where V is the set of

central and proxy servers and $E \subseteq V \times V$ is the set of overlay edges connecting nodes in V (may not be complete). Let M be the set of movies and $L^{(m)}$ be the movie length (i.e., movie length before source coding). Let $p^{(m)}$ be the popularity of movie m, which is the probability that a user requests movie m, where $\sum_{m \in M} p^{(m)} = 1$.

Each movie is source-coded to different length (obviously no less than $L^{(m)}$). Let $I_v^{(m)}$ (seconds) be the amount of coded movie m that server v stores. Obviously, we have

$$0 \leq I_v^{(m)} \leq L^{(m)}, \quad \forall v \in V, m \in M. \tag{3.1}$$

Note that for the repository (i.e., central server), we require $I_v^{(m)} = L^{(m)}, \forall m \in M$.

Server v has a certain storage capacity B_v (seconds). To meet storage requirement, we require

$$\sum_{m \in M} I_v^{(m)} \leq B_v, \quad \forall v \in V. \tag{3.2}$$

Let λ_v be the total movie request rate at server v (requests per second); the request rate for movie m at the server is hence $p^{(m)}\lambda_v$. Further let $\alpha^{(m)}L^{(m)}$ be the average holding (or viewing) time for movie m, where $\alpha^{(m)} \geq 0$.

Each user retrieves data from the servers (including his home server) proportional to his holding time. Let $r_{uv}^{(m)}$ (seconds) be the amount of movie m supplied from server u to server v for a user holding time of $L^{(m)}$. We hence must have

$$\sum_{u \in V} r_{uv}^{(m)} \geq L^{(m)}, \quad \forall v \in V, m \in M. \tag{3.3}$$

Note that we have considered user interactivity on movie through $\alpha^{(m)}$, which may be different for different movies (interesting movies may have $\alpha^{(m)} > 1$, or vice versa). Furthermore, we are interested in *average* holding time, as we are considering time-averaged cost at steady state (hence the distribution of the holding time may be different for different movies). While interacting with the movie, a user holds up a stream and may uniformly visit any symbols of the movie over time.

Therefore, the actual amount of streamed data is given by $\alpha^{(m)}r_{uv}^{(m)}$. As the server cannot supply more than that it stores, we need

$$0 \leq r_{uv}^{(m)} \leq I_u^{(m)}, \quad \forall u, v \in V, m \in M, \tag{3.4}$$

and, by definition, $r_{vv}^{(m)} = I_v^{(m)}$.

Let Γ_{uv} (bits/s) be the total network bandwidth used for symbol transmission from server u to v, which can be obtained as

$$\Gamma_{uv} = \sum_{m \in M} p^{(m)} \lambda_v \alpha^{(m)} r_{uv}^{(m)} s, \quad \forall u, v \in V, \tag{3.5}$$

for $u \neq v$, and, by definition, $\Gamma_{uu} = 0$.

Let C_{uv}^{N} be the network cost due to the directed traffic from server u to v. It is a monotonically non-decreasing piece-wise linear function in Γ_{uv}, i.e., $C_{uv}^{\mathrm{N}} = \mathbb{C}_{uv}^{\mathrm{N}}(\Gamma_{uv})$ for all $u, v \in V$ with $C_{uu}^{\mathrm{N}} = 0$. Note that our model is general as C_{uv}^{N} does not have to be the same as C_{vu}^{N}.

The total network cost C^{N} is hence

$$C^{\mathrm{N}} = \sum_{u,v \in V} C_{uv}^{\mathrm{N}}. \tag{3.6}$$

The servers help each other using "cache and stream" model, i.e., a remote server streams to a user *through* his home server. In other words, the home server is an intermediate node between the remote server and the users. For any remote server $v \in V$, the data rate the server u "pulls" from server v for movie m is $p^{(m)} \lambda_u (\alpha^{(m)} r_{vu}^{(m)} s)$. The total rate (bits/s) that server v serves other servers is hence

$$R_v = \sum_{u \in V, u \neq v} \Gamma_{vu}, \quad \forall v \in V. \tag{3.7}$$

While network cost depends on the traffic between *pairs* of servers, the cost of a server depends on its total storage and uploading rate used (in order to serve other servers in the network). Such storage and rate are limited by its disk capacities independent of other servers. Let C_v^{S} be the cost of operating server v, which is a monotonically non-decreasing piece-wise linear function in B_v and R_v, i.e., $C_v^{\mathrm{S}} = \mathbb{C}_v^{\mathrm{S}}(B_v, R_v)$ for all $v \in V$. In other words, the server cost consists of storage cost and streaming cost.

Therefore, the aggregated server cost C^{S} is

$$C^{\mathrm{S}} = \sum_{v \in V} C_v^{\mathrm{S}}. \tag{3.8}$$

Finally, the total system deployment cost C is

$$C = C^{\mathrm{S}} + C^{\mathrm{N}}. \tag{3.9}$$

We state our cost-optimization problem as follows:

Optimal Movie Distribution and Retrieval Problem to Minimize Deployment Cost: Given topology T, user demand $\{\lambda_v\}$, storage capacity $\{B_v\}$, movie popularity $\{p^{(m)}\}$ and cost functions $\mathbb{C}^{\mathrm{N}}_{uv}$ and $\mathbb{C}^{\mathrm{S}}_{v}$, we seek to minimize the total cost given by Equation (3.9), subject to Equations (3.1) to (3.5). The output is the optimal solution of the amount of the movie stored in each server (i.e., $\{I^{(m)}_v\}$) and the retrieval amount between servers (i.e., $\{r^{(m)}_{uv}\}$).

Note that, for arbitrary piece-wise linear functions of $\mathbb{C}^{\mathrm{S}}_{v}$ and $\mathbb{C}^{\mathrm{N}}_{uv}$, the above problem becomes a linear programming (LP) problem which can be solved efficiently.

Time complexity of the LP solution: To solve the LP, we may employ CVX [15] which implements the wide-region centering-predictor-corrector algorithm (an interior-point method) to solve this problem [29]. The number of variables of the above formulation is $O(|V|^2|M|)$. Therefore, it has an $O(|V||M|^{1/2})$ worst-case iteration bound and $O(|V|^6|M|^3)$ overall expected time complexity.

CHAPTER 4

BUCKET-FILLING: SYMBOL STORAGE AND RETRIEVAL

The solution of the LP formulation in Section 3.2 is for the case $q \to \infty$, which requires further discretization for finite q. We present a discretization process which achieves closely optimal and is asymptotically optimal in q. The system approaches exactly optimum as q approaches infinity (Section 4.1). In addition, we propose an efficient movie grouping algorithm based on K-means clustering for large movie pool (Section 4.2) and an efficient symbol redistribution scheme to respond to changes in system parameters (Section 4.3).

Note that the solutions for symbol distribution and retrieval can be implemented in a central optimizer, and updated regularly based on the prediction interval on user traffic and movie popularity over time.

4.1 Parameter Discretization to Achieve Asymptotic Optimum

The LP yields optimal solution for system parameters $\{I_v^{(m)}\}$ and $\{r_{uv}^{(m)}\}$ for movie m. Given these parameters, the movie can then be encoded, distributed and retrieved according to the following (for large q):

- *Movie encoding:* To obtain the encoding parameter $n^{(m)}$, observe that the source-coded and raw movie lengths must satisfy the following equation:

$$\frac{\sum_{v \in V} I_v^{(m)}}{L^{(m)}} = \frac{n^{(m)}}{q}, \quad \forall m \in M,$$

i.e.,

$$n^{(m)} = \frac{\sum_{v \in V} I_v^{(m)}}{L^{(m)}} q, \quad \forall m \in M. \tag{4.1}$$

- *Symbol distribution (storage):* The number of symbols that server v stores out of $n^{(m)}$ coded symbols is given by

$$n_v^{(m)} = \frac{I_v^{(m)}}{L^{(m)}} q, \quad \forall m \in M. \tag{4.2}$$

- *Symbol retrieval (collection):* The number of symbols for server u to stream to server v out of $n^{(m)}$ coded symbols is

$$n_{uv}^{(m)} = \frac{r_{uv}^{(m)}}{L^{(m)}} q, \quad \forall m \in M. \tag{4.3}$$

It is clear that $\{n^{(m)}\}$, $\{n_v^{(m)}\}$ and $\{n_{uv}^{(m)}\}$ in Equations (4.1)–(4.3) are exactly the LP optimal solutions as $q \to \infty$. For finite q, they should be discretized to integral values. We present below a simple discretization approach which converges to the asymptotic optimum as q increases. The basic idea is that each proxy tries to match the optimal LP solution as much as possible through integer rounding. In symbol retrieval by filling a bucket, the shortfall in symbols due to rounding can be obtained from the repository:

- *Discretize* $\{n_v^{(m)}\}$: We first round down the result $\{n_v^{(m)}\}$ as obtained in Equation (4.2) to its closest integers. For each server v, it first stores according to these integers for all the movies. This clearly does not violate its storage constraint (given in Equation (3.2)). For the residual storage it then stores a symbol of each movie in decreasing popularity until its total storage is fully used up.

  After this, the new $\{n_v^{(m)}\}$ are of integral values. The coding information $n^{(m)}$ for movie m is then given by

$$n^{(m)} = \sum_{v \in V} n_v^{(m)}, \quad \forall m \in M. \tag{4.4}$$

- *Discretize* $\{n_{uv}^{(m)}\}$: This is similar to the discretization of $\{n_v^{(m)}\}$. First we write $\{n_{uv}^{(m)}\}$ in Equation (4.3) as the sum of an integral part and a positive fractional part. Clearly, the integral part does not violate the supply constraint as given in Equation (3.4).

To recover the source packets (Equation (3.3)), we first rank the movies according to decreasing popularity. We then conditionally round up the fractional parts to 1 of the movies until Equation (3.3) is satisfied, and the remaining fraction is rounded down to 0. If Equation (3.3) is still violated after all the rounding, the remaining symbols are assigned to the repository.

In $\{n_v^{(m)}\}$ discretization, all the home servers and movies are traversed, and the time complexity is $O(|V||M|)$; in $\{n_{uv}^{(m)}\}$ discretization, all the home servers, remote servers and movies are traversed, and the time complexity is $O(|V|^2|M|)$. The time complexity of the discretization steps is hence $O(|V|^2|M|)$. Therefore, the total time complexity of our algorithm is $O(|V|^6|M|^3)$.

The discretization steps guarantee that all the movies can be recovered at each server. In Section 5.2, we can see from the simulation that the performance penalty due to rounding is very low. We prove that the system cost can be arbitrarily close to the *exact optimum* as q increases (i.e., asymptotically optimal) in Appendix B.

4.2 Efficient Computation for Large Movie Pool

In terms of the number of movies $|M|$, the computational complexity of linear programming for bucket-filling is $O(|M|^3)$ (Section 4.1). Even though it is of polynomial complexity, for a large movie pool, it is necessary to find a more efficient way to compute the solutions.

We propose an efficient movie grouping algorithm based on K-means clustering which achieves a polynomial reduction of a substantial factor of $O(|M|)$ in complexity with close-to-optimal deployment cost. We begin by denoting the *load index* $d^{(m)}$ as

$$d^{(m)} = p^{(m)}\alpha^{(m)}, \quad \forall m \in M, \tag{4.5}$$

since the access and its holding time indicate the streaming load of the movie. The key idea is to put the movies with the similar load indices into one group and minimize the sum of the difference on load index within each group. Let G be the set of groups and g_i be the ith group in G, and the number of groups $|G|$ is given as a network parameter. We illustrate the performance with simulation results in Section 5.3.

To motivate our grouping algorithm, consider a set of movies $m \in g$. From Equations (3.5) and (4.3), we get the network transmission due to movies in g as

$$\Gamma_{uv}^{(g)} = \frac{\lambda_v s}{q} \sum_{m \in g} d^{(m)} n_{uv}^{(m)} L^{(m)}, \quad \forall u, v \in V. \tag{4.6}$$

If all the movies $m \in g$ has the same $d^{(m)}$ (given by $d^{(g)}$) and $n_{uv}^{(m)}$ (given by $n_{uv}^{(g)}$), Equation (4.6) can be written as

$$\Gamma_{uv}^{(g)} = \frac{\lambda_v s}{q} d^{(g)} n_{uv}^{(g)} \sum_{m \in g} L^{(m)}, \quad \forall u, v \in V. \tag{4.7}$$

Namely, if we group the movies with the same $d^{(m)}$ and let $n_{uv}^{(m)}$ be the same, g can be treated as a "super movie" with load index $d^{(g)}$ and length $L^{(g)} = \sum_{m \in g} L^{(m)}$ for the linear program.

In the case that the movies are of different $d^{(m)}$, we hence may put the movies with similar $d^{(m)}$ into a group and minimize the sum of the differences of load index within each group. To formulate it mathematically, the objective of our movie grouping algorithm is to minimize

$$\text{arg}_{g_i} \sum_{i=1}^{|G|} \sum_{m \in g_i} |d^{(m)} - \mu^{(g_i)}|^2, \tag{4.8}$$

where $\mu^{(g_i)}$ is the mean $d^{(m)}$ of group g_i. This formulation is exactly K-means [24], a method to partition data into clusters in which each data belongs to the cluster with the nearest mean. Note that the group size of each group may not be the same by K-means clustering.

The above grouping scheme leads to the following:

- *Length*: Each group size $L^{(g_i)}$ satisfies

$$L^{(g_i)} = \sum_{m \in g_i} L^{(m)}, \quad \forall g_i \in G. \tag{4.9}$$

- *Group load index*: The group load index $d^{(g_i)}$ is given by

$$d^{(g_i)} = \frac{\sum_{m \in g_i} d^{(m)} L^{(m)}}{\sum_{m \in g_i} L^{(m)}}, \quad \forall g_i \in G. \tag{4.10}$$

After movie grouping, we run linear programming on these groups by treating them as $|G|$ "super movies" with load index $d^{(g_i)}$ and length $L^{(g_i)}$. In the phase of parameter discretization, we use the methods described in Section 4.1 to get the storage and retrieval parameters $n_v^{(g_i)}$ and $n_{uv}^{(g_i)}$ for each "super movies" g_i. Then we need an extra step to get the storage and retrieval parameters $n_v^{(m)}$ and $n_{uv}^{(m)}$ for each movie m.

In the server v, we have $n_v^{(g_i)}$ space to store all the movies $m \in g_i$. The guiding principle of our placement algorithm is that all the movies in the same group should have similar $n_v^{(m)}$. Accordingly, we use *rarest first* in symbol placement. Specifically, when a server v makes a symbol placement for a group g_i, it increases the smallest $n_v^{(m)}$ by 1 for movie $m \in g_i$ until the space budget of g_i is fully consumed. In our retrieval algorithm, we make $n_{uv}^{(m)} = n_{uv}^{(g_i)}$ for $m \in g_i$. If $n_{uv}^{(m)} > n_u^{(m)}$ for some u, we reduce $n_{uv}^{(m)}$ to make $n_{uv}^{(m)} = n_u^{(m)}$ and the remaining symbols are assigned to the repository.

General K-means clustering problem is usually regarded as NP-hard. However, K-means clustering in one dimension (our case) can be solved exactly in $O(|M|^2|G|)$ time by dynamic programming [31]. After we group the movies, the complexity to solve the linear program has been reduced to $O(|V|^6|G|^3)$. As the discretization still takes $O(|V|^2|M|)$ time, the total complexity is $O(|V|^6|G|^3 + |M|^2|G| + |V|^2|M|)$. In terms of the number of movies $|M|$, the complexity is reduced by a factor of $O(|M|)$. In addition, because $O(|V|^6)$ is usually quite large, for the running time of linear programming, the complexity reduction from $O(|V|^6|M|^3)$ to $O(|V|^6|G|^3)$ is also significant.

4.3 Re-optimization due to Changes in System Parameters

Due to changes in system parameters, a VoD network needs to be periodically "re-optimized" for the best performance. Such changes in system parameters may be due to, for example, the following:

- *Movie changes*: The VoD movie pool may be updated from time to time, due to movie introduction, movie removal, change in the expected movie popularity, etc.

- *Server changes*: Servers may be introduced, replaced or removed from the network. The storage and bandwidth of some of the servers may also be increased or reduced due to a change in longer-term user traffic.

- *Network changes*: Network transmission cost may change due to changes in network technologies or contractual terms in bandwidth.

Though we do not expect the system going through significant changes very frequently (e.g., on the daily or weekly basis), such change needs to be considered to achieve the best performance over time through symbol redistribution (retrieval mechanism is similar and hence will not be discussed here).

A strength of bucket-filling is that it is easily amendable to system changes, and does not lead to much overhead in symbol redistribution in times of re-optimizing the system. We present here a simple and efficient solution which involves only incremental network transmission upon system changes which requires no re-encoding or re-distributing all the movies. This is due to source coding we use — stored coded symbols can be independently re-used, added, or replaced to reduce symbol transmission. This greatly saves the re-optimization work of the network.

We redistribute the movie symbols when the system changes from time t to $t+1$. Let $\overrightarrow{n}_v(t) = [n_v^{(1)}(t), n_v^{(2)}(t), ..., n_v^{(|M|)}(t)]$ be the number of symbols for movie m that server v stores at time t ($\forall m \in M$). Let $\overrightarrow{\triangle}_v(t) = [\triangle_v^{(1)}(t), \triangle_v^{(2)}(t), ..., \triangle_v^{(|M|)}(t)]$ be the symbol difference between time t and $t+1$, given by $\overrightarrow{\triangle}_v(t) = \overrightarrow{n}_v(t+1) - \overrightarrow{n}_v(t)$ for all $v \in V$.

Therefore, if $\triangle_v^{(m)}(t) > 0$, the repository will transfer $|\triangle_v^{(m)}(t)|$ symbols of movie m to server v. On the other hand, if $\triangle_v^{(m)}(t) < 0$, server v will discard $|\triangle_v^{(m)}(t)|$ symbols of movie m.

To make the encoding more efficient, the repository may initially generate coded symbols once off-line by a generator matrix with some high $\{n_v^{(m)}\}$. This is to avoid encoding the same movies again under system changes. In this way, the repository only needs to send its pre-computed symbols to proxy servers upon system changes. Note that our symbol redistribution is a one-step process (and hence no convergence problem), and is able to maintain the optimal movie distribution among the servers while keeping the symbol transmission minimal.

CHAPTER 5

ILLUSTRATIVE SIMULATION RESULTS

5.1 Simulation Environment and Performance Metrics

In this chapter, we present our simulation environment and performance metrics to study the performance of bucket-filling.

Bucket-filling can be applied to any movie popularity. For concreteness in our simulation, we consider that movie popularity follows the Zipf distribution with Zipf parameter s, i.e., the request probability of the ith movie, denoted as $f(i)$, is given by $f(i) \propto 1/i^s$. In our simulation, we consider that requests arrive at each proxy server according to a Poisson process with total rate λ (req./second). It is clear from Section 4 that the optimization of bucket-filling does not depend on the specific request process at server v but its arrival rate λ_v. Therefore, the results and conclusions may be extended to any request processes or traces so long as they share the same request rate. The proxy servers have heterogeneous storage space and bandwidth following a Zipf distribution (independent of each other). The repository stores all the movies with a streaming capacity twice of the average streaming capacity of the proxy servers. The VoD network consists of a number of distributed proxy servers. All our results are obtained at steady state. Unless otherwise stated, we use the default values shown in Table 5.1 for our system parameters (the baseline case).

We consider the network cost function from server u to server v proportional to the bandwidth between them, i.e.,

$$C_{uv}^{\mathrm{N}}(\Gamma_{uv}) = c_{uv}\Gamma_{uv}, \quad \forall u, v \in V, \tag{5.1}$$

where c_{uv} is some constant (by definition, $c_{vv} = 0$).

The server cost is a function of its storage and its total bandwidth used to serve the remote servers, modeled as

$$C_v^{\mathrm{S}} = \sigma_B B_v + C_v(R_v), \quad \forall v \in V, \tag{5.2}$$

Table 5.1: Baseline parameters used in our study.

Parameter	Default value
q	30
Number of proxy servers	20
Number of movies	200
Average server storage	20 movies
Zipf parameter of server storage	0.4
Average proxy server bandwidth capacity	160 Mbits/s
Zipf parameter of server bandwidth	0 (i.e., same bandwidth)
Zipf parameter of movie popularity	0.6
Movie length	90 minutes
Average movie holding time	Movie length (i.e., $\alpha^{(m)} = 1$)
Movie streaming rate	1 Mbits/s
Total request rate in the network	0.6 req./s (equally distributed to the proxies)
c_{uv} between central and proxy server	0.01 unit/s
c_{uv} between proxies	Zipf with parameter 0.6 and mean 0.005 unit/s

where σ_B is a constant ($\sigma_B = 3.33 \times 10^{-6}$ in our simulation), and $C_v(R_v)$ is a piece-wise linear function monotonically increasing in R_v. We show in Figure 5.1 $C_v(R_v)$ versus R_v/U_v in our simulation, where U_v is the streaming capacity of the server and hence R_v/U_v is the bandwidth utilization of the server. There are three linear segments formed by points $(0, 0)$, $(0.8, 0.125)$, $(0.93, 0.4375)$ and $(0.99, 1.925)$ (these coordinates are obtained from the queueing model $\sigma_S/(U_v - R_v)$, where σ_S is some constant). The cost increases with the bandwidth utilization. As the consumed bandwidth R_v approaches the bandwidth capacity U_v, the server cost increases more sharply.

To validate the streaming cost in our simulation setting, we have conducted an experiment to measure the queueing time of a server given the bandwidth utilization levels.[1] Figure 5.1 shows that the experiment results match our queueing model well.

We validate our storage and network transmission cost in our simulation with the data of Google Cloud Platform [14]. We show in Table 5.2 the network transmission cost in North America and Asia, which indicates that linear functions fit the price well. The cloud platform also indicates the storage cost as fixed at $0.026 for a GB per month, which agrees well with our simulation setting.

The performance metrics we are interested in are:

- *Total cost* (unit/s), which is the sum of server cost and network cost according to Equa-

[1] We use a 64-bit ThinkCenter with Intel Core 2 4400 CPU 2.00 GHz and RAM 1.00 GB. The unit is in second.

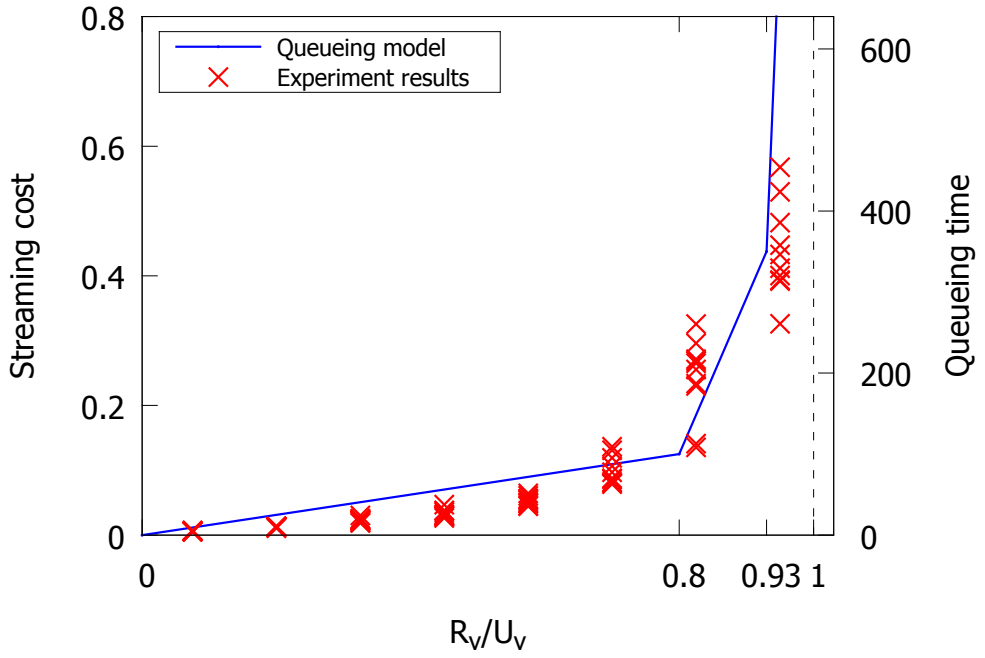

Figure 5.1: Streaming cost model at proxy server.

Table 5.2: Network Transmission Cost.

Used per month	North America (per GB)	Asia (per GB)
Up to 1TB	$ 0.12	$ 0.21
Up to 9TB	$ 0.11	$ 0.18
Up to 90TB	$ 0.08	$ 0.15

tion (3.9). This is the deployment cost of the network.

- *Server cost* (unit/s), which is the sum of its storage and streaming defined in Equations (3.8) and (5.2). We further examine the following cost components:
  - *Storage cost*, which is the total cost due to server storage.
  - *Streaming cost*, which is the server bandwidth cost to support other servers.
- *Network cost* (unit/s), which is the network transmission cost defined in Equations (3.6) and (5.1).
- *Movie cost* (unit/s), which is the average cost to access movie m.
- *Computation time*, which is the running time of the optimization.[2]

We compare bucket-filling with the following traditional and recent movie replication schemes:

- *Random*, where each server randomly stores movies without considering their popularity. This is a simple storage strategy.
- *MPF (Most Popular First)*, where each server stores the most popular movies. This is a greedy strategy, but does not take advantage of cooperative replication.
- *Local Greedy* [4], which divides the movies into three categories: those popular ones which all servers store (full replication), those medium popular ones which only one proxy server store (single copy), and those unpopular ones which only the repository stores (no copy). By formulating an LP problem, it seeks to minimize network cost. As Local Greedy assumes homogeneous access cost, we set its access cost to be equal to the average access cost between servers in our network.
- *LP-SR* [35, 6], which partitions movies into segments for storage and retrieval. LP-SR is also based on linear programming to achieve closely optimal solution.

[2] As the running time depends on the machine used, we have normalized the time in terms of some unit. For a 64-bit ThinkPad T420s with Intel i7 2620M CPU 2.70 GHz and RAM 8.00 GB, the unit is in second.

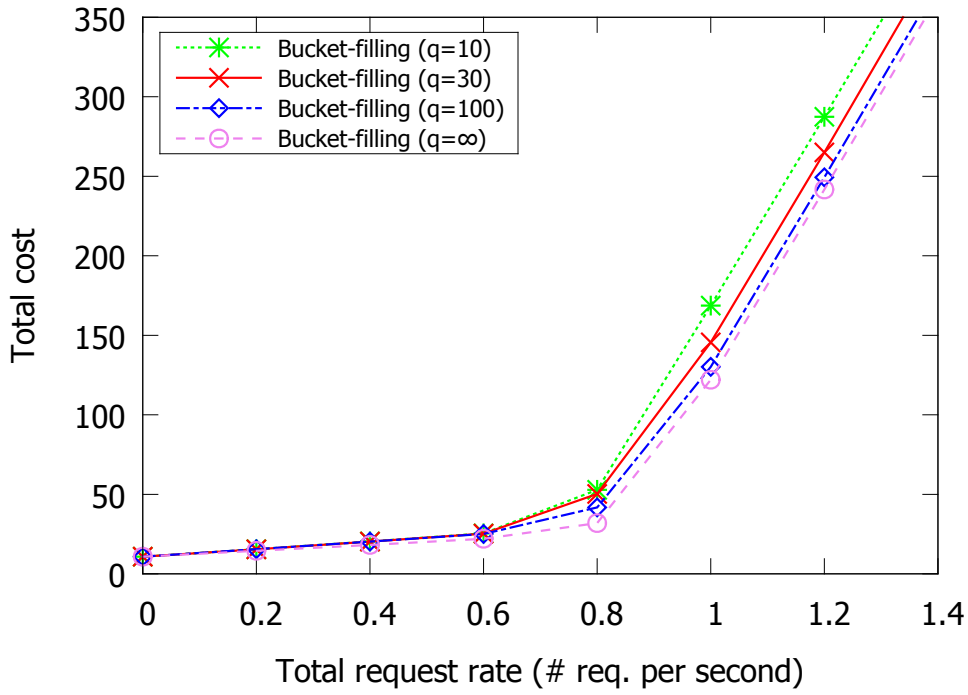

Figure 5.2: Total cost versus request rate given q.

For all the comparison schemes except LP-SR, upon a miss request, the home server v chooses an available server u which has the requested content with a probability proportional to $1/c_{uv}$. It is a reasonable, simple and effective strategy because the server with lower access cost has higher chance to be chosen. With this probabilistic approach, a server with low access cost is not always selected so as to avoid congestion, and hence high streaming cost, at the server.

5.2 Bucket-filling Performance

We plot in Figure 5.2 the total cost versus request rate given q. The total cost increases with the request rate mainly because of the increase in network traffic. As q increases, the network approaches the *exact optimum* given by linear programming (corresponding to the case $q \to \infty$). However, for humble value of q (say 30), the performance is already very close to the optimum (less than 6% deviation in this). This shows that our network is highly efficient, with closely optimal performance even for all the practical (finite) value of q.

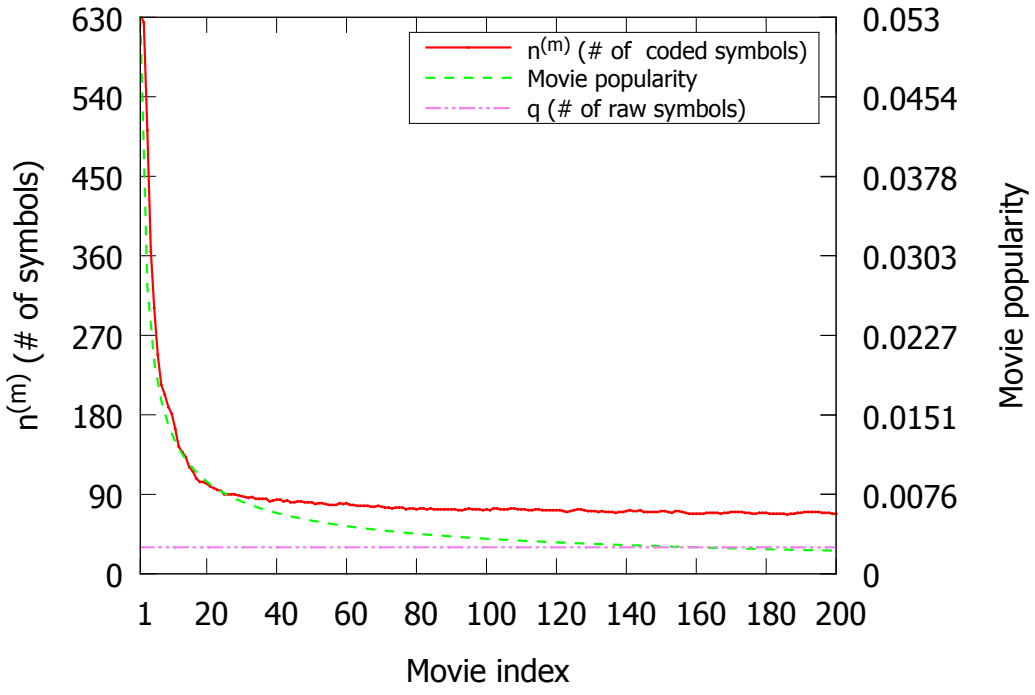

Figure 5.3: Optimal $n^{(m)}$ versus movie index.

We show in Figure 5.3 the optimal $n^{(m)}$ versus movie index. Also shown is the corresponding movie popularity (default setting) and the number of raw/source symbols q. We see that the movie popularity exhibits some skewness with a tail (with $s = 0.6$ and $M = 200$, the top 30% of the movies account for close to 60% of the total traffic). The optimal $n^{(m)}$ decreases with movie popularity. This is reasonable because the servers tend to locally store more of those popular movies to reduce transmission cost in the network. For the unpopular ones, fewer symbols are generated and stored in the whole network. We see that no matter how unpopular the movie is, the number of symbols is higher than q, meaning that some symbols are stored in the network besides those at the repository.

We show in Figure 5.4 the cost components (server streaming, server storage and network traffic) and total cost versus proxy storage (the average storage at proxies). The total cost falls off quite sharply initially but rises up gradually again, showing a minimum at some storage point. At the beginning when the proxy servers have little storage, all the traffic concentrates on the repository, leading to high overall streaming cost. As proxy storage increases, the repository load

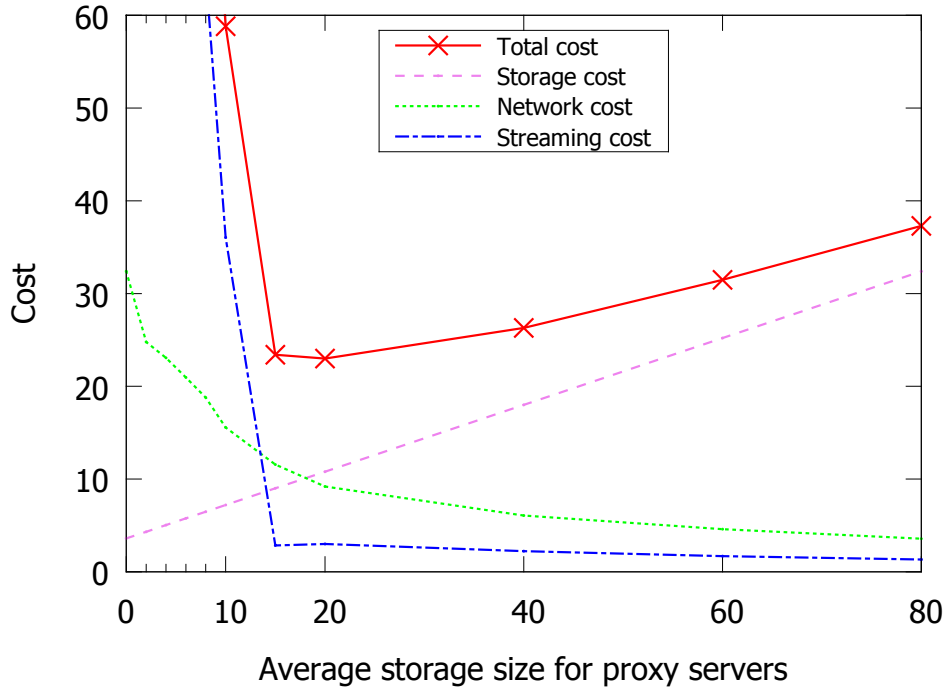

Figure 5.4: Total cost and the cost components versus proxy storage.

decreases and so does the streaming and network transmission cost. As storage further increases, storage cost becomes the major cost component.

We compare in Figure 5.5 the total cost versus the request rate for different schemes. Total cost increases with request rate mainly due to the increase in network traffic. Bucket-filling clearly achieves much lower total cost among all the schemes, beating them by multiple times except LP-SR. In other words, given the same deployment budget, bucket-filling can support much higher request rate (i.e., more concurrent users in the system). MPF does not perform well because it mainly relies on the central server to serve the requests for the unpopular movies. Random, due to its popularity-blind nature, stores insufficient copy of the popular movies, leading to considerable cost. As Local Greedy is based on LP optimization and has a proven upper bound of optimality gap, it performs substantially better as compared with other approaches in practice (MPF and Random). Bucket-filling achieves by far the best performance (near optimality) because of its use of source coding to achieve design simplicity and comprehensive consideration of every components of the deployment cost. Though bucket-filling is similar and slightly better than the state-of-the-art

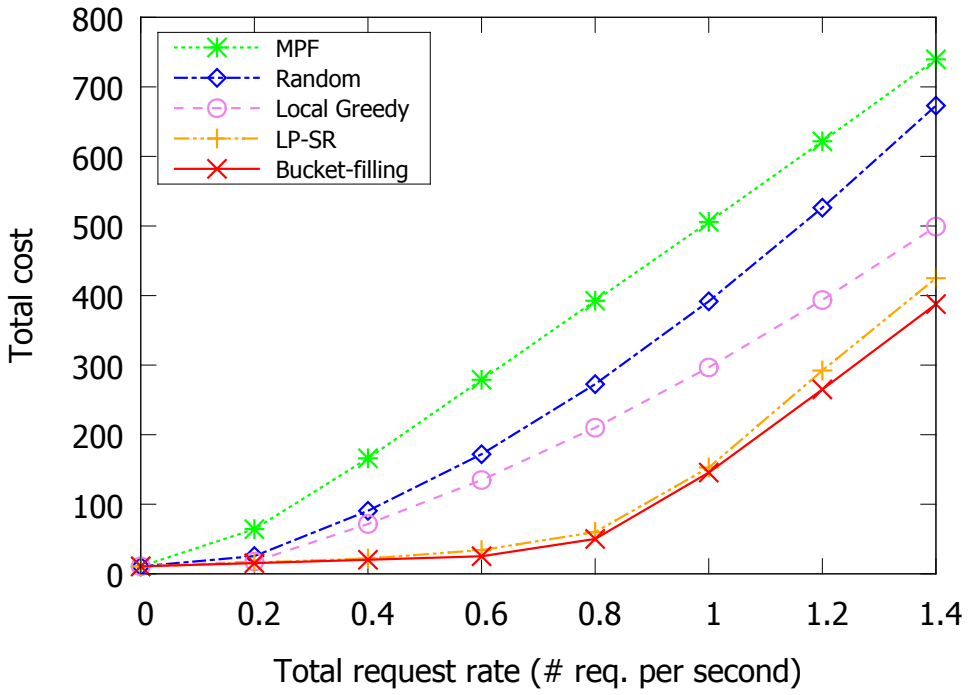

Figure 5.5: Total cost versus request rate given different schemes.

scheme LP-SR, it has much lower computation time as shown in Figure 5.6.

We show in Figure 5.6 the computation time of bucket-filling and LP-SR for different movie numbers. The computation time increases with the total number of movies. It is because both bucket-filling and LP-SR capture the information of every movie, and hence increasing movie number introduces more variables to the linear program. Unlike LP-SR which considers every movie segment in the linear program, bucket-filling has a simple and efficient discretization algorithm with orders of magnitude improvement in running time. Bucket-filling can be readily and practically applied to a VoD network with thousands of movies which is not generally feasible with LP-SR. Due to the complexity of LP-SR, we do not further compare it in the following figures.

We plot in Figure 5.7 the total cost versus the Zipf parameter of movie popularity given different schemes. The total cost in general decreases with the skewness. This is because more requests are concentrated on fewer popular movies, which leads to lower miss rate, and hence lower streaming and network cost. Bucket-filling achieves substantially the lowest total cost even for low skewness (i.e. when the popularity is quite uniform). This shows that bucket-filling makes good decision on

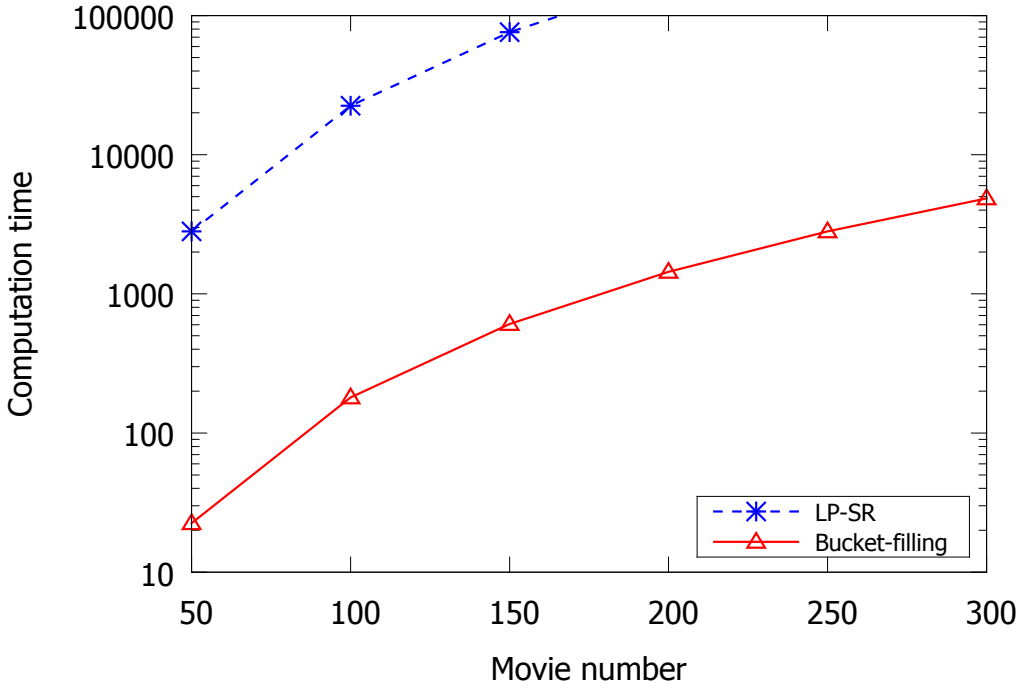

Figure 5.6: Computation time versus movie number given different schemes.

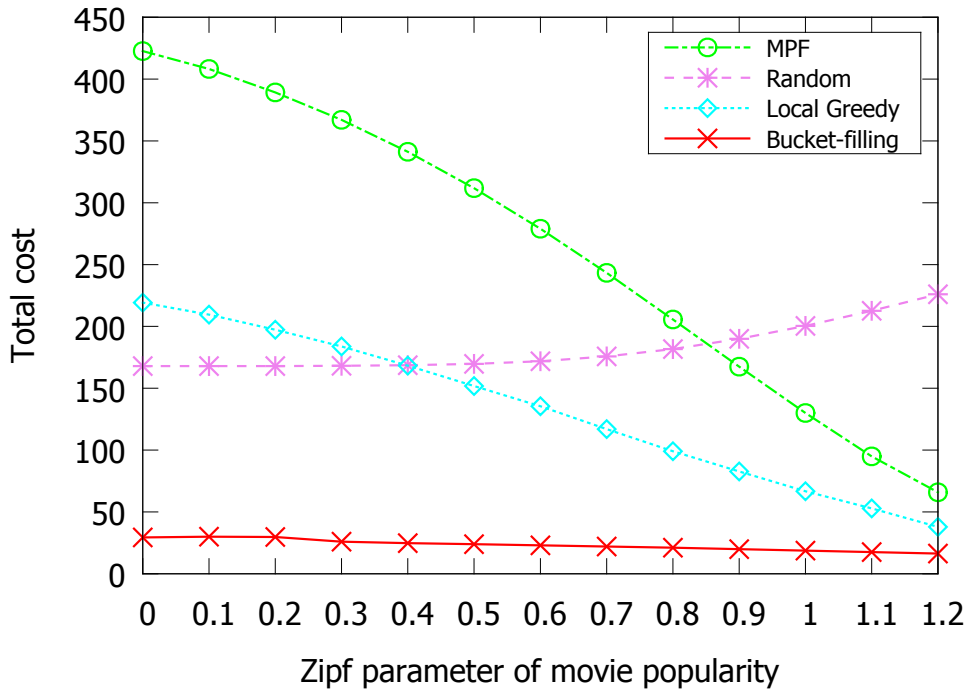

Figure 5.7: Total cost versus Zipf parameter of movie popularity given different schemes.

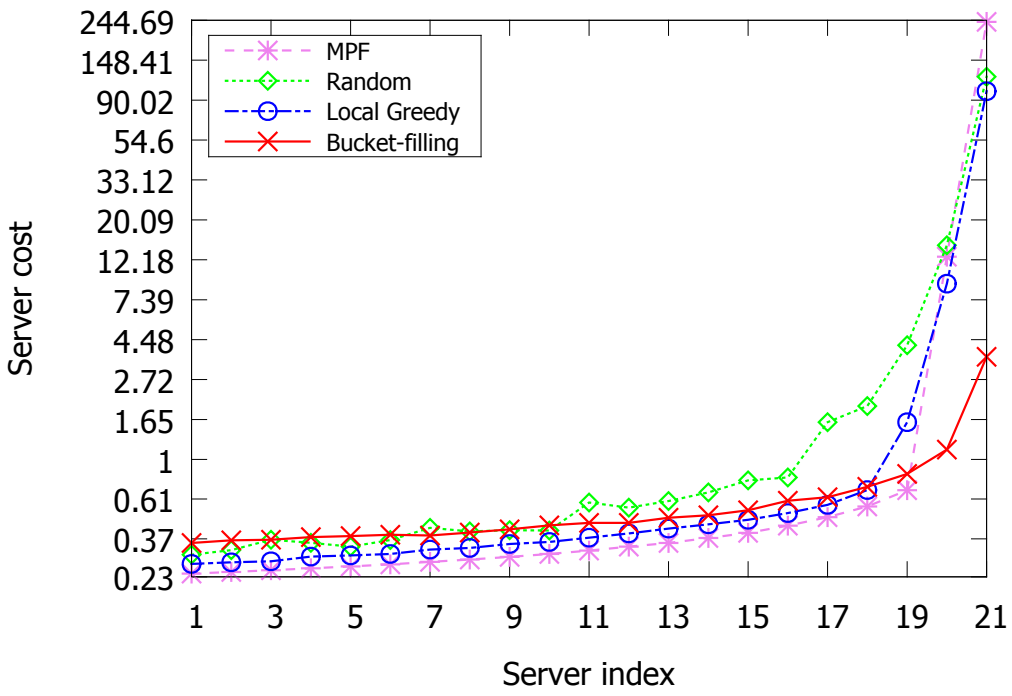

Figure 5.8: Server cost distribution given different schemes.

movie storage and retrieval. Local Greedy performs better than MPF because it takes network cost into consideration. The cost of Random increases with skewness because it is popularity-blind. The popular movies, due to the fact that its copy does not increase with its popularity, suffers from high streaming and network cost.

We plot in Figure 5.8 individual server cost for different schemes. We sort the proxy servers according to their storage in ascending order (as their streaming capacity is the same in the default setting), and the last one refers to the repository. As the storage of a proxy increases, its cost increases because it needs to serve more remote requests. Bucket-filling utilizes very well the finite storage and bandwidth resources of proxy servers, leading to significantly lower repository streaming cost. It has strong server cooperation to achieve near optimal system performance. As MPF only stores the most popular movies at the proxy servers, it has lower proxy cost at the steep sacrifice of repository cost. The proxies barely contribute their bandwidth and storage to cooperatively help each other. Local Greedy, with network cost optimization, outperforms Random in both proxy server cost and repository cost.

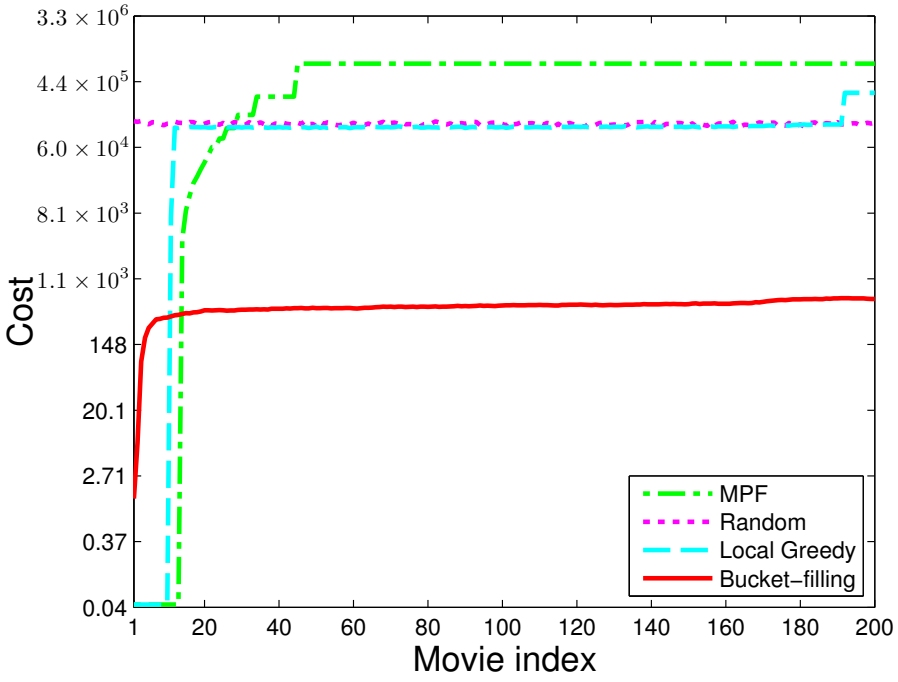

Figure 5.9: Cost of each movie given different schemes.

We show in Figure 5.9 the cost of each movie for different schemes. The movies are sorted according to their popularity in descending order. The popularity-based schemes (i.e., bucket-filling, Local Greedy and MPF) tend to locally store the popular movies, and hence those popular ones enjoy lower cost at much sacrifice of those not-so-popular movies. Bucket-filling makes better movie storage decision by cooperatively storing unpopular movies. While bucket-filling has slightly higher cost for popular movies, most of the movies have quite uniform access cost. Bucket-filling accomplishes much better optimality with the cost of unpopular movies strikingly much lower by orders of magnitude than the other schemes. This is the main factor of its success. For MPF, its high cost mainly comes from the less popular movies. Random treats each movie equally and thus has the most uniform cost distribution.

5.3 Movie Grouping with K-means Clustering

We conduct simulation to study the performance of our grouping algorithm. For Figure 5.10, we use the same baseline parameters as given in Section 5.1 with the group number $|G| = 10$. Due to the

Table 5.3: Parameters for large movie pool.

Parameter	Value
Number of movies	10000
Total request rate in the network	30 req./s (equally distributed to the proxies)
c_{uv} between central and proxy server	0.0002 unit/second
c_{uv} between proxies	Zipf parameter 0.6 and mean 0.0001 unit/s
Average server storage	1000 movies
Average proxy server bandwidth capacity	8000 Mbits/s
Unit storage cost σ_B	6.66×10^{-8}

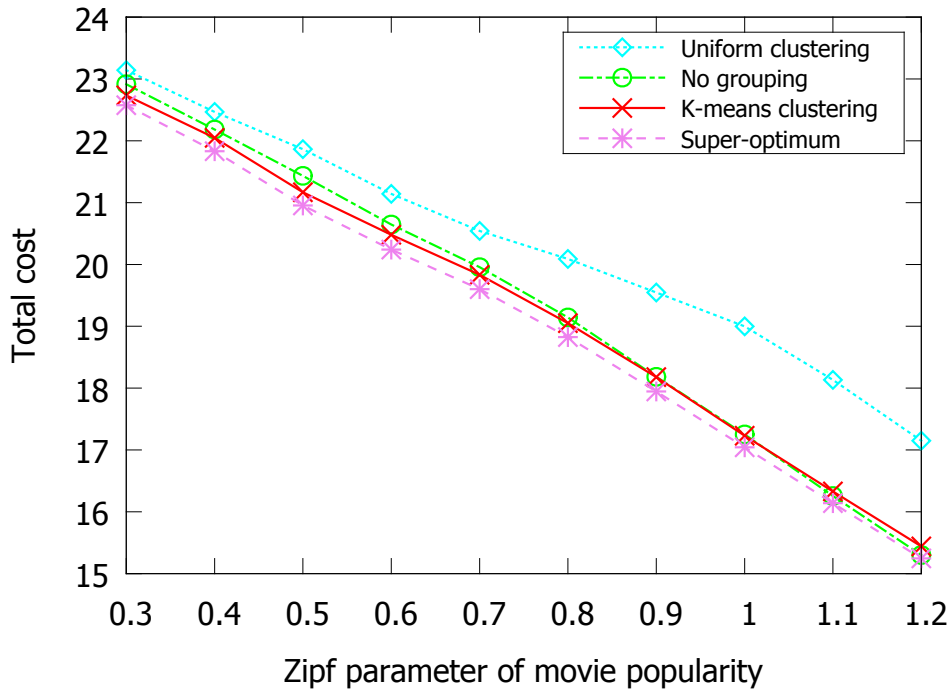

Figure 5.10: Total cost versus Zipf parameter of movie popularity given different schemes.

time complexity of linear programming, we cannot get the *super-optimum* with a very large movie pool. For Figure 5.11 and Figure 5.12, we use the parameters shown in Table 5.3 for large movie pool. We choose 3 comparison schemes. *Super-optimum* is the result from the linear programming and serves as a lower bound. *No grouping* is the performance of bucket-filling without any grouping scheme. In *uniform clustering*, we still group the movies with similar load indices together, but the number of movies in each group is the same.

We plot in Figure 5.10 the total cost versus the Zipf parameter of movie popularity given d-

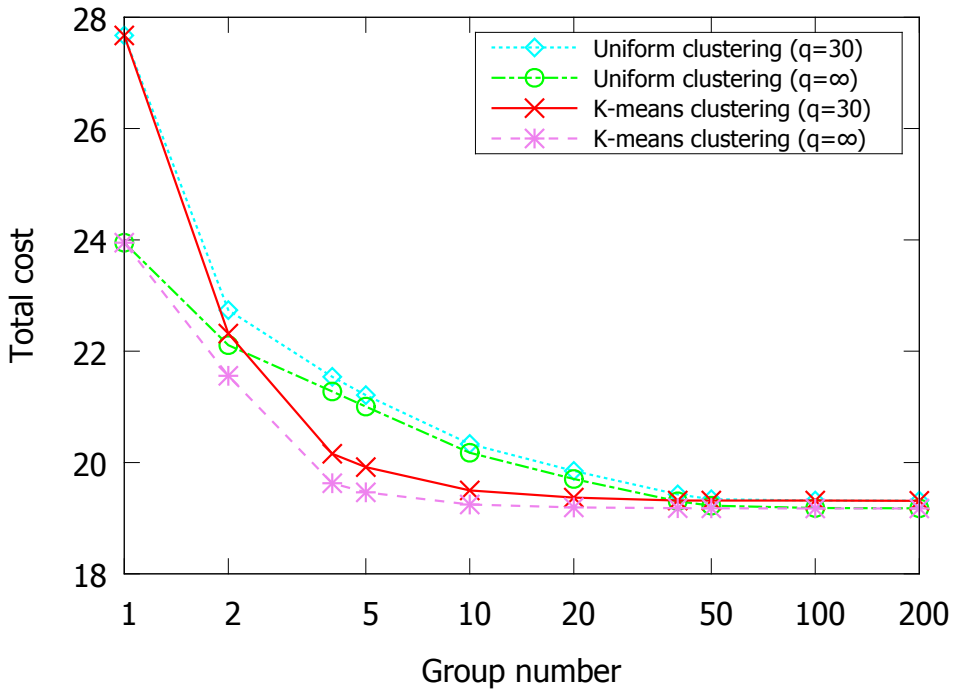

Figure 5.11: Total cost versus group number given different schemes.

ifferent schemes. Total cost goes down with the Zipf parameter mainly because there are fewer requests for not-so-popular movies, which usually have higher cost. K-means clustering performs better for large Zipf parameter. This is because, when Zipf parameter is small, the load index difference among movies is also small. Therefore, K-means clustering and uniform clustering will give similar grouping results. Note that in some cases K-means clustering may slightly outperform no grouping because, while the grouping scheme approximates the exact movie load index, a smaller group number indicates less discretization and less performance loss due to such discretization.

We show in Figure 5.11 the total cost of bucket-filling given different group sizes and grouping schemes. Due to the large movie number, it is impossible to calculate the *super-optimum*. The total cost decreases with the total number of groups. It is because, with more groups, the load indices in the same group are closer to each other. For K-means clustering, the total cost achieves satisfactory level even with a small group number ($|G| = 5$) and K-means clustering outperforms with a 10% margin compared with uniform clustering. As the group number increases, both K-means clustering and uniform clustering converge to the same value given by the case $|G| = |M|$.

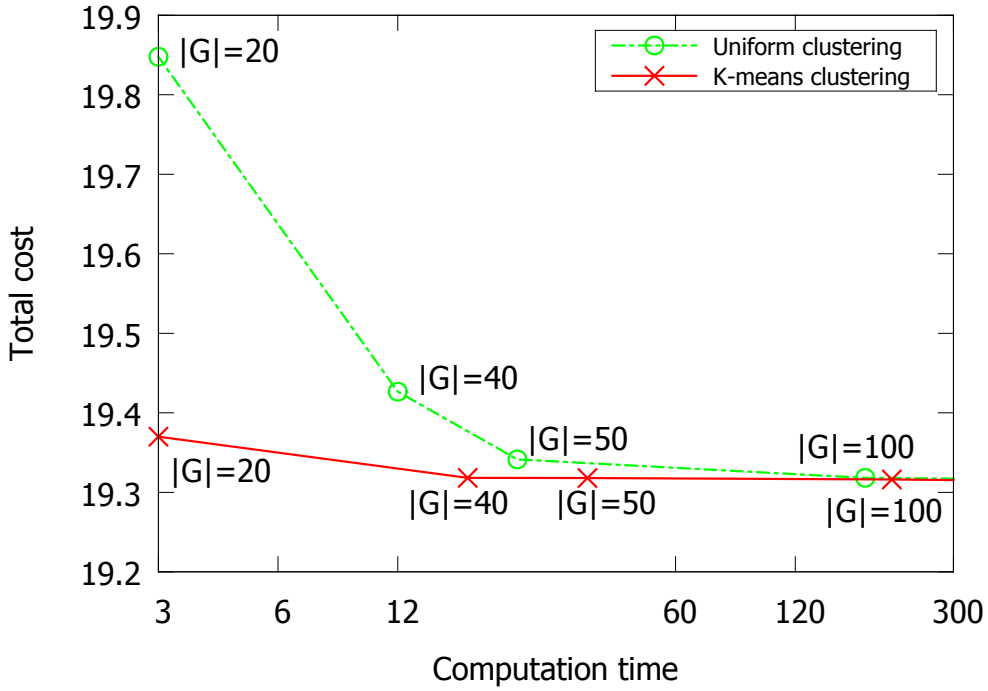

Figure 5.12: Total cost versus computation time given different schemes.

We next examine the total running time to compute the bucket-filling solution with grouping algorithms. We plot in Figure 5.12 the tradeoff curve between total cost and running time for movie grouping. K-means clustering already achieves good performance even for very short time and increasing running time has little impact on performance. As running time increases, uniform clustering achieves better performance and the gap between two grouping schemes becomes very small. Since for both K-means clustering and uniform clustering, the running time is quite short. The time complexity would not be a bottleneck for a VoD network with bucket-filling implementation.

5.4 Re-optimization due to System Changes

We conduct simulation to study the number of symbols which need to be transferred from the repository (i.e., the overhead) upon movie changes. We use the same baseline parameters as given in Section 5.1. The approach for server or network change is similar and will not be discussed here for brevity.

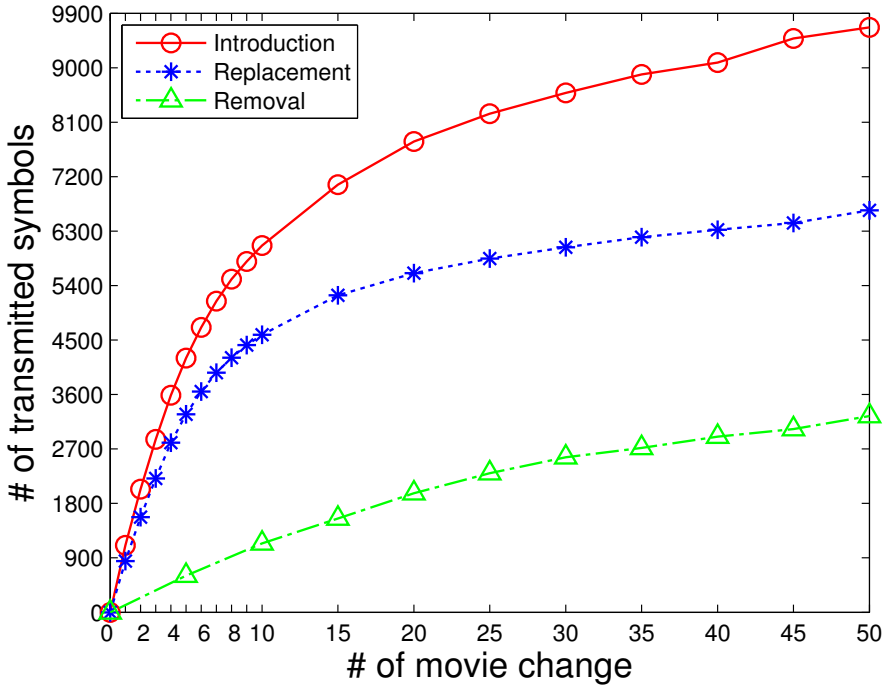

Figure 5.13: Number of transmitted symbols versus number of movie change.

We mainly study the changes in movie as follows (while using the same Zipf distribution) during the changes:

- *Introduction*: The additional movies are the most popular ones, and the "older" movies have lower popularity due to such movie introduction.
- *Replacement*: The most unpopular movies are *replaced* with new movies, which are the most popular.
- *Removal*: The most unpopular movies are removed.

We show in Figure 5.13 the number of transferred symbols versus the number of movie change. The number of transferred symbols increases with the number of movie change due to the larger scale of the system change. As the number of movie change increases, the transmission increases sub-linearly. This is because there are more not-so-popular movies in the system change, and hence the number of transmitted symbols does not increase linearly. Movie introduction leads to the

highest symbol transmission because popular movies are added, which means that the proxies have to make much room to store them by replacing others. Movie replacement transfers fewer symbols than introduction because the removed movies have already made room for the newly introduced movies. Movie removal has the least number of transferred symbols because the additional space released by the removed movies is used to store more of the other movies, leading to less shuffling, and hence transmission, of symbols.

CHAPTER 6

CONCLUSION

In this work, we have proposed and studied a VoD network based on source coding, and studied optimal movie distribution and retrieval to minimize deployment cost. Movies, encoded in symbols with a parameter q, are distributed and retrieved efficiently using a "bucket-filling" algorithm. The deployment cost captures the costs of server streaming, server storage and network transmission cost. We have presented a solution which asymptotically achieves the exact optimum as q increases.

We have formulated the optimization problem for large q as a linear program which can be solved efficiently. For finite q, we have presented a discretization process which is *closely and asymptotically optimal*. For very large movie pool, we have proposed a movie grouping algorithm based on K-means clustering which greatly reduces the running time with closely optimal performance. We have also presented an efficient on-line re-optimization method to ensure good performance with low symbol redistribution overhead when system parameters change in the VoD network.

We have conducted extensive simulation to compare bucket-filling performance with other traditional and state-of-the-art schemes. The results show that our scheme achieves close optimality with much lower cost, and outperforms the other schemes by a wide margin (multiple times in many cases, and more than 100% in most cases). Our results show that many previously proposed heuristics performs quite far from the optimum, and our VoD network can achieve performance arbitrarily close to the optimum (depending on the coding complexity and decoding delay one is willing to accept).

REFERENCES

[1] Vijay Kumar Adhikari, Yang Guo, Fang Hao, Matteo Varvello, Volker Hilt, Moritz Steiner, and Zhi-Li Zhang. Unreeling netflix: Understanding and improving multi-cdn movie delivery. In *Proceedings of IEEE INFOCOM 2012*, pages 1620–1628, 2012.

[2] Amr Alasaad, Kaveh Shafiee, Sathish Gopalakrishnan, and Victor Leung. Prediction-based resource allocation in clouds for media streaming applications. In *Globecom Workshops (GC Wkshps), 2012 IEEE*, pages 753–757. IEEE, 2012.

[3] David Applegate, Aaron Archer, Vijay Gopalakrishnan, Seungjoon Lee, and KK Ramakrishnan. Content placement via the exponential potential function method. In *Integer Programming and Combinatorial Optimization*, pages 49–61. Springer, 2013.

[4] S. Borst, V. Gupta, and A. Walid. Distributed caching algorithms for content distribution networks. In *Proceedings of IEEE INFOCOM 2010*, pages 1–9, March 2010.

[5] S.-H. Gary Chan and Fouad Tobagi. Distributed servers architecture for networked video services. *IEEE/ACM Transactions on Networking*, 9(2):125–136, April 2001.

[6] Shueng-Han Gary Chan and Zhuolin Fannie Xu. LP-SR: Approaching optimal storage and retrieval for video-on-demand. *IEEE Transactions on Multimedia*, 15:2125–2136, December 2013.

[7] Shueng-Han Gary Chan and Zhuolin Fannie Xu. Optimizing video-on-demand with source coding. In *Proceedings of the 2013 International Conference on Multimedia and Expo (ICME)*, San Jose, CA, USA, 15-19 July 2013.

[8] Le Chang and Jianping Pan. Reducing the overhead of view-upload decoupling in peer-to-peer video on-demand systems. In *IEEE International Conference on Communications (ICC)*, pages 1–5, June 2011.

[9] Yung R. Choe, Derek L. Schuff, Jagadeesh M. Dyaberi, and Vijay S. Pai. Improving VoD server efficiency with bittorrent. In *MULTIMEDIA '07: Proceedings of the 15th international conference on Multimedia*, pages 117–126, New York, NY, USA, 2007. ACM.

[10] Y.-M. Chu, N.-F. Huang, and S.-H. Lin. Quality of service provision in cloud-based storage system for multimedia delivery. *Systems Journal, IEEE*, PP(99), 2013.

[11] Jie Dai, Zhan Hu, Bo Li, Jiangchuan Liu, and Baochun Li. Collaborative hierarchical caching with dynamic request routing for massive content distribution. In *Proceedings of IEEE INFOCOM 2012*, pages 2444–2452, March 2012.

[12] C. Dana, D. Li, D. Harrison, and C. N. Chuah. BASS: Bittorrent assisted streaming system for video-on-demand. In *Multimedia Signal Processing, 2005 IEEE 7th Workshop on*, pages 1–4, November 2006.

[13] Christos Gkantsidis, John Miller, and Pablo Rodriguez. Comprehensive view of a live network coding P2P system. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 177–188. ACM, 2006.

[14] Google. Google Cloud Platform. `https://cloud.google.com/products/cloud-storage/`, 2014. [Online; accessed 6-May-2014].

[15] M. Grant and S. Boyd. CVX: Matlab software for disciplined convex programming, version 1.21. `cvxr.com/cvx`, April 2011.

[16] Yifeng He, I. Lee, and Ling Guan. Distributed throughput maximization in P2P VoD applications. *IEEE Transactions on Multimedia*, 11(3):509–522, April 2009.

[17] Yifeng He, Guobin Shen, Yongqiang Xiong, and Ling Guan. Optimal prefetching scheme in P2P VoD applications with guided seeks. *IEEE Transactions on Multimedia*, 11(1):138–151, January 2009.

[18] M. Hefeeda and B. Noorizadeh. On the benefits of cooperative proxy caching for peer-to-peer traffic. *Parallel and Distributed Systems, IEEE Transactions on*, 21(7):998–1010, July 2010.

[19] Cheng Huang, Jin Li, and Keith W. Ross. Can internet video-on-demand be profitable? In *SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 133–144, New York, NY, USA, 2007. ACM.

[20] J. Kangasharju, K. W. Ross, and D. A. Turner. Optimizing file availability in peer-to-peer content distribution. In *26th IEEE International Conference on Computer Communications (INFOCOM)*, pages 1973–1981, May 2007.

[21] Y. Kao, C. Lee, P. Wu, and H. Kao. A network coding equivalent content distribution scheme for efficient peer-to-peer interactive VoD streaming. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1, 2011.

[22] Fangming Liu, Shijun Shen, Bo Li, Baochun Li, Hao Yin, and Sanli Li. Novasky: Cinematic-quality VoD in a P2P storage cloud. In *Proceedings of IEEE INFOCOM 2011*, pages 936–944, April 2011.

[23] Zimu Liu, Chuan Wu, Baochun Li, and Shuqiao Zhao. Uusee: large-scale operational on-demand streaming with random network coding. In *Proceedings of IEEE INFOCOM 2010*, pages 1–9, 2010.

[24] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. California, USA, 1967.

[25] M. Mehyar, D. Spanos, J. Pongsajapan, S. H. Low, and R. M. Murray. Asynchronous distributed averaging on communication networks. *IEEE/ACM Transactions on Networking*, 15(3):512–520, June 2007.

[26] A. Nimkar, C. Mandal, and C. Reade. Video placement and disk load balancing algorithm for VoD proxy server. In *IEEE International Conference on Internet Multimedia Services Architecture and Applications (IMSAA)*, pages 1–6, December 2009.

[27] Di Niu, Hong Xu, Baochun Li, and Shuqiao Zhao. Quality-assured cloud bandwidth auto-scaling for video-on-demand applications. In *Proceedings of IEEE INFOCOM 2012*, pages 460–468, 2012.

[28] Hyung Rai Oh, Dapeng Oliver Wu, and Hwangjun Song. An effective mesh-pull-based P2P video streaming system using fountain codes with variable symbol sizes. *Computer Networks*, 55(12):2746–2759, 2011.

[29] J. F. Sturm. *Primal-dual interior point approach to semidefinite programming*. PhD thesis, Erasmus Universiteit Rotterdam, 1997.

[30] Bo Tan and Laurent Massoulié. Optimal content placement for peer-to-peer video-on-demand systems. *IEEE/ACM Transactions on Networking (TON)*, 21(2):566–579, 2013.

[31] H. Wang and M. Song. Ckmeans.1d.dp: Optimal k-means clustering in one dimension by dynamic programming. *The R Journal*, 3:29–33, 2011.

[32] Mea Wang and Baochun Li. R2: Random push with random network coding in live peer-to-peer streaming. *Selected Areas in Communications, IEEE Journal on*, 25(9):1655–1666, December 2007.

[33] D Wu, J He, Yupeng Zeng, Xiaojun Hei, and Yonggang Wen. Towards optimal deployment of cloud-assisted video distribution services. *IEEE Transactions on Circuits and Systems for Video Technology*, 2013.

[34] Weijie Wu and J. C. S. Lui. Exploring the optimal replication strategy in P2P-VoD systems: Characterization and evaluation. In *Proceedings of IEEE INFOCOM 2011*, pages 1206–1214, April 2011.

[35] Z. Xu and S.-H. Gary Chan. LP-based optimization of storage and retrieval for distributed video-on-demand. In *Proceedings of Globecom 2012 - Communications Software, Services and Multimedia Symposium*, pages 2161–2166, 3-7 December 2012.

[36] Wai-Pun Ken Yiu, Xing Jin, and S.-H. Gary Chan. VMesh: Distributed segment storage for peer-to-peer interactive video streaming. *IEEE Journal on Selected Areas in Communication-*

s Special Issue on Advances in Peer-to-Peer Streaming Systems, 25(9):1717–31, December 2007.

[37] S. Zaman and D. Grosu. A distributed algorithm for the replica placement problem. *IEEE Transactions on Parallel and Distributed Systems*, 22(9):1455–1468, September 2011.

[38] Y. Zhou, T. Z. J. Fu, and D. M. Chiu. On replication algorithm in P2P VoD. *IEEE/ACM Transactions on Networking*, PP(99):1, 2012.

[39] Yipeng Zhou, T. Z. J. Fu, and Dah Ming Chiu. A unifying model and analysis of P2P VoD replication and scheduling. In *Proceedings of IEEE INFOCOM 2012*, pages 1530–1538, March 2012.

APPENDIX A

LINEAR SOURCE CODING AND ITS USE IN BUCKET-FILLING

In linear source coding (LSC), q source symbols are linearly combined to form n ($n \geq q$) coded symbols such that the original data can be recovered from any q out of the n coded symbols. We discuss in the following how bucket-filling makes use of LSC.

There is a *code generator matrix* $\boldsymbol{G}$ given by

$$\mathbf{G} = \begin{bmatrix} 1 & g_1 & g_1^2 & \dots & g_1^{q-1} \\ 1 & g_2 & g_2^2 & \dots & g_2^{q-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & g_n & g_n^2 & \dots & g_n^{q-1} \end{bmatrix}, \tag{A.1}$$

where g_i are the non-zero coefficients from the finite field $\mathbf{Z}_p (p > n)$ for some prime p. Denote the q source symbols as $\boldsymbol{X} = [x_1, x_2, ..., x_q]$, $x_i \in \mathbf{Z}_p$. Let the n coded symbols be $\boldsymbol{Y} = [y_1, y_2, ..., y_n]$, $y_i \in \mathbf{Z}_p$. For each movie, its coded symbol y_i is computed once from the source symbols by

$$y_{\mathbf{i}} = \sum_{j=0}^{q-1} g_{\mathbf{i}}^j x_j \bmod p, \tag{A.2}$$

or equivalently, $\boldsymbol{GX} = \boldsymbol{Y}$.

To apply LSC in our VoD network, we take $n = \max_m n^{(m)}$. The system-wide parameters of coding coefficients $[g_1, g_2, ..., g_n]$ and p can be initially transmitted to all the proxies, so that they know the code generator matrix in advance. This only needs to be transmitted once at server join time. There is no need to change the coefficients upon changes in system parameters, and coefficients only need to be incrementally transmitted if the newly optimized system has a higher n. For movie m, the server only needs to use the subset $[g_1, g_2, ..., g_n]$ to decode the source packets.

With the knowledge of G, anyone in the network only needs to receive no less than q distinct y_i to derive the original data. Let $i_1, i_2, ..., i_q$ be q distinct integers in the set $\{1, 2, ..., n\}$. Then the following *Vandermonde matrix*

$$A = \begin{bmatrix} 1 & g_{i_1} & g_{i_1}^2 & \cdots & g_{i_1}^{q-1} \\ 1 & g_{i_2} & g_{i_2}^2 & \cdots & g_{i_2}^{q-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & g_{i_n} & g_{i_n}^2 & \cdots & g_{i_q}^{q-1} \end{bmatrix} \tag{A.3}$$

is invertible over the finite field $\mathbf{Z}_p$.

Suppose that coded symbols $y_{i_1}, y_{i_2}, ...y_{i_q}$ are received and denoted as $\boldsymbol{Y'} = [y_{i_1}, y_{i_2}, ...y_{i_q}]$. Since A is invertible, solving $\boldsymbol{AX} = \boldsymbol{Y'}$ gives source content $\boldsymbol{X}$.

APPENDIX B

PROOF OF ASYMPTOTIC OPTIMALITY OF BUCKET-FILLING

In Section 3.2, we have a continuous solution of VoD optimization problem given by linear program. This solution serves as the *super-optimum* (lower bound) since, for any q, the practical solution always satisfies the constraints given by Section 3.2 and cannot be better than the LP solution. For a particular q, the solution that achieves the lowest cost is the *exact optimum*. Bucket-filling also gives a solution for this q, but it may not be the *exact optimum*. It is not computationally feasible to know the *exact optimum* due to the NP-hardness of this VoD optimization problem. By definition, the total deployment cost given by bucket-filling (C_{BF}), *exact optimum* (C_{EO}) and *super-optimum* (C_{SO}) satisfies

$$C_{\mathrm{SO}} \leq C_{\mathrm{EO}} \leq C_{\mathrm{BF}}. \tag{B.1}$$

To prove the asymptotic optimality, if we can show that bucket-filling approaches *super-optimum* as q increases, it also approaches *exact optimum*. For clarity, we denote the symbols related to *super-optimum* (i.e., the continuous linear program) with superscript tildes. For bucket-filling, we use plain symbols. We also use Δ to denote the difference of a parameter between *super-optimum* and bucket-filling.

Claim: $\forall \delta > 0\ \exists q$ s.t. $\Delta C = |C - \tilde{C}| \leq \delta$.

Proof: From Equations (3.5) and (4.3) we get

$$\Gamma_{uv} = \frac{1}{q} \sum_{m \in M} p^{(m)} \lambda_v \alpha^{(m)} n_{uv}^{(m)} L^{(m)} s, \quad \forall u, v \in V. \tag{B.2}$$

If u is not the repository, $n_{uv}^{(m)}$ is rounded up or down by at most 1. Therefore,

$$\Delta\Gamma_{uv} = |\Gamma_{uv} - \tilde{\Gamma}_{uv}| \leq \frac{1}{q} \sum_{m \in M} p^{(m)} \lambda_v \alpha^{(m)} L^{(m)} s. \tag{B.3}$$

Consider the worst case that, for a server v, $n_{uv}^{(m)}$ is rounded down by 1 $\forall u$ and $\forall m$. In this case, the remaining symbols are assigned to the repository. Therefore, for repository w,

$$\Delta\Gamma_{wv} \leq \frac{|V|}{q} \sum_{m \in M} p^{(m)} \lambda_v \alpha^{(m)} L^{(m)} s. \tag{B.4}$$

As all the parameters other than q in Equations (B.3) and (B.4) are fixed, we can increase q to make $\Delta\Gamma_{uv}$ and $\Delta\Gamma_{wv}$ arbitrarily small. Similarly, by Equation (3.7), $\Delta R_v = |R_v - \tilde{R}_v|$ can also be arbitrarily small by increasing q.

B_v is fixed in the optimization. As functions $\mathbb{C}_{uv}^{\mathrm{N}}$ and $\mathbb{C}_{v}^{\mathrm{S}}$ are continuous and depend on B_v, Γ_{uv} and R_v, $\forall u, v \in V, \exists q$ such that we can also make $\Delta C_{uv}^{\mathrm{N}}$ and $\Delta C_{v}^{\mathrm{S}}$ small enough ($\leq \delta/(|V|^2+|V|)$). Therefore $\Delta C \leq \delta$.