Distributed Cooperative Caching in Social Wireless Networks

Mahmoud Taghizadeh, *Member*, *IEEE*, Kristopher Micinski, *Member*, *IEEE*, Charles Ofria, Eric Torng, and Subir Biswas, *Senior Member*, *IEEE*

Abstract—This paper introduces cooperative caching policies for minimizing electronic content provisioning cost in Social Wireless Networks (*SWNET*). *SWNET*s are formed by mobile devices, such as data enabled phones, electronic book readers etc., sharing common interests in electronic content, and physically gathering together in public places. Electronic object caching in such *SWNETs* are shown to be able to reduce the content provisioning cost which depends heavily on the service and pricing dependences among various stakeholders including content providers (CP), network service providers, and End Consumers (EC). Drawing motivation from Amazon's Kindle electronic book delivery business, this paper develops practical network, service, and pricing models which are then used for creating two object caching strategies for minimizing content provisioning costs in networks with homogenous and heterogeneous object demands. The paper constructs analytical and simulation models for analyzing the proposed caching strategies in the presence of selfish users that deviate from network-wide cost-optimal policies. It also reports results from an Android phone-based prototype *SWNET*, validating the presented analytical and simulation results.

Index Terms—Social wireless networks, cooperative caching, content provisioning, ad hoc networks

1 Introduction

1.1 Motivation

Recent emergence of data enabled mobile devices and wireless-enabled data applications have fostered new content dissemination models in today's mobile ecosystem. A list of such devices includes Apple's iPhone, Google's Android, Amazon's Kindle, and electronic book readers from other vendors. The array of data applications includes electronic book and magazine readers and mobile phone Apps. The level of proliferation of mobile applications is indicated by the example fact that as of October 2010, Apple's App Store offered over 100,000 apps that are downloadable by the smart phone users.

With the conventional download model, a user downloads contents directly from a Content Provider's (CP) server over a Communication Service Provider's (CSP) network. Downloading content through CSP's network involves a cost which must be paid either by end users or by the content provider. In this work, we adopt Amazon Kindle electronic book delivery business model in which the CP (Amazon), pays to Sprint, the CSP, for the cost of network usage due to downloaded e-books by Kindle users.

- *M. Taghizadeh is with the Department of Electrical and Computer Engineering, Michigan State University, 2920 Trappers Cove, APT 2B, Lansing, MI 48824. E-mail: taghizad@msu.edu.*
- *K. Micinski is with the Department of Computer Science, University of Maryland, 9808 53rd Ave., College Park, MD 20740. E-mail: micinski@cs.umd.edu.*
- *C. Ofria and E. Torng are with the Department of Computer Science and Engineering, Michigan State University, 428 S. Shaw Ln., RM 3115, East Lansing, MI 48824-1226. E-mail: {charles.ofria, etorng}@gmail.com.*
- *S. Biswas is with the Department of Electrical and Computer Engineering, Michigan State University, 2120 Engineering Building, East Lansing, MI 48824. E-mail: sbiswas@egr.msu.edu.*

Manuscript received 16 Nov. 2011; accepted 24 Feb. 2012; published online 13 Mar. 2012.
For information on obtaining reprints of this article, please send e-mail to: tmc@computer.org, and reference IEEECS Log Number TMC-2011-11-0616. Digital Object Identifier no. 10.1109/TMC.2012.66.

When users carrying mobile devices physically gather in settings such as University campus, work place, Mall, Airport and other public places, Social Wireless Networks (*SWNETs*) can be formed using ad hoc wireless connections between the devices. With the existence of such *SWNET*s, an alternative approach to content access by a device would be to first search the local *SWNET* for the requested content before downloading it from the CP's server. The expected content provisioning cost of such an approach can be significantly lower since the download cost to the CSP would be avoided when the content is found within the local *SWNET*. This mechanism is termed as *cooperative caching*.

In order to encourage the End-Consumers (EC) to cache previously downloaded content and to share it with other end-consumers, a peer-to-peer rebate mechanism is proposed. This mechanism can serve as an incentive so that the end-consumers are enticed to participate in cooperative content caching in spite of the storage and energy costs. In order for cooperative caching to provide cost benefits, this peer-to-peer rebate must be dimensioned to be smaller than the content download cost paid to the CSP. This rebate should be factored in the content provider's overall cost.

Due to their limited storage, mobile handheld devices are not expected to store all downloaded content for long. This means after downloading and using a purchased electronic content, a device may remove it from the storage. For example in Amazon Kindle clients (iPhone, iPad, etc.) an archive mode is available using which a user simply removes a book after reading it, although it remains archived as a purchased item in Amazon's cloud server. Under the above pricing and data storage model a key question for cooperative caching is: *How to store contents in nodes such that the average content provisioning cost in the network is minimized*?

1.2 Optimal Solution

For contents with varying level of popularity, a greedy approach for each node would be to store as many distinctly

popular contents as its storage allows. This approach amounts to noncooperation and can give rise to heavy network-wide content duplications. In the other extreme case, which is fully cooperative, a node would try to maximize the total number of unique contents stored within the *SWNET* by avoiding duplications. In this paper, we show that none of the above extreme approaches can minimize the content provider's cost. We also show that for a given rebate-to-download-cost ratio, there exists an object placement policy which is somewhere in between those two extremes, and can minimize the content provider's cost by striking a balance between the greediness and full cooperation [26].

This is referred to as *optimal object placement* policy in the rest of this paper. The proposed cooperative caching algorithms strive to attain this *optimal object placement* with the target of minimizing the network-wide content provisioning cost.

1.3 User Selfishness

The potential for earning peer-to-peer rebate may promote selfish behavior in some users. A selfish user is one that deviates from the network-wide optimal policy in order to earn more rebates. Any deviation from the optimal policy is expected to incur higher network-wide provisioning cost. In this work, we analyze the impacts of such selfish behavior on object provisioning cost and the earned rebate within the context of an *SWNET*. It is shown that beyond a threshold selfish node population, the amount of per-node rebate for the selfish users is lower than that for the nonselfish users. In other words, when the selfish node population is beyond a critical point, selfish behavior ceases to produce more benefit from a rebate standpoint.

1.4 Contributions

First, based on a practical service and pricing case, a stochastic model for the content provider's cost computation is developed. Second, a cooperative caching strategy, *Split Cache*, is proposed, numerically analyzed, and theoretically proven to provide optimal object placement for networks with homogenous content demands. Third, a benefit-based strategy, *Distributed Benefit*, is proposed to minimize the provisioning cost in heterogeneous networks consisting of nodes with different content request rates and patterns. Fourth, the impacts of user selfishness on object provisioning cost and earned rebate is analyzed. Finally, numerical results for both strategies are validated using simulation and compared with a series of traditional caching policies.

2 Network, Service, and Pricing Model

2.1 Network Model

Fig. 1 illustrates an example *SWNET* within a University campus. End Consumers carrying mobile devices form *SWNET* partitions, which can be either multi-hop (i.e., MANET) as shown for partitions 1, 3, and 4, or single hop access point based as shown for partition 2. A mobile device can download an object (i.e., content) from the CP's server using the CSP's cellular network, or from its local *SWNET* partition. In the rest of this paper, the terms *object* and *content* are used synonymously.

We consider two types of *SWNETs*. The first one involves stationary [1] *SWNET* partitions. Meaning, after a partition is formed, it is maintained for sufficiently long so that the cooperative object caches can be formed and reach steady states. We also investigate a second type to explore as to what happens when the stationary assumption is relaxed. To investigate this effect, caching is applied to *SWNETs* formed using human interaction traces obtained from a set of real *SWNET* nodes [2].

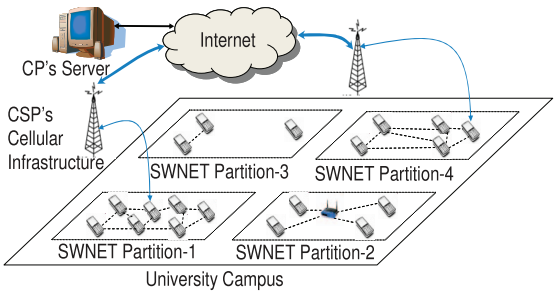

Fig. 1. Content access from an *SWNET* in a University Campus.

2.2 Search Model

After an object request is originated by a mobile device, it first searches its local cache. If the local search fails, it searches the object within its *SWNET* partition using limited broadcast message. If the search in partition also fails, the object is downloaded from the CP's server using the CSP's 3G/4G cellular network. In this paper, we have modeled objects such as electronic books, music, etc., which are time nonvarying, and therefore cache consistency is not a critical issue. We first assume that all objects have the same size and each node is able to store up to "C" different objects in its cache. Later, in Section 5.3, we relax this assumption to support objects with varying size. We also assume that all objects are popularity-tagged by the CP's server [3]. The popularity-tag of an object indicates its global popularity; it also indicates the probability that an arbitrary request in the network is generated for this specific object.

2.3 Pricing Model

We use a pricing model similar to the Amazon Kindle business model in which the CP (e.g., Amazon) pays a download cost C_d to the CSP when an End-Consumer downloads an object from the CP's server through the CSP's cellular network. Also, whenever an EC provides a locally cached object to another EC within its local *SWNET* partition, the provider EC is paid a rebate C_r by the CP. Optionally, this rebate can also be distributed among the provider EC and the ECs of all the intermediate mobile devices that take part in content forwarding. Fig. 2 demonstrates the cost and content flow model. As it is shown in Fig. 2, C_d corresponds to the CP's object delivering cost when it is delivered through the CSP's network, and C_r corresponds to the rebate given out to an EC when the object is found within the *SWNET* (e.g., node A receives rebate C_r after it provides a content to node B over the *SWNET)*. For a given C_r/C_d ratio, the paper aims to develop *optimal object placement* policies that can minimize the network-wide content provisioning cost.

Note that these cost items, namely, C_d and C_r, do not represent the selling price of an object (e.g., e-book). The selling price is directly paid to the CP (e.g., Amazon) by an

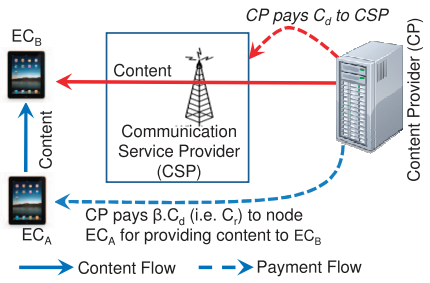

Fig. 2. Content and cost flow model.

EC (e.g., a Kindle user) through an out-of-band secure payment system.

A digitally signed rebate framework needs to be supported so that the rebate recipient ECs can electronically validate and redeem the rebate with the CP. Also, a digital usage right mechanism [4] is needed so that an EC which is caching an object (e.g., an e-book) should not necessarily be able to open/read it unless it has explicitly bought the object from the CP. We assume the presence of these two mechanisms on which the proposed caching mechanism is built.

Operationally, the parameters C_d and C_r are set by a CP and CSP based on their operating cost and revenue models. The end-consumers do not have any control on those parameters.

2.4 Request Generation Model

We study two request generation models, namely, *homogenous* and *heterogeneous*. In the *homogenous* case, all mobile devices maintain the same content request rate and pattern which follow a Zipf distribution. Zipf distribution is widely used in the literature for modeling popularity based online object request distributions [5]. According to Zipf law, the popularity of the ith popular object out of N different objects can be expressed as

$$p_i = \frac{\Omega}{i^{\alpha}}, \Omega = \frac{1}{\sum_{i=1}^{N} \frac{1}{i^{\alpha}}} (0 \leq \alpha \leq 1). \quad (1)$$

The parameter α $(0 < \alpha < 1)$ is a Zipf parameter that determines the skewness in a request pattern. The quantity p_i indicates the probability that an arbitrary request is for the ith popular object $(p_1 > p_2 > \cdots > p_N)$. As α increases, the access pattern becomes more concentrated on the popular data items.

In the *heterogeneous* request model, each mobile device follows an individual Zipf distribution. This means popularity of object j is not necessarily the same from two different nodes standpoints. This is in contrast to the *homogenous* model in which the popularity of object j is same from the perspective of all network nodes. Also, the object request rate from different nodes is not necessarily the same in the *heterogeneous* model.

3 Cost under Homogeneous Request Model

In this section, we compute the average object provisioning cost under a homogenous request model. Let P_L be the probability of finding a requested object in the local cache (i.e., *local* hit rate), P_V be the probability that a requested object can be found in the local *SWNET* partition (i.e., *remote* hit rate) after its local search fails, and P_M be the probability that a requested object is not found in the local cache and in the remote cache (i.e., *miss* rate). We can write P_M in terms of P_V and P_L as

$$P_M = 1 - P_L - P_V. \quad (2)$$

According to the pricing model in Section 2.3, the provisioning cost for an object is zero if it is found in the local cache, C_r when it is found in the *SWNET*, and C_d when it is downloaded from the CP's server through the CSP's network. Thus, the average content provisioning cost

$$Cost = P_V C_r + P_M C_d. \quad (3)$$

Expressing C_r/C_d as β and substituting P_M from (2), cost can be expressed as

$$Cost = (1 - (1 - \beta)P_V - P_L)C_d. \quad (4)$$

Let m be the number of devices within an *SWNET* partition, and S_j be the set of objects stored in device-j $(1 \leq j \leq m)$. With $p_i (1 \leq i \leq N)$ as defined in (1), the probability of finding an object in device-j's cache can be written as $P_L^j = \sum_{i \in S_j} p_i$. The resulting probability of finding the object at any given device in the partition is $\sum_{j=1}^{m} P_L^j/m$ or $\sum_{j=1}^{m} \sum_{i \in S_j} p_i/m$ (recall that the request rate of all nodes is the same). This is the average local hit rate P_L, and can be simplified as

$$P_L = \frac{1}{m} \sum_{i=1}^{N} n_i p_i, \quad (5)$$

where n_i represents the number of copies of object-i within the partition. If C is the available cache size (i.e., the number of objects that can be stored) at each mobile device, then the maximum number of objects that can be stored within an *SWNET* partition is mC. Thus, parameter N in (5) can be replaced by mC.

Let $\mathcal{S}$ represent the set of all stored objects in a partition. The probability of finding an object in the partition can be expressed as $\sum_{i \in \mathcal{S}} p_i$. The quantity $\sum_{i \in \mathcal{S}} p_i$ represents the overall cache hit rate in the partition which is equal to $1 - P_M$. Substituting $\sum_{i \in \mathcal{S}} p_i$ for $1 - P_M$ and the value of P_L from (5) in (2), we can write $P_V = \sum_{i \in \mathcal{S}} p_i - \frac{1}{m} \sum_{i=1}^{mC} n_i p_i$. Using (4), the cost expression can be written as $(1 - (1 - \beta)$ $(\sum_{i \in \mathcal{S}} p_i - \frac{1}{m} \sum_{i=1}^{mC} n_i p_i) - \frac{1}{m} \sum_{i=1}^{mC} n_i p_i) C_d$, and can be simplified as

$$Cost = \left(1 - (1 - \beta) \sum_{i \in \mathcal{S}} p_i - \beta \frac{1}{m} \sum_{i=1}^{mC} n_i p_i\right) C_d. \quad (6)$$

4 Optimal Object Placement

For a given β, the cost in (6) is a function of the vector $\vec{n} = <n_1, n_2, \ldots, n_N>$, where n_i shows the number of copies of object "i" in the *SWNET* partition in question. An object placement $\vec{n}$ is optimal when it leads to minimum object provisioning cost in (6). In this section, we aim to determine the optimal $\vec{n}$.

Lemma 1. *With any popularity-based object request model (e.g., Zipf), the optimal placement approach must ensure the following constraint at steady state.*

An object should not be stored in a partition when at least one object of higher popularity is missing in that partition. That is, object i (i.e., ith popular object) cannot be cached while a higher popularity object k ($k < i$) is missing. This is referred to as *popularity storage constraint*.

Proof. Let us assume that there is an optimal placement which minimizes the object provisioning cost in (6) and violates the *popularity storage constraint*. It means there is a missing object "i" in the *SWNET* (i.e., $n_i = 0$) while a less popular object "j" is present (i.e., $j > i, n_j > 0$). □

Using (6), it can be shown that if a less popular object "j" is replaced with the missing object "i," the cost will be lower. This contradicts the assumption and therefore, the optimal object placement must preserve the *popularity storage constraint*.

Now let us assume that "T" is the least popular object in the optimal solution. According to the popularity storage constraint, there is at least one copy of objects "1" to "T" in the partition. Therefore, (6) can be written as

$$Cost = \left(1 - (1-\beta)\sum_{i=1}^{T} p_i - \beta\frac{1}{m}\sum_{i=1}^{T} n_i p_i\right) C_d. \qquad (7)$$

Lemma 2. *In the optimal object placement, an object k (i.e., kth popular object) should not be duplicated unless all other objects with higher popularity have been duplicated in all nodes.*

Proof. According to the *storage popularity constraint* in the optimal solution, at least one copy of object "1" to object "T" exists. Since object "T" is the least popular object in the optimal solution, (7) can be rewritten as □

$$Cost = 1 - \sum_{i=1}^{T}\left(1 - \beta - \frac{\beta}{m}\right)p_i - \frac{\beta}{m}\sum_{i=1}^{T}(n_i - 1)p_i.$$

Now, let $n_\ell \neq n_k, 1 < n_\ell, n_k < m$, and $\ell < k$. It can be observed that by increasing n_ℓ and by reducing n_k it is possible to lower the cost. This can lead to the following claim: while there is room for increasing the number of copies of object ℓ (i.e., $n_\ell < m$), less popular objects (e.g., object k, $k > l$ should not be duplicated. Following the above logic, we cannot duplicate object "2" unless we have duplicated object "1" in all nodes (i.e., $n_1 = m$). Similarly, we cannot duplicate objet "i" unless we have already duplicated more popular objects in all nodes.

Claim. The optimal object placement $\vec{n}$ has the following properties:

1. $n_i = m \; for \; 1 \leq i \leq \ell$, where ℓ is the least popular duplicated object in the network, and its value should be determined based on β. One copy of objects $1 \cdots \ell$ will be stored in all nodes.
2. $n_i = 1 \; for \; \ell + 1 \leq i \leq T$, where $T = NC - N\ell + l + 1$. This means the remaining space of caches is filled with unique objects.
3. $n_i = 0 \; for \; i > T$.

Proof. According to *Lemma 1*, There must be at least one copy of objects $1 \cdots T$ in the network (i.e., there is no missing object). *Lemma 2* states that an object should not duplicated before all other objects with higher popularity have been duplicated in all nodes. This means if ℓ is the least duplicated popular object in the network, there should be m number of copies of objects $1 \ldots \ell$ in the network. □

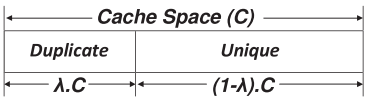

Fig. 3. Cache partitioning in split cache policy.

Note that the above analysis does not help deciding the value of ℓ, or the set of objects that need to be duplicated for the optimal object placement solution. It only shows that if the optimal solution requires duplication, it must be across all nodes. In the next section, we show how to determine the value of ℓ.

5 Caching for Optimal Object Placement

5.1 Split Cache Replacement

To realize the optimal object placement under homogeneous object request model we propose the following *Split Cache* policy in which the available cache space in each device is divided into a duplicate segment (λ fraction) and a unique segment (see Fig. 3). In the first segment, nodes can store the most popular objects without worrying about the object duplication and in the second segment only unique objects are allowed to be stored. The parameter λ in Fig. 3 ($0 \leq \lambda \leq 1$) indicates the fraction of cache that is used for storing duplicated objects.

With the *Split Cache* replacement policy, soon after an object is downloaded from the CP's server, it is categorized as a *unique* object as there is only one copy of this object in the network. Also, when a node downloads an object from another *SWNET* node, that object is categorized as a *duplicated* object as there are now at least two copies of that object in the network.

For storing a new *unique* object, the least popular object in the whole cache is selected as a candidate and it is replaced with the new object if it is less popular than the new incoming object. For a *duplicated* object, however, the *evictee* candidate is selected only from the first duplicate segment of the cache. In other words, a unique object is never evicted in order to accommodate a duplicated object. The *Split Cache* object replacement mechanism realizes the optimal strategy established in Section 4. With this mechanism, at steady state all devices' caches maintain the same object set in their duplicate areas, but distinct objects in their unique areas. The pseudocode of *Split Cache* replacement policy is shown in Algorithm 1.

INPUT: Object O_{new}
IF (O_{new} *is downloaded from another node*)
  O_{min} = *the least popular obj in the* ***duplicate*** *area*
ELSE
  O_{min} = *The least popualar obj in the* ***entire*** *cache*
END
IF (O_{nel} . $popularity > O_{min}.popularity$)
  replace O_{min} *l ith* O_{nel}

Algorithm 1: Split Cache object replacement policy

5.2 Object Provisioning Cost with Split Cache

To compute the provisioning cost for *Split Cache* we need to compute P_L and P_V used in (4). We first define function $f(k)$ to be the probability of finding an arbitrary object within a device's cache that is filled with the k most popular objects. This function can be expressed as $\sum_{i=1}^{k} p_i$. Substituting p_i for the Zipf distribution (see Section 2), we can write

$$f(k) = \sum_{i=1}^{k} p_i \approx \int_1^k \frac{\Omega}{i^\alpha} di = \Omega \frac{k^{(1-\alpha)} - 1}{1-\alpha}.$$

Similarly, $\Omega = 1/\sum_{i=1}^{N} \frac{1}{i^\alpha} \approx 1/\int_1^k \frac{1}{i^\alpha} di = \frac{1-\alpha}{N^{(1-\alpha)}-1}$. Therefore, $f(k)$ can be simplified as

$$f(k) = \frac{k^{(1-\alpha)} - 1}{N^{(1-\alpha)} - 1}. \tag{8}$$

Local hit rate P_L. At steady state, total number of unique objects stored in the partition is equal to $mC(1-\lambda)$, where m is the number of mobile devices. Also, number of duplicated objects is equal to λC. Therefore, the total number of different objects stored in the partition is $\lambda C + mC(1-\lambda)$. Probability that a device can find a new requested object in its local cache is equal to

$$P_L = H_D + \frac{H_U}{m}, \tag{9}$$

where $H_D = f(\lambda C)$ corresponds to the cache hits contributed by the objects stored in the duplicate area of cache and $H_U = f[(\lambda + m(1-\lambda))C] - f(\lambda C)$ represents the hit rate contributed by all unique objects (in the partition) which are assumed to be uniformly distributed over all m devices' caches.

Remote hit rate P_V. It is equal to the hit probability contributed by the objects stored in the unique area of all devices in the partition, minus the unique area of the local cache. This can be expressed as

$$P_V = \frac{m-1}{m} H_U. \tag{10}$$

Substituting P_L and P_V from (9) and (10) in (4), the expression of cost can be simplified as

$$Cost = \left(1 - \left(\frac{(1-\beta)m + \beta}{m}\right) H_U - H_D\right) C_d. \tag{11}$$

Using (8) to expand H_U and H_D, (11) can be written as a function of λ. By equating the derivative of the cost expression to zero, we can compute the λ_{opt} at which cost is minimized.

5.3 Handling Objects with Different Size

So far we assumed that all objects have the same size. In this section, the minimum-cost object replacement mechanism is extended for scenarios in which objects can have different sizes. In such situations, in order to insert a new downloaded object "i" from the CP's server, instead of finding the least popular object, a node needs to identify a set of objects ψ in the cache. The set ψ should be identified such that the quantity $\sum_{j\in\Psi} p_j$ is minimized while $\sum_{j\in\Psi} p_j < p_i$ and $\sum_{j\in\Psi} x_j > x_i$; the quantity x_i shows the size of object "i." This is a traditional knapsack problem for which a number of heuristics-based solutions are available in the literature. If a set ψ, satisfying the above conditions, is found, then all objects in that set are evicted from the cache to accommodate the new incoming object; otherwise the incoming object "i" is not admitted. When an object is downloaded from another node in the *SWNET*, the members of ψ can be selected only from the objects stored in the duplicate area of the cache. Note that dimensioning of the split factor λ with varying object size is not addressed in this paper.

6 Caching under Heterogeneous Requests

The *Split Cache* policy in Section 5 may not able to minimize the provisioning cost for nonhomogenous object requests where nodes have different request rates and request patterns. In this section, we propose and analyze a *benefit-based heuristics* approach to minimize the object provisioning cost in a network with nonhomogenous request model.

The probability that a node "i" finds the requested object in its own cache is $\sum_{j\in s_i} p_i^j$, where s_i indicates the set of stored object in node "i" and p_i^j shows the probability that a generated request in node "i" is for object "j." The probability that a request is found in the network after its local search fails is equal to $\sum_{j\in(\mathcal{S}-s_i)} p_i^j$, where δ represents the set of all objects stored in the network. Finally, the probability that an object is not available in the network and needs to be downloaded from the CP's server is $1 - \sum_{j\in\mathcal{S}} p_i^j$. Therefore, the average provision cost for node "i" can be expressed as

$$Cost_i = \left(\beta \sum_{j\in(\mathcal{S}-S_i)} p_i^j + \left(1 - \sum_{j\in\mathcal{S}} p_i^j\right)\right) C_d. \tag{12}$$

Average provision cost across all nodes can be calculated as

$$\begin{aligned} Cost &= \frac{\sum_i \mu_i cost_i}{\sum_i \mu_i} \\ &= \left(1 - \frac{\sum_i \mu_i \sum_{j\in\mathcal{S}} p_i^j}{\sum_i \mu_i} + \beta \frac{\sum_i \mu_i \sum_{j\in\mathcal{S}-s_i} p_i^j}{\sum_i \mu_i}\right) C_d, \end{aligned} \tag{13}$$

where μ_i shows the request generation rate in node "i."

6.1 Benefits of Caching

Suppose Q is the set of nodes that store a copy of object "j" in their cache. Let μ_i be the object request rate for node "i" and p_i^j be the probability that a generated request in node "i" is for object "j" (i.e., node "i" generates $\mu_i p_i^j$ requests for object "j" per unit time). The cost of network usage for downloading an object directly from CP's server is C_d. Therefore, storing object "j" reduces cost at node "i" by the amount $\mu_i p_i^j C_d$ per unit time. This reflects the *benefit* of storing object "j" in node "i." Thus, the *benefit* of storing object "j" in the set of nodes specified by Q can be written as $\sum_{\forall i\in Q} \mu_i p_i^j C_d$.

Additionally, every other node in an *SWNET* partition (i.e., nodes that do not store object "j" locally) is able to download object "j" from one of nodes in Q with cost βC_d. This reduces the cost of providing object "j" to any other node in the network by the amount $(1-\beta)C_d$ for each request for object "j." Total number of requests for object

"j" by the other nodes in the *SWNET* is equal to $\sum_{\forall k \notin \mathcal{Q}} \mu_k p_k^j$. Therefore, the *remote benefit* of storing a unique object "j" in the network is equal to $(1-\beta)C_d \sum_{\forall k \notin \mathcal{Q}} \mu_k p_k^j$. The *total benefit* (the overall amount of cost reduction) of storing a *object* "j" in set of nodes specified by "Q" can be written as

$$\sum_{\forall i \in \mathcal{Q}} \mu_i p_i^j C_d + (1-\beta) C_d \sum_{\forall k \notin \mathcal{Q}} \mu_k p_k^j.$$

This can be rewritten as

$$(1-\beta) C_d \sum_{\forall k} \mu_k p_k^j + \sum_{\forall i \in \mathcal{Q}} \beta \mu_i p_i^j C_d. \quad (14)$$

The first term of (14) refers to the *global benefit* of storing object "j" in the network. Note that *global benefit* of storing an object in the network does not depend on the location and the number of copies of that object. The global benefit of objects $(1 \ldots N)$ be represented by a vector $\overrightarrow{U}$ where

$$U_j = (1-\beta) C_d \sum_{\forall k} \mu_k p_k^j. \quad (15)$$

The second term of (14) shows the *local benefits* of storing object "j" in set of nodes specified by $\mathcal{Q}$. The *local benefit* of storing object $(1 \ldots N)$ in nodes $(1 \ldots m)$ can be represented by a matrix $D_{m \times N}$ where

$$D_{ij} = \beta \mu_i p_i^j C_d. \quad (16)$$

Using the above notations, the *total benefit* of storing object "j" in a set of nodes specified by "Q" can be written as

$$U_j + \sum_{k \in \mathcal{Q}} D_{kj}.$$

6.2 Benefit-Based Distributed Caching Heuristics

With the *Distributed Benefit*-based caching strategy presented in this section, when there is not enough space in the cache for accommodating a new object, the existing object with the minimum benefit is identified and replaced with the new object only if the new object shows more total benefit. The benefit of a newly downloaded object is calculated based on its source. When a new object "j" is downloaded by node i directly from the CP's server using the CSP's 3G/4G connection (i.e., no other copy of the object is present in the *SWNET* partition), the copy is labeled as *primary* and its benefit is equal to $U_j + D_{ij}$.

When the object is downloaded from another node in the *SWNET* partition (i.e., at least one more copy of the object already exists in the partition), the copy is labeled as *secondary* and its benefit is equal to D_{ij}. The new object is cached if its benefit is higher than that of any existing cached object.

In addition to the benefit-based object replacement logic as presented above, provisioning cost minimization requires that a primary object within an *SWNET* partition should be cached in the node that is most likely to generate requests for that object. In other words, a primary object j in the partition must be stored in node i such that $\mu_i p_i^j > \mu_k p_k^j$ *for all* $k \neq i$.

To satisfy the above constraint, the *primary* copy of an object "j" must always be stored in a node with the highest request generation rate for that object. To enforce this, in addition to the object-ID, a node sends its estimated request generation rate for the requested-object during the search process within *SWNET*. Upon receiving the search request, an object holder compares its own request rate for the object with that of the requesting node. If the request rate of the requesting node is higher and the object copy is a *primary* copy, then the object provider sends the object along with a *change_status* flag to the requesting node. This flag informs the requesting node that the object must be considered as a *primary* copy. Upon receiving of the object and the *change_status* flag, the requesting node considers the object as a *primary* copy and if it can find an object with lower benefit or if it has an empty slot, it stores the new object in its cache. After storing it, the requesting node sends another *change_status* message to the provider node which causes the provider node labels its object as a *secondary* copy. The complete logic of the *Distributed Benefit* heuristics is summarized in Algorithm 2.

INPUT: O_j //The new downloaded object
flag // Change status flag
IF (*O_j is downloaded from Internet || flag == True*)
$O_j.benefit = U_j + D_{ij}$
$O_j.label = Primary$
ELSE
$O_j.benefit = D_{ij}$
$O_j.label = l\,econdary$
END
$O_{min} = Object\ l\ ith\ minimum\ benefit$
IF ($O_j.benefit > O_{min}.benefit$)
replace $O_{min}\ l\ ith\ O_j$
send *change status message to the provider node*
END

Algorithm-2: A distributed heuristic for object placement in *SWNETs* with heterogeneous content requests in node 'i'

Note that in certain rare situations the *object status modification* process fails to satisfy the above constraint. For example, consider a situation in which only one node in the network generates requests and other nodes make no requests. In this case, due to storage limitations, the active node can only store a limited number of objects. The *object status modification* process does not help the active node to offload some objects to the other nodes in the network. Offloading objects to other caches needs extra protocol syntax and requires additional complexity and overhead in the algorithm and it's beyond the scope of our current work. Object status modification *process* also fails to work perfectly in highly mobile situations. For example, two nodes may consider an object as *primary copy* while they are in the same *SWNET* partition. This may result in storing additional number of copies of some objects. Due to these inconsistencies *Distributed Benefit* heuristics does not guarantee a cost-optimal object placement.

6.3 Performance Upper Bound: Optimal Object Placement

In this section, we introduce a centralized mechanism in order to find the optimal object placement. First, we map the object placement task to a maximum weight matching problem in a bipartite graph. Then, we formulate an integer linear objective function to find the maximum

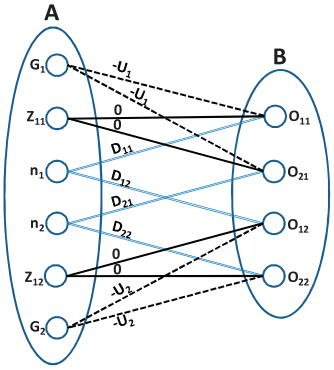

Fig. 4. An object placement problem as bipartite graph.

weight matching, and we show that the linear programming relaxation of this problem in fact provides the optimal solution.

In a maximum weight bipartite matching problem, for a given bipartite graph $G = (V, E)$ with bipartition $(\mathcal{A}, \mathcal{B})$ and weight function $w : E \rightarrow \mathbb{R}$, the objective is to find a matching of maximum weight where the weight of matching M is given by $w(M) = \sum_{e \in M} w(e)$. Without loss of generality, it can be assumed that G is a complete weighted bipartite graph (zero weight edges can be added as necessary); it can be also assumed that G is balanced, i.e., $|\mathcal{A}| = |\mathcal{B}| = \frac{1}{2V}$, as we can add dummy vertices as necessary.

6.3.1 Optimal Object Placement as a Matching Problem

To map the object placement problem to a *maximum weight bipartite matching*, nodes are modeled by vertices $n_1 \dots n_m$ in partition $\mathcal{A}$, and objects are modeled as vertices in partition B. Initially, we assume that each node is able to store only one object (i.e., cache size is equal to 1) and later we relax this assumption.

In object placement, we may put one object in multiple nodes therefore every object must be modeled by m vertices. For example, for object "j" we create vertices $O_{1j} \dots O_{mj}$ in partition B A vertex O_{ij} then is connected to the vertex n_i with the weight of D_{ij} which shows the local benefit of storing object "j" in node "i." We also add vertices $Z_{1j} \dots Z_{m-1j}$ in partition A and connect vertices $O_{1j} \dots O_{mj}$ to them using the edges with weight zero. These new vertices are added to model the situation when object "j" is not stored in that node. When there is no copy of object "j" in the network the *global benefit* of object "j" is lost. To model this situation, vertex G_j is added in partition $\mathcal{A}$ and it is connected with vertices $O_{1j} \dots O_{mj}$ using the edges with weight $-U_j$. Note that there is only $m - 1$ edges with weight zero and therefore, in perfect matching at least one edge with weight of $-U_j$ must be selected when object "j" is not stored in any node. The above process is repeated for all objects in the network. Also for every slot of cache space a vertex must be created in partition A and the whole process of mapping must be repeated again. Fig. 4 shows a modeled object placement problem when $m = 2$, $N = 2$, and $C = 1$.

To make sure all weights are positive, a large enough constant Δ is added to all weights. By adding dummy vertices and edges with weight 0, the graph becomes a complete bipartite graph.

6.3.2 Maximum Weight Matching

For the resulting complete bipartite graph, we can formulate *maximum weight perfect matching* as an Integer Linear Programming (ILP) problem as follows:

$$Max \sum_{\forall (i,j)} w_{ij} x_{ij}.$$

Subject to

1. $for\ i \in \mathcal{A} : \sum_j x_{ij} = 1.$
2. $for\ j \in \mathcal{B} : \sum_i x_{ij} = 1.$
3. $x_{ij} \in \{0, 1\}\ i \in \mathcal{A}, j \in \mathcal{B},$

where $x_{ij} = 1$ if $(i, j) \in matching\ M$ and 0 otherwise. We can relax the integrality constraints by replacing constraint 3 with

$$x_{ij} \geq 0\ i \in \mathcal{A}, j \in \mathcal{B}.$$

This gives linear programming relaxation of the above integer program. In a linear program, the variable can take fractional values and therefore there are many feasible solutions to the set of constraints above which do not corresponds to matching. This set of feasible solution forms a *polytope*, and when we optimize a linear constraint over a polytope, the optimum will be attained [27] at one of the "corners" or *extreme points of the polytope.*

In general, the extreme points of a linear program are not guaranteed to have all coordinates integral. In other words, in general there is no guarantee that the solution for linear programming relaxation and the original integer program are the same. However, for matching problem we notice that the constraint matrix of linear program is *totally unimodular* and therefore any *extreme point* of the *polytope* defined by the constraints in linear program is integral [27]. Moreover, if an optimum solution to a linear programming relaxation is integral, then it must also be an optimum solution to the integer program [18]. Therefore, the solution found by linear programming is optimal for the maximum weight bipartite matching problem to which our object placement problem is mapped into.

The *maximum weight matching M* represents the optimal object placement which minimizes provisioning cost in (13). The optimum result of the linear program can be treated as the upper bound of cooperative caching performance. Such upper bounds are reported in the experimental results in Section 10.

The *maximum weight perfect matching* can be also found by Hungarian method (also known as Kuhn-Munkres algorithm) in polynomial time [18], [19]. In the literature, there are many other algorithms for finding the maximum weight perfect matching.

7 User Selfishness and its Impacts

In Section 5, we computed the cost and rebate in a cooperative *SWNET* with homogeneous requests where all nodes run the split replacement policy with optimal λ. The impacts of user selfishness on object provisioning cost are analyzed in this section. Note that the following study is limited only for homogenous content requests and it

assumes that there is no collusion among nodes that behave in a selfish manner.

Selfishness. A node is defined to be selfish when it deviates from optimal caching in order to earn more rebates. A rational selfish node stores an object only if that object increases the amount of its own potential rebate. The set of objects in such a node is expected to be different from that of a nonselfish node. To analyze the impacts of user selfishness we first need to know which policy maximizes the rebate for a selfish node.

Let $\mathcal{S}$ be the set of objects that can maximize the earned rebate for a selfish node. Each member of δ should be in one of the following situations:

1. There is at least one other copy of that object in nonselfish nodes in the *SWNET* partition (i.e., duplicated).
2. There is no other copy of that object in the *SWNET* partition (i.e., unique).

A selfish node will not store a duplicated object "i" before storing objects with higher popularity. This is simply because by replacing object "j" by object "k," the node is able to increase its rebate. Likewise, a selfish node will not store a unique object "j" unless it has already stored other unique objects with higher popularity. In other word, a selfish node follows the *popularity storage constraint* (see Section 4). From the discussion above it emerges that a selfish node duplicates part of popular objects and then fills the rest of its cache with unique object. This trend is similar to the *Split Cache* policy described in Section 5.

In the absence of collusion, each selfish node assumes that there is no other such node in the network. Therefore, all selfish nodes choose the same policy, namely, *Split Cache*, but with a split factor λ that is different from the optimal λ as established for nonselfish operation in Section 5. If a selfish node decides to store a duplicated object, the other selfish nodes will do the same thing since they are not colluding and are forced to run the exact same policy when they use the same exact information. Therefore, there is no partially duplicated object in the network.

Degree of selfishness in an *SWNET* is modeled by a parameters η, which is the number of selfish nodes, and λ_S which is the non-optimal split-factor chosen by those nodes.

7.1 Cost and Rebate for Nonselfish Nodes

Average provisioning cost in an m-node network with both non-selfish and selfish nodes can be written as

$$Cost = \frac{\eta Cost_{selfish} + (m-\eta)Cost_{nonselfish}}{m}. \tag{17}$$

In order to compute $Cost_{nonselfish}$ using (4), we need to compute the quantities P_L and P_V for the nonselfish nodes. The steady state cache status for both nonselfish and selfish nodes for the case $\lambda_S < \lambda_o$ are pictorially demonstrated in Fig. 5.

Local hit rate. The nonselfish nodes store $\lambda_0 C$ most popular objects in the duplicate segment of their cache (areas A_1 and A_2 in Fig. 5), and fill the rest with unique objects (area A_3 in Fig. 5). Assuming unique objects are uniformly distributed in all nodes, the quantity P_L for a nonselfish node can now be computed as

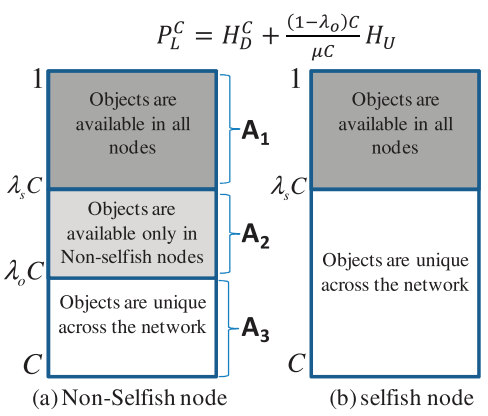

Fig. 5. Cache status at steady state.

$$P_L^C = H_D^C + \frac{(1-\lambda_o)c}{\mu c} H_U, \tag{18}$$

where $\mu C = ((m-\eta)(1-\lambda_o) + \eta(1-\lambda_s))C$ and it represents the total number of unique objects stored in an *SWNET* partition. The quantity $H_U = f(\mu C + \lambda_o C) - f(\lambda_o C)$ represents the corresponding hit rate for all unique objects in the partition. The first term $H_D^C = f(\lambda_o C)$ refers to the hit rate contributed by the duplicated objects and the second term refers to the hit rate contributed by the unique objects stored in a nonselfish node.

Remote hit rate. After a local search fails, the requested object can only be found in the unique area of the remote caches in the partition. Therefore, the probability of finding the desired object in the partition (after its local search failed) is equal to the corresponding hit rate contributed by all the unique objects stored in the partition except those unique objects stored in the requesting device. Thus, the remote hit rate is equal to

$$P_V^C = \left(1 - \frac{(1-\lambda_0)c}{\mu c}\right) H_U. \tag{19}$$

Substituting the value of P_L and P_V from (18) and (19) in (4), the cost for provisioning objects to the nonselfish nodes can be simplified as

$$Cost_{nonselfish} = \left(1 - \frac{((1-\beta)\mu + \beta(1-\lambda_0))C}{\mu C} H_U - H_D^C\right) C_D. \tag{20}$$

Note that H_U in the above equation must be computed as

$$\begin{cases} H_U = f(\mu C + \lambda_o C) - f(\lambda_o C) \ when \ \lambda_s < \lambda_o, \\ H_U = f(\mu C + \lambda_s C) - f(\lambda_s C) \ when \ \lambda_s \geq \lambda_o. \end{cases} \tag{21}$$

The amount of rebate earned by a node depends on the number of requests generated in the network for objects stored in its local cache. In addition to the globally available objects (stored in area A_1 in Fig. 5) all nonselfish nodes maintain certain duplicated objects in their cache which are not available in the selfish nodes (objects in area A_2 in Fig. 5). Thus, in addition to the unique objects (area A_3 in Fig. 5), each nonselfish node provides certain duplicated objects (stored in area A_2) to the selfish nodes. Therefore, amount of rebate per requested object in the *SWNET* partition for nonselfish nodes can be written as

$$Rebate_{nonselfish} = \left(\frac{\frac{1-\lambda_O}{\mu} H_U(m-1) + \frac{f(\lambda_o C) - f(\lambda_s C)}{m-\eta} \eta}{} \right) \beta C_d. \quad (22)$$

The first term in (22) indicates the corresponding rebate for providing unique objects (from area A_3) to all other nodes in the network, and the second term indicates the rebate for providing certain duplicated objects (from area A_2) to the selfish nodes. The hit rate contributed by the duplicated objects that are not available in the selfish nodes is equal to $f(\lambda_o C) - f(\lambda_s C)$. It is assumed that the generated requests from the selfish nodes are serviced by all nonselfish nodes in a uniform manner (that is why the quantity $f(\lambda_o C) - f(\lambda_s C)$ is divided by $\mathrm{m} - \eta$). Note when $\lambda_s > \lambda_o$, the selfish nodes maintain more duplicated objects, and therefore the second term in (22) vanishes. The rebate in this case can be written as

$$Rebate_{nonselfish} = \left(\frac{1-\lambda_O}{\mu} H_U(m-1) \right) \beta C_d. \quad (23)$$

7.2 Cost and Rebate for Selfish Nodes

Similar to the nonselfish nodes, cost and rebate for the selfish nodes are computed using (4). P_L for a selfish node can be computed as

$$P_L^N = H_D^N + \frac{(1-\lambda_s)c}{\mu c} H_U, \quad (24)$$

where $H_D^N = f(\lambda_S C)$ refers to the hit rate contributed by the duplicated objects stored in a selfish node, H_U is the hit rate contributed by all unique objects in the partition, Remote hit rate for a selfish node can be computed as

$$P_V^N = \left(1 - \frac{(1-\lambda_s)C}{\mu C} \right) H_U. \quad (25)$$

Substituting the value of P_L and P_V from (24) and (25) in (4), the cost for selfish nodes can be simplified as

$$Cost_{selfish} = \left(1 - \frac{((1-\beta)\mu + \beta(1-\lambda_s))c}{\mu c} H_U - H_D^N \right) C_d. \quad (26)$$

Assuming that unique objects are uniformly distributed among all *SWNET* nodes, the rebate for a selfish node when $\lambda_s < \lambda_o$ is

$$Rebate_{selfish} = \left(\frac{1-\lambda_s}{\mu} H_U(m-1) \right) \beta C_d. \quad (27)$$

When $\lambda_s \geq \lambda_o$, the rebate for selfish nodes changes to

TABLE 1
Baseline Simulation Parameters

Number of ECs in a partition(V)	40
Download cost (C_d)	10
Rebate-to-download-cost ratio (β)	$0 \leq \beta \leq 1$
Cache size in each mobile device (C)	50
Zipf parameter (α)	0.8
Object population (N)	5000
Warm up phase to reach steady state	2000 requests
Total simulation duration	10000 requests

$$Rebate_{selfish} = \left(\frac{\frac{1-\lambda_s}{\mu} H_U(m-1) + \frac{f(\lambda_s C) - f(\lambda_o C)}{\eta} (m-\eta)}{} \right) \beta C_d. \quad (28)$$

In the above equation, the first term indicates the amount of rebate a selfish node earns by providing its unique objects to all other nodes, and the second term represents the rebate earned by providing its duplicated objects to the non-selfish nodes. The quantity $f(\lambda_s C) - f(\lambda_o C)$ corresponds to the hit rate contributed by the objects that are not available in nonselfish nodes and duplicated across all selfish nodes. These objects are provided only to the $m - \eta$ nonselfish nodes.

8 Performance with Homogeneous Requests

The performance of *Split Cache* was evaluated using the analytical expressions in Section 5, and then via ns2 network simulation. For simulation, a flooding-based object search mechanism has been implemented using the baseline AODV [6] route discovery syntaxes. Baseline experimental parameters are summarized in Table 1.

8.1 Hit Rates and Provisioning Cost

Fig. 6a depicts the impacts of λ on the hit rates. The $\lambda = 0$ case represents zero duplication, leading to the maximum number of unique objects in the partition. The $\lambda = 1$ case causes maximum duplication. In this case, all nodes cache the same set of C (cache size) most popular objects. Smaller λ values lead to very few copies of the popular objects within the local cache and the subsequent low local hit rates. Since with larger λ, more and more popular objects are duplicated, the likelihood of finding objects locally improves, leading to higher P_L values.

The miss rate P_M depends on the total number of unique objects in the *SWNET* partition, which increases with higher

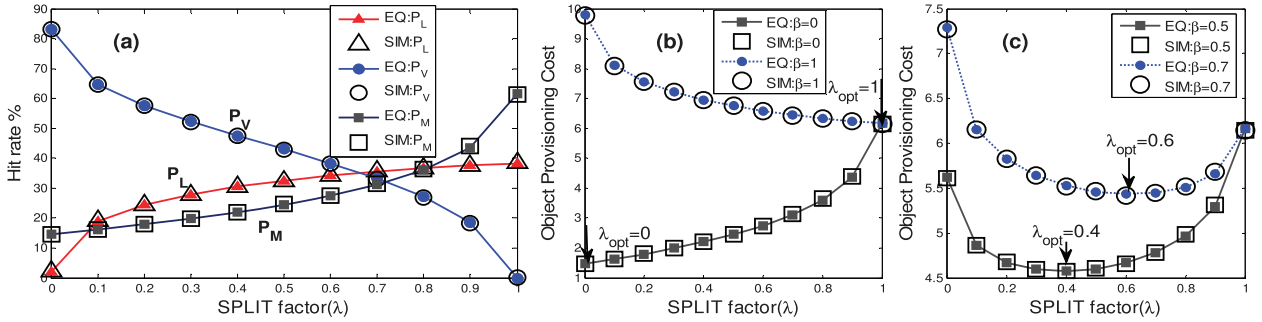

Fig. 6. (a) $\mathrm{P_L}$, $\mathrm{P_V}$, and $\mathrm{P_M}$ as a function of the split factor λ. (b) and (c) Provisioning cost as a function of the split factor λ.

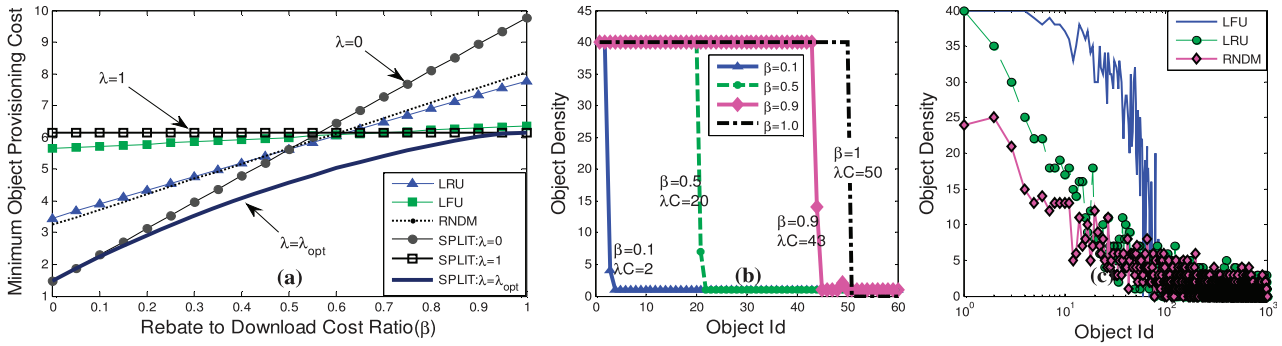

Fig. 7. (a) Comparative minimum cost. (b) Partition object density for *Split Cache* policy. (c) Density for traditional policies.

duplications when λ is increased. Smaller λ values lead to less duplication and as a result $\mathrm{P_M}$ reduces. The excellent agreement between the analytical and the simulation results in Fig. 6a indicates the correctness of the equations in Section 5.

Figs. 6b and 6c depict the provisioning cost as a function of λ. When $\beta = 0$ (i.e., $C_r = 0$), the cost expression in (4) reduces to $Cost = P_M C_d$. Meaning, for a given C_d, cost depends only on the miss rate P_M which reduces as λ reduces. Therefore, when $\beta = 0$, $\lambda = 0$ gives the minimum P_M and consequently, the minimum cost. When $\beta = 1$ (i.e., the rebate is same as the download cost C_d) the expression in (4) reduces to $Cost = (1 - P_L)C_d$, indicating that it depends only on the local hit rate for a given C_d. This explains why the cost decreases with increasing λ, whereas the local hit rate increases. Intuitively, when $C_r = C_d$, there is no advantage of fetching objects from the *SWNET*. The only way to reduce cost in this situation is to maximize P_L.

Observe in Fig. 6c that for both $\beta = 0.5$ and $\beta = 0.7$, the cost reduces initially for increasing λ but after a critical $\lambda = \lambda_{opt}$, the cost starts to increase. This critical point can be found numerically from (11). The reason for this λ_{opt} was explained in Section 4. It was established that starting from the state of zero partition-wide duplication, if the iterative duplication/replacement process stops at the correct point, the cost can be minimized. This translates to finding the appropriate level of duplication, which is decided by λ. As shown in Fig. 6c, λ_{opt} is 0.4 when β is 0.5, and it is 0.6 when β is 0.7. Thus, a larger λ_{opt} is needed when the rebate is larger with respect to the download cost from the CP's server.

8.2 Comparison with Traditional Caching

Fig. 7a shows the cost for Least Recently Used (LRU) [7], Least Frequently Used (LFU)[7], and Random (RNDM) [7] along with those for *Split Cache* with λ set to 0, 1, and λ_{opt}. While LRU and LFU implicitly leverage object popularity by storing the most popular objects, RNDM policy is completely insensitive to object popularity. As expected, Fig. 7a depicts that *Split Cache* with λ_{opt} provides the best cost. $\lambda = 0$ delivers near-best performance for small β values. This is because as shown in (4), for small β (i.e., small rebate C_r), the cost depends mainly on P_M. From Fig. 6a, the miss rate is minimum for $\lambda = 0$, which corresponds to no duplication exclusive caching.

When β is large (e.g., $\beta > 0.7$), $\lambda = 1$ delivers near-best performance. This is because as shown in (4), for large β, the cost depends mainly on P_L, which is maximized when $\lambda = 1$. All traditional policies perform in between *Split Cache* with $\lambda = 0$ and $\lambda = 1$. Since RNDM is insensitive to popularity, by uniformly distributing the objects in the partition, it is able to increase P_V, which helps it outperform LRU and LFU for small βs. LFU, on the other hand, attempts to distinguish popular objects by keeping track of the number of hits for an object. This explains its performance proximity with *Split Cache* when $\lambda = 1$.

8.3 Partition Object Density

As discussed in Section 4, in order to minimize cost, certain objects should be duplicated in all devices and the remaining space in the *SWNET* partition should be filled with unique objects. Therefore, the possible density values are m (number of ECs) for the duplicated objects, 1 for the unique objects, and 0 for the objects that are not stored in any node. This is confirmed in Fig. 7b which reports object density from simulation for different values of βs. With increasing β, since λ_{opt} increases, more objects are duplicated, thus increasing the object density. We show results only up to object-id 50 (i.e., cache size C), because objects beyond C have only one or zero copy. When $\beta = 0$, since λ_{opt} is also zero, there is no duplication, causing the mC most popular objects to have one copy and the rest of the objects with zero copy.

Fig. 7c depicts simulated object densities for LFU, LRU, and RNDM. Certain amount of density skew (i.e., higher density for more popular objects) is generated by the Zipf-based object requests, which favor more popular objects. Observe that for RNDM, the object densities are minimally skewed, since the policy itself is not at all sensitive to object popularities. LFU, on the other hand, shows a density pattern that is closest to the *Split Cache* when $\lambda = 1$ due to its effective sensitivity to object popularity. Similar to the cost results, the density pattern for LRU lies somewhere in between RNDM and LFU. This is because the effective sensitivity to object popularity for LRU is weaker than LFU, but stronger than RNDM.

8.4 Cost Dynamics over Time

The graph in Figs. 8a and 8b demonstrates how the provisioning cost and different hit rates in an *SWNET* partition (with 40 nodes) changes during the network warm-up phase and also when the object popularities change as a response to external news and events. As shown in Fig. 8a, initially, when all caches are empty, nodes have to download their desired objects directly from the CP's server and therefore the average provisioning cost is

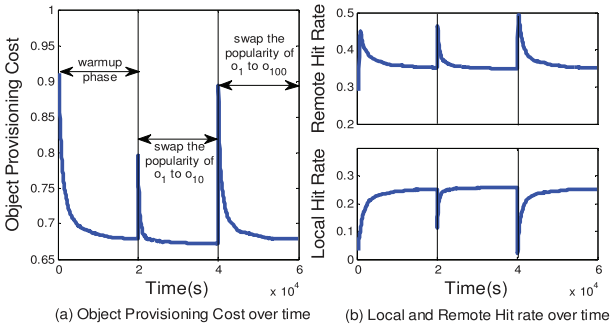

Fig. 8. Cost and hit rates over time.

very high. This can also be confirmed by Fig. 8b which shows the low local hit rate and the high remote hit rate during the warm-up phase. Gradually, when nodes download and store the object in their cache, less number of requests needs to be served through the CP's server, resulting lower costs. For this experiment, the value λ was set to 0.68 which is the optimal value when β is 0.8.

At steady state, each node stores object 1 through objet 34 (i.e., λC) in the duplicate segment of their individual cache, and the remaining 14 slots in the cache are filled with unique objects. At time 30,000 seconds, the object popularity profile is altered by swapping the popularity of objects 1 to 10 with popularity of objects 1,000 to 1,010. Meaning, object 1,000 becomes the most popular object and object 1 becomes the 1,000th popular object. A reduction in local hit rate and an increase in remote hit rate can be observed immediately after this alteration in Fig. 8b. Due to this low local hit rate and high remote hit rate, the provisioning cost suffers from an immediate surge after 30,000 second. The algorithm, however, is able to gradually bring the cost down by storing the new set of popular objects in the caches in an optimal manner. Figs. 8a and 8b demonstrate the effects of another popularity alteration that is created again at 40,000 second. The surge in provisioning cost in this case is larger because of the intensity (i.e., size) of popularity changes. The demonstrated dynamics in this section show how the proposed cooperative caching can cope with runtime popularity alteration which is expected in a social network due to external news and events.

8.5 Performance with Nonstationary Networks

The stationary partition assumption is relaxed in this section. We evaluated *Split Cache* and the traditional policies on a dynamic 98-node *SWNET* formed by 98 individuals attending the INFOCOM '05 conference [8]. We have extracted the *SWNET* partition dynamics from a pair-wise interaction trace obtained from [2]. The trace contained synchronized time-stamped pair-wise individual interaction information with a granularity of 4 minutes, which is the Hello packet interval used by a small RF transceiver attached to all 98 individuals while attending the conference. Fig. 9a reports the extracted partition dynamics as the average partition size from individual nodes' perspective. For example, at time 20, average partition size across all nodes is 12.

Fig. 9b depicts the simulated cost as a function of λ. Observe that the pattern in this graph is exactly the same as that observed for the stationary partition case in Fig. 6c, indicating that the concept of optimal λ also holds for networks with dynamic partitions. Analytical computation of the λ_{opt} in this dynamic case, however, may not be as straightforward due to the wide variation of the partition size as shown in Fig. 9a. A heuristic approach would be to compute λ_{opt} for each node individually based on its own observed average partition size.

Also, unlike in the stationary case, it is relatively harder to keep consistency of duplication under the dynamic scenario. This is because when a node is in a small partition, it has to download a large number of objects from the CP's server. In other word, from the standpoint of a node, which is in a small partition, those objects are unique. Later, when such a node enters a bigger partition, some of those unique objects may not remain unique anymore in the new partition. To avoid such situations, current partition size is stored along with the object in the cache and during cache replacement objects with smaller partition size are evicted before other objects.

Fig. 9c depicts that *Split Cache* with parameter λ_{opt} provides the best cost compared to the traditional policies even with dynamic *SWNET* partitions. Similar to the stationary case, $\lambda = 0$ and $\lambda = 1$ deliver near-best performance for small and large β values, respectively. Also note that the cost for all policies except *Split Cache* with parameter grows linearly with β. This is because the quantities P_L and P_V for these policies do not depend on β. Therefore, as seen in (4), the cost is simply a linear function of β. One main difference between the dynamic and stationary network scenarios is that for the dynamic

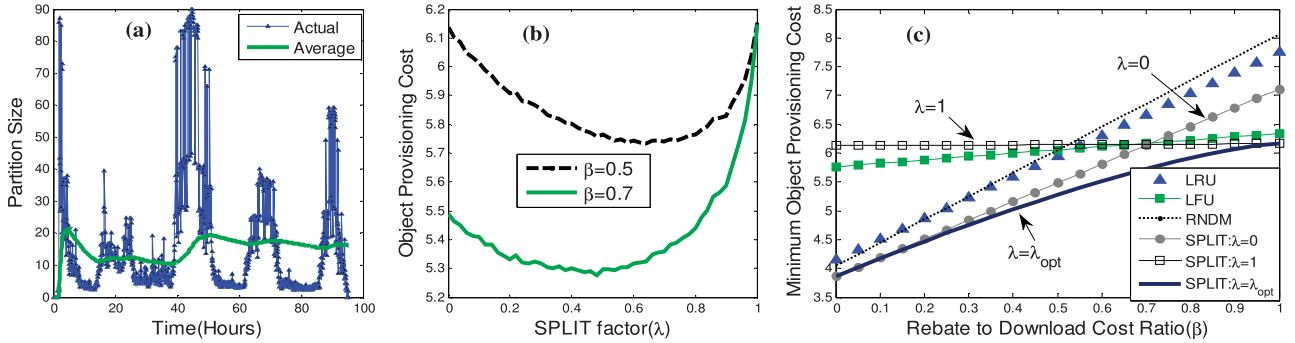

Fig. 9. (a) SWNET partition dynamics from human interaction traces. (b) Provisioning cost. (c) Comparative minimum cost.

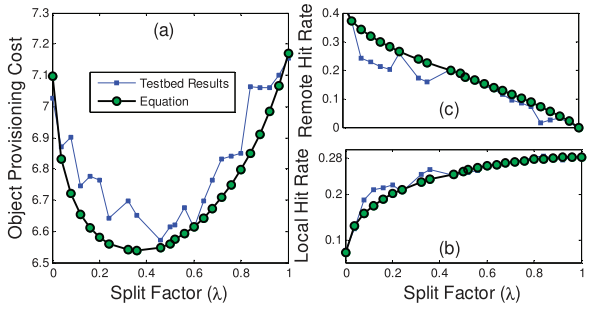

Fig. 10. (a) Cost and (b) local and (c) remote hit rates.

case, *Split Cache* policy with $\lambda = 0$ outperforms LRU and RNDM for all values of β.

9 Android *SWNET* Testbed

The *Split Cache* protocol was also implemented as an Android App on a seven-phone Social Wireless Network. Based on Zipf distribution over 5,000 objects, each node was programmed to generate 1 request per second. The requests are homogeneous in these experiments. Each phone is able to store up to 50 different objects in its local cache (i.e., $C = 50$). After generating a request for an object, a phone first checks its local cache and if its local search fails, it searches the object in the other six phones using a ad hoc WiFi network acting as the interphone peer-to-peer links. If the node does not receive a reply within two seconds after sending the request, it downloads the object directly from a desk-top machine that emulates the CP's server. Note that any object downloaded directly from the CP's server is considered as a unique object and it is stored in the unique area of the cache.

Fig. 10a reports object provisioning costs from both the analytical expressions and from the testbed when λ varies between 0 and 1, and the rebate to cost ratio β is set to 0.5. The cost is analytically computed according to (11) when the parameters m, C, and α are set to 7, 50, and 0.8, respectively.

Observe that although the costs obtained from the testbed maintain values and trends very similar to those from the equation, they are always slightly higher. These differences stem from undesired object duplication as a result of search inaccuracy as follows: when two or more nodes register cache misses at the same time for a supposedly unique object, all of them may attempt to download the object from the CP's server. This can result in undesired object duplications, causing an effective λ which is larger than the target λ_{opt}. Such faulty duplication was also found to happen due to erroneous object search in the events of lost search requests in the WiFi phone network. The impacts of undesired object duplication are higher local hit rates and lower remote hit rates (compared to the equation) and therefore higher provisioning costs.

It should be also observed that the higher costs due to undesired object duplication happens more when λ is small (i.e., $\lambda \ll 1$). This is because when λ is very small, the local hit rate is very low. Thus, the number of search requests to the other nodes is quite high. As a result, the absolute number of simultaneous requests and lost search requests as described above are also high. These cause more frequent erroneous object duplications and subsequently higher cost.

10 Performance of Benefit-Based Heuristics

In this section, we study the *Distributed Benefit* heuristics when nodes have different request rates and request patterns. To create node-specific object popularity profiles we have used the following web proxy and web server traces:

BU [9]: A Boston University's proxy trace which contains access information of 28 end users requesting pages from 1840 distinctly different websites during April and May, 1998.

NLANR [23]: A one day trace of HTTP requests to four proxy caches at the National Lab for Applied Network Research, on 10 January 2007. This trace contains access information of 117 end users to 241,173 different websites.

For the above two proxy traces we map the websites to individual objects, and the users to *SWNET* mobile devices.

NASA [24]: This trace contains access information of 81,983 clients to 21,670 web pages of the NASA Kennedy space center web server in Florida during July 1995.

SASK [25]: This trace contains access information of 162,523 clients to 36,825 webpage of a web server in the University of Saskatchewan during June to December of 1995. For the NASA and SASK traces, we map the webpages to individual objects, and the clients to *SWNET* mobile devices.

Since the smallest number of clients among all four traces is 28 (i.e., for BU), in order to be able to compare the results across all traces, we extract the access information of 28 nodes with the highest request generation rate from all the trace files and use them in the caching simulation. In all following simulation experiments nodal cache size is set to 25.

Fig. 11a depicts the *global popularity* of objects in BU and NLANR traces. Global popularity of object "i" is computed as

$$\text{global popularity}_i = \frac{\text{number of requests in the network for object } 'i'}{\text{The total number of requests in the network}}.$$

It can be observed that the graph in Fig. 11a closely follows a straight line on a log-log scale, indicating the Zipf distribution [5] for object requests as assumed in Section 2.4.

Fig. 11b depicts the cumulative probability density function of global popularity for both BU and NLANR traces. Observe that when the requests are generated from the BU trace, by storing the first 25 popular objects, each node is able to find 40 percent of its requested objects in theist local cache. This number is around 20 percent for requests following the NLANR trace. This confirms that the object popularity in the BU trace is indeed more skewed compared to the NLANR trace.

The probability of generating a request for an object in a single node is referred to as the *local popularity* at that node. Similar to *global popularity*, *local popularity* also follows a Zipf distribution. However, the set of objects from a single node's standpoint is smaller compared to that in the entire network. The *Local popularity* of object "i" at node "j" can be computed as

$$\text{local popularity}_i^j = \frac{\text{number of requests from node } 'j' \text{ for object } 'i'}{\text{The total number of requests from node } 'j'}.$$

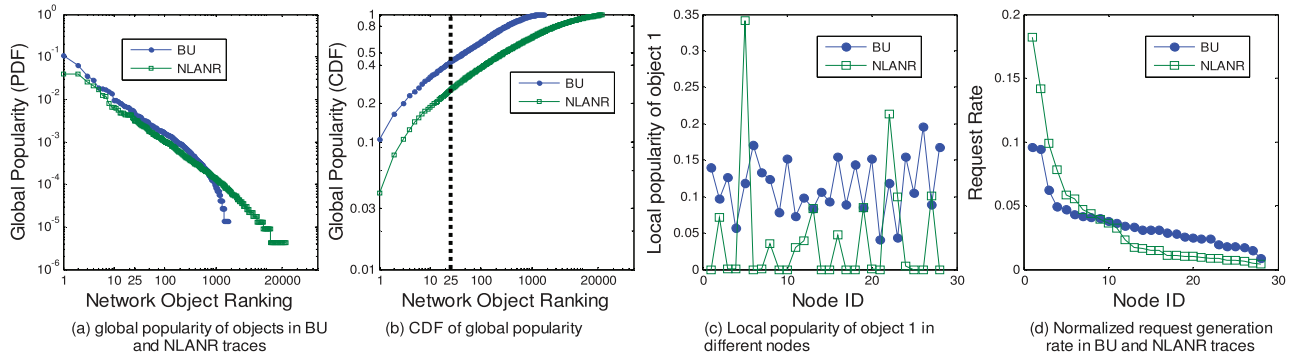

Fig. 11. (a) and (b) PDF and CDF of global popularity for accessed objects in BU and NLANR. (c) Local popularity of the most global popular object. (d) Normalized request generation rate in BU and NLANR trace files.

The *local popularity* of objects is expected to be different at different nodes. For the BU and NLANR traces, Fig. 11c depicts the *local popularity* of the most globally popular objects from the standpoint of different individual nodes in the network.

Fig. 11d depicts the normalized (by network-wide request rate) request generation rates from different nodes. As shown, few nodes are more active and generate more requests per unit time compared to the others. Individual node-specific request rates can have a significant impact on the average object provisioning cost, and therefore it is crucial to consider this parameter in object placement algorithm as presented in Algorithm 2 for the *Distributed Benefit* strategy and its associated text in Section 6. It can be seen that the diversity of request generation rate for NLANR is higher than that of BU.

Figs. 12a and 12b depict the object provisioning cost for *Distributed Benefit*, *Split Cache*, linear programming (cost lower bound), and traditional LRU with the BU and NLANR traces. Due to the heterogeneous nature of those traces, (11) cannot be used for finding the optimal λ in *Split Cache*. Instead, the optimal λ for *Split Cache* is experimentally found by running the protocol for all possible values of λ, and then selecting the one that generates the minimum cost. This minimum cost is shown as the *Best Split*. Note that LRU is the only representative traditional cache replacement policy for which the results are included in Fig. 12. This is because it outperformed the other traditional policies, namely, LFU and RNDM.

The following observation can be made from Figs. 12a and 12b. First, due to optimal object placement, the linear programming has lowest cost compared to those in the other approaches. The cost difference stems mainly from offloading objects from the active nodes (i.e., with very high request generation rate) to other less active nodes as explained in Section 6.3.

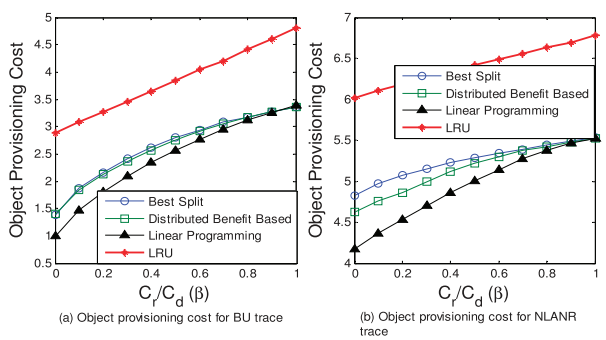

Fig. 12. Cost for heterogeneous demands: (a) BU and (b) NLANR.

Second, the cost in *Distributed Benefit* is always less that with *Best Split* (i.e., *Split Cache* run with the experimentally found optimal λ) and *LRU* replacement policy. The reason is that *Distributed Benefit* attempts, although heuristically, to attain the same object placement goals as by the cost lower bound obtained by *linear programming*. It is however noted that there exists room for improving the *Benefit-Based* heuristics in order to reduce its cost to the lower bound obtained by the linear programming.

Third, the cost increases with increasing β because by increasing β, the benefit of cooperative caching is reduced. In an extreme case, when $\beta = 1$, nodes can rely solely on their local cache for reducing the cost. In that case, the performance of *Best Split*, *Distributed Benefit* and *linear programming* become similar.

Fourth, we can also see that for the experiment with the BU trace, *Best Split* and *Distributed Benefit* offer almost the same provisioning cost whereas with the NLANR trace, the difference between the two mechanisms is relatively higher. The reason is that the diversity of request generation rate in the BU trace is less than that in the NLANR trace (see Fig. 11d). Furthermore, the variation of local popularity of objects in NLANR is much higher than that in BU. This is demonstrated in Fig. 11c. To summarize, the lack of diversity in local popularity and request generation rates in the BU trace make this request model perform very similar to the homogeneous case. As a result, the *Split Cache* mechanism is able to provide the same provisioning cost as *Distributed Benefit* whereas in NLANR due to the higher heterogeneity of request generation rates and local popularities, the *Distributed Benefit* heuristics provides better results compared to the *Split Cache* with best performing λ.

Finally, as Fig. 12 reports, the object provisioning cost for the BU trace is lower than that for the NLANR trace. This can be explained from the graph in Fig. 11b which shows the cumulative probability density function of popular objects for the BU and the NLANR traces. It can be observed that in the BU trace, storing the same number of objects

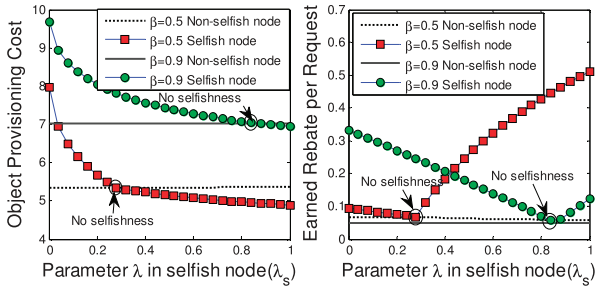

Fig. 13. (a) Cost and (b) rebate with one selfish node in *SWNET*.

results in higher hit rates compared to NLANR. In other words, the Zipf distribution parameter α in BU is higher than that for NLANR, which results in lower provisioning cost for BU. Experiments with the NASA and SASK traces showed performance differences very similar to those between the BU and NLANR traces.

11 Split Cache Performance with Selfishness

11.1 Networks with Single Selfish Node

Fig. 13a demonstrates the impacts of different λ_s on the object provisioning cost of nonselfish nodes when there is exactly one selfish node in the network. The average provisioning cost for nonselfish node does not change significantly in the presence of a single selfish node. However, the object provisioning cost for selfish node reduces as we increase λ_s. The reason is that more objects can be found locally as we increase λ_s.

Fig. 13b demonstrates the amount of rebate for each object request when there is exactly one selfish node in the network. With a single selfish node, choosing any λ_s that is different from the optimal λ increases the amount of rebate for the selfish node. The maximum value of earned rebate, however, depends on the value of λ_{opt} which is a function of β. For example, when $\beta = 0.9$ (i.e., when the optimal value for λ is around 0.81), a selfish node can maximize its earned rebate by setting λ_s to 0. On the other hand, when $\beta = 0.5$ (i.e., when λ_s is equal to 0.21) the selfish node's rebate is maximized when its λ_s is set to 1.

In summary, a single selfish node can maximize its own rebate by setting λ_s to either 0 or 1, whichever is farther than λ_{opt}. The "No Selfishness" marked points in Figs. 13a and 13b correspond to the situation where $\lambda_s = \lambda_{opt}$.

11.2 Networks with Multiple Selfish Nodes

11.2.1 Impacts on Cost

Fig. 14a depicts the impacts of selfish node-count on the object provisioning cost when $\beta = 0.9$. As expected, any deviation from the optimal policy increases the average provisioning cost in the network. For $\beta = 0.9$, the selfish nodes choose $\lambda_s = 0$ to maximize their rebates. Meaning, selfish nodes store only unique objects in their cache which in turn increase their provisioning cost. By increasing the number of selfish nodes, the number of uniquely stored objects in the network also increases. This new set of unique objects reduces the cost for the nonselfish nodes.

Fig. 14b demonstrates the cost of provisioning objects to the selfish nodes, the nonselfish nodes, and the network wide average when β is set to 0.5. In this case, the selfish nodes set λ_s to 1, causing them to store the most popular objects in their local cache. This helps the cost of object provisioning to the selfish nodes to come down. The cost for the nonselfish nodes, however, increases in the presence of selfish nodes due to the following two reasons: 1) a selfish node prevents other cooperative nodes to store popular objects by storing the most popular objects in its cache (remember that a nonselfish node will not store a duplicated object in the unique area of its cache). 2) a selfish node wastes the global cache capacity by filling its cache with duplicated objects. As a result, less number of objects is stored in the network, which in turn reduces the chance of finding a requested object in remote caches. The excellent agreement between the analytical (i.e., EQ) and the simulation results (i.e., SIM) proves the correctness of (20) through (28).

11.2.2 Impacts on Rebate

Earning higher rebate is the only motivation for a node to run the selfish policy. From the rebate standpoint, a *steady state* can be defined as a situation in which a node cannot deviate from the optimal caching policy to earn higher rebate. In this section, we demonstrate the existence of such steady state in a typical social wireless network.

Fig. 14c represents the amount of rebate per object request earned by the selfish and the nonselfish nodes when β is set to 0.9. Initially, when only few nodes deviate from the optimal policy, they are able to supply enough unique objects to the nonselfish nodes so that the earned rebate is higher. By increasing the number of selfish nodes, the rebate per request for each selfish node reduces for two

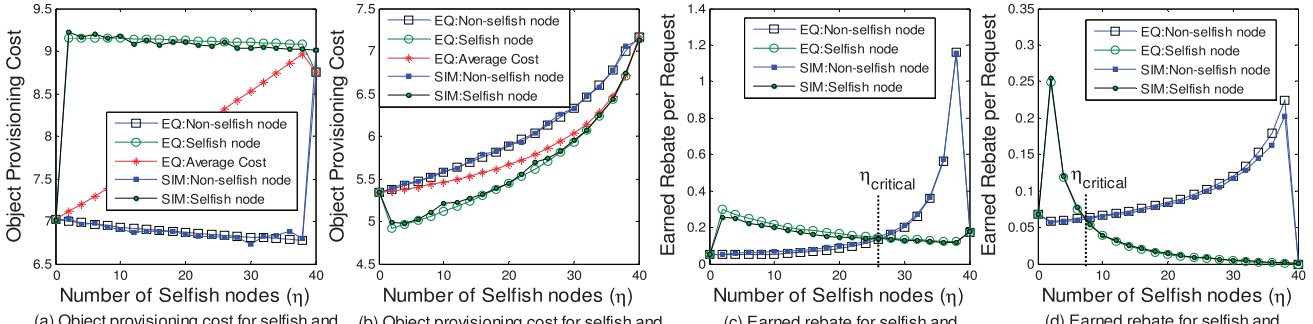

Fig. 14. (a) and (b) Object provisioning cost. (c) and (d) Earned rebate per request for nonselfish and selfish nodes.

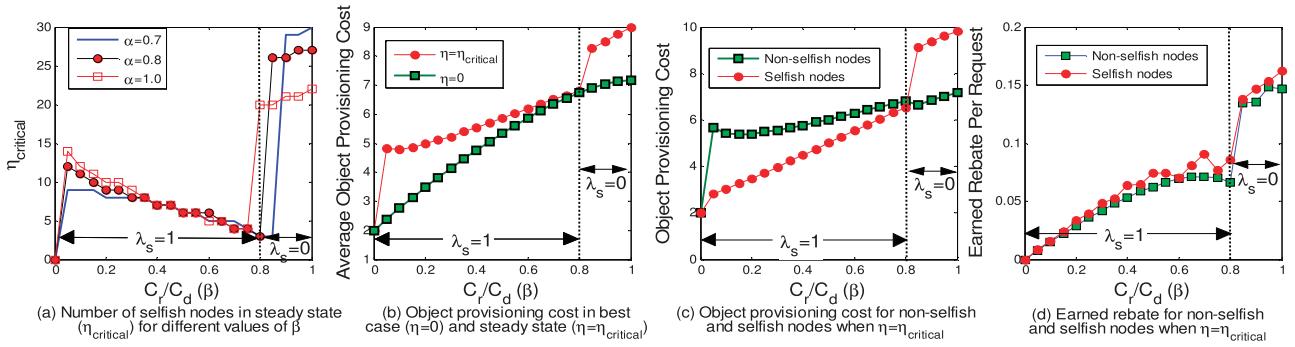

Fig. 15. Analysis of rebate and object provisioning cost in steady state (i.e., $\eta = \eta_{\text{critical}}$).

reasons: 1) number of requests from nonselfish nodes becomes less, and 2) rebate must be shared among more number of selfish nodes. When the number of selfish nodes reaches a critical value η_{critical}, the rebate for selfish and nonselfish nodes become equal. In fact, when a node chooses a selfish policy while there are η_{critical} selfish nodes in the network, its rebate become less than that of the nonselfish nodes. This leads to an important claim, namely, *having more than η_{critical} selfish nodes in an SWNET does not gain the selfish nodes*.

Fig. 14d represents the amount of rebate for selfish and nonselfish nodes when β is set to 0.5. The value of λ_s at the selfish nodes is set to 1, so that the earned rebates by those nodes are maximized. Observe that for higher β (Fig. 14c), η is also higher. Meaning, more nodes can be selfish and still receive higher rebates compared to the nonselfish nodes.

11.3 Steady State Analysis

This section presents analysis in the steady state, which is when a network contains exactly $\eta_{critical}$ number of selfish nodes. Fig. 15a depicts the impacts of β on $\eta_{critical}$, which represents the maximum number of nodes that can run the selfish policy and still get higher rebates compared to the nonselfish nodes. When β is small, the optimal λ_{opt} is also small which means each nonselfish node stores only a very few popular duplicated objects in its cache. Therefore, a selfish node can earn a high rebate by storing the most popular objects in its local cache (i.e., choosing $\lambda_s = 1$) and serving a large number of requests from other nodes. With increasing β, the quantity λ_{opt} increases, which causes each node to store more popular objects in its local cache and therefore, the number of remote requests to the selfish nodes reduces. This in turn reduces the rebate difference between the selfish and the nonselfish nodes, and therefore less number of nodes can run the selfish policy. This explains the decreasing trend (see Fig. 15a) of $\eta_{critical}$ as β changes from 0 to 0.8. At $\beta = 0.8$, the amount of rebate for the selfish nodes is very close to the amount of rebate for the nonselfish nodes. As a result, only very few nodes can benefit from running the selfish policy. For $\beta > 0.8$, the selfish nodes must set λ_s to 0 in order to get higher rebates compared to the cooperative nodes. In this case, the difference between the rebate earned by the selfish and the nonselfish nodes becomes very high which encourages a lot of nodes to be selfish, thus drastically increasing the quantity $\eta_{critical}$.

Fig. 15b demonstrates the average object provisioning cost in the presence of η_{critical} selfish nodes. For $\beta = 0$, as there is no selfish node in the network, the provisioning cost in this case represents the minimum possible value. In other cases, as expected the average provisioning cost is always higher than the provisioning cost when all nodes run the optimal policy. Observe that the impacts of selfishness for small βs are always higher than those for large βs. For higher β, the nonselfish nodes store more popular objects in their local cache and they become less sensitive to the presence of selfish nodes. For $\beta = 0.8$, the average provisioning cost in the presence of selfish nodes is very close to the minimum possible provisioning cost. For β greater than 0.8, the selfish nodes set λ_s to zero and as demonstrated in Fig. 15a, the number of selfish nodes η_{critical} becomes very large. Because of too many selfish nodes, the difference between the cost for nonselfish and selfish nodes becomes noticeable again.

Fig. 15c depicts the cost of object provisioning for the selfish and the non-selfish nodes in the presence of η_{critical} selfish nodes in the network. For small βs, the cost of provisioning objects to the selfish nodes is much lower than that for the nonselfish nodes. This is because for small βs the selfish nodes set $\lambda_s = 1$ and store popular objects locally. Therefore, a high percentage of requests in selfish nodes are satisfied locally without any provisioning cost. The difference between nonselfish and selfish nodes becomes less as β increases because due to higher λ_{opt}, the nonselfish nodes also start storing more popular objects. After $\beta = 0.8$, the cost for provisioning to the selfish nodes becomes higher than that to the nonselfish nodes. The reason is by choosing $\lambda_s = 0$, a selfish node is deprived of having popular objects and only a few percentage of requests in selfish nodes in this case can be satisfied from the local caches, which in turn increases the provisioning cost.

Fig. 15d depicts the rebate earned by the selfish and the nonselfish nodes. Observe that the amount of rebate earned by the selfish nodes is close and always higher than the amount of rebate earned by the nonselfish nodes. Adding even a single selfish node beyond η_{critical} brings the amount of rebate for the selfish nodes below that of the nonselfish nodes. The sharp jump in the rebate at $\beta = 0.8$ is because of switching from $\lambda_s = 1$ to $\lambda_s = 0$.

12 Related Work

There is a rich body of the existing literature [10], [11] on several aspects of cooperative caching including object replacements, reducing cooperation overhead [12], and

cooperation performance in traditional wired networks. The Social Wireless Networks explored in this paper, which are often formed using mobile ad hoc network protocols, are different in the caching context due to their additional constraints such as topological insatiability and limited resources. As a result, most of the available cooperative caching solutions for traditional static networks are not directly applicable for the *SWNETs*.

Three caching schemes for MANET have been presented in [13]. In the first scheme, CacheData, a forwarding node checks the passing-by objects and caches the ones deemed useful according to some predefined criteria. This way, the subsequent requests for the cached objects can be satisfied by an intermediate node. A problem with this approach is that storing large number of popular objects in large number of intermediate nodes does not scale well.

The second approach, CachePath, is different in that the intermediate nodes do not save the objects; instead they only record paths to the closest node where the objects can be found. The idea in CachePath is to reduce latency and overhead of cache resolution by finding the location of objects. This strategy works poorly in a highly mobile environment since most of the recorded paths become obsolete very soon. The last approach in [13] is the HybridCache in which either CacheData or CachePath is used based on the properties of the passing-by objects through an intermediate node. While all three mechanisms offer a reasonable solution, it is shown in [14], [15], and [16] that relying only on the nodes in an object's path is not most efficient. Using a limited broadcast-based cache resolution can significantly improve the overall hit rate and the effective capacity overhead of cooperative caching.

According to the protocols in [17] the mobile hosts share their cache contents in order to reduce both the number of server requests and the number of access misses. The concept is extended in [15] for tightly coupled groups with similar mobility and data access patterns. This extended version adopts an intelligent bloom filter-based peer cache signature to minimize the number of flooded message during cache resolution. A notable limitation of this approach is that it relies on a centralized mobile support center to discover nodes with common mobility pattern and similar data access patterns. Our work, on the contrary, is fully distributed in which the mobile devices cooperate in a peer-to-peer fashion for minimizing the object access cost.

In summary, in most of the existing work on collaborative caching, there is a focus on maximizing the cache hit rate of objects, without considering its effects on the overall cost which depends heavily on the content service and pricing models. This paper formulated two object replacement mechanisms to minimize the provisioning cost, instead of just maximizing the hit rate. Also, the validation of our protocol on a real *SWNET* interaction trace [2] with dynamic partitions, and on a multiphone Android prototype is unique compared to the existing literature.

From a user selfishness standpoint, Laoutaris et al. [20] investigate its impacts and mistreatment on caching. A mistreated node is a cooperative node that experiences an increase in its access cost due to the selfish behavior by other nodes in the network. In [21], Chun et al. study selfishness in a distributed content replication strategy in which each user tries to minimize its individual access cost by replicating a subset of objects locally (up to the storage capacity), and accessing the rest from the nearest possible location. Using a game theoretic formulation, the authors prove the existence of a pure Nash equilibrium under which network reaches a stable situation. Similar approach has been used in [22] in which the authors model a distributed caching as a market sharing game.

Our work in this paper has certain similarity with the above works as we also use a monetary cost and rebate for content dissemination in the network. However, as opposed to using game theoretic approaches, we propose and prove an optimal caching policy. Analysis of selfishness in our work is done in a steady state over all objects whereas the previous works mainly analyze the impact of selfishness only for a single data item. Additionally, the pricing model of our work which is based on the practical Amazon Kindle business model is substantially different and practical compared to those used in [21] and [22].

13 Summary and Ongoing Work

The objective of this work was to develop a cooperative caching strategy for provisioning cost minimization in Social Wireless Networks. The key contribution is to demonstrate that the best cooperative caching for provisioning cost reduction in networks with homogeneous content demands requires an optimal split between object duplication and uniqueness. Such a split replacement policy was proposed and evaluated using ns2 simulation and on an experimental testbed of seven android mobile phones. Furthermore, we experimentally (using simulation) and analytically evaluated the algorithm's performance in the presence of user selfishness. It was shown that selfishness can increase user rebate only when the number of selfish nodes in an *SWNET* is less than a critical number. It was shown that with heterogeneous requests, a benefit-based heuristics strategy provides better performance compared to split cache which is proposed mainly for homogeneous demand.

Ongoing work on this topic includes the development of an efficient algorithm for the heterogeneous demand scenario, with a goal of bridging the performance gap between the Benefit Based heuristics and the centralized greedy mechanism which was proven to be optimal in Section 6.4. Removal of the no-collusion assumption for user selfishness is also being worked on.

Acknowledgments

The authors would like to acknowledge Dr. Eric Torng and Dr. Charles Ofria for their contribution and input to this paper.

References

[1] M. Zhao, L. Mason, and W. Wang, "Empirical Study on Human Mobility for Mobile Wireless Networks," *Proc. IEEE Military Comm. Conf. (MILCOM)*, 2008.

[2] "Cambridge Trace File, Human Interaction Study," http://www.crawdad.org/download/cambridge/haggle/Exp6.tar.gz, 2012.

[3] E. Cohen, B. Krishnamurthy, and J. Rexford, "Evaluating Server-Assisted Cache Replacement in the Web," *Proc. Sixth Ann. European Symp. Algorithms,* pp. 307-319, 1998.
[4] S. Banerjee and S. Karforma, "A Prototype Design for DRM Based Credit Card Transaction in E-Commerce," *Ubiquity,* vol. 2008, 2008.
[5] L. Breslau, P. Cao, L. Fan, and S. Shenker, "Web Caching and Zipf-Like Distributions: Evidence and Implications," *Proc. IEEE INFOCOM,* 1999.
[6] C. Perkins and E. Royer, "Ad-Hoc On-Demand Distance Vector Routing," *Proc. IEEE Second Workshop Mobile Systems and Applications,* 1999.
[7] S. Podlipnig and L. Boszormenyi, "A Survey of Web Cache Replacement Strategies," *ACM Computing Surveys,* vol. 35, pp. 374-398, 2003.
[8] A. Chaintreau, P. Hui, J. Crowcroft, C. Diot, R. Gass, and J. Scott, "Impact of Human Mobility on Opportunistic Forwarding Algorithms," *IEEE Trans. Mobile Computing,* vol. 6, no. 6, pp. 606-620, June 2007.
[9] "BU-Web-Client - Six Months of Web Client Traces," http://www.cs.bu.edu/techreports/1999-011-usertrace-98.gz, 2012.
[10] A. Wolman, M. Voelker, A. Karlin, and H. Levy, "On the Scale and Performance of Cooperative Web Caching," *Proc. 17th ACM Symp. Operating Systems Principles,* pp. 16-31, 1999.
[11] S. Dykes and K. Robbins, "A Viability Analysis of Cooperative Proxy Caching," *Proc. IEEE INFOCOM,* 2001.
[12] M. Korupolu and M. Dahlin, "Coordinated Placement and Replacement for Large-Scale Distributed Caches," *IEEE Trans. Knowledge and Data Eng.,* vol. 14, no. 6, pp. 1317-1329, Nov. 2002.
[13] L. Yin and G. Cao, "Supporting Cooperative Caching in Ad Hoc Networks," *IEEE Trans. Mobile Computing,* vol. 5, no. 1, pp. 77-89, Jan. 2006.
[14] Y. Du, S. Gupta, and G. Varsamopoulos, "Improving On-Demand Data Access Efficiency in MANETs with Cooperative Caching," *Ad Hoc Networks,* vol. 7, pp. 579-598, May 2009.
[15] C. Chow, H. Leong, and A. Chan, "GroCoca: Group-Based Peer-to-Peer Cooperative Caching in Mobile Environment," *IEEE J. Selected Areas in Comm.,* vol. 25, no. 1, pp. 179-191, Jan. 2007.
[16] F. Sailhan and V. Issarny, "Cooperative Caching in Ad Hoc Networks," *Proc. Fourth Int'l Conf. Mobile Data Management,* pp. 13-28, 2003.
[17] C. Chow, H. Leong, and A. Chan, "Peer-to-Peer Cooperative Caching in Mobile Environments," *Proc. 24th Int'l Conf. Distributed Computing Systems Workshops,* pp. 528-533, 2004.
[18] A. Schrijver, *Theory of Linear and Integer Programming.* Wiley-Interscience, 1986.
[19] H.K. Kuhn, "The Hungarian Method for the Assignment Problem," *Naval Research Logistics,* vol 52, no. 1, pp. 7-21, 2005.
[20] N. Laoutaris et al., "Mistreatment in Distributed Caching: Causes and Implications," *Proc. IEEE INFOCOM,* 2006.
[21] B. Chun et al., "Selfish Caching in Distributed Systems: A Game-Theoretic Analysis," *Proc. 23th ACM Symp. Principles of Distributed Computing,* 2004.
[22] M. Goemans, L. Li, and M. Thottan, "Market Sharing Games Applied to Content Distribution in Ad Hoc Networks," *IEEE J. Selected Areas in Comm.,* vol. 24, no. 5, pp. 1020-1033, May 2006.
[23] National Laboratory of Applied Network Research, Sanitized Access Log, ftp://ircache.nlanr.net/Traces, July 1997.
[24] NASA Kennedy Space Center, WWW Server Access Log, ftp://ita.ee.lbl.gov/traces/NASA_access_log_Jul95.gz, 2012.
[25] Univ. of Saskatchewan, WWW Server Access Log, ftp://ita.ee.lbl.gov/traces/usask_access_log.gz, 2012.
[26] M. Taghizadeh, A. Plummer, A. Aqel, and S. Biswas, "Optimal Cooperative Caching in Social Wireless Networks," *Proc. IEEE Global Telecomm. Conf. (GlobeCom),* 2010.
[27] A. Almohamad and S.O. Duffuaa, "A Linear Programming Approach for the Weighted Graph Matching Problem," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 15, no. 5, pp. 522-525, May 1993.

Mahmoud Taghizadeh received the BS and MS degrees in computer engineering from the Sharif University of Technology in Tehran, and the PhD degree in the electrical and computer engineering from Michigan State University. His current research interests include the field of wireless networking, cooperative caching algorithms, data dissemination, and vehicular networks. He is a recipient of three best paper awards from GlobeCom 2009, COMSNET 2011, and ICICS 2011. He is a member of the IEEE.

Kristopher Micinski received the BS degree in computer engineering from Michigan State University in 2011. He is currently beginning graduate school at the University of Maryland. His research interests focus on programming languages, but also include systems design and formal verification. He is a member of the IEEE.

Charles Ofria received his Ph.D. in Computation and Neural Systems from the California Institute of Technology in 1999. He is now the director of the MSU Digital Evolution Laboratory, deputy director of the BEACON Center for the Study of Evolution in Action, and an associate professor in the Department of Computer Science and Engineering at Michigan State University. His research lies at the intersection of computer science and evolutionary biology, developing a two-way flow of ideas between the fields.

Eric Torng received his Ph.D. degree in computer science from StanfordUniversity in 1994. He is currently an associate professor and graduatedirector in the Department of Computer Science and Engineering at Michigan-State University. He received an NSF CAREER award in 1997. His researchinterests include algorithms, scheduling, and networking.

Subir Biswas received the PhD degree from the University of Cambridge and held research positions at the NEC Research Institute, Princeton, AT&T Laboratories, Cambridge, and Tellium Optical Systems, New Jersey. He is a professor and the director of the Networked Embedded and Wireless Systems Laboratory at Michigan State University. He has published more than 135 peer-reviewed articles and coinvented six (awarded and pending) US patents. His current research includes pricing calculus in social networks, capacity scavenging in cognitive networks, UWB switching in sensor networks, safety and content-based applications in vehicular networks, and wearable sensing for health applications. He is a senior member of the IEEE and a fellow of the Cambridge Philosophical Society.