Caching Structures for Distributed Data Management in P2P-based Social Networks

Newton Masinde, Moritz Kanzler, Kalman Graffi
Email: firstname.lastname@hhu.de
University of Düsseldorf, HHU

***Abstract*—Distributed applications require novel solutions to tackle problems that arise due to the scarcity of resources such as bandwidth, memory and processing power. One of these challenges is seen in distributed data management. The challenge is the two part problem of ensuring that the content is valid when accessed and updating it immediately when changed. This is especially difficult when considering p2p-based distributed online social networks, which aim to build reliable, secure social networking platforms on top of often unreliable and insecure devices. In this paper, we propose three selection strategies, random, trend and social score, for a social caching mechanism. They consider the social interaction patterns in the social network. We implement and evaluate them in a DHT-based distributed online social networks called LibreSocial and show that the social score is the best strategy. Further we implement the social caching solution and also show that when used in combination with the existing caching solution almost all requests can be serviced via cache while retaining the consistency of data during updates.**

I. Introduction

Online social networks have revolutionized how users communicate with each other and share content as well as information. As a consequence, there has been a rapid growth in the amount of information that is available online, as well as the volume of content that users upload into the various social media platforms. The obvious downside to this phenomenal increase in shared content and the connections initiated is seen in the growing concerns raised regarding the privacy of personal information and the ownership of uploaded content as well as the high costs of scaling the network. These concerns are a direct result of the use of centrally managed systems by the online social network (OSN) system providers. The system providers essentially own all the content uploaded to their servers, including private information and in many cases utilize this information to generate an income by selling it to third parties. As a solution to these concerns, distributed online social networks (DOSNs) have been proposed. A distributed online social network is designed with the goal of ensuring that the OSN operates in a distributed fashion with minimal or no central control. This design offers three key benefits over the centralized OSNs: *a*) general reduction on the operational costs as most resources are provided by the users; *b*) possibility for implementing better user-oriented privacy control; and *c*) innovative development [1] as more resources, such as communication and storage, are availed via the users.

The gains realized in offering solution to mitigate the privacy and scaling concerns of centralized OSNs by the DOSNs are met by challenges in the actual design and deployment of the DOSNs. One of these challenges is the management of social content or data in a distributed environment [2] which can also be viewed as the problem of *scalable dissemination of social updates* [3]. "Social data" or "social update" is basically all the data that is exchanged between the users (profile information, relationships, community memberships and so on) and the generated content (comments, messages, posts, images and so on) [2], [3]. While a content delivery network (CDN) is used in a centralized network to help reduce the load on the network, the selection of a good social content delivery strategy can effectively achieve load reduction on the content storing nodes. One method of handling the dissemination of this social data is the use of social replication/caching of the data. Social caching differs from distributed caching [4] and dynamic data replication [5] on several points [6]: social network topology shows non-trivial clustering and positive degree correlation [7], social cache selection is social-relationship driven, and social caches only cache updates for friends to ensure a predictive pre-loading and one-hop communication.

In this paper, we propose a social caching mechanism for a P2P-based DOSN application called *LibreSocial* (previously called LifeSocial.KOM [8]–[11]) with the aim of reducing the load on the content storing nodes. We propose three caching selection strategies called *random algorithm*, *trend algorithm* and *social-score algorithm*, and compare the performance of each of these selection strategies. The social-score algorithm is designed to utilize the social graph while taking into account the various centrality measures within a nodes ego network. We show that the social-score selection strategy outperforms the others based on the cache hit ratio. Further, we evaluate the overall performance of the network taking into account four instances, that is, no cache, default cache, social cache and social cache plus default cache. The results show that the network gives the best values when the default and social cache are enabled together.

II. Related work

There have been several ways in which DOSNs have been classified such as based on the decentralized data model [12], privacy and security considerations [13]–[15], and design decisions such as storage, access control and interaction [16]. These classifications, though comprehensive do not provide insightful information about how data management in terms of handling social data is performed in DOSNs. Probably the best comprehensive study on social data management in DOSN is given in [2] in which DOSNs are classified into three groups, namely *DHT-based*, *social overlay (SO) based* and *external resource-based* DOSNs.

DHT-based DOSNs, such as DECENT [17], Cachet [18], and LibreSocial [8]–[11], are characterized by an overlay that relies on a distributed hash table (DHT), which is built through all participating peers and provides a key-value storage as well as ID-based routing. The DHT usually is used for storing the social content and also offers indexing services. On the other hand, for *social overlay (SO) based DOSNs*, the social overlay is a network of logical connection between pairs of nodes that correspond to their friendship relations, and they also exploit the DHT for indexing services. An examples of a SO-based DOSN is DiDuSoNet [19]. Finally, external resource-based DOSNs like Diaspora (https://diasporafoundation.org) and Vis-á-vis [20], are set up through a federation of private web servers and thus are a compromise between centralized servers and completely decentralized solutions.

The appropriate solutions that help in ensuring reliable decentralized data management services for DOSNs are a great necessity that must always be kept in mind. However, it requires handling three challenges associated with are *data availability/persistence*,

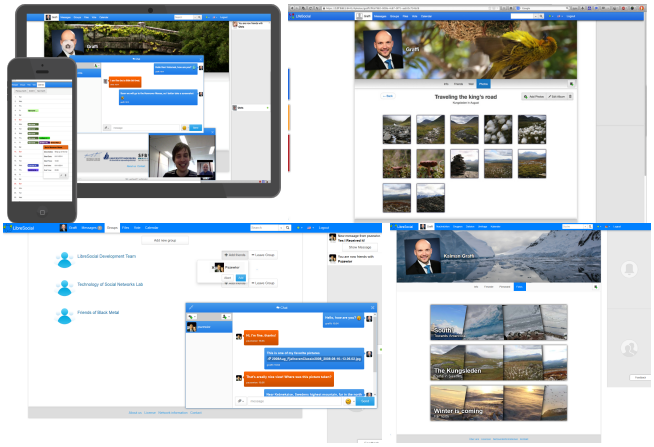


Fig. 1. Screenshots of LibreSocial

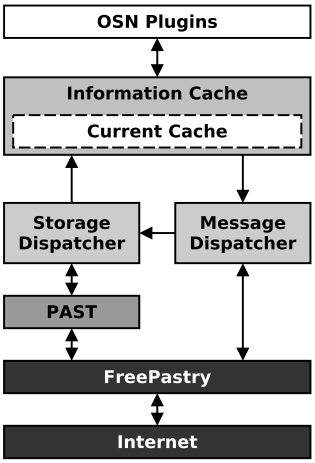


Fig. 2. LibreSocial's architecture

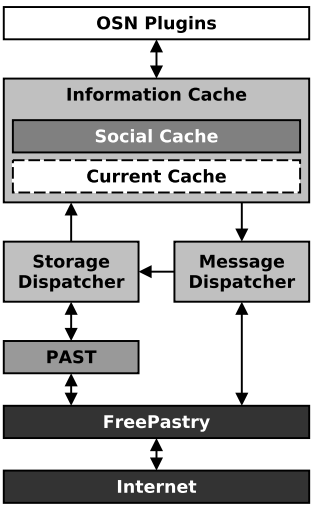


Fig. 3. With social cache extension

information diffusion and *privacy* [2]. In the context of this work , we discuss the first two points in detail.

A. Data availability

This challenge entails the persistence of publicly accessible data in a distributed environment for which several techniques such as distributed dynamic data replication [5], erasure codes [21] and caching [4] are proposed. However these traditional approaches do not work well for DOSNs. Thus, suitable solutions in DOSNs can be divided based on two considerations, *external resource-based strategies* and *replica selection strategies* [2].

In external resource-based DOSNs, the data availability is guaranteed by utilizing some external resource management service such as cloud-based storage services, or even hybrid solutions that entail distributed storage augmented with cloud services. Cadros [22] is an example of a DOSN that uses cloud assisted data replication. Replication requires the design of a replica selection strategy to guarantee availability and the strategies can be separated into two general groups, *replication in the DHT (untrusted nodes)* and *replication on trusted nodes*. In *replication in the DHT (untrusted nodes)*, replication of stored content is handled by the underlying DHT overlay, therefore the social network has no control whatsoever of the replica placement. In LibreSocial the content stored in the DHT is encrypted and a sophisticated access control scheme is applied, thus the data storage nodes do not have to be trusted. On the other hand, *replication on trusted nodes* does not depend on the overlay replication mechanisms. Replica placement is controlled by the social network, where each node selects a suitable nearby replica node based on some calculated measurements. Two key aspects must be accounted for in this replication strategy. The first aspect is that nodes can go offline at any time which can affect persistent storage. This can be handled by analysis of online behavior of the nodes. The second aspect is how to gauge the trust between nodes so that data can be exchanged. This can be solved by inclusion of encryption as well as determination of social strengths that peers have with one another based on factors such as number of common friends or number of actual interactions between each other.

B. Information diffusion

The second challenge to consider is information diffusion. This covers the possibilities available in efficient access of needed information. There, costs have to be taken into account in term of factors such as network bandwidth, response times and so on, while ensuring data consistency. In the case of DOSNs, this challenge relates to the update dissemination process wherein updates to a user's data are made available to all interested nodes [2]. The update dissemination process can fall into three classes, *request-reply*, *active dissemination* and *hybrid* approaches

Request-reply approaches are based on the notion that, if data availability is guaranteed, all available content can be accessed by simply making a request for it. The node responsible for the content will then reply by availing the requested content to the requester and in case the owner of the content is offline, replica nodes will give a response. This approach is seen in DOSNs such as PeerSoN, DECENT, SocialCDN [3], Vegas and LibreSocial. *Active dissemination approaches* may be viewed as a form of network flooding common in unstructured P2P networks but the distribution of the updates is limited using mechanisms such as gossip protocols like rumor mongering and anti-entropy [23], or based measures such as Weighted Ego Betweenness Centrality (WEBC) [24], social interaction behavior [25] and so on, to ascertain that only the nodes that require the update receive it. An examples of a DOSN that use this approach is DiDuSoNet. The last class, *hybrid approaches*, the DOSNs implement a combination of request-reply and active dissemination approaches. This is seen in Cachet [18].

III. Data Management in LibreSocial

LibreSocial, previously LifeSocial.KOM [8]–[11], is a DOSN designed based on a P2P framework model, hence relies on a DHT for its underlying functionality. Screenshots of LibreSocial are presented in 1. LibreSocial is designed using (a heavily modified) FreePastry (http://www.freepastry.org/FreePastry), an open source implementation of Pastry [26] which includes PAST [27], a persistent storage utility that manages data replication. PAST relies on FreePastry for routing, and hence LibreSocial performs replication within the structure of the DHT. LibreSocial's architecture has been previously discussed. We focus here primarily on how LibreSocial achieves data management, and specifically its storage structure.

A. Storage structure of LibreSocial

In Fig. 2, LibreSocial's data management structure is shown. In addition to the use of (a heavily secured and extended) PAST for persistence storage, three other components are included for efficient data storage, retrieval and dissemination: *Storage Dispatcher*, *Message Dispatcher*, and *Information Cache*. The *Storage Dispatcher*

provides an interface to interact with the data stored in the overlay network via PAST. It provides additional functionality not previously available in PAST, in particular, the abilities to update data, to remove data and all security elements. The *Message Dispatcher* is useful in handling system messages and specific events triggered by users such as sending, receiving and replying to friend requests. Therefore messages sent using the Message Dispatcher must entail a receiving user account along with a payload. When a message arrives via the Message Dispatcher, the user receives a notification. In case the user is offline, this message is stored persistently in the network.

The *Information Cache* provides a caching mechanisms for LibreSocial in addition to PAST's internal caching mechanism. It is designed on top of both the Storage Dispatcher and the Message Dispatcher, building a single layer responsible for data and message access and dissemination for the OSN application. Messages are accessed via a hash map in accordance to the OSN plugins that the messages are intended for or coming from. It hides the delays and complexities due to the asynchronous nature of the system by providing the needed content if available in cache and requesting it otherwise. The data management process for a node is as follows:

- *First time request from plugins, cache miss*: The Information Cache sends a request to the Storage Dispatcher which performs an overlay lookup. Once retrieved, the content is sent to the Information Cache which stores it in the local cache storage, before forwarding it to the requesting plugin. Cached content is valid for a fixed amount of time.
- *Consequent requests, cache hit/miss*: A lookup is performed in the local cache. If available and valid, it is sent to the requesting plugin. If unavailable or not valid, a request is sent to the overlay. The received content is updated to the local cache before being delivered it to the requesting plugin.
- *Cache management*: In case the local cache has reached its maximum limit, some of the cache content is deleted based on a "Least Recently Used" (LRU) strategy.
- *Adding of new content*: When new content is added by a plugin, the Information Cache first adds it to the local cache before forwarding it to PAST for persistent storage in the network.

LibreSocial thus utilizes replication in the DHT for data availability and the request-reply approach for information diffusion. The request-reply approach is efficient for DHT-based DOSNs but can result in high network traffic due to recurring overlay lookups (for the same content). Also, in view of the Information Cache freshness, the value set for cache freshness can cause regular overlay lookups if selected too short. Thus to tackle this, we propose introducing a social cache that employs update dissemination that ensures (relevant) cache data is synchronized after it is updated. In the next section, we review two DOSNs, Cachet and DiDuSoNet, to gather insight into a possible social caching solution.

B. Cachet vs DiDuSoNet

Cachet [18] performs replication in the DHT and uses both reply-request and active dissemination approaches for information diffusion. Cachet implements social caching using a *pull-push* based gossip algorithm in which a new joining node pulls content from online nodes while a node that generates an update pushes it to other online nodes. Attribute-based encryption is used to prevent unauthorized data exchanges from the caches when pulling content. However, in Cachet, the social contact selection algorithm tends to select all online users in a continuous fashion rather than dynamically selecting a few contacts. This can be achieved easily by collecting information on node interactions by analyzing the social graph structure formed by the social connections but this is not possible in DHT-based DOSNs such as Cachet due to the network topology.

In [24], an epidemic protocol for spreading social updates is suggested, with the links between nodes based on the social interactions. The protocol uses an egocentric social measure to approximate Betweenness Centrality called the *Weighted Ego Betweenness Centrality (WEBC)* that exploits a weighted graph with the weights referring to the tie strengths between the nodes. Thus nodes with higher interactions have a higher value of WEBC. This concept is used in DiDuSoNet. DiDuSoNet [19] is a SO-based DOSN that does replication on trusted nodes with the DHT providing indexing services, while relying on request-reply for information diffusion. Unlike Cachet, DiDuSoNet does not include an advanced caching mechanism but describes a very effective trusted nodes selection mechanisms that makes use of Dunbar-numbers [28], [29] to build an ego network for each user (*ego*) in the network, in which a an ego's network is composed of the trusted nodes. Dunbar's research on real-life social interactions pointed to the ability of a single person actively maintaining an average of 150 social relationships, referred to as *alters*, at a given time, which is referred to as the *Dunbar number*.

Further, [30] show that these 150 alters can be divided based on level such that each succeeding layers beginning from the ego itself includes the preceding layer, hence 5, 15, 50 and 150. Using social interaction analysis based on the epidemic protocol in [24] to generate the tie strengths, a choice of alters to store the replicas can be made. This way the selection of friends to replicate content done.

IV. A SOCIAL CACHING MECHANISM FOR LIBRESOCIAL

The fundamental structure of LibreSocial's design based on a DHT overlay ensures reliability, scalability and fault-tolerance. Although LibreSocial assumes no trust among the nodes, not even among "friends", it implements key security features that can allow us to develop a social cache structure within the framework, which contains content from nodes that are considered friends hence maintain privacy. Fig. 3 show the proposed social cache integrated into the Information Cache. This placement is informed by the fact that the Information Cache interacts directly with the OSN plugins and therefore would provide a platform for the social cache. We now have to define the social updates, handle lookup requests and selection of suitable users to subscribe to based on social interaction.

A. Social updates

In LibreSocial the cached data is not actively re-synchronized with the original data, hence cached data consistency is not guaranteed but the maximum age of the data is limited. By use of *social updates* we can actively disseminate changed content to selected users for the case. Once they would request the content (as predicted) it will be already available locally. Performing social updates is characterized by two aspects: *detection and distribution of content changes*, and *receiving and storing the updates*. We consider each aspect carefully.

1) Detection & distribution of content changes: The social update includes content modified by its originator disseminated via a messaging channel, called an *instance channel*, to a defined group of users listening to that channel. Each social caching instance maintains a list of selected users subscribed to the upcoming content modifications of this instance called *receiver list*. When a user performs any content modifications that updates the data in the DHT, the social cache also transmits a social update for each of the subscribed users stored in its receiver list. This process is illustrated in Fig. 4. This guarantees that users subscribed to upcoming social updates of a particular user's instance channel receive content updates.

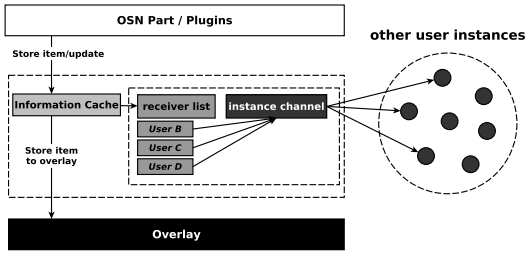


Fig. 4. Distribution of social updates

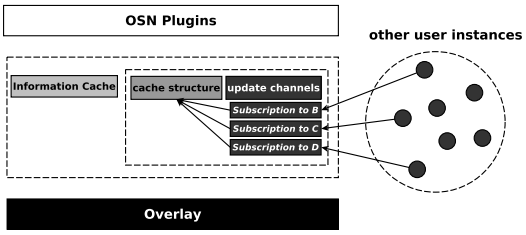


Fig. 5. Receiving updates from subscribed channels

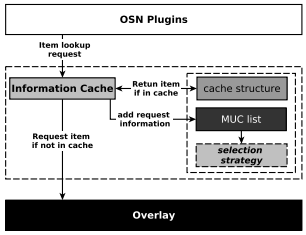


Fig. 6. Lookup request using the social cache

Algorithm 1: Information Cache Lookup

Input : *storageKey*, *MUC_ List subscriptionChannels*, *updateLimit*, *selectionStrategy*
Output: Subscription list based on selection strategy
username ← `getUsername(`*storageKey*`)`; **if** selectionStrategy == *random* **then**
  if username ∈ MUC_List **then**
    add username to MUC_List
    `RandomSelectionStrategy(`username`)`
  endif
endif
else if selectionStrategy == *trend* **then**
  if username ∉ MUC_List **then**
    numOfChannels ← `sizeOfList(`subscriptionChannels`)`
    if *numOfChannels* < updateLimit **then**
      `subscribe2Updates(`username`)`;
    endif
    else
      `TrendSelectionStrategy(`username`)`
    endif
  endif
endif
else if selectionStrategy == *socialScore* **then**
  if username ∈ MUC_List **then**
    numOfChannels ← `sizeOfList(`subscriptionChannels`)`
    if *numOfChannels* < updateLimit **then**
      `subscribe2Updates(`username`)`;
    endif
    else
      `SocialScoreSelectionStrategy(`username`)`
    endif
  endif
endif

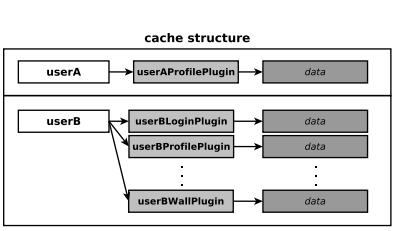


Fig. 7. Social cache with two nested abstraction levels

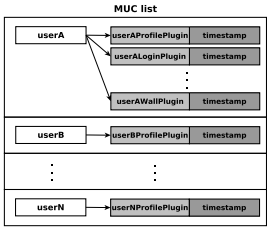


Fig. 8. MUC list structure

Algorithm 2: Random Selection Strategy

Input : *userName*, *subscriptionChannels*, *updateLimit*
Output: Randomly selected replication list
numOfChannels ← `sizeOfList(`subscriptionChannels`)`
if *numOfChannels* < updateLimit **then**
  `subscribe2Updates` (userName)
else
  randomIndex ← `getRandomNum(`*0, numOfChannels-1*`)`
  `unsubscribe2Updates` (subscriptionChannels[*randomIndex*])
  `subscribe2Updates(`userName`)`;

2) Receiving & storing update: Each social caching instance handles its own subscriptions to other users' instance channels. This is shown in Fig. 5. The social cache listens to the instance channels for updates and stores them into a data structure called *update channels*. Every time a social update is populated through one of this update channels, the social cache receives the stored information from the update message and stores it in its own caching data structure. The caching structure in the social cache is organized in two-layers and is shown in figure 7. For each subscribed user, received information is stored based on its *StorageKey* and every social update contains information regarding the subscribed user and the particular StorageKey. Previous information is overwritten by a newly incoming social update. By this, availability and consistency for all stored items in the social cache is assured.

The next issue of concern are now the extensions to the Information Cache to support the lookup process using the received social updates instead of performing overlay requests. This is considered next.

B. Handling lookup requests

The social cache is integrated to work with the lookup process within the Information Cache. Lookup requests contain the username of the information requester as well as the StorageKey of the needed information. Upon receipt of a request from an OSN plugin, the Information Caches first tries to locate the content in the caching structure of the social cache by matching cached data against the StorageKey. If available, the content is sent to the requesting plugin. If the requested content is not found in the social cache, then a request is issued to the DHT. This process is shown in Fig. 6.

C. Selecting suitable users based on social interaction

The next major step is the selection strategy for subscriptions. The *selection strategy* is the method used to filter and select a set of suitable users to which an instance then subscribes to so as to receive social updates. The suitable subscriptions should ideally be those that will be accessed frequently in the future. The only way to achieve a high ratio of cache hits for the social caching mechanism is to ensure that data accessed in future mainly belongs to users that are currently subscribed to. Therefore, the selection of those users is a crucial component of the social caching mechanism.

To make an informed decision on subscriptions to make, the social cache monitors the interactions an instance makes with other users. We further make the restriction that the social cache does not monitor the application plugins but only keeps track of all incoming requests received by the Information Cache from the OSN plugins without consideration of the source of the final response to the request. For every request, details on the user making the request, the type of request and the timestamp of the request are stored. This information is stored in a separate data structure called *most-used-contacts* (MUC) list, from which suitable subscriptions are to be made. In Fig. 8 a representation of such a MUC list is shown. This data is independent from the actual social behavior of the users as viewed from the social network, but it nevertheless offers information about how frequently and in which way the own instance is interacting with other users.

The MUC list is the basis for the selection strategy that is used to choose the most suitable users to subscribe to. The selection process is triggered in fixed intervals called *update intervals*. Let m be a predefined maximum number of tracked lookup requests and n the number of parallel maintainable update channels with $n < m$. After m lookups, n candidates are chosen who are deemed most suitable to subscribe to during the next update interval. We propose three

Algorithm 3: Trend Selection Strategy

```
Input : username, subscriptionChannels, updateLimit
Output: Replication list based on trends
trendArray ← ∅
foreach user ∈ MUC_List do
    trendArray ← number of requests by user
end
sortByDescending(trendArray)
newSubscriptions ← ∅
i ← 0
while i < updateLimit do
    newSubscriptions ← trendArray[i]
    i = i + 1
end
oldSubscriptions ← ∅
oldSubscriptions ← subscriptionChannels
foreach user ∈ (newSubscriptions ∩¬ oldSubscriptions ) do
    subscribe2Updates(user)
end
foreach user ∈ (¬ newSubscriptions ∩ oldSubscriptions ) do
    unsubscribe2Updates(user)
end
MUC_List ← ∅
```

Algorithm 4: Social Score Selection Strategy

```
Input : username, subscriptionChannels, updateLimit
Output: Replication list based on trends
socialScoreArray ← ∅
foreach user ∈ MUC_List do
    socialScore ← calculateSocialScore(user)
    socialScoreArray ← socialScore
endfch
sortByDescending(socialScoreArray)
newSubscriptions ← ∅
i ← 0
while i < updateLimit do
    newSubscriptions ← socialScoreArray[i]
    i = i + 1
end
oldSubscriptions ← ∅
oldSubscriptions ← subscriptionChannels
foreach user ∈ (newSubscriptions ∩¬ oldSubscriptions ) do
    subscribe2Updates(user)
endfch
foreach user ∈ (¬ newSubscriptions ∩ oldSubscriptions ) do
    unsubscribe2Updates(user)
endfch
```

selection strategies for these n candidates, namely *random*, *trend*, and *social score* selection strategy which are further described. The general Information Cache lookup is shown in Algorithm 1.

1) Random selection strategy: The random selection strategy is shown in Algorithm 2. In this strategy, the user subscribes to every new user who is tracked in lookup process until the limit of n candidates to subscribe to is reached. If that limit is reached and a new user is discovered through the MUC list process, a currently subscribed user is randomly chosen to unsubscribe from and the new subscription is added. In this selection strategy the MUC list is only used to monitor known and unknown user. The stored lookup information does not affect the selection's decision in any way. An unsubscription of a user also leads to the removal of this user from the MUC list. Therefore the number of stored users in the MUC List is the same as the currently maintained parallel subscriptions.

2) Trend selection strategy: The trend selection strategy is given in Algorithm 3. It utilizes the MUC list in a different way from the random selection strategy. Once the required m lookups are achieved, the tracked users stored in the MUC list are ranked based on the number of performed lookups from highest to lowest. Then the first n users are chosen, which are assumed to have the most interaction with the user over the current lookup interval. During the next lookup interval, subscriptions are then sent to these n users as long as there are no existing subscriptions. In case of existing subscriptions, subscribed users not in the generated ranking are unsubscribed from. Thereafter, the MUC List is cleared for the new lookup interval.

3) Social score selection strategy: The strategy is shown in Algorithm 4. It follows the idea of using a *SocialScore* based on Dunbar's [28] social interaction model, as we sketched for DiDuSoNet [19]. While DiDuSoNet is based on a social overlay and uses a mixture of overlay connection and interaction information to determine trusted nodes for replication based on social closeness, we adopt the mechanism in the scope of the DHT-based nature of LibreSocial to express strong social connections between a user with other users. This strategy deviates from the trend selection strategy based on how the ranking of users in the MUC list during the subscription is done, in addition to not clearing the MUC list during a new lookup interval. When the lookup interval of m lookups is reached, for each user in the MUC list a social score is calculated. The social score for a user x is calculated as shown in Equation 1. It is based on the value of the *TieStrength* and the *MediumInteractionLength* (MIL) where α and β are weights that are defined by the users based on relevance given to each criteria.

$$\mathrm{SocialScore_{LS}(x)} = \alpha * \mathrm{TieStrength(x)} + \beta * \mathrm{MIL(x)} \tag{1}$$

The *TieStrength* as defined by [19] is a numerical value that helps to define the relationship between two users and can be calculating by considering factors such as frequency of contact between users, number of likes, posts, tags, comments and so on [30]. It is obtained by summing all tracked interactions for a user x divided by the total amount of all monitored interactions I so as to normalize it based on the total number of interactions by user x.

$$\mathrm{TieStrength(x)} = \frac{\sum_{i=1}^{n \in I(x)} \mathrm{weight(interaction_i)}}{|I|} \tag{2}$$

The metric MIL, shown in Equation 3, is the average duration of interactions between two users and is introduced in place of the ranking of users done for the trend selection strategy. It is calculated by taking the medium time span between all tracked interactions for a user x and in relation to the time period between the first recorded interaction and the current timestamp available for user x at the time of the calculation.

$$\mathrm{MIL(x)} = \frac{\sum_{i=1}^{n} \left[\frac{(T_i(x) - T_{i-1}(x))}{n-1} \right]}{T_{\mathrm{now}} - T_{0(\mathrm{x})}} \tag{3}$$

By combining the values of *TieStrength* and *MIL* it is now possible to obtain a relation between the number of interactions against length of interactions for every monitored user. Hence, the $SocialScore_{LS}$ ranks a user with more frequent interactions over a longer period of time higher, than a user with fewer interactions and shorter period. This mechanism ensures consistent cache entries, takes into account the social component of an OSN, dynamically adopts to changes of users preferences, and is fully working without undermining the existing processes of the given application.

Nevertheless, there are some special cases in which the social caching mechanism has conceptual weaknesses or limits. Those weaknesses and limits need to be considered and can be solved by additional processes which are discussed in the following.

D. Addressing constraints in social caching

With the combination of the proposed concepts of social updates, the tracking of interactions by the MUC list, and the selection of subscriptions by selection strategies, we have created a caching mechanism, which uses active dissemination of newly updated information to a fixed set of users. This proactive caching is optional and allows for a quicker, as local, resolving of requests at the LibreSocial instances of friends. However, there are some cases in which the social caching mechanism presents conceptual weaknesses or limits which need to be addressed via additional processes which we highlight.

1) Recurrent overlay requests for own content: Sometimes, lookups may not result in cache hits thus leading to overlay requests, negating the use of the social cache. This can occur when a user does not have his/her friends' content in the social cache, but may need to also access this content often and may therefore result in recurrent overlay requests. As changes to the content are initiated by the OSN plugins which are on top of the social caching process, we can easily store them in an additional data structure and serve the requests from the social cache instead. With this data structure we do not undermine the consistency of the stored information but reduce the needed overlay requests by a significant amount.

2) Social bootstrapping: When one of the current subscription hardly make changes to content, there will be no social updates received from them. Since only social updates lead to cache entries, requests for this user will always cause recurrent overlay request. To tackle this case we introduced the concept of *social bootstrapping*. Every time a subscription between users is performed, the user receiving the subscription request sends a message to the subscriber to inform him/her to receive social updates henceforth. The user who is now subscribed to then responds with all of his/her own cached content which is inserted into the caching structure of the subscribing user. The consistency of cached content is not affected by this process because a change in the initially sent content will lead to a social update which will overwrite the initial content.

3) Limitation of active subscription and Dunbar-number: So far it has been assumed that there is a limitation placed on the number of parallel active subscriptions without giving an actual limiting value. As the social cache is designed as additional mechanism in a given DHT-based overlay, the number of connections between users needs to be restricted to prevent establishment of a second overlay. This limiting value is based on the *Dunbar number*. Because the maximum amount of maintainable contacts is at about 150 according to Dunbar [28], we also restrict the MUC list to store information for a maximum of 150 distinct users. When this limit is reached, the users with the least rank according to the selection strategy is removed to create space for newly monitored users.

V. Experimental Setup

Evaluation of the social caching mechanism is the process of benchmarking the overall performance of LibreSocial in the face of the new mechanism. A benchmark is in general a tuple of *quality attributes* (or *properties*) Q, *metrics* M, and test scenarios S [31]. In the discussion that follows, we describe each of these in view of the social caching mechanism evaluation. Thereafter, the results are presented and discussed.

A. Quality properties and metrics

While the metrics focus on only one attribute of the system or mechanisms within the test scenario, the quality attributes generally describe the same system or mechanism characteristics while taking into account several metric measurements. Thus, with the use of a set of quality metrics, useful statements about the general characteristics can be made which describe the overall quality of the system. For purposes of analyzing the social caching mechanism, the quality properties chosen were performance and efficiency. This are further considered.

1) Performance: This characterizes the system in terms of how it responds, its throughput and the validity of the results as a consequence of a particular workload within the bounds of the test environment. In the case of the proposed social caching mechanism, the overall objective is a reduction in overlay requests and an increase in cache-based responses to the content requested. Thus by collecting the overall requests against the answered requests from the cache and from the overlay, we can calculate the metric *cache hit ratio* as shown in Equation 4, which helps in quantifying the performance. The higher the value of the cache hit ratio, the better the performance

$$Cache_hit_ratio = \frac{Cache_replies}{Total_replies} \tag{4}$$

2) Efficiency: This is defined as the ratio between performance and costs when considering a particular task that the system is undertaking. For the proposed social caching mechanism, we distinguish between two types, that is *local efficiency* and *overall efficiency*.

- **Local efficiency**: With this, we desire to find out how certain resources in the local environment, in this case, the local node, is utilized. As more content is to be stored locally in the cache, it is expected that more system resource will be utilized. We desire to ensure that the local resources are not used up, such as memory as well as processing power. Therefore, we monitor the number of objects cached in order to find out the required memory to achieve a desired cache hit ratio, as well as the MUC list size, number of subscription and unsubscription processes, the number of simultaneously used update channels and the bootstrap messages. Generally, smaller values observed in all these metrics in combination are an indicator of good local efficiency.
- **Overlay efficiency**: Since we are working in a distributed environment, we would like to find out the overall workload generated by the system and its effects on several aspects of the system, and in particular, the overlay. Thus we keep track of the number of messages in the overlay, the number of PAST and Pastry data operations, and average storage used to maintain overlay processes. Therefore, we observe different aspects of the LibreSocial's overlay like the number of messages passed through it, the number of Pastry/PAST data operations, and the amount of storage used to maintain the overlay processes in each node.

Having established the quality properties and their relevant metrics, we now consider the test scenario.

B. Test scenario

The test scenario constitutes the environment in which a test is being undertaken, involving all the system parameters, the workload parameters and the workload itself. Different test scenarios affect the quality of the system in different ways. To effectively evaluate the proposed social caching mechanism, we take into consideration the number of interactions and the time interval occurring between these interactions. This points to the need of a more realistic test scenario. For this, three suitable OSN data sets were considered, which provide realistic information about the number of interactions as well as time intervals between interactions. The data sets are the Facebook'09 [32], SNAP'14 [33], and Facebook'14 [19]. These data sets are also ego-based, contain information about the relations and are all based in analysis of Facebook. The information that can be gleaned from these data sets is shown in Table I.

The SNAP'14 data set only include information about the friend relations and the user profiles, and the data has no information about interactions between users nor interaction times through timestamps and hence is unsuitable. The Facebook'14 data set provides information about friend relations and different types of interactions (such as wall posts, likes, tagged photos) as well as relevant timestamp indicating time of interactions. Unfortunately due to privacy concerns, this data set could not be available at the time of testing. Therefore, Facebook'09 data was chosen, as it was available and fulfills the need

for a data set that reveals information about friend connections between users, interactions between the users as well as the timestamps for the interactions. The Facebook'09 data set is further analyzed so as to obtain relevant real world statistical information, shown in Table II, that would be useful in designing the test scenario.

From these values, we downsampled the experimental setup from 870 days a few days. The downsampling of this information enables the evaluation of the caching approaches in smaller time frames without losing the usage characteristics gathered from the data set. Our downsampling approach is shown in Equation 5

$$sampledInterval(x) = \frac{dataSetExperimentTime}{newExperimentTime * x} \quad (5)$$

The $sampledInterval(x)$ is the downsampled time period between an friend request or a wall post, where x represents the actual value calculated from the Facebook'09 data set. The $dataSetExperimentTime$ is the whole time the original experiment has run, while $newExperimentTime$ expresses the time on which the data is to be downsampled to. This way, we can select a fixed number of 25 friends while ensuring the time span between new interactions and friend request remains relative to the analyzed Facebook'09 data set. This helps attaining a similar utilization of user interactions in our test scenarios in comparison to the overall time period of the Facebook'09 experiment.

There are two test scenarios that are considered, each focusing on a different aspect.

- *Comparing selection strategies*: In this paper, we propose three cache selection strategies. We would like to find out which is the most suitable strategy considering all quality aspects. Therefore, for this test, the laboratory setup involved 64 LibreSocial instances on 8 distinct machines having similar hardware and operating system, Debian Linux. Each machine had 8 LibreSocial instances. We use a downsampled test scenario from the Facebook'09 data set which runs for 6 hours. For all tests only the social cache is enabled, but not the current caching mechanism. The test is for each of the three selection strategies, random, trend, and social score. For the parallel social update channels we choose the number of 15 simultaneously used channels, according to the Dunbar number, as well as a total MUC list size of 150 monitored users. The social bootstrapping functionality is enabled and the update interval is set to 50 seconds. A summary of the test setup can be found in Table III.
- *Comparing caching approaches*: We use the same parameters shown in Table III. However, in this case, the test scenario runs for 2 days and thus the interaction and friend request intervals were downsampled to 2 days. Four tests are performed considering the following settings, *caching completely disabled*, *current caching only enabled* (LibreSocial before social caching), *social caching only enabled* and both *current and social caching enabled*.

VI. Results and Discussion

We now consider the results obtained from the experiments conducted and analyze them for each test scenario in the following.

A. Comparing selection strategies

The selection strategies are evaluated and the results are tabulated or graphed for comparison. The performance of each of the strategies is shown in Table IV, the results for local efficiency and overall efficiency are shown in Fig. 9. We discuss them in detail.

1) Performance: From the values tabulated in Table IV, we calculate the cache hit ratio and compare the three selection strategies' performance. The cache hit ratio ranges between 94.9% and 92.4%, which are very good values, as more than 92% of the overlay lookups related to data requests have been saved. While the random selection strategy shows the highest cache ratio and the social score selection strategy has the lowest value, the values are very close.

2) Local efficiency: We now consider the local efficiency for the three selection strategies.

- *Social cache size*: This is shown in Fig. 9a. The social score selection strategy stores the smallest number of data objects within the cache structure, closely followed by the trend algorithm, with both algorithms maintaining roughly constant number of stored objects throughout the whole experiment. With the random selection strategy the social cache size keeps increasing. While the number of stored cache objects may not be directly related to the needed memory size to maintain them, we see that the random selection strategy needs a much higher amount of local resources to achieve similar performance as trend and social score. Thus the latter two approaches are much more efficient.
- *Number of subscription messages*: Fig. 9b depicts the subscriptions during the experiments. The social score strategy has the least number of subscriptions hence fewer changes in the social cache participants. This correlates very well with the cache size results indicating high efficiency as it uses fewer local resources to achieve a high performance rate as evidenced by the cache hit ratio. While the trend strategy has more subscription messages than the social score strategy, it also uses the local resources well when we consider the social cache size. The results for the random selection strategy correlate with observations for the social cache size, having a large number of ever increasing subscriptions messages.
- *Average size of MUC list*: This is depicted in Fig. 9c. The social score and random strategies have almost similar values throughout. A low MUC list number can be seen as equivalent to a smaller amount of local resources used by the social cache and therefore a better *local efficiency*. In comparison, the size of the MUC list for the trend strategy is constantly increasing contrary to expectations. This may need further investigation and a possible error may be due to the monitoring process rather than the actual MUC list structure.

Thus in view of the results gathered in analyzing the local efficiency, it can be inferred that, in general, the social score strategy is the preferred choice over trend and random selection strategies as it attains a very high performance of 92.4% cache hits against a drastically lower costs at the local node.

3) Overlay efficiency: We analyze the overall efficiency of the selection strategies by considering the following overlay aspects.

- *Overlay data load generated*: Fig. 9d shows the overall amount of data objects sent in the Pastry overlay for each selection strategy. Social score and trend selection strategies seem to have similar values of data sent until the second phase of friend requests is introduced to the system at about 257 minutes of the experiment. Then the trend strategy produces less data transactions than the social score strategy. The random selection strategy generates a higher amount of traffic due to overlay requests.
- *Average memory consumed per instance*: The results presented in Fig. 9e are helpful in understanding the memory utilization for a single instance. For each instance, it can be seen that, depending on the selection strategy, average memory utilization is between 250 and 400 MB. Social score strategy has the

TABLE I
AVAILABLE DATA SETS WITH RELEVANT FEATURES

Data set	**Year**	**Available data**			
		Timestamps	Friends	Interactions	Profile
Facebook'09	2009	✓	✓	✓	
SNAP'14	2014		✓		✓
Facebook'14	2014	✓	✓	✓	?

TABLE II
AVAILABLE FEATURES IN FACEBOOK'09 DATA SET

Total # of egos	60102
Total # of alters	1545690
Average # of alters	25.7177
Average TS/new friend request	36.7332 days
Average TS/new interaction	43.0402 days
Experiment time span	869.458 days

TABLE III
EXPERIMENTAL PARAMETERS AND CONDITIONS

Test Environment	LibreSocial instances	64
	Physical machines	8
Application settings	Social update channels	15
	MUC list size	150
	Update interval	50 sec.
	Social bootstrapping	enabled
Duration of Experiment	Selection strategies	6 hours
	Caching approaches	2 days

TABLE IV
ANSWERED REQUESTS COMPARISON BASED ON SELECTION STRATEGIES

Strategy	**Requests answered**			**Cache hit ratio**
	Cache	*Overlay*	*Total*	
Random	635663	33032	669476	0.949
Trend	387253	26987	414971	0.933
Social Score	1054618	85355	1140900	0.924

TABLE V
DISTRIBUTION OF REQUEST RESPONSES

Cache setup	**Request responses**				**Cache hit ratio**
	Cache		***Overlay***	***Total***	
	Current	*Social*			
Current	3427562		197264	3626229	0.945
Social		5170354	918365	6090445	0.849
Current +Social	786123	4606187	44299	5437792	0.992

TABLE VI
CACHE SIZES AND RESPONSES/ITEM COMPARED

Cache setup	**Items in cache**			**Response / item**
	Current	*Social*	*Total*	
Current	200674	—	200674	17.0802
Social	—	584968	584968	8.8386
Current +Social	50474	889148	939622	5.7401

least memory utilization followed very closely by the trend strategy, while the random selection strategy has the highest consumption of memory. In addition, it can be seen that the two times when friend requests occur have a considerable impact on the memory usage. The sudden sharp increases observed close to the end of the experiment is due to lookup request for unknown user leading to additional or change of subscriptions. The defined limit of parallel subscription channels therefore translates to a trade off between memory consumption and additional subscriptions for social updates.

- *Messages at the MessageDispatcher*: Focus here is on the messages produced at the overlay due to registration, updating, and bootstrapping processes of the social cache. The results can be seen in Fig. 9f in view of the MessageDispatcher messages. We see that the social score selection strategy generates the highest number of messages followed by the random strategy and finally the trend strategy. However, it is worth noting that more messages due to registration, updating and social bootstrapping is not a equivalent to a higher resource consumption, but can be an indicator for that. One reason for producing more messages may be the higher amount of total processed lookup request in the social score strategy, seen at the beginning of this evaluation.

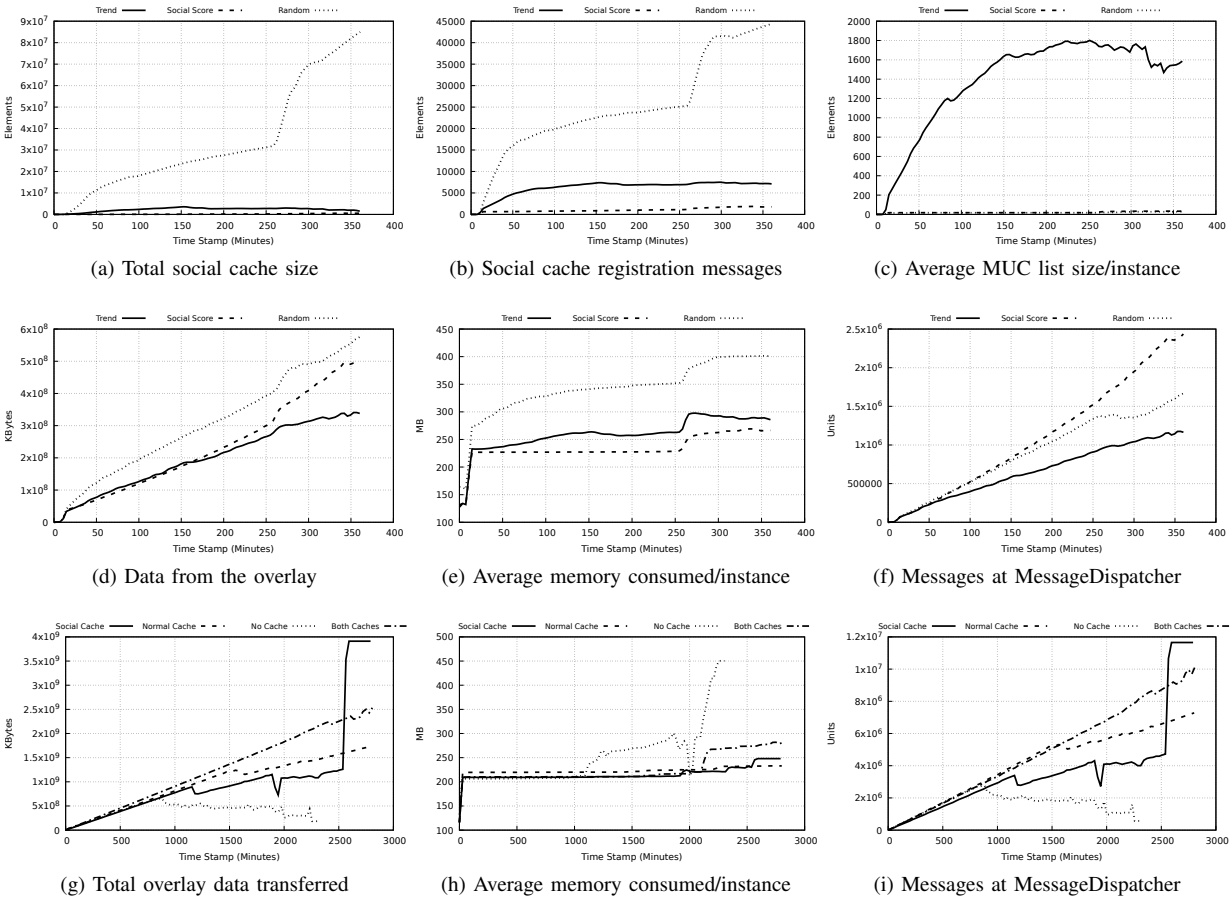
In summary, the social score again shows the lowest costs in terms of data transfer and used memory, but produces more messages over the MessageDispatcher. The trend selection strategy shows a similar pattern as the social score strategy albeit having slightly higher values for the data transfer and memory consumption but with the least messages observed a the MessageDispatcher. The random selection strategy, may be considered to have poor efficiency when considering the data transferred and the memory consumed without consideration for the messages generated.

B. Comparing caching approaches

From the results obtained in comparing the selection strategies, based on a comparison of the performance, local efficiency and overall efficiency, the most efficient selection strategy is the social score selection strategy. Therefore, in the test scenarios that we conduct to compare the caching strategies, the social score strategy is used. Before discussing the results of Figures 9g-9i, it is important to note that the tests without any cache solution generally failed quarter way during the experimental period due to internal errors generated in Pastry. Nevertheless, we include the results for comparison to show their influence on the data storage costs. We now discuss the results obtained from the cache comparison tests.

1) Performance: Once again, the cache hit ratio is used to get the effectiveness of the cache mechanism when requests were sent by an instance. The results are given in Table V for three experiments, current (i.e. Pastry internal) cache only, the (new) social cache only, and both caches in combination. With the social cache only, we reach a cache hit ratio of 84.9%, which is nearly 10% less than the current cache (94.5%). Also the total amount of performed requests is much higher with the social cache only as compared to the current cache only, with 918,365 lookups compared to 197,264 lookups. However, when the two caching solutions are used in combination, a cache hit ratio of 99.2% is observed which outperforms the current caching solution by nearly 5%. In this scenario, the current caching solution is accessed when the social cache does not answer the requests and therefore the current cache shows fewer cache hits in comparison to the social cache when in combination. The social cache in this case responds to nearly 85% of the cache answered requests. It can thus be seen that the combination of both caching approaches leads to a more efficient a solution than having the two caching solutions separately.

2) Local efficiency: We now look at the use of local resources for the three scenarios to set the performance in relation to utilization of the local system and therefore measure their efficiency. Thus we focus on the number of stored items for each of the different caching structures. In Table VI the number of objects in the cache for each test scenarios as well as the ratio of requests per item is shown. The number of items stored in the current cache alone as well as in the social cache solution alone are much fewer than when the two caches combined. Comparing the current cache alone to the social cache alone, the current cache stores less than 50% of the items in the social cache while initiating about twice as many cache responses per items (17 and 9 respectively). Thus, the current cache appears to use the local resources more efficiently by achieving a higher cache hit ratio with fewer items in cache. However, we have to contend with twice as many lookups. The combination of both caches is accompanied by a drastic increase in the cached items (almost five times as much as the current caching only) but an equally significant drop in the number of cache responses per item (thrice as much as current cache only). It is also observable, as expected, that there are much fewer items stored in the current cache as in the social cache. This is explained by the hierarchy and arrangement of the caches. The lower count of items in the current cache indicates a higher coverage of lookups responded to by the social cache. Therefore it appears that in order to achieve a high performance as indicated by the hit ratio, the trade-off is a higher storage capability as indicated by the number of items stored



(a) Total social cache size (b) Social cache registration messages (c) Average MUC list size/instance

(d) Data from the overlay (e) Average memory consumed/instance (f) Messages at MessageDispatcher

(g) Total overlay data transferred (h) Average memory consumed/instance (i) Messages at MessageDispatcher

Fig. 9. Evaluation results of the local efficiency (resource utilization) and overall system efficiency

by the combined solution local resource usage can be determined when it comes to use both caches at the same time.

3) Overlay efficiency: Finally, we analyze the impact of the caching mechanisms in terms of generated workload at the overlay.

- *Data transferred*: Fig. 9g shows the amount of data transferred and monitored in the Pastry layer of LibreSocial. The current caching solution causes more data transfer to occur at the overlay than the social caching solution. This may be due to the need to keep the data refreshed as a result of the validity time in cache for the current caching solution. There is a higher data transfer for the combined caching solution in comparison indicating higher bandwidth utilization. This is as both caches are operating simultaneously resulting in an addition on the communication payload and the needed lookup requests at the overlay.
- *Average memory consumed per instance*: This is illustrated in Fig. 9h. The current cache and social cache when used separately consume similar amount of memory. For the combined caching approach, there is a similar trend until the second group of friend requests are made at about the 2200th minute of the experiment. Thereafter, the memory usage increases slightly in comparison to the other solutions.
- *Messages at MessageDispatcher*: The results are shown in Fig. 9i. The social cache alone generates the fewest messages to the overlay and the current cache alone. It also has a higher number of messages, which indicates that not actively sharing content updates, as done by the current cache, results in the need for the system to exchange more messages with other system components. The combination of the two solutions caches has the highest amount of messages. We assume that the simultaneously use of both solutions tend result in a summing effect on resource consumption.

In summary based on the results of *overlay efficiency*, the evaluation shows that the social cache has the least resource consumption as compared to the current caching approach. However, the combined approach results in the highest overlay load. By combining both efficiency qualities with the examined performances results, this evaluation in general shows that when the social cache is used alone, it has a lower cache hit ratio hence lower performance rating, while also consuming the most local resources in form of the managed caching structure and the corresponding need of memory to store it at a single instance. However, at the overlay level the social cache shows slightly better results in the need of data and message transfer as well as memory usage.

The comparison on the current cache solution and the combination of the social cache and the current cache shows, that in terms of performance, this combination reaches by far the best cache hit ratio and can reduce the overlay request to 0.8% of the total used lookups in the Information Cache. This performance advantages comes with the trade-off of a higher load and resources consumption at the local level as well as in the overlay. It is also important to notice, that the use of the social cache ensures the consistency of the cached and used information through the social update mechanism. This consideration can not be guaranteed in the current cache mechanism.

VII. Conclusion

The use of a social caching mechanism presents several advantages but comes with several trade offs that must be put into consideration.

The use of the social interactions to actively select where to replicate data in combination with an active dissemination strategy for information diffusion makes for a good solution to ensure that the data is always up to date. We present in this paper three selection strategies, random, trend and social score, and show that the social score strategy is the best solution for active dissemination of content. Using LibreSocial as our testing platform, we have implemented a social caching mechanism on top of the existing request-reply caching mechanism and compared the performance and efficiency of the two solutions as well as the combination of the two. The combination of the two caching solutions gave the best performance as recorded by the cache hit ratio value but with a compromise on the local as well as overall resources. For the user it seems to be of higher value to have a very low delay in the user interface and having instant presentation of social network elements he desires. For that it seems reasonable to apply such a combined social and conventional caching approach leading to a higher user satisfaction. The corresponding higher load in the networking and storage utilization is of lower relevance, as long as other services run on the computer are not disturbed. Thus, our proposed approach presents a clear benefit to the user.

REFERENCES

[1] S. Buchegger and A. Datta, "A Case for P2P Infrastructure for Social Networks-Opportunities & Challenges," in *Proceedings of the International Conference on Wireless On-Demand Network Systems and Services (WONS'09)*. IEEE, Feb. 2009, pp. 161–168.

[2] B. Guidi, M. Conti, A. Passarella, and L. Ricci, "Managing social contents in Decentralized Online Social Networks: A Survey," *Online Social Networks and Media*, vol. 7, pp. 12–29, 2018.

[3] L. Han, M. Punceva, B. Nath, S. Muthukrishnan, and L. Iftode, "SocialCDN: Caching Techniques for Distributed Social Networks," in *Proceedings of the IEEE International Conference on Peer-to-Peer Computing (P2P)*, Sep. 2012, pp. 191–202.

[4] S. Borst, V. Gupta, and A. Walid, "Distributed Caching Algorithms for Content Distribution Networks," in *Proceedings of the 29th Conference on Information Communications*. IEEE, 2010, pp. 1478–1486.

[5] S. Acharya and S. B. Zdonik, "An Efficient Scheme for Dynamic Data Replication," Providence, RI, USA, Tech. Rep. CS-93-43, 1993.

[6] L. Han, B. Nath, L. Iftode, and S. Muthukrishnan, "Social Butterfly: Social Caches for Distributed Social Networks," in *Proceedings of the IEEE International Conference on Privacy, Security, Risk and Trust and IEEE International Conference on Social Computing*. IEEE, Oct. 2011, pp. 81–86.

[7] M. E. J. Newman and J. Park, "Why social networks are different from other types of networks," *Phys. Rev. E*, vol. 68, p. 036122, Sep 2003.

[8] K. Graffi, S. Podrajanski, P. Mukherjee, A. Kovacevic, and R. Steinmetz, "A Distributed Platform for Multimedia Communities," in *Proceedings of the IEEE International Symposium on Multimedia (ISM'08)*. IEEE, Jan. 2008, pp. 208–213.

[9] K. Graffi, P. Mukherjee, B. Menges, D. Hartung, A. Kovacevic, and R. Steinmetz, "Practical Security in P2P-Based Social Networks," in *Proceedings of the IEEE Conference on Local Computer Networks (LCN'09)*. IEEE, Oct. 2009, pp. 269–272.

[10] K. Graffi, C. Gross, P. Mukherjee, A. Kovacevic, and R. Steinmetz, "LifeSocial.KOM: A P2P-based Platform for Secure Online Social Networks," in *Proceedings of the International Conference on Peer-to-Peer Computing (P2P'10)*. IEEE, Aug. 2010, pp. 1–2.

[11] K. Graffi, C. Gross, D. Stingl, D. Hartung, A. Kovacevic, and R. Steinmetz, "LifeSocial. KOM: A secure and P2P-based solution for online social networks," in *Proceedings of the IEEE Consumer Communications and Networking Conference (CCNC'11)*. IEEE, Jan. 2011, pp. 554–558.

[12] A. Datta, S. Buchegger, L.-H. Vu, T. Strufe, and K. Rzadca, *Decentralized Online Social Networks*. Springer, 2010, pp. 349–378.

[13] B. Greschbach, G. Kreitz, and S. Buchegger, "The Devil is in the Metadata - New Privacy Challenges in Decentralised Online Social Networks," in *Proceedings of the IEEE International Conference on Pervasive Computing and Communications Workshops*. IEEE, March 2012, pp. 333–339.

[14] S. Taheri-Boshrooyeh, A. Küpçü, and Ö. Özkasap, "Security and Privacy of Distributed Online Social Networks," in *Proceedings of the IEEE International Conference on Distributed Computing Systems Workshops*. IEEE, June 2015, pp. 112–119.

[15] A. De Salve, P. Mori, and L. Ricci, "A survey on privacy in decentralized online social networks," *Computer Science Review*, vol. 27, pp. 154–176, 2018.

[16] T. Paul, A. Famulari, and T. Strufe, "A survey on decentralized Online Social Networks," *Computer Networks*, vol. 75, pp. 437–452, 2014.

[17] S. Jahid, S. Nilizadeh, P. Mittal, N. Borisov, and A. Kapadia, "DECENT: A Decentralized Architecture for Enforcing Privacy in Online Social Networks," in *Proceedings of the IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*. IEEE, Mar. 2012, pp. 326–332.

[18] S. Nilizadeh, S. Jahid, P. Mittal, N. Borisov, and A. Kapadia, "Cachet: A Decentralized Architecture for Privacy Preserving Social Networking with Caching," in *Proceedings of the International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '12. ACM, 2012, pp. 337–348.

[19] B. Guidi, T. Amft, A. De Salve, K. Graffi, and L. Ricci, "DiDuSoNet: A P2P Architecture for Distributed Dunbar-based Social Networks," *Peer-to-Peer Networking and Applications*, vol. 9, no. 6, pp. 1177–1194, 2016.

[20] A. Shakimov, H. Lim, R. Cáceres, L. P. Cox, K. A. Li, D. Liu, and A. Varshavsky, "Vis-à-Vis: Privacy-preserving Online Social Networking via Virtual Individual Servers," in *Proceedings of the International Conference on Communication Systems and Networks (COMSNETS'11)*. IEEE, 2011, pp. 1–10.

[21] S. Caron, F. Giroire, D. Mazauric, J. Monteiro, and S. Pérennes, "P2P Storage Systems: Study of Different Placement Policies," *Peer-to-Peer Networking and Applications*, vol. 7, no. 4, pp. 427–443, Dec 2014.

[22] S. Fu, L. He, X. Liao, C. Huang, K. Li, C. Chang, and B. Gao, "Cadros: The Cloud-Assisted Data Replication in Decentralized Online Social Networks," in *Proceedings of the IEEE International Conference on Services Computing*, Jun. 2014, pp. 43–50.

[23] G. Mega, A. Montresor, and G. P. Picco, "Efficient dissemination in decentralized social networks," in *Proceedings of the IEEE International Conference on Peer-to-Peer Computing*. IEEE, Aug. 2011.

[24] M. Conti, A. De Salve, B. Guidi, and L. Ricci, "Epidemic Diffusion of Social Updates in Dunbar-Based DOSN," in *Euro-Par 2014: Parallel Processing Workshops*. Cham: Springer, 2014, pp. 311–322.

[25] U. Tandukar and J. Vassileva, "Selective Propagation of Social Data in Decentralized Online Social Network," in *Advances in User Modeling*, L. Ardissono and T. Kuflik, Eds. Springer, 2012, pp. 213–224.

[26] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'01)*, ser. Lecture Notes in Computer Science, vol. 2218. Springer, 2001, pp. 329–350.

[27] P. Druschel and A. Rowstron, "PAST: A large-scale, persistent peer-to-peer storage utility," in *Proceedings of the IEEE Workshop on Hot Topics in Operating Systems*. IEEE, May 2001, pp. 75–80.

[28] R. I. M. Dunbar, "The Social Brain Hypothesis," *Evolutionary Anthropology: Issues, News, and Reviews*, vol. 6, no. 5, 1998.

[29] R. I. M. Dunbar, "The Social Brain Hypothesis and its Implications for Social Evolution," *Annals of Human Biology*, vol. 36, no. 5, 2009.

[30] V. Arnaboldi, A. Guazzini, and A. Passarella, "Egocentric Online Social Networks: Analysis of Key Features and Prediction of Tie Strength in Facebook," *Computer Communications*, vol. 36, no. 10, 2013.

[31] A. Kovacevic, K. Graffi, S. Kaune, C. Leng, and R. Steinmetz, "Towards Benchmarking of Structured Peer-to-Peer Overlays for Network Virtual Environments," in *Proceedings of the IEEE International Conference on Parallel and Distributed Systems*. IEEE, 2008, pp. 799–804.

[32] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi, "On the Evolution of User Interaction in Facebook," in *Proceedings of the ACM Workshop on Online Social Networks*. ACM, 2009, pp. 37–42.

[33] M. Conti, A. De Salve, B. Guidi, F. Pitto, and L. Ricci, "Trusted Dynamic Storage for Dunbar-Based P2P Online Social Networks," in *On the Move to Meaningful Internet Systems: OTM 2014 Conferences*, R. Meersman, H. Panetto, T. Dillon, M. Missikoff, L. Liu, O. Pastor, A. Cuzzocrea, and T. Sellis, Eds. Springer, 2014, pp. 400–417.