Honda Research Institute EU
Honda Research Institute Europe GmbH
https://www.honda-ri.de/

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

Newton Masinde, Moritz Kanzler, Kalman György Graffi

2020

Preprint:

This is an accepted article published in IEEE International Symposium on Networks, Computers and Communications. The final authenticated version is available online at: https://doi.org/10.1109/ISNCC49221.2020.9297202
Copyright 2020 IEEE

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

Newton Masinde, Moritz Kanzler, Kalman Graffi
Department of Computer Science, Heinrich Heine University
Düsseldorf, Germany
Email:*firstname.lastname*@hhu.de

***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

A direct result of increased online social networks communications is the exponential growth in online shared content uploaded to the various platforms. The obvious downside to this 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. DOSNs are designed with the goal of ensuring that the OSN operates in a distributed fashion with minimal or no central control.

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 [1], seen as the problem of *scalable dissemination of social updates* [2]. "Social data" or "social update" refers to all the data exchanged between the users (profile information, relationships, community memberships and so on) and the generated content (comments, messages, posts, images and so on) [1], [2]. In centralized systems, content delivery network (CDN) help reduce the network load. In decentralized applications designing a good social content delivery strategy will result in 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 and dynamic data replication on several points [3]: social network topology shows non-trivial clustering and positive degree correlation, 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 [4], [5]) 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

In a comprehensive studies on DOSNs focusing on data management by Guidi et al. [1], the DOSNs are classified as *DHT-based*, *social overlay (SO) based* and *external resource-based* DOSNs. *DHT-based DOSNs*, such as DECENT [6], Cachet [7], and LibreSocial [4], [5], 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 is used for storing the social content and also offers indexing services. For *social overlay (SO) based DOSNs*, such as DiDuSoNet [8], 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. Finally, external resource-based DOSNs like Diaspora, use a federation of private web servers and thus are a compromise between centralized servers and completely decentralized solutions. Appropriate solutions for reliable decentralized data management in DOSNs are necessary and must be kept in mind. However, it requires handling three challenges associated with are *data availability/persistence*, *information diffusion* and *privacy* [1]. We discuss the first two points critically.

A. Data availability

The persistence of publicly accessible data in a distributed environment is considered. Traditional approaches such as erasure codes [9] and distributed caching [10], are unsuitable for DOSNs, and appropriate solutions for DOSNs are divided into two groups, *external resource-based strategies* and *replica selection strategies* [1]. In external resource-based DOSNs, the data availability is guaranteed via use of external resource management service such as cloud-based storage services, or even hybrid solutions that entail distributed storage augmented with cloud services such as Cadros [11]. In replication data availability is realized via a replica selection strategy.

978-1-7281-5628-6/20/$31.00 ©2020 IEEE

The strategies can be either *replication in the DHT (untrusted nodes)* or *replication on trusted nodes*. In *replication in the DHT (untrusted nodes)*, the underlying DHT overlay handles replication, and the social network has no control of the replica placement. For LibreSocial, 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. In *replication on trusted nodes*, there is no dependence on overlay replication mechanisms. Replica placement is controlled by the social network. Each node selects a suitable nearby replica node based on some calculated measurements. For it to work, two aspects must be accounted for. First, nodes can go offline at any time which can affect persistent storage and is solved by analysis of online behavior of the nodes. Secondly, gauging trust between nodes for data exchange, which is solved by inclusion of encryption and determining social strengths between peers based on factors such as number of common friends or number of actual interactions between each other.

B. Information diffusion

This considers efficient access of needed information taking into account cost factors such as network bandwidth, response times and so on, to ensure data consistency. In DOSNs, the update dissemination process is the focus, with user's data updates availed to all interested nodes [1]. Update dissemination processes falls into three classes, *request-reply*, *active dissemination* and *hybrid* approaches. *Request-reply approaches* assume that, if data availability is guaranteed, available content can be accessed by making a request. The node responsible for the content sends the requested content and if the content's owner is offline, replica nodes will reply. This approach is seen in DOSNs such as DECENT, and LibreSocial. *Active dissemination approaches* is similar to network flooding common in unstructured P2P networks. However update distribution is limited by mechanisms such as gossip protocols like rumor mongering and anti-entropy [12], or measures such as Weighted Ego Betweenness Centrality [13] and social interaction behavior [14], hence only nodes requiring the update receive it. An examples of a DOSN that use this approach is DiDuSoNet. *Hybrid approaches*, implement a combination of request-reply and active dissemination approaches as in Cachet [7].

III. Data management in LibreSocial

LibreSocial (previously LifeSocial.KOM [4], [5]) is a DOSN based on a P2P framework model, hence relies on a DHT for its underlying functionality. LibreSocial is designed using (a heavily modified) FreePastry, an open source implementation of Pastry [15] which includes PAST [16], a persistent storage utility that manages data replication. 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

Fig. 1 shows LibreSocial's data management structure. 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 offers the abilities to update and 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

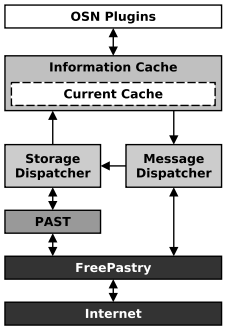


Fig. 1. LibreSocial's architecture

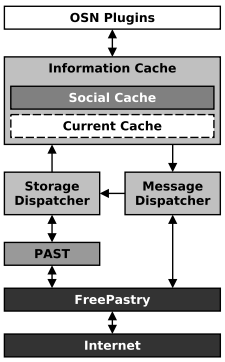


Fig. 2. With social cache extension

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* supplements PAST's internal caching mechanism, which is designed atop the Storage Dispatcher and the Message Dispatcher, forming 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 (cache miss)*: Request is sent to the Storage Dispatcher which performs an overlay lookup. Retrieved content is then stored in the local cache storage by the Information Cache 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 then 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.
- *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 utilizes replication in the DHT for data availability and the request-reply approach for information diffusion. Request-reply approach is efficient for DHT-based DOSNs but can result in high network traffic from recurring overlay lookups (for same content). Also, for Information Cache, regular overlay lookups can occur if the freshness time interval is too short. To tackle this, we propose a social cache for proactive update dissemination of (relevant) cache data after updating. We review two DOSNs, Cachet and DiDuSoNet for insight into a possible social caching solution.

B. Cachet vs DiDuSoNet

Cachet [7] performs replication in the DHT and uses both reply-request and active dissemination approaches for information diffusion. Its social cache uses 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, the social contact selection algorithm tends to select all online users in a continuous fashion rather than dynamically selecting a few contacts. By using

information collected on node interactions by analyzing the social graph structure formed by the social connections this problem can be alleviated but this is not possible in DHT-based DOSNs such as Cachet due to the network topology.

DiDuSoNet [8], on the other hand, 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. It uses an epidemic protocol proposed in [13] for spreading social updates with the links between nodes based on the social interactions. It uses an egocentric social measure to approximate Betweenness Centrality called the *Weighted Ego Betweenness Centrality (WEBC)* exploiting a weighted graph, weights referring to the tie strengths between the nodes. Nodes with higher interactions have a higher value of WEBC. Unlike Cachet, DiDuSoNet does not include an advanced caching mechanism. It describes a very effective trusted nodes selection mechanisms that makes use of Dunbar-numbers [17] to build an ego network. For each user (*ego*) in the network, 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*. Using social interaction analysis based on the epidemic protocol in [13] 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 is 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. 2 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 consider handling of social updates, 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. So cached data consistency is not guaranteed but the maximum age of the data is limited. Using *social updates*, we can actively disseminate changed content to selected users who have previously requested it (as predicted), hence already available locally. Social updating 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. 3. This guarantees that users subscribed to upcoming social updates of a particular user's instance channel receive content updates.

2) Receiving & storing update: Each social caching instance handles its own subscriptions to other users' instance channels. This is shown in Fig. 4. 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 6. 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. 5.

C. Selecting suitable users based on social interaction

The next major step is the selection strategy for subscriptions. The *selection strategy* filters and selects a set of suitable users that an instance then subscribes to, for social updates. Suitable subscriptions should ideally be frequently accessed content. To achieve a high ratio of cache hits for the social caching mechanism is to ensure that data accessed 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. A further restriction is 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, shown in Fig. 7, from which suitable subscriptions are to be made. 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 selection strategies for these n candidates, namely *random*, *trend*, and *social score* selection strategy which are further described.

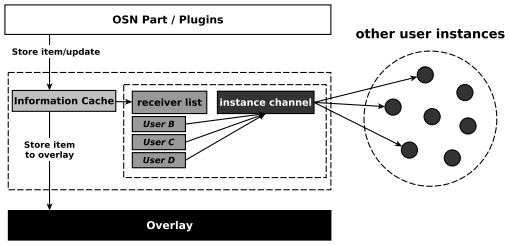


Fig. 3. Distribution of social updates

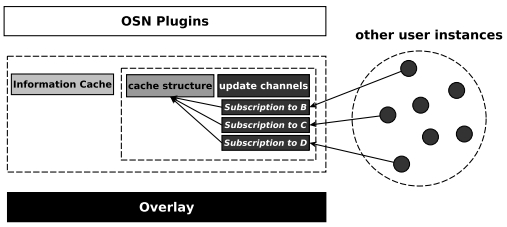


Fig. 4. Receiving updates from subscribed channels

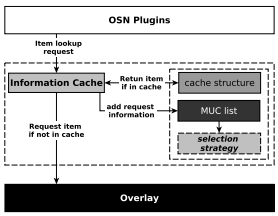


Fig. 5. Lookup request using the social cache

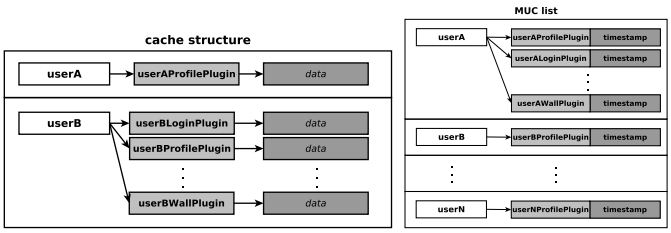


Fig. 6. Social cache with two nested abstraction levels

Fig. 7. MUC list structure

1) Random selection strategy: In this strategy, presented in Algorithm 1, 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.

Algorithm 1: 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) Trend selection strategy: 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. Algorithm 2 is a summary of the strategy.

3) Social score selection strategy: It follows the idea of using a *SocialScore* based on Dunbar's [17] social interaction model, as used for DiDuSoNet [8]. 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

Algorithm 2: 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 ← ∅
```

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 [8] 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 [18]. 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. 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.

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

Combining the *TieStrength* and *MIL* results in a relationship between the number of interactions against interactions durations for every monitored user. Hence, the $SocialScore_{LS}$ ranks a user with more frequent interactions over a longer period of time higher. This mechanism ensures consistent cache entries, takes into account the social component of an OSN, dynamically adopts to changes of users preferences, and is works well without undermining the existing processes of the given application. Algorithm 3 summarizes the strategy.

Nevertheless, there are some special cases in which the social caching mechanism has conceptual weaknesses or limits. These are considered next, and the solutions discussed.

Algorithm 3: 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
```

D. Addressing constraints in social caching

With the combination of the proposed concepts of social updates, tracking of interactions by the MUC list, and selection of subscriptions using a selection strategy, a caching mechanism which uses active dissemination of newly updated information to a fixed set of users is now possible. However, some challenges arise as the social caching mechanism exhibits 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 hence overlay requests. When a user does not have his/her friends' content in the social cache, but needs access to the content often, recurrent overlay requests occur. 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 stored information but reduce the needed overlay requests by a significantly.

2) Social bootstrapping: If a current subscription makes few content changes, no social updates come from them. Therefore requests for this user will always result in overlay request. This is addressed using *social bootstrapping*. When a subscription occurs, the user receiving the request sends a message to the subscriber to inform him/her to receive social updates henceforth. The user subscribed to then responds with all of its own cached content that is added to the caching structure of the subscribing user. Cache consistency is not affected by this process because a change in the initially sent content leads to a social update that overwrites the initial content.

3) Limitation of active subscription and Dunbar-number: As the social cache is an additional mechanism in the DHT-based overlay, the number of connections between users must to be restricted to prevent establishment of a second overlay. This limiting value is the *Dunbar number*, 150, which defines the maximum contacts stored in the MUC list. Once reached the least ranked user based on the selection strategy is removed.

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 [19]. 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 can be made about the general characteristics to describe the overall quality of the system. To analyze the social caching mechanism, the quality properties of concern are 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. The metric used to quantify performance is the *cache hit ratio*, shown in Equation 4.

$$Cache\ hit\ ratio = \frac{Cache\ replies}{Total\ replies} \tag{4}$$

2) Efficiency: This is 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*.

a) Local efficiency: With this, we desire to find out how certain resources at the local node. We monitor the number of objects cached 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.

b) 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.

B. Test scenario

The tests focus on the number of interactions and the time intervals between interactions. For a realistic scenario, three ego-based Facebook datasets, Facebook'09 [20], SNAP'14 [21], and Facebook'14 [8], having practical data on interactions and interaction intervals were considered and compared as shown in Table I. The Facebook'09 data was chosen, due to availability and for data revealing interactions between friends and providing the timestamped interactions as summarized in Table II. A downsampling formula, given by Equation 5 was applied to reduce the analysis from 870 days to the desired time to retain usage characteristics gathered from the data set. 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.

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

The experimental parameters and conditions are shown in Table III. 64 LibreSocial instances were run on 8 machines having similar hardware setup and operating system (Debian Linux). There are two test scenarios that are considered, each focusing on a different aspect.

a) Comparing selection strategies: As three cache selection strategies have been proposed. we desire to find the most suitable strategy based on its quality aspects. Downsampling was done to 6 hours for the tests. For all tests only the social cache is enabled, but not the current caching mechanism.

b) Comparing caching approaches: In these tests, downsampling was done 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*.

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

VI. RESULTS AND DISCUSSION

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. 8. 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.

a) Social cache size: This is shown in Fig. 8a. The social score selection strategy shows the smallest number of stored cache data objects, then 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.

b) Number of subscription messages: Fig. 8b depicts the subscriptions during the experiments. The social score strategy shows least number of subscriptions, meaning 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.

c) Average size of MUC list: This is depicted in Fig. 8c. The social score and random strategies have almost similar values throughout. The low MUC list size is indicative of minimal use of local resources used by the social cache, thus better *local efficiency*. In comparison, the MUC list size 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.

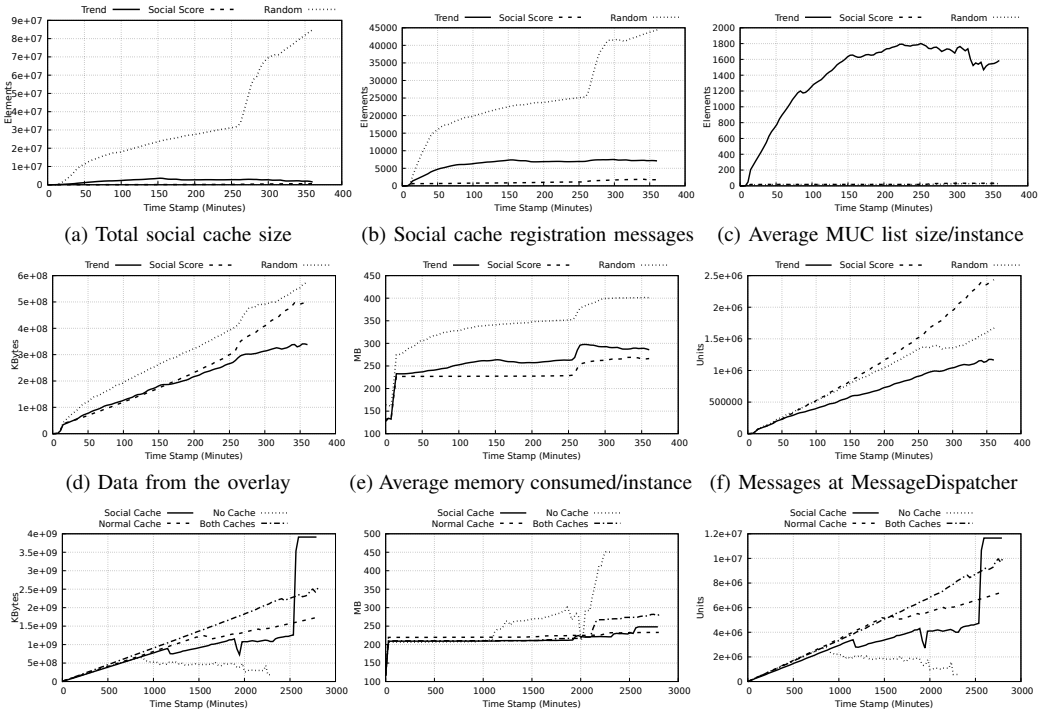
3) Overlay efficiency: We analyze the overall efficiency of the selection strategies by considering the following overlay aspects.

a) Overlay data load generated: : Fig. 8d shows the overall amount of data objects sent in the Pastry overlay for each selection strategy. Social score and trend 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 strategy generates a higher amount of traffic due to overlay requests.

b) Average memory consumed per instance: : The results presented in Fig. 8e 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 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.

c) 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. 8f 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.

Summary: Thus in view of the results gathered in analyzing the performance, local and overall efficiency, it can be inferred that, the social score strategy is the preferred choice over trend and random selection strategies. It attains a very high performance of 92.4% cache hits against a drastically lower costs at the local node and has the lowest costs in terms of data transfer and used memory, but produces more messages over the MessageDispatcher. From the overall efficiency, the trend selection strategy shows a similar pattern as the social score strategy albeit with slightly higher values for the



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

data transfer and memory consumption but with the least messages observed a the MessageDispatcher. The random selection strategy, has 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 8g-8i, it is important to note that the tests without any cache solution generally failed quarter way during the experiment due to internal errors generated in Pastry. However the results are shown for comparison to show their influence on the data storage costs. We now discuss the results obtained from the cache comparison tests.

1) Performance: Again, the cache hit ratio is used to get the effectiveness of the cache mechanism when requests are made. 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%, 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, based on the number of stored objects 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. Objects stored in the current cache alone as well as in the social cache solution alone are much fewer than when the two caches combined. The current cache alone stores less than 50% of the items in the social cache alone while initiating nearly twice as many cache responses per items (17 and 9 respectively). This indicates more efficient use of local resource (fewer items) while achieving a higher cache hit rationby the current cache, but with twice as many lookups. The combination of the caches is accompanied by a drastic increase in the cached items (almost five times as much as the current caching only). There is also an equally significant drop in the number of cache responses per item (thrice as much as current cache only), with much fewer items stored in the current cache as in the social cache (as expected). This is explained by the hierarchy and arrangement of the caches, and indicative also of 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 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.

a) Data transferred: Fig. 8g shows the amount of data transferred and monitored in the Pastry layer of LibreSocial. With the current caching only more overlay data transfer occurs than with the social caching only, probably due to the need for freshness validity time in cache. The higher data transfer for the combined caching solution indicates higher bandwidth utilization as both caches are operating simultaneously. Hence additional communication payload and needed overlay lookup requests.

b) Average memory consumed per instance: This is illustrated in Fig. 8h. Separate from each other, the current cache and social cache use similar amount of memory. In combination, there is a similar trend until the second group of friend requests are made at about the 2200th minute of the experiment. This disparity may need some further investigation. Thereafter, the memory usage increases slightly in comparison to the other solutions.

c) Messages at MessageDispatcher: The results are shown in Fig. 8i. 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.

Summary: The evaluation of *overlay efficiency* shows lower resource consumption by the social cache compared to the current caching approach. However, the combined approach results in the highest overlay load. Combining both efficiency qualities with the examined performances results, the 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] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.