Let me grab your App: a preliminary proof-of-concept design of opportunistic content recommendation

Giuseppe Bianchi, Pierpaolo Loreti, Arijana Trkulja
University of Roma “Tor Vergata”, Italy, Email: name.surname@uniroma2.it

***Abstract*—With the rate of mobile traffic growth eventually outpacing 4G technology upgrades, finding new means to offload the cellular network appear crucial. Whereas exploitation of device-to-device opportunistic short range connectivity is a promising and widely investigated direction, it might ultimately play a negligible role for offloading purposes: in front of the huge overall content universe an user may select to access, it is unlikely that a specific user request can be found in a possibly small number of neighboring mobile devices. Focusing on a concrete use-case scenario, namely Apps download from Google’s Android market, we discuss a preliminary design and implementation of CarpeDroid, a proactive local coordination approach devised to recommend end users with download opportunities for content (likely to fall within the users’ interest) locally stored in neighboring devices. Our approach does *not* rely on any centralized or operator-assisted recommending system. Efficiency and “better than nothing” privacy protection is achieved through the exchange of Bloom filters devised to summarize content stored in the network neighbor.**

I. Introduction

The widespread diffusion of smartphones and tablets is creating an unbearable pressure on cellular networks. Recent analyses [1] predict an exponential growth in mobile traffic, mostly caused by mobile video retrieval/streaming, which some analyst even refers to as “mobile data apocalypse”. In 2010, global mobile data traffic nearly tripled for the third year in a row, and predictions to 2015 suggest a further 26-fold traffic increase topping 6.3 exabytes per month by 2015. According to some researchers’ opinion, emerging 4G technologies might not succeed in providing enough capacity to accommodate future mobile traffic. Thus, massive deployment of WiFi-based [2], [3], [4] or Femtocell-based [5] offloading solutions, as well as mobile content delivery architectures integrating broadcasting technologies [6] are being envisioned.

On top of this, we argue that a further help may come from the emergence of user-centric approaches, revolving around pure end-user-based collaboration (explicit or implicit) via short-range device-to-device communication technologies. Why a user should access a quality-scaled video stream from the resource constrained access link (and eventually pay for it, if no flat rate fare is contractualized) whenever the same content may be readily available a few meters away over another end-user terminal? Indeed, a huge effort has been spent on the design and performance assessment of opportunistic networking frameworks [7], [8], [9], including their tailoring to the specific cellular network offloading scenario [10], [11], [12]. These techniques leverage the relatively large storage capacity deployed in modern mobile devices, and are devised to permit direct exchange of content information among end terminals (eventually through a multiplicity of relay nodes) without resorting on any infrastructure.

However, opportunistic networking technologies, *by themselves*, may be of little help. With the huge universe of distinct named resources, and the necessarily limited storage space opportunistically available in neighboring mobile devices, only a marginal percentage of requests may be ultimately satisfied by short-range opportunistic downloads.

The traditional way to address this issue consists in devising solutions for smartly distributing popular content across end user devices. However, especially when not assisted by a centralized entity (e.g., an operator) having an holistic view of the overall network status, an effective content placement strategy requires to overcome crucial hurdles[1] ranging from the need to deploy complex device-to-device coordination means supported by very efficient mobility and contact predictions, to the need for end users to accept to use part of their storage as distributed cache for content otherwise of no interest.

A quite different direction for increasing the chances of short range downloads consists in *opportunistically recommending* content stored in the network neighbor to an end user which, otherwise, would download *something else* from the cellular network. An end user wishing to see “Spiderman” may eventually decide to alternatively see “Fantastic Four” (a different movie, but still science fiction from the same brand), as long as she gets to know that such movie is readily available a few meters away (and possibly at a better quality and/or at no price) over another end-user terminal. And if she ultimately decides to stick to the original choice, nothing is lost; rather, a supplementary offloading opportunity has been missed. Indeed, as argued in [13], and as well known since long time by search engines and business players in the advertisement and consumer domains, end users only rarely aim at addressing a precise, named, networked content or

[1]Note that we do *not* imply that these issues cannot be addressed; indeed several literature works, including (but not limiting to) [7], [8], [9], [10], [11], [12], propose smart solutions to some of these problems. Rather, we tend to believe that real world deployment chances may improve if these hurdles are *avoided*, rather than addressed.

resource. Rather, our day-to-day web browsing experience suggests that most content retrievals are typically composed of two clearly distinct phases: a first *generic look-up* for "something" we are interested into, e.g. by querying for some keywords via a search engine's web interface, followed by a *selection* among the several listed different digital items. In several cases, more than just one single item may satisfactorily match what we are generically looking for.

Our Contribution

This paper is a preliminary work focusing on the technical feasibility of least-invasive, fully distributed, and purely end-user operated opportunistic recommendation for content locally stored in neighboring devices and likely to fall in the user's interest.

Throughout this work, we rely on a concrete, proof-of-concept, user-case scenario, namely download of android Apps from the Google's Android Market. We design and preliminary implement CarpeDroid, a framework which permits to seamlessly *augment* the list of downloadable Apps made available over the smartphone user's interface with Apps of possible interest for the end user available in short-range connected mobile devices. In essence, CarpeDroid supports a lightweight, fully distributed, recommending system which does *not* rely on any centralized entity, and exploits only the following information: i) public knowledge on android Apps classification, ii) legacy response from the Google's Android Market, and iii) sketched information (via Bloom filters' [14], [15] exchange, for "better than nothing" privacy protection) about the availability of Apps stored in neighboring devices. Another important advantage of our proposed approach is usability: the end user relies on the usual graphical user interface, and may in principle even remain unaware that some of the listed Apps may be locally downloaded.

II. System basics and design choices

In this section we introduce and motivate the basic ideas upon which CarpeDroid design is based.

Classic operation: Look-up and Select

CarpeDroid aims at extending a content retrieval operation which, in most generality, we refer to as *look-up and select*. As discussed in the introduction, the end user interaction with the Internet frequently consists in first searching for some generic information or a more or less specific topic, and then selecting for download one among a multiplicity of listed results. For our purposes, it is worth to remark that, often, multiple *different* resources among those listed may satisfy the user's needs.

As a concrete example, let us focus on a classic transaction involving the Android Market. An user aiming to install a new functionality on her smartphone opens the Market App (pre-installed on its device) and queries the server with some keywords more or less related to the problem. We call this first query as the *Look-up* phase. The server replies with a list

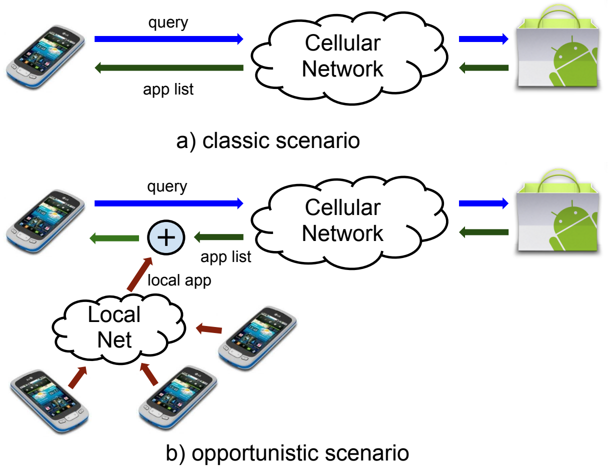

Fig. 1. Classic and opportunistic content retrieval scenario.

of Apps which the server provider *expects* to be of interest[2] for the end user. What follows is what we call the *Select* phase: the user scrolls the returned list of Apps, along with the supplementary information conveyed (price, other users' rating, etc), and finally selects the specific App to install, by clicking on the relevant entry and starting the App download, namely the most bandwidth consuming part of the overall transaction.

Proposed operation: Look-up, Expand, and Select

CarpeDroid operates by adding a *supplementary* intermediate phase (transparent to the end user) to the above described Look-up and Select operation (compare figure 1-b with 1-a). This phase is called "*Expand*". It runs on the end user device, taking as input:

- the list of Apps returned by the market server;
- an Apps' knowledge base pre-installed on the end user device, describing how Apps are classified into categories; and
- the list (actually a summary sketch, as discussed in the next section) of Apps available in the neighboring devices, suitably retrieved through a specifically devised short-range cooperation protocol.

Goal of the Expand phase is to identify which Apps, among those available in the local network neighbor, may be of interest for the end user on the basis of her actual request made during the Look-up phase. The Expand phase concludes by *adding* such supplementary identified Apps[3] to the original list retrieved from the Market server. If, during the following

[2]Typically, the construction of such list involves proprietary search and/or recommending algorithms which may further use historical and/or user profiling/behavioral information. It is *not* our goal to interfere with such proprietary algorithms; indeed, as shown in what follows the only information we rely upon is the *returned* list of Apps, independent of how it has been constructed by the Market service.

[3]The locally available Apps may be eventually rendered on the user terminal with a different color or style, to make the end user aware of their locality.

Select phase, the end user chooses to download one of such supplementary Apps, a direct short-range connection between the two devices is triggered instead of the Market download, thus saving cellular network resources.

The rationale behind the expand phase is to *further* recommend the user with Apps which can be *conveniently* retrieved from the network neighbor via short range communication. As such, we can envision the described operation as a distributed and purely end-user-operated way to fill a gap in the actual content selection and recommendation operated by the Market provider, a gap consisting in the Market provider's inability to base its own recommendation *also* on network convenience information (obviously, the list of results provided by the Market service do not account for the opportunistic availability of content locally stored in the network neighborhood).

Design choices

CarpeDroid design requires to address two major tasks. The first is how to gather information about the neighborhood status. As detailed in section III, our proposed information exchange is operated in background using short-range communication technologies and is opportunely compacted into Bloom filters. The second task is how to expand the list of Apps presented to the end user. This involves the design of a sort of lightweight distributed recommending system (we consider not practical the basic solution of presenting the end user with *all* the Apps available in the network neighbor), triggered at the time of the look-up phase. Our proposed approach, detailed in section IV, is devised to avoid any explicit cooperation with the neighboring devices (except the above Bloom filters' exchange) and relies only on information available on the end user terminal. Supplementary decisions are mostly implementation ones, and are discussed in section V, along with the current limitations and open issues.

III. Local information gathering

In principle, local information gathering may trivially reduce to each device being in charge of broadcasting over the network neighbor its list of stored Apps. In practice, two issues must be addressed.

One is merely technical. It consists in detailing how such broadcasting process is supported by available short range communication technologies, and how the neighborhood is defined. Our preliminary implementation discussed in section V relies on Bluetooth (for energy efficiency reasons), and limits the network neighbor to the directly connected devices (i.e. no multi-hop).

The second one is more meaningful, and revolves on the type and format of information to be exchanged. We believe that a direct broadcasting of the *raw* list of Apps stored on a device may be envisioned as questionable by the end users, mostly for privacy reasons. As such, we propose to exchange a *summary sketch* of the Apps stored in each device, in the form of a Bloom Filter. As discussed below, this choice has the advantage to provide a light, but at least "Better than nothing", privacy protection, as well as permit deniability (an user may ultimately deny that a specific App were stored on its device, blaming a Bloom filter's false positive). Moreover, the usage of Bloom filters further yields a low overhead, owing to the relatively small size of the exchanged filters (a filter of just 250 bytes may store up to 200 App names with a false positive in the order of 1%). However, the usage of Bloom filters is not free of disadvantages; indeed, it slightly complicates the design of the Expand phase, as discussed in section IV.

Bloom filters

We recall that a Bloom Filter [14], [15] is a probabilistic data structure used to represent set membership. A Bloom filter is implemented as an array of m bits accessed via k hash functions $H_1(x)...H_k(x)$, each of which maps a set member x to one of the m bits within the bit array.

Let $B[i]$ be the value of bit $1 \le i \le m$ within the bit array. Insertion of a set member x into a Bloom filter consists of setting $\forall j \in \{1..k\}, B[H_j(x)] \leftarrow 1$. Querying the presence of a set member x within a Bloom filter consists of computing $\forall j \in \{1..k\}, \min\{B[H_j(x)]\}$ (i.e., returning 1 only if *all* corresponding bits are 1).

In Bloom filters, false positives are possible but false negatives are not. A false positive is the probability that a set membership query returns 1 for an element *not* stored in the set. We recall from well known Bloom filter results [15] that, for a Bloom filter of size m, employing k hash functions, and storing n elements, the false positive probability ψ is closely approximated by the expression:

$$\psi \approx \left(1 - [1 - 1/m]^{km}\right)^k \approx \left(1 - e^{-nk/m}\right)^k.$$

Knowing in advance the number of elements n to be stored in the filter, the false positive probability is minimized by setting the number of hash functions to the integer value closer to $k = m/n \cdot \ln 2$. As shown in [15], it follows that for a given Bloom filter of size m, storing n elements, in such optimal conditions the false positive probability is given by:

$$\psi_{opt} = \left(\frac{1}{2}\right)^{\frac{m}{n}\ln 2} \approx 0.6185^{m/n}.$$

This last expression permits to easily parameterize the size of a Bloom filter with respect to the number n of (expected) stored elements. For instance, for a target 5% false positive probability, it suffices to deploy about $6.2 \times n$ bits, whereas for an 1% false positive target, the amount of memory required increases to slightly less than $10 \times n$.

Bloom filter usage and discussion

All devices participating to the local information gathering operation must preliminarily agree on the Bloom filter parameters, namely the Bloom filter size m, the number k of employed hash functions, and the algorithm used to compute such hash functions. Such parameters may be dimensioned by setting a target false positive probability in the assumption of a worst-case number of Apps stored on a device; in our implementation we use 200 bytes and 4 hash functions: this choice permits to store up to 256 App names with a false

TABLE I
EXAMPLE OF META DATA FOR AN ANDROID APPLICATION

Basic Info	
Meta	*value*
id	-6105825170608772224
title	Live soccer results
appType	APPLICATION
creator	mdtec
version	0.99.6
rating	4.352346383199378
ratingsCount	3857
creatorId	mdtec
packageName	net.mdtec.sportmate
versionCode	18
Extended Info	
description	live results, detailed statistics, ...
downloadsCount	0
permissionId	android.permission.INTERNET
installSize	2314161
packageName	net.mdtec.sportmate
category	Sport
contactEmail	info@mdtec.net

positive lower than 5% (dropping to less than 1% if no more than 152 Apps are stored).

Each device inserts all the downloadable App names in the filter to be broadcast to the network neighbor. To this purpose, note that a set member x inserted in a Bloom filter is an arbitrary sized string. For our purposes we could either use the App unique numeric identifier (handled as a string), or the App name (as we did, for convenience), or, to avoid ambiguity, an extended string composed of the App name followed by the App version, the creator identifier, and any other information deemed useful to uniquely identify an App, among the meta tags provided by the Android market for a given App. Such information includes the title, the identifier, a description, categories or tags for classification, etc (see the example in Table I).

A device receiving such so filled Bloom filters from the neighboring devices may trivially test whether a looked for App named x is locally stored, by querying the presence of the set member x within such filters. If neither query is successful, the App is *surely* not available in the network neighbor. Conversely, of one or more filters return a positive response, the App *may be* available (indeed, the positive response may be a false positive).

Note that the disclosure of a Bloom filter permits to query whether some *selected* App is therein stored (with the ambiguity given by the false positive probability), but by itself it does not provide the list of stored Apps in clear text. From a privacy perspective, this is clearly far from being a strong form of protection. Indeed, an attacker may trivially narrow the candidate set of stored Apps by running a so-called *Enumeration attack*, i.e. querying the Bloom filter with the universe set of deployed Apps, about 350.000 at the time of writing. Nevertheless, the false positive probability yields a superset of the actual subset of Apps stored in the device. For instance, an enumeration attack against a Bloom filter filled with 256 Apps (5% false positive, based on our parameters) would return about 17.500 candidate Apps stored in the device, versus the actual 256 stored ones. Note that this permits an end user to deny the accusation that a specific App is stored on her device. For these reasons, we believe that the weak, but "better than nothing", level of privacy protection provided by the usage of Bloom filters[4] may ultimately be acceptable by real world users, as well as by system developers (more advanced cryptographic protection techniques, such as searchable encryption [16], would bring about cumbersome key management issues).

IV. THE EXPAND PROCESS

The local information gathering procedure described in the previous section permits each node to determine whether a given App is stored in a neighboring device. In this section, we describe how this feature is exploited to expand the list of Apps returned to the end user for the Select phase.

The expand process uses three different types of information.

1) U: universe set of all possible Apps. These Apps are organized, by the Android Market, into N_C categories C_n. the Apps comprised in the category C_n are labeled c_i^n. Each device stores the set U in a local database (note that the database comprises just the App meta data, not the actual Apps).
2) B^m: set of the Apps b_i^m stored in the m-th node in the neighborhood; each set is represented in the form of a Bloom filter, as discussed in the previous section;
3) R: the set of Apps r_i^n returned by the market server in response to the user query.

The Expand process bases its operation on the assumption that a user looking for an App, classified by the Android Market as belonging into a given category, may be interested in *other* Apps belonging to the *same category*. The knowledge of which App belongs to which category is provided by the database U, pre-loaded in the user device.

Our design problem consists in mapping the user query, typically expressed in layman terms, hence using generic keywords, into actual Apps of potential interest for the user. This problem can be easily circumvented by using, as input of the Expand procedure, instead of the end user query, the actual *answer* provided by the server in the easy to process form of a list of App names and associated meta data. In other words, we rely on the "recommending logic" implemented in the Market server, and we derive further local recommendations by including all the Apps which either i) fall into a same category of at least one App included in the server response list (i.e. recommended by the Market service), and ii) are available in the network neighbor.

The detailed approach is illustrated in the Algorithm 1. For each App r_i^n returned by the Market server, the algorithm selects all Apps belonging to the same category in which r_i^n is

[4]note that a device storing a very small number of Apps, hence for which the false positive probability would significantly reduce with respect to the 5% example case (with 256 Apps), might randomly insert App names so as to artificially reach the obfuscation level granted by the 5% false positive target.

Algorithm 1 Expansion of the list of results coming from the server

```
Input: List of the $N_R$ results $r_i^n$ coming from the server; list of all items $c_i^n$ stored in the local db and $i = 1, ..., N_n$ (the index $n$ indicates that $\{r_i^n, c_i^n\} \in C_n$); Collected Bloom filters $B^m$.
Initialization: Iteration index $i = 0$. List of index of categories $L_C = []$. Output set $E = []$.
for $i = 1$ to $N_R$ do
    Select $r_i^n$
    if $n \notin L_C$ then
        add all $c_i^n$ to $E$
        add $n$ to $L_C$
    end if
end for
Output: The subset of the elements in $E$ which are also found in at least one among the Bloom filters $B^m$.
```

classified according to the local database. Then, it outputs the *subset* of such selected Apps which are found to be available in the gathered Bloom Filters. The found items are added to the results list presented to the end user. The locally available items may be optionally suitably rendered to permit the end user to distinguish them from the original Market response (and hence privilege locally stored content).

Illustrative example

Let's suppose to have a local database with just three categories C_1, C_2 and C_3 and the items $\{c_1^1, c_2^1, c_3^1, c_4^1, c_5^1\} \in C_1$, $\{c_1^2, c_2^2, c_3^2, c_4^2, c_5^2, c_6^2\} \in C_2$ and $\{c_1^3, c_2^3, c_3^3, c_4^3\} \in C_3$. The user execute a search query using a string that produces for example the following list of results $\{c_1^1, c_3^2, c_4^1, c_2^1, c_2^2\}$.

Since the results are in the C_1 and C_2 categories, the procedure creates a new list with all the *missing elements* in C_1 and C_2 i.e. $E = \{c_3^1, c_4^1, c_5^1, c_3^2, c_4^2, c_5^2, c_6^2\}$.

Let suppose now to have the information on the items $\{c_6^2, c_7^3, c_5^1, c_7^4, \}$ accumulated in the bloom filters collected fron the various devices in the neighborhood. The item in E are tested against the bloom filters and the result is clearly $\{c_6^2, c_5^1\}$.

Thus the list presented to the user is the union of the results coming from the server and the results in the local environment, i.e. $\{c_1^1, c_3^2, c_4^1, c_2^1, c_2^2\} \cup \{c_6^2, c_7^3, c_5^1, c_7^4, \}$

V. Android Implementation

For technical validation purposes, we implemented the proposed approach over Android devices, using the Bluetooth technology for local information gathering and device-to-device applications download.

System set-up

An Android application has to be installed in the terminals. When the application is first started, it downloads from a web repository the list of all deployed Apps. For each application the repository delivers the application name, category and creator. The data are exchanged according to the JSON format using the following array notation: [[”app1 name”,”app1 category”,”app2 creator”], [”app2 name”,”app2 category”, ”app2 creator”], ...]. The download of the App list can be postponed if the terminal is connected to the cellular network waiting for a wifi connection.

We have constructed the web repository by crawling, once for all, the Android Market. Since, to the best of our knowledge, the Market does not have any defined interface for accessing the information about the stored Apps, we designed a script in the server that scans the html pages listing the Apps stored on the website https://market.android.com/, and extracts the needed parameters.

Bluetooth Information Exchange

The information exchange among terminals is operated by the Bluetooth technology. The selection of Bluetooth is due mainly for its energy efficiency with respect to the WiFi: using ”continuously” the WiFi consumes the battery in few hours, while Bluetooth provides around 8 to 10 hours of battery life [17].

The local information gathering procedure is managed by a background service activated at boot time. It operates a Bluetooth discovery periodically. When a device is discovered, the service controls if the device has already been present in the local database. If it is not, a Bluetooth ”Insecure” RFCOMM connection is initiated to perform the data exchange. This kind of connection allows the creation of P2P links without the manual intervention of the user. Clearly multiple devices can be discovered: the application will try to connect all of them, sequentially. The data exchanged over the air are the bloom filters that are transferred as a stream of byte through the created RFCOMM connection.

The background service activated at the boot saves all the received Bloom filters and the Bluetooth addresses of the device in a list. Each element in the list has an insertion time: if the data is older than 10 minutes we consider the element expired and the Bloom filter has to be acquired again. But if during a discovery the device is detected we refresh the insertion time as if the node would have sent a new Bloom filter. If a node change its Bloom filter it has to send it again to all neighbor devices.

We write in the bloom filter the names (in lower case) of the Apps, retrieved by the Android PackageManager. The applications that are installed in the system folder are excluded from this operation.

Bluetooth limitations

The Bluetooth technology has some intrinsic and Android specific limitations.

The Bluetooth discovery phases is very long and there is a probability of not discovering a device dependent mainly on the active connections and the scan duration both in the searching and searched device. Moreover the simultaneous searching/discovery mode activated in a device can even sharpen this situation. This limitation is intrinsic in the Bluetooth technology and cannot be easily solved.

Moreover the Android system introduces a security features limiting the duration of the discovery states in the devices. After 300 seconds the device asks for a manual intervention of the user for remaining in the discoverable state. The only solution we found is the insertion at the system level of a process that keeps the device in discoverable mode. However this is not a simple modification because the root account in the device has to be reactivated.

Apps search

The search for the applications is performed by the open source project ”android-market-api”. This project provides a convenient way to access the Android market retrieving all the relevant information of the Apps. However the number of results for each query is limited to 10. Inspecting the project source code we noted that the ”android-market-api” interrogates the market with a standard HTTP query and parses the returned page to grab the results. The main difference with the approach of our server is related to the authentication: in fact the ”android-market-api” provides the market with the Google account bind to the device.

Further open issues

A number of issues are still open in our implementation. By far, the most critical one is that of security: a neighboring device involved in a local download may deliver a different application than that requested, and the usage of Bluetooth without coupling is prone to man in the middle attacks. Integrity verification for Apps thus appears advisable.

Another major issue is to migrate from Bluetooth to WiFi technology or other peer to peer communication solutions. The specific usage of WiFi requires to properly address energy consumption issues, for instance via suitably synchronized periodically executed local information gathering procedures, instead of persistent ones.

VI. Conclusions

In this paper, using as proof-of-concept use-case the download of Apps from the Android Market, we have introduced CarpeDroid, an approach for *opportunistically recommending* content stored in the network neighbor to an end user. CarpeDroid does not require any centralized entity, and is lightweight and not invasive. End user devices limit their interaction to the asynchronous exchange of the list of stored Apps, suitably organized in the form of Bloom filters for better than nothing privacy protection, and the recommending algorithm is purely based on a suitable integration of information locally stored in the end user device, and information returned by the (legacy) Android Market upon a standard end user request.

We believe that the CarpeDroid ideas preliminarily presented here for the specific Apps download use-case may fit well in other more general scenarios, involving distributed recommendation of non-real time content that can be cached in the terminals such as video, music, photo, newspaper, etc, and relevant access through a two-step look-up and select interface.

The main issue left open in this work is the extent to which CarpeDroid may improve network efficiency and help in offloading. This is a very complex task, left as our future challenge, as we believe that a convincing assessment may hardly get rid of a large scale experimentation of real-world end users (to the best of our knowledge, no simple but realistic models are available to simulate how users choose among content alternatives).

References

[1] Cisco Visual Networking Index Whitepaper: Global Mobile Data Traffic Forecast Update, 2010-2015, february 1, 2011, available online: http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.pdf

[2] A. Balasubramanian, R. Mahajan, A. Venkataramani, “Augmenting mobile 3G using Wi-Fi”, ACM MobiSys 2010, Jun. 2010, pp. 209-222.

[3] A. Handa, “Mobile Data Offload for 3G Networks”, White Paper, IntelliNet Technologies, Oct. 2009, available online: http://www.intellinet-tech.com/Media/PagePDF/Data Offload.pdf

[4] K. Lee, I. Rhee, J. Lee, Y. Yi, S. Chong, “Mobile data offloading: how much can WiFi deliver?, Poster, ACM SIGCOMM ’10. August 2010.

[5] V. Chandrasekhar, “Femtocell Networks: A Survey”, IEEE Communications Magazine, 46(9):59-67, Sept. 2008.

[6] R. Bhatia, G. Narlikar, I. Rimac, A. Beck, “Unap: User centric network-aware push for mobile content delivery”, IEEE INFOCOM 2009, April 2009, pp. 2034-2042.

[7] J. Su, J. Scott, P. Hui, J. Crowcroft, E. De Lara, C. Diot, A. Goel, M. H. Lim, E. Upton, “Haggle: seamless networking for mobile applications”, 9th int. conf. on Ubiquitous computing (UbiComp ’07), pp. 391-408.

[8] A. K. Pietiläinen, E. Oliver, J. LeBrun, G. Varghese, C. Diot, “MobiClique: middleware for mobile social networking”, 2nd ACM workshop on Online social networks (WOSN ’09), Barcelona, 2009, pp. 49-54.

[9] C. Boldrini, M. Conti, A. Passarella, “Design and performance evaluation of ContentPlace, a social-aware data dissemination system for opportunistic networks”, Elsevier Comp. Networks, 54(4), Mar 2010, pp. 589-604

[10] B. Han, P. Hui, V. S. Anil Kumar, M. V. Marathe, G. Pei, A. Srinivasan, “Cellular traffic offloading through Opportunistic communications, a case study”, 5th ACM workshop on Challenged networks (CHANTS ’10).

[11] B. Han, P. Hui, A. Srinivasan, “Mobile data offloading in metropolitan area networks”, SIGMOBILE Mob. Comput. Commun. Rev., vol. 14, pp. 28-30, November 2010.

[12] J. Whitbeck, Y. Lopez, J. Leguay, V. Conan, M. Dias de Amorim, Relieving the Wireless Infrastructure: When Opportunistic Networks Meet Guaranteed Delays, IEEE WoWMoM 2011, June 20-23, Lucca, IT, 2011

[13] G. Bianchi, S. Giordano, “Challenge: Network-aware Human Traffic adaptation”, 8th int. conf. on Wireless On-Demand Network Systems and Services (WONS ’11), 26-28 Jan. 2011, pp. 132-133.

[14] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Commun. ACM*, vol. 13, pp. 422–426, July 1970.

[15] A. Broder and M. Mitzenmacher, “Network applications of bloom filters: A survey,” in *Internet Mathematics*, 2002, pp. 636–646.

[16] A. Boldyreva, “Search on Encrypted Data in the Symmetric-Key Setting”, Selected Areas in Cryptography, Springer, LNCS, vol. 6544, 2011.

[17] R. Friedman, A. Kogan, Y. Krivolapov, “On Power and Throughput Tradeoffs of WiFi and Bluetooth in Smartphones”, 30th IEEE International Conference on Computer Communications (INFOCOM 2011).