Secure Distributed Data Structures for Peer-to-Peer-based Social Networks

Jens Janiuk, Alexander Mäcker
Department of Computer Science
University of Paderborn, Germany
{jjaniuk,amaecker}@mail.upb.de

Kalman Graffi
Department of Computer Science
University of Düsseldorf, Germany
graffi@cs.uni-duesseldorf.de

***Abstract*—Online social networks are attracting billions of nowadays, both on a global scale as well as in social enterprise networks. Using distributed hash tables and peer-to-peer technology allows online social networks to be operated securely and efficiently only by using the resources of the user devices, thus alleviating censorship or data misuse by a single network operator. In this paper, we address the challenges that arise in implementing reliably and conveniently to use distributed data structures, such as lists or sets, in such a distributed hash-table-based online social network. We present a secure, distributed list data structure that manages the list entries in several buckets in the distributed hash table. The list entries are authenticated, integrity is maintained and access control for single users and also groups is integrated. The approach for secure distributed lists is also applied for prefix trees and sets, and implemented and evaluated in a peer-to-peer framework for social networks. Evaluation shows that the distributed data structure is convenient and efficient to use and that the requirements on security hold.**

***Keywords*—Distributed social networks; peer-to-peer networks; distributed data management; network security**

I. Introduction

Online social networks (OSNs) are popular nowadays due to the ease of connecting billions of users and allowing them to interact through a set of communication options. Facebook, as most prominent example, connects around 1 billion users worldwide. Besides global networks such as Facebook, local networks, e.g. solely restricted to an enterprise, also arise as a tool to organize knowledge sharing in companies and academia. A limitation of the current centralized approaches is given by the single operator running the social networking site. This operator is able to censor content and opinions, read private and confidential messages, market user data or be shut off in oppressive countries that want to reduce communication on specific topics, like during the Arab spring. Although the majority of the users remain unaware of the risks of using centralized OSNs and ignore the possibility of the communication being overheard, for some users in the world it is crucial or even vital to be able to communicate and organize with friends in a secure, confidential and anonymous way.

Peer-to-Peer (p2p)-based networks [1] can be made indestructible, as no component is considered crucial. Structured p2p overlays, especially distributed hash tables (DHTs) like Chord [2] or Pastry [3] offer a key-based routing [4] interface that allows to implement an efficient and fully retrievable storage. Replication approaches like PAST [5] keep data in the DHT persistent, even when the initially hosting node fails.

Distributed data structures (DDS) such as lists, trees or sets are very popular constructs in the OSNs. Sets are used to organize memberships in groups, to manage friends and to add entries to an interest set. Lists are used to maintain and order a set of entries, e.g. in mailboxes, statuses and comments, or albums. Trees maintain dynamically entries in categories, e.g. allocating users in classes such as relationship status, age or sex. They allow to easily search for profiles or categorized entries. These DDS are essential for the creation of a p2p-based OSN, as they relieve programmers from the load to re-implement the same logic for every single small application using only simple put/get operations from the DHT.

The characteristic of DDS is that the entries have besides the content payload also a pointer to the next element(s), thus creating a data graph. Individual entries in the data structure should be editable only by the corresponding authors. The security of the pointer structure needs to be separated from the access control of the entries. One main challenge in p2p networks is how to store, maintain and secure such distributed linked lists in the presence of differentiated access rights of the entries as well as unreliable and untrusted storage nodes.

In this paper, we present our approach to host DDS on DHTs, fulfilling the following requirements:

- Basic functionality: the data structure should be creatable by any user. A data structure consists of a collection frame and individual entries. Access and read rights on the structure are defined by the data structure originator.
- New entries: new entries should be addable by the users who the data structure initiator gave permission to.
- Write control: entries should be modifiable and deletable only by the initial entry author.
- Read control: the author should be able to define a set of read-enabled users, only they and the authors should be able to read the content.
- Replicability: all entries are assumed to be replicable, i.e. no constraints must be made on the content-hosting node.
- Load balancing and traffic optimization: distributed data storage should not overload hosting nodes. Redundant communication should be omitted.

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre “On-The-Fly Computing” (SFB 901)

In this paper, we address the challenges of how to support the need of p2p-based OSNs for secure DDS. The need for such an approach is emphasized in the discussion of related work in Sec. II. We present the organization of the distributed list that is implemented on top of Pastry and PAST in Sec. III. The list is split every k entry and spread over the DHT. The security elements and access control approach is introduced in Sec. IV. For that, we first present our assumptions based on our work in [6], and elaborate in Subsec. IV-A how to enable users to encrypt content for specific users and how to authenticate and sign content. Through the remote operations presented in Subsec. IV-B, sending only entry read, insert or modify commands, load on the peers is further decreased. We present in Subsec. IV-C how the keys required for the access control are shared. While lists can grow naturally, the deletion of entries leaves free spaces, which requires secure consolidation. Our protocol for consolidation is presented in Subsec. IV-E. For the evaluation in Sec. V, we implemented the secure DDS in our p2p framework for social networks [7] and tested the effects of the list split parameter as well as the consolidation of deleted entries in the list. Evaluation shows that the secure DDS, presented in the example of the list, fulfills both the requirements for ease of use and load balancing, as well as the security and access control. We conclude in Sec. VI that the presented secure DDS are an essential building block to implement the vision of a p2p-based OSN. Its abilities go far beyond the simple put/get possibilities of a DHT and offer convenient, load-balanced and secure data structures in OSNs.

II. Related Work

Distributed social networks propose to alleviate the security and censorship risk of centralized OSN sites. An overview of the challenges is given in [8] and [9]. While friend-of-a-friend approaches, such as [10] or [11], assume that users set up web servers and create a distributed OSN in this way, the risk of data misuse and censorship is still given, as any web server may be compromised or shut down. P2P networks inherently assume failures of participating nodes and provide a reliable distributed data storage plane with DHTs, such as Pastry [3].

P2P-based online social networks came into focus a few years ago. In [12] the challenges are summarized. Here, two main classes emerged: overlay networks that do and that do not resemble the social structures. Having overlay networks mapping the friendship relation, such as in [13], Safebook [14] or GoDisco [12], offers the benefit that communication remains mainly local in the overlay. On the other hand, social overlay structures face limitations in the case of bootstrapping, nodes with few friends or nodes not trusting their friends. Recent analysis in [15] shows that the availability of friend-hosted data is at risk when assuming regular friendship behavior.

The second class of p2p-based OSNs, such as LifeSocial [16]–[18], Peerson [19] or OverSoc [20], create a DHT that is used as a distributed data plane independent of the friendship relations. DHTs offer a basic put and get functionality of data. For LifeSocial, a practical access control approach has been presented in [6], which introduces a root of trust, deploys a secure key infrastructure and allows to cryptographically provide access control for single data elements. Quality in routing can be provided by [21].

However, operations on single elements in p2p OSNs are cumbersome and limit the application scope. In OSNs there are a lot of functionalities where associated elements or collections of elements have to be organized. This is the case for entries on a wall, photo albums in a gallery or forums, for example. Common data structures like lists (e.g. for friends, wall entries, private message queues), sets (e.g. for groups and chat rooms), as well as sorted trees (e.g. for range queries on profile information, such as age), are more practical for the OSN application. The p2p data plane should support such data structures securely, capacity-aware and access controlled.

Capacity-awareness in handling the storage of large lists of content in the DHT can be either addressed by dedicated strong nodes, e.g. as in [22], or on the other extreme by having each element stored on one node and linking to the further elements of the data structure. The first approach can cause problems if the elements are big. Possibly a single peer has not enough storage space to store the whole item, e.g. albums with large images. On the other hand, one lookup is sufficient to retrieve all elements belonging to the collection. To alleviate the load issue, a dedicated function to find strong peers is needed. The second approach has the advantage that the elements and thus the load are distributed among many peers. On the other hand, if elements are rather small and many elements belonging to such a collection should be retrieved, a lot of lookups have to be issued. In this paper, we split the data structure in chunks of configurable size, thus the individual node load is manageble and the number of queries for individual entries is not high.

For individual data items, several security approaches supporting access control lists have been proposed. The work of Palomar et al. [23], [24] propose an approach in which confidentiality and integrity is implemented through encryption and certificates. Merani et al. extend the idea for streaming overlays in [25] and add onion encryption to support various access layers. Group-based approaches as presented in [26], [27] and [28] decide on the validity of data modification by using either byzantine agreement groups or a trusted group head. In both cases, it is assumed that a group is initially formed and that the participants joined willingly. This is not the case in a p2p-based OSN use case, where access groups, similar to Google+ circles, are created dynamically at single nodes without the consent of the chosen nodes. Trusted group heads, such as in [28] or [29] also cannot be assumed in this use case, as we assume that users in a social network cannot agree on a trusted entity to manage the access to their data. In [30] the authors assume that nodes might get compromised and keys get stolen. Therefore they spread key parts to a single encrypted object across several nodes. The failure and malicious behavior of a single node must not effect the security of the data of other users. Previous work focuses on the

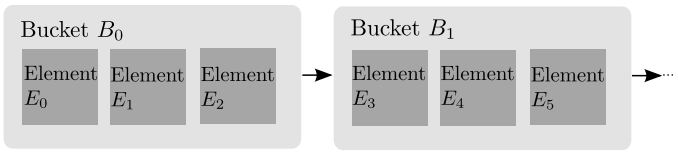


Figure 1. Example of a (bucket) list with splitsize = 3

security and access control on single data elements. DDS, such as lists or sets, define different challenges, presented next.

III. Distributed Data Structures

DDS are considered to be individual entries logically pointing to a set of further elements. This allows to create lists, sets or trees. As the principle for any of these structures is the same, we explain it throughout the paper in the example of the list. In this section, we present how the storage of such DDS can be organized in DHTs and how load-balancing and traffic optimization can be introduced.

A. Storage and Data Organization

In order to host arbitrarily long DDS both load-balanced and efficiently, we propose a tradeoff between hosting all entries on one node and hosting every entry on one individual node. We split the data structure in smaller parts, so-called buckets (full of entries), with each containing $splitsize$ many elements. Each of these buckets is stored in one item in the DHT. For an example refer to Fig. 1. The advantage of this solution is the flexibility, as $splitsize$ can be adapted to the (average) size of the elements to be stored. On the one hand we can reduce the number of lookups for the entire list of size n to $\frac{n}{splitsize}$, and on the other hand the elements are still distributed among different peers. Retrieving several elements corresponds to downloading from different peers, thus accelerating the transfer in comparison to downloading from one peer.

B. Accessing Buckets through Remote Operations

DHTs offer the function `retrieve(`key`)`, which returns the item identified by key, and `store(`key`,` $item$`)`, which inserts an $item$ identified by its key into the DHT. For distributed lists, we implement a function that accesses a certain index of the list like `get(`i`)` by retrieving the corresponding bucket, or a function like `set(`i`)` by retrieving and afterward storing the corresponding bucket. The key of a bucket containing the i^{th} element can be easily computed based on the $listName$ since it holds $key = h(listName + \lfloor \frac{i}{splitsize} \rfloor)$ for $h(\cdot)$ being the hash function used for the DHT. However, in most of the list operations we can save a lot of communication costs if we do not retrieve and store whole buckets but instead simply issue a remote operation, which contains the request (and possibly some data), to the node that stores the affected bucket. Such a remote operation message can be routed by using a basic `lookup(`key`)`. The responsible node can then locally access the bucket and perform the corresponding operation. In the case of the `get(`i`)` request the responsible node will only return the requested element, thus saving communication bandwidth. Another advantage is the possibility to perform several list operations like the contains($item$) in parallel, making this operation fast without receiving a single bucket. In this example, the request is sent to all nodes storing a bucket of the list, which just have to return a $true$ or $false$. The results can be evaluated at the requesting node without much effort. Table I shows which list operations we implemented according to their request parallelity method.

TABLE I LIST OPERATIONS RELATED TO REQUEST METHOD

Operation	Request Method
`add(`$item$`)`	single message
`addAll(`$items$`)`	parallel messages
`get(`i`)`	single message
`contains(`$item$`)`	parallel messages
`set(`i`)`	single message
`indexOf(`$item$`)`	parallel messages
`lastIndexOf(`$item$`)`	parallel messages
`remove(`i`)`	single message
`remove(`$item$`)`	parallel messages
`removeAll(`$items$`)`	parallel messages
`isEmpty()`	single message
`size()`	single message

IV. Security and Access Control for Distributed Data Structures

In order to bootstrap the security of our approach, we make assumptions on what is provided by the p2p overlay. We presented the reasoning and evaluation of these assumptions in [6]. The assumptions and the presented approach is implemented in our p2p framework for social networks [7], [18].

We assume a DHT providing the key-based routing (KBR) interface [4] and replication. In our approach, we used FreePastry [31], implementing Pastry [3] with the replication strategy PAST [5]. In order to create a basis for security and access control, we need to enable the nodes to authenticate each other and send encrypted, signed and authenticated data to each other, and to store messages or data in the network. We modified the original Pastry algorithm to embed a root of trust. We assume that the identifiers of the users, the $userIDs$, are public keys. The user name and the user's password are hashed and used to create a private key, which is used to derive a public key. These public keys are 160 bit long elliptic curves and used as $nodeID$ (equal to the $userID$) in the overlay. By this, every user can already communicate with any other node in an authenticated, confidential and integer manner, by signing and encrypting directed communication.

On top, we assume a simple data access control for individual data elements. Users should be able to define a set of other users they know to read access their data. Data integrity and authentication are provided by signatures using the public keys of the content originators. To enable (read) access control, a new data element is encrypted with a new symmetric key and this symmetric key is encrypted with the public keys of all privileged users that the author selects. The list of encrypted symmetric keys is attached to the encrypted payload and the whole combination is signed and stored in the network. This secured data bundle can be replicated and shared with any node, but only the privileged users can decrypt and access the content. Having these assumptions on the DHT and

replication, the p2p-based key infrastructure and the simple access control for individual data elements, we can already implement initial security means.

A. Read and Write Access Control

The goal for access control is to support the read/write rights on various users on a DDS, i.e. the entries in a collection. The entry authors and the collection author might be different. The basic idea for read and write access in DDS is to use cryptographic methods, i.e. encryption and signing of elements. Based on assumptions and the public/private key pair of each user, further keys can be exchanged among users and, for example, common keys for a group of users can be disseminated. Thus, basically all the elements of the data structure can be encrypted using a key which is known to everybody who should be able to read the data in order to implement a group-based read access control. Here, it is useful to encrypt the single elements instead of entire buckets and to use a symmetric key. Thereby, operations like `contains(`$item$`)` can still be performed by remote operations even if the storing node does not know the symmetric key. For that, the request simply has to contain the encrypted item. In Sec. IV-C we discuss how the keys can be shared in access rights groups.

The second point to be addressed is the write protection. The basic idea here is to store the public key e of the owner's key pair (e, d) along with the content and a signature of the content and the public key in the DHT. If the content should be (over)written, the new content must be signed with the private key d. To avoid replay attacks we additionally store a random nonce together with the public key. The writing user then has to sign the new content concatenated with the random nonce of the content to be overwritten. On insertion of the new content the storing node can verify the signature of the new content and possibly deny the write access in case the signature is invalid. In terms of a DDS, the basic idea is to use these signatures and write protection on two levels: on the one hand each bucket has an owner and must be signed when it is written using the owner's asymmetric key pair (e, d). On the other hand each element of a bucket has an owner, too. Usually the owner of buckets and the owner of different elements are not the same. Basically, the public key of buckets determines who is allowed to add elements to the list (which often corresponds to a group of users), whereas the public key of a single element determines who may change or delete this element (which often corresponds to a single user). An example might be a wall of a user: every friend is allowed to add posts on the wall (the key of buckets is known to all friends) but only the author of a wall entry may change or delete it (the personal private key is used for signatures). By signing content we also can ensure the authenticity and integrity of elements stored in the DHT. An example for a list bucket is shown in Fig. 2.

B. Remote Operations in Secure Lists

Our distributed list uses remote operations to reduce the amount of traffic in the network, and on the other hand, we use signatures in order to provide secure access to the list.

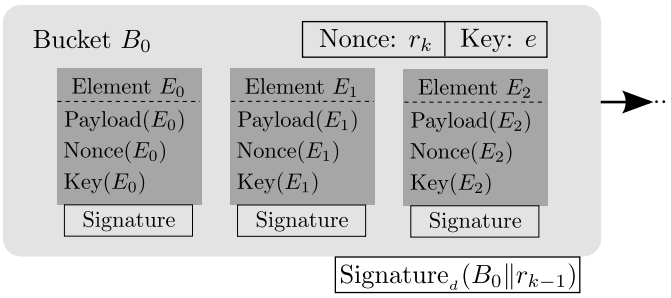


Figure 2. Example of a secured bucket

Unfortunately, a problem arises if both concepts should be combined and used at the same time: if a remote operation requires write access, e.g. `set(`i`)`, it is often not possible to send only a change request to the node holding the affected bucket because this node is usually not allowed to write the item, i.e. it does not have the required key to sign the modified bucket. Therefore, the node which issues the remote operation computes the signature and sends it to the storing node together with the change request. To do so, this node must first retrieve the bucket to be changed, compute the changes locally and finally compute the new signature. Afterwards, the signature and the change request are sent to the storing node. Note that we still reduce the network traffic since the bucket has to be transferred only once; it is not necessary to transfer the whole bucket through the network to store it in the DHT, instead only the new signature and the change request is sufficient. The storing node can now compute the same changes locally and use the given signature to store the whole bucket. Furthermore, in many scenarios the node that wants to write a bucket has retrieved this one before and thus, there is no additional communication for retrieving the bucket in order to compute the signature of the changed version. Examples for such scenarios are adding a post to a wall or adding a photo to an album. In both cases, we usually retrieve the (newest entries of the) wall or the (newest photos of the) album before adding a new element. Note that in order to fully exploit the advantage of remote operations in terms of traffic, it is useful to also adapt the replication to this technique instead of sending the whole items for updating replicas.

Our detailed protocol that manages write access (of a node C to an element stored at S) based on remote operations in distributed lists then consists of the following basic steps:

1) C requests the current version of the file that should be changed from S. If C already has a version of the file, it also sends a hash of it to S.
2) S compares the received hash with the hash of its current version of the file. If the hashes differ, S replies with the current version, otherwise it acknowledges briefly.
3) C performs its changes locally and afterwards computes the signature of the modified file and the nonce of the file that should be overwritten. Afterwards, it sends a request containing the desired changes, the signature and the hash of the file on which C locally performed the changes to S.
4) S checks whether its current version and the version on

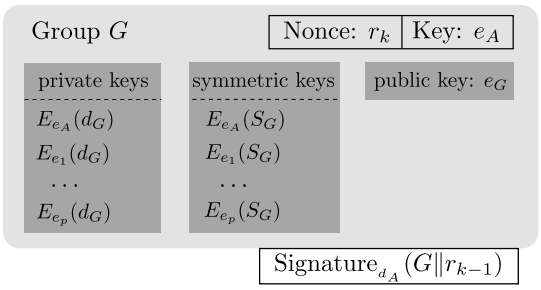


Figure 3. Basic structure of a group

which C has performed its changes are the same by comparing the hashes. If they are not equal, step 2 is applied. Otherwise, S performs the desired changes and then inserts the file into the network using the signature received from C.

Note that in step 4) it is necessary to consider the case in which the hashes received from C and the hash of the file stored at S differ. In this case, C has performed its changes on a version of the file that differs from the current version stored at S. This implies that C has performed its changes on a version which is not up-to-date and thus, the changes have to be computed again in order to be able to overwrite the file. This protocol also ensures the integrity and authenticity of buckets. In order to resort elements inside a bucket it is necessary to replace existing items. According to the protocol, this can be done only by the owner of the item.

C. Key Management and Distribution

For the assignment of access rights it is necessary that all users that are supposed to have access to a list must share a common key. Here, we address the question of how we can establish and share such a key. Although in the following we take the list as an example for our mechanism, the ideas are applicable to any kind of items stored in the DHT.

Assume that a user U_A wants to share his list with the users $U_1, \ldots, U_p$, i.e. all these users should be able to read the list entries and add new elements to the list (for the sake of simplicity we assume that the set of users who can read entries and those who can add elements are the same). In this case, it is required that all users share a common symmetric key for the encryption of elements (read access) and a asymmetric key pair for signing and verifying list buckets (write access). The basic idea of our solution is as follows: at first U_A creates a new symmetric key S_G and asymmetric keys (e_G, d_G). Afterwards S_G and d_G are encrypted with the public keys $e_1, \ldots, e_p$ of the designated group members $U_1, \ldots, U_p$. These encrypted keys $E_{e_1}(S_G), \ldots, E_{e_p}(S_G)$, $E_{e_1}(d_G), \ldots, E_{e_p}(d_G)$ and the public key e_G are then encapsulated in an object, which we call *group*. After creation, this group is stored in the DHT so that each member of the group can retrieve the group and decrypt the keys by using the own private key. Furthermore, everybody has access to the group's public key and thus, can verify signatures created based on the group's private key. In order to ensure that only U_A (the admin of the group) can change the group he created and nobody else can remove or add new members to it, the group is signed with U_A's private key and the write protection as described in the previous subsection is applied. Whenever the admin removes a member from the group, it is necessary to update the group keys so that the removed user no longer has access to the new keys. For that, the admin creates new group keys and stores them encrypted for each member exactly in the same way as is done during the first creation. Despite the new keys, the old ones are still kept in the group in order to allow the members access to files that have been created before the removal. Note that to re-encrypt, existing content is needed only if the removed member must not have access to data he was able to access (and save) before. In the case that no re-encryption of the content is performed, the old keys are still available and thus, members of the group can still read corresponding information. One has to ensure that a removed user can no longer write content to the list that is secured with the group's asymmetric key pair. For that, it is only necessary that whenever a bucket is written, the storing node verifies the signature using the most up-to-date public key of the group.

In order to reduce traffic, the same remote operation protocol as described for the list can also be applied to groups. Thereby, for example, users who want to retrieve a group key need not retrieve the whole group item from the network but just send a corresponding request to the storing node.

D. Hierarchical Access Groups

The mechanism of groups, as described so far, serves as a basic concept and can be extended in several ways. In the following we describe how we can allow several users to be admins and how we can define hierarchical, i.e. nested, groups. As described above, a group that is stored is signed with the admin's key in order to prevent unauthorized changes. If we now want to allow several admins to manage a group, it is necessary that all these admins have a common key which is used to sign the group. To do so, we introduce the idea of an *admin group*. An admin group is a usual group with the difference that when storing such an admin group, it is signed with the admin group's private key. Thus, all members of the admin group may change the group and the admin group can only be modified by an admin.

The second enhancement regards the definition of hierarchical groups. In OSNs, different groups are often related to each other in such a way that, for instance, a group G_1 is a subgroup of another group G_2, i.e. the members of G_1 inherit the access rights of G_2. In order to enable the definition of such a hierarchy, we can simply allow members of a group to be other groups. If, for example, all members of G_1 are supposed to be members of G_2, we can add the group G_1 as a member of G_2. Employing this approach we can define arbitrary complex (hierarchical) structures of groups.

If in a complex structure a user wants to access the key of a group G (e.g. to decrypt an item encrypted with G's key) and he is not a (direct) member of G, we can perform a depth-first search on the hierarchy above G in order to find

a group the user is a member of. Using this key, we can go backwards along the chosen path through the hierarchy and decrypt the required group keys until we finally can retrieve G's key. The previously described approach for admin groups is also applicable for hierarchical groups, since a hierarchy can also be reflected in the corresponding admin groups. Another enhancement of groups might be the requirement for one user to be a member of several different groups. In this case, the keys of the group can be simply encrypted several times according to the required groups memberships.

E. Element Deletion and Consolidation

A further major issue that has to be addressed is how we can implement the deletion of list elements under the constraint of access control. For that, we assume a deletion to be nothing else than overwriting an existing element with a dedicated artificial `deletedItem`. Using such an element is necessary to prove to the storing node that the acting node is allowed to perform the operation. If an element is simply deleted from the bucket, then the insertion of the modified bucket will fail. This is the case because the storing node has to verify the signatures of the elements in the bucket. If an element could be missing without providing any signature, everybody (who has the owner key of the bucket) would be able to delete elements, which must not happen.

In order to be still able to provide operations concerning elements at certain positions like `get(`i`)`, we have to be aware of the positions at which `deletedItems` are located. To handle this situation, we store each position at which a `deletedItem` is located in the metadata of the list. This metadata also contains the size and the $splitsize$ of the list and can be stored, for example, under the key $h(listName)$. Thus, it is still possible to access an element at a certain position i by simply calculating the i^{th} element of the list based on the knowledge about `deletedItems`. For example, if we want to access the i^{th} element and we know that there are j `deletedItems` in front of it, we simply have to access element $i + j$. The metadata of the list also enables us to get the size of the whole list by one request without the need to receive each bucket of the list. In terms of write protection, the metadata file can be handled like a bucket.

Unfortunately, this approach causes another problem which we have to overcome. If each element that is deleted is replaced by a `deletedItem`, the size of the list can never decrease. Therefore, the length of the list is equal to the number of all elements that once have been inserted into it. This can lead to situations where the actual (information-carrying) list elements are distributed among much more peers than we aimed at according to the $splitsize$. In the worst case we can even get a list where each actual element is stored in another bucket and thus, the advantages of grouping elements in buckets completely disappear.

To address this problem we developed a protocol, termed *consolidation*, that can close gaps in a list: if an element at position i is removed, then afterwards all elements at positions $j > i$ have to be shifted one position to the left. This would cause the `deletedItem` to be shifted to the right until it reaches the end of the list and is removed.

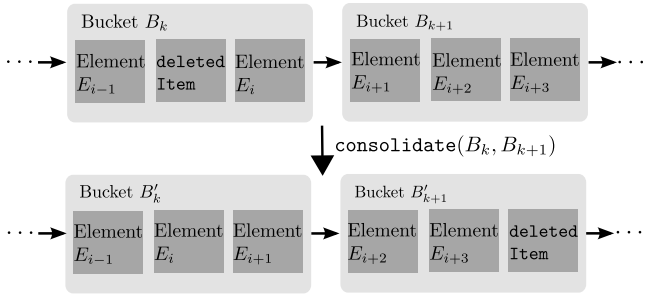


Figure 4. Consolidation of two buckets

At first, note that a shift cannot be performed by arbitrary nodes but only by nodes which have write access to the list as the buckets must be signed appropriately. One naive idea might be that the user removing an element consolidates the entire list afterwards. In fact, this can put a very high burden on a single node since this node has to compute the whole list after shifting the elements, it has to compute at least $\frac{n}{splitsize}$ many signatures and contact a corresponding number of storing nodes. Furthermore, a huge amount of communication would be necessary between the storing nodes and the initiator of the consolidation in order to avoid inconsistent states that could occur, for instance, if a bucket cannot be stored successfully. Therefore, we propose a more reasonable protocol.

The basic idea of our consolidate protocol is to shift a `deletedItem` one bucket to the right for each time the protocol is applied. Thereby, a `deletedItem` is shifted on occasion until it eventually reaches the end of the list and is removed. The advantage of this protocol is that the initiating node only has to compute the signature of two buckets and there are only two storing nodes that have to communicate in order to coordinate their actions and guarantee consistency. For the following description of the protocol, refer to Fig. 4.

Assume that bucket B_k contains a `deletedItem` and `consolidate`(B_k, B_{k+1}) is called. These two buckets are consolidated so that afterwards the `deletedItem` is located at the end of bucket B_{k+1} and the remaining elements E_j with $i \leq j \leq E_{i+3}$ are shifted one position to the left. Let B'_k and B'_{k+1} denote bucket B_k and B_{k+1} after the consolidation. Our protocol to perform this consolidation is as follows:

1) The node initiating the consolidation computes the buckets B'_k and B'_{k+1} and its corresponding signatures $Sig(B'_k)$ and $Sig(B'_{k+1})$. Both signatures are parallely sent to nodes S_k and S_{k+1} which are storing B_k and B_{k+1}, respectively.
2) S_k and S_{k+1} compute B'_k and B'_{k+1} themselves and afterwards verify the corresponding signatures received beforehand. If a node encounters an invalid signature, it cancels its consolidation process.
3) If the signatures are valid, S_{k+1} notifies S_k.
4) If S_k receives the notification of S_{k+1}, it stores B'_k in the network. Afterwards it notifies S_{k+1}.

TABLE II DISTRIBUTION OF LIST OPERATIONS IN THE SETUP

Operation	Distribution A	Distribution B
`get(`i`)`	0.5	0.93
`add(`$item$`)`	0.375	0.053
`remove(`i`)`	0.125	0.017

5) If S_{k+1} receives the notification of S_k, it stores B'_{k+1} after checking that B'_k is stored in the network as expected.

Note that because in step 4) and 5) the corresponding node waits for a notification of the other node and such a notification will possibly never arrive (e.g. the other node went offline), the waiting time is bounded by a variable. If the notification does not arrive within this time span, the consolidation process is canceled. The main idea of the communication between S_k and S_{k+1} is to avoid the loss of elements. At first, S_k has to wait for the notification of S_{k+1} to ensure that S_{k+1} already has done its verification which is based on B_k. Furthermore, because S_{k+1} should only store the new bucket B'_{k+1} (which does not contain element E_{i+1} anymore) if E_{i+1} is now contained in bucket B'_k, it waits for the notification of S_k and afterwards also checks if bucket B'_k is stored in the network. Thereby we can ensure that a list element cannot get lost. The only not desired situation that can happen is that bucket B'_k is stored in the DHT but B'_{k+1} is not (e.g. S_{k+1} leaves the system directly after step 4.), which is detectable by having two identical entries with the same nonce. In addition, this protocol prevents an unauthorized resorting of elements inside the list or any other unauthorized operation because the storing nodes compute the new buckets locally according to the defined protocol. If an attacker tries to perform an illegal operation, the signatures of the new buckets will not match and the storing nodes will cancel the consolidation.

The protocol as described above provides a secure solution for consolidation which is very flexible and can be called by any node which has write access to the list. Furthermore, there are no requirements on the points in time when it is called. A concrete application that makes use of the list can freely decide about when to apply the protocol. For instance, there might be applications where it only makes sense to consolidate the list if there is a high percentage of `deletedItems`, as the elements are rather big and the advantage of a consolidation does not outweigh the effort of moving the elements. On the other hand, there might be scenarios where a consolidation does not involve much traffic but reducing the number of lookups to retrieve elements favors the performance.

The strategy we implemented in our experiments is to call the protocol whenever a node performs a read or write access to the list. In this case, a bucket which contains a `deletedItem` is chosen at random to perform the consolidation. This randomness ensures that each affected bucket is consolidated over time. Other selection rules for consolidation, like always choosing the first affected bucket, might lead to insufficient consolidation. This can happen, for example, in a situation where we alternately have a removal and a consolidation in the first bucket for several times. In this case, the `deletedItem` is shifted to the second bucket each time and thus, many `deletedItems` are gathered in the buckets directly behind the first one, but none can reach the list's end.

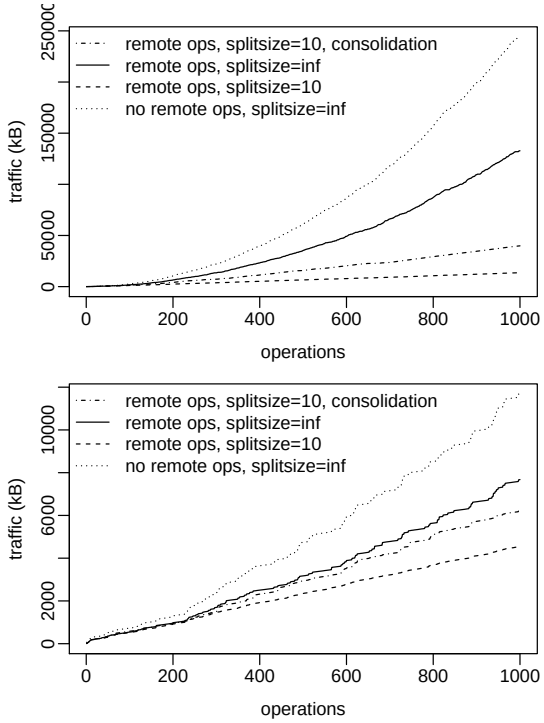


Figure 5. Traffic due to list operations. Top: A; bottom: B

V. EVALUATION

The evaluation of our approach for secure DDS can be done qualitatively on the access control and security and quantitatively on the performance and costs for maintaining the DDS. Throughout the paper we motivated in a qualitative evaluation how the presented approach and the assumptions fulfill the requirements on security and access control. In this section, we present a quantitative evaluation on the DDS, especially with focus on the effects of the split size and the remote operations.

For that, we implemented all presented mechanisms in or on top of FreePastry and performed list operations in an emulated network. We focus on three metrics in order to assess the quality of our protocols: the traffic in the network caused by operations on our list, the number of messages exchanged and the ratio of the number of `deletedItems` in the list, and the overall number of items in the list during the experiments. The last metric coincides with the fraction of buckets that we could save if the list was consolidated, i.e. without `deletedItems`, all the time. We abstract from the actual number of nodes in the network and consider each bucket of the DDS as hosted by an individual node. Thus, the worst case behavior is measured where shortcuts between buckets on the same node are not given.

In order to get meaningful results we consider two different settings: in setting A we have performed 1000 randomly chosen operations according to the distribution in Table II, and in setting B the same amount of operations according to

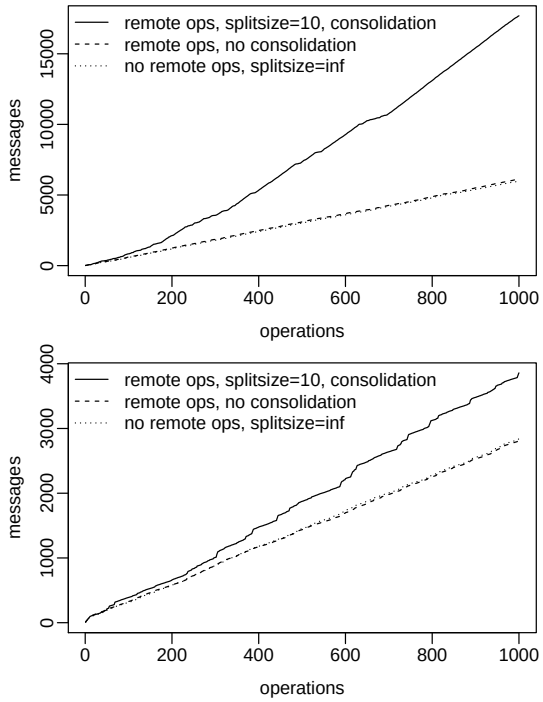


Figure 6. Messages exchanged in list operations. Top: A; bottom: B

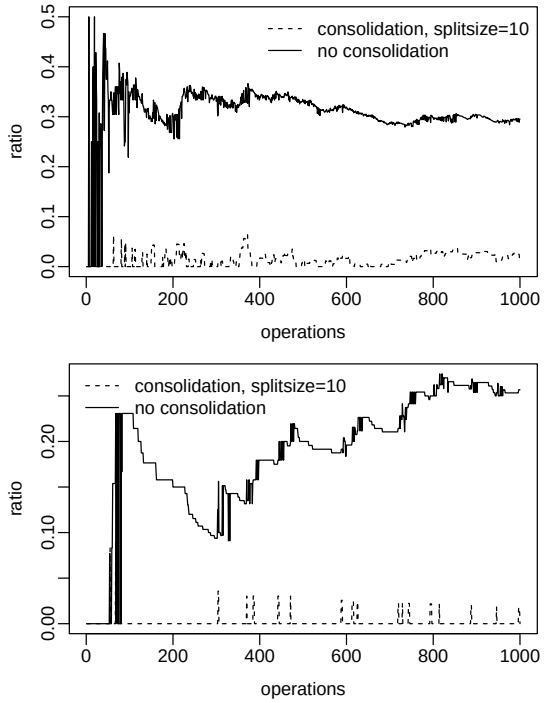


Figure 7. Ratio of `deletedItems` over time. Top: A; bottom: B

a distribution based on an analysis of realistic user behavior in OSNs in terms of read and write access as described in [32]. In both settings, all list elements that are inserted, and thus all elements that are accessed later on, are of size 1 kB. The main difference between setting A and B is the number of write accesses. While in setting B just 7% of all access requests are remove or add operations, in setting A 50% are of this kind. Thus, in setting A the number of elements as well as the number of `deletedItems` will basically be higher than in setting B. In the graphs, the results of setting A are always presented on top, those of setting B on the bottom.

In our first experiments we are interested in the basic gain of using remote operations and of splitting the list into buckets. In Fig. 5 it is shown that in both test settings a non-distributed list without remote operations clearly causes the highest amount of traffic since for each operation we have to retrieve the entire list from the network. The usage of remote operations can almost halve the traffic and both approaches are definitely beaten by the combination of remote operations and the use of buckets. If we also take the number of messages exchanged for the applied list operations into account, one can observe that this number does not differ for remote or non-remote operations (cf. Fig. 6).

Also note that the number of messages is independent of $splitsize$ since for all operations in our setting the affected bucket is known in advance. However, in both metrics considered so far an increase in the traffic is caused by the consolidation protocol. In fact, this is not a surprise since we apply this protocol after each access and thus, an additional exchange of messages and buckets becomes necessary as described in Subsec. IV-E. At any rate, this slight increase might be acceptable because of the considerable improvement regarding the `deletedItems` in the list, as shown in Fig. 7. In both settings, the fraction of `deletedItems` can be decreased from values up to 0.3 to fractions mainly not exceeding 0.05 while increasing the traffic a little in comparison to a list that does not employ the consolidation protocol.

So far we have seen the actual advantage of using remote operations and buckets. However, we stuck to a $splitsize = 10$ and the suspicion obviously is that a higher $splitsize$ leads to a higher traffic, since larger buckets have to be transferred. For that reason, we also want to consider our metrics depending on the chosen $splitsize$ of the list. In Fig. 8 and Fig. 9, the traffic caused by list operations is shown against the $splitsize$ for a list with and without the consolidation protocol, respectively. In general, one can observe that a larger $splitsize$ usually leads to a higher traffic. An explanation for this might be the fact that whenever new elements should be added to the list (or two buckets are consolidated), larger buckets have to be transferred in order to compute the required signatures.

On the other hand, larger buckets have another major advantage. As elements are distributed among less buckets than for a small bucket size, it is easier to consolidate such a list. This is true as a given list is shorter in terms of the number of buckets if the $splitsize$ is larger. Thus, fewer consolidation steps are necessary. This results in a lower number of messages as shown in Fig. 10 and a drastic reduction of the number of `deletedItems` in the DDS over time when using consolidation as shown in Fig. 11.

The size of buckets directly influences our three metrics under consideration. The choice always serves as a tradeoff

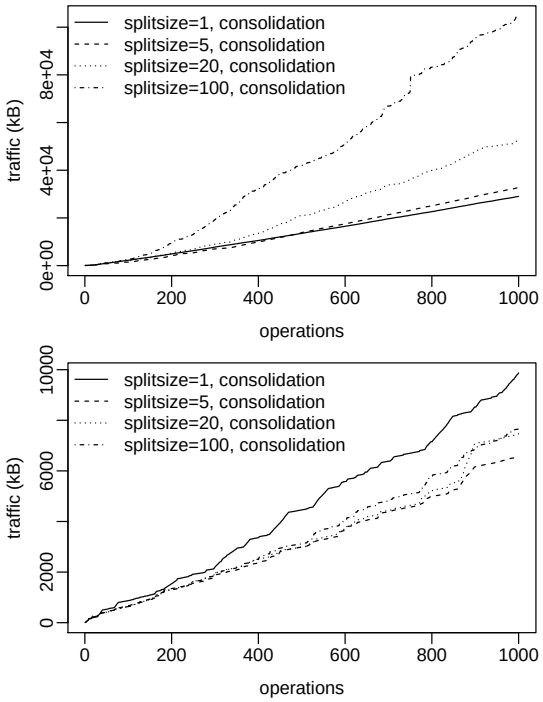


Figure 8. Traffic: list operations and consolidation. Top: A; bottom: B

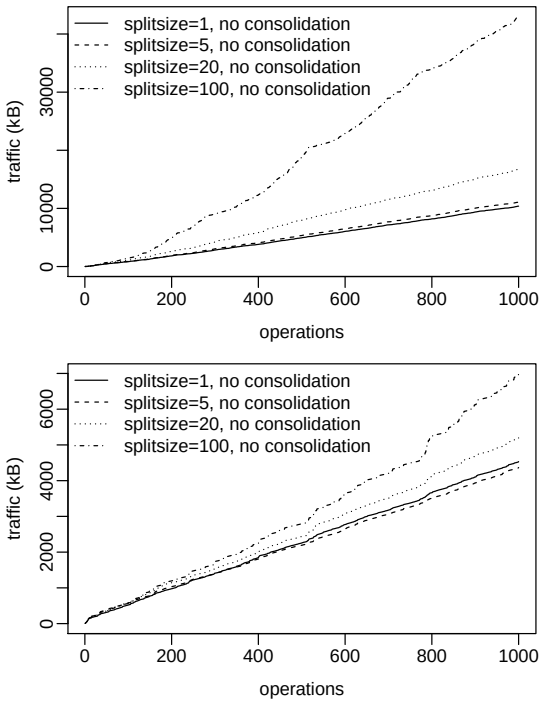


Figure 9. Traffic: list operations, no consolidation. Top: A; bottom: B

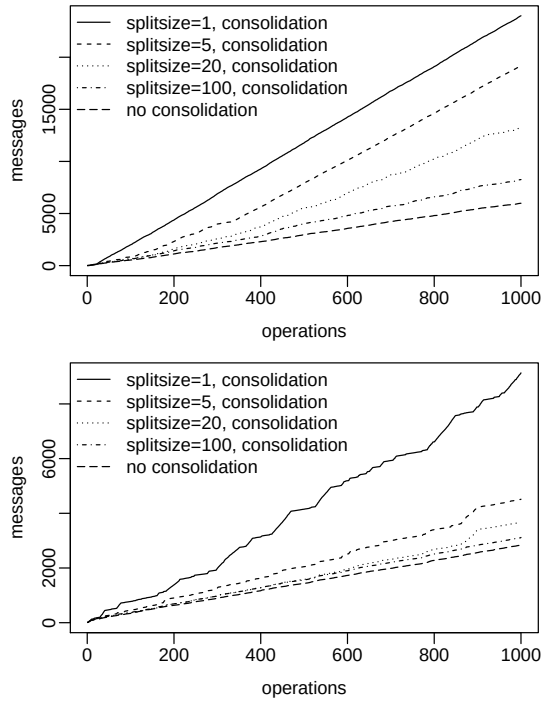


Figure 10. Messages exchanged due to list operations. Top: A; bottom: B

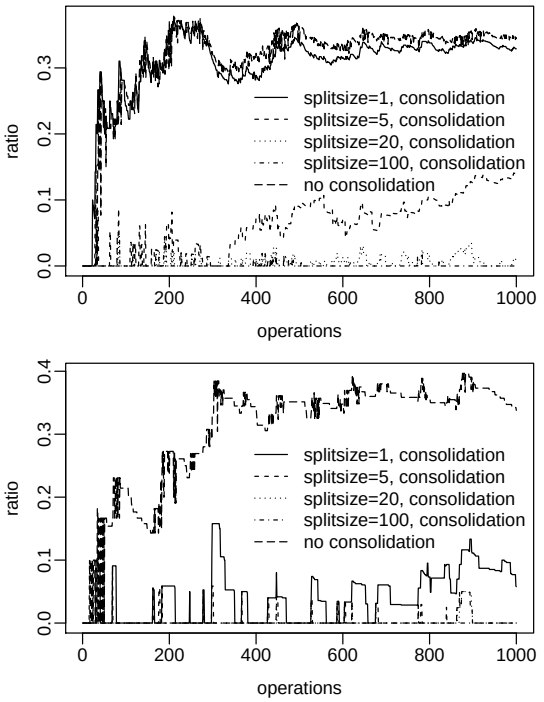


Figure 11. Fraction of `deletedItems` over time. Top: A; bottom: B

between traffic, messages and how well our consolidation protocol works. For our setting, for instance, a $splitsize$ of 20 might be a good choice. On the one hand the traffic is not much higher than for smaller values (particularly in contrast to a $splitsize$ of 100), on the other hand the number of messages is reduced and especially the consolidation works considerably better than for smaller buckets.

To conclude the evaluation, we observe that the DDS are working properly, distribute the load, and support security as well as access control for single users, various editors of a DDS and group-based access control. Through the buckets, as well as the the remote operations and the consolidation protocol, the data structure is maintained across several nodes efficiently.

VI. Conclusions

In this paper, we addressed the issue of how to maintain and secure convenient data structures, such as lists, sets or trees, in p2p systems. These data structures are very prominent and thus interesting for novel p2p applications such as p2p-based online social networks. We presented solutions for two main requirements. First, the distributed and efficient organization of the potentially large data structures is discussed. We motivate and evaluate a modular hosting of the data structures in the DHT in so-called buckets combined with remote operations. The data structure supports among read and write operations also delete, consist and index operations (see Table I). Evaluation shows that by modifying the split size of the data structure buckets, a reasonable tradeoff of the load on the nodes and the traffic in the network can be found. Using our consolidation protocol, deleted items are able to leave the data structure in a lazy, but efficient fashion.

Second, the security and access control of these data structures is discussed and evaluated qualitatively. We show how to support access rights on data structures through cryptographic means, where data structure owners and entry owners are different. Both owners might be also groups. For the group management we present an approach to organize the users, share the required cryptographic keys and allow the embedding of groups in other groups. The protocols for the secure distributed data structures presented in this paper fulfill the requirements on security and efficiency and are a valuable tool to build advanced, storage-based p2p applications, like p2p-based online social networks.

References

[1] N. Liebau, K. Pussep, K. Graffi, S. Kaune, E. Jahn, A. Beyer, and R. Steinmetz, "The Impact Of The P2P Paradigm," in *AMCIS '07: Proc. of Americas Conf. on Information Systems*, 2007.

[2] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," in *SIGCOMM '01: Proc. of the Int. Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*. ACM, 2001.

[3] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems," in *IFIP/ACM Middleware '01: Proc. of the Int. Conf. on Distributed Systems Platforms*, 2001.

[4] F. Dabek, B. Y. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, "Towards a Common API for Structured Peer-to-Peer Overlays," in *IPTPS '03: Proc. of the Int. Workshop on Peer-to-Peer Systems*, ser. LNCS, vol. 2735. Springer, 2003.

[5] P. Druschel and A. I. T. Rowstron, "PAST: A Large-scale, Persistent Peer-to-Peer Storage Utility," in *IEEE HotOS '01: Proc. of the Workshop on Hot Topics in Operating Systems*, 2001.

[6] K. Graffi, P. Mukherjee, B. Menges, D. Hartung, A. Kovacevic, and R. Steinmetz, "Practical Security in P2P-based Social Networks," in *IEEE LCN '09: Proc. of Int. Conf. on Local Computer Networks*, 2009.

[7] K. Graffi, "A Peer-to-Peer Framework for Social Networks," http://www.p2pframework.com.

[8] S. Buchegger and A. Datta, "A Case for P2P Infrastructure for Social Networks - Opportunities and Challenges," in *IEEE WONS'09: Proc. of the Int. Conf. on Wireless On-demand Network Systems and Services*, 2009.

[9] T. Paul, B. Greschbach, S. Buchegger, and T. Strufe, "Exploring Decentralization Dimensions of Social Networking Services: Adversaries and Availability," in *ACM HotSocial'12: Proc. of the Int. Workshop on Hot Topics on Interdisciplinary Social Networks Research*, 2012.

[10] J. Golbeck and M. Rothstein, "Linking Social Networks on the Web with FOAF: A Semantic Web Case Study," in *AAAI '08: Proc. of the AAAI Conf. on Artificial Intelligence*, 2008.

[11] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin, "Persona: An Online Social Network with User-defined Privacy," in *ACM SIGCOMM'09*, 2009.

[12] A. Datta and R. Sharma, "GoDisco: Selective Gossip Based Dissemination of Information in Social Community Based Overlays," in *IEEE ICDCN'11: Proc. of the Int. Conf. on Distributed Computing and Networking*, 2011.

[13] G. Mega, A. Montresor, and G. P. Picco, "Efficient Dissemination in Decentralized Social Networks," in *IEEE P2P'11: Proc. of Int. Conf. on Peer-to-Peer Computing*, 2011.

[14] L. A. Cutillo, R. Molva, and T. Strufe, "Safebook: Feasibility of Transitive Cooperation for Privacy on a Decentralized Social Network," in *IEEE WOWMOM '09: Proc. of the Int. Symposium on a World of Wireless, Mobile and Multimedia Networks*, 2009.

[15] G. Mega, A. Montresor, and G. P. Picco, "On churn and communication delays in social overlays," in *IEEE P2P'12: Proc. of Int. Conf. on Peer-to-Peer Computing*, 2012.

[16] K. Graffi, S. Podrajanski, P. Mukherjee, A. Kovacevic, and R. Steinmetz, "A Distributed Platform for Multimedia Communities," in *IEEE ISM '08: Proc. of the Int. Symp. on Multimedia*, 2008.

[17] K. Graffi, C. Groß, P. Mukherjee, A. Kovacevic, and R. Steinmetz, "LifeSocial.KOM: A P2P-based Platform for Secure Online Social Networks," in *IEEE P2P '10: Proc. of the Int. Conf. on Peer-to-Peer Computing*, 2010.

[18] 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 *IEEE CCNC '11: Proc. of the IEEE Consumer Communications and Networking Conf.*, 2011.

[19] S. Buchegger, D. Schiöberg, L. H. Vu, and A. Datta, "PeerSoN: P2P Social Networking - Early Experiences and Insights," in *ACM SNS ' 09: Proc. of the ACM Workshop on Social Network Systems Social Network Systems*, 2009.

[20] D. I. Wolinsky, P. S. Juste, P. O. Boykin, and R. J. O. Figueiredo, "OverSoc: Social Profile Based Overlays," in *IEEE WETICE'10: Proc. of Int. Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises*, 2010.

[21] K. Graffi, K. Pussep, S. Kaune, A. Kovacevic, N. Liebau, and R. Steinmetz, "Overlay Bandwidth Management: Scheduling and Active Queue Management of Overlay Flows," in *IEEE LCN '07: Proc. of the Annual Conf. on Local Computer Networks*, 2007.

[22] P. Wette and K. Graffi, "Adding Capacity-Aware Storage Indirection to Homogeneous Distributed Hash Tables," in *NetSys'13: Proc. of the Networked Systems Annual Technical Conf.*, 2013.

[23] E. Palomar, J. M. Estévez-Tapiador, J. C. H. Castro, and A. Ribagorda, "Certificate-based Access Control in Pure P2P Networks," in *IEEE P2P'06: Proc. of the Int. Conf. on Peer-to-Peer Computing*, 2006.

[24] E. Palomar, J. M. Estévez-Tapiador, J. C. H. Castro, and A. Ribagorda, "Secure Content Access and Replication in Pure P2P Networks," *Computer Communications*, vol. 31, no. 2, Feb. 2008.

[25] M. L. Merani, D. Saladino, and M. Capetta, "Taking Advantage of Social Network Relationships in P2P Streaming Overlays," in *IEEE GLOBECOM'11: Proc. of the Global Communications Conf.*, 2011.

[26] Y. Zhang, X. Li, J. Huai, and Y. Liu, "Access Control in Peer-to-Peer Collaborative Systems," in *IEEE ICDCS'05: Proc. of the Int. Conf. on Distributed Computing Systems Workshops*, 2005.

[27] W. Xiaoming and C. Fan, "A Group-oriented Access Control Scheme for P2P Networks," *MECS Wireless and Microwace Techn.*, no. 1, 2011.

[28] V. Pathak and L. Iftode, "Byzantine Fault Tolerant Public Key Authentication in Peer-to-Peer Systems," *Computer Networks*, vol. 50, no. 4, pp. 579–596, 2006.

[29] X. Wang, Y. Liu, and H. Yang, "An Anonymous Authentication Scheme Based on Fully Homomorphic Encryption in P2P Networks," *Information Technology*, vol. 11, no. 5, 2012.

[30] A. Bonifati, R. Liu, and H. Wang, "Distributed and Secure Access Control in P2P Databases," in *Data and Applications Security and Privacy XXIV*, ser. LNCS. Springer, 2010, vol. 6166, pp. 113–129.

[31] Rice University, Houston, USA, "FreePastry," http://www.freepastry.org/FreePastry/.

[32] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida, "Characterizing User Behavior in Online Social Networks," in *ACM SIGCOMM IMC'09: Proc. of the Int. Conf. on Internet Measurement*, 2009.